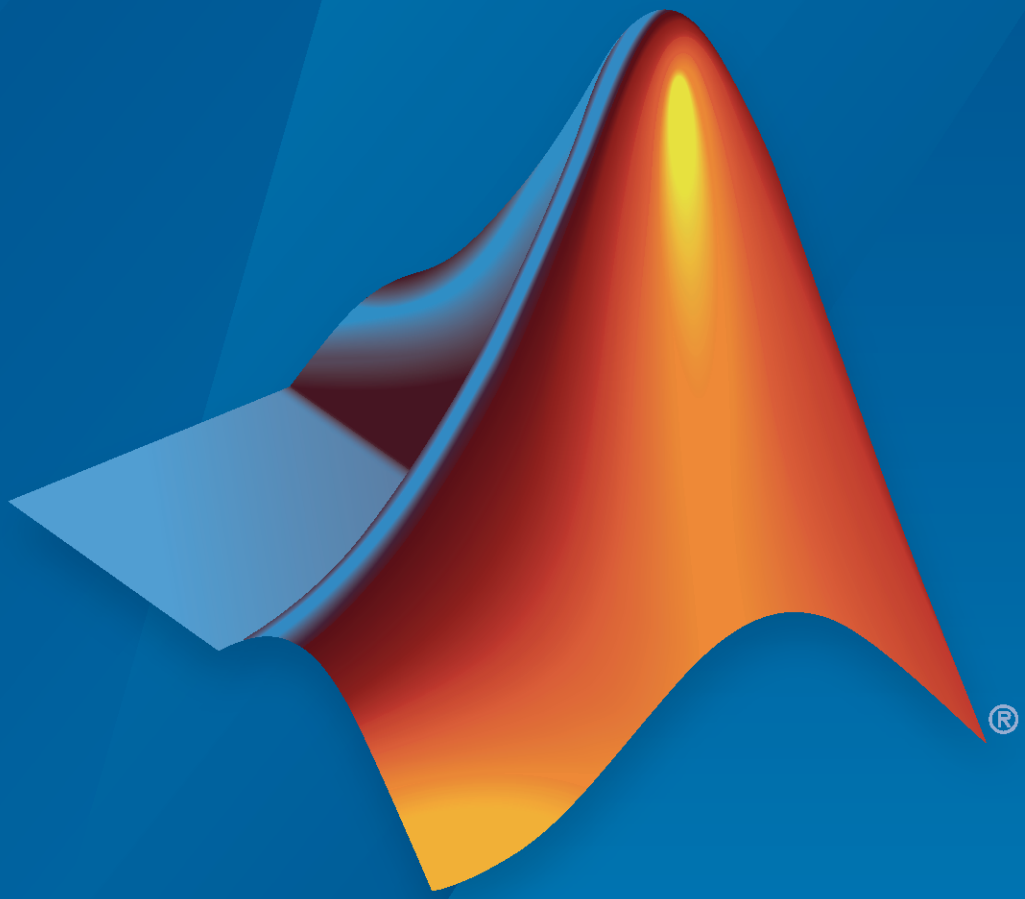
Deep Learning Toolbox™

User's Guide

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth



MATLAB®

R2021b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Deep Learning Toolbox™ User's Guide

Revision History

June 1992	First printing	
April 1993	Second printing	
January 1997	Third printing	
July 1997	Fourth printing	
January 1998	Fifth printing	Revised for Version 3 (Release 11)
September 2000	Sixth printing	Revised for Version 4 (Release 12)
June 2001	Seventh printing	Minor revisions (Release 12.1)
July 2002	Online only	Minor revisions (Release 13)
January 2003	Online only	Minor revisions (Release 13SP1)
June 2004	Online only	Revised for Version 4.0.3 (Release 14)
October 2004	Online only	Revised for Version 4.0.4 (Release 14SP1)
October 2004	Eighth printing	Revised for Version 4.0.4
March 2005	Online only	Revised for Version 4.0.5 (Release 14SP2)
March 2006	Online only	Revised for Version 5.0 (Release 2006a)
September 2006	Ninth printing	Minor revisions (Release 2006b)
March 2007	Online only	Minor revisions (Release 2007a)
September 2007	Online only	Revised for Version 5.1 (Release 2007b)
March 2008	Online only	Revised for Version 6.0 (Release 2008a)
October 2008	Online only	Revised for Version 6.0.1 (Release 2008b)
March 2009	Online only	Revised for Version 6.0.2 (Release 2009a)
September 2009	Online only	Revised for Version 6.0.3 (Release 2009b)
March 2010	Online only	Revised for Version 6.0.4 (Release 2010a)
September 2010	Online only	Revised for Version 7.0 (Release 2010b)
April 2011	Online only	Revised for Version 7.0.1 (Release 2011a)
September 2011	Online only	Revised for Version 7.0.2 (Release 2011b)
March 2012	Online only	Revised for Version 7.0.3 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.0.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.1 (Release 2013b)
March 2014	Online only	Revised for Version 8.2 (Release 2014a)
October 2014	Online only	Revised for Version 8.2.1 (Release 2014b)
March 2015	Online only	Revised for Version 8.3 (Release 2015a)
September 2015	Online only	Revised for Version 8.4 (Release 2015b)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 10.0 (Release 2017a)
September 2017	Online only	Revised for Version 11.0 (Release 2017b)
March 2018	Online only	Revised for Version 11.1 (Release 2018a)
September 2018	Online only	Revised for Version 12.0 (Release 2018b)
March 2019	Online only	Revised for Version 12.1 (Release 2019a)
September 2019	Online only	Revised for Version 13 (Release 2019b)
March 2020	Online only	Revised for Version 14 (Release 2020a)
September 2020	Online only	Revised for Version 14.1 (Release 2020b)
March 2021	Online only	Revised for Version 14.2 (Release 2021a)
September 2021	Online only	Revised for Version 14.3 (Release 2021b)

Contents

1 Deep Networks

2 Deep Network Designer

3 Deep Learning with Images

4 Deep Learning with Time Series, Sequences, and Text

5 Deep Learning Tuning and Visualization

6 Manage Deep Learning Experiments

7 Deep Learning in Parallel and the Cloud

8 Computer Vision Examples

9 Image Processing Examples

10 Automated Driving Examples

11 Lidar Examples

12 Signal Processing Examples

13 Wireless Comm Examples

14 Audio Examples

15 Reinforcement Learning Examples

16 Predictive Maintenance Examples

17 Computational Finance Examples

18 Import, Export, and Customization

19 Deep Learning Data Preprocessing

20 Deep Learning Code Generation

Neural Network Design Book

21 Neural Network Objects, Data, and Training Styles

22 Multilayer Shallow Neural Networks and Backpropagation Training

23 Dynamic Neural Networks

24 Control Systems

25 Radial Basis Neural Networks

26 Self-Organizing and Learning Vector Quantization Networks

27 Adaptive Filters and Adaptive Training

28 Advanced Topics

29 Historical Neural Networks

30 Neural Network Object Reference

31 Function Approximation, Clustering, and Control Examples

Code Notes

C

Deep Networks

Deep Learning in MATLAB

In this section...

What Is Deep Learning?

Deep learning is a branch of machine learning that teaches computers to do what comes naturally to humans: learn from experience. Machine learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as a model. Deep learning is especially suited for image recognition, which is important for solving problems such as facial recognition, motion detection, and many advanced driver assistance technologies such as autonomous driving, lane detection, pedestrian detection, and autonomous parking.

Deep Learning Toolbox provides simple MATLAB commands for creating and interconnecting the layers of a deep neural network. Examples and pretrained networks make it easy to use MATLAB for deep learning, even without knowledge of advanced computer vision algorithms or neural networks.

For a free hands-on introduction to practical deep learning methods, see Deep Learning Onramp.

What Do You Want to Do?	Learn More
Perform transfer learning to fine-tune a network with your data	“Start Deep Learning Faster Using Transfer Learning” on page 1-5 **Tip** Fine-tuning a pretrained network to learn a new task is typically much faster and easier than training a new network.
Classify images with pretrained networks	“Pretrained Deep Neural Networks” on page 1-8
Create a new deep neural network for classification or regression	“Create Simple Deep Learning Network for Classification” on page 3-47 “Train Convolutional Neural Network for Regression” on page 3-53
Resize, rotate, or preprocess images for training or prediction	“Preprocess Images for Deep Learning” on page 19-16
Label your image data automatically based on folder names, or interactively using an app	“Train Network for Image Classification” Image Labeler (Computer Vision Toolbox)

What Do You Want to Do?	Learn More
Create deep learning networks for sequence and time series data.	"Sequence Classification Using Deep Learning" on page 4-2 "Time Series Forecasting Using Deep Learning" on page 4-15
Classify each pixel of an image (for example, road, car, pedestrian)	"Getting Started with Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
Detect and recognize objects in images	"Recognition, Object Detection, and Semantic Segmentation" (Computer Vision Toolbox)
Classify text data	"Classify Text Data Using Deep Learning" on page 4-89
Classify audio data for speech recognition	"Speech Command Recognition Using Deep Learning" on page 4-23
Visualize what features networks have learned	"Deep Dream Images Using GoogLeNet" on page 5-15 "Visualize Activations of a Convolutional Neural Network" on page 5-141
Train on CPU, GPU, multiple GPUs, in parallel on your desktop or on clusters in the cloud, and work with data sets too large to fit in memory	"Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud" on page 7-2 "Deep Learning with Big Data" on page 7-17

To learn more about deep learning application areas, including automated driving, see "Deep Learning Applications".

To choose whether to use a pretrained network or create a new deep network, consider the scenarios in this table.

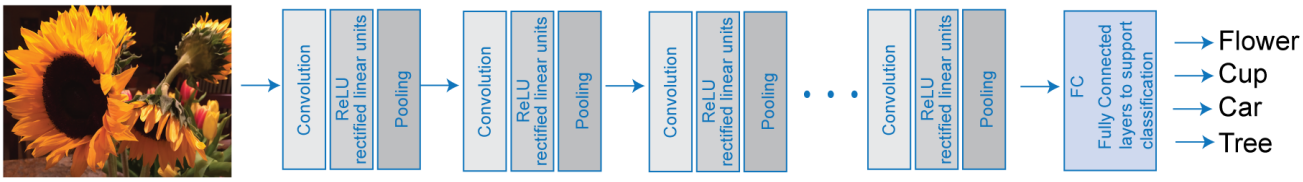
	Use a Pretrained Network for Transfer Learning	Create a New Deep Network
Training Data	Hundreds to thousands of labeled images (small)	Thousands to millions of labeled images
Computation	Moderate computation (GPU optional)	Compute intensive (requires GPU for speed)
Training Time	Seconds to minutes	Days to weeks for real problems
Model Accuracy	Good, depends on the pretrained model	High, but can overfit to small data sets

For more information, see "Choose Network Architecture" on page 1-67.

Deep learning uses neural networks to learn useful representations of features directly from data. Neural networks combine multiple nonlinear processing layers, using simple elements operating in parallel and inspired by biological nervous systems. Deep learning models can achieve state-of-the-art accuracy in object classification, sometimes exceeding human-level performance.

You train models using a large set of labeled data and neural network architectures that contain many layers, usually including some convolutional layers. Training these models is computationally

intensive and you can usually accelerate training by using a high performance GPU. This diagram shows how convolutional neural networks combine layers that automatically learn features from many images to classify new images.



Many deep learning applications use image files, and sometimes millions of image files. To access many image files for deep learning efficiently, MATLAB provides the `imageDatastore` function. Use this function to:

- Automatically read batches of images for faster processing in machine learning and computer vision applications
- Import data from image collections that are too large to fit in memory
- Label your image data automatically based on folder names

Try Deep Learning in 10 Lines of MATLAB Code

This example shows how to use deep learning to identify objects on a live webcam using only 10 lines of MATLAB code. Try the example to see how simple it is to get started with deep learning in MATLAB.

1. Run these commands to get the downloads if needed, connect to the webcam, and get a pretrained neural network.

   ```
   camera = webcam; % Connect to the camera
   net = alexnet;   % Load the neural network
   ```

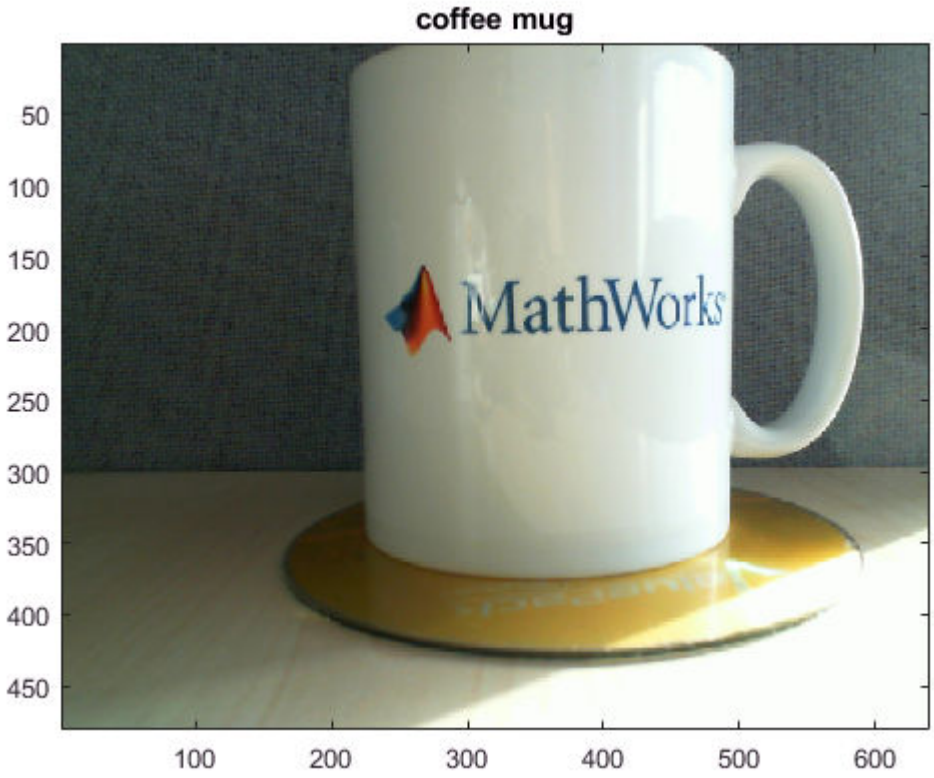
   If you need to install the `webcam` and `alexnet` add-ons, a message from each function appears with a link to help you download the free add-ons using Add-On Explorer. Alternatively, see Deep Learning Toolbox Model *for AlexNet Network* and MATLAB Support Package for USB Webcams.

   After you install Deep Learning Toolbox Model *for AlexNet Network*, you can use it to classify images. AlexNet is a pretrained convolutional neural network (CNN) that has been trained on more than a million images and can classify images into 1000 object categories (for example, keyboard, mouse, coffee mug, pencil, and many animals).
2. Run the following code to show and classify live images. Point the webcam at an object and the neural network reports what class of object it thinks the webcam is showing. It will keep classifying images until you press **Ctrl+C**. The code resizes the image for the network using `imresize`.

   ```
   while true
       im = snapshot(camera);       % Take a picture
       image(im);                   % Show the picture
       im = imresize(im,[227 227]); % Resize the picture for alexnet
       label = classify(net,im);    % Classify the picture
       title(char(label));          % Show the class label
   ```

```
    drawnow
end
```

In this example, the network correctly classifies a coffee mug. Experiment with objects in your surroundings to see how accurate the network is.



To watch a video of this example, see Deep Learning in 11 Lines of MATLAB Code.

To learn how to extend this example and show the probability scores of classes, see “Classify Webcam Images Using Deep Learning” on page 3-2.

For next steps in deep learning, you can use the pretrained network for other tasks. Solve new classification problems on your image data with transfer learning or feature extraction. For examples, see “Start Deep Learning Faster Using Transfer Learning” on page 1-5 and “Train Classifiers Using Features Extracted from Pretrained Networks” on page 1-6. To try other pretrained networks, see “Pretrained Deep Neural Networks” on page 1-8.

Start Deep Learning Faster Using Transfer Learning

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is much faster and easier than training from scratch. You can quickly make the network learn a new task using a smaller number of training images. The advantage of transfer learning is that the pretrained network has already learned a rich set of features that can be applied to a wide range of other similar tasks.

For example, if you take a network trained on thousands or millions of images, you can retrain it for new object detection using only hundreds of images. You can effectively fine-tune a pretrained

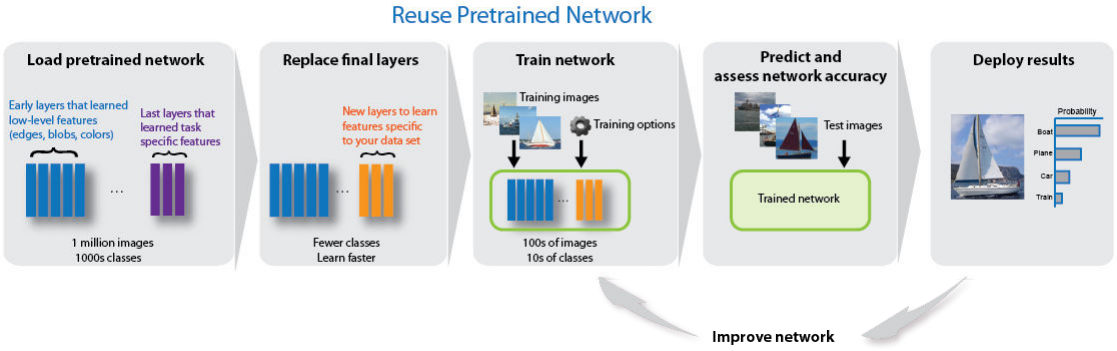
network with much smaller data sets than the original training data. If you have a very large dataset, then transfer learning might not be faster than training a new network.

Transfer learning enables you to:

- Transfer the learned features of a pretrained network to a new problem
- Transfer learning is faster and easier than training a new network
- Reduce training time and dataset size
- Perform deep learning without needing to learn how to create a whole new network

For an interactive example, see “Transfer Learning with Deep Network Designer” on page 2-2.

For a programmatic example, see “Train Deep Learning Network to Classify New Images” on page 3-6.



Train Classifiers Using Features Extracted from Pretrained Networks

Feature extraction allows you to use the power of pretrained networks without investing time and effort into training. Feature extraction can be the fastest way to use deep learning. You extract learned features from a pretrained network, and use those features to train a classifier, for example, a support vector machine (SVM — requires Statistics and Machine Learning Toolbox™). For example, if an SVM trained using `alexnet` can achieve >90% accuracy on your training and validation set, then fine-tuning with transfer learning might not be worth the effort to gain some extra accuracy. If you perform fine-tuning on a small dataset, then you also risk overfitting. If the SVM cannot achieve good enough accuracy for your application, then fine-tuning is worth the effort to seek higher accuracy.

For an example, see “Extract Image Features Using Pretrained Network” on page 3-28.

Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud

Neural networks are inherently parallel algorithms. You can take advantage of this parallelism by using Parallel Computing Toolbox™ to distribute training across multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs.

Training deep networks is extremely computationally intensive and you can usually accelerate training by using a high performance GPU. If you do not have a suitable GPU, you can train on one or more CPU cores instead. You can train a convolutional neural network on a single GPU or CPU, or on

multiple GPUs or CPU cores, or in parallel on a cluster. Using GPU or parallel options requires Parallel Computing Toolbox.

You do not need multiple computers to solve problems using data sets too large to fit in memory. You can use the `imageDatastore` function to work with batches of data without needing a cluster of machines. However, if you have a cluster available, it can be helpful to take your code to the data repository rather than moving large amounts of data around.

To learn more about deep learning with large data sets, see “Deep Learning with Big Data” on page 7-17.

See Also

Related Examples

- “Classify Webcam Images Using Deep Learning” on page 3-2
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Pretrained Deep Neural Networks” on page 1-8
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Deep Learning with Big Data” on page 7-17
- “Recognition, Object Detection, and Semantic Segmentation” (Computer Vision Toolbox)
- “Classify Text Data Using Deep Learning” on page 4-89
- “Deep Learning Tips and Tricks” on page 1-67

Pretrained Deep Neural Networks

In this section...

You can take a pretrained image classification network that has already learned to extract powerful and informative features from natural images and use it as a starting point to learn a new task. The majority of the pretrained networks are trained on a subset of the ImageNet database [1], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [2]. These networks have been trained on more than a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. Using a pretrained network with transfer learning is typically much faster and easier than training a network from scratch.

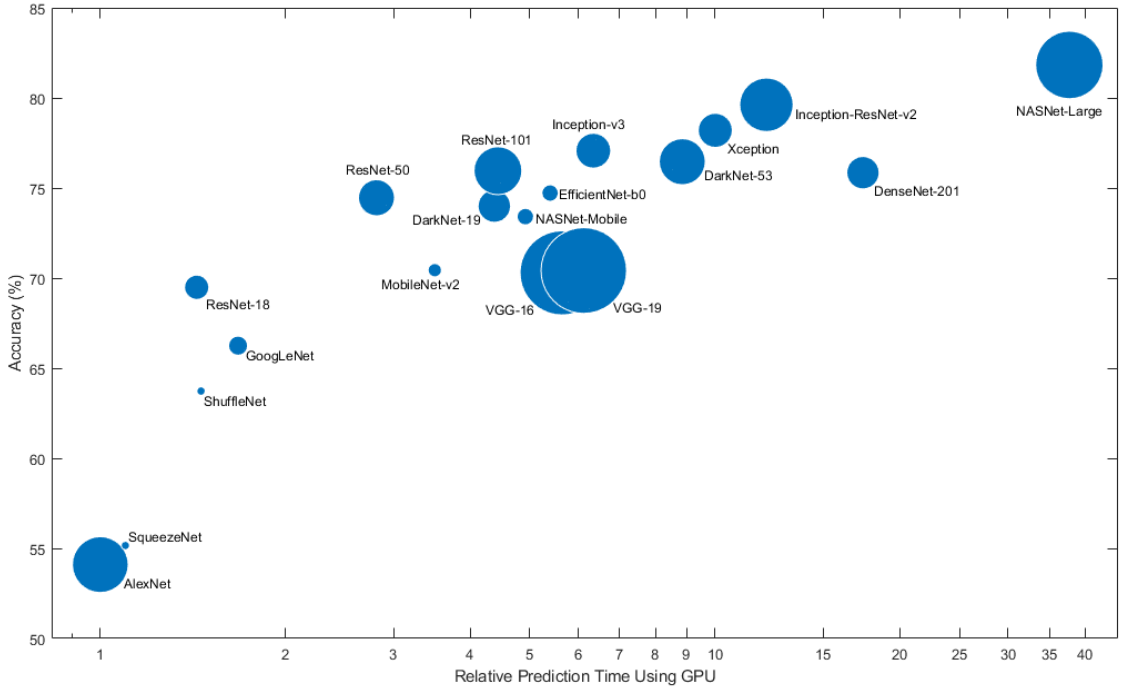
You can use previously trained networks for the following tasks:

Purpose	Description
Classification	Apply pretrained networks directly to classification problems. To classify a new image, use `classify`. For an example showing how to use a pretrained network for classification, see “Classify Image Using GoogLeNet” on page 3-23.
Feature Extraction	Use a pretrained network as a feature extractor by using the layer activations as features. You can use these activations as features to train another machine learning model, such as a support vector machine (SVM). For more information, see “Feature Extraction” on page 1-13. For an example, see “Extract Image Features Using Pretrained Network” on page 3-28.
Transfer Learning	Take layers from a network trained on a large data set and fine-tune on a new data set. For more information, see “Transfer Learning” on page 1-14. For a simple example, see “Get Started with Transfer Learning”. To try more pretrained networks, see “Train Deep Learning Network to Classify New Images” on page 3-6.

Compare Pretrained Networks

Pretrained networks have different characteristics that matter when choosing a network to apply to your problem. The most important characteristics are network accuracy, speed, and size. Choosing a network is generally a tradeoff between these characteristics. Use the plot below to compare the ImageNet validation accuracy with the time required to make a prediction using the network.

Tip To get started with transfer learning, try choosing one of the faster networks, such as SqueezeNet or GoogLeNet. You can then iterate quickly and try out different settings such as data preprocessing steps and training options. Once you have a feeling of which settings work well, try a more accurate network such as Inception-v3 or a ResNet and see if that improves your results.



Note The plot above only shows an indication of the relative speeds of the different networks. The exact prediction and training iteration times depend on the hardware and mini-batch size that you use.

A good network has a high accuracy and is fast. The plot displays the classification accuracy versus the prediction time when using a modern GPU (an NVIDIA® Tesla® P100) and a mini-batch size of 128. The prediction time is measured relative to the fastest network. The area of each marker is proportional to the size of the network on disk.

The classification accuracy on the ImageNet validation set is the most common way to measure the accuracy of networks trained on ImageNet. Networks that are accurate on ImageNet are also often accurate when you apply them to other natural image data sets using transfer learning or feature

extraction. This generalization is possible because the networks have learned to extract powerful and informative features from natural images that generalize to other similar data sets. However, high accuracy on ImageNet does not always transfer directly to other tasks, so it is a good idea to try multiple networks.

If you want to perform prediction using constrained hardware or distribute networks over the Internet, then also consider the size of the network on disk and in memory.

Network Accuracy

There are multiple ways to calculate the classification accuracy on the ImageNet validation set and different sources use different methods. Sometimes an ensemble of multiple models is used and sometimes each image is evaluated multiple times using multiple crops. Sometimes the top-5 accuracy instead of the standard (top-1) accuracy is quoted. Because of these differences, it is often not possible to directly compare the accuracies from different sources. The accuracies of pretrained networks in Deep Learning Toolbox are standard (top-1) accuracies using a single model and single central image crop.

Load Pretrained Networks

To load the SqueezeNet network, type `squeezenet` at the command line.

```
net = squeezenet;
```

For other networks, use functions such as `googlenet` to get links to download pretrained networks from the Add-On Explorer.

The following table lists the available pretrained networks trained on ImageNet and some of their properties. The network depth is defined as the largest number of sequential convolutional or fully connected layers on a path from the input layer to the output layer. The inputs to all networks are RGB images.

Network	Depth	Size	Parameters (Millions)	Image Input Size
`squeezenet`	18	5.2 MB	1.24	227-by-227
`googlenet`	22	27 MB	7.0	224-by-224
`inceptionv3`	48	89 MB	23.9	299-by-299
`densenet201`	201	77 MB	20.0	224-by-224
`mobilenetv2`	53	13 MB	3.5	224-by-224
`resnet18`	18	44 MB	11.7	224-by-224
`resnet50`	50	96 MB	25.6	224-by-224
`resnet101`	101	167 MB	44.6	224-by-224
`xception`	71	85 MB	22.9	299-by-299
`inceptionresnetv2`	164	209 MB	55.9	299-by-299
`shufflenet`	50	5.4 MB	1.4	224-by-224
`nasnetmobile`	*	20 MB	5.3	224-by-224

Network	Depth	Size	Parameters (Millions)	Image Input Size
`nasnetlarge`	*	332 MB	88.9	331-by-331
`darknet19`	19	78 MB	20.8	256-by-256
`darknet53`	53	155 MB	41.6	256-by-256
`efficientnetb0`	82	20 MB	5.3	224-by-224
`alexnet`	8	227 MB	61.0	227-by-227
`vgg16`	16	515 MB	138	224-by-224
`vgg19`	19	535 MB	144	224-by-224

*The NASNet-Mobile and NASNet-Large networks do not consist of a linear sequence of modules.

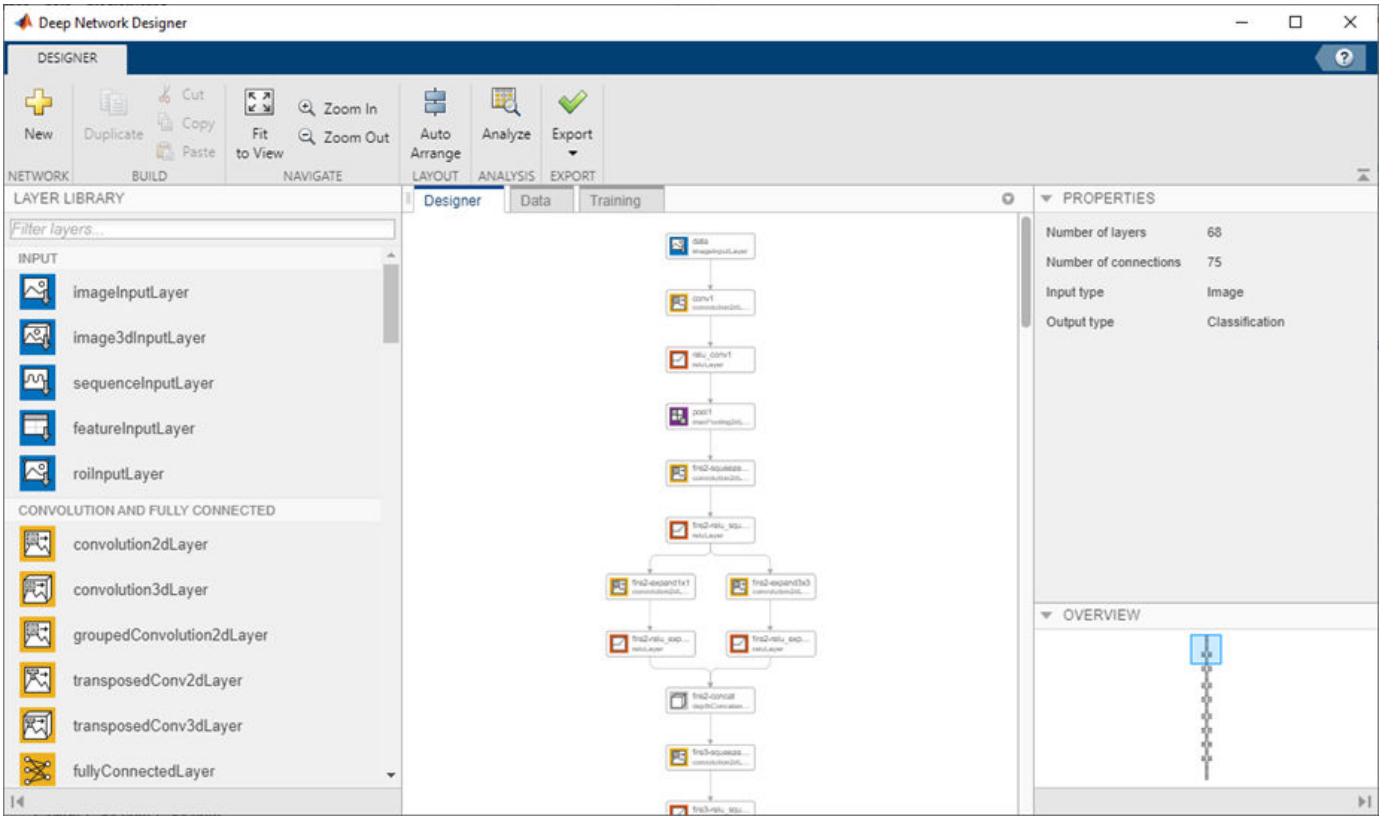
GoogLeNet Trained on Places365

The standard GoogLeNet network is trained on the ImageNet data set but you can also load a network trained on the Places365 data set [3] [4]. The network trained on Places365 classifies images into 365 different place categories, such as field, park, runway, and lobby. To load a pretrained GoogLeNet network trained on the Places365 data set, use `googlenet('Weights','places365')`. When performing transfer learning to perform a new task, the most common approach is to use networks pretrained on ImageNet. If the new task is similar to classifying scenes, then using the network trained on Places365 could give higher accuracies.

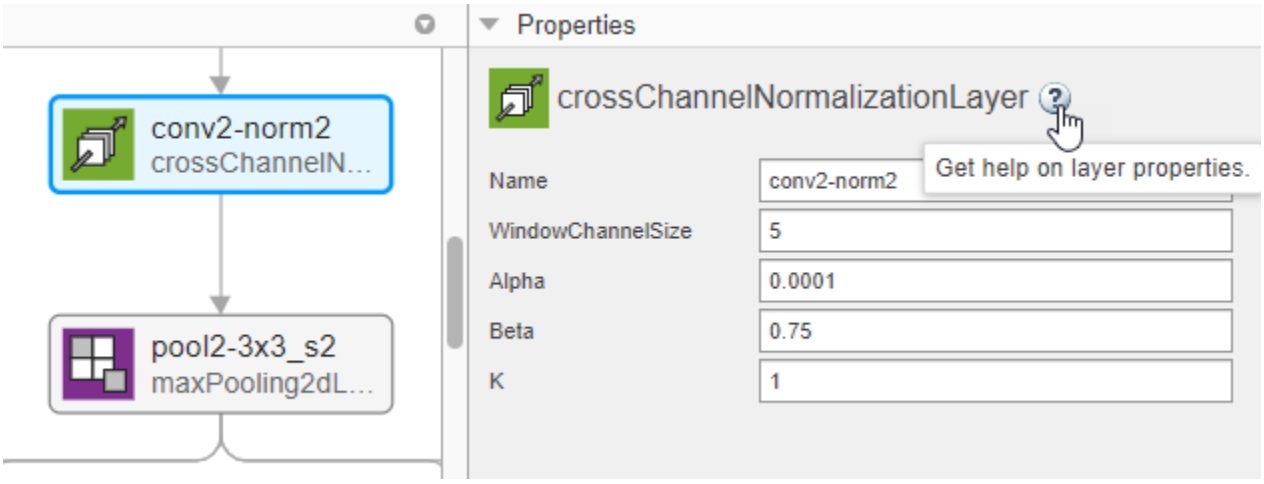
Visualize Pretrained Networks

You can load and visualize pretrained networks using **Deep Network Designer**.

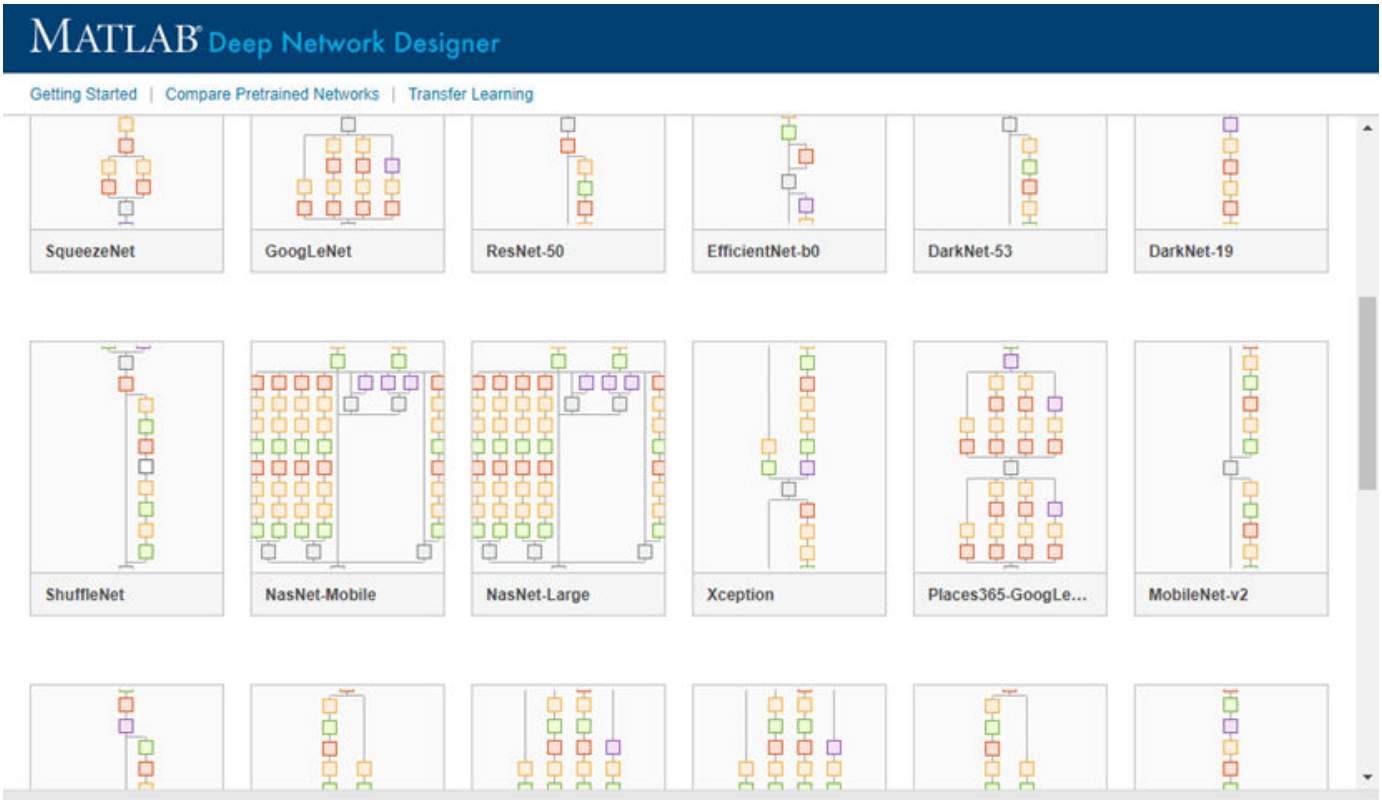
```
deepNetworkDesigner(squeezenet)
```



To view and edit layer properties, select a layer. Click the help icon next to the layer name for information on the layer properties.



Explore other pretrained networks in Deep Network Designer by clicking **New**.



If you need to download a network, pause on the desired network and click **Install** to open the Add-On Explorer.

Feature Extraction

Feature extraction is an easy and fast way to use the power of deep learning without investing time and effort into training a full network. Because it only requires a single pass over the training images, it is especially useful if you do not have a GPU. You extract learned image features using a pretrained network, and then use those features to train a classifier, such as a support vector machine using `fitcsvm`.

Try feature extraction when your new data set is very small. Since you only train a simple classifier on the extracted features, training is fast. It is also unlikely that fine-tuning deeper layers of the network improves the accuracy since there is little data to learn from.

- If your data is very similar to the original data, then the more specific features extracted deeper in the network are likely to be useful for the new task.
- If your data is very different from the original data, then the features extracted deeper in the network might be less useful for your task. Try training the final classifier on more general features extracted from an earlier network layer. If the new data set is large, then you can also try training a network from scratch.

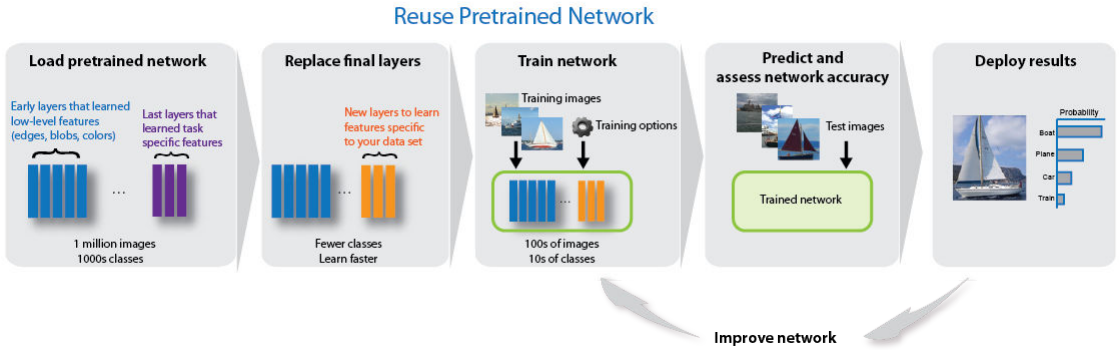
ResNets are often good feature extractors. For an example showing how to use a pretrained network for feature extraction, see “Extract Image Features Using Pretrained Network” on page 3-28.

Transfer Learning

You can fine-tune deeper layers in the network by training the network on your new data set with the pretrained network as a starting point. Fine-tuning a network with transfer learning is often faster and easier than constructing and training a new network. The network has already learned a rich set of image features, but when you fine-tune the network it can learn features specific to your new data set. If you have a very large data set, then transfer learning might not be faster than training from scratch.

Tip Fine-tuning a network often gives the highest accuracy. For very small data sets (fewer than about 20 images per class), try feature extraction instead.

Fine-tuning a network is slower and requires more effort than simple feature extraction, but since the network can learn to extract a different set of features, the final network is often more accurate. Fine-tuning usually works better than feature extraction as long as the new data set is not very small, because then the network has data to learn new features from. For examples showing how to perform transfer learning, see “Transfer Learning with Deep Network Designer” on page 2-2 and “Train Deep Learning Network to Classify New Images” on page 3-6.



Import and Export Networks

You can import networks and network architectures from TensorFlow®-Keras, Caffe, and the ONNX™ (Open Neural Network Exchange) model format. You can also export trained networks to the ONNX model format.

Import from Keras

Import pretrained networks from TensorFlow-Keras by using `importKerasNetwork`. You can import the network and weights either from the same HDF5 (.h5) file or separate HDF5 and JSON (.json) files. For more information, see `importKerasNetwork`.

Import network architectures from TensorFlow-Keras by using `importKerasLayers`. You can import the network architecture, either with or without weights. You can import the network architecture and weights either from the same HDF5 (.h5) file or separate HDF5 and JSON (.json) files. For more information, see `importKerasLayers`.

Import from Caffe

Import pretrained networks from Caffe by using the `importCaffeNetwork` function. There are many pretrained networks available in Caffe Model Zoo [5]. Download the desired `.prototxt`

and `.caffemodel` files and use `importCaffeNetwork` to import the pretrained network into MATLAB. For more information, see `importCaffeNetwork`.

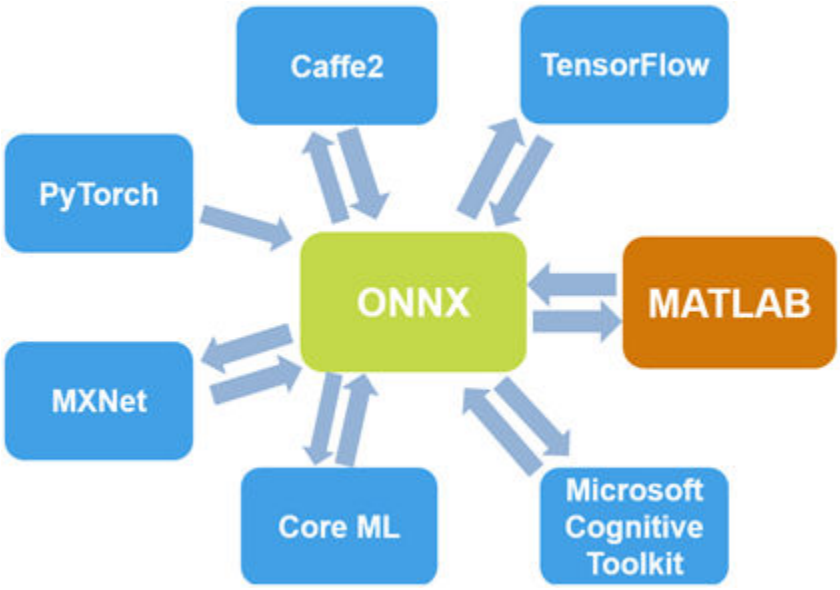
You can import network architectures of Caffe networks. Download the desired `.prototxt` file and use `importCaffeLayers` to import the network layers into MATLAB. For more information, see `importCaffeLayers`.

Export to and Import from ONNX

By using ONNX as an intermediate format, you can interoperate with other deep learning frameworks that support ONNX model export or import, such as TensorFlow, PyTorch, Caffe2, Microsoft® Cognitive Toolkit (CNTK), Core ML, and Apache MXNet.

Export a trained Deep Learning Toolbox network to the ONNX model format by using the `exportONNXNetwork` function. You can then import the ONNX model to other deep learning frameworks that support ONXX model import.

Import pretrained networks from ONNX using `importONNXNetwork` and import network architectures with or without weights using `importONNXLayers`.



Pretrained Networks for Audio Applications

Use pretrained networks for audio and speech processing applications by using Deep Learning Toolbox together with Audio Toolbox™.

Audio Toolbox provides the pretrained VGGish and YAMNet networks. Use the `vggish` and `yamnet` functions to interact directly with the pretrained networks. The `classifySound` function performs required preprocessing and postprocessing for YAMNet so that you can locate and classify sounds into one of 521 categories. You can explore the YAMNet ontology using the `yamnetGraph` function. The `vggishFeatures` function performs necessary preprocessing and postprocessing for VGGish so that you can extract feature embeddings to input to machine learning and deep learning systems. For more information on using deep learning for audio applications, see “Introduction to Deep Learning for Audio Applications” (Audio Toolbox).

Use VGGish and YAMNet to perform transfer learning and feature extraction. For example, see "Transfer Learning with Pretrained Audio Networks" (Audio Toolbox).

Pretrained Models on GitHub

To find the latest pretrained models and examples, see MATLAB Deep Learning (GitHub).

For example:

- For transformer models, such as GPT-2, BERT, and FinBERT, see the Transformer Models for MATLAB GitHub® repository.
- For pretrained YOLO-v4 object detection models such as YOLOv4-coco and YOLOv4-tiny-coco, see the Pretrained YOLO-v4 Network for Object Detection GitHub repository.

References

[1] *ImageNet*. http://www.image-net.org

[2] Russakovsky, O., Deng, J., Su, H., et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision (IJCV)*. Vol 115, Issue 3, 2015, pp. 211–252

[3] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. "Places: An image database for deep scene understanding." *arXiv preprint arXiv:1610.02055* (2016).

[4] *Places*. http://places2.csail.mit.edu/

[5] *Caffe Model Zoo*. http://caffe.berkeleyvision.org/model_zoo.html

See Also

`alexnet` | `googlenet` | `inceptionv3` | `densenet201` | `darknet19` | `darknet53` | `resnet18` | `resnet50` | `resnet101` | `vgg16` | `vgg19` | `shufflenet` | `nasnetmobile` | `nasnetlarge` | `mobilenetv2` | `xception` | `inceptionresnetv2` | `squeezenet` | `importCaffeNetwork` | `importCaffeLayers` | `importKerasLayers` | `importKerasNetwork` | `exportONNXNetwork` | `importONNXLayers` | `importONNXNetwork` | **Deep Network Designer**

Related Examples

- "Deep Learning in MATLAB" on page 1-2
- "Transfer Learning with Deep Network Designer" on page 2-2
- "Extract Image Features Using Pretrained Network" on page 3-28
- "Classify Image Using GoogLeNet" on page 3-23
- "Train Deep Learning Network to Classify New Images" on page 3-6
- "Visualize Features of a Convolutional Neural Network" on page 5-156
- "Visualize Activations of a Convolutional Neural Network" on page 5-141
- "Deep Dream Images Using GoogLeNet" on page 5-15

Learn About Convolutional Neural Networks

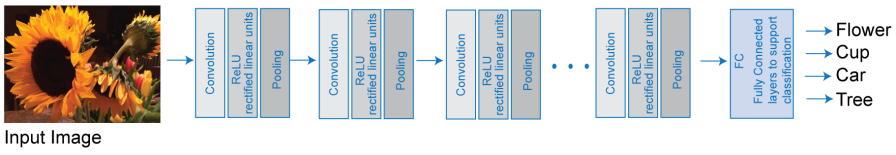
Convolutional neural networks (ConvNets) are widely used tools for deep learning. They are specifically suitable for images as inputs, although they are also used for other applications such as text, signals, and other continuous responses. They differ from other types of neural networks in a few ways:

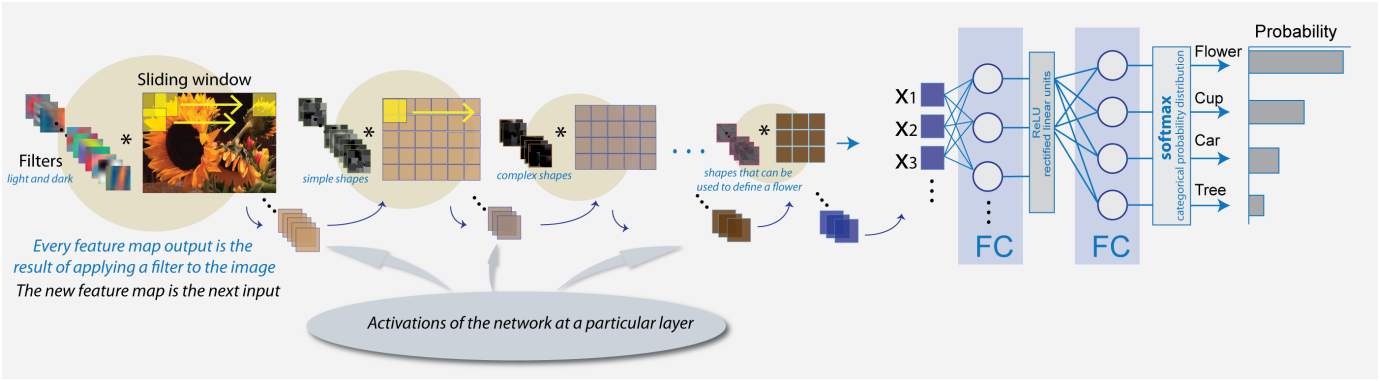
Convolutional neural networks are inspired from the biological structure of a visual cortex, which contains arrangements of simple and complex cells [1]. These cells are found to activate based on the subregions of a visual field. These subregions are called receptive fields. Inspired from the findings of this study, the neurons in a convolutional layer connect to the subregions of the layers before that layer instead of being fully-connected as in other types of neural networks. The neurons are unresponsive to the areas outside of these subregions in the image.

These subregions might overlap, hence the neurons of a ConvNet produce spatially-correlated outcomes, whereas in other types of neural networks, the neurons do not share any connections and produce independent outcomes.

In addition, in a neural network with fully-connected neurons, the number of parameters (weights) can increase quickly as the size of the input increases. A convolutional neural network reduces the number of parameters with the reduced number of connections, shared weights, and downsampling.

A ConvNet consists of multiple layers, such as convolutional layers, max-pooling or average-pooling layers, and fully-connected layers.





The neurons in each layer of a ConvNet are arranged in a 3-D manner, transforming a 3-D input to a 3-D output. For example, for an image input, the first layer (input layer) holds the images as 3-D inputs, with the dimensions being height, width, and the color channels of the image. The neurons in the first convolutional layer connect to the regions of these images and transform them into a 3-D output. The hidden units (neurons) in each layer learn nonlinear combinations of the original inputs, which is called feature extraction [2]. These learned features, also known as activations, from one layer become the inputs for the next layer. Finally, the learned features become the inputs to the classifier or the regression function at the end of the network.

The architecture of a ConvNet can vary depending on the types and numbers of layers included. The types and number of layers included depends on the particular application or data. For example, if you have categorical responses, you must have a classification function and a classification layer, whereas if your response is continuous, you must have a regression layer at the end of the network. A smaller network with only one or two convolutional layers might be sufficient to learn a small number of gray scale image data. On the other hand, for more complex data with millions of colored images, you might need a more complicated network with multiple convolutional and fully connected layers.

You can concatenate the layers of a convolutional neural network in MATLAB in the following way:

```
layers = [imageInputLayer([28 28 1])
          convolution2dLayer(5,20)
          reluLayer
          maxPooling2dLayer(2,'Stride',2)
          fullyConnectedLayer(10)
          softmaxLayer
          classificationLayer];
```

After defining the layers of your network, you must specify the training options using the `trainingOptions` function. For example,

```
options = trainingOptions('sgdm');
```

Then, you can train the network with your training data using the `trainNetwork` function. The data, layers, and training options become the inputs to the training function. For example,

```
convnet = trainNetwork(data,layers,options);
```

For detailed discussion of layers of a ConvNet, see “Specify Layers of Convolutional Neural Network” on page 1-31. For setting up training parameters, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

References

[1] Hubel, H. D. and Wiesel, T. N. '' Receptive Fields of Single neurones in the Cat’s Striate Cortex.'' *Journal of Physiology*. Vol 148, pp. 574-591, 1959.

[2] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.

See Also

`trainNetwork` | `trainingOptions`

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Get Started with Transfer Learning”
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Pretrained Deep Neural Networks” on page 1-8

Multiple-Input and Multiple-Output Networks

In Deep Learning Toolbox, you can define network architectures with multiple inputs (for example, networks trained on multiple sources and types of data) or multiple outputs (for example, networks that predicts both classification and regression responses).

Multiple-Input Networks

Define networks with multiple inputs when the network requires data from multiple sources or in different formats. For example, networks that require image data captured from multiple sensors at different resolutions.

Training

To define and train a deep learning network with multiple inputs, specify the network architecture using a `layerGraph` object and train using the `trainNetwork` function with datastore input.

To use a datastore for networks with multiple input layers, use the `combine` and `transform` functions to create a datastore that outputs a cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs. In this case, the first `numInputs` columns specify the predictors for each input and the last column specifies the responses. The order of inputs is given by the `InputNames` property of the layer graph `layers`.

Tip If the network also has multiple outputs, then you must use a custom training loop. for more information, see “Multiple-Output Networks” on page 1-19.

Prediction

To make predictions on a trained deep learning network with multiple inputs, use either the `predict` or `classify` function. Specify multiple inputs using one of the following:

- `combinedDatastore` object
- `transformedDatastore` object
- multiple numeric arrays

Multiple-Output Networks

Define networks with multiple outputs for tasks requiring multiple responses in different formats. For example, tasks requiring both categorical and numeric output.

Training

To train a deep learning network with multiple outputs, use a custom training loop. For an example, see “Train Network with Multiple Outputs” on page 3-61.

Prediction

To make predictions using a model function, use the model function directly with the trained parameters. For an example, see “Make Predictions Using Model Function” on page 18-286.

Alternatively, convert the model function to a `DAGNetwork` object using the `assembleNetwork` function. With the assembled network, you can:

- Make predictions with other data types such as datastores using the `predict` function for `DAGNetwork` objects.
- Specify prediction options such as the mini-batch size using the `predict` function for `DAGNetwork` objects.
- Save the network in a MAT file.

For an example, see “Assemble Multiple-Output Network for Prediction” on page 18-196.

See Also

`assembleNetwork` | `predict`

More About

- “Train Network with Multiple Outputs” on page 3-61
- “Assemble Multiple-Output Network for Prediction” on page 18-196
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Deep Learning Layers” on page 1-21

List of Deep Learning Layers

This page provides a list of deep learning layers in MATLAB.

To learn how to create networks from layers for different tasks, see the following examples.

Task	Learn More
Create deep learning networks for image classification or regression.	“Create Simple Deep Learning Network for Classification” on page 3-47 “Train Convolutional Neural Network for Regression” on page 3-53 “Train Residual Network for Image Classification” on page 3-13
Create deep learning networks for sequence and time series data.	“Sequence Classification Using Deep Learning” on page 4-2 “Time Series Forecasting Using Deep Learning” on page 4-15
Create deep learning network for audio data.	“Speech Command Recognition Using Deep Learning” on page 4-23
Create deep learning network for text data.	“Classify Text Data Using Deep Learning” on page 4-89 “Generate Text Using Deep Learning” on page 4-180

Deep Learning Layers

Use the following functions to create different layer types. Alternatively, use the **Deep Network Designer** app to create networks interactively.

To learn how to define your own custom layers, see “Define Custom Deep Learning Layers” on page 18-9.

Input Layers

Layer	Description
 `imageInputLayer`	An image input layer inputs 2-D images to a network and applies data normalization.
 `image3dInputLayer`	A 3-D image input layer inputs 3-D images or volumes to a network and applies data normalization.
 `sequenceInputLayer`	A sequence input layer inputs sequence data to a network.

Layer	Description
 `featureInputLayer`	A feature input layer inputs feature data to a network and applies data normalization. Use this layer when you have a data set of numeric scalars representing features (data without spatial or time dimensions).
 `roiInputLayer`	An ROI input layer inputs images to a Fast R-CNN object detection network.

Convolution and Fully Connected Layers

Layer	Description
 `convolution1dLayer`	A 1-D convolutional layer applies sliding convolutional filters to 1-D input.
 `convolution2dLayer`	A 2-D convolutional layer applies sliding convolutional filters to 2-D input.
 `convolution3dLayer`	A 3-D convolutional layer applies sliding cuboidal convolution filters to 3-D input.
 `groupedConvolution2dLayer`	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution.
 `transposedConv2dLayer`	A transposed 2-D convolution layer upsamples feature maps.
 `transposedConv3dLayer`	A transposed 3-D convolution layer upsamples three-dimensional feature maps.
 `fullyConnectedLayer`	A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.

Sequence Layers

Layer	Description
 `sequenceInputLayer`	A sequence input layer inputs sequence data to a network.
 `lstmLayer`	An LSTM layer learns long-term dependencies between time steps in time series and sequence data.
 `bilstmLayer`	A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.

Layer	Description
 `gruLayer`	A GRU layer learns dependencies between time steps in time series and sequence data.
 `convolution1dLayer`	A 1-D convolutional layer applies sliding convolutional filters to 1-D input.
 `maxPooling1dLayer`	A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.
 `averagePooling1dLayer`	A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region.
 `globalMaxPooling1dLayer`	A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.
 `sequenceFoldingLayer`	A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.
 `sequenceUnfoldingLayer`	A sequence unfolding layer restores the sequence structure of the input data after sequence folding.
 `flattenLayer`	A flatten layer collapses the spatial dimensions of the input into the channel dimension.
 `wordEmbeddingLayer`	A word embedding layer maps word indices to vectors.
 `peepholeLSTMLayer` on page 18-75 (Custom layer example)	

Activation Layers

Layer	Description
 `reluLayer`	A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.
 `leakyReluLayer`	A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.
 `clippedReluLayer`	A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the *clipping ceiling* is set to that clipping ceiling.

Layer	Description
 `eluLayer`	An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.
 `tanhLayer`	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.
 `swishLayer`	A swish activation layer applies the swish function on the layer inputs.
 `softplusLayer`	A softplus layer applies the softplus activation function $Y = \log(1 + e^X)$, which ensures that the output is always positive. This activation function is a smooth continuous version of `reluLayer`. You can incorporate this layer into the deep neural networks you define for actors in reinforcement learning agents. This layer is useful for creating continuous Gaussian policy deep neural networks, for which the standard deviation output must be positive.
 `functionLayer`	A function layer applies a specified function to the layer input.
 `preluLayer` on page 18-35 (Custom layer example)	A peephole LSTM layer is a variant of an LSTM layer, where the gate calculations use the layer cell state.

Normalization, Dropout, and Cropping Layers

Layer	Description
 `batchNormalizationLayer`	A batch normalization layer normalizes a mini-batch of data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.
 `groupNormalizationLayer`	A group normalization layer normalizes a mini-batch of data across grouped subsets of channels for each observation independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use group normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

Layer	Description
 `instanceNormalizationLayer`	An instance normalization layer normalizes a mini-batch of data across each channel for each observation independently. To improve the convergence of training the convolutional neural network and reduce the sensitivity to network hyperparameters, use instance normalization layers between convolutional layers and nonlinearities, such as ReLU layers.
 `layerNormalizationLayer`	A layer normalization layer normalizes a mini-batch of data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization layers after the learnable layers, such as LSTM and fully connected layers.
 `crossChannelNormalizationLayer`	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.
 `dropoutLayer`	A dropout layer randomly sets input elements to zero with a given probability.
 `crop2dLayer`	A 2-D crop layer applies 2-D cropping to the input.
 `crop3dLayer`	A 3-D crop layer crops a 3-D volume to the size of the input feature map.
 `scalingLayer`	A scaling layer linearly scales and biases an input array `U`, giving an output `Y = Scale.*U + Bias`. You can incorporate this layer into the deep neural networks you define for actors or critics in reinforcement learning agents. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and sigmoid.
 `quadraticLayer`	A quadratic layer takes an input vector and outputs a vector of quadratic monomials constructed from the input elements. This layer is useful when you need a layer whose output is some quadratic function of its inputs. For example, to recreate the structure of quadratic value functions such as those used in LQR controller design.
 `resize2dLayer`	A 2-D resize layer resizes 2-D input by a scale factor, to a specified height and width, or to the size of a reference input feature map.

Layer	Description
 `resize3dLayer`	A 3-D resize layer resizes 3-D input by a scale factor, to a specified height, width, and depth, or to the size of a reference input feature map.
 `stftLayer`	An STFT layer computes the short-time Fourier transform of the input.

Pooling and Unpooling Layers

Layer	Description
 `averagePooling1dLayer`	A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region.
 `averagePooling2dLayer`	A 2-D average pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the average values of each region.
 `averagePooling3dLayer`	A 3-D average pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the average values of each region.
 `globalAveragePooling1dLayer`	A 1-D global average pooling layer performs downsampling by outputting the average of the time or spatial dimensions of the input.
 `globalAveragePooling2dLayer`	A 2-D global average pooling layer performs downsampling by computing the mean of the height and width dimensions of the input.
 `globalAveragePooling3dLayer`	A 3-D global average pooling layer performs downsampling by computing the mean of the height, width, and depth dimensions of the input.
 `maxPooling1dLayer`	A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.
 `maxPooling2dLayer`	A 2-D max pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the maximum of each region.
 `maxPooling3dLayer`	A 3-D max pooling layer performs downsampling by dividing three-dimensional input into cuboidal pooling regions, then computing the maximum of each region.
 `globalMaxPooling1dLayer`	A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.

Layer	Description
 `globalMaxPooling2dLayer`	A 2-D global max pooling layer performs downsampling by computing the maximum of the height and width dimensions of the input.
 `globalMaxPooling3dLayer`	A 3-D global max pooling layer performs downsampling by computing the maximum of the height, width, and depth dimensions of the input.
 `maxUnpooling2dLayer`	A 2-D max unpooling layer unpools the output of a 2-D max pooling layer.

Combination Layers

Layer	Description
 `additionLayer`	An addition layer adds inputs from multiple neural network layers element-wise.
 `multiplicationLayer`	A multiplication layer multiplies inputs from multiple neural network layers element-wise.
 `depthConcatenationLayer`	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).
 `concatenationLayer`	A concatenation layer takes inputs and concatenates them along a specified dimension. The inputs must have the same size in all dimensions except the concatenation dimension.
 `weightedAdditionLayer` on page 18-48 (Custom layer example)	A weighted addition layer scales and adds inputs from multiple neural network layers element-wise.

Object Detection Layers

Layer	Description
 `roiInputLayer`	An ROI input layer inputs images to a Fast R-CNN object detection network.
 `roiMaxPooling2dLayer`	An ROI max pooling layer outputs fixed size feature maps for every rectangular ROI within the input feature map. Use this layer to create a Fast or Faster R-CNN object detection network.
 `roiAlignLayer`	An ROI align layer outputs fixed size feature maps for every rectangular ROI within an input feature map. Use this layer to create a Mask R-CNN network.
 `anchorBoxLayer`	An anchor box layer stores anchor boxes for a feature map used in object detection networks.

Layer	Description
 `regionProposalLayer`	A region proposal layer outputs bounding boxes around potential objects in an image as part of the region proposal network (RPN) within Faster R-CNN.
 `ssdMergeLayer`	An SSD merge layer merges the outputs of feature maps for subsequent regression and classification loss computation.
 `yolov2TransformLayer`	A transform layer of the you only look once version 2 (YOLO v2) network transforms the bounding box predictions of the last convolution layer in the network to fall within the bounds of the ground truth. Use the transform layer to improve the stability of the YOLO v2 network.
 `spaceToDepthLayer`	A space to depth layer permutes the spatial blocks of the input into the depth dimension. Use this layer when you need to combine feature maps of different size without discarding any feature data.
 `depthToSpace2dLayer`	A 2-D depth to space layer permutes data from the depth dimension into blocks of 2-D spatial data.
 `rpnSoftmaxLayer`	A region proposal network (RPN) softmax layer applies a softmax activation function to the input. Use this layer to create a Faster R-CNN object detection network.
 `focalLossLayer`	A focal loss layer predicts object classes using focal loss.
 `rpnClassificationLayer`	A region proposal network (RPN) classification layer classifies image regions as either *object* or *background* by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.
 `rcnnBoxRegressionLayer`	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.

Output Layers

Layer	Description
 `softmaxLayer`	A softmax layer applies a softmax function to the input.
 `sigmoidLayer`	A sigmoid layer applies a sigmoid function to the input such that the output is bounded in the interval (0,1).

Layer	Description
 `classificationLayer`	A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes.
 `regressionLayer`	A regression layer computes the half-mean-squared-error loss for regression tasks.
 `pixelClassificationLayer`	A pixel classification layer provides a categorical label for each image pixel or voxel.
 `dicePixelClassificationLayer`	A Dice pixel classification layer provides a categorical label for each image pixel or voxel using generalized Dice loss.
 `focalLossLayer`	A focal loss layer predicts object classes using focal loss.
 `rpnSoftmaxLayer`	A region proposal network (RPN) softmax layer applies a softmax activation function to the input. Use this layer to create a Faster R-CNN object detection network.
 `rpnClassificationLayer`	A region proposal network (RPN) classification layer classifies image regions as either *object* or *background* by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.
 `rcnnBoxRegressionLayer`	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.
 `yolov2OutputLayer`	An output layer of the you only look once version 2 (YOLO v2) network refines the bounding box locations by minimizing the mean squared error loss between the predicted locations and ground truth.
 `tverskyPixelClassificationLayer` on page 8-181 (Custom layer example)	A Tversky pixel classification layer provides a categorical label for each image pixel or voxel using Tversky loss.
 `sseClassificationLayer` on page 18-91 (Custom layer example)	A classification SSE layer computes the sum of squares error loss for classification problems.
 `maeRegressionLayer` on page 18-99 (Custom layer example)	A regression MAE layer computes the mean absolute error loss for regression problems.

See Also

`trainingOptions` | `trainNetwork` | **Deep Network Designer**

More About

- “Build Networks with Deep Network Designer” on page 2-15
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Define Custom Deep Learning Layers” on page 18-9
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Sequence Classification Using Deep Learning” on page 4-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning Tips and Tricks” on page 1-67

Specify Layers of Convolutional Neural Network

In this section...

The first step of creating and training a new convolutional neural network (ConvNet) is to define the network architecture. This topic explains the details of ConvNet layers, and the order they appear in a ConvNet. For a complete list of deep learning layers and how to create them, see “List of Deep Learning Layers” on page 1-21. To learn about LSTM networks for sequence classification and regression, see “Long Short-Term Memory Networks” on page 1-75. To learn how to create your own custom layers, see “Define Custom Deep Learning Layers” on page 18-9.

The network architecture can vary depending on the types and numbers of layers included. The types and number of layers included depends on the particular application or data. For example, classification networks typically have a softmax layer and a classification layer, whereas regression networks must have a regression layer at the end of the network. A smaller network with only one or two convolutional layers might be sufficient to learn on a small number of grayscale image data. On the other hand, for more complex data with millions of colored images, you might need a more complicated network with multiple convolutional and fully connected layers.

To specify the architecture of a deep network with all layers connected sequentially, create an array of layers directly. For example, to create a deep network which classifies 28-by-28 grayscale images into 10 classes, specify the layer array

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,16,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding',1)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

`layers` is an array of `Layer` objects. You can then use `layers` as an input to the training function `trainNetwork`.

To specify the architecture of a neural network with all layers connected sequentially, create an array of layers directly. To specify the architecture of a network where layers can have multiple inputs or outputs, use a `LayerGraph` object.

Image Input Layer

Create an image input layer using `imageInputLayer`.

An image input layer inputs images to a network and applies data normalization.

Specify the image size using the `inputSize` argument. The size of an image corresponds to the height, width, and the number of color channels of that image. For example, for a grayscale image, the number of channels is 1, and for a color image it is 3.

Convolutional Layer

A 2-D convolutional layer applies sliding convolutional filters to 2-D input. Create a 2-D convolutional layer using `convolution2dLayer`.

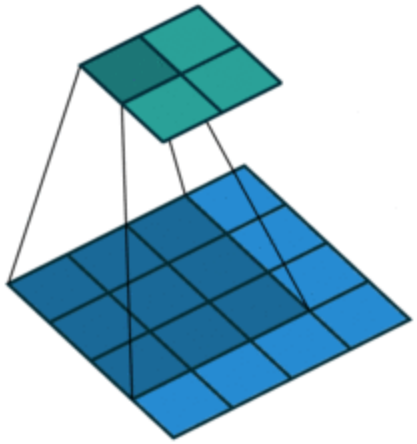
The convolutional layer consists of various components.[1]

Filters and Stride

A convolutional layer consists of neurons that connect to subregions of the input images or the outputs of the previous layer. The layer learns the features localized by these regions while scanning through an image. When creating a layer using the `convolution2dLayer` function, you can specify the size of these regions using the `filterSize` input argument.

For each region, the `trainNetwork` function computes a dot product of the weights and the input, and then adds a bias term. A set of weights that is applied to a region in the image is called a *filter*. The filter moves along the input image vertically and horizontally, repeating the same computation for each region. In other words, the filter convolves the input.

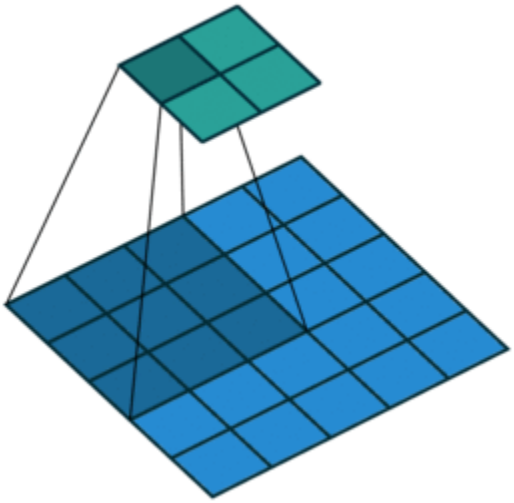
This image shows a 3-by-3 filter scanning through the input. The lower map represents the input and the upper map represents the output.



1. Image credit: Convolution arithmetic (License)

The step size with which the filter moves is called a *stride*. You can specify the step size with the `Stride` name-value pair argument. The local regions that the neurons connect to can overlap depending on the `filterSize` and `'Stride'` values.

This image shows a 3-by-3 filter scanning through the input with a stride of 2. The lower map represents the input and the upper map represents the output.



The number of weights in a filter is *h* * *w* * *c*, where *h* is the height, and *w* is the width of the filter, respectively, and *c* is the number of channels in the input. For example, if the input is a color image, the number of color channels is 3. The number of filters determines the number of channels in the output of a convolutional layer. Specify the number of filters using the `numFilters` argument with the `convolution2dLayer` function.

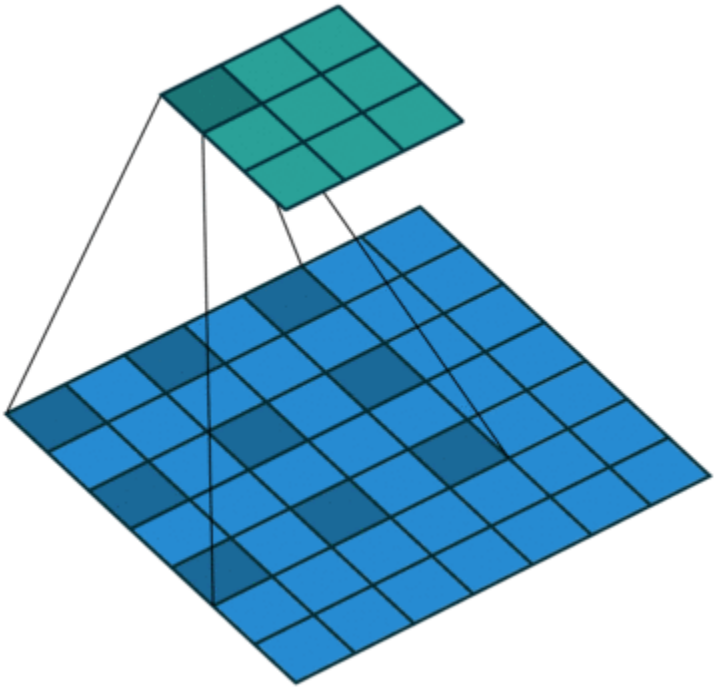
Dilated Convolution

A dilated convolution is a convolution in which the filters are expanded by spaces inserted between the elements of the filter. Specify the dilation factor using the `'DilationFactor'` property.

Use dilated convolutions to increase the receptive field (the area of the input which the layer can see) of the layer without increasing the number of parameters or computation.

The layer expands the filters by inserting zeros between each filter element. The dilation factor determines the step size for sampling the input or equivalently the upsampling factor of the filter. It corresponds to an effective filter size of (*Filter Size* – 1) .* *Dilation Factor* + 1. For example, a 3-by-3 filter with the dilation factor `[2 2]` is equivalent to a 5-by-5 filter with zeros between the elements.

This image shows a 3-by-3 filter dilated by a factor of two scanning through the input. The lower map represents the input and the upper map represents the output.



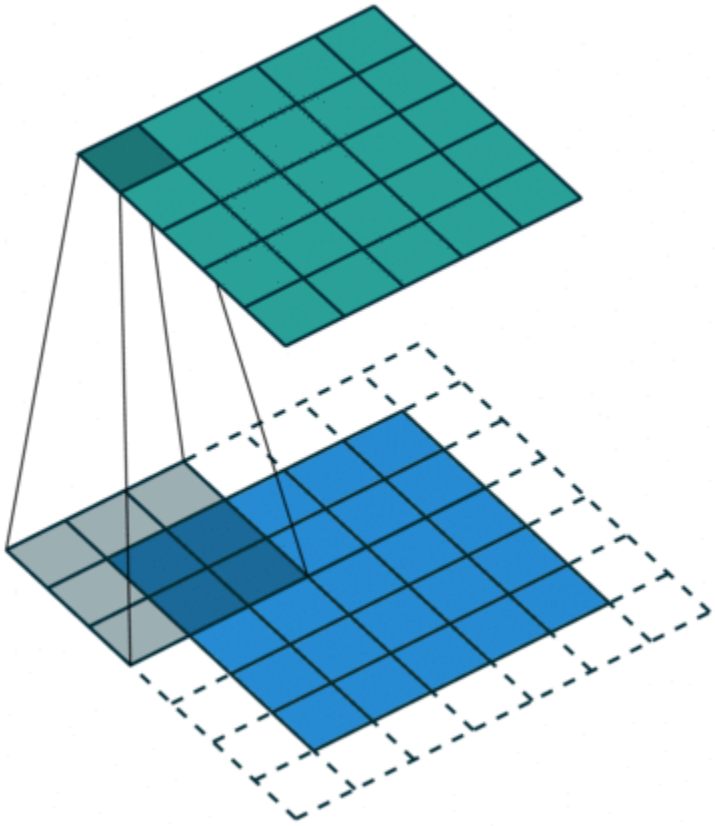
Feature Maps

As a filter moves along the input, it uses the same set of weights and the same bias for the convolution, forming a *feature map*. Each feature map is the result of a convolution using a different set of weights and a different bias. Hence, the number of feature maps is equal to the number of filters. The total number of parameters in a convolutional layer is ((*h***w***c* + 1)**Number of Filters*), where 1 is the bias.

Padding

You can also apply padding to input image borders vertically and horizontally using the `'Padding'` name-value pair argument. Padding is values appended to the borders of a the input to increase its size. By adjusting the padding, you can control the output size of the layer.

This image shows a 3-by-3 filter scanning through the input with padding of size 1. The lower map represents the input and the upper map represents the output.



Output Size

The output height and width of a convolutional layer is (*Input Size* – ((*Filter Size* – 1)**Dilation Factor* + 1) + 2**Padding*)/*Stride* + 1. This value must be an integer for the whole image to be fully covered. If the combination of these options does not lead the image to be fully covered, the software by default ignores the remaining part of the image along the right and bottom edges in the convolution.

Number of Neurons

The product of the output height and width gives the total number of neurons in a feature map, say *Map Size*. The total number of neurons (output size) in a convolutional layer is *Map Size***Number of Filters*.

For example, suppose that the input image is a 32-by-32-by-3 color image. For a convolutional layer with eight filters and a filter size of 5-by-5, the number of weights per filter is 5 * 5 * 3 = 75, and the total number of parameters in the layer is (75 + 1) * 8 = 608. If the stride is 2 in each direction and padding of size 2 is specified, then each feature map is 16-by-16. This is because (32 – 5 + 2 * 2)/2 + 1 = 16.5, and some of the outermost padding to the right and bottom of the image is discarded. Finally, the total number of neurons in the layer is 16 * 16 * 8 = 2048.

Usually, the results from these neurons pass through some form of nonlinearity, such as rectified linear units (ReLU).

Learning Parameters

You can adjust the learning rates and regularization options for the layer using name-value pair arguments while defining the convolutional layer. If you choose not to specify these options, then `trainNetwork` uses the global training options defined with the `trainingOptions` function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

Number of Layers

A convolutional neural network can consist of one or multiple convolutional layers. The number of convolutional layers depends on the amount and complexity of the data.

Batch Normalization Layer

Create a batch normalization layer using `batchNormalizationLayer`.

A batch normalization layer normalizes a mini-batch of data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

The layer first normalizes the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the layer shifts the input by a learnable offset β and scales it by a learnable scale factor γ. β and γ are themselves learnable parameters that are updated during network training.

Batch normalization layers normalize the activations and gradients propagating through a neural network, making network training an easier optimization problem. To take full advantage of this fact, you can try increasing the learning rate. Since the optimization problem is easier, the parameter updates can be larger and the network can learn faster. You can also try reducing the L_2 and dropout regularization. With batch normalization layers, the activations of a specific image during training depend on which images happen to appear in the same mini-batch. To take full advantage of this regularizing effect, try shuffling the training data before every training epoch. To specify how often to shuffle the data during training, use the `'Shuffle'` name-value pair argument of `trainingOptions`.

ReLU Layer

Create a ReLU layer using `reluLayer`.

A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.

Convolutional and batch normalization layers are usually followed by a nonlinear activation function such as a rectified linear unit (ReLU), specified by a ReLU layer. A ReLU layer performs a threshold operation to each element, where any input value less than zero is set to zero, that is,

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

The ReLU layer does not change the size of its input.

There are other nonlinear activation layers that perform different operations and can improve the network accuracy for some applications. For a list of activation layers, see “Activation Layers” on page 1-23.

Cross Channel Normalization (Local Response Normalization) Layer

Create a cross channel normalization layer using `crossChannelNormalizationLayer`.

A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.

This layer performs a channel-wise local response normalization. It usually follows the ReLU activation layer. This layer replaces each element with a normalized value it obtains using the elements from a certain number of neighboring channels (elements in the normalization window). That is, for each element x in the input, `trainNetwork` computes a normalized value x' using

$$x' = \frac{x}{\left(K + \frac{\alpha * ss}{windowChannelSize}\right)^{\beta}},$$

where K, α, and β are the hyperparameters in the normalization, and ss is the sum of squares of the elements in the normalization window [2]. You must specify the size of the normalization window using the `windowChannelSize` argument of the `crossChannelNormalizationLayer` function. You can also specify the hyperparameters using the `Alpha`, `Beta`, and `K` name-value pair arguments.

The previous normalization formula is slightly different than what is presented in [2]. You can obtain the equivalent formula by multiplying the `alpha` value by the `windowChannelSize`.

Max and Average Pooling Layers

A 2-D max pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the maximum of each region. Create a max pooling layer using `maxPooling2dLayer`.

A 2-D average pooling layer performs downsampling by dividing the input into rectangular pooling regions, then computing the average values of each region. Create an average pooling layer using `averagePooling2dLayer`.

Pooling layers follow the convolutional layers for down-sampling, hence, reducing the number of connections to the following layers. They do not perform any learning themselves, but reduce the number of parameters to be learned in the following layers. They also help reduce overfitting.

A max pooling layer returns the maximum values of rectangular regions of its input. The size of the rectangular regions is determined by the `poolSize` argument of `maxPoolingLayer`. For example, if `poolSize` equals `[2,3]`, then the layer returns the maximum value in regions of height 2 and width 3.An average pooling layer outputs the average values of rectangular regions of its input. The size of the rectangular regions is determined by the `poolSize` argument of `averagePoolingLayer`. For example, if `poolSize` is [2,3], then the layer returns the average value of regions of height 2 and width 3.

Pooling layers scan through the input horizontally and vertically in step sizes you can specify using the `'Stride'` name-value pair argument. If the pool size is smaller than or equal to the stride, then the pooling regions do not overlap.

For nonoverlapping regions (*Pool Size* and *Stride* are equal), if the input to the pooling layer is *n*-by-*n*, and the pooling region size is *h*-by-*h*, then the pooling layer down-samples the regions by *h* [6]. That is, the output of a max or average pooling layer for one channel of a convolutional layer is *n*/*h*-by-*n*/*h*. For overlapping regions, the output of a pooling layer is (*Input Size* – *Pool Size* + 2**Padding*)/*Stride* + 1.

Dropout Layer

Create a dropout layer using `dropoutLayer`.

A dropout layer randomly sets input elements to zero with a given probability.

At training time, the layer randomly sets input elements to zero given by the dropout mask `rand(size(X))<Probability`, where `X` is the layer input and then scales the remaining elements by `1/(1-Probability)`. This operation effectively changes the underlying network architecture between iterations and helps prevent the network from overfitting [7], [2]. A higher number results in more elements being dropped during training. At prediction time, the output of the layer is equal to its input.

Similar to max or average pooling layers, no learning takes place in this layer.

Fully Connected Layer

Create a fully connected layer using `fullyConnectedLayer`.

A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.

The convolutional (and down-sampling) layers are followed by one or more fully connected layers.

As the name suggests, all neurons in a fully connected layer connect to all the neurons in the previous layer. This layer combines all of the features (local information) learned by the previous layers across the image to identify the larger patterns. For classification problems, the last fully connected layer combines the features to classify the images. This is the reason that the `outputSize` argument of the last fully connected layer of the network is equal to the number of classes of the data set. For regression problems, the output size must be equal to the number of response variables.

You can also adjust the learning rate and the regularization parameters for this layer using the related name-value pair arguments when creating the fully connected layer. If you choose not to adjust them, then `trainNetwork` uses the global training parameters defined by the `trainingOptions` function. For details on global and layer training options, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

A fully connected layer multiplies the input by a weight matrix W and then adds a bias vector b.

If the input to the layer is a sequence (for example, in an LSTM network), then the fully connected layer acts independently on each time step. For example, if the layer before the fully connected layer outputs an array X of size D-by-N-by-S, then the fully connected layer outputs an array Z of size `outputSize`-by-N-by-S. At time step t, the corresponding entry of Z is $WX_t + b$, where X_t denotes time step t of X.

Output Layers

Softmax and Classification Layers

A softmax layer applies a softmax function to the input. Create a softmax layer using `softmaxLayer`.

A classification layer computes the cross-entropy loss for classification and weighted classification tasks with mutually exclusive classes. Create a classification layer using `classificationLayer`.

For classification problems, a softmax layer and then a classification layer usually follow the final fully connected layer.

The output unit activation function is the softmax function:

$$y_r(x) = \frac{\exp(a_r(x))}{\sum_{j=1}^{k} \exp\left(a_j(x)\right)},$$

where $0 \le y_r \le 1$ and $\sum_{j=1}^{k} y_j = 1$.

The softmax function is the output unit activation function after the last fully connected layer for multi-class classification problems:

$$P(c_r|x,\theta) = \frac{P(x,\theta|c_r)P(c_r)}{\sum_{j=1}^{k} P\left(x,\theta|c_j\right)P\left(c_j\right)} = \frac{\exp(a_r(x,\theta))}{\sum_{j=1}^{k} \exp\left(a_j(x,\theta)\right)},$$

where $0 \le P(c_r|x,\theta) \le 1$ and $\sum_{j=1}^{k} P\left(c_j\middle|x,\theta\right) = 1$. Moreover, $a_r = \ln(P(x,\theta|c_r)P(c_r))$, $P(x,\theta|c_r)$ is the conditional probability of the sample given class r, and $P(c_r)$ is the class prior probability.

The softmax function is also known as the *normalized exponential* and can be considered the multi-class generalization of the logistic sigmoid function [8].

For typical classification networks, the classification layer usually follows a softmax layer. In the classification layer, `trainNetwork` takes the values from the softmax function and assigns each input to one of the K mutually exclusive classes using the cross entropy function for a 1-of-K coding scheme [8]:

$$\text{loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} w_i t_{ni} \ln y_{ni},$$

where N is the number of samples, K is the number of classes, w_i is the weight for class i, t_{ni} is the indicator that the nth sample belongs to the ith class, and y_{ni} is the output for sample n for class i, which in this case, is the value from the softmax function. In other words, y_{ni} is the probability that the network associates the nth input with class i.

Regression Layer

Create a regression layer using `regressionLayer`.

A regression layer computes the half-mean-squared-error loss for regression tasks. For typical regression problems, a regression layer must follow the final fully connected layer.

For a single observation, the mean-squared-error is given by:

$$\text{MSE} = \sum_{i=1}^{R} \frac{(t_i - y_i)^2}{R},$$

where R is the number of responses, t_i is the target output, and y_i is the network's prediction for response i.

For image and sequence-to-one regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses, not normalized by R:

$$\text{loss} = \frac{1}{2} \sum_{i=1}^{R} (t_i - y_i)^2 .$$

For image-to-image regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each pixel, not normalized by R:

$$\text{loss} = \frac{1}{2} \sum_{p=1}^{HWC} (t_p - y_p)^2,$$

where H, W, and C denote the height, width, and number of channels of the output respectively, and p indexes into each element (pixel) of t and y linearly.

For sequence-to-sequence regression networks, the loss function of the regression layer is the half-mean-squared-error of the predicted responses for each time step, not normalized by R:

$$\text{loss} = \frac{1}{2S} \sum_{i=1}^{S} \sum_{j=1}^{R} (t_{ij} - y_{ij})^2,$$

where S is the sequence length.

When training, the software calculates the mean loss over the observations in the mini-batch.

References

[1] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Massachusetts: The MIT Press, 2012.

[2] Krizhevsky, A., I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. Vol 25, 2012.

[3] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D., et al. ''Handwritten Digit Recognition with a Back-propagation Network.'' In *Advances of Neural Information Processing Systems*, 1990.

[4] LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. ''Gradient-based Learning Applied to Document Recognition.'' *Proceedings of the IEEE*. Vol 86, pp. 2278–2324, 1998.

[5] Nair, V. and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines." In Proc. 27th International Conference on Machine Learning, 2010.

[6] Nagi, J., F. Ducatelle, G. A. Di Caro, D. Ciresan, U. Meier, A. Giusti, F. Nagi, J. Schmidhuber, L. M. Gambardella. ''Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition''. *IEEE International Conference on Signal and Image Processing Applications (ICSIPA2011)*, 2011.

[7] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*. Vol. 15, pp. 1929-1958, 2014.

[8] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

[9] Ioffe, Sergey, and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” Preprint, submitted March 2, 2015. https://arxiv.org/abs/1502.03167.

See Also

`convolution2dLayer` | `batchNormalizationLayer` | `dropoutLayer` | `averagePooling2dLayer` | `maxPooling2dLayer` | `classificationLayer` | `regressionLayer` | `softmaxLayer` | `crossChannelNormalizationLayer` | `fullyConnectedLayer` | `reluLayer` | `leakyReluLayer` | `clippedReluLayer` | `imageInputLayer` | `trainingOptions` | `trainNetwork`

More About

- “List of Deep Learning Layers” on page 1-21
- “Learn About Convolutional Neural Networks” on page 1-17
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Resume Training from Checkpoint Network” on page 5-95
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Set Up Parameters and Train Convolutional Neural Network

In this section...

After you define the layers of your neural network as described in “Specify Layers of Convolutional Neural Network” on page 1-31, the next step is to set up the training options for the network. Use the `trainingOptions` function to define the global training parameters. To train a network, use the object returned by `trainingOptions` as an input argument to the `trainNetwork` function. For example:

```
options = trainingOptions('adam');
trainedNet = trainNetwork(data,layers,options);
```

Layers with learnable parameters also have options for adjusting the learning parameters. For more information, see “Set Up Parameters in Convolutional and Fully Connected Layers” on page 1-44.

Specify Solver and Maximum Number of Epochs

`trainNetwork` can use different variants of stochastic gradient descent to train the network. Specify the optimization algorithm by using the `solverName` argument of `trainingOptions`. To minimize the loss, these algorithms update the network parameters by taking small steps in the direction of the negative gradient of the loss function.

The `'adam'` (derived from *adaptive moment estimation*) solver is often a good optimizer to try first. You can also try the `'rmsprop'` (root mean square propagation) and `'sgdm'` (stochastic gradient descent with momentum) optimizers and see if this improves training. Different solvers work better for different problems. For more information about the different solvers, see “Stochastic Gradient Descent”.

The solvers update the parameters using a subset of the data each step. This subset is called a *mini-batch*. You can specify the size of the mini-batch by using the `'MiniBatchSize'` name-value pair argument of `trainingOptions`. Each parameter update is called an *iteration*. A full pass through the entire data set is called an *epoch*. You can specify the maximum number of epochs to train for by using the `'MaxEpochs'` name-value pair argument of `trainingOptions`. The default value is 30, but you can choose a smaller number of epochs for small networks or for fine-tuning and transfer learning, where most of the learning is already done.

By default, the software shuffles the data once before training. You can change this setting by using the `'Shuffle'` name-value pair argument.

Specify and Modify Learning Rate

You can specify the global learning rate by using the `'InitialLearnRate'` name-value pair argument of `trainingOptions`. By default, `trainNetwork` uses this value throughout the entire

training process. You can choose to modify the learning rate every certain number of epochs by multiplying the learning rate with a factor. Instead of using a small, fixed learning rate throughout the training process, you can choose a larger learning rate in the beginning of training and gradually reduce this value during optimization. Doing so can shorten the training time, while enabling smaller steps towards the minimum of the loss as training progresses.

Tip If the mini-batch loss during training ever becomes `NaN`, then the learning rate is likely too high. Try reducing the learning rate, for example by a factor of 3, and restarting network training.

To gradually reduce the learning rate, use the `'LearnRateSchedule','piecewise'` name-value pair argument. Once you choose this option, `trainNetwork` multiplies the initial learning rate by a factor of 0.1 every 10 epochs. You can specify the factor by which to reduce the initial learning rate and the number of epochs by using the `'LearnRateDropFactor'` and `'LearnRateDropPeriod'` name-value pair arguments, respectively.

Specify Validation Data

To perform network validation during training, specify validation data using the `'ValidationData'` name-value pair argument of `trainingOptions`. By default, `trainNetwork` validates the network every 50 iterations by predicting the response of the validation data and calculating the validation loss and accuracy (root mean squared error for regression networks). You can change the validation frequency using the `'ValidationFrequency'` name-value pair argument. If your network has layers that behave differently during prediction than during training (for example, dropout layers), then the validation accuracy can be higher than the training (mini-batch) accuracy. You can also use the validation data to stop training automatically when the validation loss stops decreasing. To turn on automatic validation stopping, use the `'ValidationPatience'` name-value pair argument.

Performing validation at regular intervals during training helps you to determine if your network is overfitting to the training data. A common problem is that the network simply "memorizes" the training data, rather than learning general features that enable the network to make accurate predictions for new data. To check if your network is overfitting, compare the training loss and accuracy to the corresponding validation metrics. If the training loss is significantly lower than the validation loss, or the training accuracy is significantly higher than the validation accuracy, then your network is overfitting.

To reduce overfitting, you can try adding data augmentation. Use an `augmentedImageDatastore` to perform random transformations on your input images. This helps to prevent the network from memorizing the exact position and orientation of objects. You can also try increasing the L_2 regularization using the `'L2Regularization'` name-value pair argument, using batch normalization layers after convolutional layers, and adding dropout layers.

Select Hardware Resource

If a GPU is available, then `trainNetwork` uses it for training, by default. Otherwise, `trainNetwork` uses a CPU. Alternatively, you can specify the execution environment you want using the `'ExecutionEnvironment'` name-value pair argument. You can specify a single CPU (`'cpu'`), a single GPU (`'gpu'`), multiple GPUs (`'multi-gpu'`), or a local parallel pool or compute cluster (`'parallel'`). All options other than `'cpu'` require Parallel Computing Toolbox. Training on a GPU requires a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

Save Checkpoint Networks and Resume Training

Deep Learning Toolbox enables you to save networks as .mat files after each epoch during training. This periodic saving is especially useful when you have a large network or a large data set, and training takes a long time. If the training is interrupted for some reason, you can resume training from the last saved checkpoint network. If you want `trainNetwork` to save checkpoint networks, then you must specify the name of the path by using the `'CheckpointPath'` name-value pair argument of `trainingOptions`. If the path that you specify does not exist, then `trainingOptions` returns an error.

`trainNetwork` automatically assigns unique names to checkpoint network files. In the example name, `net_checkpoint__351__2018_04_12__18_09_52.mat`, 351 is the iteration number, `2018_04_12` is the date, and `18_09_52` is the time at which `trainNetwork` saves the network. You can load a checkpoint network file by double-clicking it or using the load command at the command line. For example:

```
load net_checkpoint__351__2018_04_12__18_09_52.mat
```

You can then resume training by using the layers of the network as an input argument to `trainNetwork`. For example:

```
trainNetwork(XTrain,YTrain,net.Layers,options)
```

You must manually specify the training options and the input data, because the checkpoint network does not contain this information. For an example, see “Resume Training from Checkpoint Network” on page 5-95.

Set Up Parameters in Convolutional and Fully Connected Layers

You can set the learning parameters to be different from the global values specified by `trainingOptions` in layers with learnable parameters, such as convolutional and fully connected layers. For example, to adjust the learning rate for the biases or weights, you can specify a value for the `BiasLearnRateFactor` or `WeightLearnRateFactor` properties of the layer, respectively. The `trainNetwork` function multiplies the learning rate that you specify by using `trainingOptions` with these factors. Similarly, you can also specify the L_2 regularization factors for the weights and biases in these layers by specifying the `BiasL2Factor` and `WeightL2Factor` properties, respectively. `trainNetwork` then multiplies the L_2 regularization factors that you specify by using `trainingOptions` with these factors.

Initialize Weights in Convolutional and Fully Connected Layers

The layer weights are learnable parameters. You can specify the initial value for the weights directly using the `Weights` property of the layer. When you train a network, if the `Weights` property of the layer is nonempty, then `trainNetwork` uses the `Weights` property as the initial value. If the `Weights` property is empty, then `trainNetwork` uses the initializer specified by the `WeightsInitializer` property of the layer.

Train Your Network

After you specify the layers of your network and the training parameters, you can train the network using the training data. The data, layers, and training options are all input arguments of the `trainNetwork` function, as in this example.

```
layers = [imageInputLayer([28 28 1])
          convolution2dLayer(5,20)
          reluLayer
          maxPooling2dLayer(2,'Stride',2)
          fullyConnectedLayer(10)
          softmaxLayer
          classificationLayer];
options = trainingOptions('adam');
convnet = trainNetwork(data,layers,options);
```

Training data can be an array, a table, or an `ImageDatastore` object. For more information, see the `trainNetwork` function reference page.

See Also

`trainingOptions` | `trainNetwork` | `Convolution2dLayer` | `FullyConnectedLayer`

More About

- “Learn About Convolutional Neural Networks” on page 1-17
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Resume Training from Checkpoint Network” on page 5-95

Train Network with Numeric Features

This example shows how to create and train a simple neural network for deep learning feature data classification.

If you have a data set of numeric features (for example a collection of numeric data without spatial or time dimensions), then you can train a deep learning network using a feature input layer. For an example showing how to train a network for image classification, see “Create Simple Deep Learning Network for Classification” on page 3-47.

This example shows how to train a network to classify the gear tooth condition of a transmission system given a mixture of numeric sensor readings, statistics, and categorical labels.

Load Data

Load the transmission casing dataset for training. The data set consists of 208 synthetic readings of a transmission system consisting of 18 numeric readings and three categorical labels:

1. `SigMean` — Vibration signal mean
2. `SigMedian` — Vibration signal median
3. `SigRMS` — Vibration signal RMS
4. `SigVar` — Vibration signal variance
5. `SigPeak` — Vibration signal peak
6. `SigPeak2Peak` — Vibration signal peak to peak
7. `SigSkewness` — Vibration signal skewness
8. `SigKurtosis` — Vibration signal kurtosis
9. `SigCrestFactor` — Vibration signal crest factor
10. `SigMAD` — Vibration signal MAD
11. `SigRangeCumSum` — Vibration signal range cumulative sum
12. `SigCorrDimension` — Vibration signal correlation dimension
13. `SigApproxEntropy` — Vibration signal approximate entropy
14. `SigLyapExponent` — Vibration signal Lyap exponent
15. `PeakFreq` — Peak frequency.
16. `HighFreqPower` — High frequency power
17. `EnvPower` — Environment power
18. `PeakSpecKurtosis` — Peak frequency of spectral kurtosis
19. `SensorCondition` — Condition of sensor, specified as "Sensor Drift" or "No Sensor Drift"
20. `ShaftCondition` — Condition of shaft, specified as "Shaft Wear" or "No Shaft Wear"
21. `GearToothCondition` — Condition of gear teeth, specified as "Tooth Fault" or "No Tooth Fault"

Read the transmission casing data from the CSV file `"transmissionCasingData.csv"`.

```
filename = "transmissionCasingData.csv";
tbl = readtable(filename,'TextType','String');
```

Convert the labels for prediction to categorical using the `convertvars` function.

```
labelName = "GearToothCondition";
tbl = convertvars(tbl,labelName,'categorical');
```

View the first few rows of the table.

```
head(tbl)
```

```
ans=8×21 table
    SigMean     SigMedian    SigRMS    SigVar     SigPeak    SigPeak2Peak    SigSkewness    SigKu
    ________    _________    ______    _______    _______    ____________    ___________    _____

    -0.94876     -0.9722     1.3726    0.98387    0.81571       3.6314        -0.041525       2.2
    -0.97537    -0.98958     1.3937    0.99105    0.81571       3.6314        -0.023777       2.2
      1.0502      1.0267     1.4449    0.98491     2.8157       3.6314         -0.04162       2.2
      1.0227      1.0045     1.4288    0.99553     2.8157       3.6314        -0.016356       2.2
      1.0123      1.0024     1.4202    0.99233     2.8157       3.6314        -0.014701       2.2
      1.0275      1.0102     1.4338     1.0001     2.8157       3.6314         -0.02659       2.2
      1.0464      1.0275     1.4477     1.0011     2.8157       3.6314        -0.042849       2.2
      1.0459      1.0257     1.4402    0.98047     2.8157       3.6314        -0.035405       2.2
```

To train a network using categorical features, you must first convert the categorical features to numeric. First, convert the categorical predictors to categorical using the `convertvars` function by specifying a string array containing the names of all the categorical input variables. In this data set, there are two categorical features with names `"SensorCondition"` and `"ShaftCondition"`.

```
categoricalInputNames = ["SensorCondition" "ShaftCondition"];
tbl = convertvars(tbl,categoricalInputNames,'categorical');
```

Loop over the categorical input variables. For each variable:

- Convert the categorical values to one-hot encoded vectors using the `onehotencode` function.
- Add the one-hot vectors to the table using the `addvars` function. Specify to insert the vectors after the column containing the corresponding categorical data.
- Remove the corresponding column containing the categorical data.

```
for i = 1:numel(categoricalInputNames)
    name = categoricalInputNames(i);
    oh = onehotencode(tbl(:,name));
    tbl = addvars(tbl,oh,'After',name);
    tbl(:,name) = [];
end
```

Split the vectors into separate columns using the `splitvars` function.

```
tbl = splitvars(tbl);
```

View the first few rows of the table. Notice that the categorical predictors have been split into multiple columns with the categorical values as the variable names.

```
head(tbl)
```

```
ans=8×23 table
    SigMean     SigMedian    SigRMS    SigVar     SigPeak    SigPeak2Peak    SigSkewness    SigKu
    ________    _________    ______    _______    _______    ____________    ___________    _____

    -0.94876     -0.9722     1.3726    0.98387    0.81571       3.6314        -0.041525       2.2
    -0.97537    -0.98958     1.3937    0.99105    0.81571       3.6314        -0.023777       2.2
```

```
      1.0502      1.0267      1.4449     0.98491      2.8157       3.6314         -0.04162       2.2
      1.0227      1.0045      1.4288     0.99553      2.8157       3.6314        -0.016356       2.2
      1.0123      1.0024      1.4202     0.99233      2.8157       3.6314        -0.014701       2.2
      1.0275      1.0102      1.4338      1.0001      2.8157       3.6314         -0.02659       2.2
      1.0464      1.0275      1.4477      1.0011      2.8157       3.6314        -0.042849       2.2
      1.0459      1.0257      1.4402     0.98047      2.8157       3.6314        -0.035405       2.2
```

View the class names of the data set.

```
classNames = categories(tbl{:,labelName})
```

```
classNames = 2x1 cell
    {'No Tooth Fault'}
    {'Tooth Fault'   }
```

Split Data Set into Training and Validation Sets

Partition the data set into training, validation, and test partitions. Set aside 15% of the data for validation, and 15% for testing.

View the number of observations in the dataset.

```
numObservations = size(tbl,1)
```

```
numObservations = 208
```

Determine the number of observations for each partition.

```
numObservationsTrain = floor(0.7*numObservations)
```

```
numObservationsTrain = 145
```

```
numObservationsValidation = floor(0.15*numObservations)
```

```
numObservationsValidation = 31
```

```
numObservationsTest = numObservations - numObservationsTrain - numObservationsValidation
```

```
numObservationsTest = 32
```

Create an array of random indices corresponding to the observations and partition it using the partition sizes.

```
idx = randperm(numObservations);
idxTrain = idx(1:numObservationsTrain);
idxValidation = idx(numObservationsTrain+1:numObservationsTrain+numObservationsValidation);
idxTest = idx(numObservationsTrain+numObservationsValidation+1:end);
```

Partition the table of data into training, validation, and testing partitions using the indices.

```
tblTrain = tbl(idxTrain,:);
tblValidation = tbl(idxValidation,:);
tblTest = tbl(idxTest,:);
```

Define Network Architecture

Define the network for classification.

Define a network with a feature input layer and specify the number of features. Also, configure the input layer to normalize the data using Z-score normalization. Next, include a fully connected layer with output size 50 followed by a batch normalization layer and a ReLU layer. For classification, specify another fully connected layer with output size corresponding to the number of classes, followed by a softmax layer and a classification layer.

```
numFeatures = size(tbl,2) - 1;
numClasses = numel(classNames);

layers = [
    featureInputLayer(numFeatures,'Normalization', 'zscore')
    fullyConnectedLayer(50)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify Training Options

Specify the training options.

- Train the network using Adam.
- Train using mini-batches of size 16.
- Shuffle the data every epoch.
- Monitor the network accuracy during training by specifying validation data.
- Display the training progress in a plot and suppress the verbose command window output.

The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights.

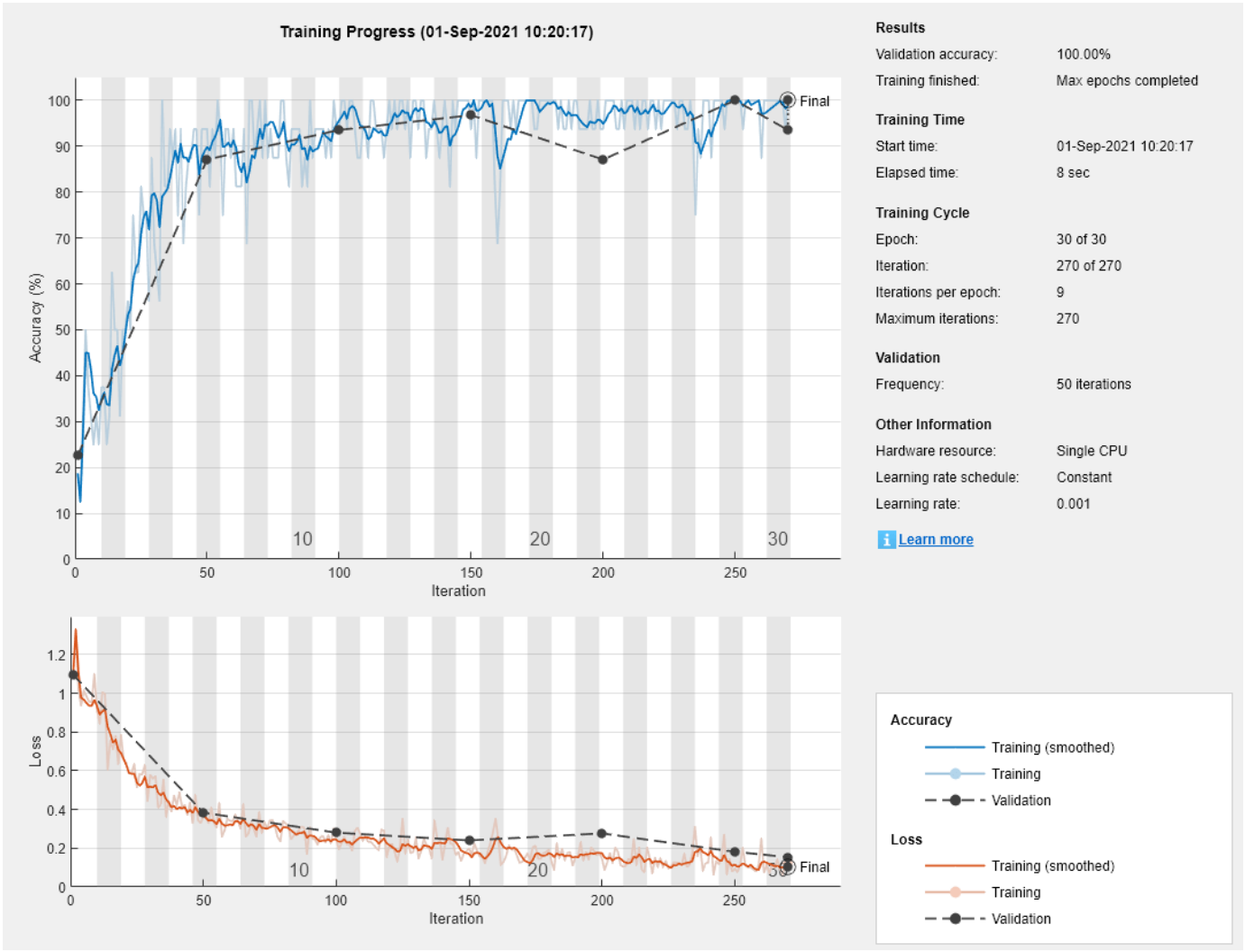
```
miniBatchSize = 16;

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'ValidationData',tblValidation, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train Network

Train the network using the architecture defined by `layers`, the training data, and the training options. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. For more information on the training progress plot, see “Monitor Deep Learning Training Progress” on page 5-115.

```
net = trainNetwork(tblTrain,labelName,layers,options);
```



Test Network

Predict the labels of the test data using the trained network and calculate the accuracy. Specify the same mini-batch size used for training.

```
YPred = classify(net,tblTest(:,1:end-1),'MiniBatchSize',miniBatchSize);
```

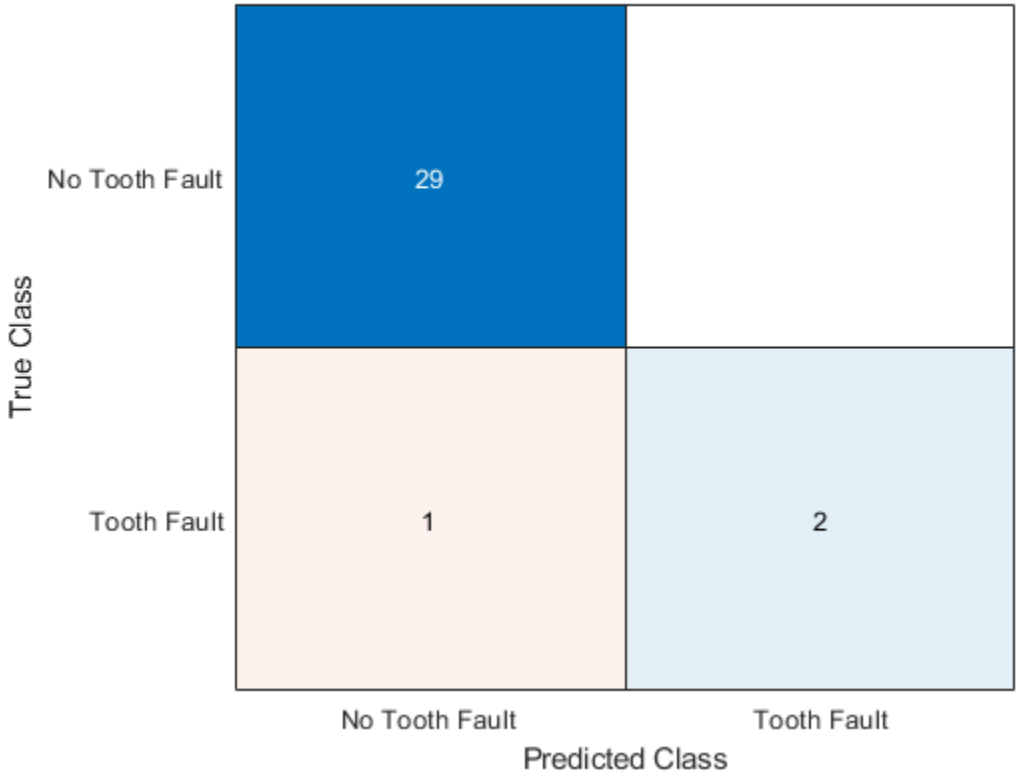
Calculate the classification accuracy. The accuracy is the proportion of the labels that the network predicts correctly.

```
YTest = tblTest{:,labelName};
accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9688
```

View the results in a confusion matrix.

```
figure
confusionchart(YTest,YPred)
```



See Also

`trainNetwork` | `trainingOptions` | `fullyConnectedLayer` | **Deep Network Designer** | `featureInputLayer`

Related Examples

- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Train Convolutional Neural Network for Regression” on page 3-53

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “List of Deep Learning Layers” on page 1-21

Train Network on Image and Feature Data

This example shows how to train a network that classifies handwritten digits using both image and feature input data.

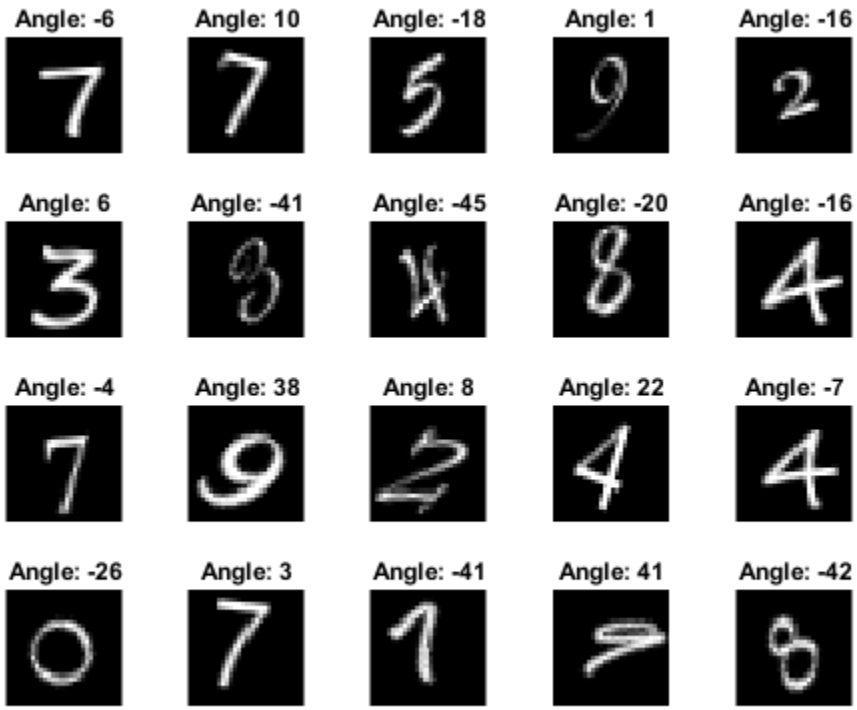
Load Training Data

Load the digits images `XTrain`, labels `YTrain`, and clockwise rotation angles `anglesTrain`. Create an `arrayDatastore` object for the images, labels, and angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and the height, width, number of channels, and number of nondiscrete responses.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsAnglesTrain = arrayDatastore(anglesTrain);
dsYTrain = arrayDatastore(YTrain);

dsTrain = combine(dsXTrain,dsAnglesTrain,dsYTrain);

classes = categories(YTrain);
[h,w,c,numObservations] = size(XTrain);
```

Display 20 random training images.

```
numTrainImages = numel(YTrain);
figure
idx = randperm(numTrainImages,20);
for i = 1:numel(idx)
    subplot(4,5,i)
    imshow(XTrain(:,:,:,idx(i)))
    title("Angle: " + anglesTrain(idx(i)))
end
```



Define Network

Define the size of the input image, the number of features of each observation, the number of classes, and the size and number of filters of the convolution layer.

```
imageInputSize = [h w c];
numFeatures = 1;
numClasses = numel(classes);
filterSize = 5;
numFilters = 16;
```

To create a network with two input layers, you must define the network in two parts and join them, for example, by using a concatenation layer.

Define the first part of the network. Define the image classification layers and include a concatenation layer before the last fully connected layer.

```
layers = [
    imageInputLayer(imageInputSize,'Normalization','none','Name','images')
    convolution2dLayer(filterSize,numFilters,'Name','conv')
    reluLayer('Name','relu')
    fullyConnectedLayer(50,'Name','fc1')
    concatenationLayer(1,2,'Name','concat')
    fullyConnectedLayer(numClasses,'Name','fc2')
    softmaxLayer('Name','softmax')];
```

Convert the layers to a layer graph.

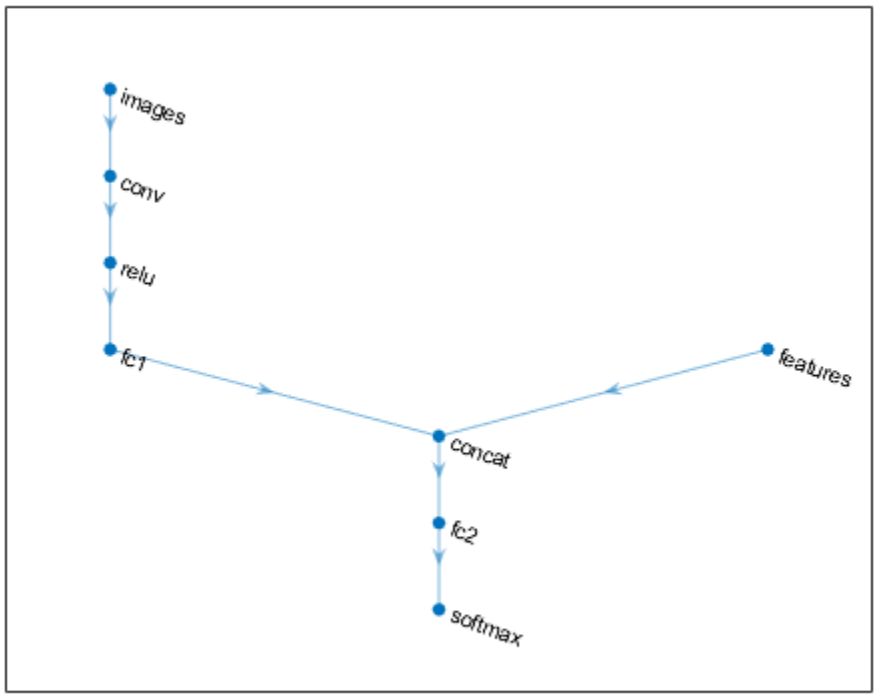
```
lgraph = layerGraph(layers);
```

For the second part of the network, add a feature input layer and connect it to the second input of the concatenation layer.

```
featInput = featureInputLayer(numFeatures,'Name','features');
lgraph = addLayers(lgraph, featInput);
lgraph = connectLayers(lgraph, 'features', 'concat/in2');
```

Visualize the network.

```
figure
plot(lgraph)
```



Create a `dlnetwork` object.

```
dlnet = dlnetwork(lgraph);
```

When you use the functions `predict` and `forward` on a `dlnetwork` object, the input arguments must match the order given by the `InputNames` property of the `dlnetwork` object. Inspect the name and order of the input layers.

```
dlnet.InputNames
```

```
ans = 1×2 cell
    {'images'}    {'features'}
```

Define Model Gradients Function

The `modelGradients` function, listed in the Model Gradients Function on page 1-0 section of the example, takes as input a `dlnetwork` object `dlnet`, a mini-batch of input image data `dlX1`, a mini-batch of input feature data `dlX2`, and the corresponding labels `dlY`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss.

Specify Training Options

Train with a mini-batch size of 128 for 15 epochs.

```
numEpochs = 15;
miniBatchSize = 128;
```

Specify the options for SGDM optimization. Specify an initial learning rate of 0.01 with a decay of 0.01, and momentum of 0.9.

```
learnRate = 0.01;
decay = 0.01;
momentum = 0.9;
```

To monitor the training progress, you can plot the training loss after each iteration. Create the variable `plots` that contains `"training-progress"`. If you do not want to plot the training progress, then set this value to `"none"`.

```
plots = "training-progress";
```

Train Model

Train the model using a custom training loop. Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Use `minibatchqueue` to process and manage mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessData` (defined at the end of this example) to one-hot encode the class labels.
- By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Format the images with the dimension labels `'SSCB'` (spatial, spatial, channel, batch), and the angles with the dimension labels `'CB'` (channel, batch). Do not add a format to the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','CB',''});
```

For each epoch, shuffle the data and loop over mini-batches of data. At the end of each epoch, display the training progress. For each mini-batch:

- Evaluate the model gradients, state, and loss using `dlfeval` and the `modelGradients` function and update the network state.

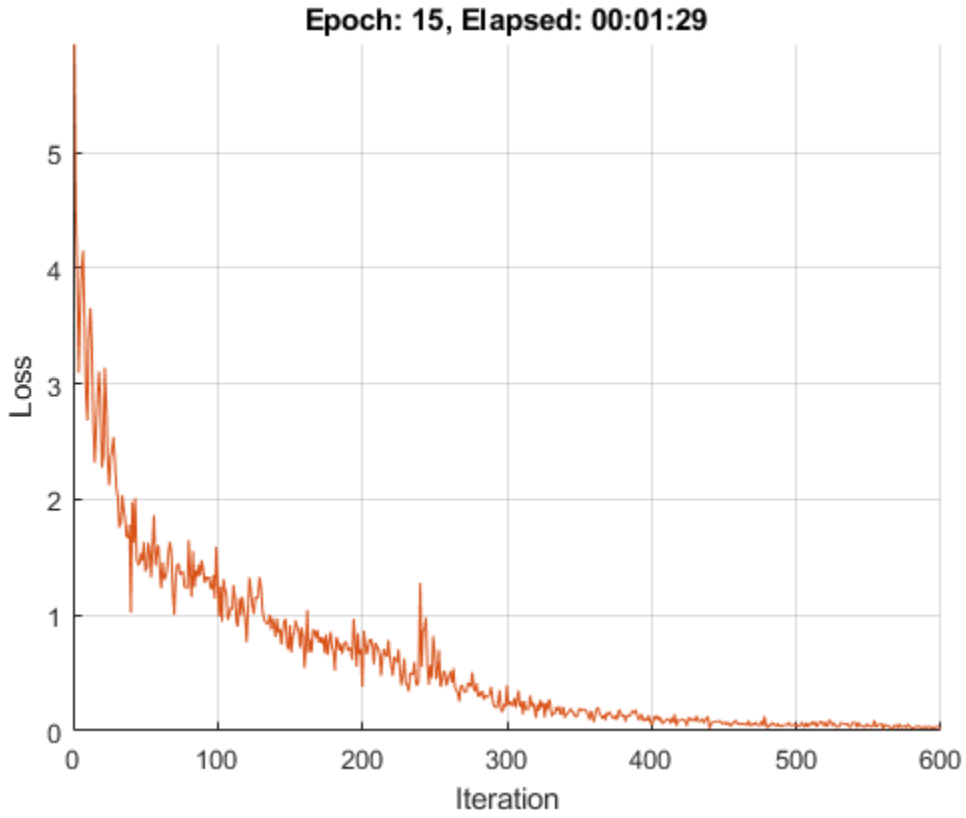
- Update the network parameters using the `sgdmupdate` function.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)

        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX1,dlX2,dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function and update the network state.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX1,dlX2,dlY);
        dlnet.State = state;

        % Update the network parameters using the SGDM optimizer.
        [dlnet, velocity] = sgdmupdate(dlnet, gradients, velocity, learnRate, momentum);

        if plots == "training-progress"
            % Display the training progress.
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            %completionPercentage = round(iteration/numIterations*100,0);
            title("Epoch: " + epoch + ", Elapsed: " + string(D));
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels. Test the classification accuracy of the model by comparing the predictions on a test set with the true labels and angles. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
[XTest,YTest,anglesTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsAnglesTest = arrayDatastore(anglesTest);
dsYTest = arrayDatastore(YTest);

dsTest = combine(dsXTest,dsAnglesTest,dsYTest);

mbqTest = minibatchqueue(dsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','CB',''});
```

Loop over the mini-batches and classify the images using `modelPredictions` function, listed at the end of the example.

```
[predictions,predCorr] = modelPredictions(dlnet,mbqTest,classes);
```

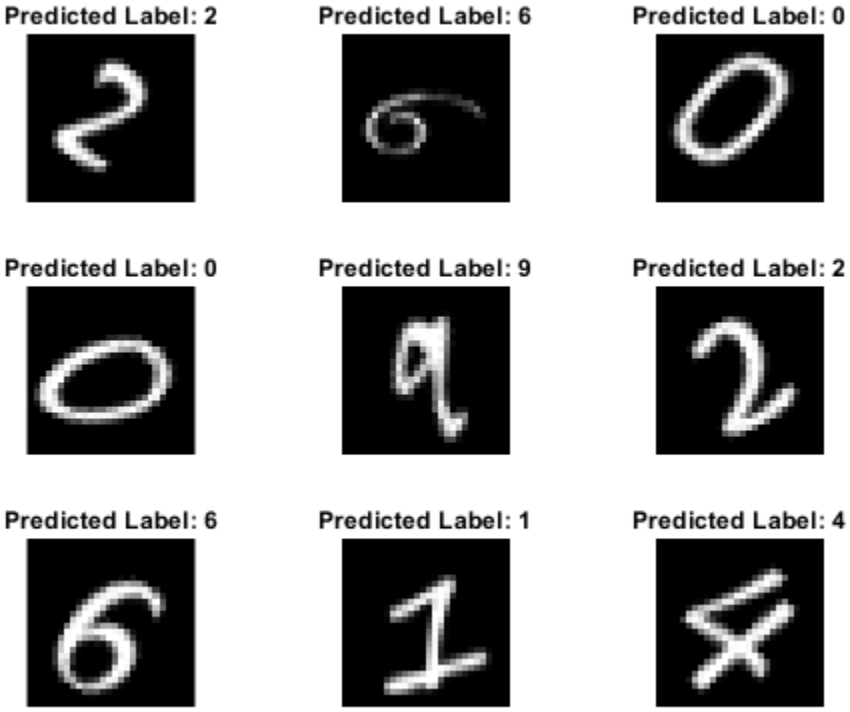
Evaluate the classification accuracy.

```
accuracy = mean(predCorr)
```

```
accuracy = 0.9818
```

View some of the images with their predictions.

```
idx = randperm(size(XTest,4),9);
figure
for i = 1:9
    subplot(3,3,i)
    I = XTest(:,:,:,idx(i));
    imshow(I)

    label = string(predictions(idx(i)));
    title("Predicted Label: " + label)
end
```



Model Gradients Function

The `modelGradients` function takes as input a `dlnetwork` object `dlnet`, a mini-batch of input image data `dlX1`, a mini-batch of input feature data `dlX2`, and corresponding labels `Y`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

When you use the `forward` function on a `dlnetwork` object, the input arguments must match the order given by the `InputNames` property of the `dlnetwork` object.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX1,dlX2,Y)
```

```
[dlYPred,state] = forward(dlnet,dlX1,dlX2);

loss = crossentropy(dlYPred,Y);
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score and then compares the prediction with the true label. The function returns the predictions and a vector of ones and zeros that represents correct and incorrect predictions.

```
function [classesPredictions,classCorr] = modelPredictions(dlnet,mbq,classes)

    classesPredictions = [];
    classCorr = [];

    while hasdata(mbq)
        [dlX1,dlX2,dlY] = next(mbq);

        % Make predictions using the model function.
        dlYPred = predict(dlnet,dlX1,dlX2);

        % Determine predicted classes.
        YPredBatch = onehotdecode(dlYPred,classes,1);
        classesPredictions = [classesPredictions YPredBatch];

        % Compare predicted and true classes.
        Y = onehotdecode(dlY,classes,1);
        classCorr = [classCorr YPredBatch == Y];

    end

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label and angle data from the incoming cell arrays and concatenate along the second dimension into a categorical array and a numeric array, respectively.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,angle,Y] = preprocessMiniBatch(XCell,angleCell,YCell)

    % Extract image data from cell and concatenate.
    X = cat(4,XCell{:});
    % Extract angle data from cell and concatenate.
```

```
    angle = cat(2,angleCell{:});
    % Extract label data from cell and concatenate.
    Y = cat(2,YCell{:});

    % One-hot encode labels.
    Y = onehotencode(Y,1);

end
```

See Also

`dlnetwork` | `dlfeval` | `dlarray` | `fullyConnectedLayer` | **Deep Network Designer** | `featureInputLayer` | `minibatchqueue` | `onehotencode` | `onehotdecode`

Related Examples

- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Train Convolutional Neural Network for Regression” on page 3-53

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “List of Deep Learning Layers” on page 1-21

Compare Activation Layers

This example shows how to compare the accuracy of training networks with ReLU, leaky ReLU, ELU, and swish activation layers.

Training deep learning neural networks requires using nonlinear activation functions such as the ReLU and swish operations. Some activation layers can yield better training performance at the cost of extra computation time. When training a neural network, you can try using different activation layers to see if training improves.

This example shows how to compare the validation accuracy of training a SqueezeNet neural network when you use ReLU, leaky ReLU, ELU, or swish activation layers given a validation set of images.

Load Data

Download the Flowers data set.

```
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'flower_dataset.tgz');

dataFolder = fullfile(downloadFolder,'flower_photos');
if ~exist(dataFolder,'dir')
    fprintf("Downloading Flowers data set (218 MB)... ")
    websave(filename,url);
    untar(filename,downloadFolder)
    fprintf("Done.\n")
end
```

Prepare Data for Training

Load the data as an image datastore using the `imageDatastore` function and specify the folder containing the image data.

```
imds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

View the number of classes of the training data.

```
numClasses = numel(categories(imds.Labels))
```

```
numClasses = 5
```

Divide the datastore so that each category in the training set has 80% of the images and the validation set has the remaining images from each label.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.80,'randomize');
```

Specify augmentation options and create an augmented image datastore containing the training images.

- Randomly reflect the images on the horizontal axis.
- Randomly scale the images by up to 20%.
- Randomly rotate the images by up to 45 degrees.

- Randomly translate the images by up to 3 pixels.
- Resize the images to the input size of the network (227-by-227).

```
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandScale',[0.8 1.2], ...
    'RandRotation',[-45,45], ...
    'RandXTranslation',[-3 3], ...
    'RandYTranslation',[-3 3]);

augimdsTrain = augmentedImageDatastore([227 227],imdsTrain,'DataAugmentation',imageAugmenter);
```

Create an augmented image datastore for the validation data that resizes the images to the input size of the network. Do not apply any other image transformations to the validation data.

```
augimdsValidation = augmentedImageDatastore([227 227],imdsValidation);
```

Create Custom Plotting Function

When training multiple networks, to monitor the validation accuracy for each network on the same axis, you can use the `OutputFcn` training option and specify a function that updates a plot with the provided training information.

Create a function that takes the information structure provided by the training process and updates an animated line plot. The `updatePlot` function, listed in the Plotting Function on page 1-0 section of the example, takes the information structure as input and updates the specified animated line.

Specify Training Options

Specify the training options:

- Train using a mini-batch size of 128 for 60 epochs.
- Shuffle the data each epoch.
- Validate the neural network once per epoch using the held-out validation set.

```
miniBatchSize = 128;
numObservationsTrain = numel(imdsTrain.Files);
numIterationsPerEpoch = floor(numObservationsTrain / miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',60, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Verbose',false);
```

Train Neural Networks

For each of the activation layer types—ReLU, leaky ReLU, ELU, and swish—train a SqueezeNet network.

Specify the types of activation layers.

```
activationLayerTypes = ["relu" "leaky-relu" "elu" "swish"];
```

Initialize the customized training progress plot by creating animated lines with colors specified by `colororder` function.

```
figure

colors = colororder;

for i = 1:numel(activationLayerTypes)
    line(i) = animatedline('Color',colors(i,:));
end

ylim([0 100])

legend(activationLayerTypes,'Location','southeast');

xlabel("Iteration")
ylabel("Accuracy")
title("Validation Accuracy")
grid on
```

Loop over each of the activation layer types and train the neural network. For each activation layer type:

- Create a function handle `activationLayer` that creates the activation layer.
- Create a new SqueezeNet network without weights and replace the activation layers (the ReLU layers) with layers of the activation layer type using the function handle `activationLayer`.
- Replace the final convolution layer of the neural network with one specifying the number of classes of the input data.
- Update the validation accuracy plot by setting the `OutputFcn` property of the training options object to a function handle representing the `updatePlot` function with the animated line corresponding to the activation layer type.
- Train and time the network using the `trainNetwork` function.

```
for i = 1:numel(activationLayerTypes)
    activationLayerType = activationLayerTypes(i);

    % Determine activation layer type.
    switch activationLayerType
        case "relu"
            activationLayer = @reluLayer;
        case "leaky-relu"
            activationLayer = @leakyReluLayer;
        case "elu"
            activationLayer = @eluLayer;
        case "swish"
            activationLayer = @swishLayer;
    end

    % Create SqueezeNet layer graph.
    lgraph = squeezenet('Weights','none');

    % Replace activation layers.
    if activationLayerType ~= "relu"
        layers = lgraph.Layers;
        for j = 1:numel(layers)
            if isa(layers(j),'nnet.cnn.layer.ReLULayer')
```

```
                layerName = layers(j).Name;
                layer = activationLayer('Name',activationLayerType+"_new_"+j);
                lgraph = replaceLayer(lgraph,layerName,layer);
            end
        end
    end

    % Specify number of classes in final convolution layer.
    layer = convolution2dLayer([1 1],numClasses,'Name','conv10');
    lgraph = replaceLayer(lgraph,'conv10',layer);

    % Specify custom plot function.
    options.OutputFcn = @(info) updatePlot(info,line(i));

    % Train the network.
    start = tic;
    [net{i},info{i}] = trainNetwork(augimdsTrain,lgraph,options);
    elapsed(i) = toc(start);
end
```

Visualize the training times in a bar chart.

```
figure
bar(categorical(activationLayerTypes),elapsed)
title("Training Time")
ylabel("Time (seconds)")
```

In this case, using the different activation layers yields similar final validation accuracies, with the leaky ReLU and swish layers having slightly higher values. Using swish activation layers enables convergence in fewer iterations. When compared to the other activation layers, using ELU layers makes the validation accuracy converge in more iterations and requires more computation time.

Plotting Function

The `updatePlot` function takes as input the information structure `info` and updates the validation plot specified by the animated line `line`.

```
function updatePlot(info,line)

if ~isempty(info.ValidationAccuracy)
    addpoints(line,info.Iteration,info.ValidationAccuracy);
    drawnow limitrate
end

end
```

See Also

`trainingOptions` | `trainNetwork` | `reluLayer` | `leakyReluLayer` | `swishLayer`

More About

- “Deep Learning in MATLAB” on page 1-2

- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Deep Learning Tips and Tricks

This page describes various training options and techniques for improving the accuracy of deep learning networks.

Choose Network Architecture

The appropriate network architecture depends on the task and the data available. Consider these suggestions when deciding which architecture to use and whether to use a pretrained network or to train from scratch.

Data	Description of Task	Learn More
Images	Classification of natural images	Try different pretrained networks. For a list of pretrained deep learning networks, see “Pretrained Deep Neural Networks” on page 1-8. To learn how to interactively prepare a network for transfer learning using Deep Network Designer, see “Transfer Learning with Deep Network Designer” on page 2-2.
	Regression of natural images	Try different pretrained networks. For an example showing how to convert a pretrained classification network into a regression network, see “Convert Classification Network into Regression Network” on page 3-70.
	Classification and regression of non-natural images (for example, tiny images and spectrograms	For an example showing how to classify tiny images, see “Train Residual Network for Image Classification” on page 3-13. For an example showing how to classify spectrograms, see “Speech Command Recognition Using Deep Learning” on page 4-23.

Data	Description of Task	Learn More
	Semantic segmentation	Computer Vision Toolbox™ provides tools to create deep learning networks for semantic segmentation. For more information, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).
Sequences, time series, and signals	Sequence-to-label classification	For an example, see “Sequence Classification Using Deep Learning” on page 4-2.
	Sequence-to-sequence classification and regression	To learn more, see “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42 and “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47.
	Time series forecasting	For an example, see “Time Series Forecasting Using Deep Learning” on page 4-15.
Text	Classification and regression	Text Analytics Toolbox™ provides tools to create deep learning networks for text data. For an example, see “Classify Text Data Using Deep Learning” on page 4-89.
	Text generation	For an example, see “Generate Text Using Deep Learning” on page 4-180.
Audio	Audio classification and regression	For an example, see “Speech Command Recognition Using Deep Learning” on page 4-23.

Choose Training Options

The `trainingOptions` function provides a variety of options to train your deep learning network.

Tip	More Information
Monitor training progress	To turn on the training progress plot, set the `'Plots'` option in `trainingOptions` to `'training-progress'`.

Tip	More Information
Use validation data	To specify validation data, use the `'ValidationData'` option in `trainingOptions`. **Note** If your validation data set is too small and does not sufficiently represent the data, then the reported metrics might not help you. Using a too large validation data set can result in slower training.
For transfer learning, speed up the learning of new layers and slow down the learning in the transferred layers	Specify higher learning rate factors for new layers by using, for example, the `WeightLearnRateFactor` property of `convolution2dLayer`. Decrease the initial learning rate using the `'InitialLearnRate'` option of `trainingOptions`. When transfer learning, you do not need to train for as many epochs. Decrease the number of epochs using the `'MaxEpochs'` option in `trainingOptions`. To learn how to interactively prepare a network for transfer learning using Deep Network Designer, see “Transfer Learning with Deep Network Designer” on page 2-2.
Shuffle your data every epoch	To shuffle your data every epoch (one full pass of the data), set the `'Shuffle'` option in `trainingOptions` to `'every-epoch'`. **Note** For sequence data, shuffling can have a negative impact on the accuracy as it can increase the amount of padding or truncated data. If you have sequence data, then sorting the data by sequence length can help. To learn more, see “Sequence Padding, Truncation, and Splitting” on page 1-79.
Try different optimizers	To specify different optimizers, use the `solverName` argument in `trainingOptions`.

For more information, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

Improve Training Accuracy

If you notice problems during training, then consider these possible solutions.

Problem	Possible Solution
NaNs or large spikes in the loss	Decrease the initial learning rate using the `'InitialLearnRate'` option of `trainingOptions`. If decreasing the learning rate does not help, then try using gradient clipping. To set the gradient threshold, use the `'GradientThreshold'` option in `trainingOptions`.
Loss is still decreasing at the end of training	Train for longer by increasing the number of epochs using the `'MaxEpochs'` option in `trainingOptions`.
Loss plateaus	If the loss plateaus at an unexpectedly high value, then drop the learning rate at the plateau. To change the learning rate schedule, use the `'LearnRateSchedule'` option in `trainingOptions`. If dropping the learning rate does not help, then the model might be underfitting. Try increasing the number of parameters or layers. You can check if the model is underfitting by monitoring the validation loss.
Validation loss is much higher than the training loss	To prevent overfitting, try one or more of the following: • Use data augmentation. For more information, see “Train Network with Augmented Images”. • Use dropout layers. For more information, see `dropoutLayer`. • Increase the global L2 regularization factor using the `'L2Regularization'` option in `trainingOptions`.
Loss decreases very slowly	Increase the initial learning rate using the `'InitialLearnRate'` option of `trainingOptions`. For image data, try including batch normalization layers in your network. For more information, see `batchNormalizationLayer`.

For more information, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

Fix Errors in Training

If your network does not train at all, then consider the possible solutions.

Error	Description	Possible Solution
Out-of-memory error when training	The available hardware is unable to store the current mini-batch, the network weights, and the computed activations.	Try reducing the mini-batch size using the `'MiniBatchSize'` option of `trainingOptions`. If reducing the mini-batch size does not work, then try using a smaller network, reducing the number of layers, or reducing the number of parameters or filters in the layers.
Custom layer errors	There could be an issue with the implementation of the custom layer.	Check the validity of the custom layer and find potential issues using `checkLayer`. If a test fails when you use `checkLayer`, then the function provides a test diagnostic and a framework diagnostic. The test diagnostic highlights any layer issues, whereas the framework diagnostic provides more detailed information. To learn more about the test diagnostics and get suggestions for possible solutions, see “Diagnostics” on page 18-158.
Training throws the error `'CUDA_ERROR_UNKNOWN'`	Sometimes, the GPU throws this error when it is being used for both compute and display requests from the OS.	Try reducing the mini-batch size using the `'MiniBatchSize'` option of `trainingOptions`. If reducing the mini-batch size does not work, then in Windows®, try adjusting the Timeout Detection and Recovery (TDR) settings. For example, change the `TdrDelay` from 2 seconds (default) to 4 seconds (requires registry edit).

You can analyze your deep learning network using `analyzeNetwork`. The `analyzeNetwork` function displays an interactive visualization of the network architecture, detects errors and issues with the network, and provides detailed information about the network layers. Use the network analyzer to visualize and understand the network architecture, check that you have defined the architecture correctly, and detect problems before training. Problems that `analyzeNetwork` detects include missing or disconnected layers, mismatched or incorrect sizes of layer inputs, an incorrect number of layer inputs, and invalid graph structures.

Prepare and Preprocess Data

You can improve the accuracy by preprocessing your data.

Weight or Balance Classes

Ideally, all classes have an equal number of observations. However, for some tasks, classes can be imbalanced. For example, automotive datasets of street scenes tend to have more sky, building, and road pixels than pedestrian and bicyclist pixels because the sky, buildings, and roads cover more image area. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes.

For classification tasks, you can specify class weights using the `'ClassWeights'` option of `classificationLayer`. For semantic segmentation tasks, you can specify class weights using the `ClassWeights` property of `pixelClassificationLayer`.

Alternatively, you can balance the classes by doing one or more of the following:

- Add new observations from the least frequent classes.
- Remove observations from the most frequent classes.
- Group similar classes. For example, group the classes "car" and "truck" into the single class "vehicle".

Preprocess Image Data

For more information about preprocessing image data, see “Preprocess Images for Deep Learning” on page 19-16.

Task	More Information
Resize images	To use a pretrained network, you must resize images to the input size of the network. To resize images, use `augmentedImageDatastore`. For example, this syntax resizes images in the image datastore `imds`: `auimds = augmentedImageDatastore(inputSize,imds);` **Tip** Use `augmentedImageDatastore` for efficient preprocessing of images for deep learning including image resizing. Do not use the `readFcn` option of `imageDatastore` for preprocessing or resizing as this option is usually significantly slower.
Image augmentation	To avoid overfitting, use image transformation. To learn more, see “Train Network with Augmented Images”.
Normalize regression targets	Normalize the predictors before you input them to the network. If you normalize the responses before training, then you must transform the predictions of the trained network to obtain the predictions of the original responses. For more information, see “Train Convolutional Neural Network for Regression” on page 3-53.

Preprocess Sequence Data

For more information about working with LSTM networks, see “Long Short-Term Memory Networks” on page 1-75.

Task	More Information
Normalize sequence data	To normalize sequence data, first calculate the per-feature mean and standard deviation for all the sequences. Then, for each training observation, subtract the mean value and divide by the standard deviation. To learn more, see “Normalize Sequence Data” on page 1-82.
Reduce sequence padding and truncation	To reduce the amount of padding or discarded data when padding or truncating sequences, try sorting your data by sequence length. To learn more, see “Sequence Padding, Truncation, and Splitting” on page 1-79.
Specify mini-batch size and padding options for prediction	When making predictions with sequences of different lengths, the mini-batch size can impact the amount of padding added to the input data which can result in different predicted values. Try using different values to see which works best with your network. To specify mini-batch size and padding options, use the `'MiniBatchSize'` and `'SequenceLength'` options of the `classify`, `predict`, `classifyAndUpdateState`, and `predictAndUpdateState` functions.

Use Available Hardware

To specify the execution environment, use the `'ExecutionEnvironment'` option in `trainingOptions`.

Problem	More Information
Training on CPU is slow	If training is too slow on a single CPU, try using a pretrained deep learning network as a feature extractor and train a machine learning model. For an example, see “Extract Image Features Using Pretrained Network” on page 3-28.
Training LSTM on GPU is slow	The CPU is better suited for training an LSTM network using mini-batches with short sequences. To use the CPU, set the `'ExecutionEnvironment'` option in `trainingOptions` to `'cpu'`.

Problem	More Information
Software does not use all available GPUs	If you have access to a machine with multiple GPUs, simply set the `'ExecutionEnvironment'` option in `trainingOptions` to `'multi-gpu'`. For more information, see “Deep Learning with MATLAB on Multiple GPUs” on page 7-13.

For more information, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2.

Fix Errors With Loading from MAT-Files

If you are unable to load layers or a network from a MAT-file and get a warning of the form

```
Warning: Unable to load instances of class layerType into a
heterogeneous array.  The definition of layerType could be
missing or contain an error.  Default objects will be
substituted.
Warning: While loading an object of class 'SeriesNetwork':
Error using 'forward' in Layer nnet.cnn.layer.MissingLayer. The
function threw an error and could not be executed.
```

then the network in the MAT-file may contain unavailable layers. This could be due to the following:

- The file contains a custom layer not on the path – To load networks containing custom layers, add the custom layer files to the MATLAB path.
- The file contains a custom layer from a support package – To load networks using layers from support packages, install the required support package at the command line by using the corresponding function (for example, `resnet18`) or using the Add-On Explorer.
- The file contains a custom layer from a documentation example that is not on the path – To load networks containing custom layers from documentation examples, open the example as a Live Script and copy the layer from the example folder to your working directory.
- The file contains a layer from a toolbox that is not installed – To access layers from other toolboxes, for example, Computer Vision Toolbox or Text Analytics Toolbox, install the corresponding toolbox.

After trying the suggested solutions, reload the MAT-file.

See Also

`trainingOptions` | `checkLayer` | `analyzeNetwork` | **Deep Network Designer**

More About

- “Pretrained Deep Neural Networks” on page 1-8
- “Preprocess Images for Deep Learning” on page 19-16
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Convert Classification Network into Regression Network” on page 3-70

Long Short-Term Memory Networks

This topic explains how to work with sequence and time series data for classification and regression tasks using long short-term memory (LSTM) networks. For an example showing how to classify sequence data using an LSTM network, see “Sequence Classification Using Deep Learning” on page 4-2.

An LSTM network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data.

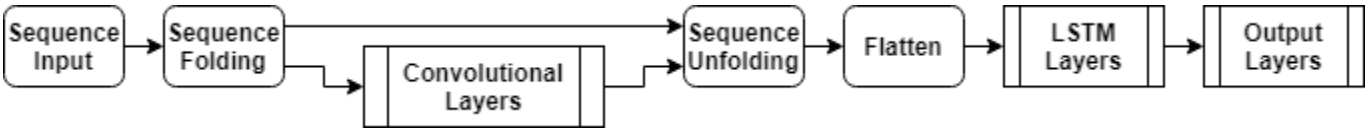
LSTM Network Architecture

The core components of an LSTM network are a sequence input layer and an LSTM layer. A *sequence input layer* inputs sequence or time series data into the network. An *LSTM layer* learns long-term dependencies between time steps of sequence data.

This diagram illustrates the architecture of a simple LSTM network for classification. The network starts with a sequence input layer followed by an LSTM layer. To predict class labels, the network ends with a fully connected layer, a softmax layer, and a classification output layer.

This diagram illustrates the architecture of a simple LSTM network for regression. The network starts with a sequence input layer followed by an LSTM layer. The network ends with a fully connected layer and a regression output layer.

This diagram illustrates the architecture of a network for video classification. To input image sequences to the network, use a sequence input layer. To use convolutional layers to extract features, that is, to apply the convolutional operations to each frame of the videos independently, use a sequence folding layer followed by the convolutional layers, and then a sequence unfolding layer. To use the LSTM layers to learn from sequences of vectors, use a flatten layer followed by the LSTM and output layers.



Classification LSTM Networks

To create an LSTM network for sequence-to-label classification, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, a softmax layer, and a classification output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of classes. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode `'last'`.

```
numFeatures = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For an example showing how to train an LSTM network for sequence-to-label classification and classify new data, see “Sequence Classification Using Deep Learning” on page 4-2.

To create an LSTM network for sequence-to-sequence classification, use the same architecture as for sequence-to-label classification, but set the output mode of the LSTM layer to `'sequence'`.

```
numFeatures = 12;
numHiddenUnits = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Regression LSTM Networks

To create an LSTM network for sequence-to-one regression, create a layer array containing a sequence input layer, an LSTM layer, a fully connected layer, and a regression output layer.

Set the size of the sequence input layer to the number of features of the input data. Set the size of the fully connected layer to the number of responses. You do not need to specify the sequence length.

For the LSTM layer, specify the number of hidden units and the output mode `'last'`.

```
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

To create an LSTM network for sequence-to-sequence regression, use the same architecture as for sequence-to-one regression, but set the output mode of the LSTM layer to `'sequence'`.

```
numFeatures = 12;
numHiddenUnits = 125;
numResponses = 1;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
```

```
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

For an example showing how to train an LSTM network for sequence-to-sequence regression and predict on new data, see “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47.

Video Classification Network

To create a deep learning network for data containing sequences of images such as video data and medical images, specify image sequence input using the sequence input layer.

To use convolutional layers to extract features, that is, to apply the convolutional operations to each frame of the videos independently, use a sequence folding layer followed by the convolutional layers, and then a sequence unfolding layer. To use the LSTM layers to learn from sequences of vectors, use a flatten layer followed by the LSTM and output layers.

```
inputSize = [28 28 1];
filterSize = 5;
numFilters = 20;
numHiddenUnits = 200;
numClasses = 10;

layers = [ ...
    sequenceInputLayer(inputSize,'Name','input')

    sequenceFoldingLayer('Name','fold')

    convolution2dLayer(filterSize,numFilters,'Name','conv')
    batchNormalizationLayer('Name','bn')
    reluLayer('Name','relu')

    sequenceUnfoldingLayer('Name','unfold')
    flattenLayer('Name','flatten')

    lstmLayer(numHiddenUnits,'OutputMode','last','Name','lstm')

    fullyConnectedLayer(numClasses, 'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')];
```

Convert the layers to a layer graph and connect the `miniBatchSize` output of the sequence folding layer to the corresponding input of the sequence unfolding layer.

```
lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'fold/miniBatchSize','unfold/miniBatchSize');
```

For an example showing how to train a deep learning network for video classification, see “Classify Videos Using Deep Learning” on page 4-54.

Deeper LSTM Networks

You can make LSTM networks deeper by inserting extra LSTM layers with the output mode `'sequence'` before the LSTM layer. To prevent overfitting, you can insert dropout layers after the LSTM layers.

For sequence-to-label classification networks, the output mode of the last LSTM layer must be `'last'`.

```
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','last')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

For sequence-to-sequence classification networks, the output mode of the last LSTM layer must be `'sequence'`.

```
numFeatures = 12;
numHiddenUnits1 = 125;
numHiddenUnits2 = 100;
numClasses = 9;
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Layers

Layer	Description
 `sequenceInputLayer`	A sequence input layer inputs sequence data to a network.
 `lstmLayer`	An LSTM layer learns long-term dependencies between time steps in time series and sequence data.
 `bilstmLayer`	A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.
 `gruLayer`	A GRU layer learns dependencies between time steps in time series and sequence data.

Layer	Description
 `convolution1dLayer`	A 1-D convolutional layer applies sliding convolutional filters to 1-D input.
 `maxPooling1dLayer`	A 1-D max pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.
 `averagePooling1dLayer`	A 1-D average pooling layer performs downsampling by dividing the input into 1-D pooling regions, then computing the average of each region.
 `globalMaxPooling1dLayer`	A 1-D global max pooling layer performs downsampling by outputting the maximum of the time or spatial dimensions of the input.
 `sequenceFoldingLayer`	A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.
 `sequenceUnfoldingLayer`	A sequence unfolding layer restores the sequence structure of the input data after sequence folding.
 `flattenLayer`	A flatten layer collapses the spatial dimensions of the input into the channel dimension.
 `wordEmbeddingLayer`	A word embedding layer maps word indices to vectors.

Classification, Prediction, and Forecasting

To classify or make predictions on new data, use `classify` and `predict`.

LSTM networks can remember the state of the network between predictions. The network state is useful when you do not have the complete time series in advance, or if you want to make multiple predictions on a long time series.

To predict and classify on parts of a time series and update the network state, use `predictAndUpdateState` and `classifyAndUpdateState`. To reset the network state between predictions, use `resetState`.

For an example showing how to forecast future time steps of a sequence, see “Time Series Forecasting Using Deep Learning” on page 4-15.

Sequence Padding, Truncation, and Splitting

LSTM networks support input data with varying sequence lengths. When passing data through the network, the software pads, truncates, or splits sequences so that all the sequences in each mini-batch have the specified length. You can specify the sequence lengths and the value used to pad the sequences using the `SequenceLength` and `SequencePaddingValue` name-value pair arguments in `trainingOptions`.

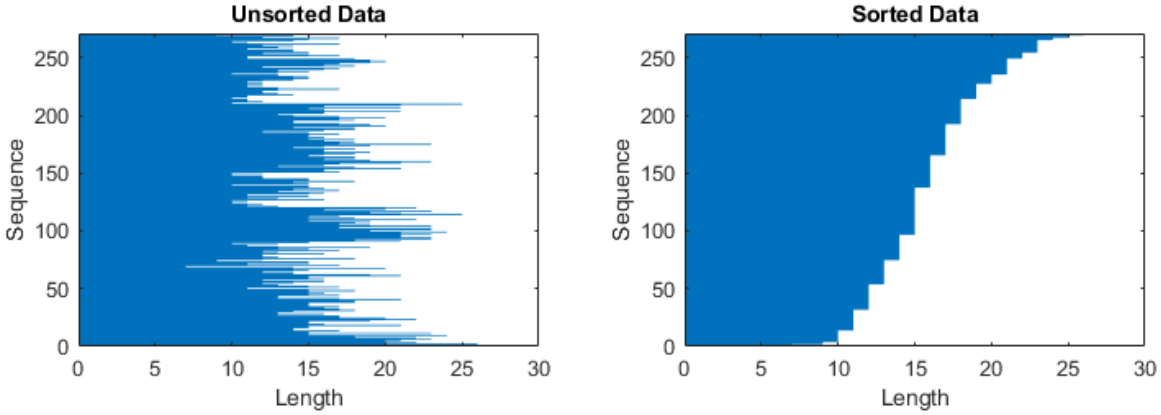
After training the network, use the same mini-batch size and padding options when using the `classify`, `predict`, `classifyAndUpdateState`, `predictAndUpdateState`, and `activations` functions.

Sort Sequences by Length

To reduce the amount of padding or discarded data when padding or truncating sequences, try sorting your data by sequence length. To sort the data by sequence length, first get the number of columns of each sequence by applying `size(X,2)` to every sequence using `cellfun`. Then sort the sequence lengths using `sort`, and use the second output to reorder the original sequences.

```
sequenceLengths = cellfun(@(X) size(X,2), XTrain);
[sequenceLengthsSorted,idx] = sort(sequenceLengths);
XTrain = XTrain(idx);
```

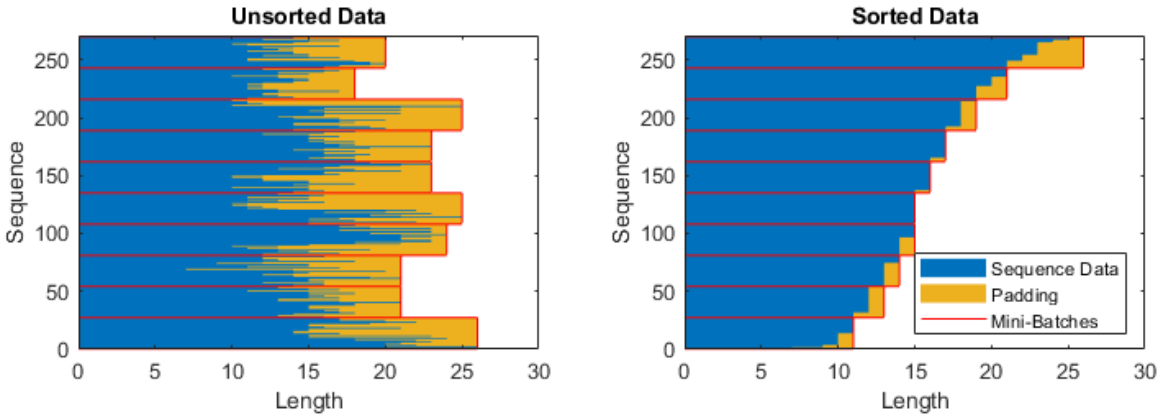
The following figures show the sequence lengths of the sorted and unsorted data in bar charts.



Pad Sequences

If you specify the sequence length `'longest'`, then the software pads the sequences so that all the sequences in a mini-batch have the same length as the longest sequence in the mini-batch. This option is the default.

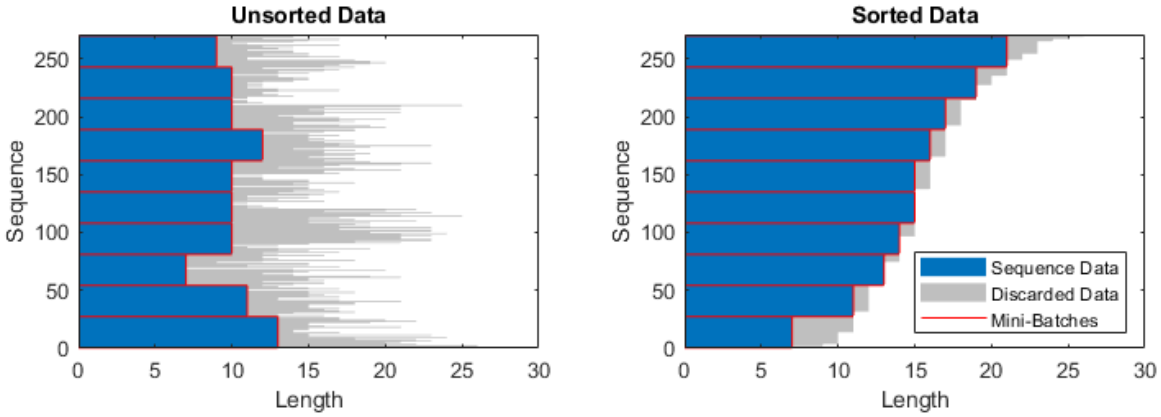
The following figures illustrate the effect of setting `'SequenceLength'` to `'longest'`.



Truncate Sequences

If you specify the sequence length `'shortest'`, then the software truncates the sequences so that all the sequences in a mini-batch have the same length as the shortest sequence in that mini-batch. The remaining data in the sequences is discarded.

The following figures illustrate the effect of setting `'SequenceLength'` to `'shortest'`.



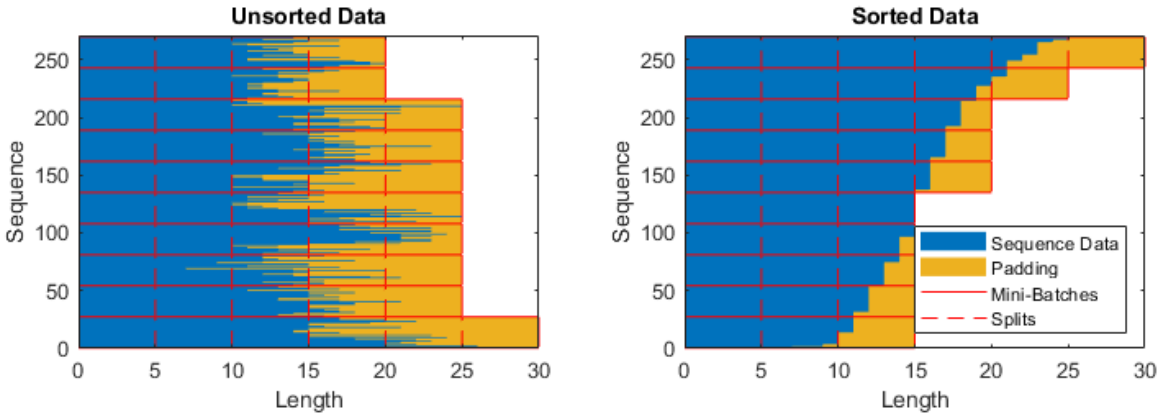
Split Sequences

If you set the sequence length to an integer value, then software pads all the sequences in a mini-batch to the nearest multiple of the specified length that is greater than the longest sequence length in the mini-batch. Then, the software splits each sequence into smaller sequences of the specified length. If splitting occurs, then the software creates extra mini-batches.

Use this option if the full sequences do not fit in memory. Alternatively, you can try reducing the number of sequences per mini-batch by setting the `'MiniBatchSize'` option in `trainingOptions` to a lower value.

If you specify the sequence length as a positive integer, then the software processes the smaller sequences in consecutive iterations. The network updates the network state between the split sequences.

The following figures illustrate the effect of setting `'SequenceLength'` to 5.



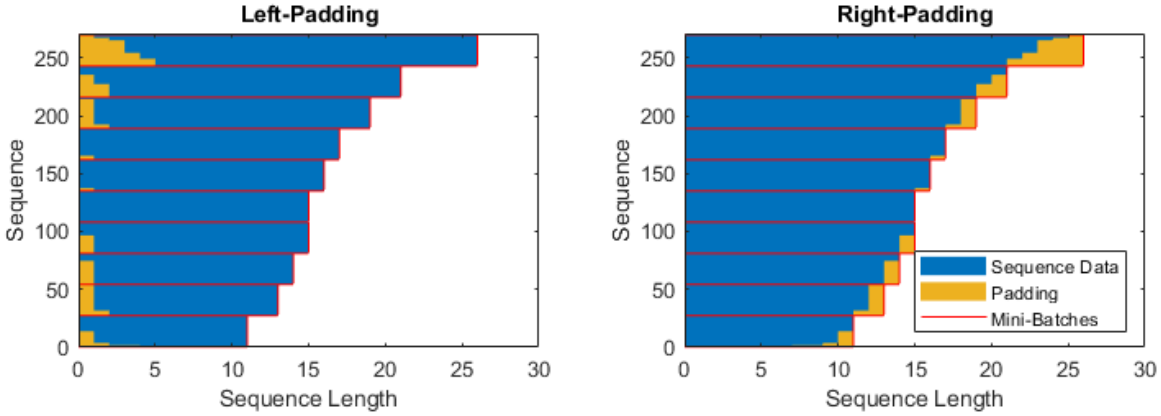
Specify Padding Direction

The location of the padding and truncation can impact training, classification, and prediction accuracy. Try setting the `'SequencePaddingDirection'` option in `trainingOptions` to `'left'` or `'right'` and see which is best for your data.

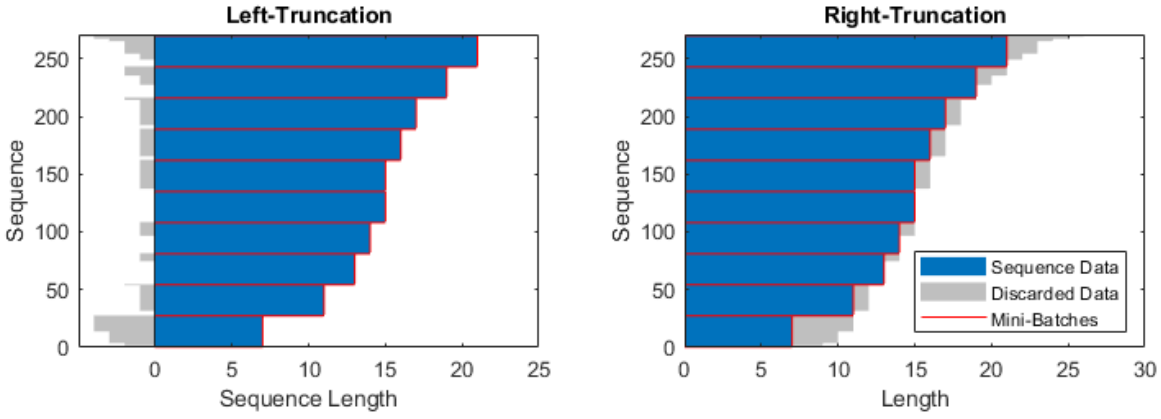
Because LSTM layers process sequence data one time step at a time, when the layer `OutputMode` property is `'last'`, any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the `'SequencePaddingDirection'` option to `'left'`.

For sequence-to-sequence networks (when the `OutputMode` property is `'sequence'` for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the `'SequencePaddingDirection'` option to `'right'`.

The following figures illustrate padding sequence data on the left and on the right.



The following figures illustrate truncating sequence data on the left and on the right.



Normalize Sequence Data

To recenter training data automatically at training time using zero-center normalization, set the `Normalization` option of `sequenceInputLayer` to `'zerocenter'`. Alternatively, you can normalize sequence data by first calculating the per-feature mean and standard deviation of all the

sequences. Then, for each training observation, subtract the mean value and divide by the standard deviation.

```
mu = mean([XTrain{:}],2);
sigma = std([XTrain{:}],0,2);
XTrain = cellfun(@(X) (X-mu)./sigma,XTrain,'UniformOutput',false);
```

Out-of-Memory Data

Use datastores for sequence, time series, and signal data when data is too large to fit in memory or to perform specific operations when reading batches of data.

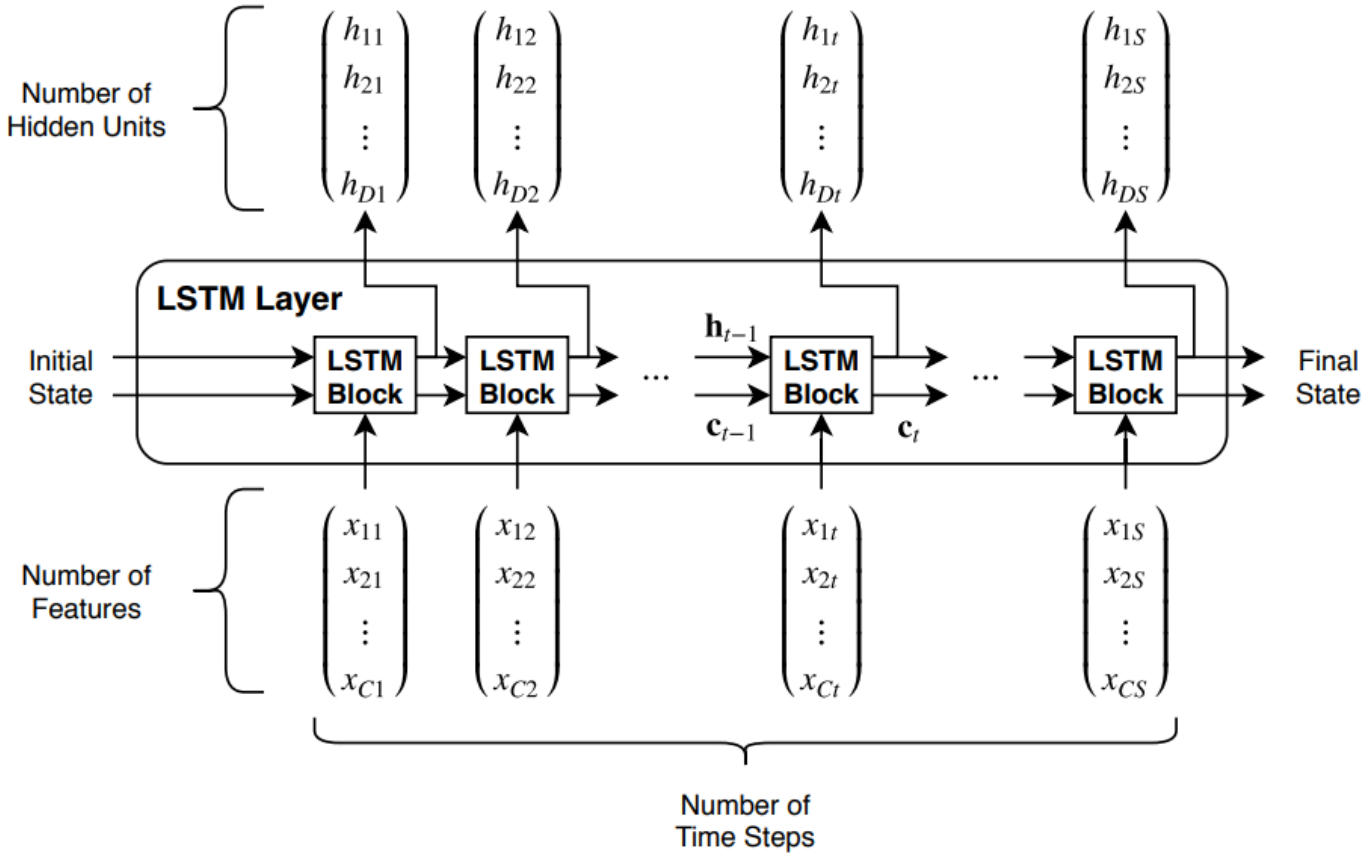
To learn more, see “Train Network Using Out-of-Memory Sequence Data” on page 19-99 and “Classify Out-of-Memory Text Data Using Deep Learning” on page 19-108.

Visualization

Investigate and visualize the features learned by LSTM networks from sequence and time series data by extracting the activations using the `activations` function. To learn more, see “Visualize Activations of LSTM Network” on page 5-152.

LSTM Layer Architecture

This diagram illustrates the flow of a time series X with C features (channels) of length S through an LSTM layer. In the diagram, $\mathbf{h}_t$ and $\mathbf{c}_t$ denote the output (also known as the *hidden state*) and the *cell state* at time step t, respectively.



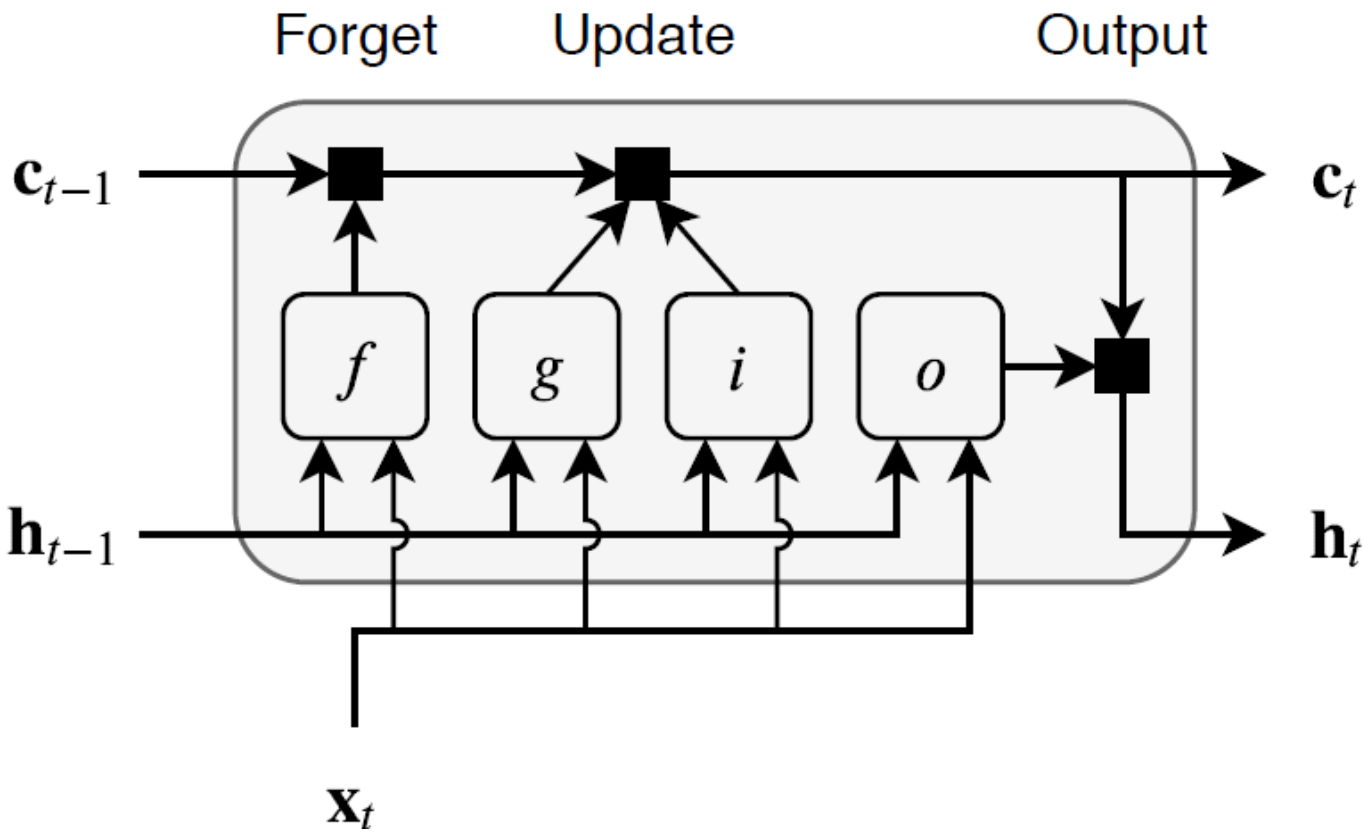
The first LSTM block uses the initial state of the network and the first time step of the sequence to compute the first output and the updated cell state. At time step t, the block uses the current state of the network ($\mathbf{c}_{t-1}, \mathbf{h}_{t-1}$) and the next time step of the sequence to compute the output and the updated cell state $\mathbf{c}_t$.

The state of the layer consists of the *hidden state* (also known as the *output state*) and the *cell state*. The hidden state at time step t contains the output of the LSTM layer for this time step. The cell state contains information learned from the previous time steps. At each time step, the layer adds information to or removes information from the cell state. The layer controls these updates using *gates*.

The following components control the cell state and hidden state of the layer.

Component	Purpose
Input gate (i)	Control level of cell state update
Forget gate (f)	Control level of cell state reset (forget)
Cell candidate (g)	Add information to cell state
Output gate (o)	Control level of cell state added to hidden state

This diagram illustrates the flow of data at time step t. The diagram highlights how the gates forget, update, and output the cell and hidden states.



The learnable weights of an LSTM layer are the input weights W (`InputWeights`), the recurrent weights R (`RecurrentWeights`), and the bias b (`Bias`). The matrices W, R, and b are concatenations of the input weights, the recurrent weights, and the bias of each component, respectively. These matrices are concatenated as follows:

$$W = \begin{bmatrix} W_i \\ W_f \\ W_g \\ W_o \end{bmatrix}, R = \begin{bmatrix} R_i \\ R_f \\ R_g \\ R_o \end{bmatrix}, b = \begin{bmatrix} b_i \\ b_f \\ b_g \\ b_o \end{bmatrix},$$

where i, f, g, and o denote the input gate, forget gate, cell candidate, and output gate, respectively.

The cell state at time step t is given by

$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot g_t,$$

where $\odot$ denotes the Hadamard product (element-wise multiplication of vectors).

The hidden state at time step t is given by

$$\mathbf{h}_t = o_t \odot \sigma_c(\mathbf{c}_t),$$

where σ_c denotes the state activation function. The `lstmLayer` function, by default, uses the hyperbolic tangent function (tanh) to compute the state activation function.

The following formulas describe the components at time step t.

Component	Formula
Input gate	$i_t = \sigma_g(W_i\mathbf{x}_t + \mathrm{R}_i\mathbf{h}_{t-1} + b_i)$
Forget gate	$f_t = \sigma_g(W_f\mathbf{x}_t + \mathrm{R}_f\mathbf{h}_{t-1} + b_f)$
Cell candidate	$g_t = \sigma_c(W_g\mathbf{x}_t + \mathrm{R}_g\mathbf{h}_{t-1} + b_g)$
Output gate	$o_t = \sigma_g(W_o\mathbf{x}_t + \mathrm{R}_o\mathbf{h}_{t-1} + b_o)$

In these calculations, σ_g denotes the gate activation function. The `lstmLayer` function, by default, uses the sigmoid function given by $\sigma(x) = (1 + e^{-x})^{-1}$ to compute the gate activation function.

References

[1] Hochreiter, S., and J. Schmidhuber. "Long short-term memory." *Neural computation*. Vol. 9, Number 8, 1997, pp.1735–1780.

See Also

`sequenceInputLayer` | `lstmLayer` | `bilstmLayer` | `gruLayer` | `classifyAndUpdateState` | `predictAndUpdateState` | `resetState` | `sequenceFoldingLayer` | `sequenceUnfoldingLayer` | `flattenLayer` | `wordEmbeddingLayer` | `activations`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Classify Videos Using Deep Learning” on page 4-54
- “Visualize Activations of LSTM Network” on page 5-152
- “Develop Custom Mini-Batch Datastore” on page 19-36
- “Deep Learning in MATLAB” on page 1-2

Deep Network Designer

Transfer Learning with Deep Network Designer

This example shows how to perform transfer learning interactively using the Deep Network Designer app.

Transfer learning is the process of taking a pretrained deep learning network and fine-tuning it to learn a new task. Using transfer learning is usually faster and easier than training a network from scratch. You can quickly transfer learned features to a new task using a smaller amount of data.

Use Deep Network Designer to perform transfer learning for image classification by following these steps:

1. Open the Deep Network Designer app and choose a pretrained network.
2. Import the new data set.
3. Replace the final layers with new layers adapted to the new data set.
4. Set learning rates so that learning is faster in the new layers than in the transferred layers.
5. Train the network using Deep Network Designer, or export the network for training at the command line.

Extract Data

In the workspace, extract the MathWorks Merch data set. This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*).

```
unzip("MerchData.zip");
```

Select a Pretrained Network

To open Deep Network Designer, on the **Apps** tab, under **Machine Learning and Deep Learning**, click the app icon. Alternatively, you can open the app from the command line:

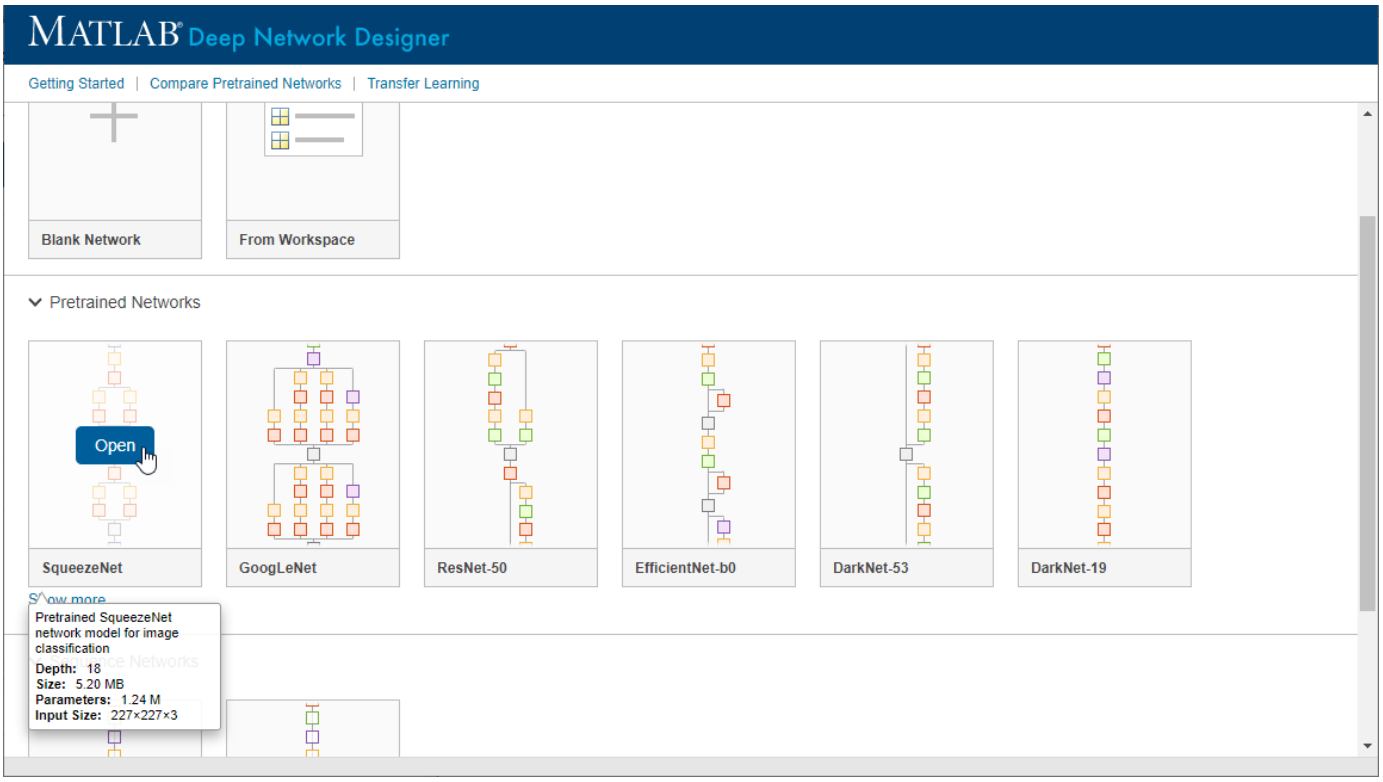
```
deepNetworkDesigner
```

Deep Network Designer provides a selection of pretrained image classification networks that have learned rich feature representations suitable for a wide range of images. Transfer learning works best if your images are similar to the images originally used to train the network. If your training images are natural images like those in the ImageNet database, then any of the pretrained networks is suitable. For a list of available networks and how to compare them, see “Pretrained Deep Neural Networks” on page 1-8.

If your data is very different from the ImageNet data—for example, if you have tiny images, spectrograms, or nonimage data—training a new network might be better. For examples showing how to train a network from scratch, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-51 and “Train Simple Semantic Segmentation Network in Deep Network Designer” on page 8-148.

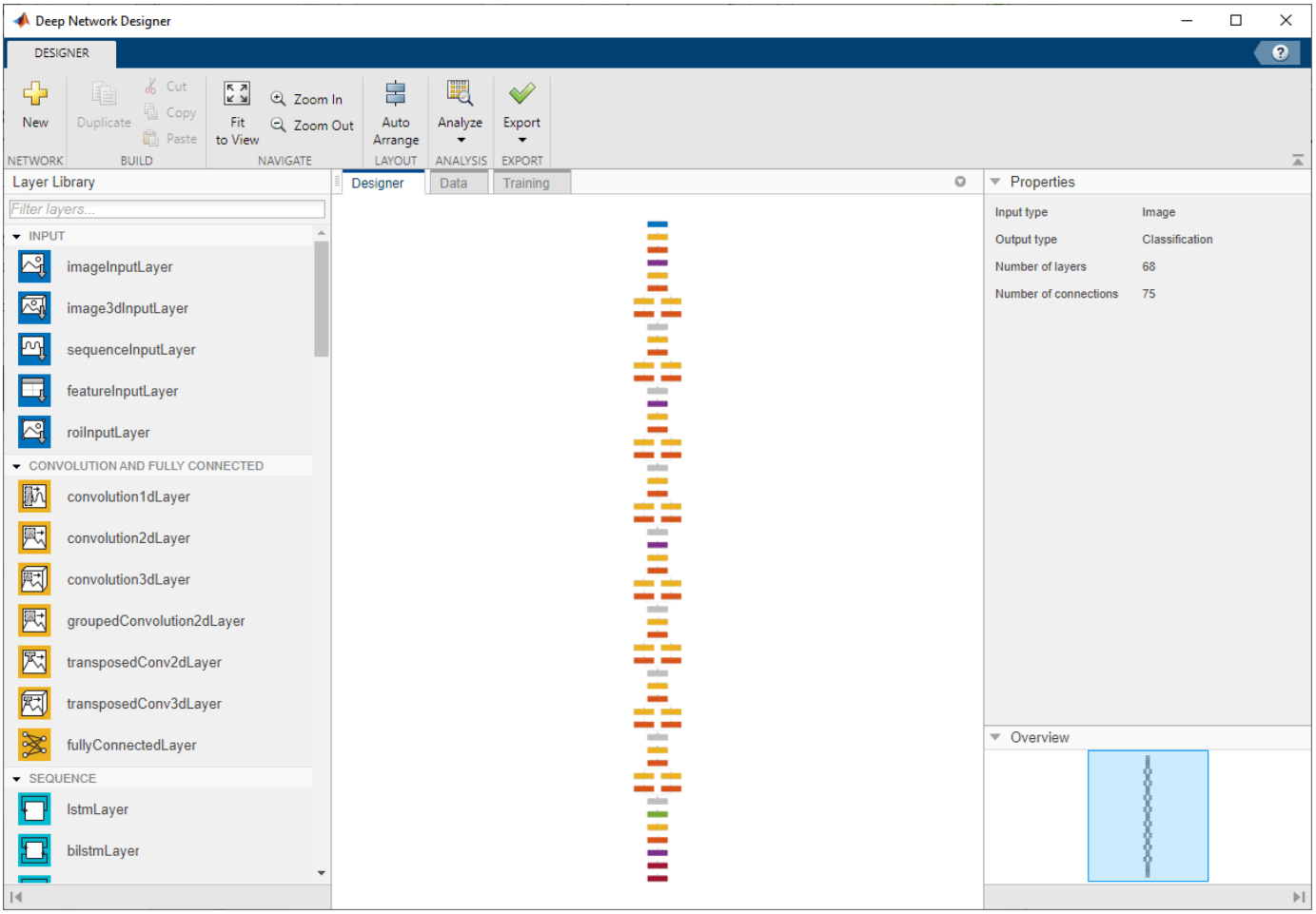
SqueezeNet does not require an additional support package. For other pretrained networks, if you do not have the required support package installed, then the app provides the **Install** option.

Select **SqueezeNet** from the list of pretrained networks and click **Open**.



Explore Network

Deep Network Designer displays a zoomed-out view of the whole network in the **Designer** pane.



Explore the network plot. To zoom in with the mouse, use **Ctrl**+scroll wheel. To pan, use the arrow keys, or hold down the scroll wheel and drag the mouse. Select a layer to view its properties. Deselect all layers to view the network summary in the **Properties** pane.

Import Data

To load the data into Deep Network Designer, on the **Data** tab, click **Import Data** > **Import Image Data**. The Import Image Data dialog box opens.

In the **Data source** list, select **Folder**. Click **Browse** and select the extracted MerchData folder.

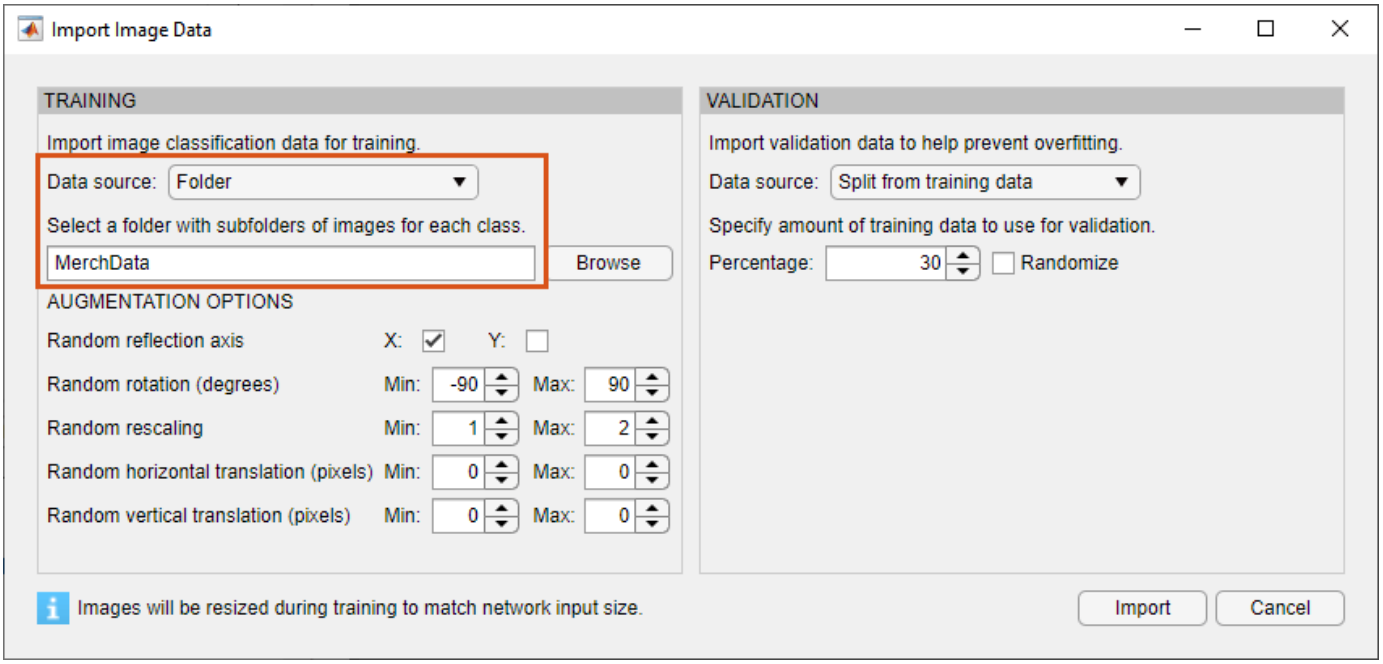


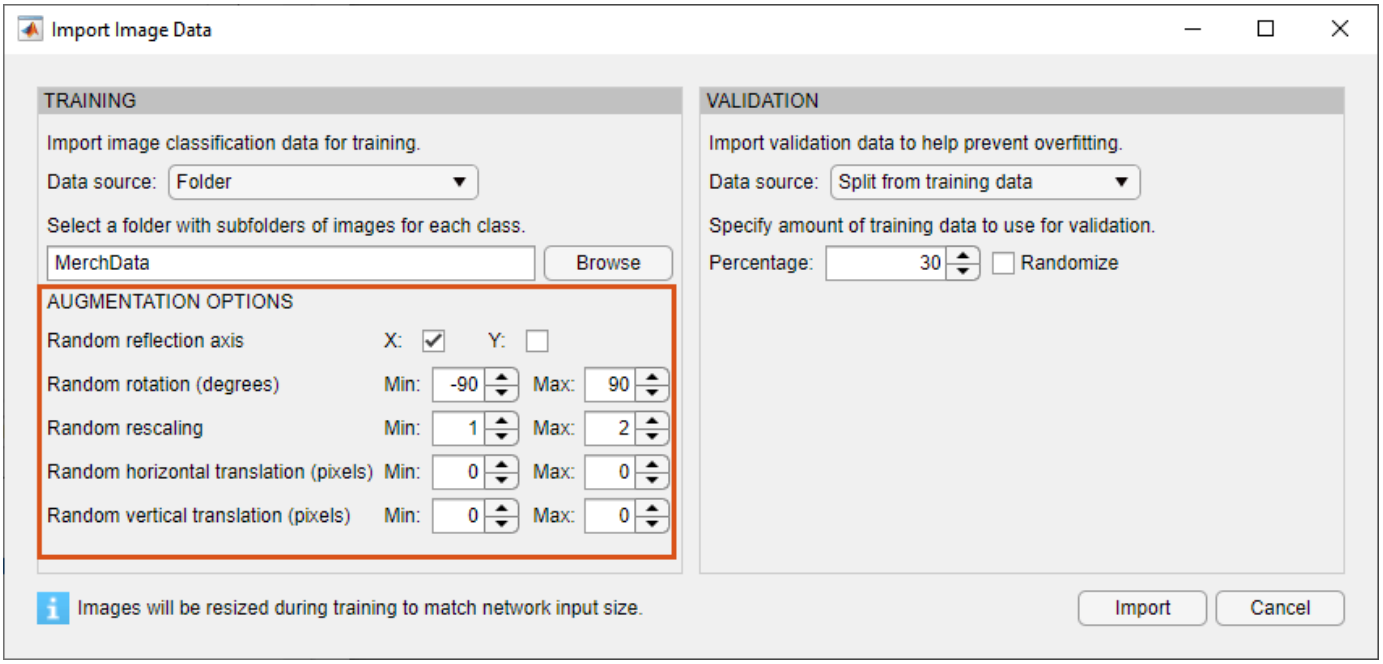
Image Augmentation

You can choose to apply image augmentation to your training data. The Deep Network Designer app provides the following augmentation options:

- Random reflection in the x-axis
- Random reflection in the y-axis
- Random rotation
- Random rescaling
- Random horizontal translation
- Random vertical translation

You can effectively increase the amount of training data by applying randomized augmentation to your data. Augmentation also enables you to train networks to be invariant to distortions in image data. For example, you can add randomized rotations to input images so that a network is invariant to the presence of rotation in input images.

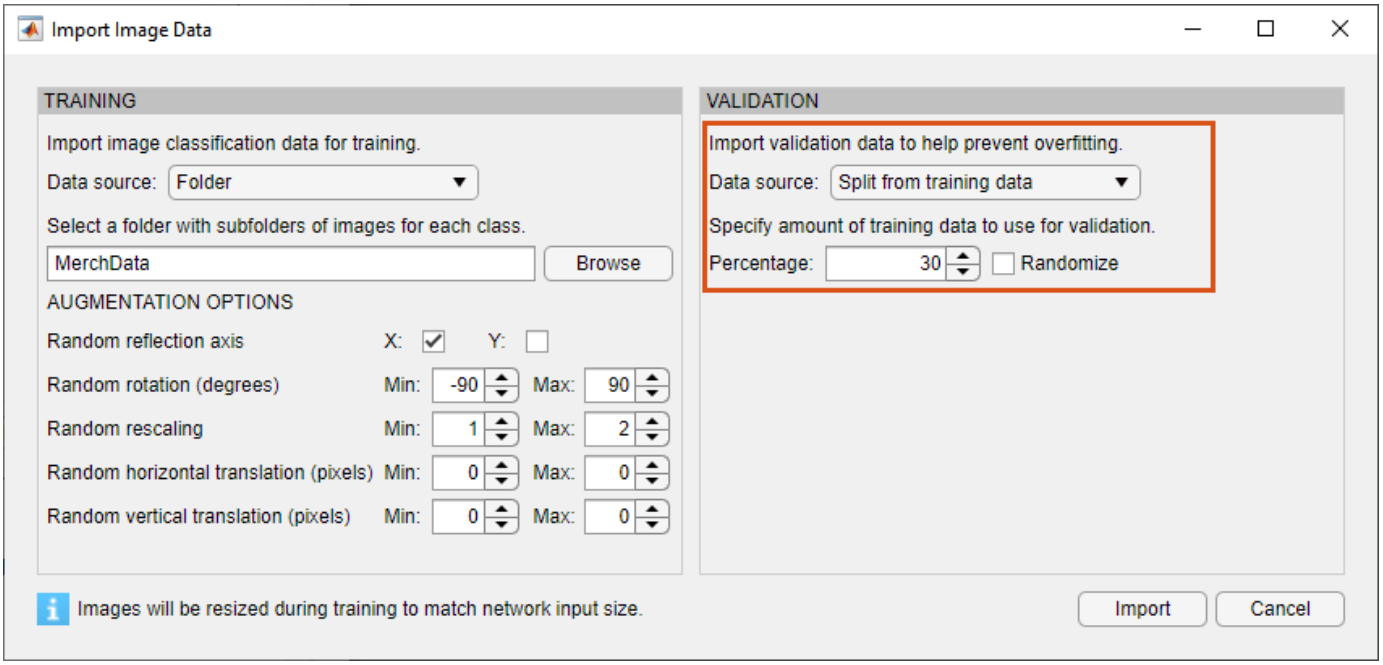
For this example, apply a random reflection in the x-axis, a random rotation from the range [-90,90] degrees, and a random rescaling from the range [1,2].



Validation Data

You can also choose to import validation data either by splitting it from the training data, or by importing it from another source. Validation estimates model performance on new data compared to the training data, and helps you to monitor performance and protect against overfitting.

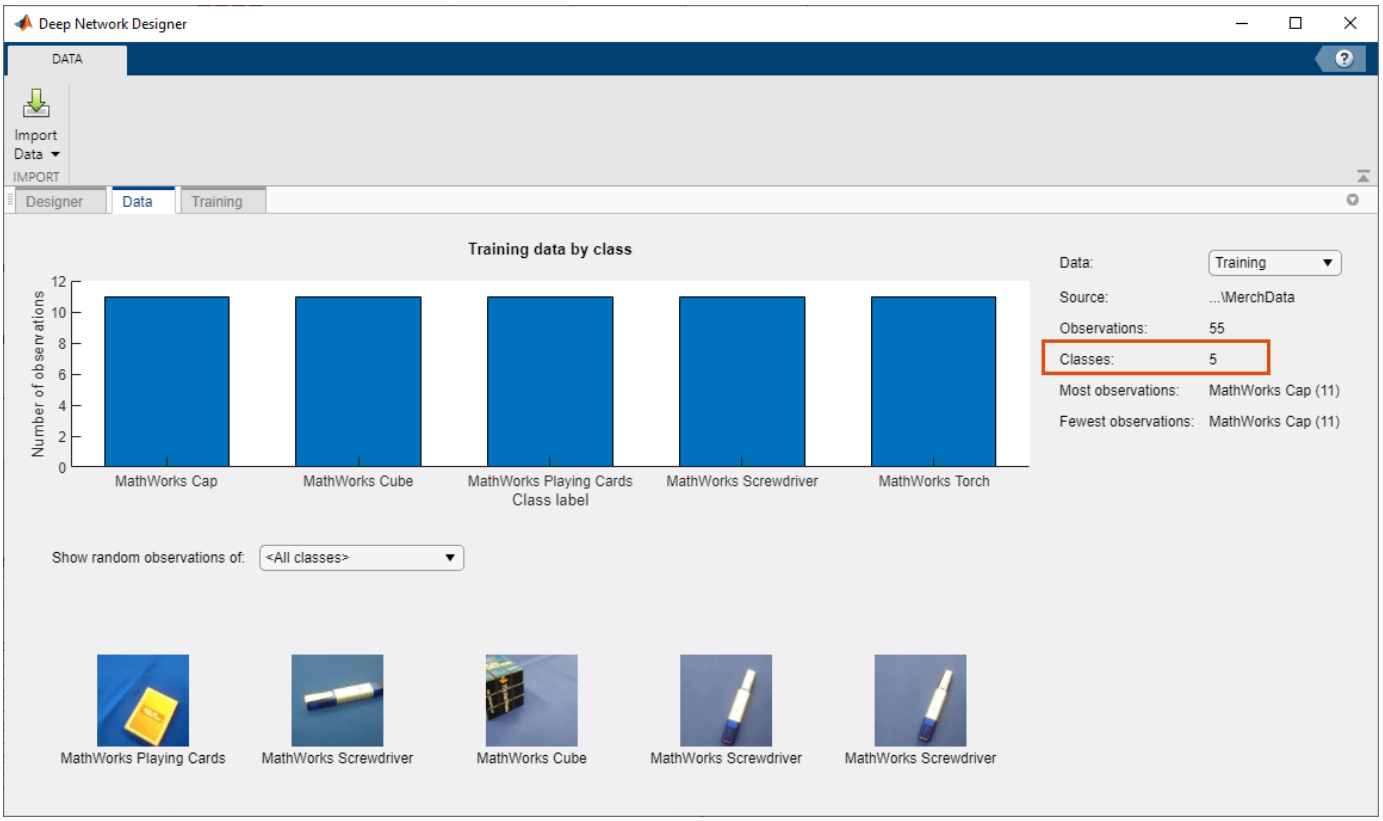
For this example, use 30% of the images for validation.



Click **Import** to import the data into Deep Network Designer.

Visualize Data

Using Deep Network Designer, you can visually inspect the distribution of the training and validation data in the **Data** tab. You can see that, in this example, there are five classes in the data set. You can also see random observations from each class.



Prepare Network for Training

Edit the network in the **Designer** pane to specify a new number of classes in your data. To prepare the network for transfer learning, replace the last learnable layer and the final classification layer.

Replace Last Learnable Layer

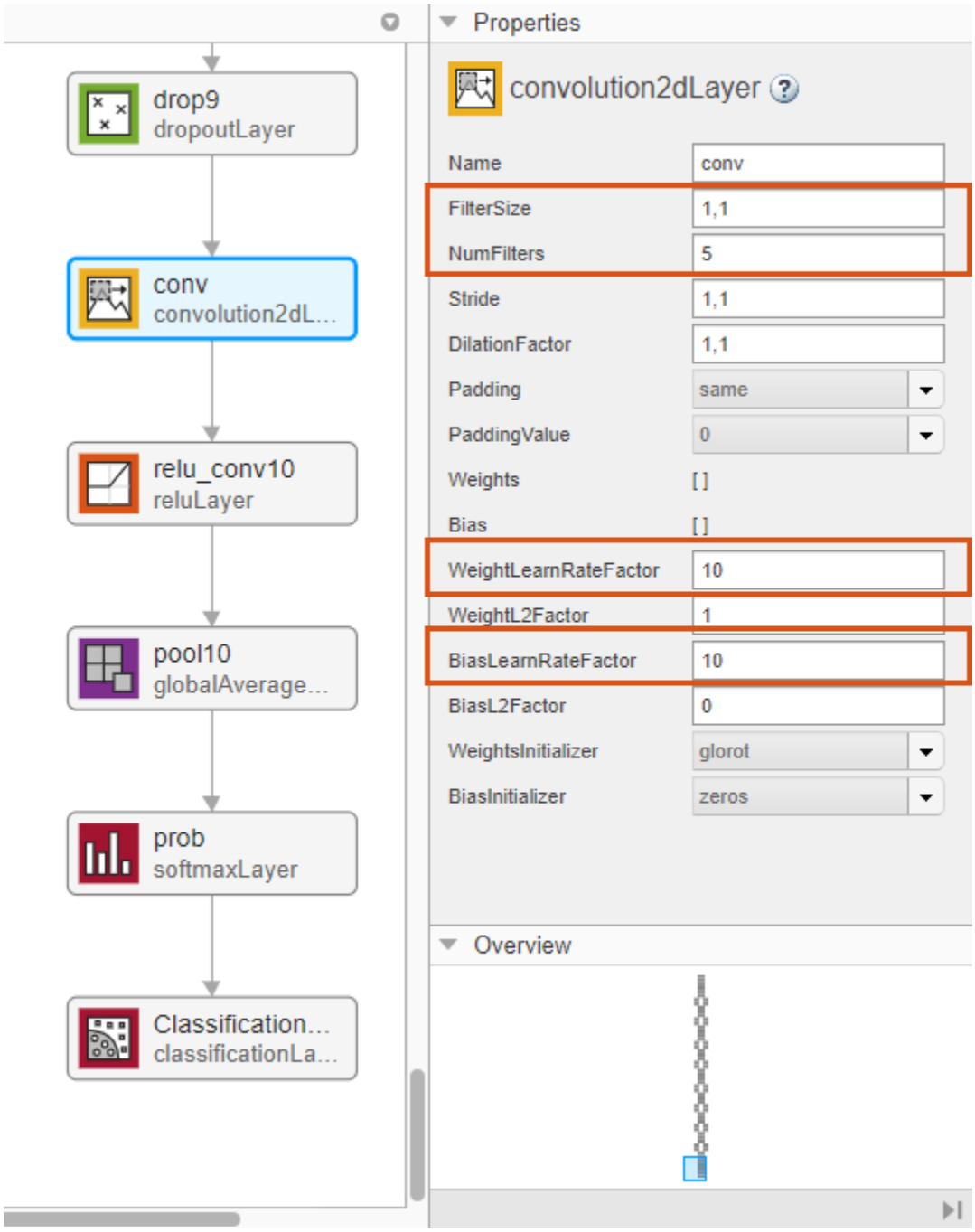
To use a pretrained network for transfer learning, you must change the number of classes to match your new data set. First, find the last learnable layer in the network. For SqueezeNet, the last learnable layer is the last convolutional layer, `'conv10'`. In this case, replace the convolutional layer with a new convolutional layer with the number of filters equal to the number of classes.

Drag a new `convolution2dLayer` onto the canvas. To match the original convolutional layer, set `FilterSize` to `1,1`.

The `NumFilters` property defines the number of classes for classification problems. Change `NumFilters` to the number of classes in the new data, in this example, `5`.

Change the learning rates so that learning is faster in the new layer than in the transferred layers by setting `WeightLearnRateFactor` and `BiasLearnRateFactor` to `10`.

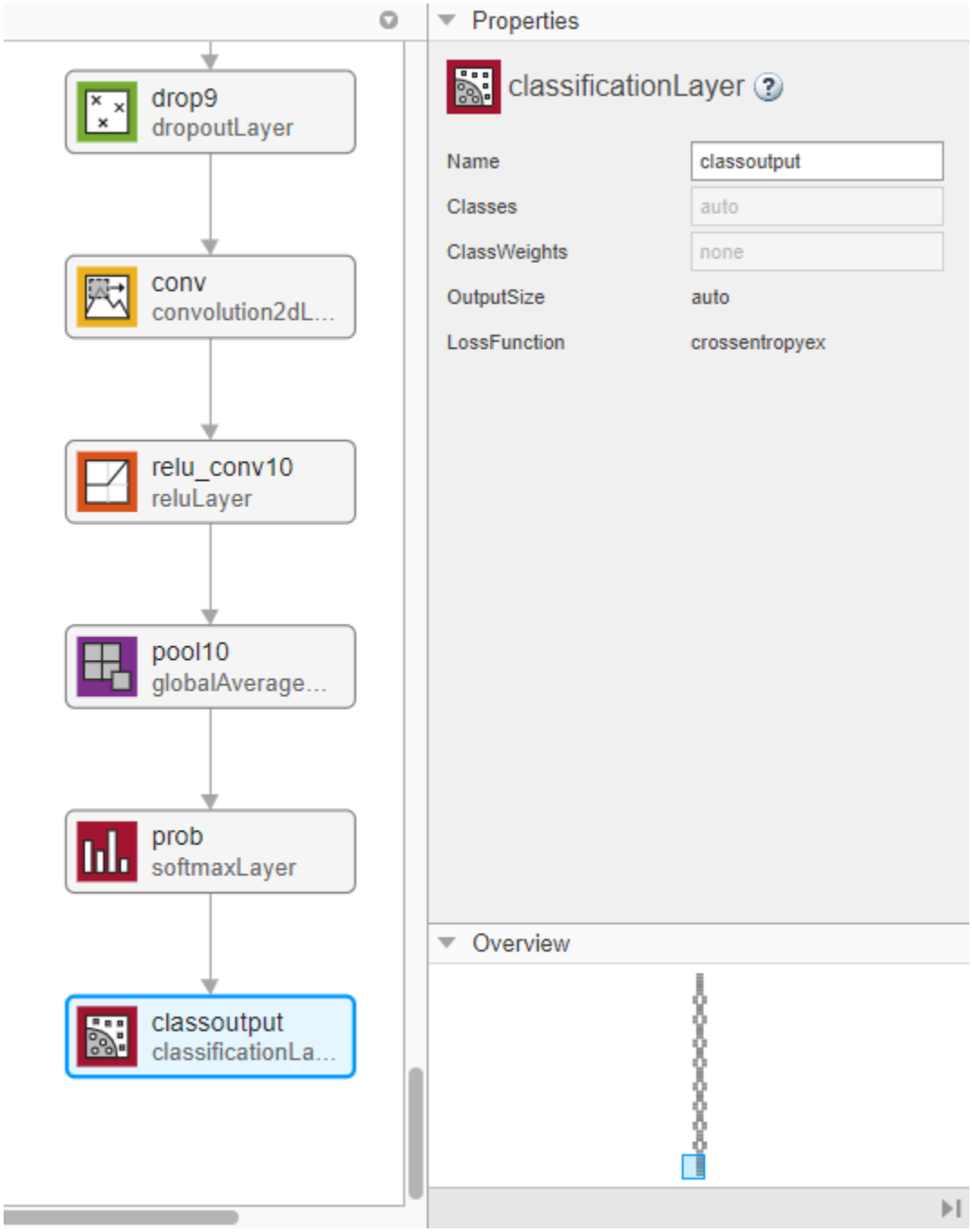
Delete the last 2-D convolutional layer and connect your new layer instead.



Replace Output Layer

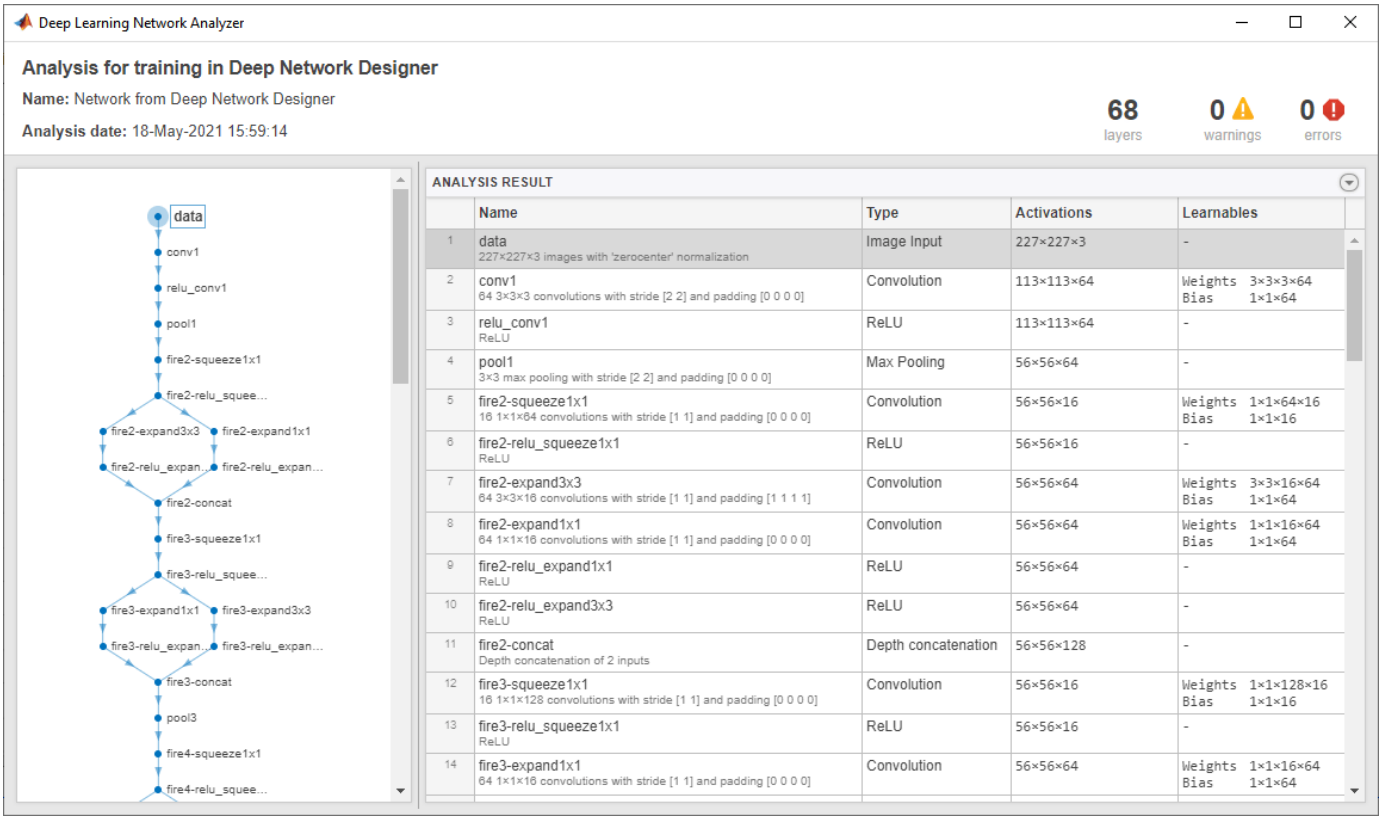
For transfer learning, you need to replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original classification layer and connect your new layer in its place.

For a new output layer, you do not need to set the `OutputSize`. At training time, Deep Network Designer automatically sets the output classes of the layer from the data.



Check Network

To check that the network is ready for training, click **Analyze**. If the Deep Learning Network Analyzer reports zero errors, then the edited network is ready for training.



Train Network

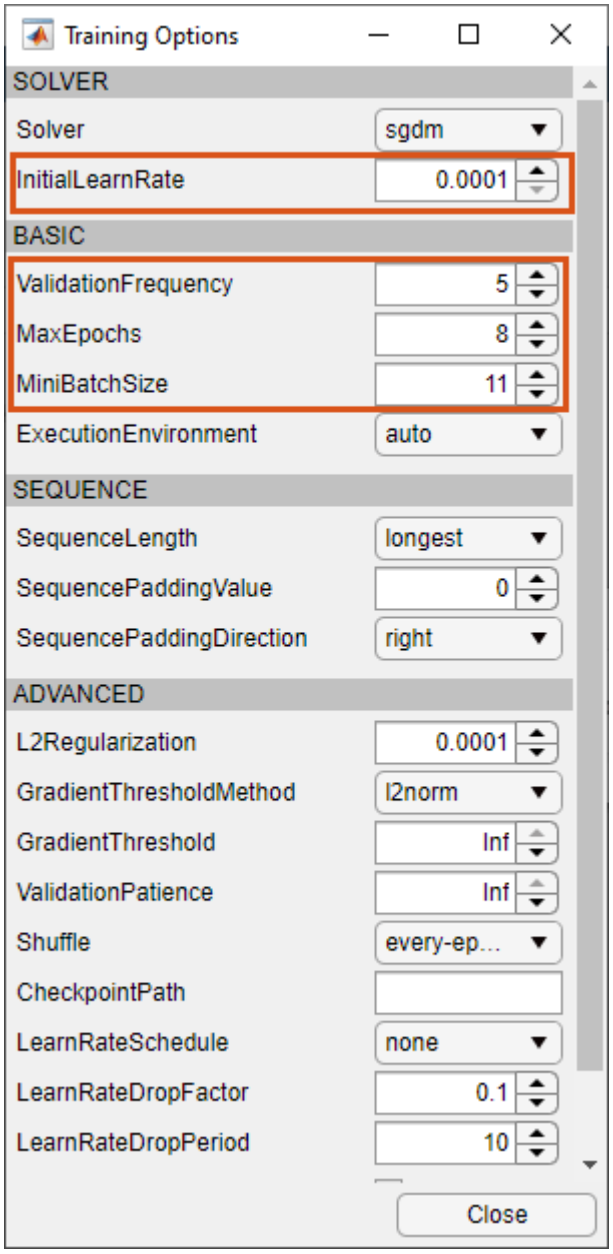
In Deep Network Designer you can train networks imported or created in the app.

To train the network with the default settings, on the **Training** tab, click **Train**. The default training options are better suited for large data sets, for small data sets reduce the mini-batch size and the validation frequency.

If you want greater control over the training, click **Training Options** and choose the settings to train with.

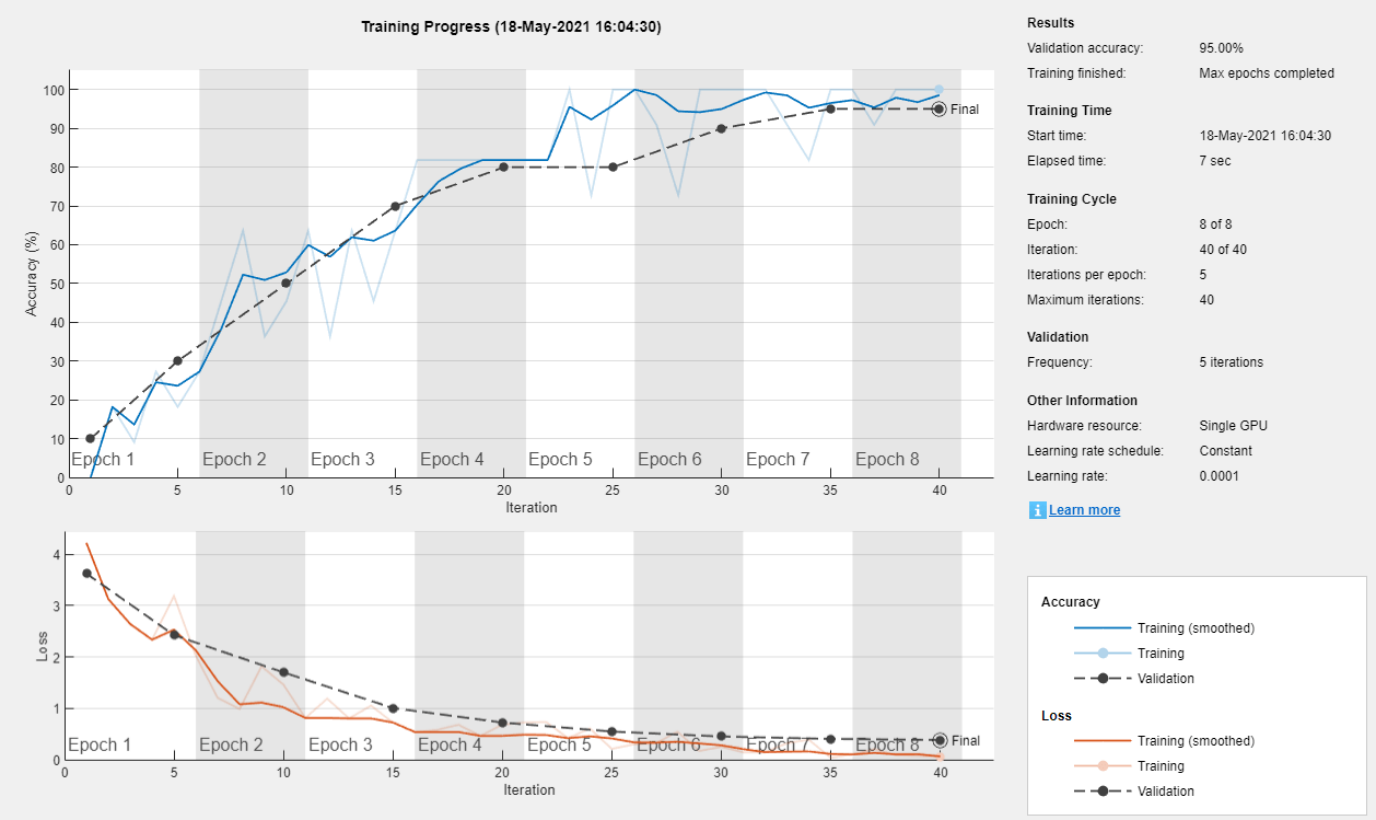
- Set the initial learn rate to a small value to slow down learning in the transferred layers.
- Specify validation frequency so that the accuracy on the validation data is calculated once every epoch.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data set. For transfer learning, you do not need to train for as many epochs.
- Specify the mini-batch size, that is, how many images to use in each iteration. To ensure the whole data set is used during each epoch, set the mini-batch size to evenly divide the number of training samples.

For this example, set **InitialLearnRate** to `0.0001`, **ValidationFrequency** to `5`, and **MaxEpochs** to `8`. As there are 55 observations, set **MiniBatchSize** to `11` to divide the training data evenly and ensure you use the whole training data set during each epoch. For more information on selecting training options, see `trainingOptions`.



To train the network with the specified training options, click **Close** and then click **Train**.

Deep Network Designer allows you to visualize and monitor training progress. You can then edit the training options and retrain the network, if required.



Export Results and Generate MATLAB Code

To export the network architecture with the trained weights, on the **Training** tab, select **Export** > **Export Trained Network and Results**. Deep Network Designer exports the trained network as the variable `trainedNetwork_1` and the training info as the variable `trainInfoStruct_1`.

```
trainInfoStruct_1
```

```
trainInfoStruct_1 = struct with fields:
              TrainingLoss: [4.2191 3.1255 2.6411 2.3362 3.1848 2.0195 1.2013 0.9902 1.8173 1.46
          TrainingAccuracy: [0 18.1818 9.0909 27.2727 18.1818 27.2727 45.4545 63.6364 36.3636 45
            ValidationLoss: [3.6204 NaN NaN NaN 2.4244 NaN NaN NaN NaN 1.7002 NaN NaN NaN NaN 1
        ValidationAccuracy: [10.0000 NaN NaN NaN 30 NaN NaN NaN NaN 50 NaN NaN NaN NaN 70 NaN Na
            BaseLearnRate: [1.0000e-04 1.0000e-04 1.0000e-04 1.0000e-04 1.0000e-04 1.0000e-04 1
      FinalValidationLoss: 0.3854
  FinalValidationAccuracy: 95
   OutputNetworkIteration: 40
```

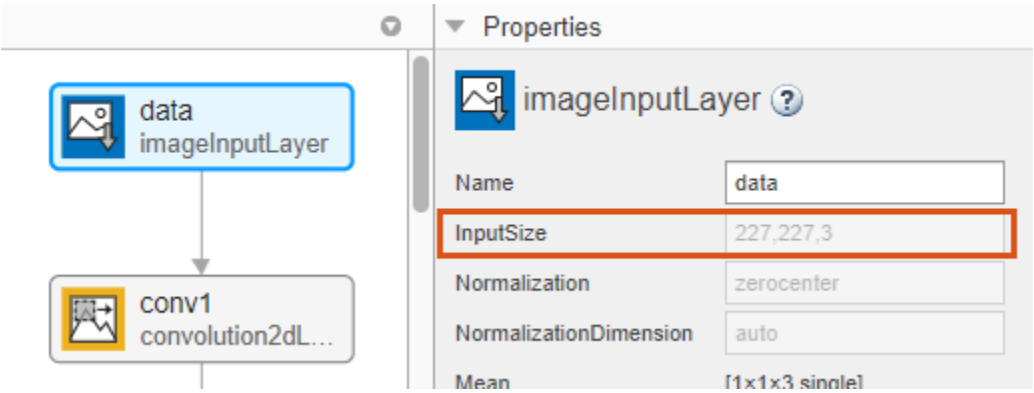
You can also generate MATLAB code, which recreates the network and the training options used. On the **Training** tab, select **Export** > **Generate Code for Training**. Examine the MATLAB code to learn how to programmatically prepare the data for training, create the network architecture, and train the network.

Classify New Image

Load a new image to classify using the trained network.

```
I = imread("MerchDataTest.jpg");
```

Deep Network Designer resizes the images during training to match the network input size. To view the network input size, go to the **Designer** pane and select the `imageInputLayer` (first layer). This network has an input size of 227-by-227.



Resize the test image to match the network input size.

```
I = imresize(I, [227 227]);
```

Classify the test image using the trained network.

```
[YPred,probs] = classify(trainedNetwork_1,I);
imshow(I)
label = YPred;
title(string(label) + ", " + num2str(100*max(probs),3) + "%");
```

See Also

Deep Network Designer

Related Examples

- “Build Networks with Deep Network Designer” on page 2-15
- “Import Data into Deep Network Designer” on page 2-39
- “Generate MATLAB Code from Deep Network Designer” on page 2-69
- “Deep Learning Tips and Tricks” on page 1-67
- “List of Deep Learning Layers” on page 1-21

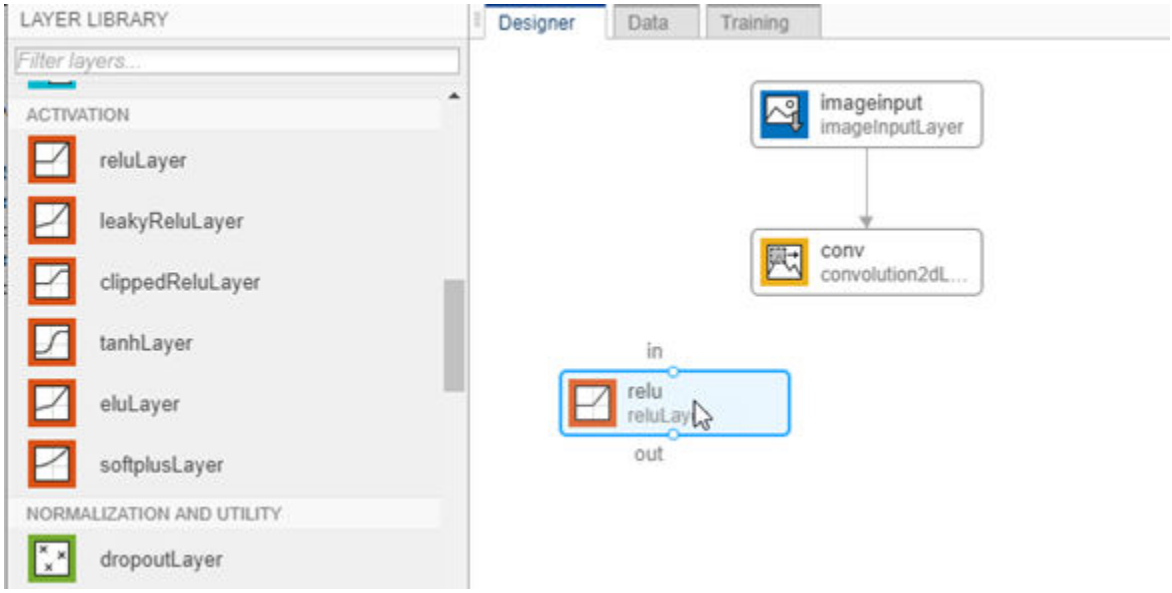
Build Networks with Deep Network Designer

Build and edit deep learning networks interactively using the **Deep Network Designer** app. Using this app, you can import networks or build a network from scratch, view and edit layer properties, combine networks, and generate code to create the network architecture. You can then train your network using Deep Network Designer, or export the network for training at the command line.

You can use Deep Network Designer for a range of network construction tasks:

- “Transfer Learning” on page 2-16
- “Image Classification” on page 2-18
- “Sequence Classification” on page 2-20
- “Numeric Data Classification” on page 2-21
- “Convert Classification Network into Regression Network” on page 2-23
- “Multiple-Input and Multiple-Output Networks” on page 2-23
- “Deep Networks” on page 2-25
- “Advanced Deep Learning Applications” on page 2-26
- “dlnetwork for Custom Training Loops” on page 2-29

Assemble a network by dragging blocks from the **Layer Library** and connecting them. To quickly search for layers, use the **Filter layers** search box in the **Layer Library** pane.

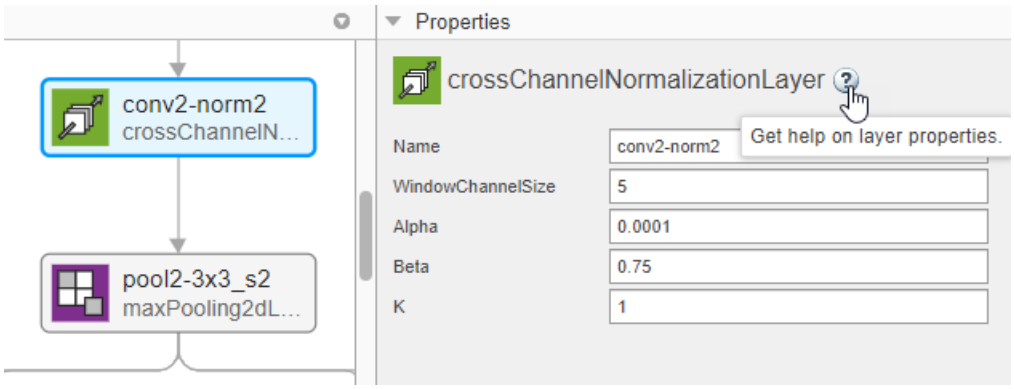


You can add layers from the workspace to the network in the **Designer** pane.

1. Click **New**.
2. Pause on **From Workspace** and click **Import**.
3. Choose the layers or network to import and click **OK**.
4. Click **Add** to add the layers or network to the **Designer** pane.

You can also load pretrained networks by clicking **New** and selecting them from the start page.

To view and edit layer properties, select a layer. Click the help icon next to the layer name for information on the layer properties.



For information on all layer properties, click the layer name in the table on the “List of Deep Learning Layers” on page 1-21 page.

Once you have constructed your network, you can analyze it to check for errors. For more information, see “Check Network” on page 2-29.

Transfer Learning

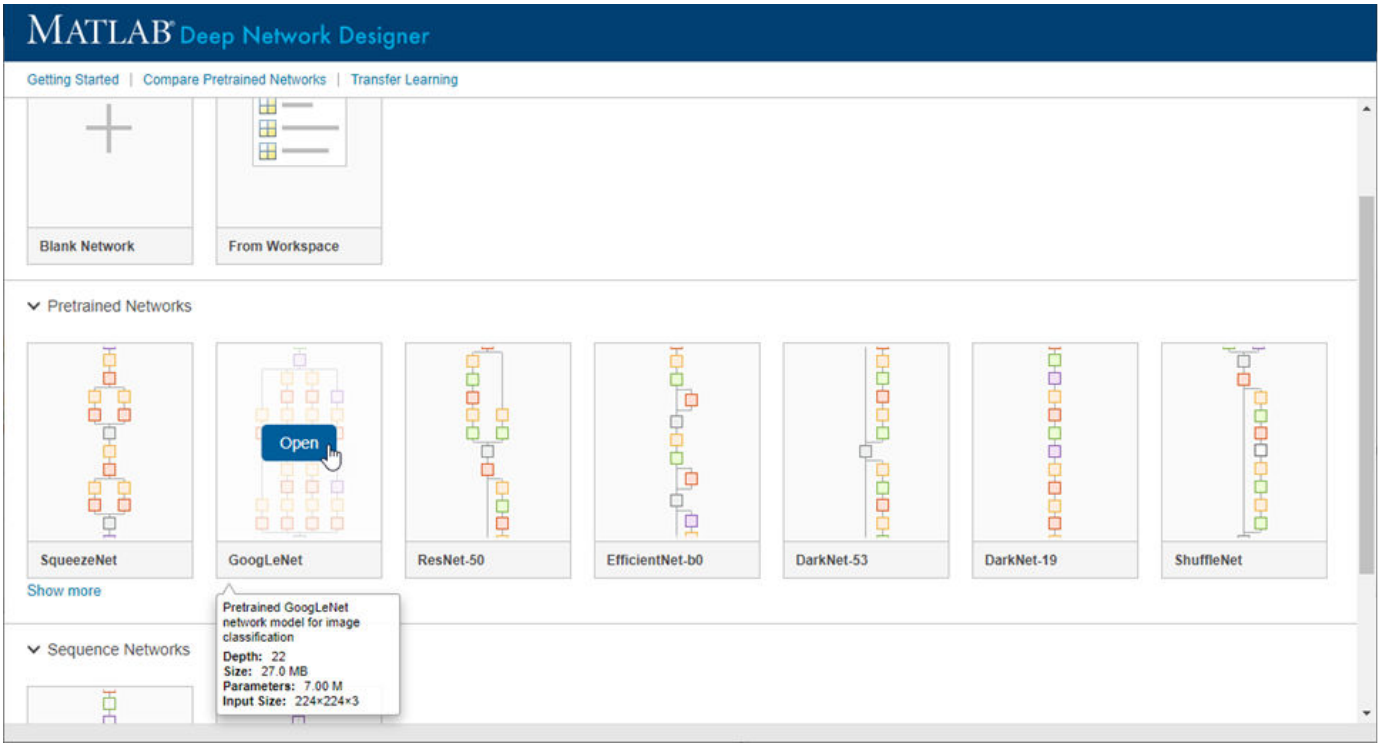
Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Deep Network Designer has a selection of pretrained networks suitable for transfer learning with image data.

Load Pretrained Network

Open the app and select a pretrained network. You can also load a pretrained network by selecting the **Designer** tab and clicking **New**. If you need to download the network, pause on the network and click **Install** to open the Add-On Explorer.

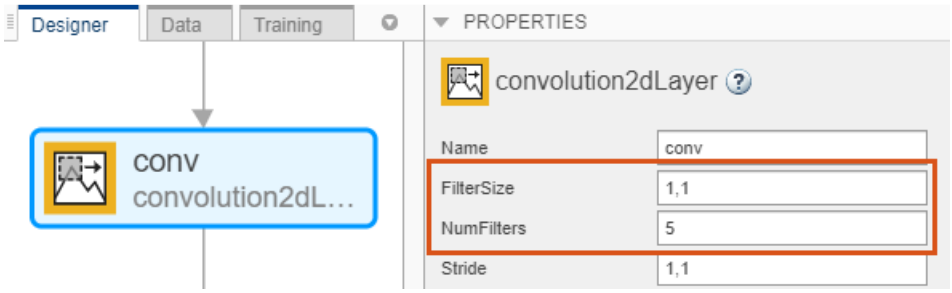
Tip To get started, try choosing one of the faster networks, such as SqueezeNet or GoogLeNet. Once you gain an understanding of which settings work well, try a more accurate network, such as Inception-v3 or a ResNet, and see if that improves your results. For more information on selecting a pretrained network, see “Pretrained Deep Neural Networks” on page 1-8.



Adapt Pretrained Network

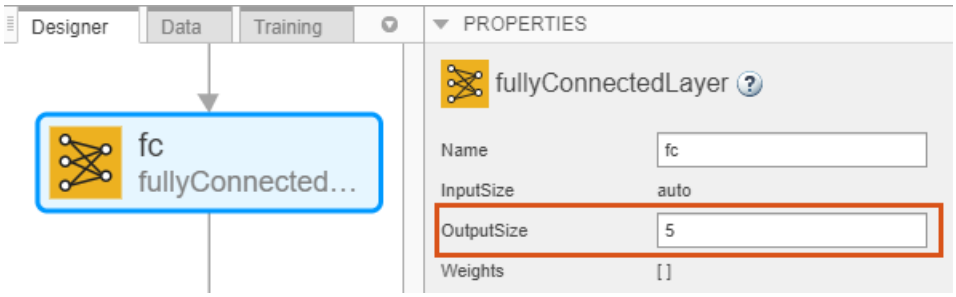
To prepare the network for transfer learning, replace the last learnable layer and the final classification layer.

- If the last learnable layer is a 2-D convolutional layer (for example, the `'conv10'` layer in SqueezeNet):
  - Drag a new **convolution2dLayer** onto the canvas. Set the **NumFilters** property to the new number of classes and **FilterSize** to `1,1`.
  - Change the learning rates so that learning is faster in the new layer than in the transferred layers by increasing the **WeightLearnRateFactor** and **BiasLearnRateFactor** values.
  - Delete the last **convolution2dLayer** and connect your new layer instead.

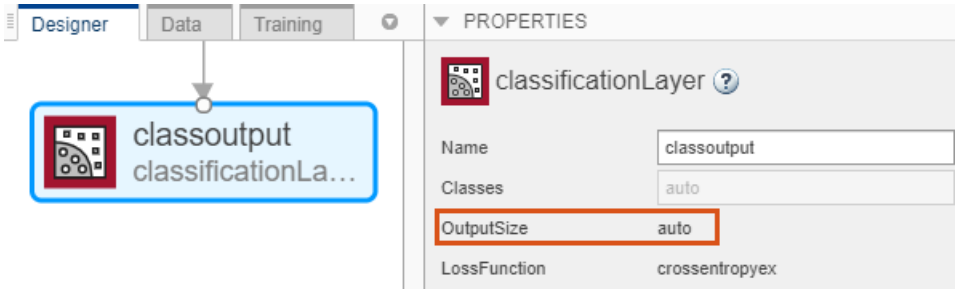


- If the last learnable layer is a fully connected layer (most pretrained networks, for example, GoogLeNet):
  - Drag a new **fullyConnectedLayer** onto the canvas and set the **OutputSize** property to the new number of classes.

- Change the learning rates so that learning is faster in the new layer than in the transferred layers by increasing the **WeightLearnRateFactor** and **BiasLearnRateFactor** values.
- Delete the last **fullyConnectedLayer** and connect your new layer instead.



Next, delete the classification output layer. Then, drag a new **classificationLayer** onto the canvas and connect it instead. The default settings for the output layer mean it will learn the number of classes during training.



To check that the network is ready for training, on the **Designer** tab, click **Analyze**.

For an example showing how to retrain a pretrained network to classify new images, see “Transfer Learning with Deep Network Designer” on page 2-2.

You can also use pretrained networks and transfer learning for regression tasks. For more information, see “Convert Classification Network into Regression Network” on page 2-23.

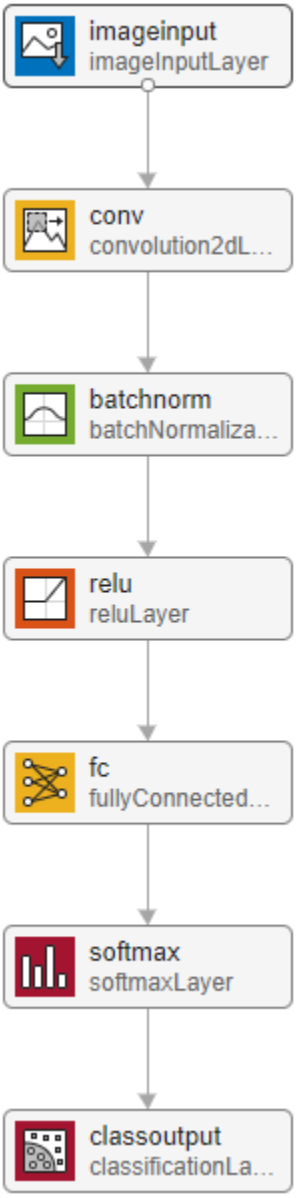
Image Classification

You can build an image classification network using Deep Network Designer by dragging layers from the **Layer Library** and connecting them. You can also create the network at the command line and then import the network into Deep Network Designer.

For example, create a network to train for image classification on a data set of 28-by-28 images divided into 10 classes.

```
inputSize = [28 28 1];
numClasses = 10;

layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
```

```
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

deepNetworkDesigner(layers)
```

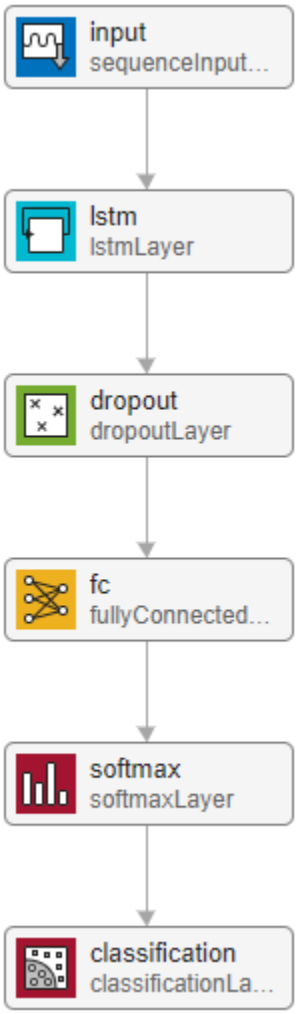


To adapt this network to your own data, set the **InputSize** of the image input layer to match your image input size and set the **OutputSize** of the fully connected layer to the number of classes in your data. For more complex classification tasks, create a deeper network. For more information, see “Deep Networks” on page 2-25.

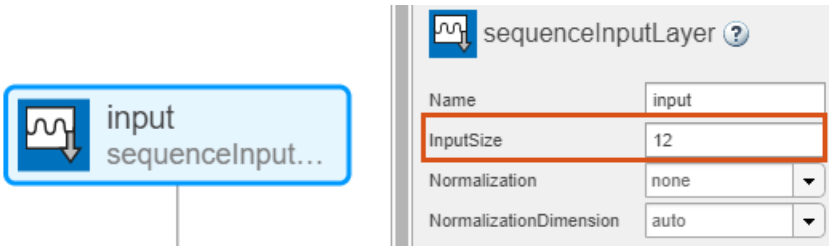
For an example showing how to create and train an image classification network, see “Create Simple Image Classification Network Using Deep Network Designer”.

Sequence Classification

You can use Deep Network Designer to build a sequence network from scratch, or you can use one of the prebuilt untrained networks from the start page. Open the Deep Network Designer start page. Pause on **Sequence-to-Label** and click **Open**. Doing so opens a prebuilt network suitable for sequence classification problems.



You can adapt this sequence network for training with your data. Suppose you have data with 12 features and 9 classes. To adapt this network, select **sequenceInputLayer** and set the **InputSize** to 12.



Then, select the **fullyConnectedLayer** and set the **OutputSize** to 9, the number of classes.

The network is now ready to train. To train the network in Deep Network Designer, create a `CombinedDatastore` containing the predictors and responses. For more information, see “Import Data into Deep Network Designer” on page 2-39. For an example showing how to create a combined datastore and train a sequence-to-sequence regression network using Deep Network Designer, see “Train Network for Time Series Forecasting Using Deep Network Designer” on page 2-57. For an example showing how to export a network built in Deep Network Designer and train using command line functions, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-51.

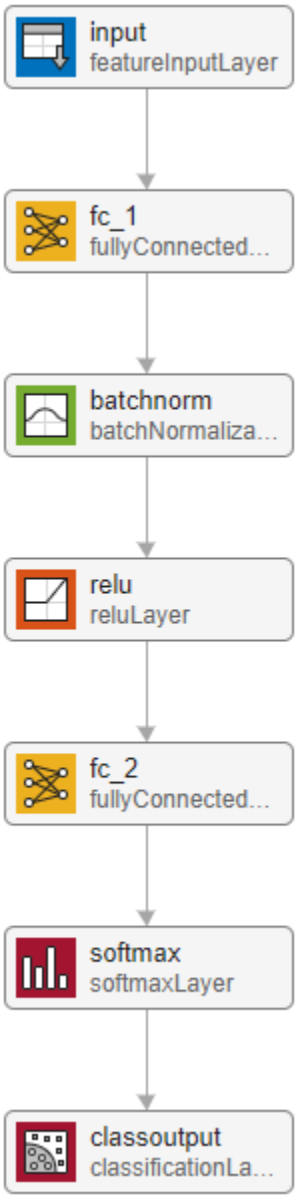
Numeric Data Classification

If you have a data set of numeric features (for example, a collection of numeric data without spatial or time dimensions), then you can train a deep learning network using a feature input layer. For more information about the feature input layer, see `featureInputLayer`.

You can construct a suitable network using Deep Network Designer, or you can create the network at the command line and import the network into Deep Network Designer.

For example, create a network for numeric data with 10 classes, where each observation consists of 20 features.

```
inputSize = 20;
numClasses = 10;

layers = [
featureInputLayer(inputSize,'Normalization','zscore')
fullyConnectedLayer(50)
batchNormalizationLayer
reluLayer
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer];

deepNetworkDesigner(layers)
```



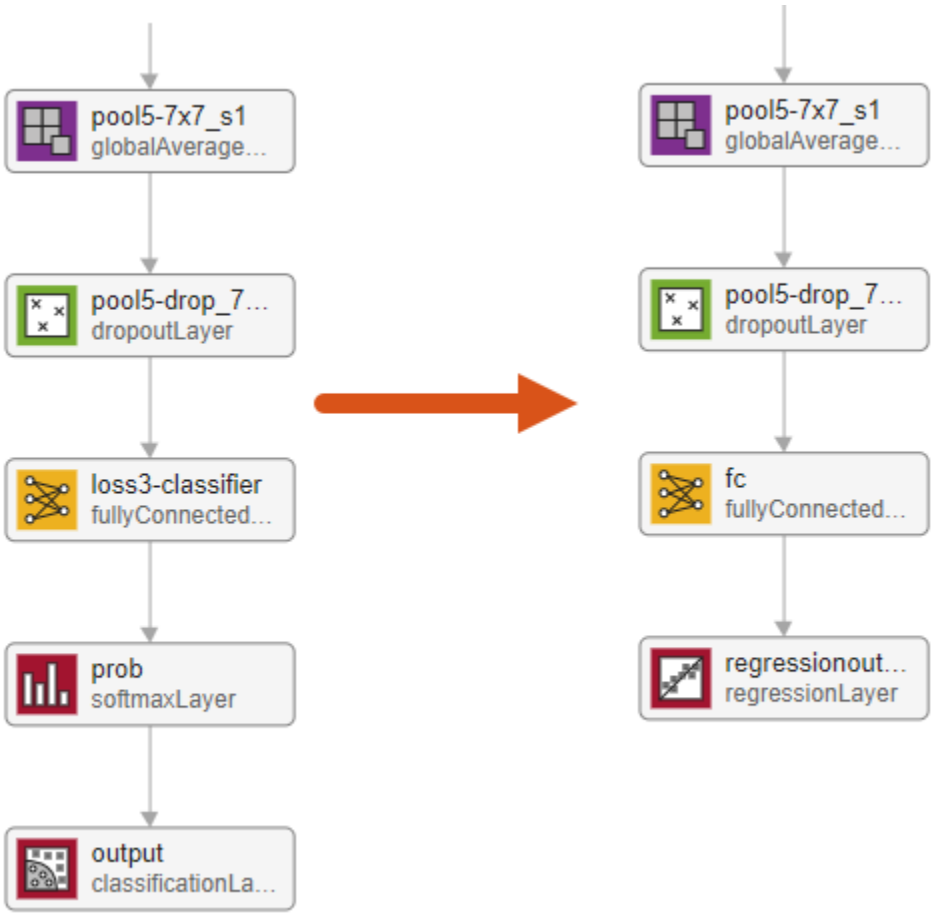
To adapt this network to your own data, set the **InputSize** of the feature input layer to match the number of features in your data and set the **OutputSize** of the fully connected layer to the number of classes in your data. For more complex classification tasks, create a deeper network. For more information, see “Deep Networks” on page 2-25.

To train a network in Deep Network Designer using data in a table, you must first convert your data into a suitable datastore. For example, start by converting your table into arrays containing the predictors and responses. Then, convert the arrays into `arrayDatastore` objects. Finally, combine the predictor and response array datastores into a `CombinedDatastore` object. You can then use the combined datastore to train in Deep Network Designer. For more information, see “Import Data into Deep Network Designer” on page 2-39. You can also train with tabular data and the `trainNetwork` function by exporting the network to the workspace.

Convert Classification Network into Regression Network

You can convert a classification network into a regression network by replacing the final layers of the network. Conversion is useful when you want to take a pretrained classification network and retrain it for regression tasks.

For example, suppose you have a GoogLeNet pretrained network. To convert this network into a regression network with a single response, replace the final fully connected layer, the softmax layer, and the classification output layer with a fully connected layer with **OutputSize** set to 1 (the number of responses) and a regression layer.



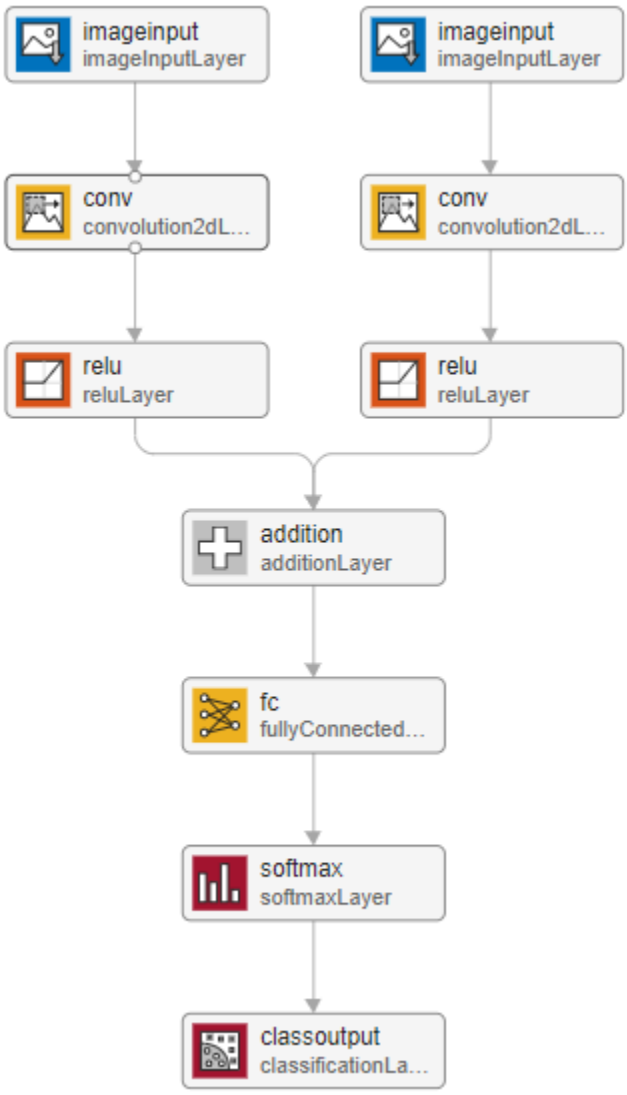
If your output has multiple responses, change the **OutputSize** value of the fully connected layer to the number of responses.

Multiple-Input and Multiple-Output Networks

Multiple Inputs

You can define a network with multiple inputs if the network requires data from multiple sources or in different formats. For example, some networks require image data captured from multiple sensors at different resolutions.

Using Deep Network Designer, you can control the inputs and outputs of each layer. For example, to create a network with multiple image inputs, create two branches, each starting with an image input layer.

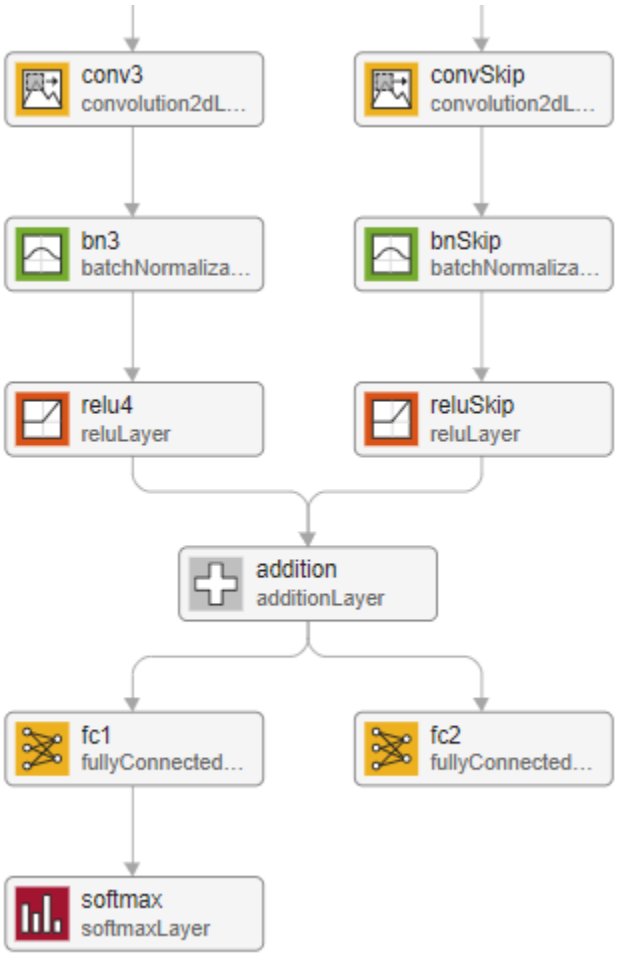


You can train a multi-input network with the same type of input, for example, images from two difference sources, using Deep Network Designer and a datastore object. For networks with data in multiple formats, for example, image and sequence data, train the network using a custom training loop. For more information, see “dlnetwork for Custom Training Loops” on page 2-29.

Multiple Outputs

You can define networks with multiple outputs for tasks requiring multiple responses in different formats, for example, tasks requiring both categorical and numeric output.

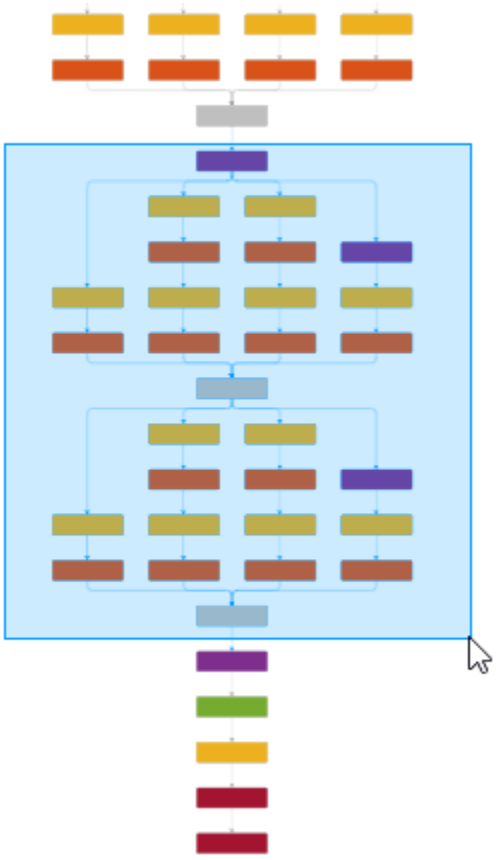
Using Deep Network Designer, you can control the outputs of each layer.



To train a multi-output network, you must use a custom training loop. Custom training loops must use a `dlnetwork` object that does not contain any output layers. For more information, see “dlnetwork for Custom Training Loops” on page 2-29.

Deep Networks

Building large networks can be difficult, you can use Deep Network Designer to speed up construction. You can work with blocks of layers at a time. Select multiple layers, then copy and paste or delete. For example, you can use blocks of layers to create multiple copies of groups of convolution, batch normalization, and ReLU layers.



For trained networks, copying layers also copies the weights and the biases.

You can also copy sub-networks from the workspace to connect up easily using the app. To import a network or layers into the app, click **New > Import from workspace**. Click **Add** to add the layers to the current network.

Advanced Deep Learning Applications

You can use Deep Network Designer to build and train networks for advanced applications, such as computer vision or image processing tasks.

Create Semantic Segmentation Network

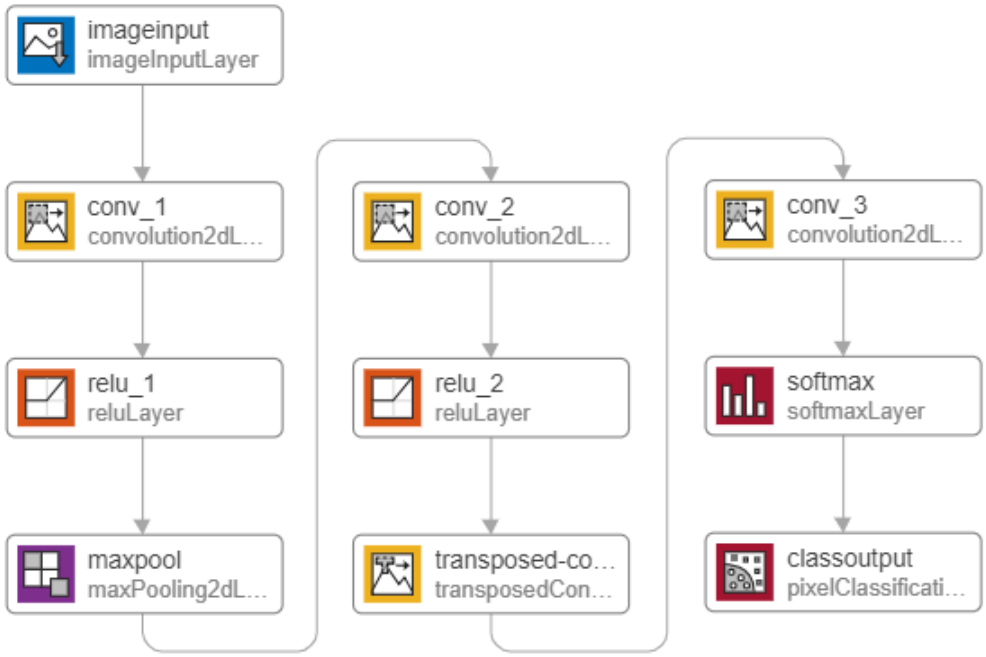
Semantic segmentation describes the process of associating each pixel of an image with a class label. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis.

Create a semantic segmentation network by dragging layers from the **Layer Library** to the **Designer** pane or creating the network at the command-line and importing the network into Deep Network Designer.

For example, create a simple semantic segmentation network based on a downsampling and upsampling design.

```
inputSize = [32 32 1];
```

```
layers = [
    imageInputLayer(inputSize)
    convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
    reluLayer
    maxPooling2dLayer([2,2],'Stride',[2,2])
    convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
    reluLayer
    transposedConv2dLayer([4,4],64,'Stride',[2,2],'Cropping',[1,1,1,1])
    convolution2dLayer([1,1],2)
    softmaxLayer
    pixelClassificationLayer
    ];

deepNetworkDesigner(layers)
```



For more information on constructing and training a semantic segmentation network, see “Train Simple Semantic Segmentation Network in Deep Network Designer” (Computer Vision Toolbox).

Create Image-to-Image Regression Network

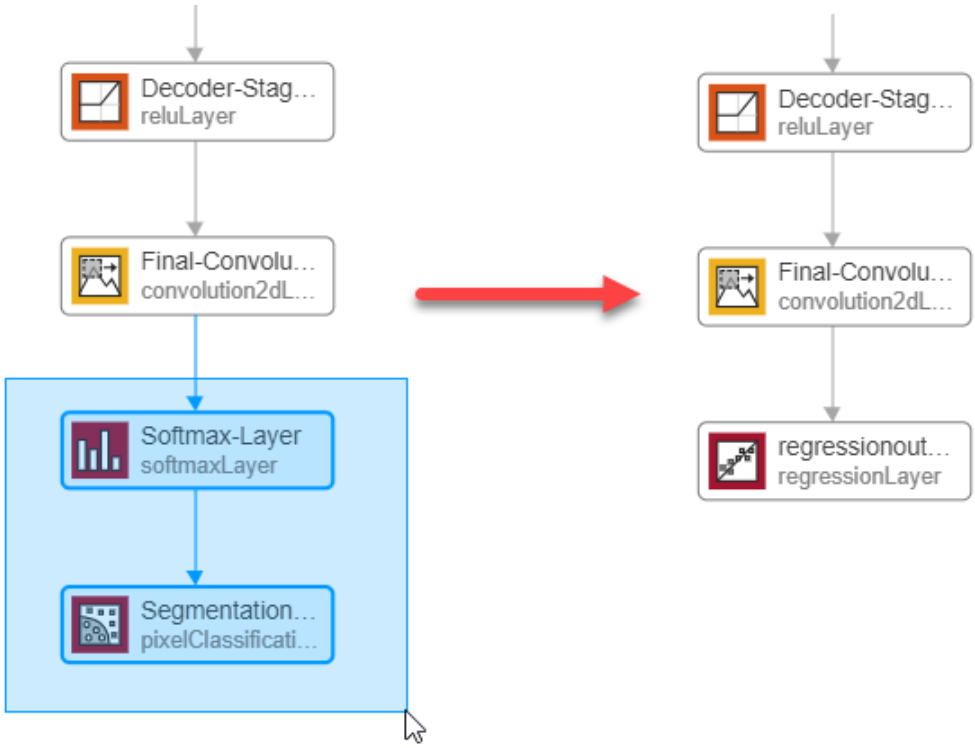
Image-to-image regression involves taking an input image and producing an output image, often of the same size. This type of network is useful for super-resolution, colourization, or image deblurring.

You can create image-to-image regression networks using Deep Network Designer. For example, create a simple network architecture suitable for image-to-image regression using the `unetLayers` function from Computer Vision Toolbox™. This function provides a network suitable for semantic segmentation, that can be easily adapted for image-to-image regression.

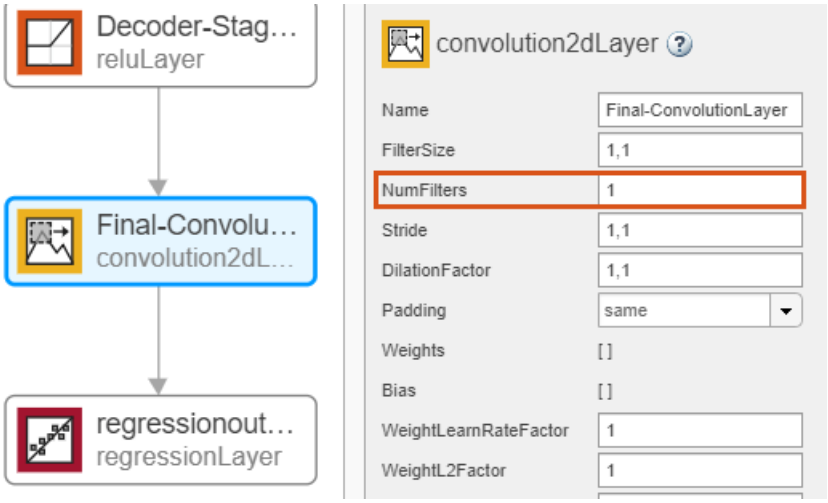
Create a network with input size 28-by-28-by-1 pixels.

```
layers = unetLayers([28,28,1],2,'encoderDepth',2);
deepNetworkDesigner(layers);
```

In the **Designer** pane, replace the softmax and pixel classification layers with a regression layer from the **Layer Library**.



Select the final convolutional layer and set the **NumFilters** property to 1.



For more information on constructing and training an image-to-image regression network, see “Image-to-Image Regression in Deep Network Designer” on page 2-72.

dlnetwork for Custom Training Loops

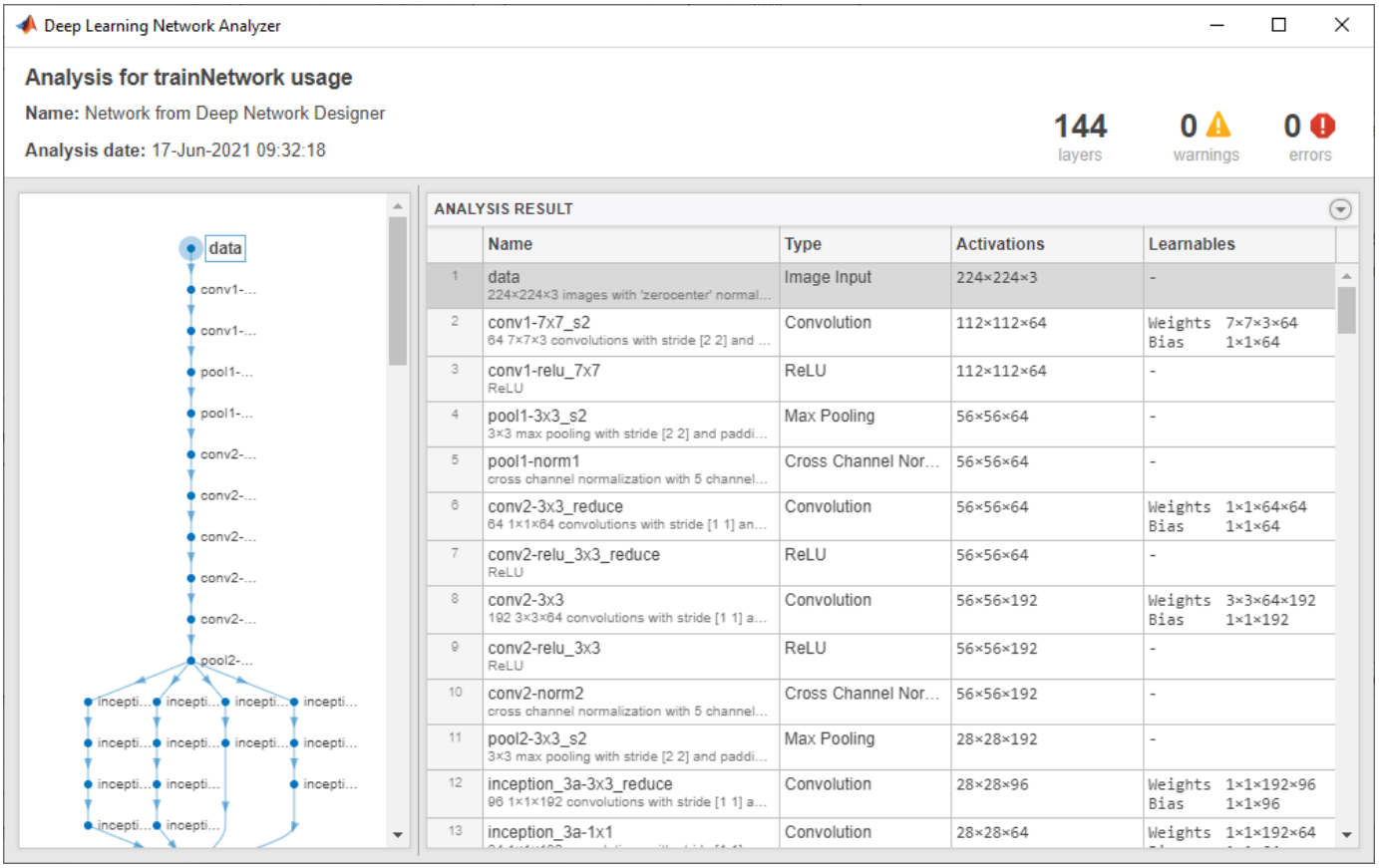
You can build and analyze `dlnetwork` objects using Deep Network Designer. A `dlnetwork` object enables support for custom training loops using automatic differentiation. Use custom training loops when the built-in training options do not provide the training options that you need for your task.

To check that your network is ready for training using a custom training loop, click **Analyze** > **Analyze for dlnetwork**. For more information, see “Check Network” on page 2-29.

Training with a custom training loop is not supported in Deep Network Designer. To train your network using a custom training loop, first export the network to the workspace and convert it to a `dlnetwork` object. You can then train the network using the `dlnetwork` object and a custom training loop. For more information, see “Train Network Using Custom Training Loop” on page 18-225.

Check Network

To check your network and examine the layers in further detail, on the **Designer** tab, click **Analyze**. Investigate problems and examine the layer properties to resolve size mismatches in the network. Return to Deep Network Designer to edit layers, then check the results by clicking **Analyze** again. If Deep Learning Network Analyzer reports zero errors, then the edited network is ready for training.



You can also analyze networks for custom training workflows. Click **Analyze > Analyze for dlnetwork** to analyze the network for usage with `dlnetwork` objects. For example, the Network Analyzer checks that the layer graph does not have any output layers.

See Also

Deep Network Designer

Related Examples

- “Import Data into Deep Network Designer” on page 2-39
- “Train Networks Using Deep Network Designer” on page 2-31
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Network for Time Series Forecasting Using Deep Network Designer” on page 2-57
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Train Networks Using Deep Network Designer

The **Deep Network Designer** app lets you build and train deep neural networks. Deep Network Designer supports `trainNetwork` training using image data or datastore objects. You can also export your untrained network for training at the command line, for example, to train your network using custom training loops.

To train a network, follow these steps:

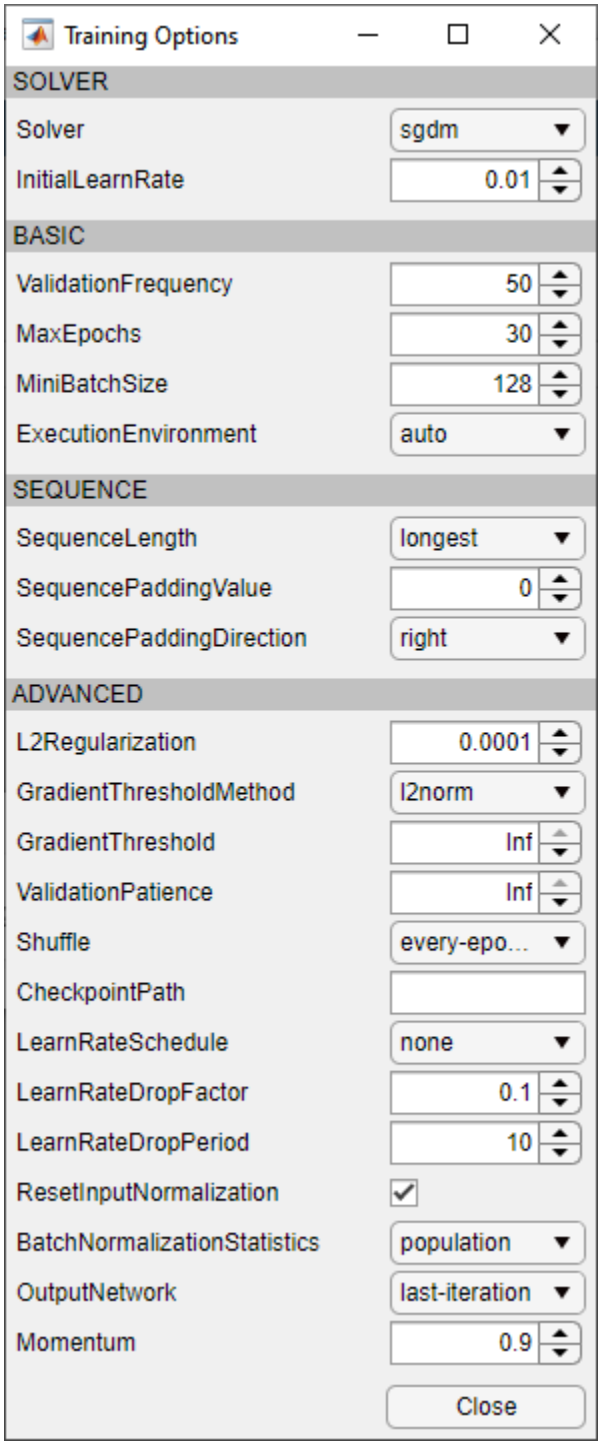
1. Create network
2. Import data
3. Select training options
4. Train network
5. Export network

You can build a network interactively using Deep Network Designer, or import a network from the workspace. You can also select a pretrained network from the Deep Network Designer start page for transfer learning. For more information, see “Build Networks with Deep Network Designer” on page 2-15.

To train a deep learning model, you must have a suitable network and training data. To import image data from a folder containing a subfolder of images for each class, or from an `imageDatastore` object, on the **Data** tab, click **Import Data > Import Image Data**. To import any datastore, on the **Data** tab, click **Import Data > Import Datastore**. After import, Deep Network Designer displays a preview of the imported data so that you can check that the data is as expected prior to training. For more information, see “Import Data into Deep Network Designer” on page 2-39.

Select Training Options

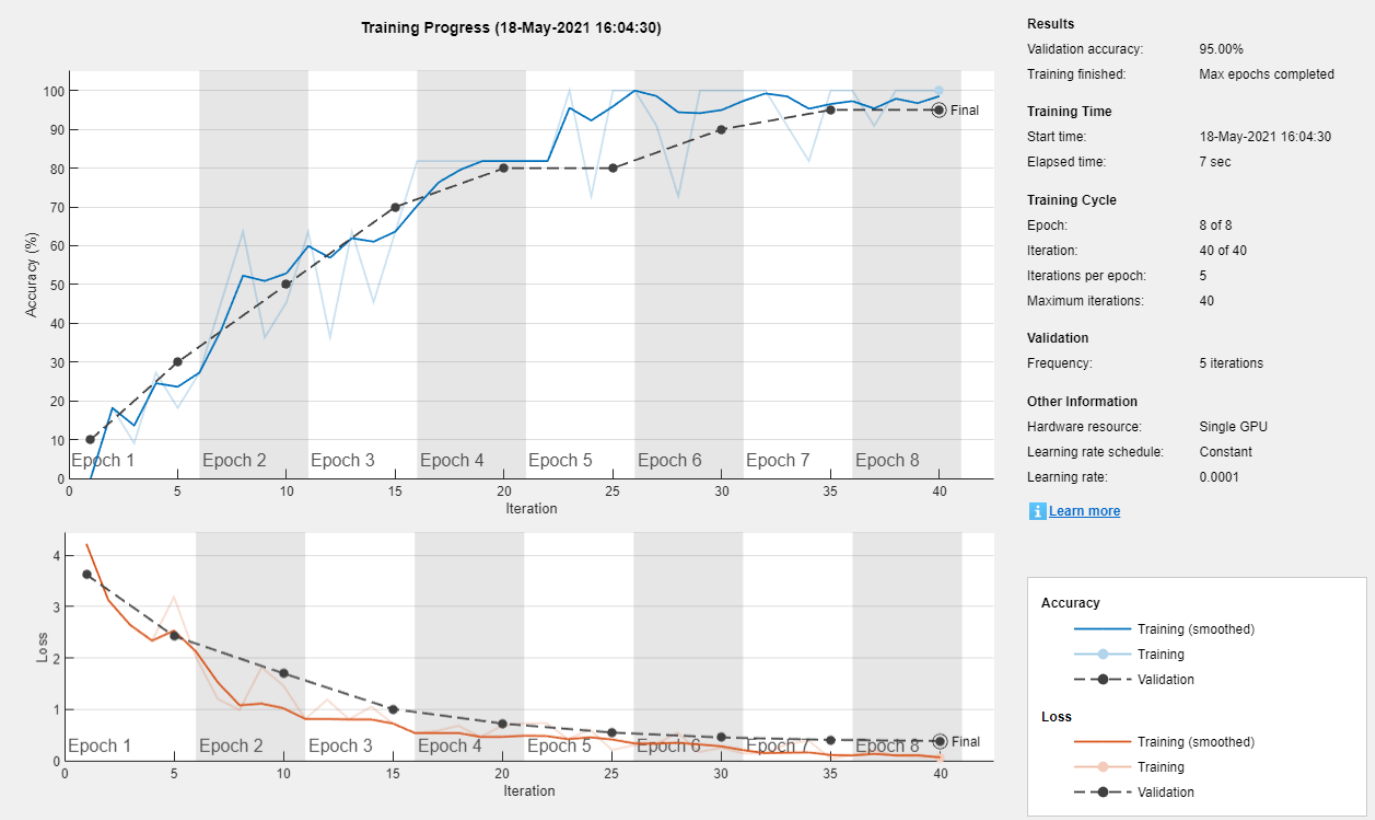
Once you have your network and data, the next step is to select the training options. On the **Training** tab, click **Training Options**. If you do not know which training options to use, try training with the default settings and then adjusting them to suit your network and data. For example, try adjusting the initial learning rate, or train for longer by increasing the number of epochs. For information about techniques for improving the accuracy of deep learning networks, see “Deep Learning Tips and Tricks” on page 1-67. For more information about the training options, see `trainingOptions`.



Train Network

After you select your training options, train the network by clicking **Train**. The Deep Network Designer app displays an animated plot of the training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress. The plot

has a stop button in the top-right corner. Click the button to stop training and return the current state of the network. For more information on the training progress plot, see “Monitor Deep Learning Training Progress”.



You can train a variety of networks using Deep Network Designer. For example, image classification or regression networks, sequence networks, numeric data networks, semantic segmentation networks, and image-to-image regression networks. In Deep Network Designer, you can train a network using the `trainNetwork` function on any data that you can express as a datastore object. The following examples show how to build and train a network using Deep Network Designer.

- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Network for Time Series Forecasting Using Deep Network Designer” on page 2-57
- “Train Simple Semantic Segmentation Network in Deep Network Designer” on page 8-148
- “Image-to-Image Regression in Deep Network Designer” on page 2-72

Once training is complete, on the **Training** tab, click **Export** to export your trained network and results to the workspace.

Deep Network Designer does not support training using custom training loops. To train your network using a custom training loop, first export the network to the workspace and convert it to a `dlnetwork` object. You can then train the network using the `dlnetwork` object and a custom training loop. For more information, see “Train Network Using Custom Training Loop” on page 18-225.

Next Steps

You can learn how to build and train your network using command line functions by clicking **Export > Generate Code for Training** and examining the generated live script. You can also use the generated script as a starting point to create deep learning experiments that sweep through a range of hyperparameter values or use Bayesian optimization to find optimal training options. For an example showing how to use **Experiment Manager** to tune the hyperparameters of a network trained in Deep Network Designer, see “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager” on page 2-79.

See Also

Deep Network Designer | Experiment Manager

Related Examples

- “Build Networks with Deep Network Designer” on page 2-15
- “Import Data into Deep Network Designer” on page 2-39
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Train Network for Time Series Forecasting Using Deep Network Designer” on page 2-57
- “Image-to-Image Regression in Deep Network Designer” on page 2-72
- “Generate MATLAB Code from Deep Network Designer” on page 2-69

Import Custom Layer into Deep Network Designer

This example shows how to import a custom classification output layer with the sum of squares error (SSE) loss and add it to a pretrained network in Deep Network Designer.

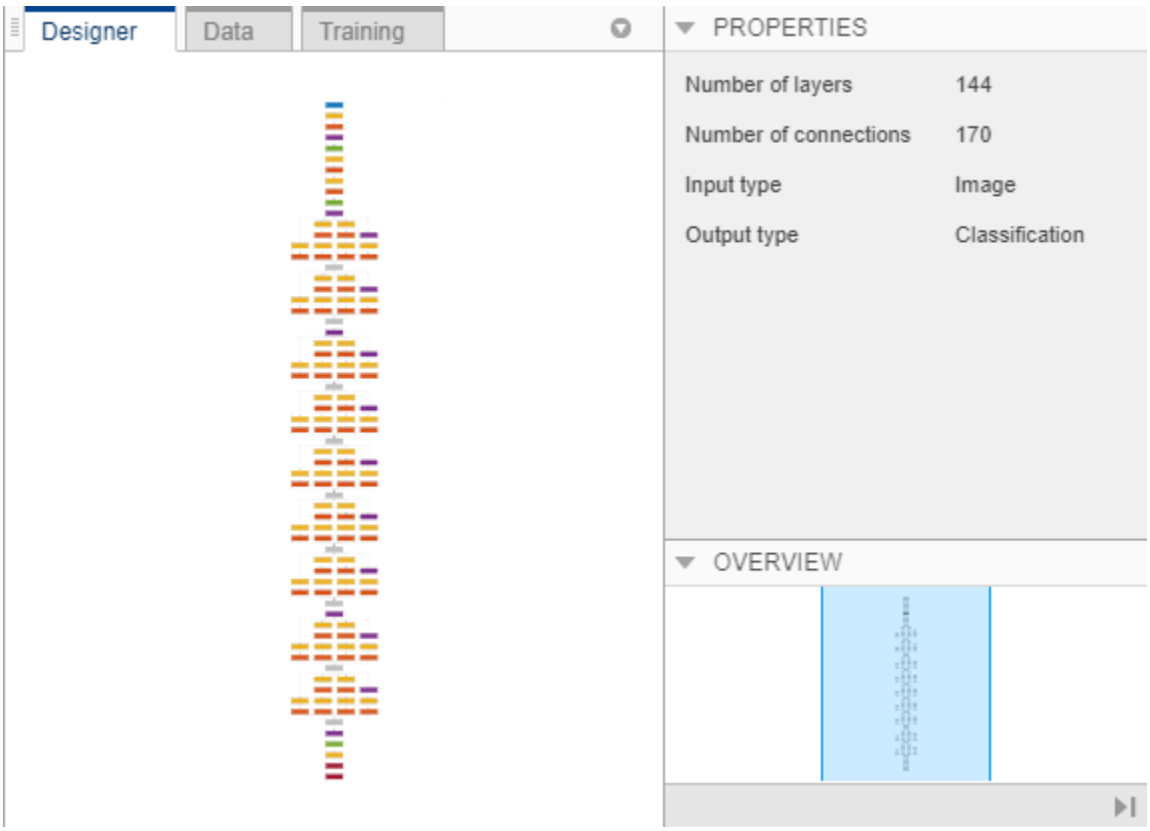
Define a custom classification output layer. To create this layer, save the file `sseClassificationLayer.m` in the current folder. For more information on constructing this layer, see “Define Custom Classification Output Layer” on page 18-91.

Create an instance of the layer.

```
sseClassificationLayer = sseClassificationLayer('sse');
```

Open Deep Network Designer with a pretrained GoogLeNet network.

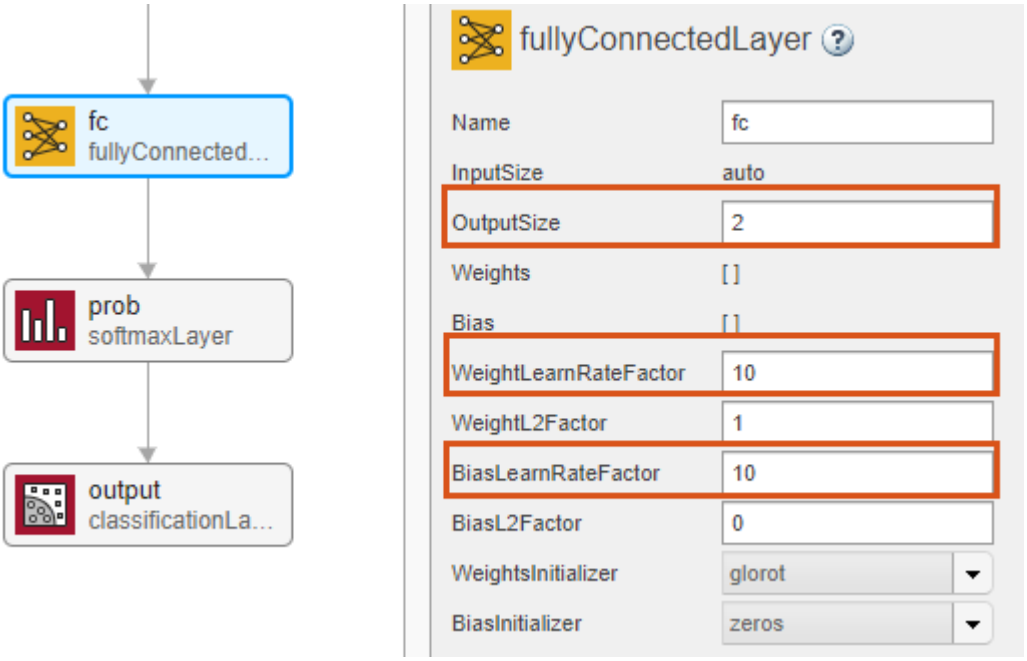
```
deepNetworkDesigner(googlenet);
```



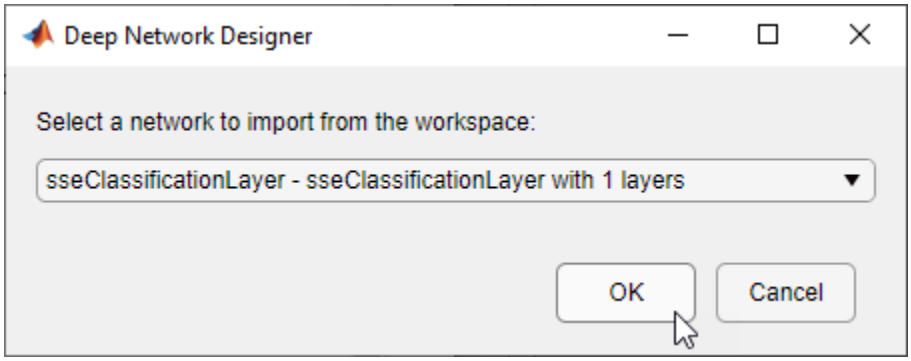
To adapt a pretrained network, replace the last learnable layer and the final classification layer with new layers adapted to the new data set. In GoogLeNet, these layers have the names `'loss3-classifier'` and `'output'`, respectively.

In the **Designer** pane, drag a new `fullyConnectedLayer` from the **Layer Library** onto the canvas. Set `OutputSize` to the new number of classes, in this example, `2`.

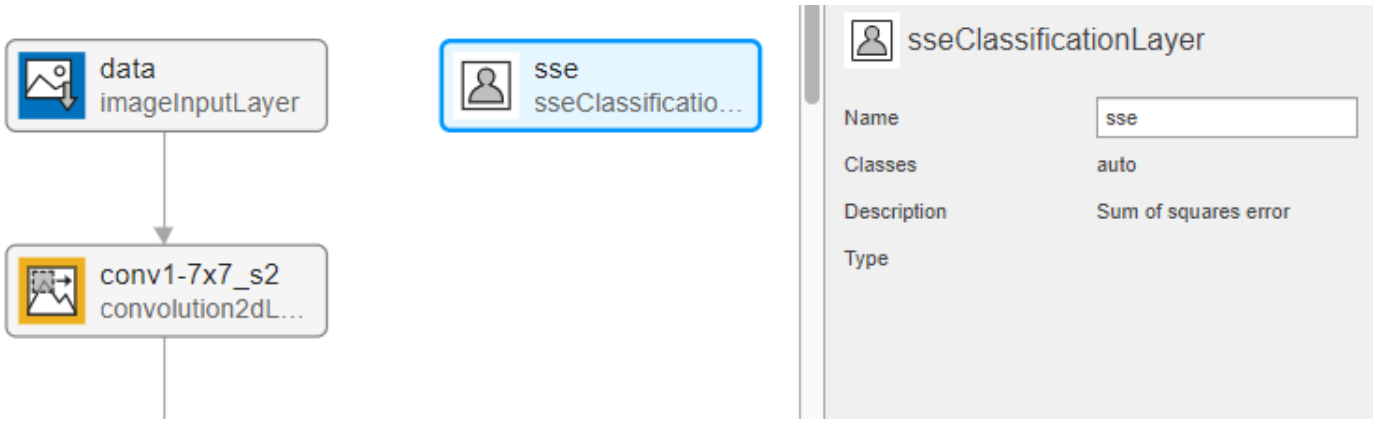
Edit learning rates to learn faster in the new layers than in the transferred layers. Set `WeightLearnRateFactor` and `BiasLearnRateFactor` to `10`. Delete the last fully connected layer and connect your new layer instead.



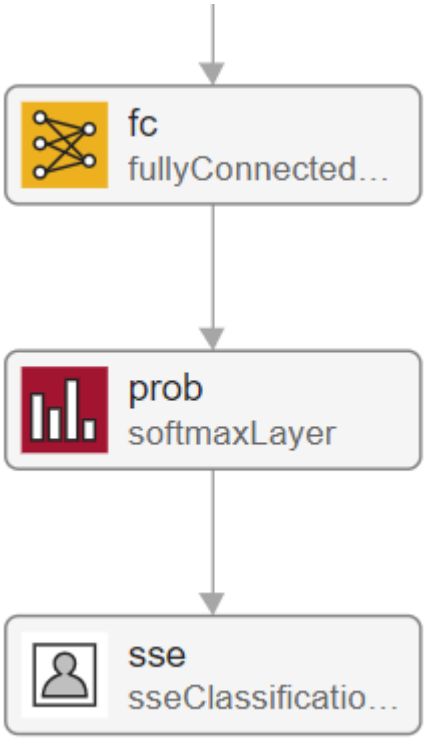
Next, replace the output layer with your custom classification output layer. Click **New** in the **Designer** pane. Pause on **From Workspace** and click **Import**. To import the custom classification layer, select `sseClassificationLayer` and click **OK**.



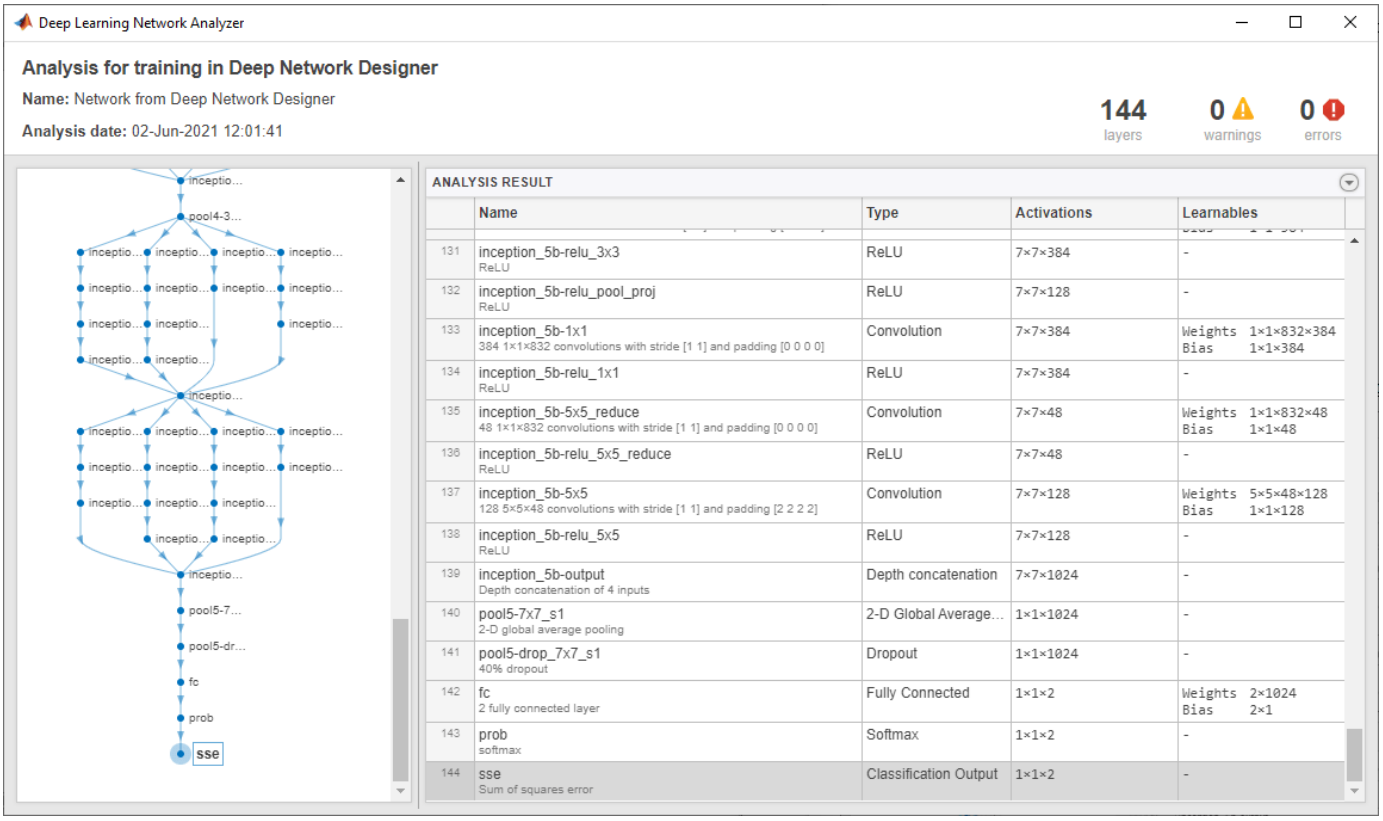
Add the layer to the current GoogLeNet pretrained network by clicking **Add**. The app adds the custom layer to the top of the **Designer** pane. To see the new layer, zoom-in using a mouse or click **Zoom in**.



Drag the custom layer to the bottom of the **Designer** pane. Replace the output layer with the new classification output layer and connect the new layer.



Check your network by clicking **Analyze**. The network is ready for training if Deep Learning Network Analyzer reports zero errors.



After you construct your network, you are ready to import data and train. For more information on importing data and training in Deep Network Designer, see “Transfer Learning with Deep Network Designer” on page 2-2.

See Also

Deep Network Designer

Related Examples

- “Build Networks with Deep Network Designer” on page 2-15
- “Import Data into Deep Network Designer” on page 2-39
- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-51
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-69

Import Data into Deep Network Designer

You can import and visualize training and validation data in **Deep Network Designer**. Using this app you can:

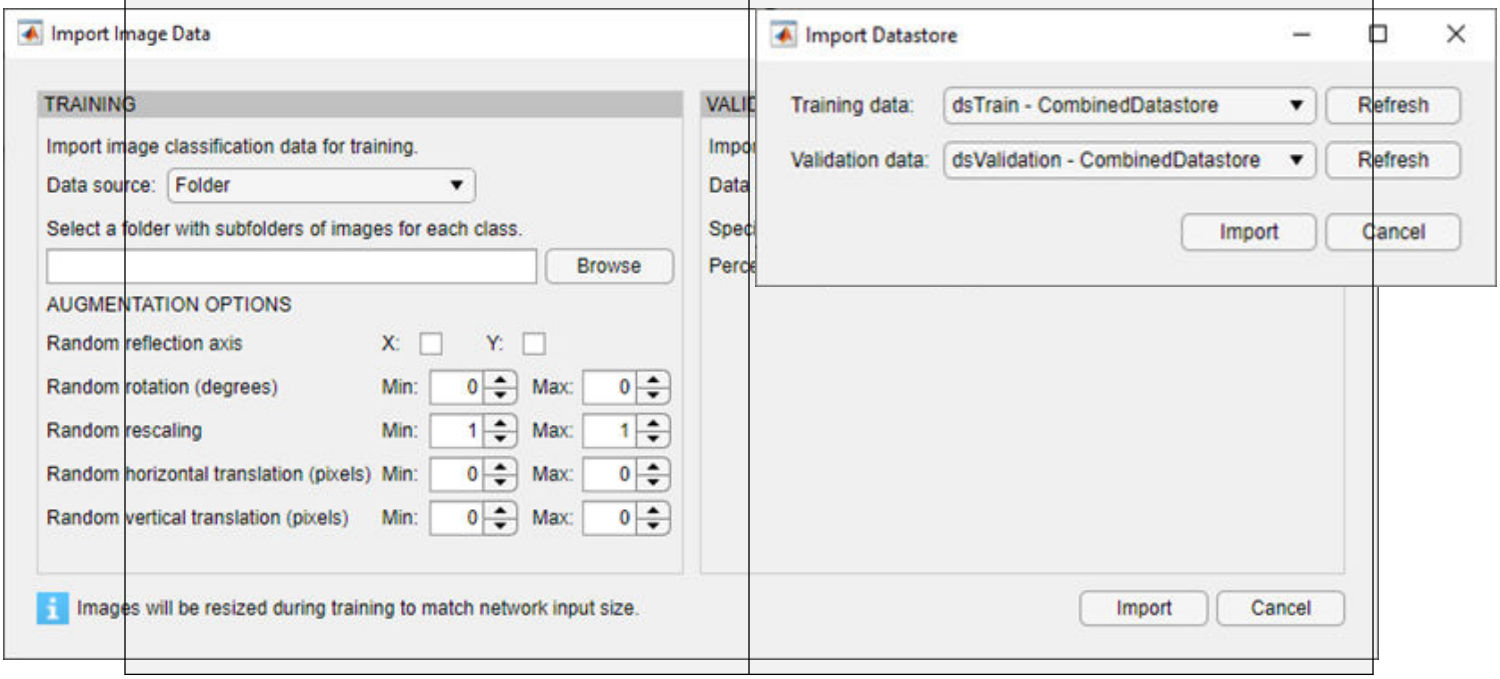
- Import datastore objects for training. After import, Deep Network Designer displays a preview of the data. For more information, see “Import Data” on page 2-39.
- Import training data for image classification problems from an `ImageDatastore` object or a folder containing subfolders of images per class. You can also select built-in options to augment the training images during training. For more information, see “Image Augmentation” on page 2-48.
- Import validation data from a datastore object. For image classification you can also select validation data from a folder containing subfolders of images for each class, or choose to split the validation data from the training data. For more information, see “Validation Data” on page 2-49.

For more information about data sets you can use to get started with deep learning, see “Data Sets for Deep Learning” on page 19-118. For more information on constructing and using datastore objects for deep learning applications, see “Datastores for Deep Learning” on page 19-2.

Import Data

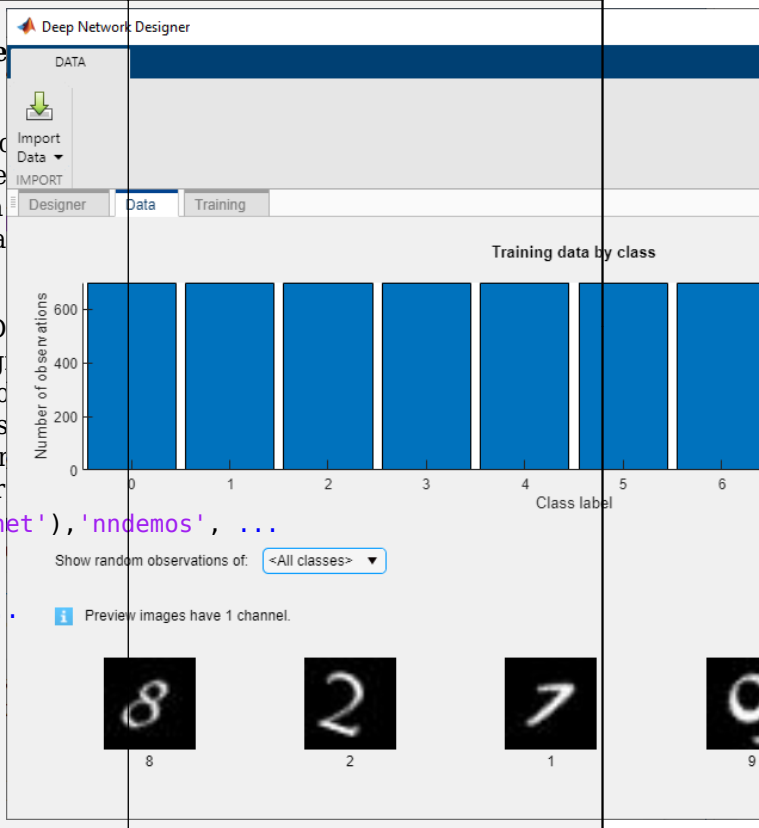
In Deep Network Designer, you can import image classification data from an image datastore or a folder containing subfolders of images from each class. You can also import and train any datastore object that works with the `trainNetwork` function. Select an import method based on the type of datastore you are using.

Import `ImageDatastore` Object	Import Any Other Datastore Object (Not Recommended for `ImageDatastore`)
Select **Import Data** > **Import Image Data**.	Select **Import Data** > **Import Datastore**.



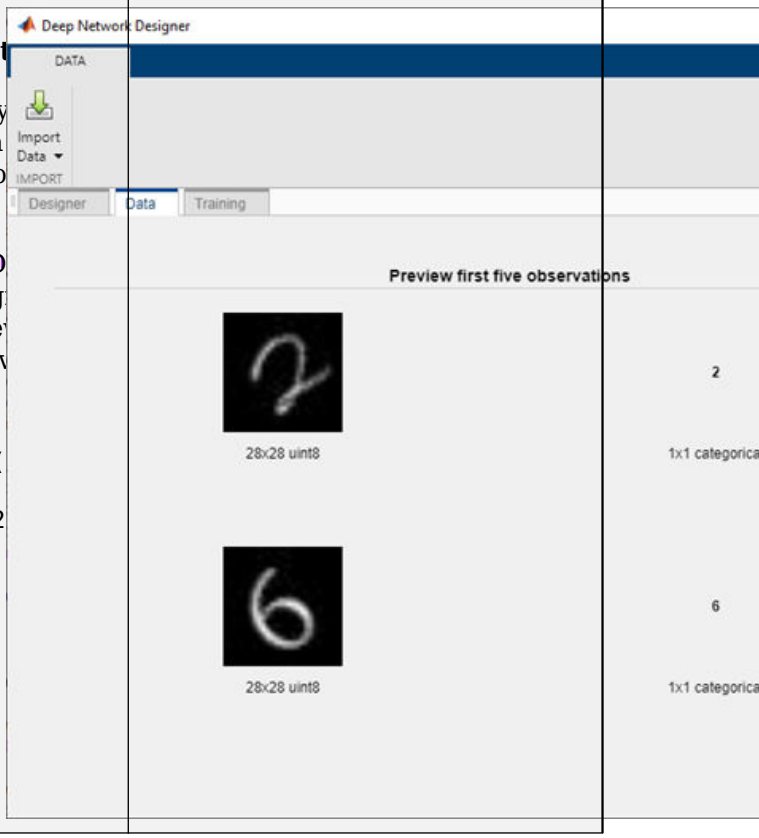
After import, Deep Network Designer provides a preview of the imported data so that you can check that the data is as expected, prior to training. For image classification data, Deep Network Designer also displays a histogram of the class labels and a random selection of images from the imported data. You can also choose to see random images belonging to a specific class.

Import Data by Task

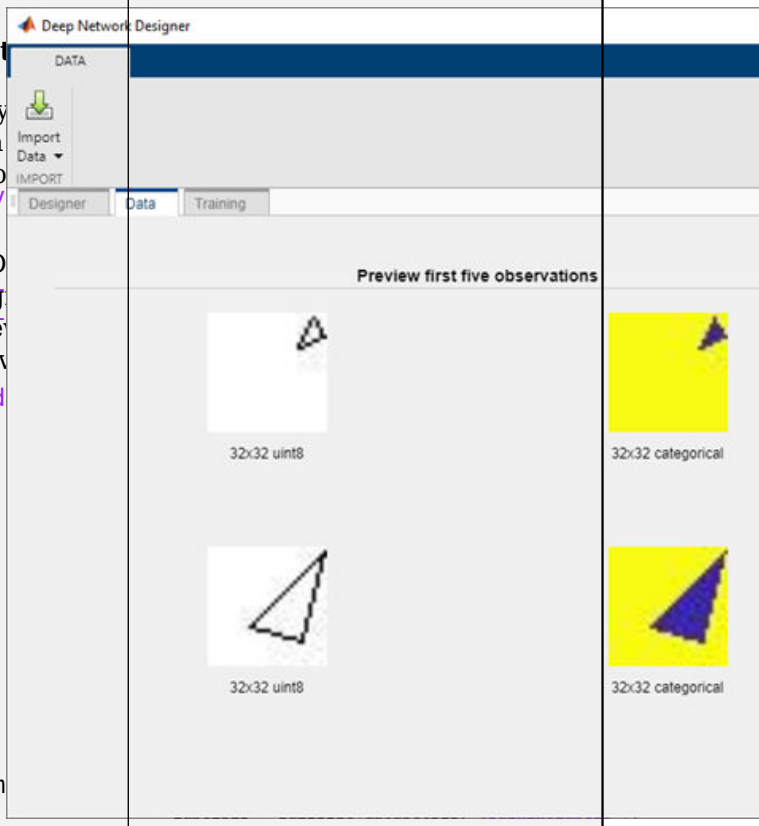
Task	Data Type	Data Import Method	Example Visualization
Image classification	Folder with subfolders containing images for each class. The class labels are sourced from the subfolder names. For an example, see "Transfer Learning with Deep Network Designer" on page 2-2.	Select **Import** > **Import Image** You can select augmentation and specify the validation data the Import Ima dialog box. After import, D Network Desig displays a histo the class labels also see rando observations fr class.	

	`ImageDatastore`
	
	For example, create an image datastore containing digits data.
	
	`dataFolder = fullfile(toolboxdir('nnet'),'nndemos', ...`
	`'nndatasets','DigitDataset');`
	
	`imds = imageDatastore(dataFolder, ...`
	`'IncludeSubfolders',true, ...`
	`'LabelSource','foldernames');`
	
	For more information, see "Create Simple Image Classification Network Using Deep Network Designer".		

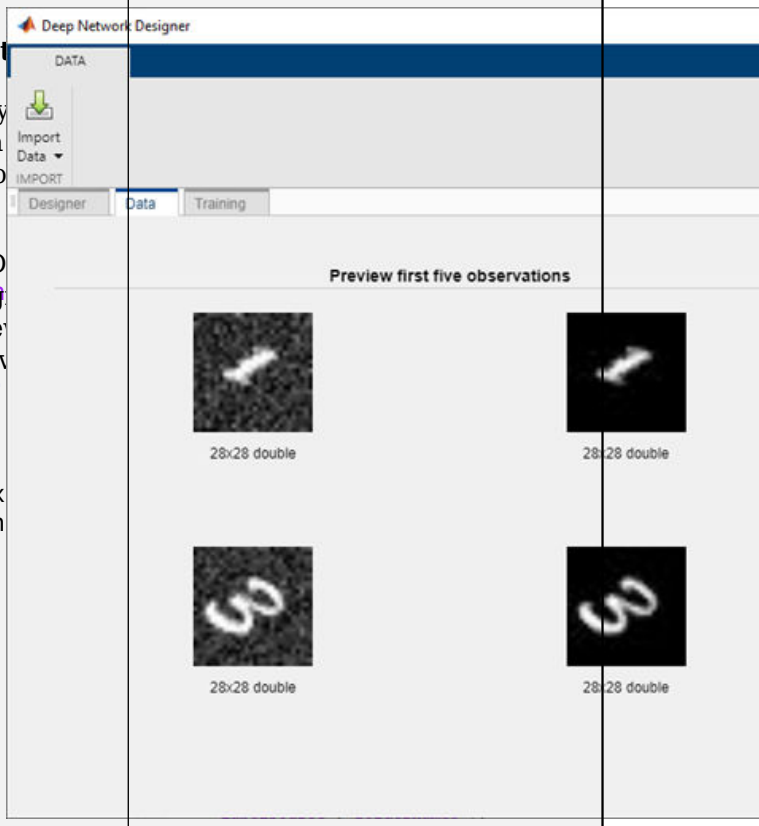
Task	Data Type	Data Import Method	Example Visualization

	`AugmentedImageDatastore`
	
	For example, create an augmented image datastore containing digits data.
	
	`dataFolder = fullfile(toolboxdir('nn','nndatasets','DigitDataset');`
	
	`imds = imageDatastore(dataFolder, ...`
	`'IncludeSubfolders',true, ...`
	`'LabelSource','foldernames');`
	
	`imageAugmenter = imageDataAugmenter(`
	`'RandRotation',[1,2]);`
	`augimds = augmentedImageDatastore([2`
	`'DataAugmentation',imageAugmenter);`
	
	`augimds = shuffle(augimds);`
	
	For more information, see “Transfer Learning Using Pretrained Network” on page 3-33.
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		
		After import, D Network Desig shows a previe first five observ the datastore.
			

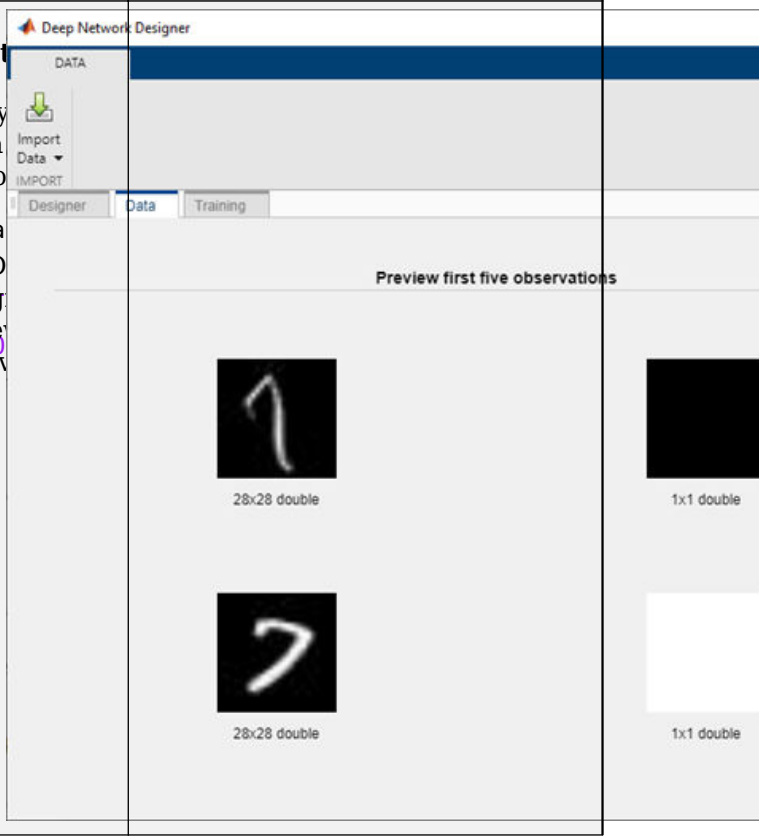
Task	Data Type	Data Import Method	Example Visualization

Semantic segmentation	`CombinedDatastore`
	
	For example, combine an `ImageDatastore` and a `PixelLabelDatastore`.
	
	`dataFolder = fullfile(toolboxdir('v`
	`'visiondata','triangleImages');`
	
	`imageDir = fullfile(dataFolder,'trai`
	`labelDir = fullfile(dataFolder,'trai`
	
	`imds = imageDatastore(imageDir);`
	`classNames = ["triangle","backgrou`
	`labelIDs = [255 0];`
	
	`pxds = pixelLabelDatastore(labelDir,`
	`cds = combine(imds,pxds);`
	
	You can also combine an `ImageDatastore` and a `PixelLabelDatastore` in a `pixelLabelImageDatastore`.
	
	`pximds = pixelLabelImageDatastore(im`
	
	For more information about creating and training a semantic segmentation network, see “Train Simple Semantic Segmentation Network in Deep Network Designer” on page 8-148.
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		
		After import, D Network Desig shows a previe first five observ the datastore.
			

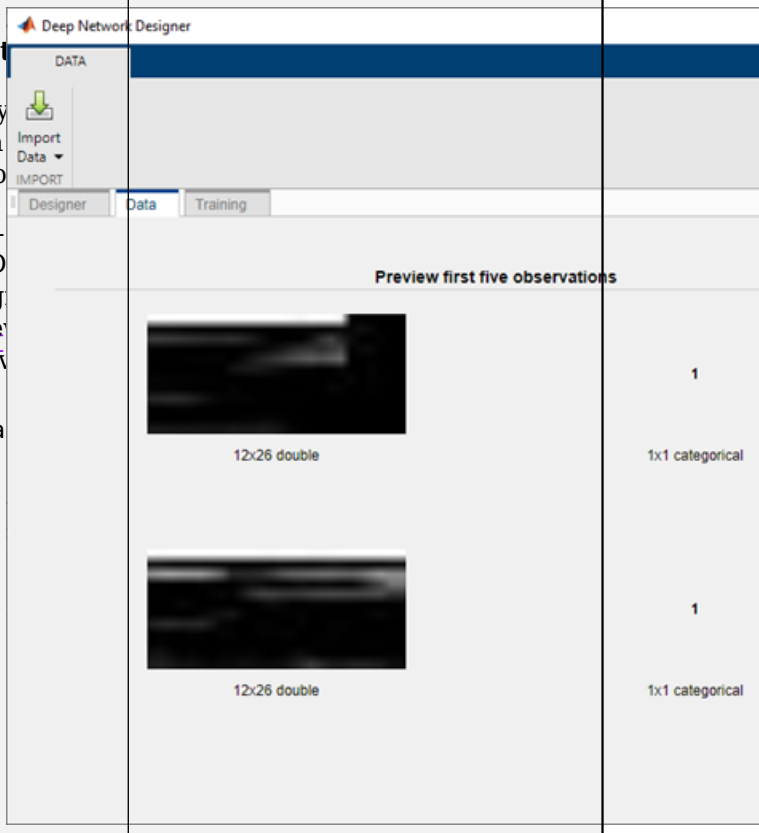
Task	Data Type	Data Import Method	Example Visualization

Image-to-image regression	`CombinedDatastore`
	
	For example, combine noisy input images and pristine output images to create data suitable for image-to-image regression.
	
	`dataFolder = fullfile(toolboxdir('nn','nndatasets','DigitDataset');`
	
	`imds = imageDatastore(dataFolder, ...`
	`    'IncludeSubfolders',true, ...`
	`    'LabelSource','foldernames');`
	
	`imds = transform(imds,@(x) rescale(x`
	`imdsNoise = transform(imds,@(x) {imn`
	
	`cds = combine(imdsNoise,imds);`
	`cds = shuffle(cds);`
	
	For more information about training an image-to-image regression network, see “Image-to-Image Regression in Deep Network Designer” on page 2-72.
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		
		After import, D Network Desig shows a previe first five observ the datastore.
			

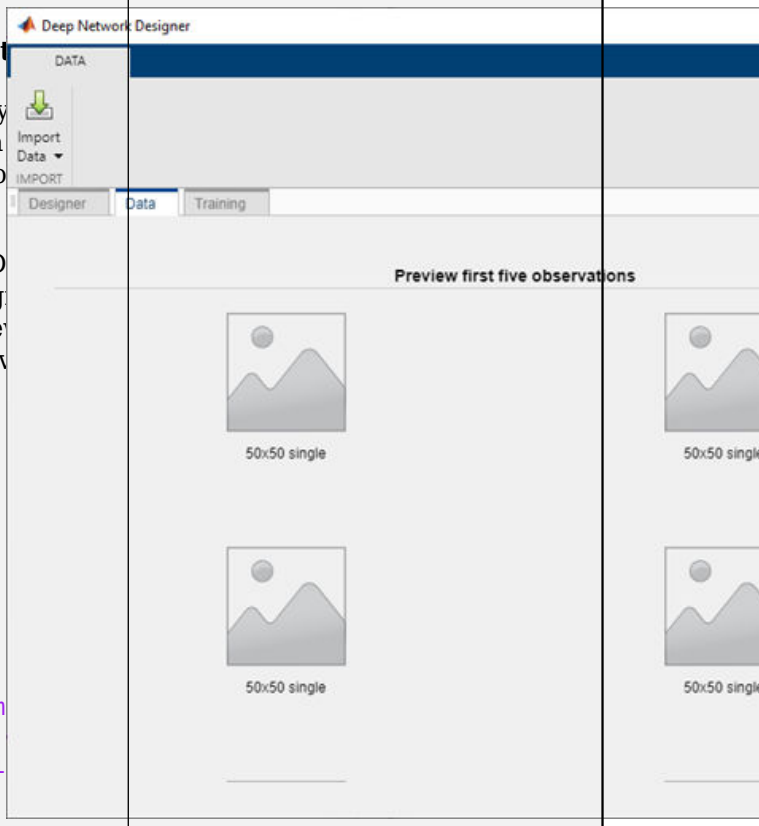
Task	Data Type	Data Import Method	Example Visualization

Regression	`CombinedDatastore`
	
	Create data suitable for training regression networks by combining array datastore objects.
	
	`[XTrain,~,YTrain] = digitTrain4DArra`
	
	`ads = arrayDatastore(XTrain,'Iterati`
	`'OutputType','cell');`
	`adsAngles = arrayDatastore(YTrain,'O`
	
	`cds = combine(ads,adsAngles);`
	
	For more information about training a regression network, see "Train Convolutional Neural Network for Regression" on page 3-53.
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		After import, D Network Desig shows a previe first five observ the datastore.
			

Task	Data Type	Data Import Method	Example Visualization

Sequences and time series	`CombinedDatastore`
	
	For in-memory data, create data suitable for training by combining array datastore objects.
	
	`[XTrain,YTrain] = japaneseVowelsTrai`
	
	`XTrain = padsequences(XTrain,2);`
	
	`adsXTrain = arrayDatastore(XTrain,'I`
	`adsYTrain = arrayDatastore(YTrain);`
	
	`cdsTrain = combine(adsXTrain,adsYTra`
	
	For information about training a network on time series data, see "Train Network for Time Series Forecasting Using Deep Network Designer" on page 2-57.
	
	For out-of-memory data, use a custom datastore object. For an example showing how to create a custom sequence datastore, see "Train Network Using Custom Mini-Batch Datastore for Sequence Data" on page 19-104.
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		
		After import, D Network Desig shows a previe first five obser the datastore.
			

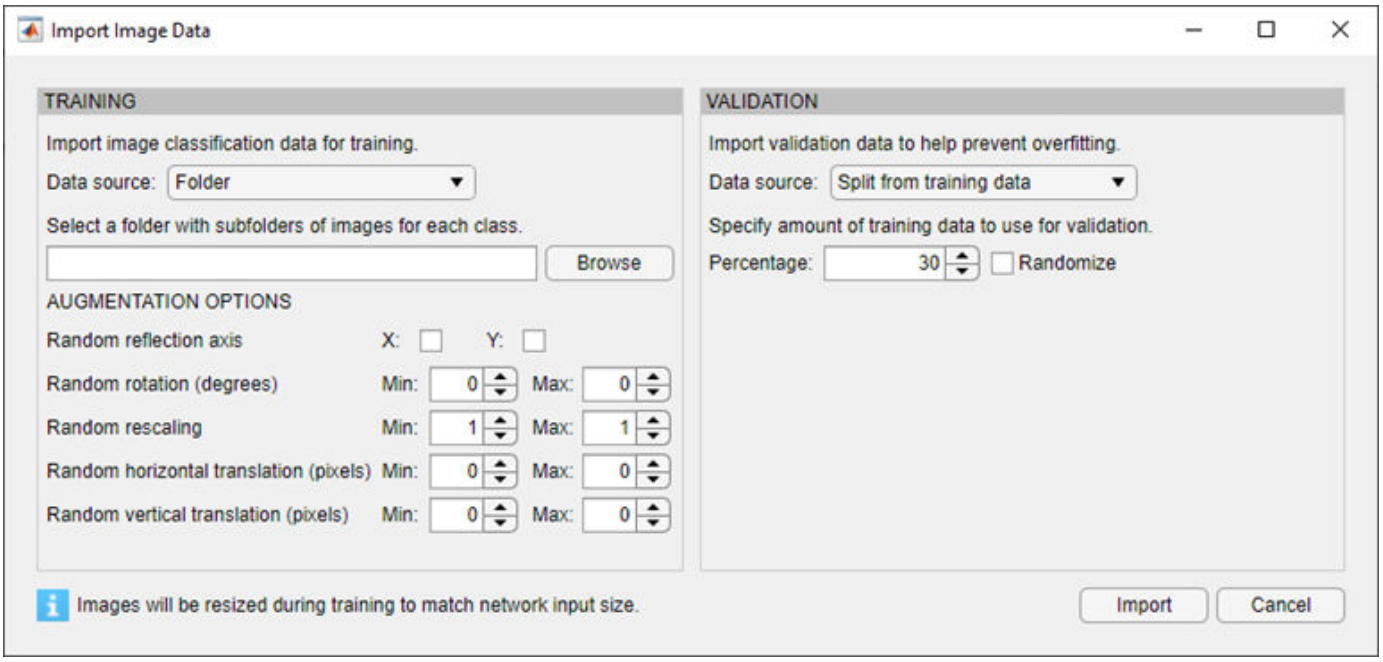
Task	Data Type	Data Import Method	Example Visualization

Other extended workflows (such as numeric feature input, out-of-memory data, image processing, and audio and speech processing)	Datastore
	
	For other extended workflows, use a suitable datastore object. For example, custom datastore, `randomPatchExtractionDatastore`, `denoisingImageDatastore`, or `audioDatastore`. For more information, see "Datastores for Deep Learning" on page 19-2.
	
	For example, create a `denoisingImageDatastore` object using Image Processing Toolbox™.
	
	`dataFolder = fullfile(toolboxdir('im`
	
	`imds = imageDatastore(dataFolder,'Fi`
	
	`dnds = denoisingImageDatastore(imds,...`
	`    'PatchesPerImage',512,...`
	`    'PatchSize',50,...`
	`    'GaussianNoiseLevel',[0.01 0.1]);`
	
	For table array data, you must convert your data into a suitable datastore to train using Deep Network Designer. For example, start by converting your table into arrays containing the predictors and responses. Then, convert the arrays into `arrayDatastore` objects. Finally, combine the predictor and response array datastores into a `CombinedDatastore` object. You can then use the combined datastore to train in Deep Network Designer. For more information on suitable
		Select **Import** > **Import Datast**
		
		You can specify validation data Import Datasto box.
		
		After import, D Network Desig shows a previe first five observ the datastore.
			

Task	Data Type	Data Import Method	Example Visualization
	datastores, see "Datastores for Deep Learning" on page 19-2.		

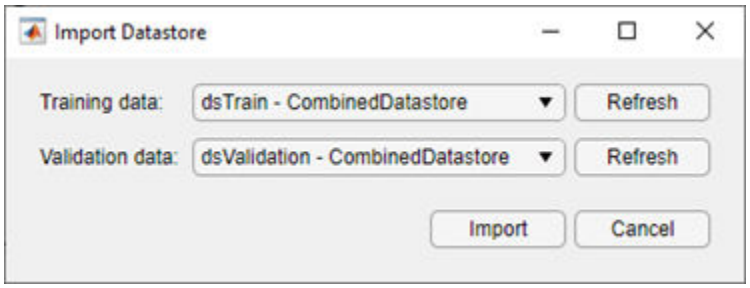
Image Augmentation

For image classification problems, Deep Network Designer provides simple augmentation options to apply to the training data. Open the Import Image Data dialog box by selecting **Import Data** > **Import Image Data**. You can select options to apply a random combination of reflection, rotation, rescaling, and translation operations to the training data.



You can effectively increase the amount of training data by applying randomized augmentation to your data. Augmentation also enables you to train networks to be invariant to distortions in image data. For example, you can add randomized rotations to input images so that a network is invariant to the presence of rotation in input images. Data augmentation can also help prevent the network from overfitting and memorizing the exact details of the training images. When you use data augmentation, one randomly augmented version of each image is used during each epoch of training, where an epoch is a full pass of the training algorithm over the entire training data set. Therefore, each epoch uses a slightly different data set, but the actual number of training images in each epoch does not change. For more information, see "Create and Explore Datastore for Image Classification" on page 19-10.

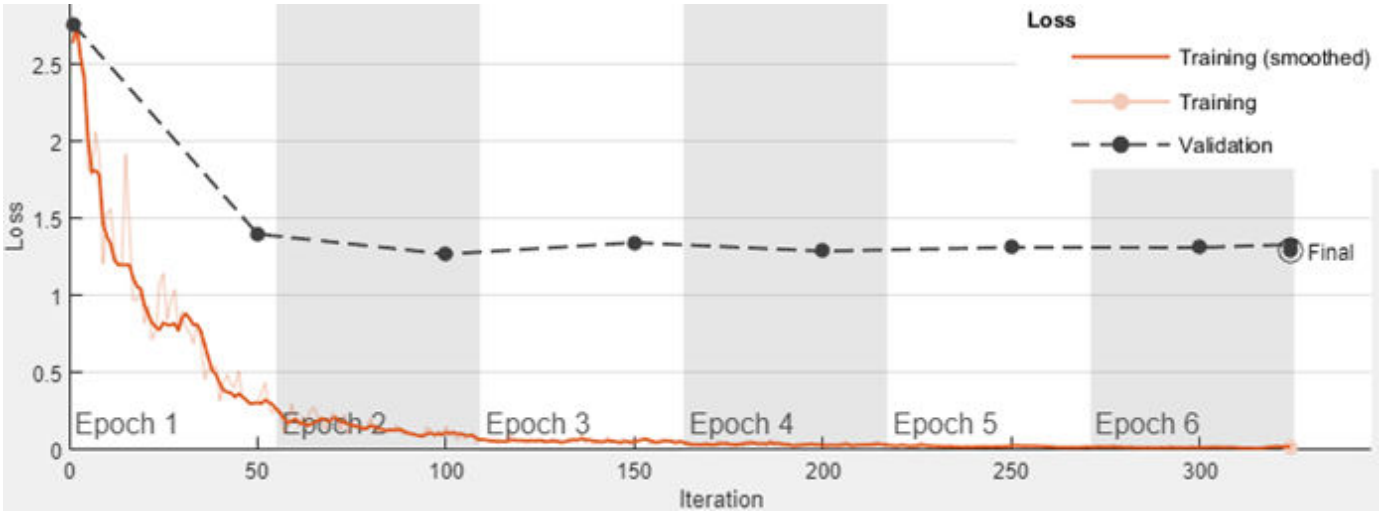
To perform more general and complex image preprocessing operations than those offered by Deep Network Designer, use `TransformedDatastore` and `CombinedDatastore` objects. To import `CombinedDatastore` and `TransformedDatastore` objects, select **Import Data** > **Import Datastore**.



For more information on image augmentation, see “Preprocess Images for Deep Learning” on page 19-16.

Validation Data

In Deep Network Designer, you can import validation data to use during training. Validation data is data that the network does not use to update the weights and biases during training. As the network does not directly use this data, it is useful for assessing the true accuracy of the network during training. You can monitor validation metrics, such as loss and accuracy, to assess if the network is overfitting or underfitting and adjust the training options as required. For example, if the validation loss is much higher than the training loss, then the network might be overfitting.



For more information on improving the accuracy of deep learning networks, see “Deep Learning Tips and Tricks” on page 1-67.

In Deep Network Designer, you can import validation data:

- From a datastore in the workspace.
- From a folder containing subfolders of images for each class (image classification data only).
- By splitting a portion of the training data to use as validation data (image classification data only).

Split Validation Data from Training Data

When splitting the validation data from the training data, Deep Network Designer splits a percentage of the training data from each class. For example, suppose you have a data set with two classes, cat

and dog, and choose to use 30% of the training data for validation. Deep Network Designer uses the last 30% of images with the label "cat" and the last 30% with the label "dog" as the validation set.

Rather than using the last 30% of the training data as validation data, you can choose to randomly allocate the observations to the training and validation sets by selecting the **Randomize** check box in the Import Image Data dialog box. Randomizing the images can improve the accuracy of networks trained on data stored in a nonrandom order. For example, the digits data set consists of 10,000 synthetic grayscale images of handwritten digits. This data set has an underlying order in which images with the same handwriting style appear next to each other within each class. An example of the display follows.

Randomizing ensures that when you split the data, the images are shuffled so that the training and validation sets contain random images from each class. Using training and validation data that consists of a similar distribution of images can help prevent overfitting. Not randomizing the data ensures that the training and validation data split is the same each time, and can help improve the reproducibility of results. For more information, see `splitEachLabel`.

See Also

Deep Network Designer | `TransformedDatastore` | `CombinedDatastore` | `imageDatastore` | `augmentedImageDatastore` | `splitEachLabel`

Related Examples

- “Data Sets for Deep Learning” on page 19-118
- “Build Networks with Deep Network Designer” on page 2-15
- “Create and Explore Datastore for Image Classification” on page 19-10
- “Image-to-Image Regression in Deep Network Designer” on page 2-72
- “Train Simple Semantic Segmentation Network in Deep Network Designer” on page 8-148
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-69

Create Simple Sequence Classification Network Using Deep Network Designer

This example shows how to create a simple long short-term memory (LSTM) classification network using Deep Network Designer.

To train a deep neural network to classify sequence data, you can use an LSTM network. An LSTM network is a type of recurrent neural network (RNN) that learns long-term dependencies between time steps of sequence data.

The example demonstrates how to:

- Load sequence data.
- Construct the network architecture.
- Specify training options.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

Load Data

Load the Japanese Vowels data set, as described in [1] on page 2-0 and [2] on page 2-0 . The predictors are cell arrays containing sequences of varying length with a feature dimension of 12. The labels are categorical vectors of labels 1,2,...,9.

```
[XTrain,YTrain] = japaneseVowelsTrainData;
[XValidation,YValidation] = japaneseVowelsTestData;
```

View the sizes of the first few training sequences. The sequences are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

```
XTrain(1:5)
```

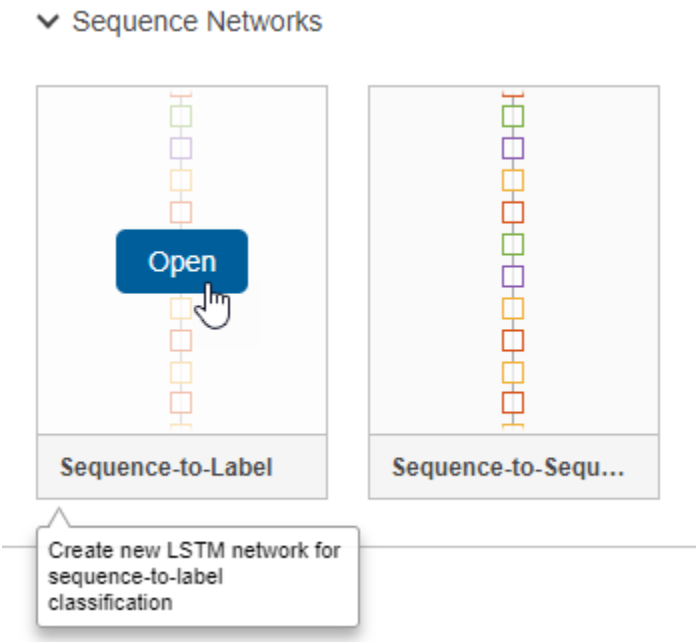
```
ans=5×1 cell array
    {12×20 double}
    {12×26 double}
    {12×22 double}
    {12×20 double}
    {12×21 double}
```

Define Network Architecture

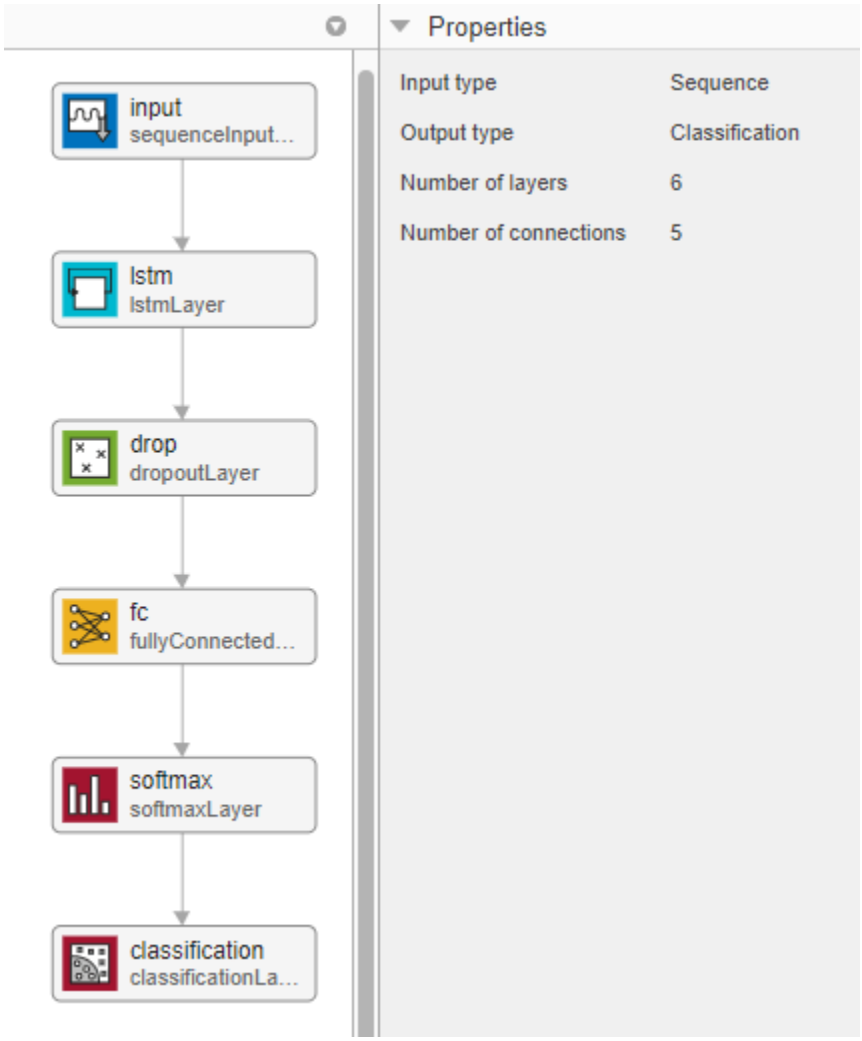
Open Deep Network Designer.

```
deepNetworkDesigner
```

Pause on **Sequence-to-Label** and click **Open**. This opens a prebuilt network suitable for sequence classification problems.

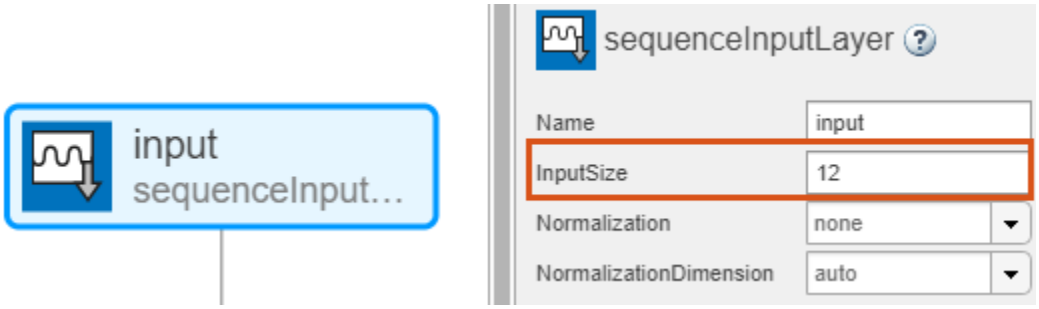


Deep Network Designer displays the prebuilt network.

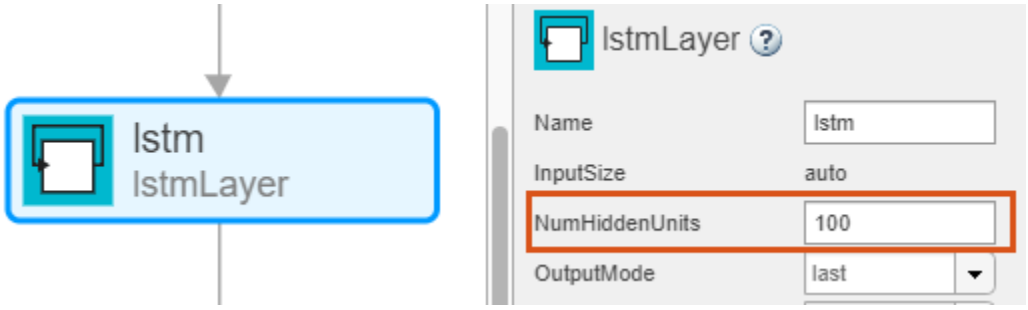


You can easily adapt this sequence network for the Japanese Vowels data set.

Select **sequenceInputLayer** and check that **InputSize** is set to 12 to match the feature dimension.



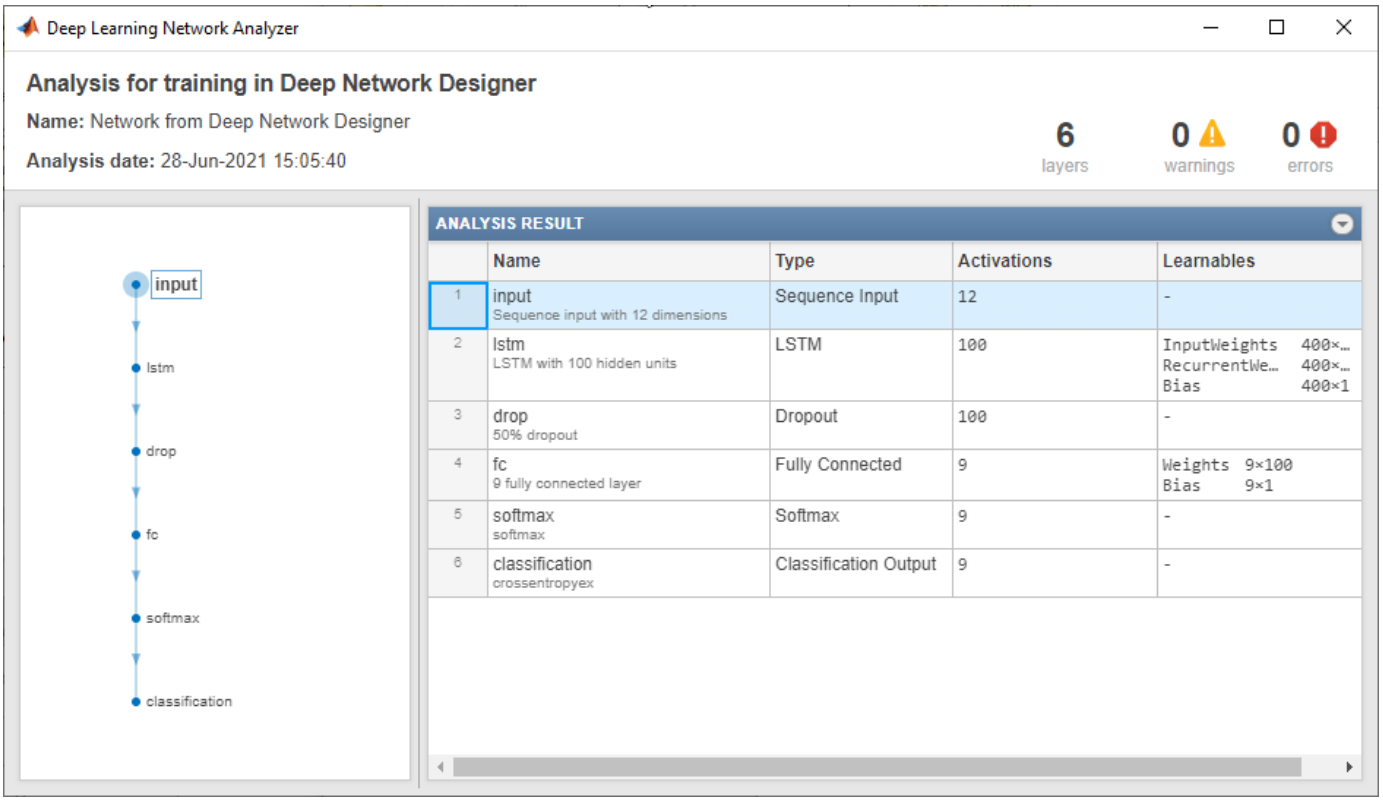
Select **lstmLayer** and set **NumHiddenUnits** to 100.



Select **fullyConnectedLayer** and check that **OutputSize** is set to 9, the number of classes.

Check Network Architecture

To check the network and examine more details of the layers, click **Analyze**.



Export Network Architecture

To export the network architecture to the workspace, on the **Designer** tab, click **Export**. Deep Network Designer saves the network as the variable `layers_1`.

You can also generate code to construct the network architecture by selecting **Export** > **Generate Code**.

Train Network

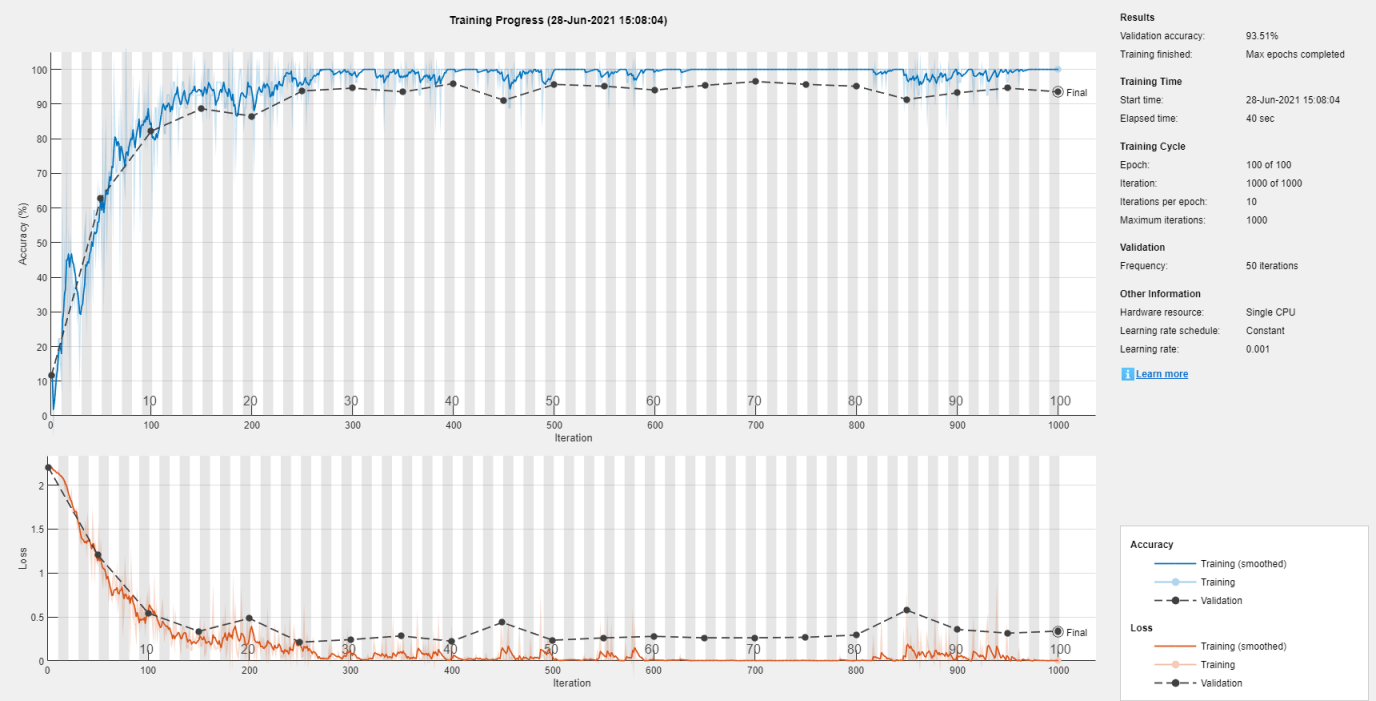
Specify the training options and train the network.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set `'ExecutionEnvironment'` to `'cpu'`. To train on a GPU, if available, set `'ExecutionEnvironment'` to `'auto'` (the default value).

```
miniBatchSize = 27;
options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',100, ...
    'MiniBatchSize',miniBatchSize, ...
    'ValidationData',{XValidation,YValidation}, ...
    'GradientThreshold',2, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers_1,options);
```



You can also train this network using Deep Network Designer and datastore objects. For an example showing how to train a sequence-to-sequence regression network in Deep Network Designer, see “Train Network for Time Series Forecasting Using Deep Network Designer” on page 2-57.

Test Network

Classify the test data and calculate the classification accuracy. Specify the same mini-batch size as for training.

```
YPred = classify(net,XValidation,'MiniBatchSize',miniBatchSize);
acc = mean(YPred == YValidation)
```

```
acc = 0.9405
```

For next steps, you can try improving the accuracy by using bidirectional LSTM (BiLSTM) layers or by creating a deeper network. For more information, see “Long Short-Term Memory Networks” on page 1-75.

For an example showing how to use convolutional networks to classify sequence data, see “Speech Command Recognition Using Deep Learning” on page 4-23.

References

[1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. “Multidimensional Curve Classification Using Passing-through Regions.” Pattern Recognition Letters 20, no. 11–13 (November 1999): 1103–11. https://doi.org/10.1016/S0167-8655(99)00077-X.

[2] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. Japanese Vowels Data Set. Distributed by UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

Deep Network Designer

Related Examples

- “List of Deep Learning Layers” on page 1-21
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-69
- “Deep Learning Tips and Tricks” on page 1-67

Train Network for Time Series Forecasting Using Deep Network Designer

This example shows how to forecast time series data by training a long short-term memory (LSTM) network in **Deep Network Designer**.

Deep Network Designer allows you to interactively create and train deep neural networks for sequence classification and regression tasks.

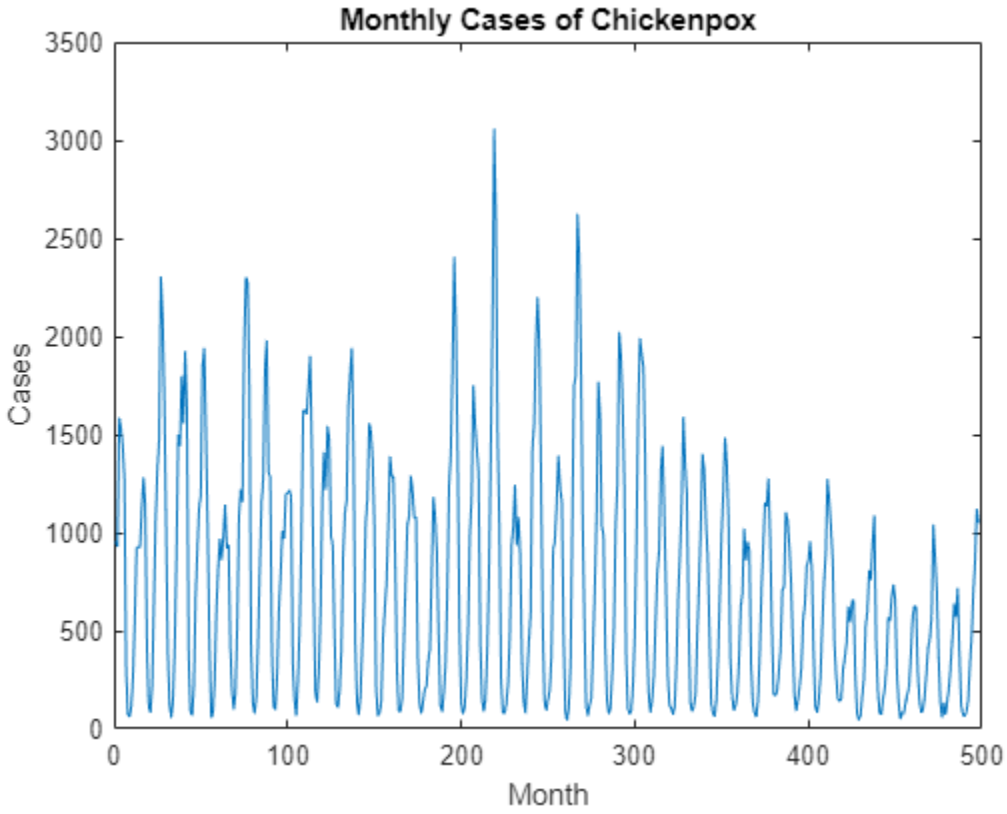
To forecast the values of future time steps of a sequence, you can train a sequence-to-sequence regression LSTM network, where the responses are the training sequences with values shifted by one time step. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step.

This example uses the data set `chickenpox_dataset`. The example creates and trains an LSTM network to forecast the number of chickenpox cases given the number of cases in previous months.

Load Sequence Data

Load the example data. `chickenpox_dataset` contains a single time series, with time steps corresponding to months and values corresponding to the number of cases. The output is a cell array, where each element is a single time step. Reshape the data to be a row vector.

```
data = chickenpox_dataset;
data = [data{:}];

figure
plot(data)
xlabel("Month")
ylabel("Cases")
title("Monthly Cases of Chickenpox")
```



Partition the training and test data. Train on the first 90% of the sequence and test on the last 10%.

```
numTimeStepsTrain = floor(0.9*numel(data))
```

```
numTimeStepsTrain = 448
```

```
dataTrain = data(1:numTimeStepsTrain+1);
dataTest = data(numTimeStepsTrain+1:end);
```

Standardize Data

For a better fit and to prevent the training from diverging, standardize the training data to have zero mean and unit variance. For prediction, you must standardize the test data using the same parameters as the training data.

```
mu = mean(dataTrain);
sig = std(dataTrain);

dataTrainStandardized = (dataTrain - mu) / sig;
```

Prepare Predictors and Responses

To forecast the values of future time steps of a sequence, specify the responses as the training sequences with values shifted by one time step. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step. The predictors are the training sequences without the final time step.

```
XTrain = dataTrainStandardized(1:end-1);
YTrain = dataTrainStandardized(2:end);
```

To train the network using Deep Network Designer, convert the training data to a datastore object. Use `arrayDatastore` to convert the training data predictors and responses into `ArrayDatastore` objects. Use `combine` to combine the two datastores.

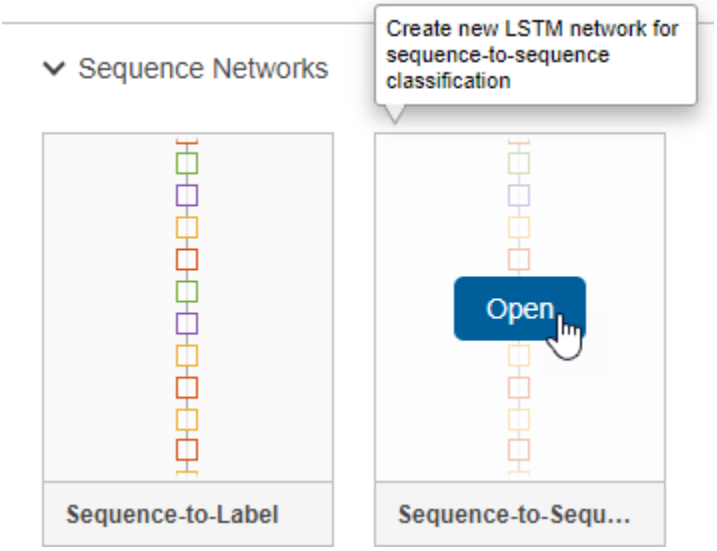
```
adsXTrain = arrayDatastore(XTrain);
adsYTrain = arrayDatastore(YTrain);

cdsTrain = combine(adsXTrain,adsYTrain);
```

Define LSTM Network Architecture

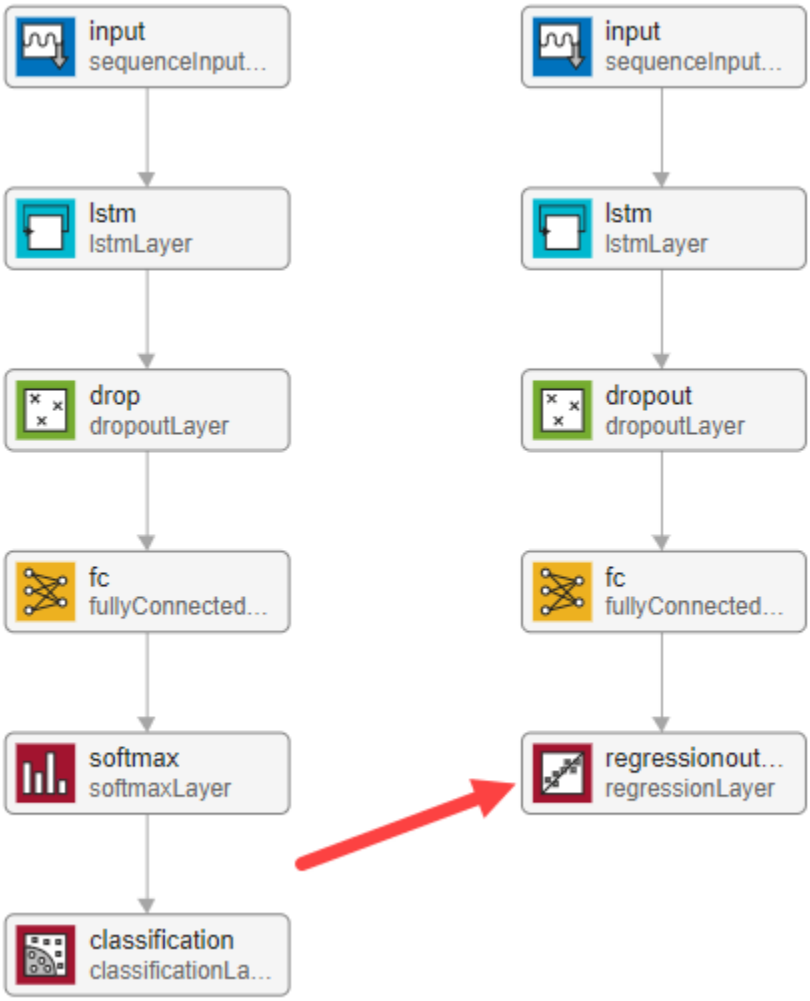
To create the LSTM network architecture, use **Deep Network Designer**. The **Deep Network Designer** app lets you build, visualize, edit, and train deep learning networks.

```
deepNetworkDesigner
```

On the **Deep Network Designer** start page, pause on **Sequence-to-Sequence** and click **Open**. Doing so opens a prebuilt network suitable for sequence-to-sequence classification tasks. You can convert the classification network into a regression network by replacing the final layers.

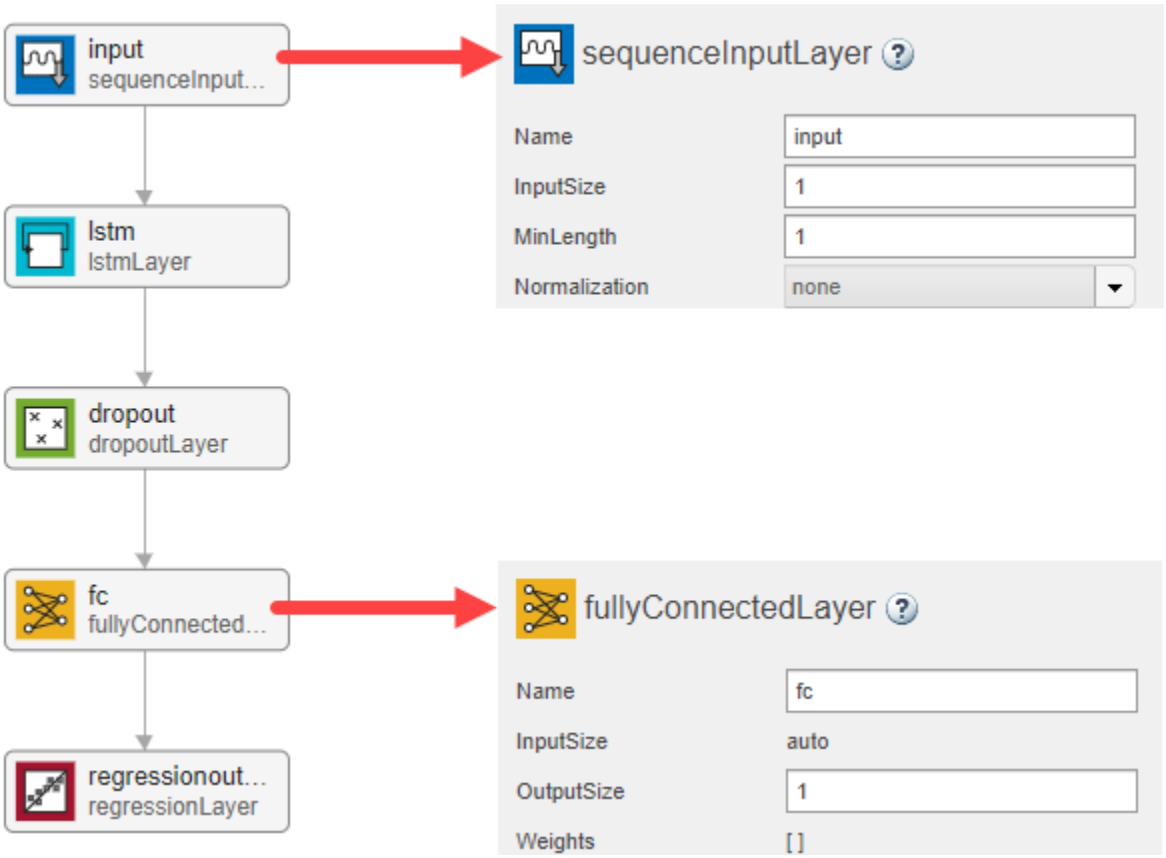


Delete the softmax layer and the classification layer and replace them with a regression layer.

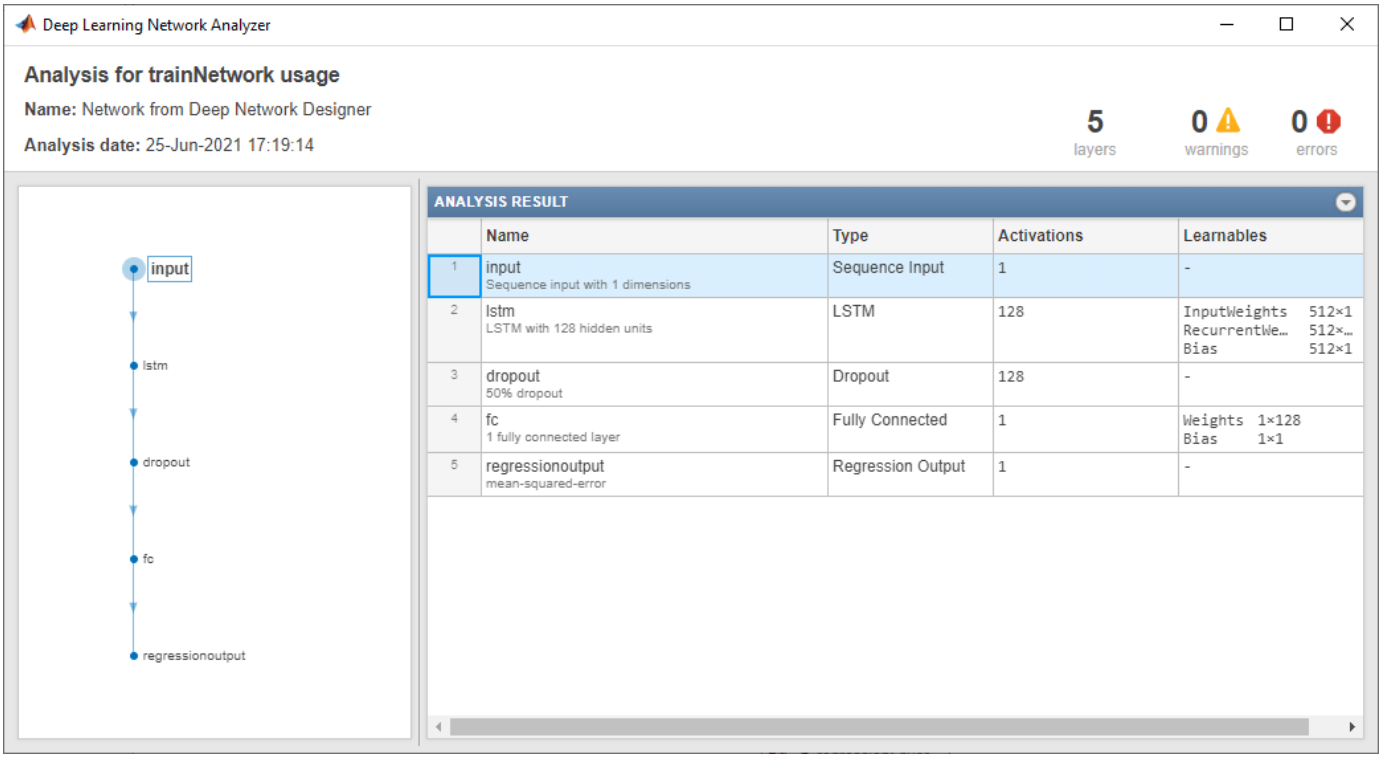


Adjust the properties of the layers so that they are suitable for the chickenpox data set. This data has a single input feature and a single output feature. Select **sequenceInputLayer** and set the

InputSize to 1. Select **fullyConnectedLayer** and set the **OutputSize** to 1.

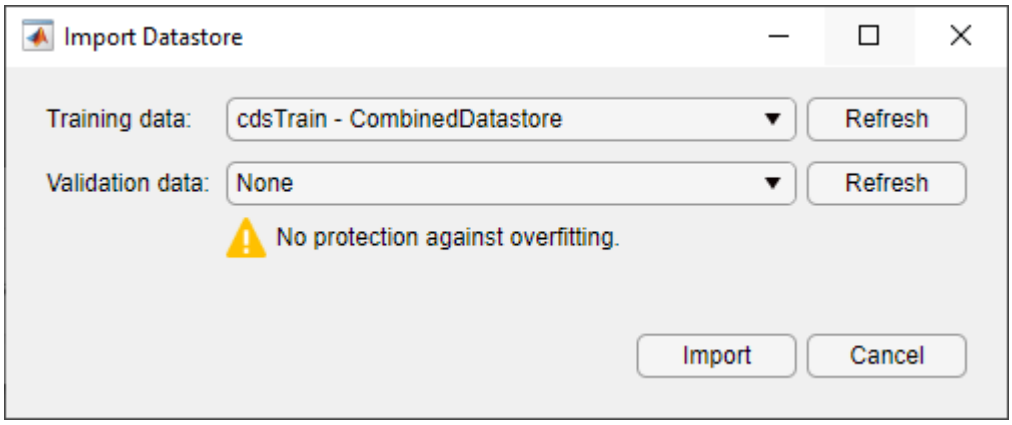


Check your network by clicking **Analyze**. The network is ready for training if **Deep Learning Network Analyzer** reports zero errors.

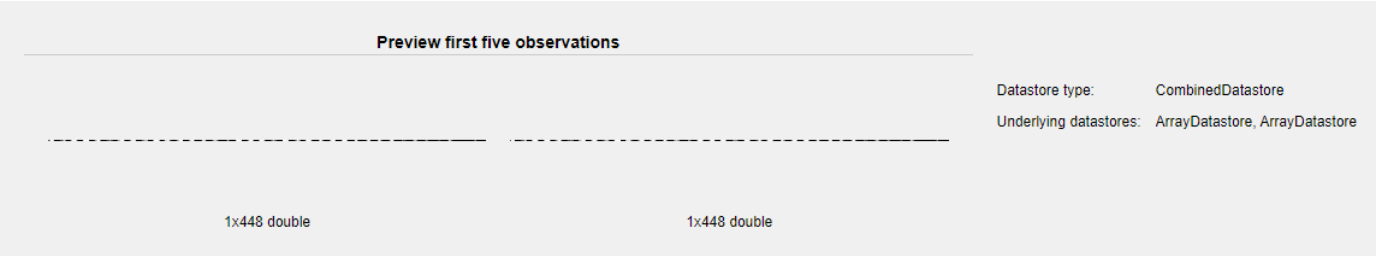


Import Data

To import the training datastore, select the **Data** tab and click **Import Data** > **Import Datastore**. Select `cdsTrain` as the training data and `None` as the validation data. Click **Import**.

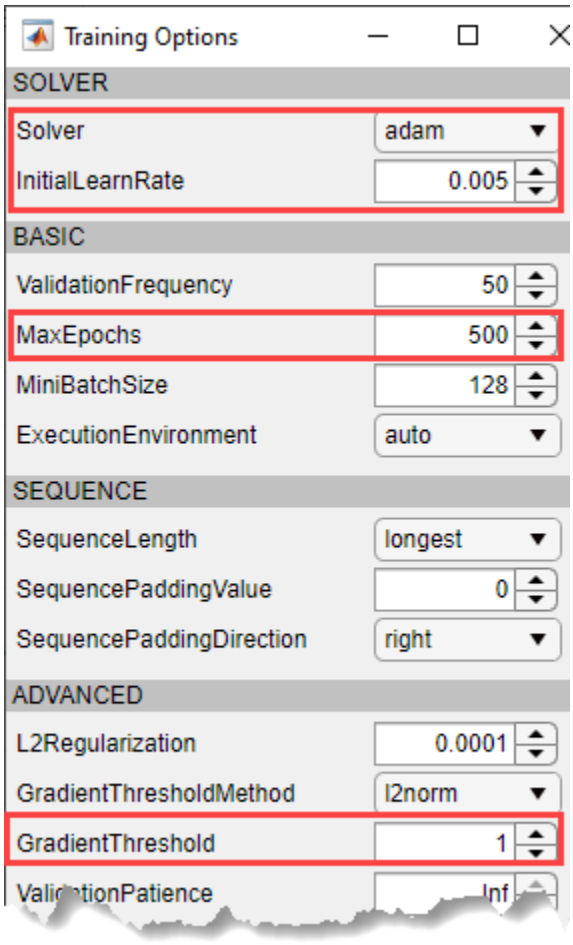


The data preview shows a single input time series and a single response time series, each with 448 time steps.



Specify Training Options

On the **Training** tab, click **Training Options**. Set **Solver** to `adam`, **InitialLearnRate** to `0.005`, and **MaxEpochs** to `500`. To prevent the gradients from exploding, set the **GradientThreshold** to `1`.

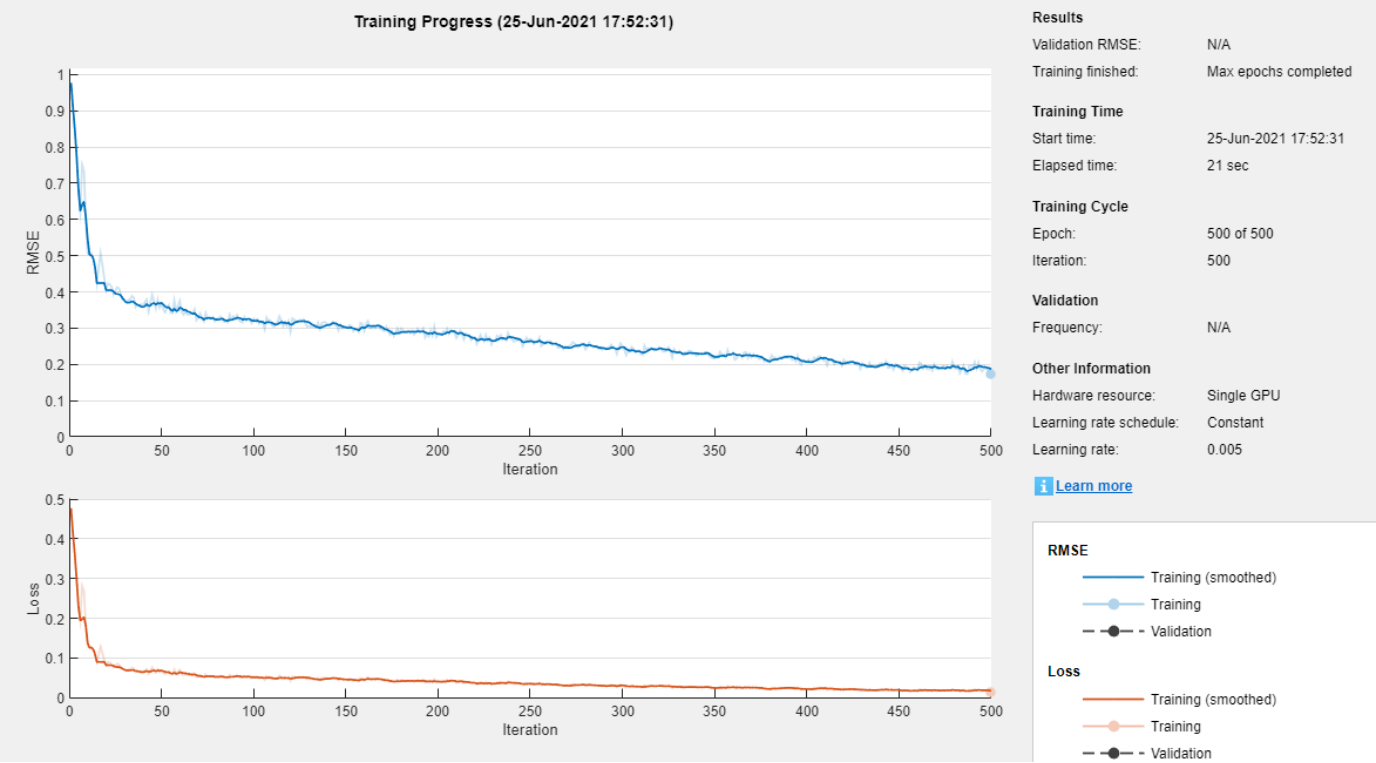


For more information about setting the training options, see `trainingOptions`.

Train Network

Click **Train**.

Deep Network Designer displays an animated plot showing the training progress. The plot shows mini-batch loss and accuracy, validation loss and accuracy, and additional information on the training progress.



Once training is complete, export the trained network by clicking **Export** in the **Training** tab. The trained network is saved as the `trainedNetwork_1` variable.

Forecast Future Time Steps

Test the trained network by forecasting multiple time steps in the future. Use the `predictAndUpdateState` function to predict time steps one at a time and update the network state at each prediction. For each prediction, use the previous prediction as input to the function.

Standardize the test data using the same parameters as the training data.

```
dataTestStandardized = (dataTest - mu) / sig;

XTest = dataTestStandardized(1:end-1);
YTest = dataTest(2:end);
```

To initialize the network state, first predict on the training data `XTrain`. Next, make the first prediction using the last time step of the training response `YTrain(end)`. Loop over the remaining predictions and input the previous prediction to `predictAndUpdateState`.

For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
net = predictAndUpdateState(trainedNetwork_1,XTrain);

[net,YPred] = predictAndUpdateState(net,YTrain(end));
```

```
numTimeStepsTest = numel(XTest);
for i = 2:numTimeStepsTest
    [net,YPred(:,i)] = predictAndUpdateState(net,YPred(:,i-1),'ExecutionEnvironment','cpu');
end
```

Unstandardize the predictions using the parameters calculated earlier.

```
YPred = sig*YPred + mu;
```

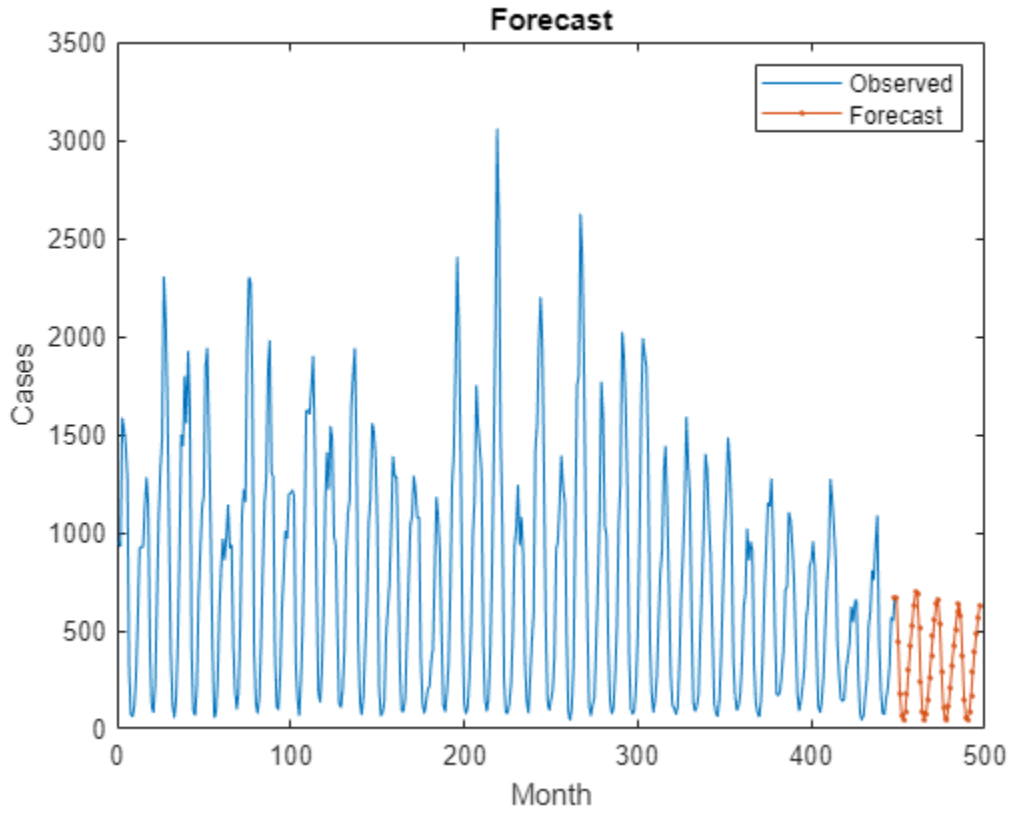
The training progress plot reports the root-mean-square error (RMSE) calculated from the standardized data. Calculate the RMSE from the unstandardized predictions.

```
rmse = sqrt(mean((YPred-YTest).^2))
```

```
rmse = single
    175.9693
```

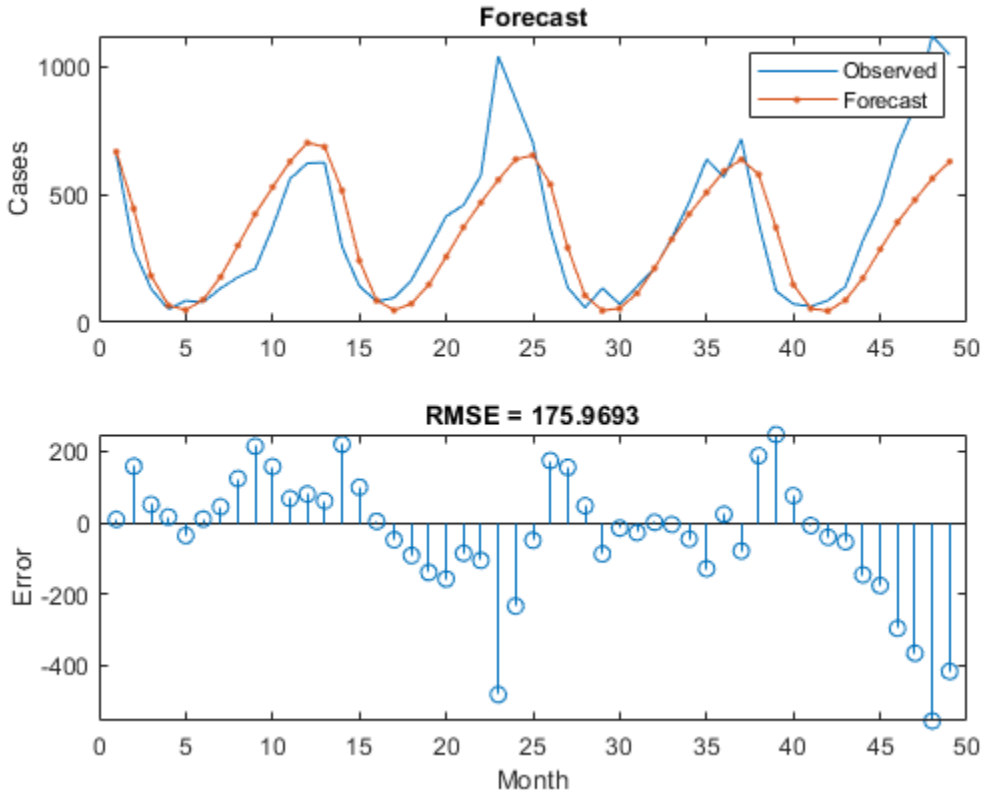
Plot the training time series with the forecasted values.

```
figure
plot(dataTrain(1:end-1))
hold on
idx = numTimeStepsTrain:(numTimeStepsTrain+numTimeStepsTest);
plot(idx,[data(numTimeStepsTrain) YPred],'.-')
hold off
xlabel("Month")
ylabel("Cases")
title("Forecast")
legend(["Observed" "Forecast"])
```



Compare the forecasted values with the test data.

```
figure
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
legend(["Observed" "Forecast"])
ylabel("Cases")
title("Forecast")

subplot(2,1,2)
stem(YPred - YTest)
xlabel("Month")
ylabel("Error")
title("RMSE = " + rmse)
```



Update Network State with Observed Values

If you have access to the actual values of time steps between predictions, then you can update the network state with the observed values instead of the predicted values.

First, initialize the network state. To make predictions on a new sequence, reset the network state using `resetState`. Resetting the network state prevents previous predictions from affecting the predictions on the new data. Reset the network state, and then initialize the network state by predicting on the training data.

```
net = resetState(net);
net = predictAndUpdateState(net,XTrain);
```

Predict on each time step. For each prediction, predict the next time step using the observed value of the previous time step. Set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
YPred = [];
numTimeStepsTest = numel(XTest);
for i = 1:numTimeStepsTest
    [net,YPred(:,i)] = predictAndUpdateState(net,XTest(:,i),'ExecutionEnvironment','cpu');
end
```

Unstandardize the predictions using the parameters calculated earlier.

```
YPred = sig*YPred + mu;
```

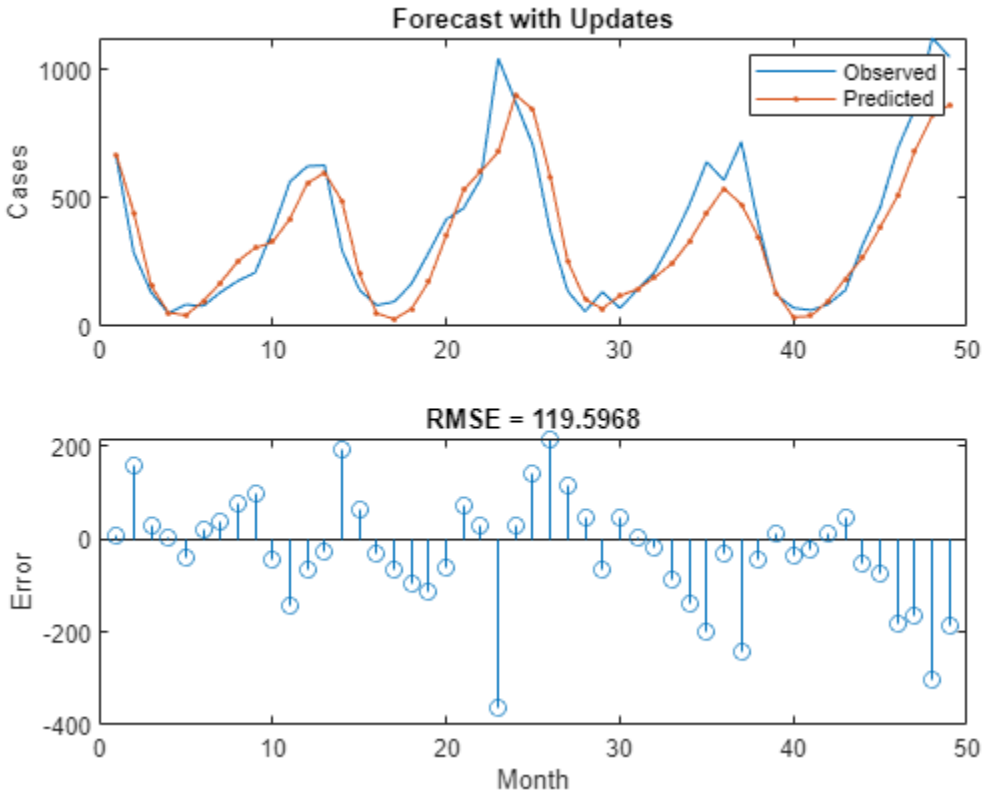
Calculate the root-mean-square error (RMSE).

```
rmse = sqrt(mean((YPred-YTest).^2))
```

```
rmse = 119.5968
```

Compare the forecasted values with the test data.

```
figure
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
legend(["Observed" "Predicted"])
ylabel("Cases")
title("Forecast with Updates")

subplot(2,1,2)
stem(YPred - YTest)
xlabel("Month")
ylabel("Error")
title("RMSE = " + rmse)
```



Here, the predictions are more accurate when updating the network state with the observed values instead of the predicted values.

See Also

Deep Network Designer

Related Examples

- “Transfer Learning with Deep Network Designer” on page 2-2
- “Build Networks with Deep Network Designer” on page 2-15
- “List of Deep Learning Layers” on page 1-21
- “Generate MATLAB Code from Deep Network Designer” on page 2-69
- “Deep Learning Tips and Tricks” on page 1-67

Generate MATLAB Code from Deep Network Designer

The Deep Network Designer app enables you to generate MATLAB code that recreates the building, editing, and training of a network in the app.

In the **Designer** tab, you can generate a live script to:

- Recreate the layers in your network. Select **Export > Generate Code**.
- Recreate the layers in your network, including any initial parameters. Select **Export > Generate Code with Initial Parameters**.

In the **Training** tab, you can generate a live script to:

- Recreate the building and training of a network you construct in Deep Network Designer. Select **Export > Generate Code for Training**.

Generate MATLAB Code to Recreate Network Layers

Generate MATLAB code for recreating the network constructed in Deep Network Designer. In the **Designer** tab, choose one of these options:

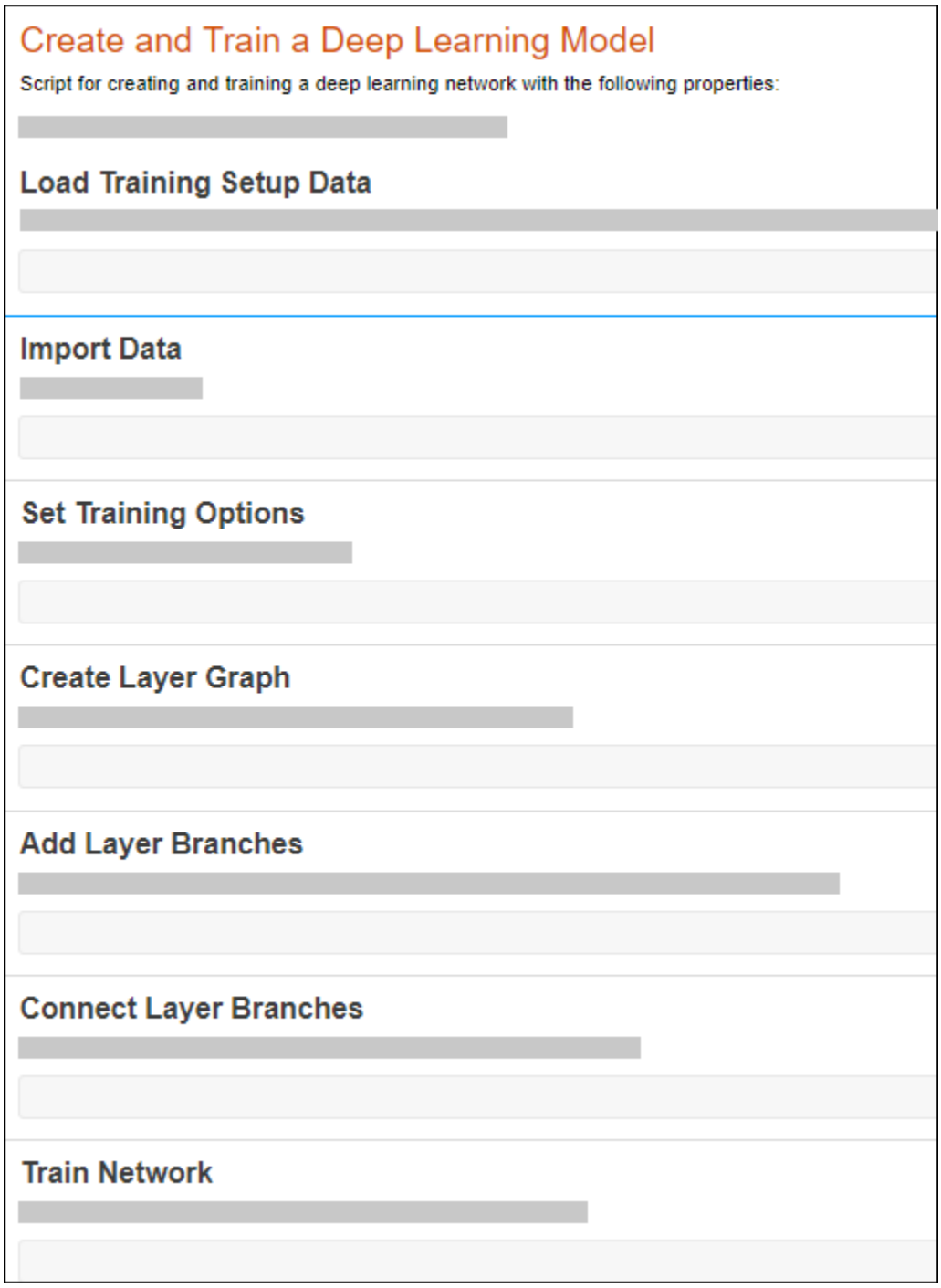
- To recreate the layers in your network, select **Export > Generate Code**. This network does not contain initial parameters, such as pretrained weights.
- To recreate the layers in your network, including any initial parameters, select **Export > Generate Code with Initial Parameters**. The app creates a live script and a MAT-file containing the initial parameters (weights and biases) from your network. Run the script to recreate the network layers, including the learnable parameters from the MAT-file. Use this option to preserve the weights if you want to perform transfer learning.

Running the generated script returns the network architecture as a variable in the workspace. Depending on the network architecture, the variable is a layer graph named `lgraph` or a layer array named `layers`. For an example of training a network exported from Deep Network Designer, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-51.

Generate MATLAB Code to Train Network

To recreate the construction and training of a network in Deep Network Designer, generate MATLAB code after training. For an example of using Deep Network Designer to train an image classification network, see “Transfer Learning with Deep Network Designer” on page 2-2.

Once training is complete, on the **Training** tab, select **Export > Generate Code for Training**. The app creates a live script and a MAT-file containing the initial parameters (weights and biases) from your network. If you import data from the workspace into Deep Network Designer then this is also contained in the generated MAT-file.



Running the generated script builds the network (including the learnable parameters from the MAT-file), imports the data, sets the training options, and trains the network. Examine the generated script to learn how to construct and train a network at the command line.

Note If you change the network, training and validation data, or training options, click **Train** before generating the live script.

You can also use the generated script as a starting point to create deep learning experiments which sweep through a range of hyperparameter values or use Bayesian optimization to find optimal training options. For an example showing how to use **Experiment Manager** to tune the hyperparameters of a network trained in Deep Network Designer, see “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager” on page 2-79.

Use Network for Prediction

Suppose that the trained network is contained in the variable `net`. To use the trained network for prediction, use the `predict` function. For example, suppose you have a trained image classification network. Use the exported network to predict the class of `peppers.png`.

```
img = imread("peppers.png");
img = imresize(img, net.Layers(1).InputSize(1:2));
label = predict(net, img);
imshow(img);
title(label);
```

References

[1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. “Multidimensional Curve Classification Using Passing-through Regions.” *Pattern Recognition Letters* 20, no. 11–13 (November 1999): 1103–11. https://doi.org/10.1016/S0167-8655(99)00077-X.

[2] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. Japanese Vowels Data Set. Distributed by UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels.

See Also

Deep Network Designer | `trainingOptions` | `trainNetwork`

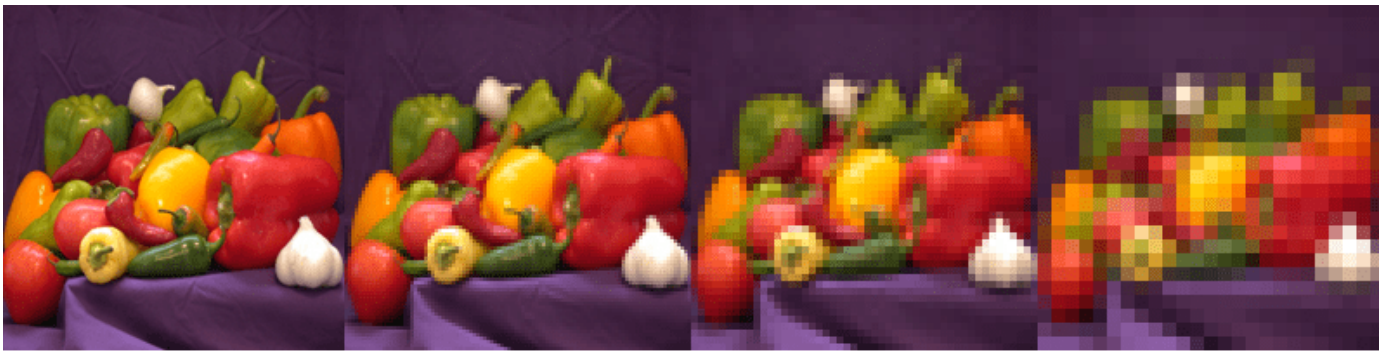
Related Examples

- “Adapt Code Generated in Deep Network Designer for Use in Experiment Manager” on page 2-79
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Build Networks with Deep Network Designer” on page 2-15
- “Import Data into Deep Network Designer” on page 2-39
- “Train Networks Using Deep Network Designer” on page 2-31

Image-to-Image Regression in Deep Network Designer

This example shows how to use Deep Network Designer to construct and train an image-to-image regression network for super resolution.

Spatial resolution is the number of pixels used to construct a digital image. An image with a high spatial resolution is composed of a greater number of pixels and as a result the image contains greater detail. Super resolution is the process of taking as input a low resolution image and upscaling it into a higher resolution image. When you work with image data, you might reduce the spatial resolution to decrease the size of the data, at the cost of losing information. To recover this lost information, you can train a deep learning network to predict the missing details of an image. In this example, you recover 28-by-28 pixel images from images that were compressed to 7-by-7 pixels.



Load Data

This example uses the digits data set, which consists of 10,000 synthetic grayscale images of handwritten digits. Each image is 28-by-28-by-1 pixels.

Load the data and create an image datastore.

```
dataFolder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');

imds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ....
    'LabelSource','foldernames');
```

Use the `shuffle` function to shuffle the data prior to training.

```
imds = shuffle(imds);
```

Use the `splitEachLabel` function to divide the image datastore into three image datastores containing images for training, validation, and testing.

```
[imdsTrain,imdsVal,imdsTest] = splitEachLabel(imds,0.7,0.15,0.15,'randomized');
```

Normalize the data in each image to the range [0,1]. Normalization helps stabilize and speed up network training using gradient descent. If your data is poorly scaled, then the loss can become NaN and the network parameters can diverge during training.

```
imdsTrain = transform(imdsTrain,@(x) rescale(x));
imdsVal = transform(imdsVal,@(x) rescale(x));
imdsTest = transform(imdsTest,@(x) rescale(x));
```

Generate Training Data

Create a training data set by generating pairs of images consisting of upsampled low resolution images and the corresponding high resolution images.

To train a network to perform image-to-image regression, the images need to be pairs consisting of an input and a response where both images are the same size. Generate the training data by downsampling each image to 7-by-7 pixels and then upsampling to 28-by-28 pixels. Using the pairs of transformed and original images, the network can learn how to map between the two different resolutions.

Generate the input data using the helper function `upsampLowRes`, which uses `imresize` to produce lower resolution images.

```
imdsInputTrain = transform(imdsTrain,@upsampLowRes);
imdsInputVal= transform(imdsVal,@upsampLowRes);
imdsInputTest = transform(imdsTest,@upsampLowRes);
```

Use the `combine` function to combine the low and high resolution images into a single datastore. The output of the `combine` function is a `CombinedDatastore` object.

```
dsTrain = combine(imdsInputTrain,imdsTrain);
dsVal = combine(imdsInputVal,imdsVal);
dsTest = combine(imdsInputTest,imdsTest);
```

Create Network Architecture

Create the network architecture using the `unetLayers` function from Computer Vision Toolbox™. This function provides a network suitable for semantic segmentation that can be easily adapted for image-to-image regression.

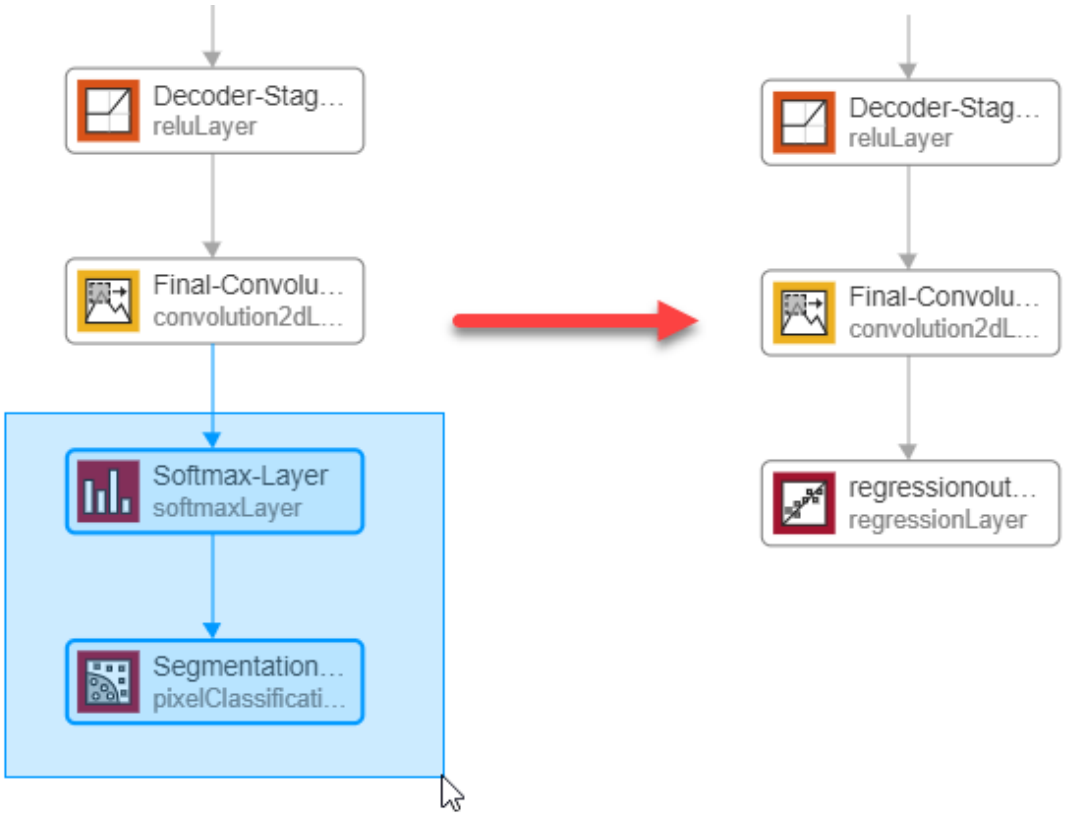
Create a network with input size 28-by-28-by-1 pixels.

```
layers = unetLayers([28,28,1],2,'encoderDepth',2);
```

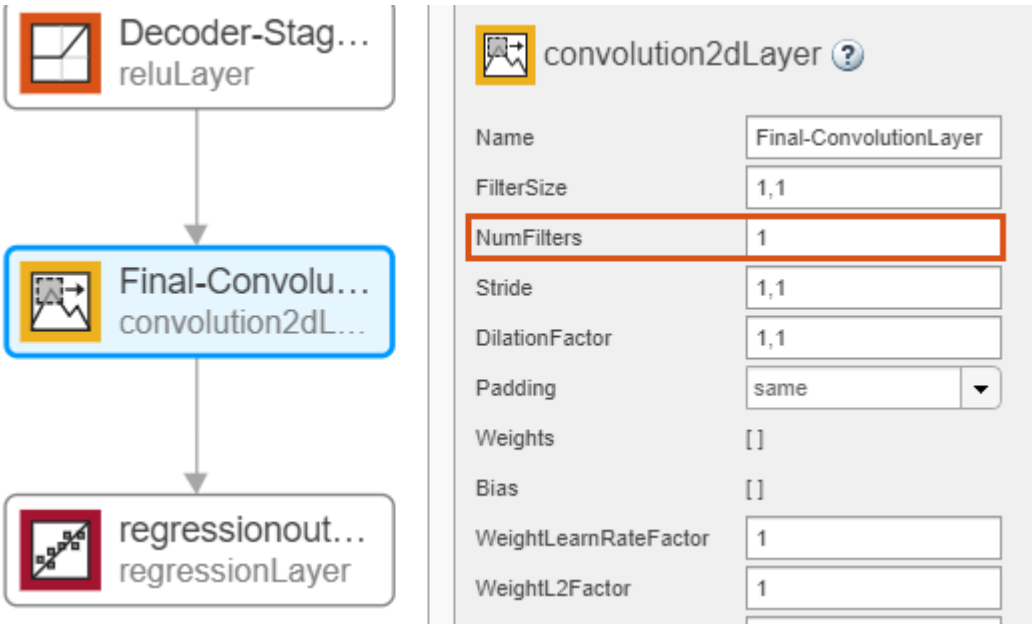
Edit the network for image-to-image regression using Deep Network Designer.

```
deepNetworkDesigner(layers);
```

In the **Designer** pane, replace the softmax and pixel classification layers with a regression layer from the **Layer Library**.



Select the final convolutional layer and set the `NumFilters` property to `1`.

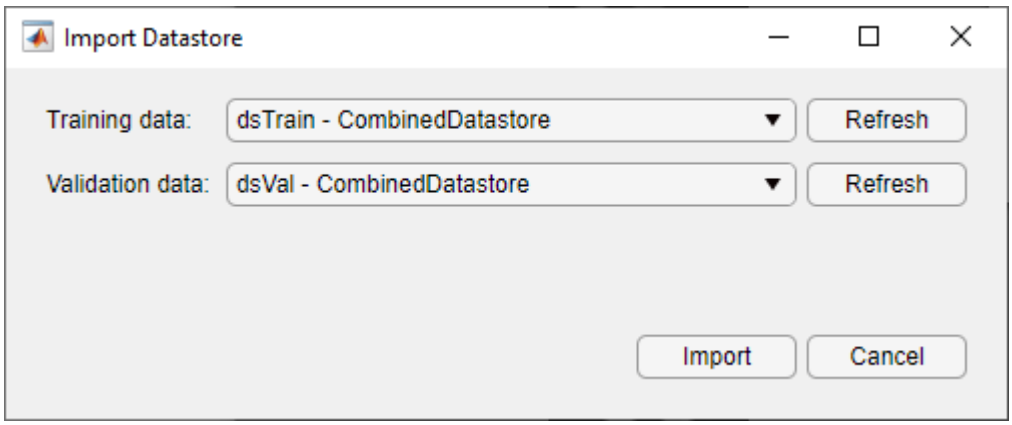


The network is now ready for training.

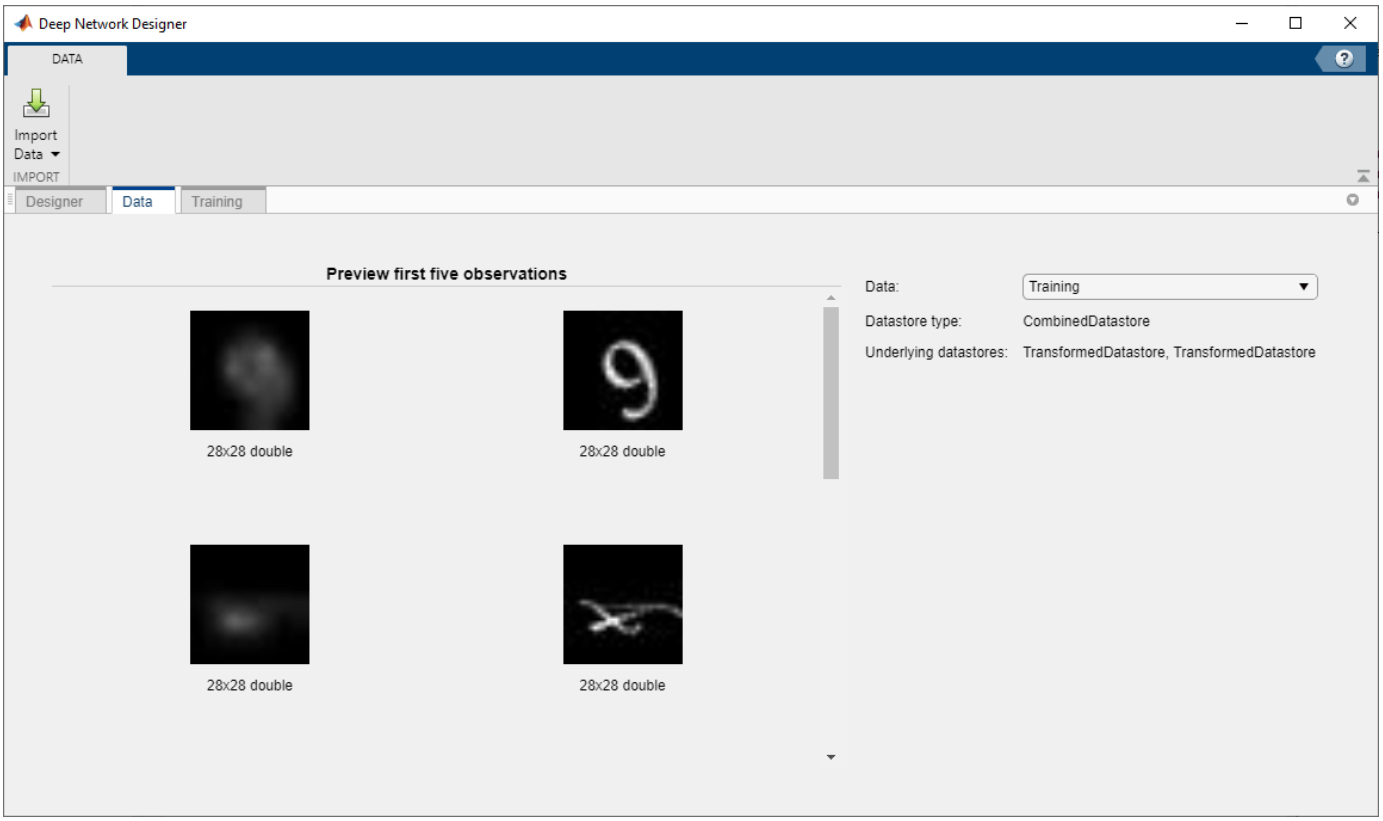
Import Data

Import the training and validation data into Deep Network Designer.

In the **Data** tab, click **Import Data** > **Import Datastore** and select `dsTrain` as the training data and `dsVal` as the validation data. Import both datastores by clicking **Import**.



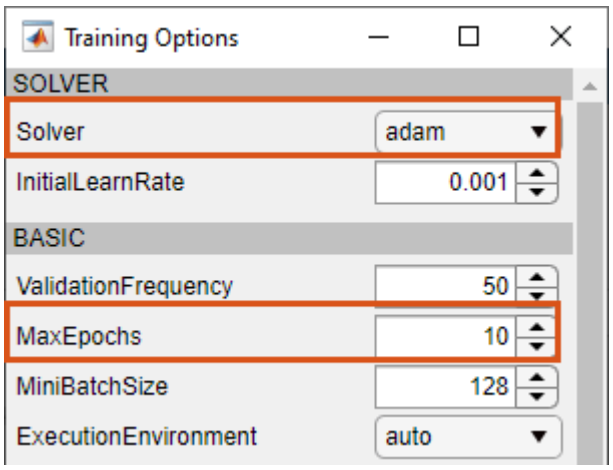
Deep Network Designer displays the pairs of images in the combined datastore. The upscaled low resolution input images are on the left, and the original high resolution response images are on the right. The network learns how to map between the input and the response images.



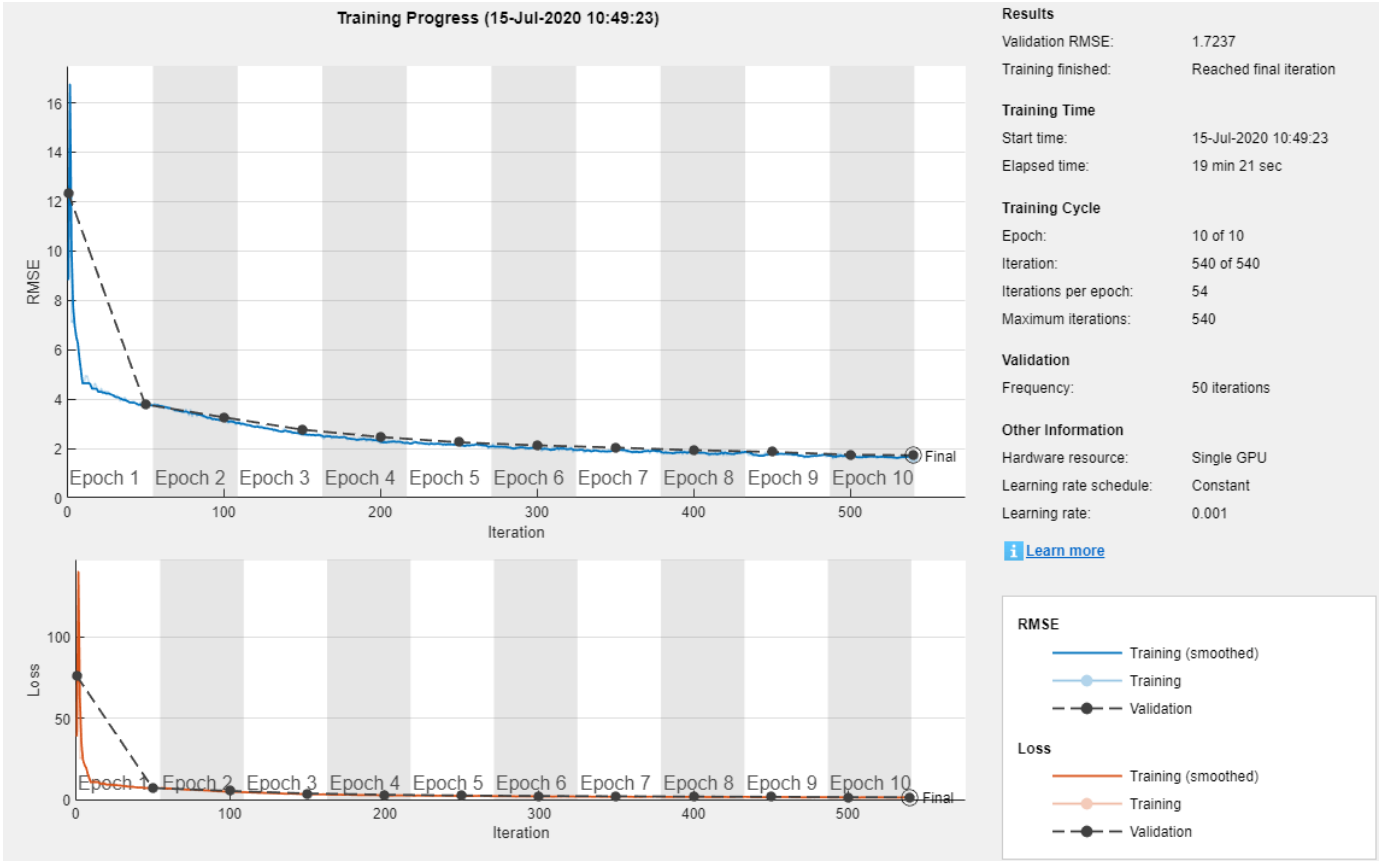
Train Network

Select the training options and train the network.

In the **Training** tab, select **Training Options**. From the **Solver** list, select `adam`. Set **MaxEpochs** to `10`. Confirm the training options by clicking **Close**.



Train the network on the combined datastore by clicking **Train**.



As the network learns how to map between the two images the validation root mean squared error (RMSE) decreases.

Once training is complete, click **Export** to export the trained network to the workspace. The trained network is stored in the variable `trainedNetwork_1`.

Test Network

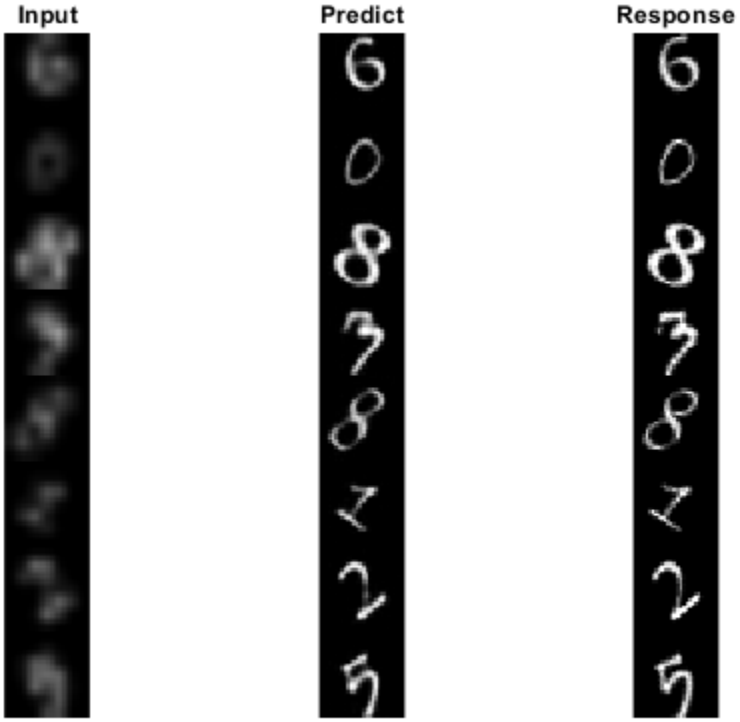
Evaluate the performance of the network using the test data.

Using `predict`, you can test if the network can produce a high resolution image from a low resolution input image that was not included in the training set.

```
ypred = predict(trainedNetwork_1,dsTest);

for i = 1:8
    I(1:2,i) = read(dsTest);
    I(3,i) = {ypred(:,:,:,i)};
end
```

Compare the input, predicted, and response images.

```
subplot(1,3,1)
imshow(imtile(I(1,:),'GridSize',[8,1]))
title('Input')
subplot(1,3,2)
imshow(imtile(I(3,:),'GridSize',[8,1]))
title('Predict')
subplot(1,3,3)
imshow(imtile(I(2,:),'GridSize',[8,1]))
title('Response')
```



The network successfully produces high resolution images from low resolution inputs.

The network in this example is very simple and highly tailored to the digits data set. For an example showing how to create a more complex image-to-image regression network for everyday images, see “Single Image Super-Resolution Using Deep Learning” on page 9-8.

Supporting Functions

```
function dataOut = upsampLowRes(dataIn)
        temp = dataIn;
        temp = imresize(temp,[7,7],'method','bilinear');
        dataOut = {imresize(temp,[28,28],'method','bilinear')};
end
```

See Also

Deep Network Designer | `trainingOptions`

Related Examples

- “Transfer Learning with Deep Network Designer” on page 2-2
- “Build Networks with Deep Network Designer” on page 2-15
- “Import Data into Deep Network Designer” on page 2-39
- “Prepare Datastore for Image-to-Image Regression” on page 19-91
- “Single Image Super-Resolution Using Deep Learning” on page 9-8

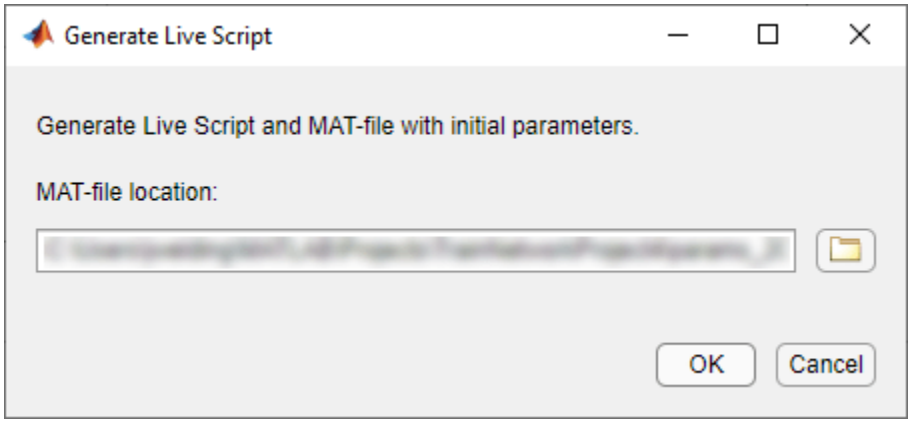
Adapt Code Generated in Deep Network Designer for Use in Experiment Manager

This example shows how to use Experiment Manager to tune the hyperparameters of a network trained in Deep Network Designer.

You can use Deep Network Designer to create a network, import data, and train the network. You can then use Experiment Manager to sweep through a range of hyperparameter values to find optimal training options.

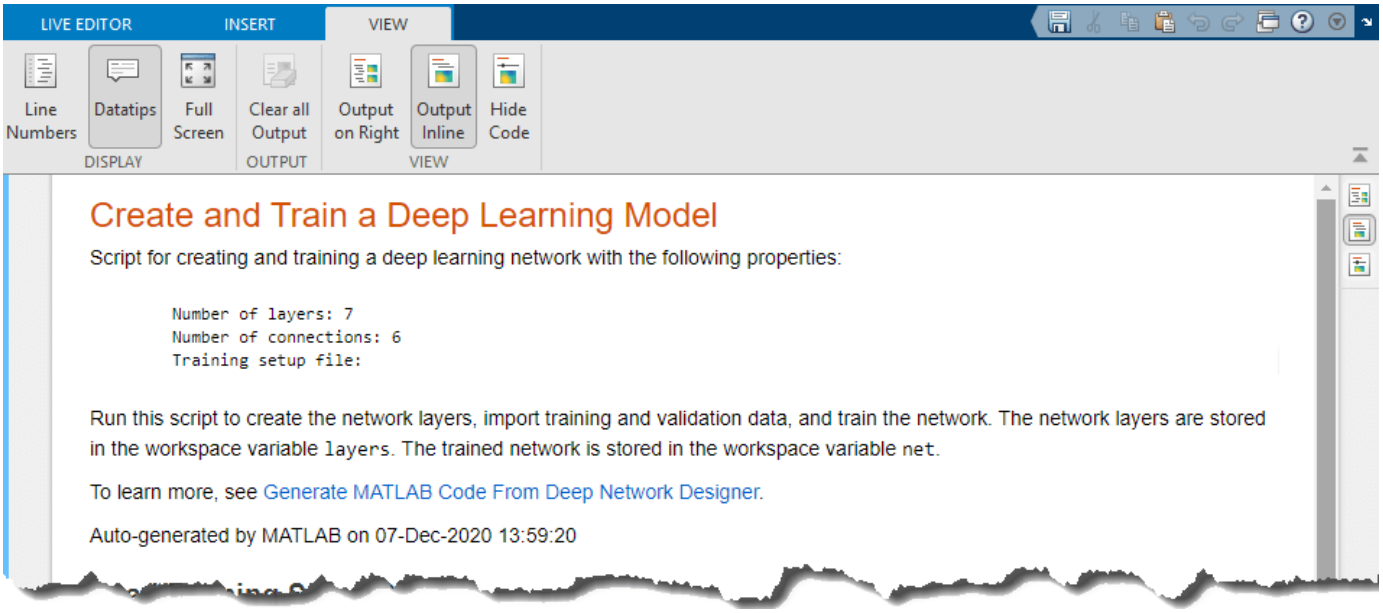
Generate Training Script

To generate a live script to recreate the building and training of a network you construct in Deep Network Designer, on the **Training** tab, select **Export > Generate Code for Training**. Select a MAT file location and click **OK**. For an example showing how to train a classification network in Deep Network Designer, see “Create Simple Image Classification Network Using Deep Network Designer”.



Deep Network Designer creates a live script and a MAT file containing the initial parameters (weights and biases) from your network. If you import data from the workspace into Deep Network Designer, then the generated MAT file contains the data as well.

Running the generated script builds the network (including the learnable parameters from the MAT file), imports the data, sets the training options, and trains the network.



Open Experiment Manager

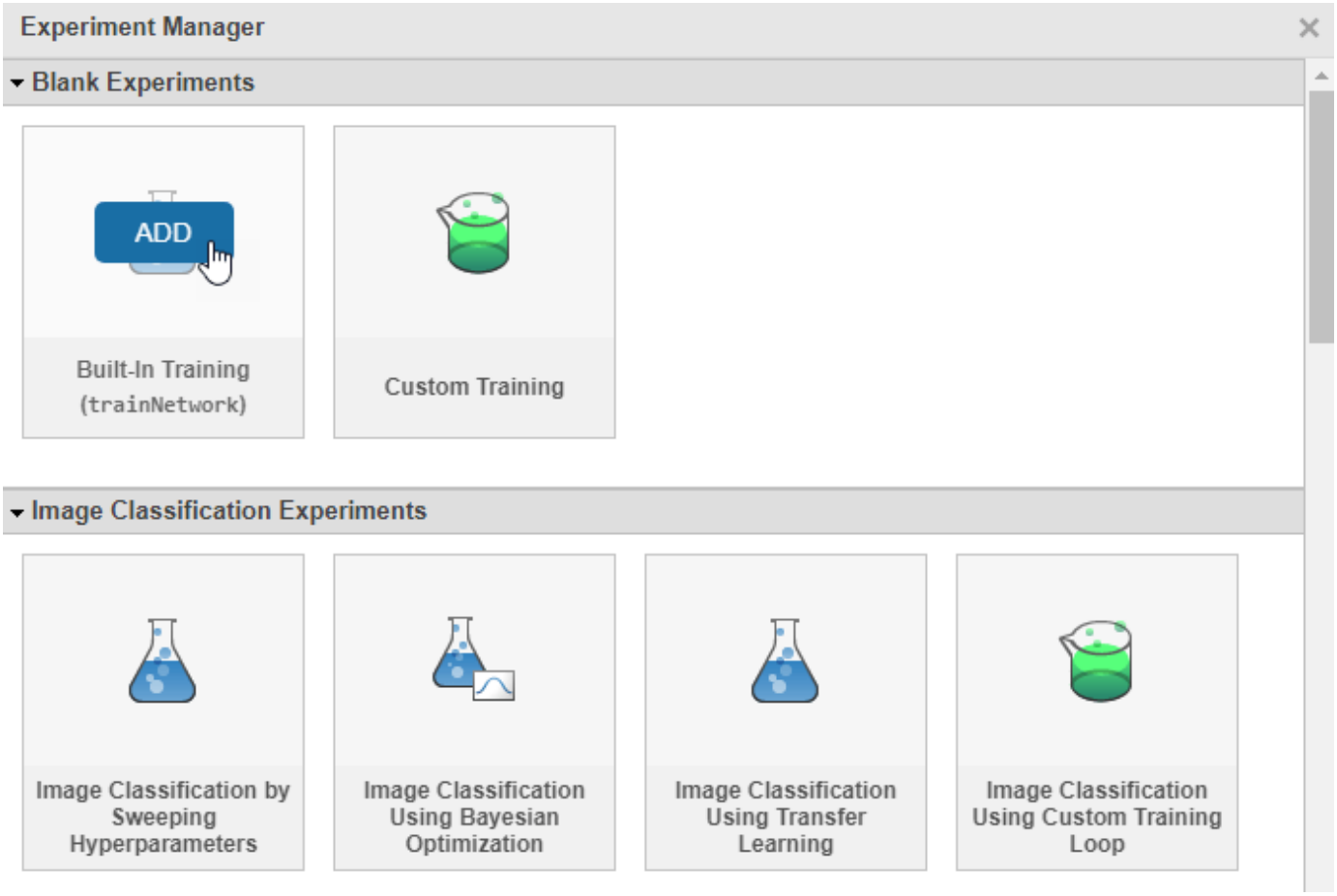
Experiment Manager enables you to create deep learning experiments to train networks under various initial conditions and compare the results. You can use Experiment Manager to tune a network you initially train in Deep Network Designer.

Open Experiment Manager.

```
experimentManager
```

Pause on **Project** and click **Create**. Experiment Manager provides several templates that support many deep learning workflows, including image classification, image regression, sequence classification, semantic segmentation, and custom training loops.

Pause on **Built-In Training** and click **ADD**.



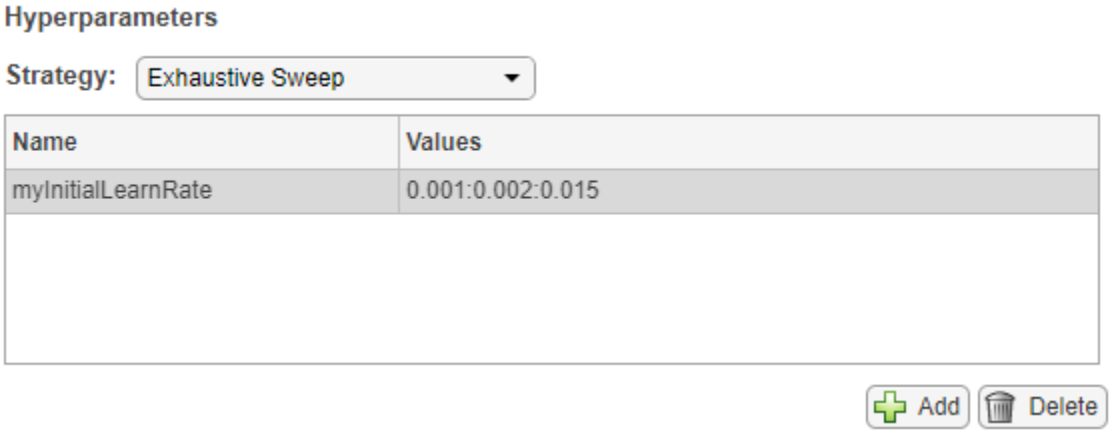
Specify the name and location for the new project and click **Save**. Experiment Manager opens a new experiment in the project. The **Experiment** pane displays the description, hyperparameters, setup function, and metrics that define the experiment.

Add Hyperparameters

In the hyperparameter table, specify the values of the hyperparameters to use in the experiment. When you run the experiment, Experiment Manager trains the network using every combination of the hyperparameter values specified in the table. For this example, sweep over the initial learning rate.

Under **Hyperparameters**, click **Add** to add a new hyperparameter to sweep over.

Add the hyperparameter `myInitialLearnRate`. Set the hyperparameter to sweep the sequence of values `0.001:0.002:0.015`.



Create Setup Function Using Generated Script

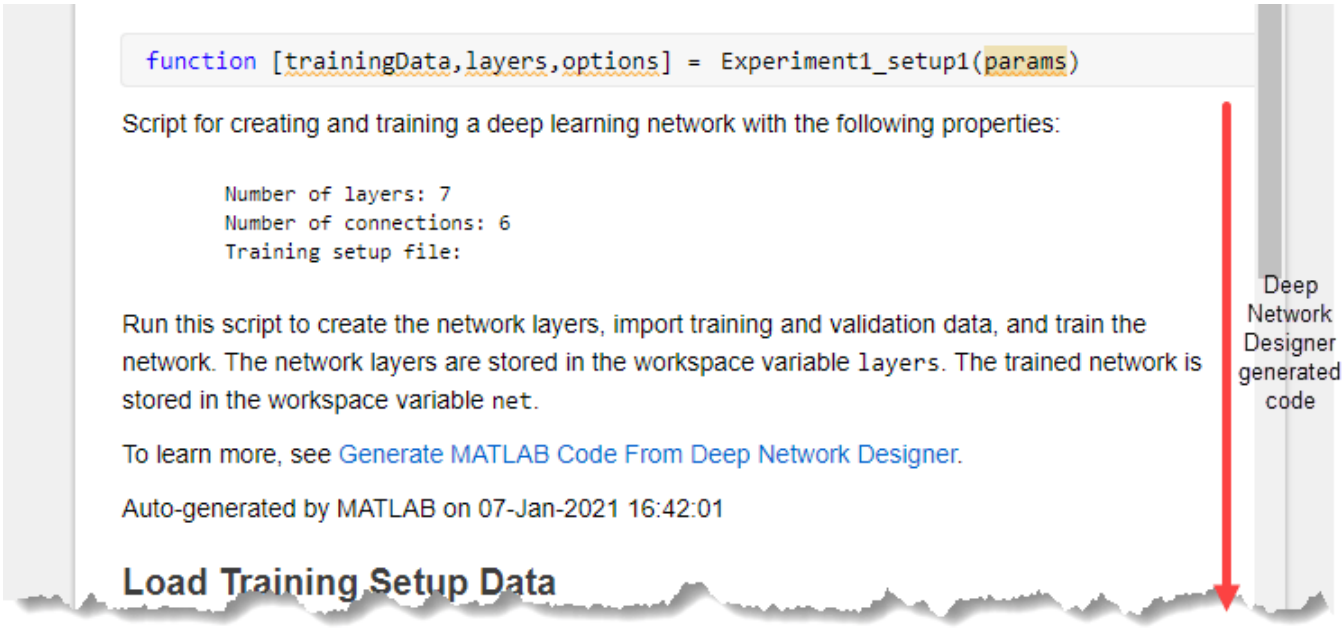
When you create an experiment, Experiment Manager creates a setup function template. To edit this function, under **Setup Function**, click **Edit**.

The empty setup function `Experiment1_setup1` opens in MATLAB Editor. Experiment Manager uses the outputs of this function to call the `trainNetwork` function.

```
function [trainingData,layers,options] = Experiment1_setup1(params)
end
```

The setup function is where you specify the training data, network architecture, and training options for the experiment.

Copy and paste the live script generated by Deep Network Designer inside the setup function.



Adapt Setup Function Input Arguments

Adapt the script for use in Experiment Manager by changing the function input arguments to match the variable names in the generated script. The input arguments to `Experiment1_setup1` must match those the generated script uses in the call to `trainNetwork`.

In the script generated by Deep Network Designer, find the variable names of the data, network, and training options at the bottom of the generated live script in the call to `trainNetwork`. Change the setup function input arguments to match. For example, if your generated live script calls `trainNetwork` with data `imdsTrain`, network `lgraph`, and training options `opts`, then you must make the following changes in the experiment setup function input arguments:

- Change `trainingData` to `imdsTrain`.
- Change `layers` to `lgraph`.
- Change `options` to `opts`.

```
function [trainingData,layers,options] = Experiment1_setup1(params)
```

```
function [imdsTrain,lgraph,opts] = Experiment1_setup1(params)
```

You can check to see if your input arguments need changing by looking for the yellow underline in the setup function input arguments.

Adapt Training Options

Change the training options so that Experiment Manager conducts a hyperparameter sweep of the learning rate.

- Set the initial learning rate to `params.myInitialLearnRate`.

- Optionally, hide the output information by adding the additional name-value argument `"Verbose",false`.

```
opts = trainingOptions("sgdm",...
    "ExecutionEnvironment","auto",...
    "InitialLearnRate",params.myInitialLearnRate,...
    "MaxEpochs",5,...
    "Shuffle","every-epoch",...
    "Plots","training-progress",...
    "ValidationData",augimdsValidation, ...
    "Verbose",false);
```

Remove Call to `trainNetwork`

Experiment Manager uses the outputs of the setup function to call the `trainNetwork` function. Remove the call to `trainNetwork` from the copied and pasted generated code.

The setup function is now ready. Click **Save** to save your edited setup function.

Run Experiment

In Experiment Manager, run the experiment by clicking **Run**. When you run the experiment, Experiment Manager trains the network defined by the setup function. Each trial uses one of the learning rates specified in the hyperparameter table.

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.

A table of results displays the accuracy and loss for each trial. When the experiment finishes, you can sort the trials by the accuracy or loss metrics to see which trial performs the best. In this experiment, Trial 6, with an initial learning rate of 0.0110, has the highest validation accuracy.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
6	Complete	100.0%	0 hr 3 min 39 sec	0.0110	100.0000	0.0200	99.3500	0.0429
8	Complete	100.0%	0 hr 3 min 33 sec	0.0150	100.0000	0.0191	99.3000	0.0418
7	Complete	100.0%	0 hr 3 min 37 sec	0.0130	100.0000	0.0197	99.2500	0.0418
5	Complete	100.0%	0 hr 3 min 19 sec	0.0090	100.0000	0.0255	99.2000	0.0488
4	Complete	100.0%	0 hr 2 min 55 sec	0.0070	100.0000	0.0354	98.9500	0.0594
3	Complete	100.0%	0 hr 2 min 57 sec	0.0050	100.0000	0.0566	98.7500	0.0794
2	Complete	100.0%	0 hr 2 min 49 sec	0.0030	98.4375	0.1081	98.1500	0.1243
1	Complete	100.0%	0 hr 2 min 50 sec	0.0010	92.1875	0.3411	94.4000	0.3220

To close the experiment, in the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

See Also

Deep Network Designer | **Experiment Manager** | `trainingOptions` | `trainNetwork`

Related Examples

- “Transfer Learning with Deep Network Designer” on page 2-2

- “Build Networks with Deep Network Designer” on page 2-15
- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Try Multiple Pretrained Networks for Transfer Learning” on page 6-36
- “Import Data into Deep Network Designer” on page 2-39

Deep Learning with Images

Classify Webcam Images Using Deep Learning

This example shows how to classify images from a webcam in real time using the pretrained deep convolutional neural network GoogLeNet.

Use MATLAB®, a simple webcam, and a deep neural network to identify objects in your surroundings. This example uses GoogLeNet, a pretrained deep convolutional neural network (CNN or ConvNet) that has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). You can download GoogLeNet and use MATLAB to continuously process the camera images in real time.

GoogLeNet has learned rich feature representations for a wide range of images. It takes the image as input and provides a label for the object in the image and the probabilities for each of the object categories. You can experiment with objects in your surroundings to see how accurately GoogLeNet classifies images. To learn more about the network's object classification, you can show the scores for the top five classes in real time, instead of just the final class decision.

Load Camera and Pretrained Network

Connect to the camera and load a pretrained GoogLeNet network. You can use any pretrained network at this step. The example requires MATLAB Support Package for USB Webcams, and Deep Learning Toolbox™ Model *for GoogLeNet Network*. If you do not have the required support packages installed, then the software provides a download link.

```
camera = webcam;
net = googlenet;
```

If you want to run the example again, first run the command `clear camera` where `camera` is the connection to the webcam. Otherwise, you see an error because you cannot create another connection to the same webcam.

Classify Snapshot from Camera

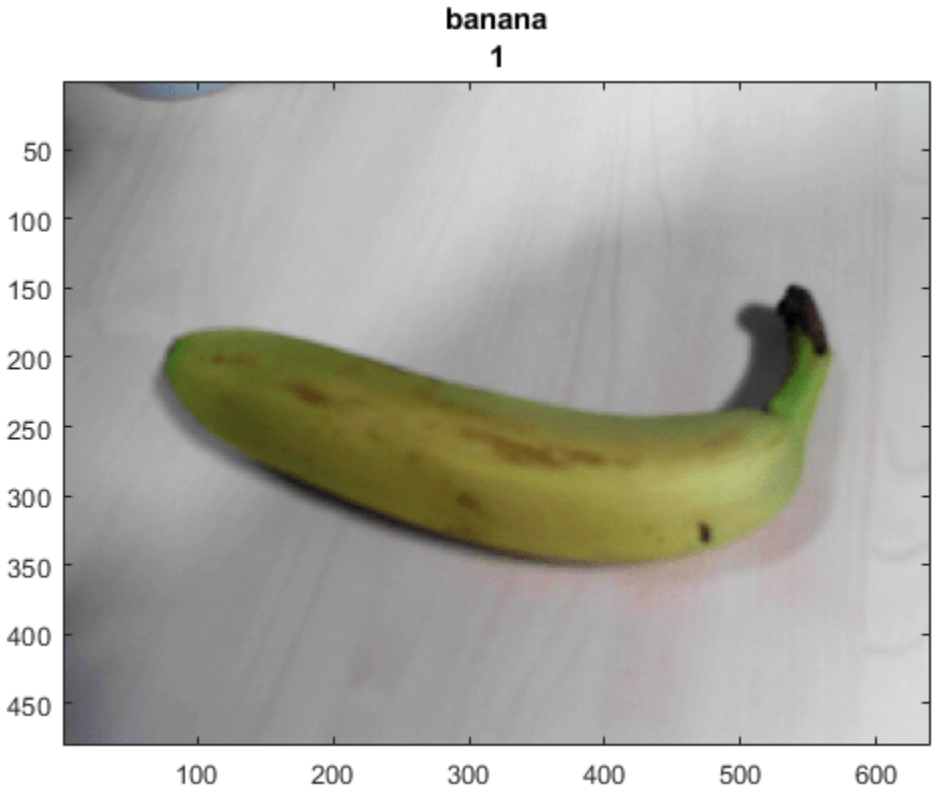
To classify an image, you must resize it to the input size of the network. Get the first two elements of the `InputSize` property of the image input layer of the network. The image input layer is the first layer of the network.

```
inputSize = net.Layers(1).InputSize(1:2)
```

```
inputSize =

   224   224
```

Display the image from the camera with the predicted label and its probability. You must resize the image to the input size of the network before calling `classify`.

```
figure
im = snapshot(camera);
image(im)
im = imresize(im,inputSize);
[label,score] = classify(net,im);
title({char(label),num2str(max(score),2)});
```



Continuously Classify Images from Camera

To classify images from the camera continuously, include the previous steps inside a loop. Run the loop while the figure is open. To stop the live prediction, simply close the figure. Use `drawnow` at the end of each iteration to update the figure.

```
h = figure;

while ishandle(h)
    im = snapshot(camera);
    image(im)
    im = imresize(im,inputSize);
    [label,score] = classify(net,im);
    title({char(label), num2str(max(score),2)});
    drawnow
end
```

Display Top Predictions

The predicted classes can change rapidly. Therefore, it can be helpful to display the top predictions together. You can display the top five predictions and their probabilities by plotting the classes with the highest prediction scores.

Classify a snapshot from the camera. Display the image from the camera with the predicted label and its probability. Display a histogram of the probabilities of the top five predictions by using the `score` output of the `classify` function.

Create the figure window. First, resize the window to have twice the width, and create two subplots.

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
```

In the left subplot, display the image and classification together.

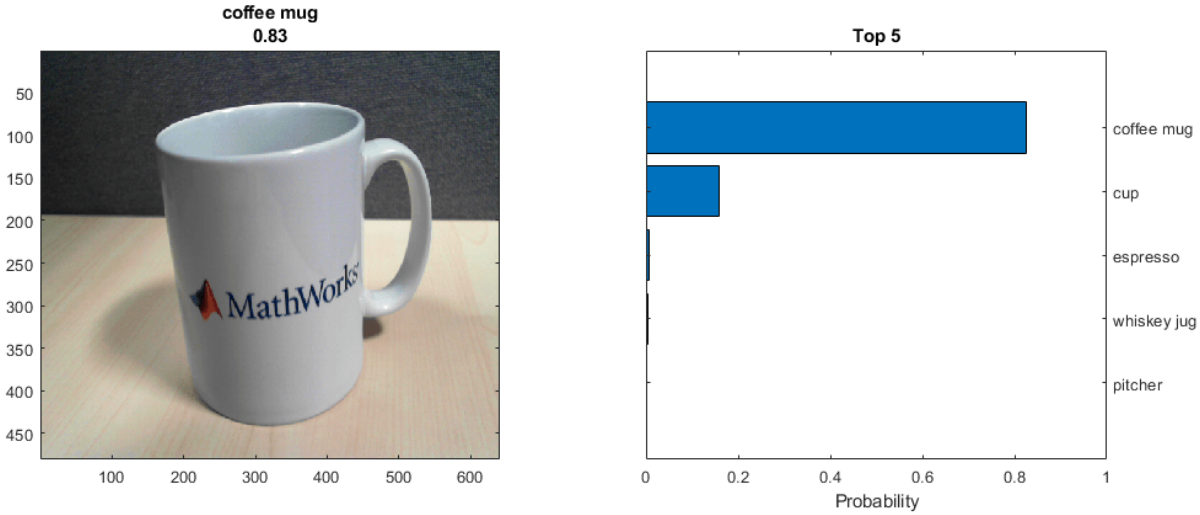
```
im = snapshot(camera);
image(ax1,im)
im = imresize(im,inputSize);
[label,score] = classify(net,im);
title(ax1,{char(label),num2str(max(score),2)});
```

Select the top five predictions by selecting the classes with the highest scores.

```
[~,idx] = sort(score,'descend');
idx = idx(5:-1:1);
classes = net.Layers(end).Classes;
classNamesTop = string(classes(idx));
scoreTop = score(idx);
```

Display the top five predictions as a histogram.

```
barh(ax2,scoreTop)
xlim(ax2,[0 1])
title(ax2,'Top 5')
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop)
ax2.YAxisLocation = 'right';
```



Continuously Classify Images and Display Top Predictions

To classify images from the camera continuously and display the top predictions, include the previous steps inside a loop. Run the loop while the figure is open. To stop the live prediction, simply close the figure. Use `drawnow` at the end of each iteration to update the figure.

Create the figure window. First resize the window, to have twice the width, and create two subplots. To prevent the axes from resizing, set `PositionConstraint` property to `'innerposition'`.

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
ax2.PositionConstraint = 'innerposition';
```

Continuously display and classify images together with a histogram of the top five predictions.

```
while ishandle(h)
    % Display and classify the image
    im = snapshot(camera);
    image(ax1,im)
    im = imresize(im,inputSize);
    [label,score] = classify(net,im);
    title(ax1,{char(label),num2str(max(score),2)});

    % Select the top five predictions
    [~,idx] = sort(score,'descend');
    idx = idx(5:-1:1);
    scoreTop = score(idx);
    classNamesTop = string(classes(idx));

    % Plot the histogram
    barh(ax2,scoreTop)
    title(ax2,'Top 5')
    xlabel(ax2,'Probability')
    xlim(ax2,[0 1])
    yticklabels(ax2,classNamesTop)
    ax2.YAxisLocation = 'right';

    drawnow
end
```

See Also

`googlenet` | `vgg19` | `classify`

Related Examples

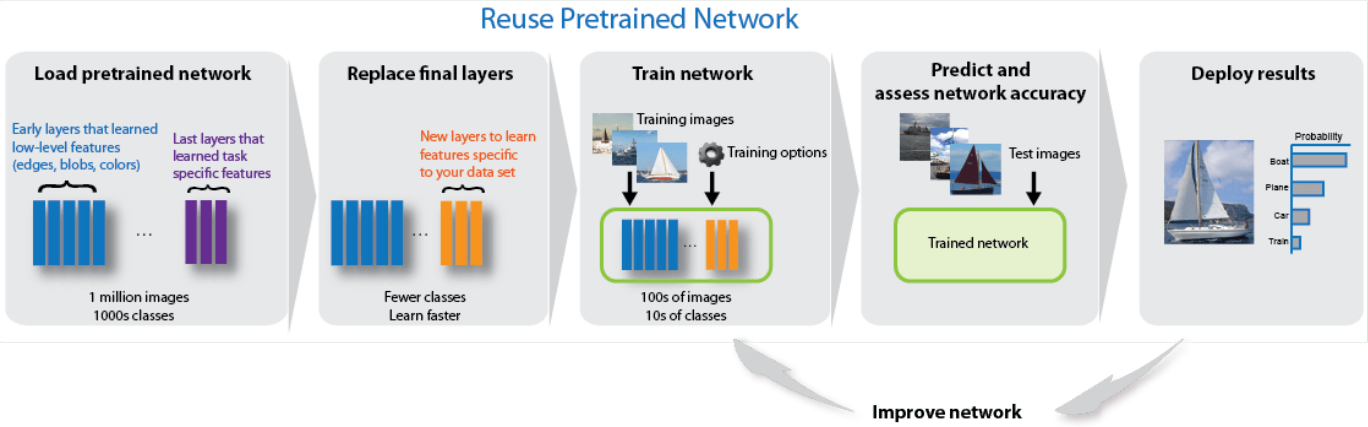
- “Transfer Learning Using Pretrained Network” on page 3-33
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Train Deep Learning Network to Classify New Images

This example shows how to use transfer learning to retrain a convolutional neural network to classify a new set of images.

Pretrained image classification networks have been trained on over a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. The networks have learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network from scratch with randomly initialized weights. You can quickly transfer learned features to a new task using a smaller number of training images.



Load Data

Unzip and load the new images as an image datastore. This very small data set contains only 75 images. Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7);
```

Load Pretrained Network

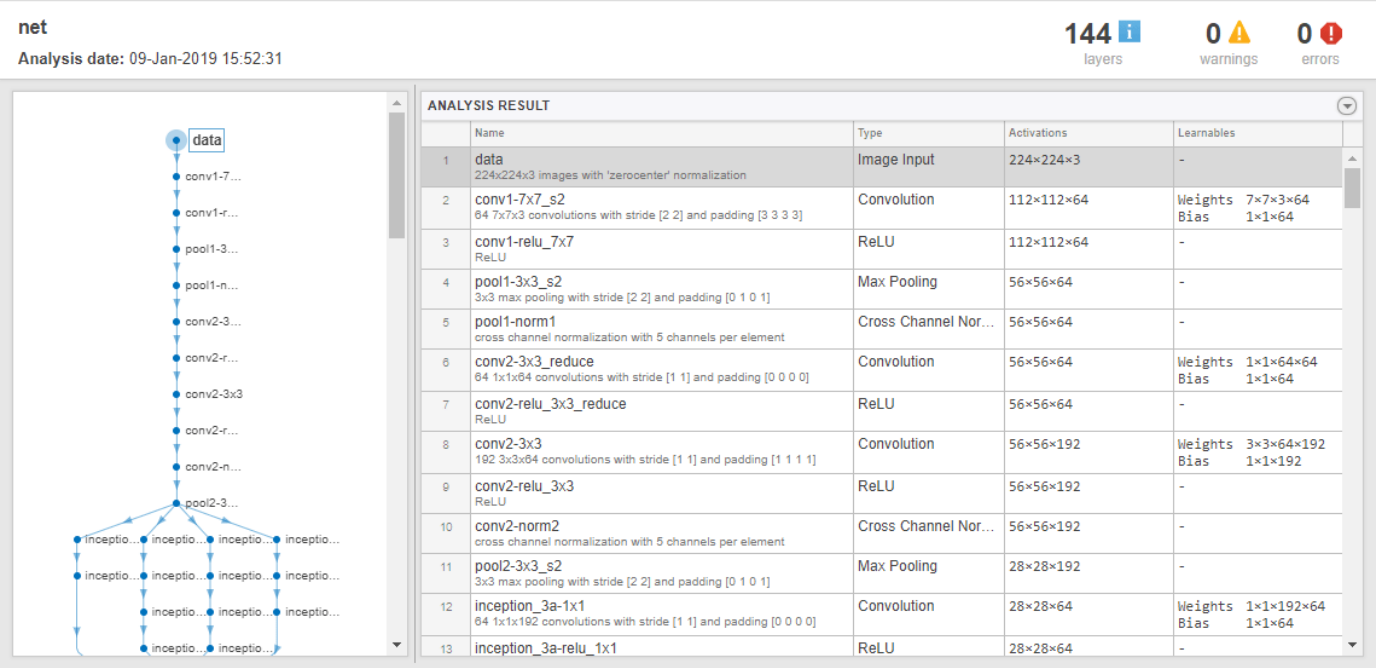
Load a pretrained GoogLeNet network. If the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

To try a different pretrained network, open this example in MATLAB® and select a different network. For example, you can try `squeezenet`, a network that is even faster than `googlenet`. You can run this example with other pretrained networks. For a list of all available networks, see “Load Pretrained Networks” on page 1-10.

```
net = googlenet;
```

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
analyzeNetwork(net)
```



The first element of the `Layers` property of the network is the image input layer. For a GoogLeNet network, this layer requires input images of size 224-by-224-by-3, where 3 is the number of color channels. Other networks can require input images with different sizes. For example, the Xception network requires images of size 299-by-299-by-3.

```
net.Layers(1)
```

```
ans =
  ImageInputLayer with properties:

                     Name: 'data'
                InputSize: [224 224 3]

   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                     Mean: [224×224×3 single]
```

```
inputSize = net.Layers(1).InputSize;
```

Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, `'loss3-classifier'` and `'output'` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Convert the trained network to a layer graph.

```
lgraph = layerGraph(net);
```

Find the names of the two layers to replace. You can do this manually or you can use the supporting function findLayersToReplace to find these layers automatically.

```
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
[learnableLayer,classLayer]
```

```
ans =
  1×2 Layer array with layers:

     1   'loss3-classifier'   Fully Connected         1000 fully connected layer
     2   'output'             Classification Output   crossentropyex with 'tench' and 999 other c
```

In most networks, the last layer with learnable weights is a fully connected layer. Replace this fully connected layer with a new fully connected layer with the number of outputs equal to the number of classes in the new data set (5, in this example). In some networks, such as SqueezeNet, the last learnable layer is a 1-by-1 convolutional layer instead. In this case, replace the convolutional layer with a new convolutional layer with the number of filters equal to the number of classes. To learn faster in the new layer than in the transferred layers, increase the learning rate factors of the layer.

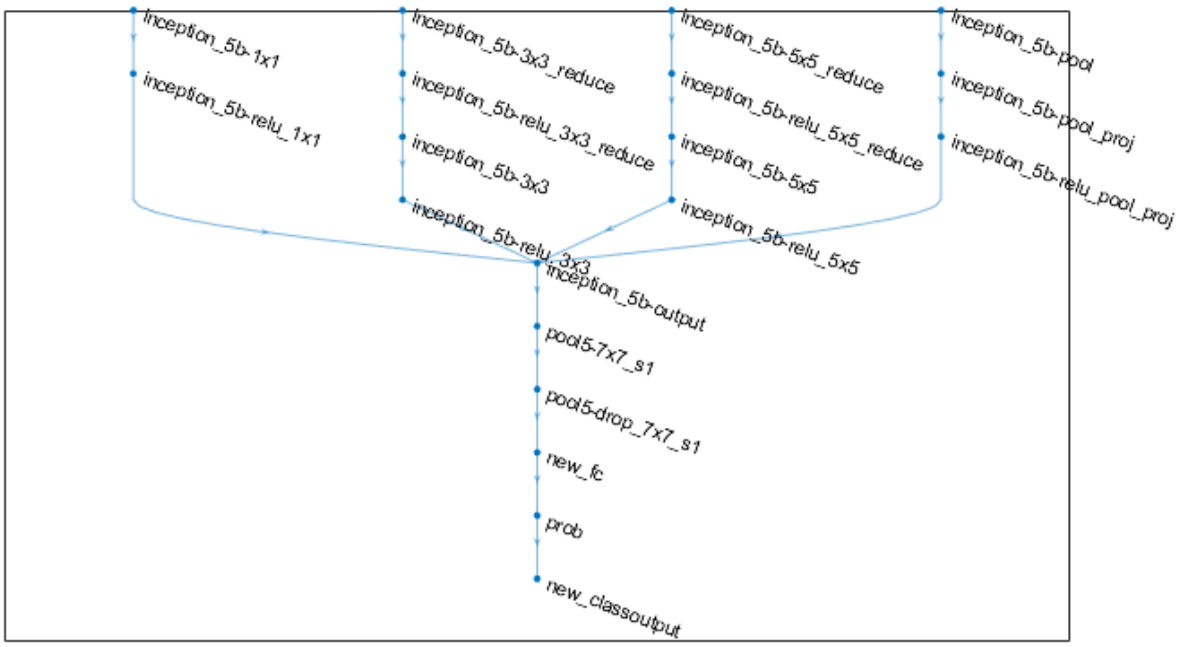
```
numClasses = numel(categories(imdsTrain.Labels));

if isa(learnableLayer,'nnet.cnn.layer.FullyConnectedLayer')
    newLearnableLayer = fullyConnectedLayer(numClasses, ...
        'Name','new_fc', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);

elseif isa(learnableLayer,'nnet.cnn.layer.Convolution2DLayer')
    newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);
end

lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);
```

To check that the new layers are connected correctly, plot the new layer graph and zoom in on the last layers of the network.

```
figure('Units','normalized','Position',[0.3 0.3 0.4 0.4]);
plot(lgraph)
ylim([0,10])
```



Freeze Initial Layers

The network is now ready to be retrained on the new set of images. Optionally, you can "freeze" the weights of earlier layers in the network by setting the learning rates in those layers to zero. During training, `trainNetwork` does not update the parameters of the frozen layers. Because the gradients of the frozen layers do not need to be computed, freezing the weights of many initial layers can significantly speed up network training. If the new data set is small, then freezing earlier network layers can also prevent those layers from overfitting to the new data set.

Extract the layers and connections of the layer graph and select which layers to freeze. In GoogLeNet, the first 10 layers make out the initial 'stem' of the network. Use the supporting function freezeWeights to set the learning rates to zero in the first 10 layers. Use the supporting function createLgraphUsingConnections to reconnect all the layers in the original order. The new layer graph contains the same layers, but with the learning rates of the earlier layers set to zero.

```
layers = lgraph.Layers;
connections = lgraph.Connections;

layers(1:10) = freezeWeights(layers(1:10));
lgraph = createLgraphUsingConnections(layers,connections);
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastore have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis and randomly translate them up to 30 pixels and scale them up to 10% horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange, ...
    'RandXScale',scaleRange, ...
    'RandYScale',scaleRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. Set `InitialLearnRate` to a small value to slow down learning in the transferred layers that are not already frozen. In the previous step, you increased the learning rate factors for the last learnable layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning in the new layers, slower learning in the middle layers, and no learning in the earlier, frozen layers.

Specify the number of epochs to train for. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. Compute the validation accuracy once per epoch.

```
miniBatchSize = 10;
valFrequency = floor(numel(augimdsTrain.Files)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',3e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',valFrequency, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network using the training data. By default, `trainNetwork` uses a GPU if one is available. This requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Because the data set is so small, training is fast.

```
net = trainNetwork(augimdsTrain,lgraph,options);
```

Classify Validation Images

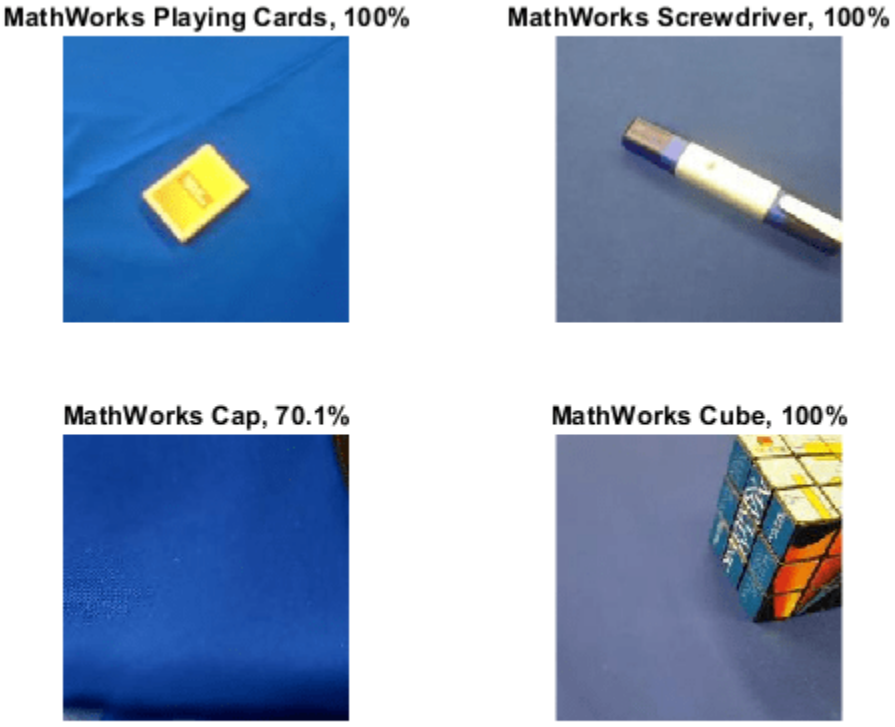
Classify the validation images using the fine-tuned network, and calculate the classification accuracy.

```
[YPred,probs] = classify(net,augimdsValidation);
accuracy = mean(YPred == imdsValidation.Labels)
```

```
accuracy = 0.9000
```

Display four sample validation images with predicted labels and the predicted probabilities of the images having those labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label) + ", " + num2str(100*max(probs(idx(i),:)),3) + "%");
end
```



References

[1] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

[2] *BVLC GoogLeNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

See Also

`vgg16` | `vgg19` | `alexnet` | `importCaffeNetwork` | `importCaffeLayers` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `googlenet` | `analyzeNetwork`

Related Examples

- “Convert Classification Network into Regression Network” on page 3-70
- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Transfer Learning Using Pretrained Network” on page 3-33
- “Train Residual Network for Image Classification” on page 3-13

Train Residual Network for Image Classification

This example shows how to create a deep learning neural network with residual connections and train it on CIFAR-10 data. Residual connections are a popular element in convolutional neural network architectures. Using residual connections improves gradient flow through the network and enables training of deeper networks.

For many applications, using a network that consists of a simple sequence of layers is sufficient. However, some applications require networks with a more complex graph structure in which layers can have inputs from multiple layers and outputs to multiple layers. These types of networks are often called directed acyclic graph (DAG) networks. A residual network is a type of DAG network that has residual (or shortcut) connections that bypass the main network layers. Residual connections enable the parameter gradients to propagate more easily from the output layer to the earlier layers of the network, which makes it possible to train deeper networks. This increased network depth can result in higher accuracies on more difficult tasks.

To create and train a network with a graph structure, follow these steps.

- Create a `LayerGraph` object using `layerGraph`. The layer graph specifies the network architecture. You can create an empty layer graph and then add layers to it. You can also create a layer graph directly from an array of network layers. In this case, `layerGraph` connects the layers in the array one after the other.
- Add layers to the layer graph using `addLayers`, and remove layers from the graph using `removeLayers`.
- Connect layers to other layers using `connectLayers`, and disconnect layers from other layers using `disconnectLayers`.
- Plot the network architecture using `plot`.
- Train the network using `trainNetwork`. The trained network is a `DAGNetwork` object.
- Perform classification and prediction on new data using `classify` and `predict`.

This example shows how to build a residual network from scratch. You can also create residual networks using the `resnetLayers` function. This function allows you to quickly construct residual networks for image classification tasks.

You can also load pretrained networks for image classification. For more information, see “Pretrained Deep Neural Networks” on page 1-8.

Prepare Data

Download the CIFAR-10 data set [1]. The data set contains 60,000 images. Each image is 32-by-32 in size and has three color channels (RGB). The size of the data set is 175 MB. Depending on your internet connection, the download process can take time.

```
datadir = tempdir;
downloadCIFARData(datadir);
```

Load the CIFAR-10 training and test images as 4-D arrays. The training set contains 50,000 images and the test set contains 10,000 images. Use the CIFAR-10 test images for network validation.

```
[XTrain,YTrain,XValidation,YValidation] = loadCIFARData(datadir);
```

You can display a random sample of the training images using the following code.

```
figure;
idx = randperm(size(XTrain,4),20);
im = imtile(XTrain(:,:,:,idx),'ThumbnailSize',[96,96]);
imshow(im)
```

Create an `augmentedImageDatastore` object to use for network training. During training, the datastore randomly flips the training images along the vertical axis and randomly translates them up to four pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(imageSize,XTrain,YTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture

The residual network architecture consists of these components:

- A main branch with convolutional, batch normalization, and ReLU layers connected sequentially.
- *Residual connections* that bypass the convolutional units of the main branch. The outputs of the residual connections and convolutional units are added element-wise. When the size of the activations changes, the residual connections must also contain 1-by-1 convolutional layers. Residual connections enable the parameter gradients to flow more easily from the output layer to the earlier layers of the network, which makes it possible to train deeper networks.

Create Main Branch

Start by creating the main branch of the network. The main branch contains five sections.

- An initial section containing the image input layer and an initial convolution with activation.
- Three stages of convolutional layers with different feature sizes (32-by-32, 16-by-16, and 8-by-8). Each stage contains N convolutional units. In this part of the example, $N = 2$. Each convolutional unit contains two 3-by-3 convolutional layers with activations. The `netWidth` parameter is the network width, defined as the number of filters in the convolutional layers in the first stage of the network. The first convolutional units in the second and third stages downsample the spatial dimensions by a factor of two. To keep the amount of computation required in each convolutional layer roughly the same throughout the network, increase the number of filters by a factor of two each time you perform spatial downsampling.
- A final section with global average pooling, fully connected, softmax, and classification layers.

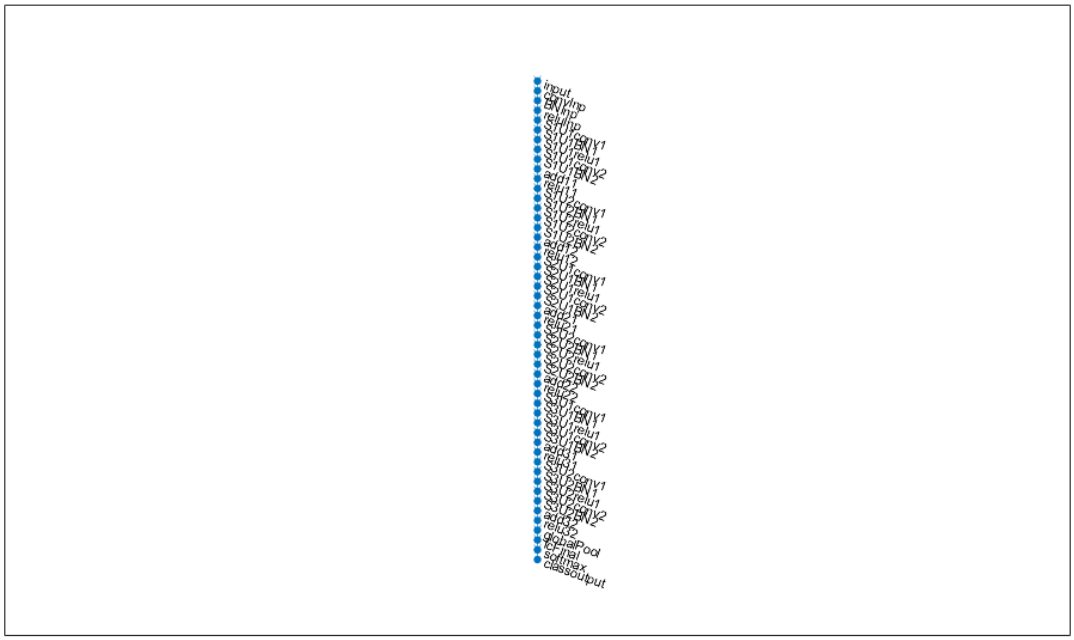
Use `convolutionalUnit(numF,stride,tag)` to create a convolutional unit. `numF` is the number of convolutional filters in each layer, `stride` is the stride of the first convolutional layer of the unit, and `tag` is a character array to prepend to the layer names. The `convolutionalUnit` function is defined at the end of the example.

Give unique names to all the layers. The layers in the convolutional units have names starting with `'SjUk'`, where `j` is the stage index and `k` is the index of the convolutional unit within that stage. For example, `'S2U1'` denotes stage 2, unit 1.

```
netWidth = 16;
layers = [
    imageInputLayer([32 32 3],'Name','input')
    convolution2dLayer(3,netWidth,'Padding','same','Name','convInp')
    batchNormalizationLayer('Name','BNInp')
    reluLayer('Name','reluInp')

    convolutionalUnit(netWidth,1,'S1U1')
    additionLayer(2,'Name','add11')
    reluLayer('Name','relu11')
    convolutionalUnit(netWidth,1,'S1U2')
    additionLayer(2,'Name','add12')
    reluLayer('Name','relu12')

    convolutionalUnit(2*netWidth,2,'S2U1')
    additionLayer(2,'Name','add21')
    reluLayer('Name','relu21')
    convolutionalUnit(2*netWidth,1,'S2U2')
    additionLayer(2,'Name','add22')
    reluLayer('Name','relu22')

    convolutionalUnit(4*netWidth,2,'S3U1')
    additionLayer(2,'Name','add31')
    reluLayer('Name','relu31')
    convolutionalUnit(4*netWidth,1,'S3U2')
    additionLayer(2,'Name','add32')
    reluLayer('Name','relu32')

    averagePooling2dLayer(8,'Name','globalPool')
    fullyConnectedLayer(10,'Name','fcFinal')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classoutput')
    ];
```

Create a layer graph from the layer array. `layerGraph` connects all the layers in `layers` sequentially. Plot the layer graph.

```
lgraph = layerGraph(layers);
figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
plot(lgraph);
```



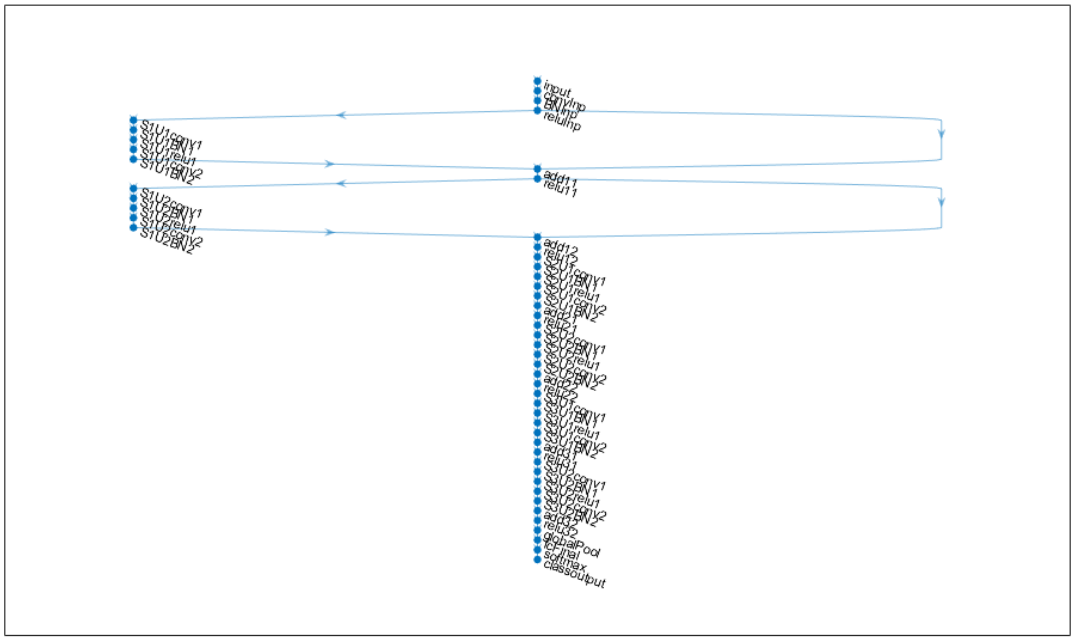
Create Residual Connections

Add residual connections around the convolutional units. Most residual connections perform no operations and simply add element-wise to the outputs of the convolutional units.

Create the residual connection from the `'reluInp'` to the `'add11'` layer. Because you specified the number of inputs to the addition layer to be two when you created the layer, the layer has two inputs with the names `'in1'` and `'in2'`. The final layer of the first convolutional unit is already connected to the `'in1'` input. The addition layer then sums the outputs of the first convolutional unit and the `'reluInp'` layer.

In the same way, connect the `'relu11'` layer to the second input of the `'add12'` layer. Check that you have connected the layers correctly by plotting the layer graph.

```
lgraph = connectLayers(lgraph,'reluInp','add11/in2');
lgraph = connectLayers(lgraph,'relu11','add12/in2');

figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
plot(lgraph);
```



When the layer activations in the convolutional units change size (that is, when they are downsampled spatially and upsampled in the channel dimension), the activations in the residual connections must also change size. Change the activation sizes in the residual connections by using a 1-by-1 convolutional layer together with its batch normalization layer.

```
skip1 = [
    convolution2dLayer(1,2*netWidth,'Stride',2,'Name','skipConv1')
    batchNormalizationLayer('Name','skipBN1')];
lgraph = addLayers(lgraph,skip1);
lgraph = connectLayers(lgraph,'relu12','skipConv1');
lgraph = connectLayers(lgraph,'skipBN1','add21/in2');
```

Add the identity connection in the second stage of the network.

```
lgraph = connectLayers(lgraph,'relu21','add22/in2');
```

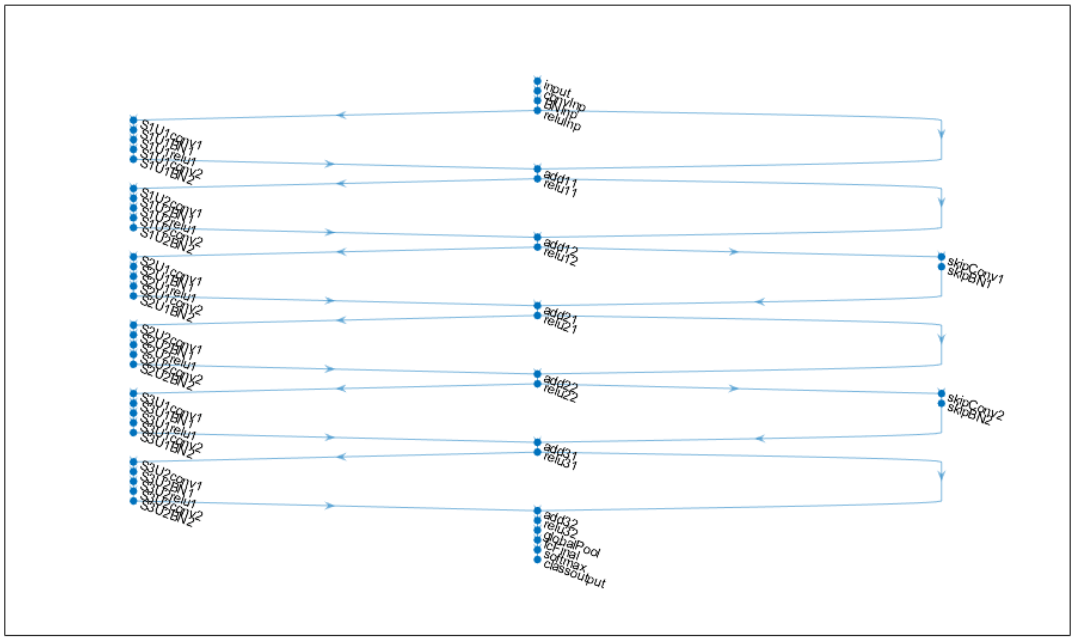
Change the activation size in the residual connection between the second and third stages by another 1-by-1 convolutional layer together with its batch normalization layer.

```
skip2 = [
    convolution2dLayer(1,4*netWidth,'Stride',2,'Name','skipConv2')
    batchNormalizationLayer('Name','skipBN2')];
lgraph = addLayers(lgraph,skip2);
lgraph = connectLayers(lgraph,'relu22','skipConv2');
lgraph = connectLayers(lgraph,'skipBN2','add31/in2');
```

Add the last identity connection and plot the final layer graph.

```
lgraph = connectLayers(lgraph,'relu31','add32/in2');
```

```
figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
plot(lgraph)
```



Create Deeper Network

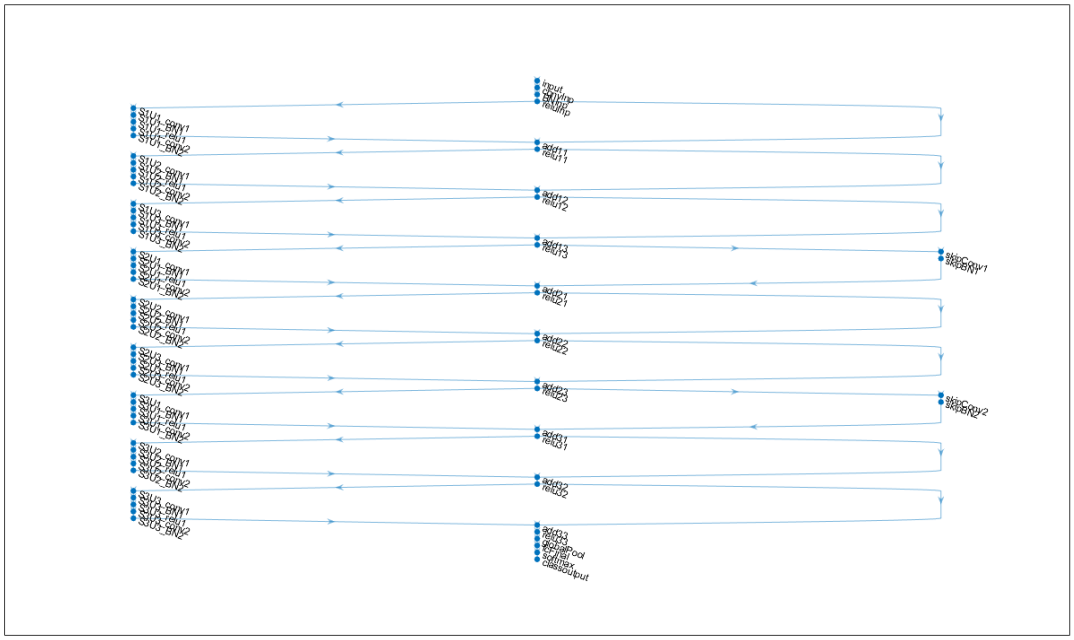
To create a layer graph with residual connections for CIFAR-10 data of arbitrary depth and width, use the supporting function residualCIFARlgraph.

`lgraph = residualCIFARlgraph(netWidth,numUnits,unitType)` creates a layer graph for CIFAR-10 data with residual connections.

- `netWidth` is the network width, defined as the number of filters in the first 3-by-3 convolutional layers of the network.
- `numUnits` is the number of convolutional units in the main branch of network. Because the network consists of three stages where each stage has the same number of convolutional units, `numUnits` must be an integer multiple of 3.
- `unitType` is the type of convolutional unit, specified as `"standard"` or `"bottleneck"`. A standard convolutional unit consists of two 3-by-3 convolutional layers. A bottleneck convolutional unit consists of three convolutional layers: a 1-by-1 layer for downsampling in the channel dimension, a 3-by-3 convolutional layer, and a 1-by-1 layer for upsampling in the channel dimension. Hence, a bottleneck convolutional unit has 50% more convolutional layers than a standard unit, but only half the number of spatial 3-by-3 convolutions. The two unit types have similar computational complexity, but the total number of features propagating in the residual connections is four times larger when using the bottleneck units. The total depth, defined as the maximum number of sequential convolutional and fully connected layers, is 2*`numUnits` + 2 for networks with standard units and 3*`numUnits` + 2 for networks with bottleneck units.

Create a residual network with nine standard convolutional units (three units per stage) and a width of 16. The total network depth is 2*9+2 = 20.

```
numUnits = 9;
netWidth = 16;
lgraph = residualCIFARlgraph(netWidth,numUnits,"standard");
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
plot(lgraph)
```



Train Network

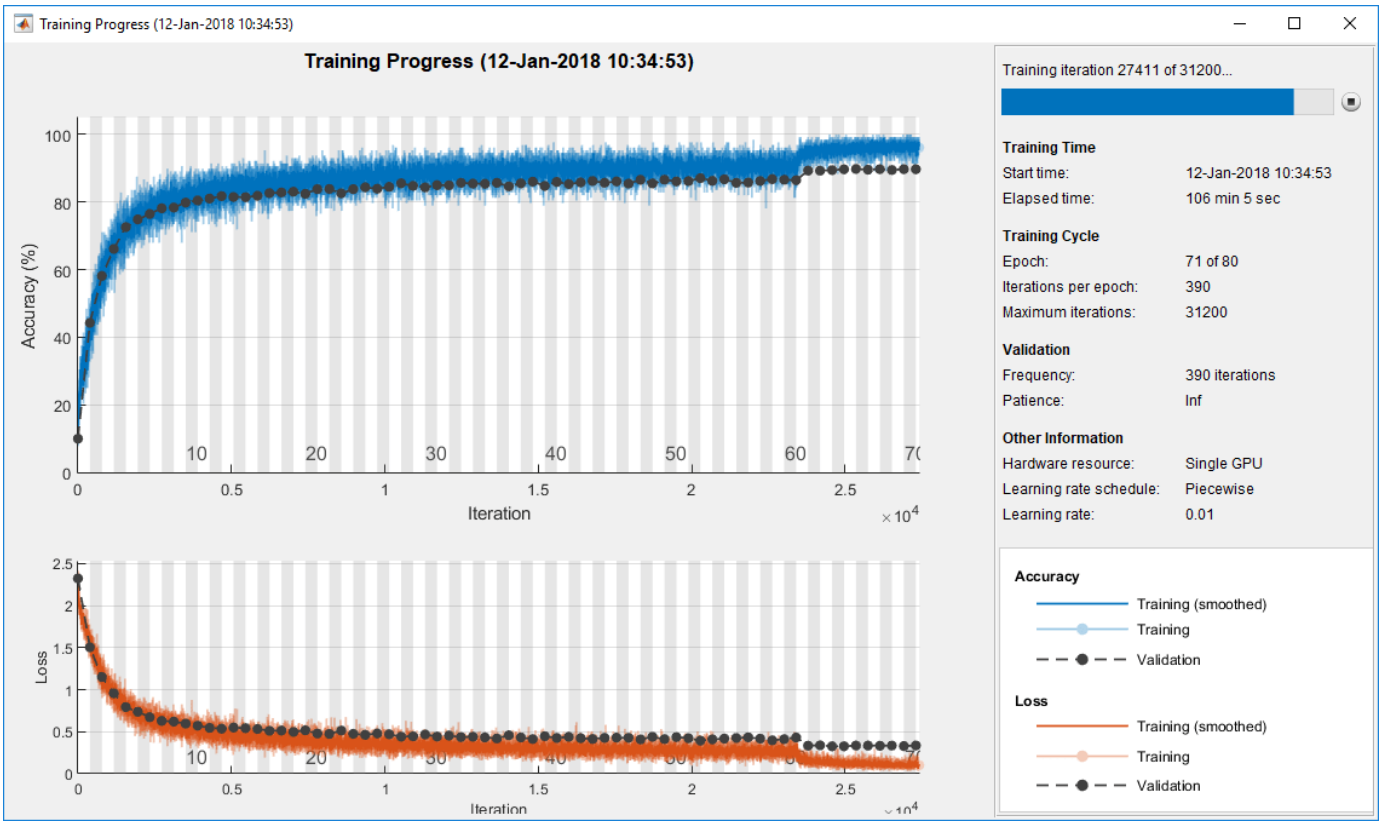
Specify training options. Train the network for 80 epochs. Select a learning rate that is proportional to the mini-batch size and reduce the learning rate by a factor of 10 after 60 epochs. Validate the network once per epoch using the validation data.

```
miniBatchSize = 128;
learnRate = 0.1*miniBatchSize/128;
valFrequency = floor(size(XTrain,4)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'InitialLearnRate',learnRate, ...
    'MaxEpochs',80, ...
    'MiniBatchSize',miniBatchSize, ...
    'VerboseFrequency',valFrequency, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',valFrequency, ...
    'LearnRateSchedule','piecewise', ...
```

```
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',60);
```

To train the network using `trainNetwork`, set the `doTraining` flag to `true`. Otherwise, load a pretrained network. Training the network on a good GPU takes about two hours. If you do not have a GPU, then training takes much longer.

```
doTraining = false;
if doTraining
    trainedNet = trainNetwork(augimdsTrain,lgraph,options);
else
    load('CIFARNet-20-16.mat','trainedNet');
end
```



Evaluate Trained Network

Calculate the final accuracy of the network on the training set (without data augmentation) and validation set.

```
[YValPred,probs] = classify(trainedNet,XValidation);
validationError = mean(YValPred ~= YValidation);
YTrainPred = classify(trainedNet,XTrain);
trainError = mean(YTrainPred ~= YTrain);
disp("Training error: " + trainError*100 + "%")
```

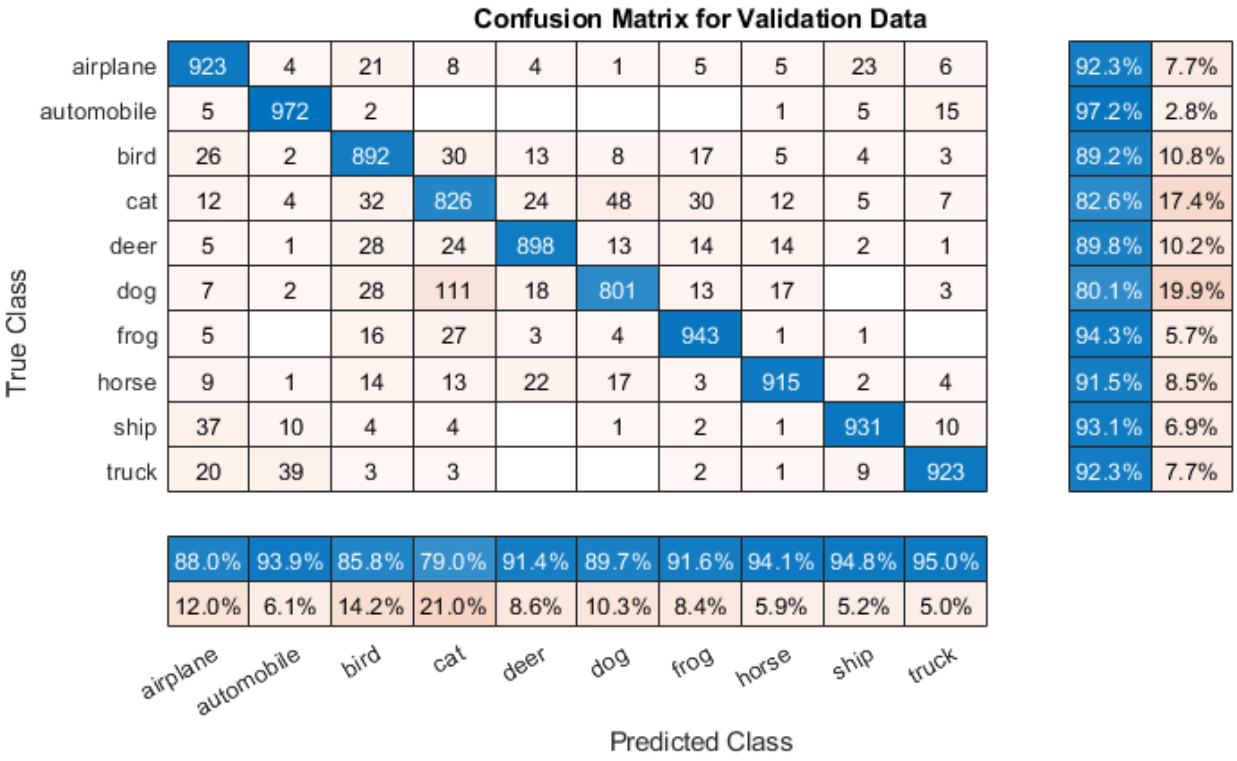
```
Training error: 2.862%
```

```
disp("Validation error: " + validationError*100 + "%")
```

```
Validation error: 9.76%
```

Plot the confusion matrix. Display the precision and recall for each class by using column and row summaries. The network most commonly confuses cats with dogs.

```
figure('Units','normalized','Position',[0.2 0.2 0.4 0.4]);
cm = confusionchart(YValidation,YValPred);
cm.Title = 'Confusion Matrix for Validation Data';
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



You can display a random sample of nine test images together with their predicted classes and the probabilities of those classes using the following code.

```
figure
idx = randperm(size(XValidation,4),9);
for i = 1:numel(idx)
    subplot(3,3,i)
    imshow(XValidation(:,:,:,idx(i)));
    prob = num2str(100*max(probs(idx(i),:)),3);
    predClass = char(YValPred(idx(i)));
    title([predClass,', ',prob,'%'])
end
```

`convolutionalUnit(numF,stride,tag)` creates an array of layers with two convolutional layers and corresponding batch normalization and ReLU layers. `numF` is the number of convolutional filters, `stride` is the stride of the first convolutional layer, and `tag` is a tag that is prepended to all layer names.

```
function layers = convolutionalUnit(numF,stride,tag)
layers = [
    convolution2dLayer(3,numF,'Padding','same','Stride',stride,'Name',[tag,'conv1'])
    batchNormalizationLayer('Name',[tag,'BN1'])
    reluLayer('Name',[tag,'relu1'])
    convolution2dLayer(3,numF,'Padding','same','Name',[tag,'conv2'])
    batchNormalizationLayer('Name',[tag,'BN2'])];
end
```

References

[1] Krizhevsky, Alex. "Learning multiple layers of features from tiny images." (2009). https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

See Also

`resnetLayers` | `resnet3dLayers` | `trainNetwork` | `trainingOptions` | `layerGraph` | `analyzeNetwork`

Related Examples

- “Deep Learning Using Bayesian Optimization” on page 5-99
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Classify Image Using GoogLeNet

This example shows how to classify an image using the pretrained deep convolutional neural network GoogLeNet.

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Load Pretrained Network

Load the pretrained GoogLeNet network. This step requires the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package. If you do not have the required support packages installed, then the software provides a download link.

You can also choose to load a different pretrained network for image classification. To try a different pretrained network, open this example in MATLAB® and select a different network. For example, you can try `squeezenet`, a network that is even faster than `googlenet`. You can run this example with other pretrained networks. For a list of all available networks, see “Load Pretrained Networks” on page 1-10.

```
net = googlenet;
```

The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the first element of the `Layers` property of the network is the image input layer. The network input size is the `InputSize` property of the image input layer.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize = 1×3

   224   224     3
```

The final element of the `Layers` property is the classification output layer. The `ClassNames` property of this layer contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).ClassNames;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
    'papillon'
    'eggnog'
    'jackfruit'
    'castle'
    'sleeping bag'
    'redshank'
    'Band Aid'
    'wok'
    'seat belt'
    'orange'
```

Read and Resize Image

Read and show the image that you want to classify.

```
I = imread('peppers.png');
figure
imshow(I)
```

Display the size of the image. The image is 384-by-512 pixels and has three color channels (RGB).

```
size(I)
```

```
ans = 1×3

   384   512     3
```

Resize the image to the input size of the network by using `imresize`. This resizing slightly changes the aspect ratio of the image.

```
I = imresize(I,inputSize(1:2));
figure
imshow(I)
```

Depending on your application, you might want to resize the image in a different way. For example, you can crop the top left corner of the image by using `I(1:inputSize(1),1:inputSize(2),:)`. If you have Image Processing Toolbox™, then you can use the `imcrop` function.

Classify Image

Classify the image and calculate the class probabilities using `classify`. The network correctly classifies the image as a bell pepper. A network for classification is trained to output a single label for each input image, even when the image contains multiple objects.

```
[label,scores] = classify(net,I);
label
```

```
label = categorical
     bell pepper
```

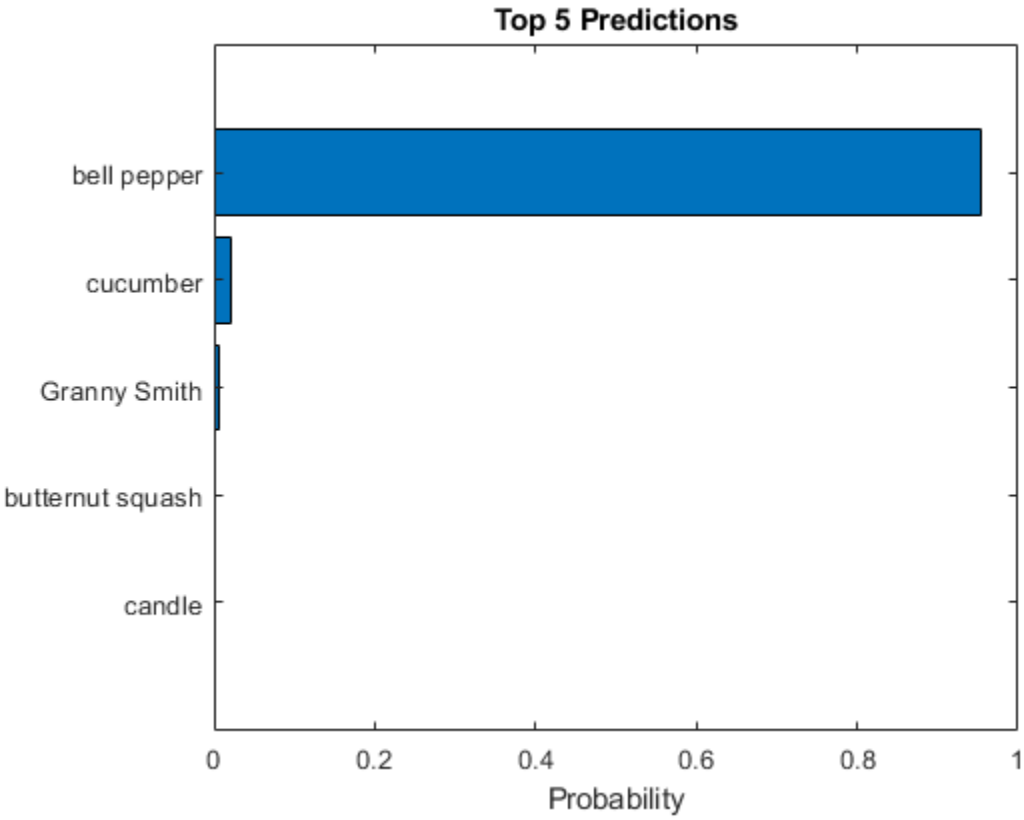
Display the image with the predicted label and the predicted probability of the image having that label.

```
figure
imshow(I)
title(string(label) + ", " + num2str(100*scores(classNames == label),3) + "%");
```

Display Top Predictions

Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
[~,idx] = sort(scores,'descend');
idx = idx(5:-1:1);
classNamesTop = net.Layers(end).ClassNames(idx);
scoresTop = scores(idx);

figure
barh(scoresTop)
xlim([0 1])
title('Top 5 Predictions')
xlabel('Probability')
yticklabels(classNamesTop)
```



References

[1] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

[2] *BVLC GoogLeNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

See Also

`DAGNetwork` | `googlenet` | `classify` | `predict` | `squeezenet`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Train Deep Learning Network to Classify New Images” on page 3-6

Extract Image Features Using Pretrained Network

This example shows how to extract learned image features from a pretrained convolutional neural network and use those features to train an image classifier. Feature extraction is the easiest and fastest way to use the representational power of pretrained deep networks. For example, you can train a support vector machine (SVM) using `fitcecoc` (Statistics and Machine Learning Toolbox™) on the extracted features. Because feature extraction only requires a single pass through the data, it is a good starting point if you do not have a GPU to accelerate network training with.

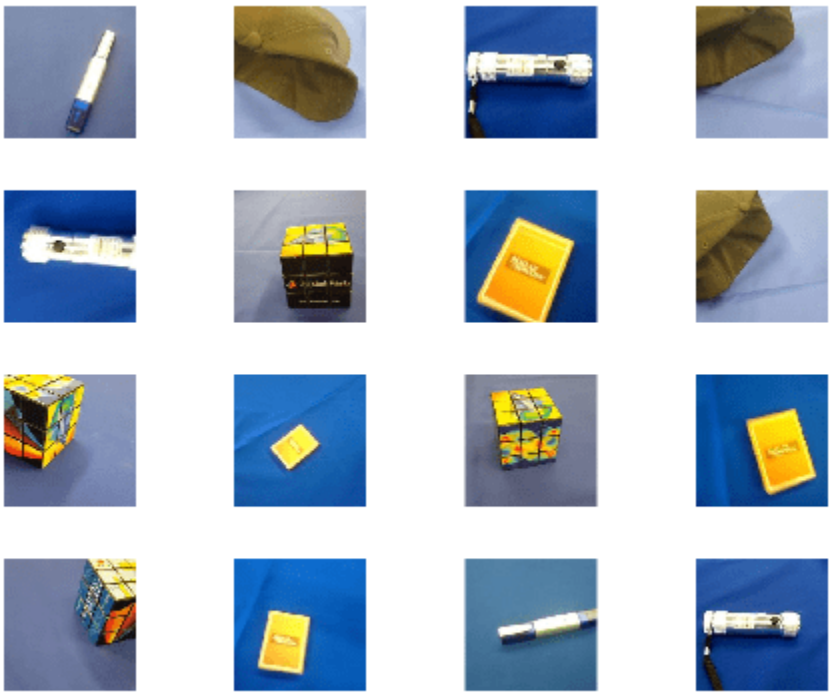
Load Data

Unzip and load the sample images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore lets you store large image data, including data that does not fit in memory. Split the data into 70% training and 30% test data.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData','IncludeSubfolders',true,'LabelSource','foldernames');
[imdsTrain,imdsTest] = splitEachLabel(imds,0.7,'randomized');
```

There are now 55 training images and 20 validation images in this very small data set. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```



Load Pretrained Network

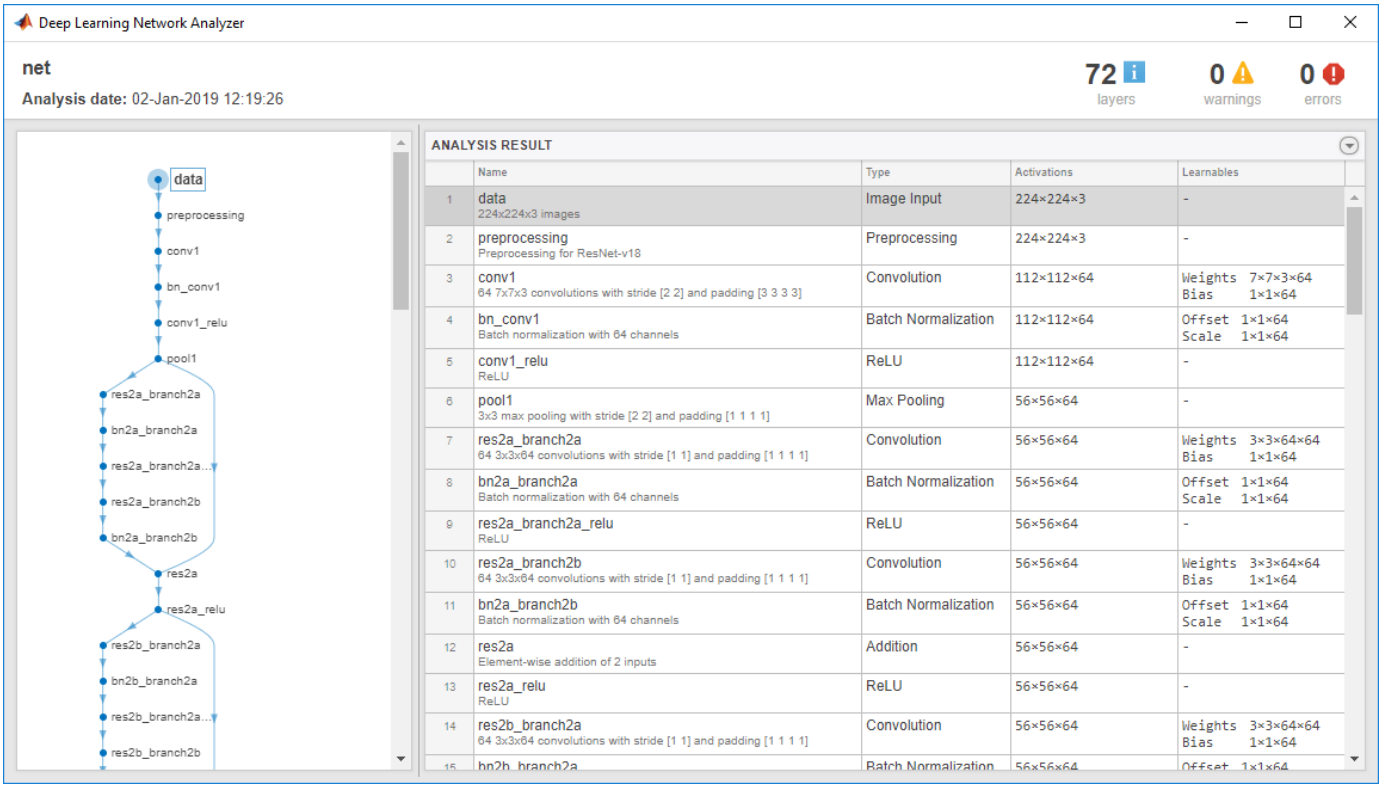
Load a pretrained ResNet-18 network. If the Deep Learning Toolbox Model *for ResNet-18 Network* support package is not installed, then the software provides a download link. ResNet-18 is trained on more than a million images and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = resnet18
```

```
net =
  DAGNetwork with properties:

         Layers: [71x1 nnet.cnn.layer.Layer]
    Connections: [78x2 table]
     InputNames: {'data'}
    OutputNames: {'ClassificationLayer_predictions'}
```

Analyze the network architecture. The first layer, the image input layer, requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize;
analyzeNetwork(net)
```



Extract Image Features

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. To automatically resize the training and test images before they are input to the network, create augmented image datastores, specify the desired image size, and use these datastores as input arguments to `activations`.

```
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain);
augimdsTest = augmentedImageDatastore(inputSize(1:2),imdsTest);
```

The network constructs a hierarchical representation of input images. Deeper layers contain higher-level features, constructed using the lower-level features of earlier layers. To get the feature representations of the training and test images, use `activations` on the global pooling layer, `'pool5'`, at the end of the network. The global pooling layer pools the input features over all spatial locations, giving 512 features in total.

```
layer = 'pool5';
featuresTrain = activations(net,augimdsTrain,layer,'OutputAs','rows');
featuresTest = activations(net,augimdsTest,layer,'OutputAs','rows');

whos featuresTrain
```

```
  Name                Size              Bytes  Class     Attributes

  featuresTrain      55x512            112640  single
```

Extract the class labels from the training and test data.

```
YTrain = imdsTrain.Labels;
YTest = imdsTest.Labels;
```

Fit Image Classifier

Use the features extracted from the training images as predictor variables and fit a multiclass support vector machine (SVM) using `fitcecoc` (Statistics and Machine Learning Toolbox).

```
classifier = fitcecoc(featuresTrain,YTrain);
```

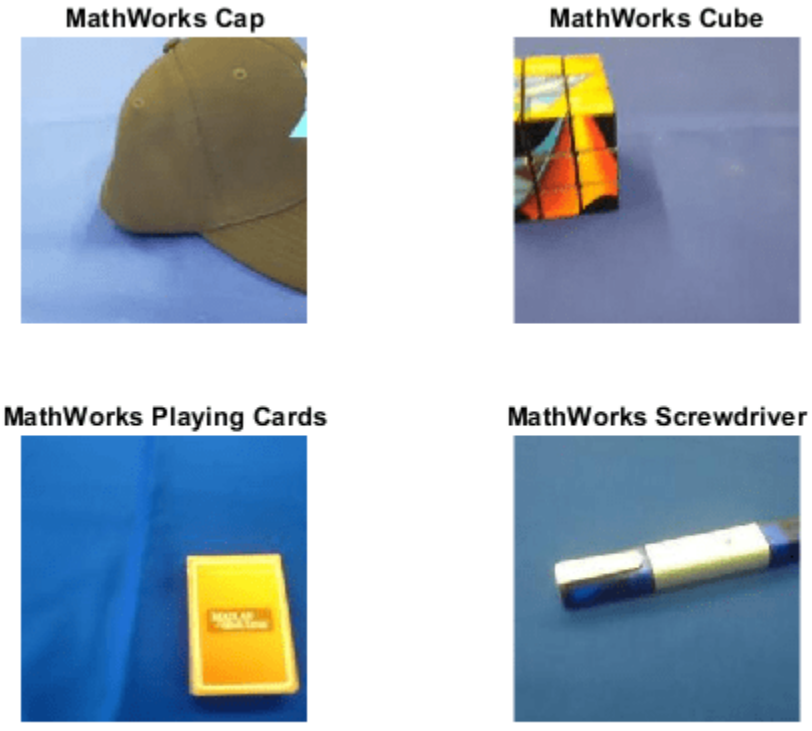
Classify Test Images

Classify the test images using the trained SVM model using the features extracted from the test images.

```
YPred = predict(classifier,featuresTest);
```

Display four sample test images with their predicted labels.

```
idx = [1 5 10 15];
figure
for i = 1:numel(idx)
    subplot(2,2,i)
    I = readimage(imdsTest,idx(i));
    label = YPred(idx(i));
    imshow(I)
    title(char(label))
end
```



Calculate the classification accuracy on the test set. Accuracy is the fraction of labels that the network predicts correctly.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 1
```

Train Classifier on Shallower Features

You can also extract features from an earlier layer in the network and train a classifier on those features. Earlier layers typically extract fewer, shallower features, have higher spatial resolution, and a larger total number of activations. Extract the features from the `'res3b_relu'` layer. This is the final layer that outputs 128 features and the activations have a spatial size of 28-by-28.

```
layer = 'res3b_relu';
featuresTrain = activations(net,augimdsTrain,layer);
featuresTest = activations(net,augimdsTest,layer);
whos featuresTrain
```

```
  Name                Size                       Bytes  Class     Attributes

  featuresTrain      28x28x128x55            22077440  single
```

The extracted features used in the first part of this example were pooled over all spatial locations by the global pooling layer. To achieve the same result when extracting features in earlier layers, manually average the activations over all spatial locations. To get the features on the form *N*-by-*C*, where *N* is the number of observations and *C* is the number of features, remove the singleton dimensions and transpose.

```
featuresTrain = squeeze(mean(featuresTrain,[1 2]))';
featuresTest = squeeze(mean(featuresTest,[1 2]))';
whos featuresTrain
```

```
  Name                Size              Bytes  Class     Attributes

  featuresTrain      55x128             28160  single
```

Train an SVM classifier on the shallower features. Calculate the test accuracy.

```
classifier = fitcecoc(featuresTrain,YTrain);
YPred = predict(classifier,featuresTest);
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.9500
```

Both trained SVMs have high accuracies. If the accuracy is not high enough using feature extraction, then try transfer learning instead. For an example, see “Train Deep Learning Network to Classify New Images” on page 3-6. For a list and comparison of the pretrained networks, see “Pretrained Deep Neural Networks” on page 1-8.

See Also

`fitcecoc` | `resnet50`

Related Examples

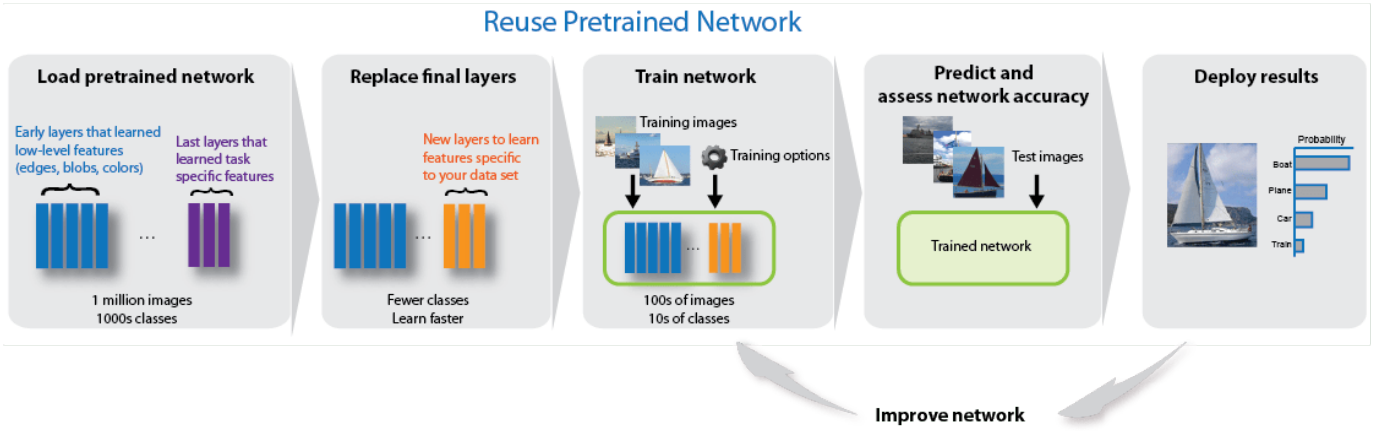
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Transfer Learning Using Pretrained Network

This example shows how to fine-tune a pretrained GoogLeNet convolutional neural network to perform classification on a new collection of images.

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.



Load Data

Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

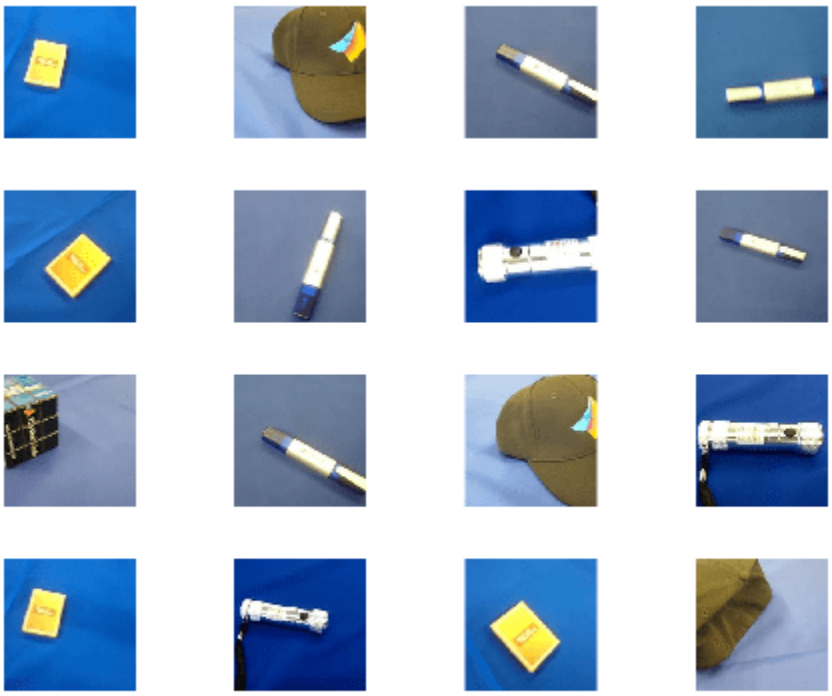
Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the image datastore into two new datastores.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This very small data set now contains 55 training images and 20 validation images. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
```

```
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```



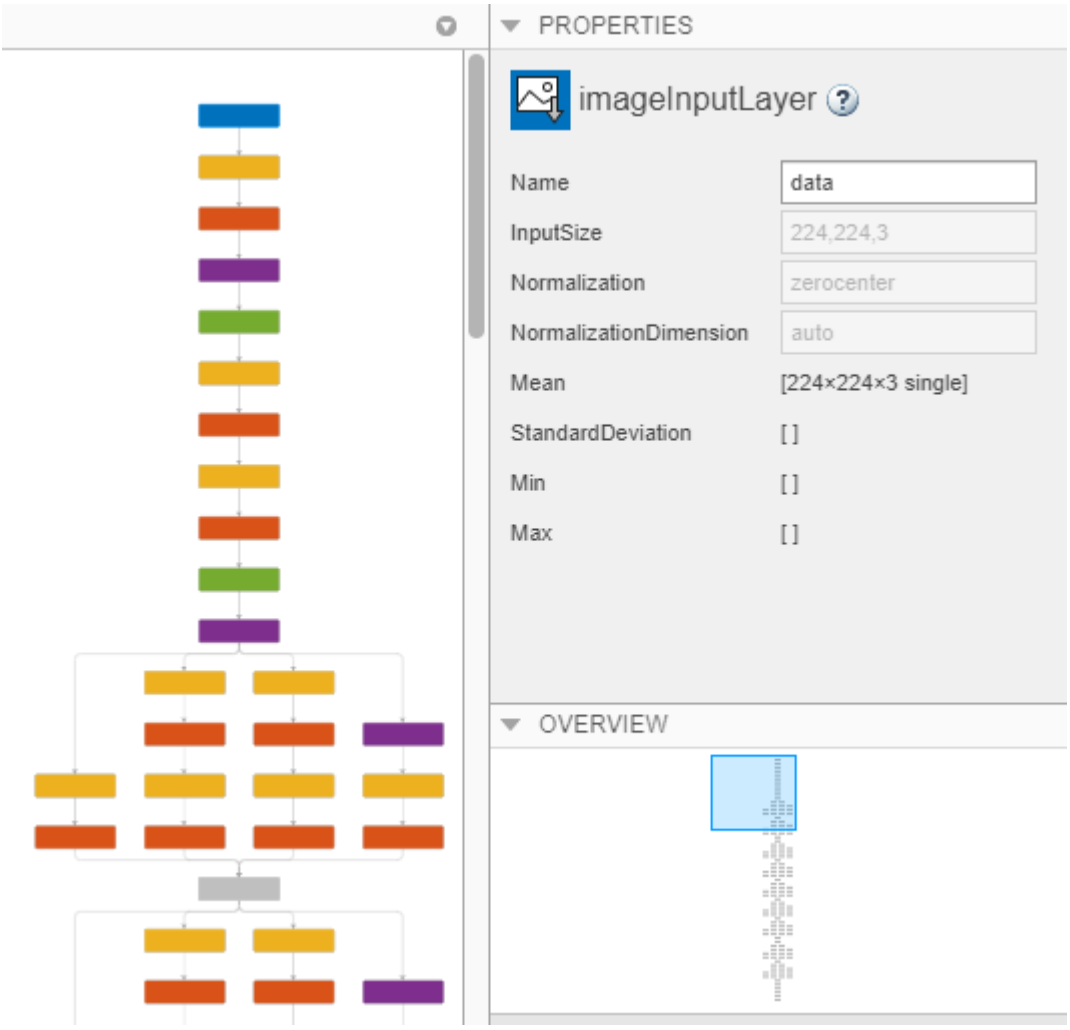
Load Pretrained Network

Load the pretrained GoogLeNet neural network. If Deep Learning Toolbox™ Model *for GoogLeNet Network* is not installed, then the software provides a download link.

```
net = googlenet;
```

Use `deepNetworkDesigner` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
deepNetworkDesigner(net)
```



The first layer, which is the image input layer, requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize = 1×3

   224   224     3
```

Replace Final Layers

The fully connected layer and classification layer of the pretrained network `net` are configured for 1000 classes. These two layers, `loss3-classifier` and `output` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net);
```

Replace the fully connected layer with a new fully connected layer that has number of outputs equal to the number of classes. To make learning faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5
```

```
newLearnableLayer = fullyConnectedLayer(numClasses, ...
    'Name','new_fc', ...
    'WeightLearnRateFactor',10, ...
    'BiasLearnRateFactor',10);

lgraph = replaceLayer(lgraph,'loss3-classifier',newLearnableLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'output',newClassLayer);
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

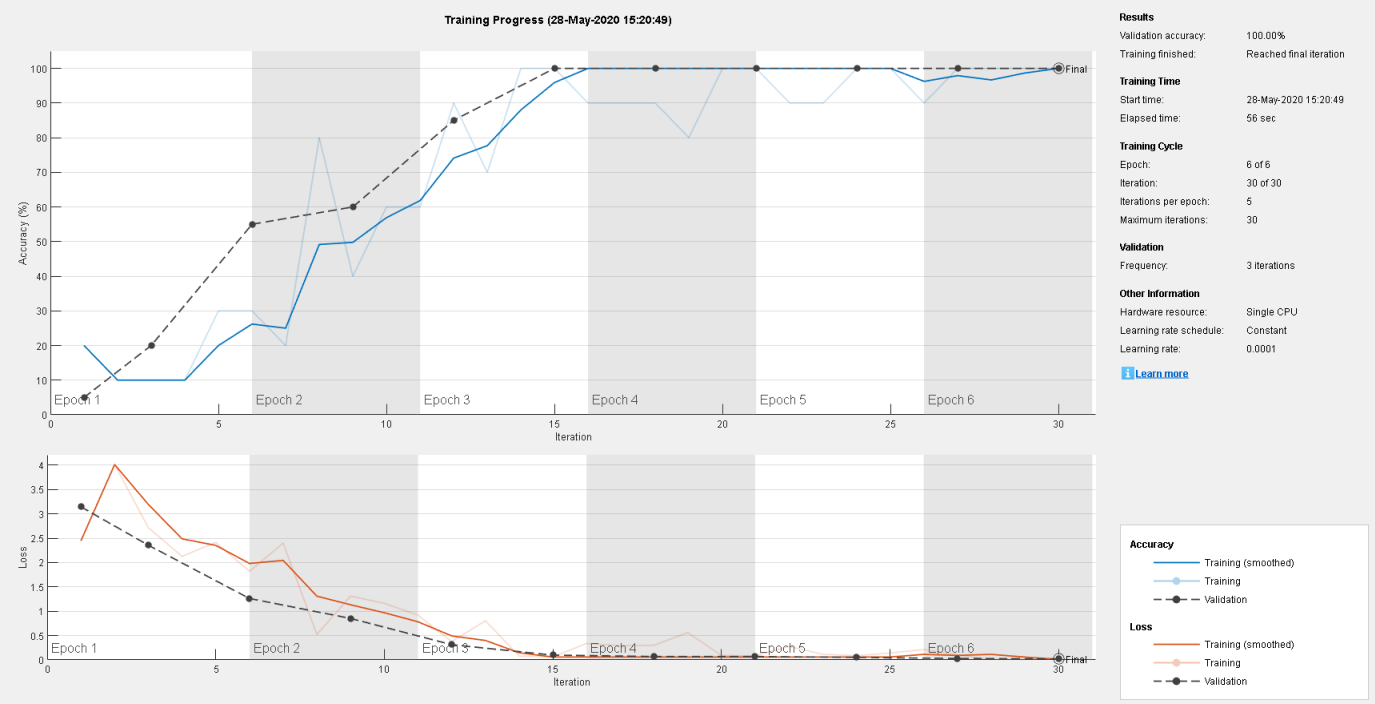
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
```

```
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network consisting of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available. This requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, it uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



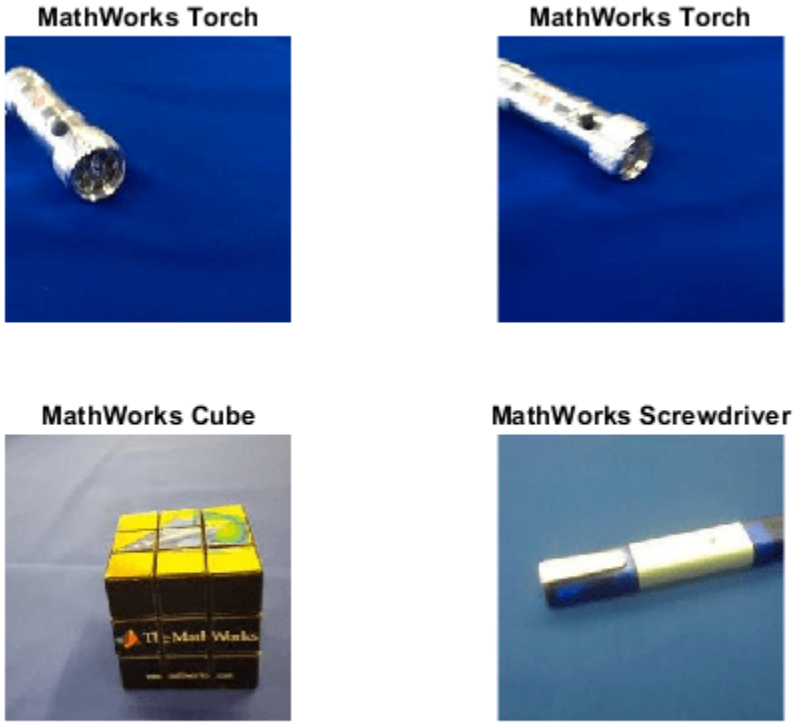
Classify Validation Images

Classify the validation images using the fine-tuned network.

```
[YPred,scores] = classify(netTransfer,augimdsValidation);
```

Display four sample validation images with their predicted labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label));
end
```



Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

```
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 1
```

For tips on improving classification accuracy, see “Deep Learning Tips and Tricks” on page 1-67.

References

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in neural information processing systems* 25 (2012).

[2] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015): 1–9.

[3] "BVLC GoogLeNet Model." https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet.

See Also

`trainNetwork` | `trainingOptions` | `squeezenet` | `googlenet` | `analyzeNetwork` | **Deep Network Designer**

Related Examples

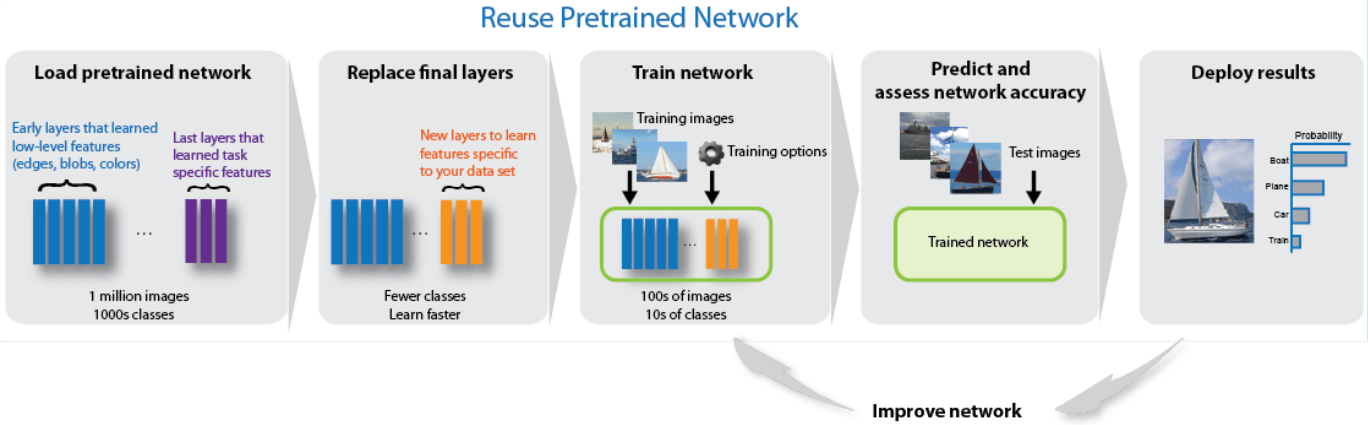
- “Learn About Convolutional Neural Networks” on page 1-17
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Create and Explore Datastore for Image Classification” on page 19-10
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Transfer Learning Using AlexNet

This example shows how to fine-tune a pretrained AlexNet convolutional neural network to perform classification on a new collection of images.

AlexNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.



Load Data

Unzip and load the new images as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

This very small data set now contains 55 training images and 20 validation images. Display some sample images.

```
numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,16);
figure
for i = 1:16
```

```
    subplot(4,4,i)
    I = readimage(imdsTrain,idx(i));
    imshow(I)
end
```

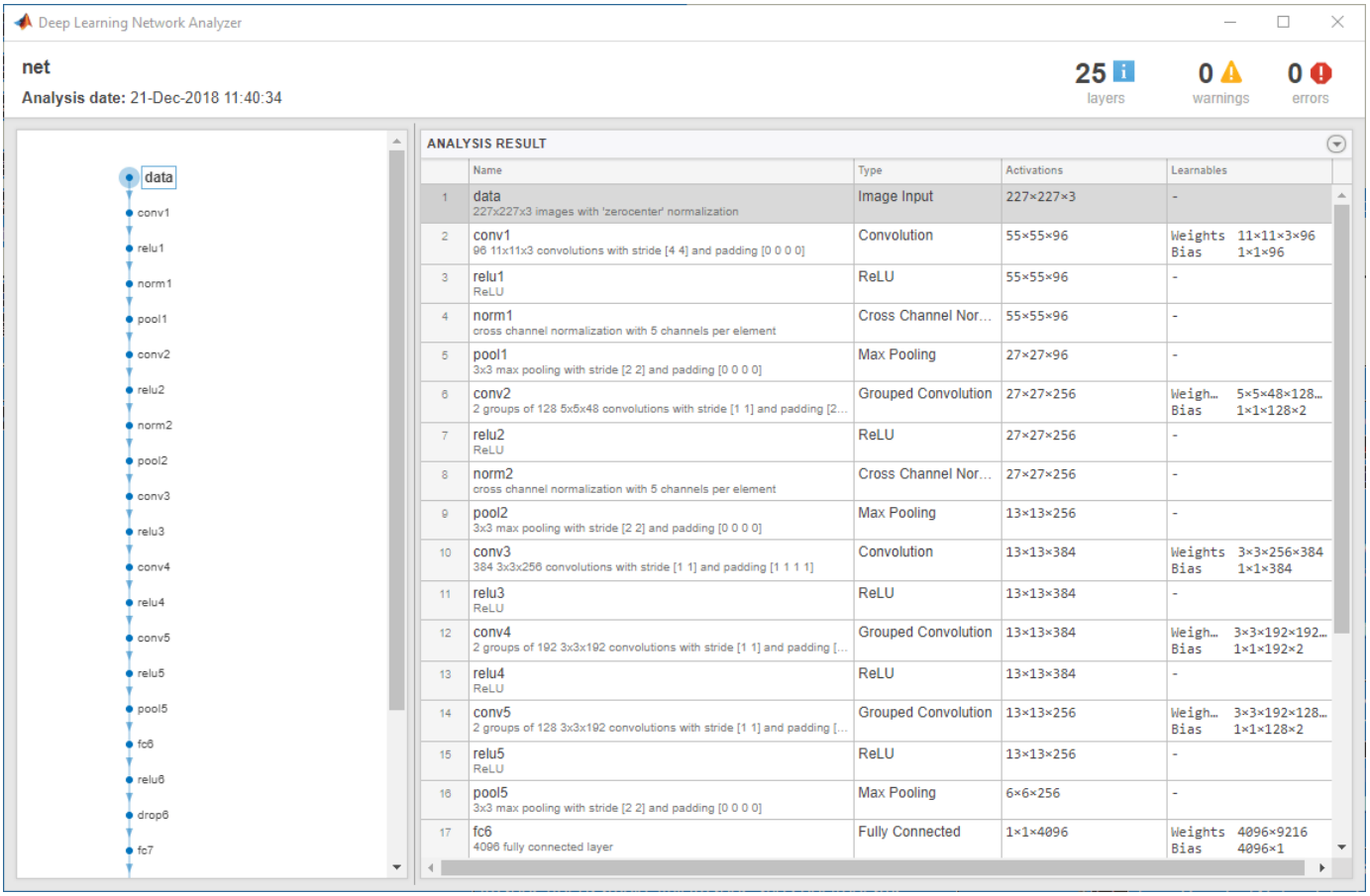
Load Pretrained Network

Load the pretrained AlexNet neural network. If Deep Learning Toolbox™ Model *for AlexNet Network* is not installed, then the software provides a download link. AlexNet is trained on more than one million images and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the model has learned rich feature representations for a wide range of images.

```
net = alexnet;
```

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
analyzeNetwork(net)
```



The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize
```

```
inputSize = 1×3

   227   227     3
```

Replace Final Layers

The last three layers of the pretrained network `net` are configured for 1000 classes. These three layers must be fine-tuned for the new classification problem. Extract all layers, except the last three, from the pretrained network.

```
layersTransfer = net.Layers(1:end-3);
```

Transfer the layers to the new classification task by replacing the last three layers with a fully connected layer, a softmax layer, and a classification output layer. Specify the options of the new fully connected layer according to the new data. Set the fully connected layer to have the same size as the number of classes in the new data. To learn faster in the new layers than in the transferred layers, increase the `WeightLearnRateFactor` and `BiasLearnRateFactor` values of the fully connected layer.

```
numClasses = numel(categories(imdsTrain.Labels))
```

```
numClasses = 5

layers = [
    layersTransfer
    fullyConnectedLayer(numClasses,'WeightLearnRateFactor',20,'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

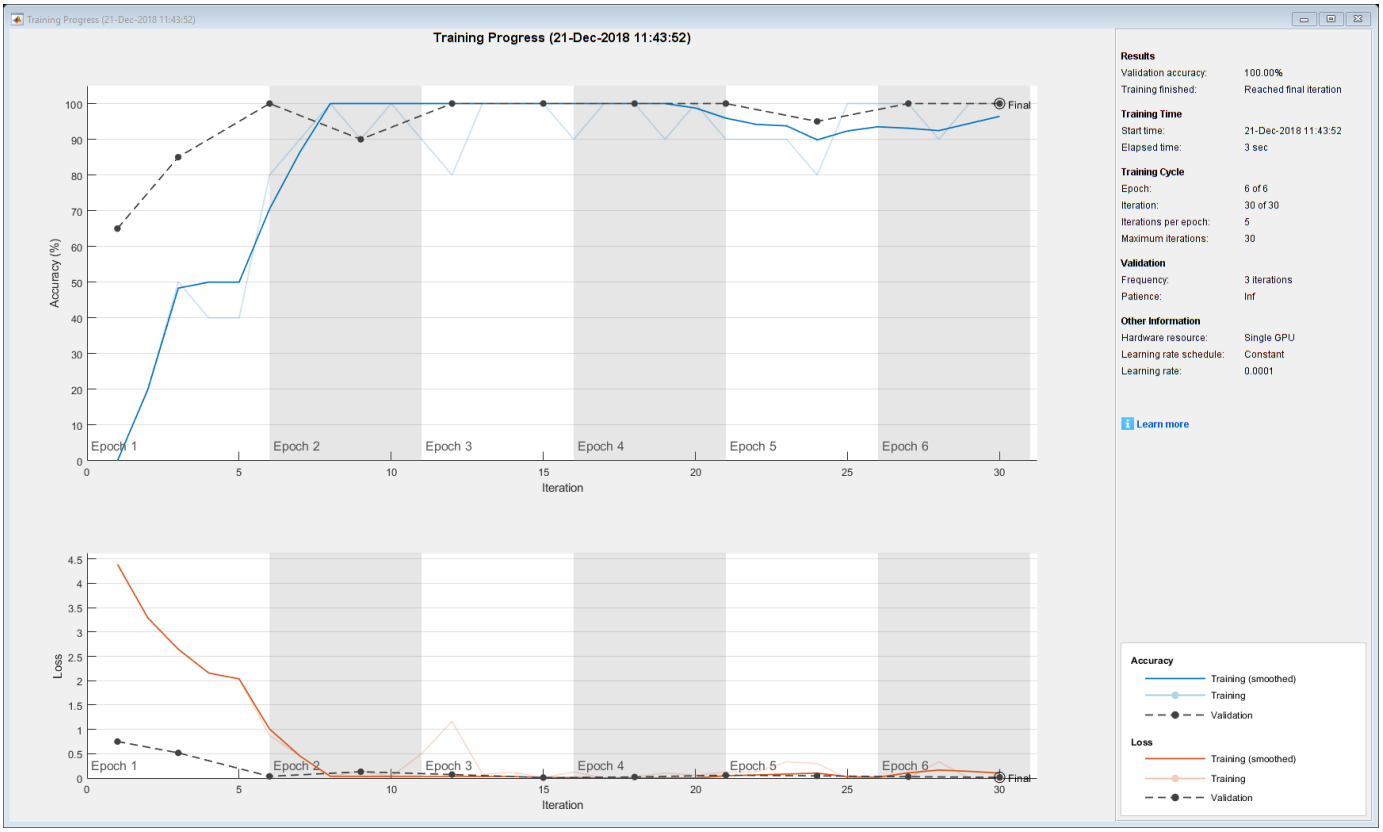
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the fully connected layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',10, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',3, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network that consists of the transferred and new layers. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
netTransfer = trainNetwork(augimdsTrain,layers,options);
```



Classify Validation Images

Classify the validation images using the fine-tuned network.

```
[YPred,scores] = classify(netTransfer,augimdsValidation);
```

Display four sample validation images with their predicted labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label));
end
```

Calculate the classification accuracy on the validation set. Accuracy is the fraction of labels that the network predicts correctly.

```
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 1
```

For tips on improving classification accuracy, see “Deep Learning Tips and Tricks” on page 1-67.

References

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in neural information processing systems*. 2012.

[2] *BVLC AlexNet Model*. https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet

See Also

`trainNetwork` | `trainingOptions` | `alexnet` | `analyzeNetwork`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-17
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42

- “Extract Image Features Using Pretrained Network” on page 3-28
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Create Simple Deep Learning Network for Classification

This example shows how to create and train a simple convolutional neural network for deep learning classification. Convolutional neural networks are essential tools for deep learning, and are especially suited for image recognition.

The example demonstrates how to:

- Load and explore image data.
- Define the network architecture.
- Specify training options.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

For an example showing how to interactively create and train a simple image classification network, see “Create Simple Image Classification Network Using Deep Network Designer”.

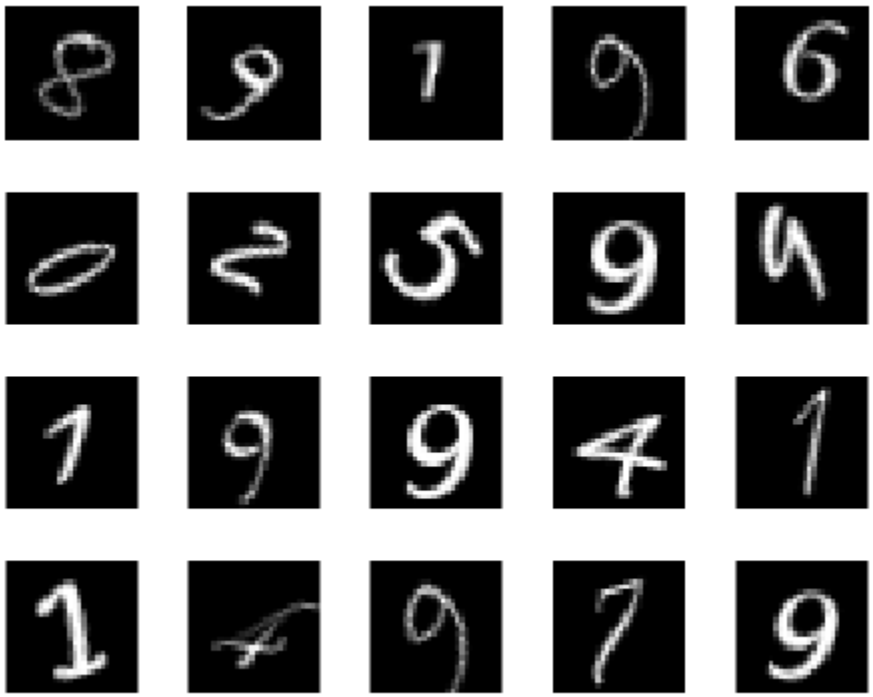
Load and Explore Image Data

Load the digit sample data as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

Display some of the images in the datastore.

```
figure;
perm = randperm(10000,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```



Calculate the number of images in each category. `labelCount` is a table that contains the labels and the number of images having each label. The datastore contains 1000 images for each of the digits 0-9, for a total of 10000 images. You can specify the number of classes in the last fully connected layer of your network as the `OutputSize` argument.

```
labelCount = countEachLabel(imds)
```

```
labelCount=10×2 table
    Label    Count
    _____    _____

      0      1000
      1      1000
      2      1000
      3      1000
      4      1000
      5      1000
      6      1000
      7      1000
      8      1000
      9      1000
```

You must specify the size of the images in the input layer of the network. Check the size of the first image in `digitData`. Each image is 28-by-28-by-1 pixels.

```
img = readimage(imds,1);
size(img)
```

```
ans = 1×2

    28    28
```

Specify Training and Validation Sets

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label. `splitEachLabel` splits the datastore `digitData` into two new datastores, `trainDigitData` and `valDigitData`.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');
```

Define Network Architecture

Define the convolutional neural network architecture.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Image Input Layer An `imageInputLayer` is where you specify the image size, which, in this case, is 28-by-28-by-1. These numbers correspond to the height, width, and the channel size. The digit data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values. You do not need to shuffle the data because `trainNetwork`, by default, shuffles the data at the beginning of training. `trainNetwork` can also automatically shuffle the data at the beginning of every epoch during training.

Convolutional Layer In the convolutional layer, the first argument is `filterSize`, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, `numFilters`, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the `'Padding'` name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, `'same'` padding ensures that the spatial

output size is the same as the input size. You can also define the stride and learning rates for this layer using name-value pair arguments of `convolution2dLayer`.

Batch Normalization Layer Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Use `batchNormalizationLayer` to create a batch normalization layer.

ReLU Layer The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use `reluLayer` to create a ReLU layer.

Max Pooling Layer Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using `maxPooling2dLayer`. The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, `poolSize`. In this example, the size of the rectangular region is [2,2]. The `'Stride'` name-value pair argument specifies the step size that the training function takes as it scans along the input.

Fully Connected Layer The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore, the `OutputSize` parameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes. Use `fullyConnectedLayer` to create a fully connected layer.

Softmax Layer The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the `softmaxLayer` function after the last fully connected layer.

Classification Layer The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer, use `classificationLayer`.

Specify Training Options

After defining the network structure, specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.01. Set the maximum number of epochs to 4. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. Turn on the training progress plot, and turn off the command window output.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
```

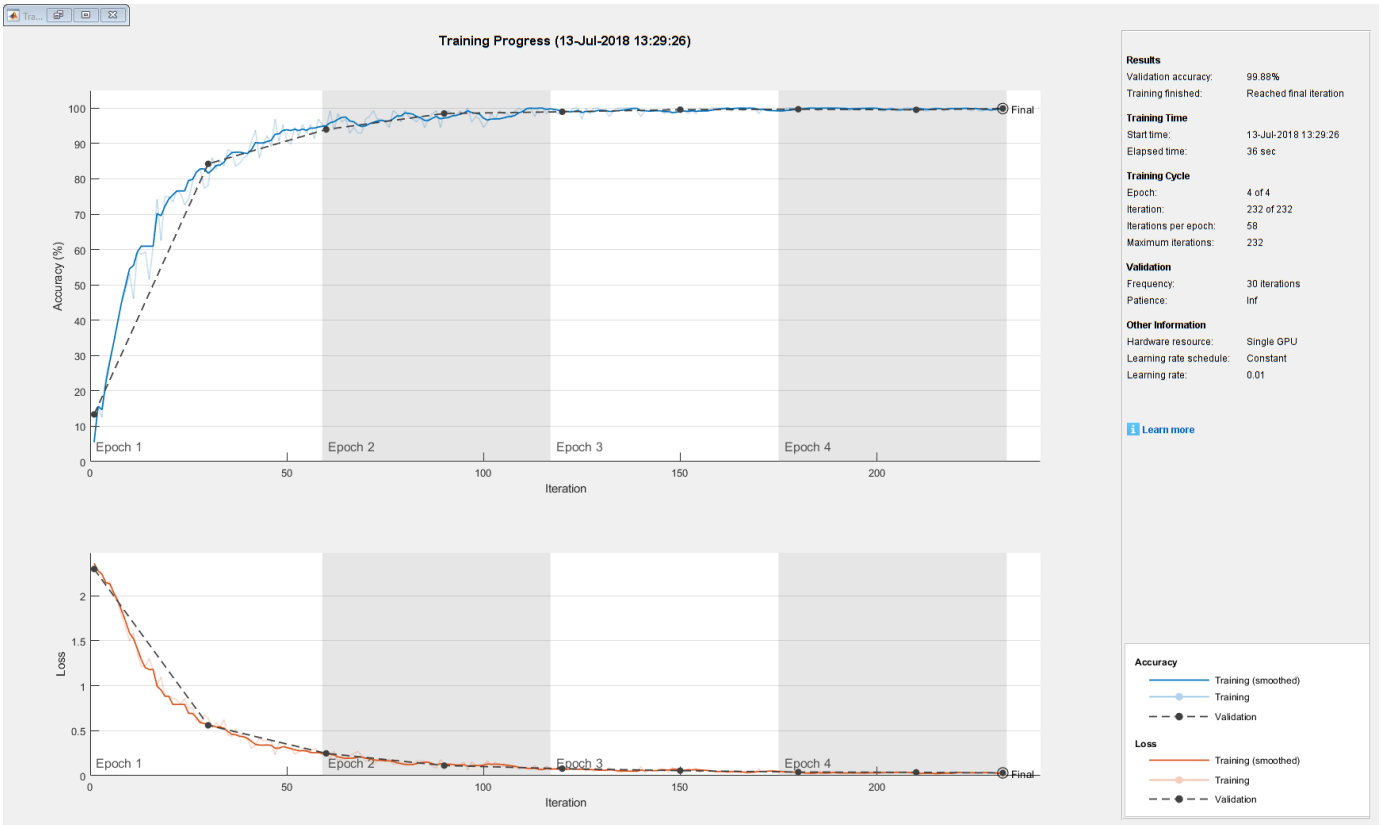
```
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train Network Using Training Data

Train the network using the architecture defined by `layers`, the training data, and the training options. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. For more information on the training progress plot, see "Monitor Deep Learning Training Progress" on page 5-115. The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

```
net = trainNetwork(imdsTrain,layers,options);
```



Classify Validation Images and Compute Accuracy

Predict the labels of the validation data using the trained network, and calculate the final validation accuracy. Accuracy is the fraction of labels that the network predicts correctly. In this case, more than 99% of the predicted labels match the true labels of the validation set.

```
YPred = classify(net,imdsValidation);
YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation)/numel(YValidation)
```

```
accuracy = 0.9988
```

See Also

`trainNetwork` | `trainingOptions` | `analyzeNetwork` | **Deep Network Designer**

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-17
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2
- Deep Learning Onramp

Train Convolutional Neural Network for Regression

This example shows how to fit a regression model using convolutional neural networks to predict the angles of rotation of handwritten digits.

Convolutional neural networks (CNNs, or ConvNets) are essential tools for deep learning, and are especially suited for analyzing image data. For example, you can use CNNs to classify images. To predict continuous data, such as angles and distances, you can include a regression layer at the end of the network.

The example constructs a convolutional neural network architecture, trains a network, and uses the trained network to predict angles of rotated handwritten digits. These predictions are useful for optical character recognition.

Optionally, you can use `imrotate` (Image Processing Toolbox™) to rotate the images, and `boxplot` (Statistics and Machine Learning Toolbox™) to create a residual box plot.

Load Data

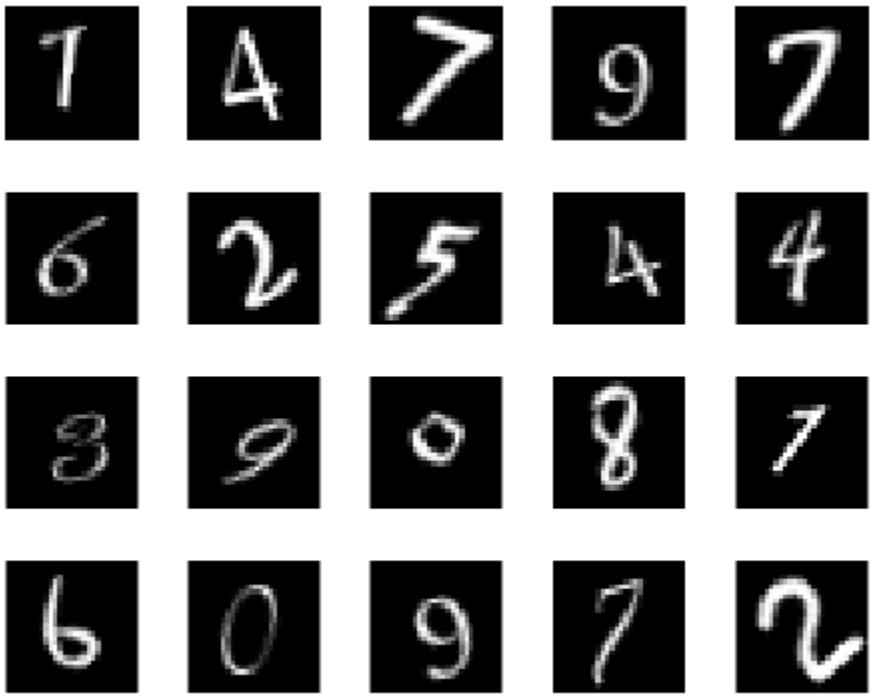
The data set contains synthetic images of handwritten digits together with the corresponding angles (in degrees) by which each image is rotated.

Load the training and validation images as 4-D arrays using `digitTrain4DArrayData` and `digitTest4DArrayData`. The outputs `YTrain` and `YValidation` are the rotation angles in degrees. The training and validation data sets each contain 5000 images.

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
[XValidation,~,YValidation] = digitTest4DArrayData;
```

Display 20 random training images using `imshow`.

```
numTrainImages = numel(YTrain);
figure
idx = randperm(numTrainImages,20);
for i = 1:numel(idx)
    subplot(4,5,i)
    imshow(XTrain(:,:,:,idx(i)))
end
```



Check Data Normalization

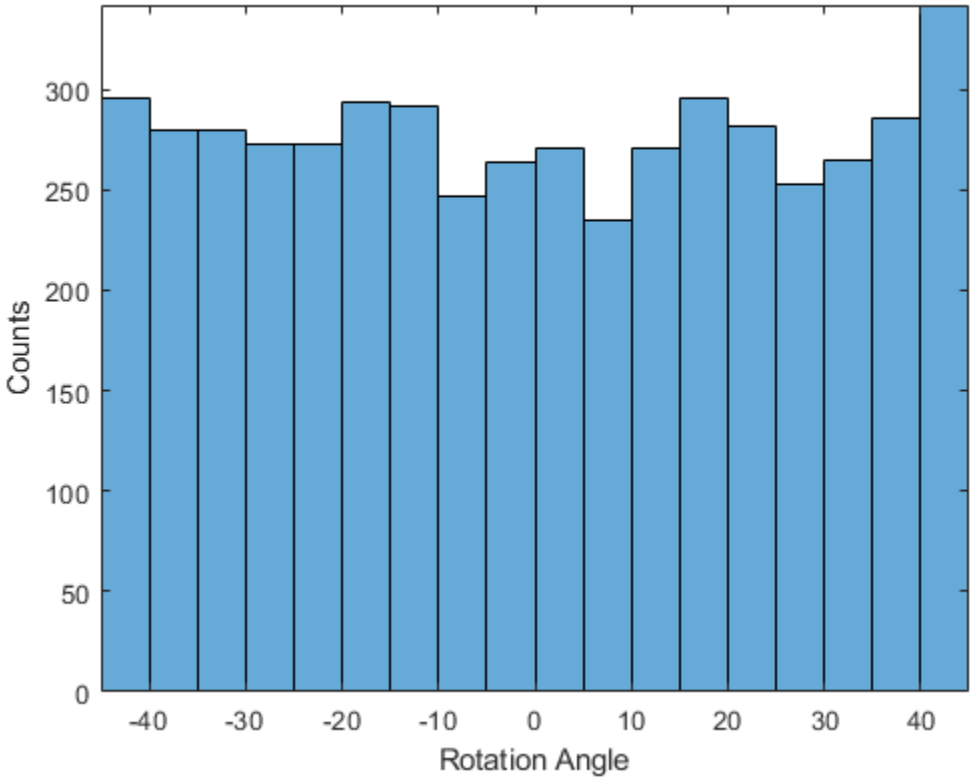
When training neural networks, it often helps to make sure that your data is normalized in all stages of the network. Normalization helps stabilize and speed up network training using gradient descent. If your data is poorly scaled, then the loss can become `NaN` and the network parameters can diverge during training. Common ways of normalizing data include rescaling the data so that its range becomes [0,1] or so that it has a mean of zero and standard deviation of one. You can normalize the following data:

- Input data. Normalize the predictors before you input them to the network. In this example, the input images are already normalized to the range [0,1].
- Layer outputs. You can normalize the outputs of each convolutional and fully connected layer by using a batch normalization layer.
- Responses. If you use batch normalization layers to normalize the layer outputs in the end of the network, then the predictions of the network are normalized when training starts. If the response has a very different scale from these predictions, then network training can fail to converge. If your response is poorly scaled, then try normalizing it and see if network training improves. If you normalize the response before training, then you must transform the predictions of the trained network to obtain the predictions of the original response.

Plot the distribution of the response. The response (the rotation angle in degrees) is approximately uniformly distributed between -45 and 45, which works well without needing normalization. In classification problems, the outputs are class probabilities, which are always normalized.

```
figure
histogram(YTrain)
```

```
axis tight
ylabel('Counts')
xlabel('Rotation Angle')
```



In general, the data does not have to be exactly normalized. However, if you train the network in this example to predict `100*YTrain` or `YTrain+500` instead of `YTrain`, then the loss becomes `NaN` and the network parameters diverge when training starts. These results occur even though the only difference between a network predicting $aY + b$ and a network predicting Y is a simple rescaling of the weights and biases of the final fully connected layer.

If the distribution of the input or response is very uneven or skewed, you can also perform nonlinear transformations (for example, taking logarithms) to the data before training the network.

Create Network Layers

To solve the regression problem, create the layers of the network and include a regression layer at the end of the network.

The first layer defines the size and type of the input data. The input images are 28-by-28-by-1. Create an image input layer of the same size as the training images.

The middle layers of the network define the core architecture of the network, where most of the computation and learning take place.

The final layers define the size and type of output data. For regression problems, a fully connected layer must precede the regression layer at the end of the network. Create a fully connected output layer of size 1 and a regression layer.

Combine all the layers together in a `Layer` array.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    dropoutLayer(0.2)
    fullyConnectedLayer(1)
    regressionLayer];
```

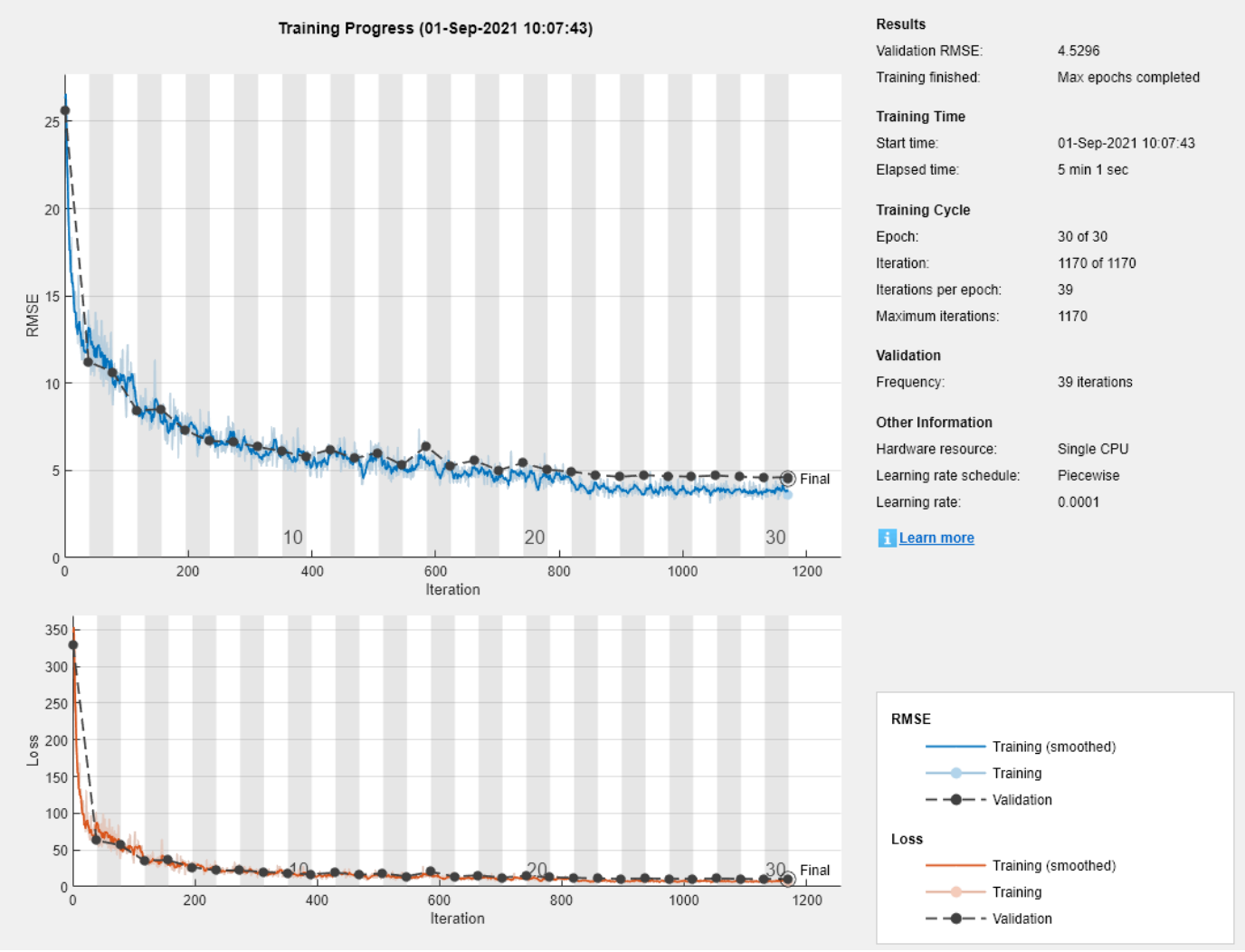
Train Network

Create the network training options. Train for 30 epochs. Set the initial learn rate to 0.001 and lower the learning rate after 20 epochs. Monitor the network accuracy during training by specifying validation data and validation frequency. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. Turn on the training progress plot, and turn off the command window output.

```
miniBatchSize  = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',30, ...
    'InitialLearnRate',1e-3, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',20, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',validationFrequency, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Create the network using `trainNetwork`. This command uses a compatible GPU if available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses the CPU.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Examine the details of the network architecture contained in the `Layers` property of `net`.

```
net.Layers
```

```
ans =
  18x1 Layer array with layers:

     1   'imageinput'        Image Input           28x28x1 images with 'zerocenter' normalizatio
     2   'conv_1'            Convolution           8 3x3x1 convolutions with stride [1  1] and p
     3   'batchnorm_1'       Batch Normalization   Batch normalization with 8 channels
     4   'relu_1'            ReLU                  ReLU
     5   'avgpool2d_1'       Average Pooling       2x2 average pooling with stride [2  2] and pa
     6   'conv_2'            Convolution           16 3x3x8 convolutions with stride [1  1] and
     7   'batchnorm_2'       Batch Normalization   Batch normalization with 16 channels
     8   'relu_2'            ReLU                  ReLU
     9   'avgpool2d_2'       Average Pooling       2x2 average pooling with stride [2  2] and pa
    10   'conv_3'            Convolution           32 3x3x16 convolutions with stride [1  1] and
    11   'batchnorm_3'       Batch Normalization   Batch normalization with 32 channels
    12   'relu_3'            ReLU                  ReLU
    13   'conv_4'            Convolution           32 3x3x32 convolutions with stride [1  1] and
```

```
    14   'batchnorm_4'        Batch Normalization   Batch normalization with 32 channels
    15   'relu_4'             ReLU                  ReLU
    16   'dropout'            Dropout               20% dropout
    17   'fc'                 Fully Connected       1 fully connected layer
    18   'regressionoutput'   Regression Output     mean-squared-error with response 'Response'
```

Test Network

Test the performance of the network by evaluating the accuracy on the validation data.

Use `predict` to predict the angles of rotation of the validation images.

```
YPredicted = predict(net,XValidation);
```

Evaluate Performance

Evaluate the performance of the model by calculating:

1. The percentage of predictions within an acceptable error margin
2. The root-mean-square error (RMSE) of the predicted and actual angles of rotation

Calculate the prediction error between the predicted and actual angles of rotation.

```
predictionError = YValidation - YPredicted;
```

Calculate the number of predictions within an acceptable error margin from the true angles. Set the threshold to be 10 degrees. Calculate the percentage of predictions within this threshold.

```
thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numValidationImages = numel(YValidation);

accuracy = numCorrect/numValidationImages
```

```
accuracy = 0.9700
```

Use the root-mean-square error (RMSE) to measure the differences between the predicted and actual angles of rotation.

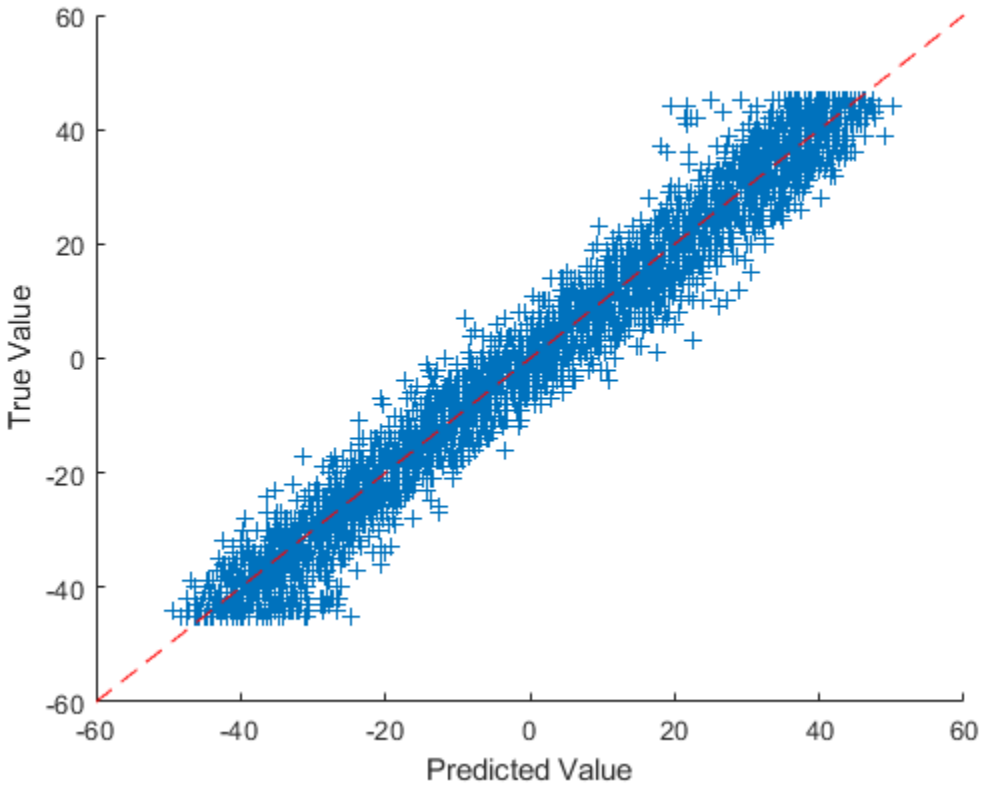
```
squares = predictionError.^2;
rmse = sqrt(mean(squares))
```

```
rmse = single
    4.5296
```

Visualize Predictions

Visualize the predictions in a scatter plot. Plot the predicted values against the true values.

```
figure
scatter(YPredicted,YValidation,'+')
xlabel("Predicted Value")
ylabel("True Value")

hold on
plot([-60 60], [-60 60],'r--')
```



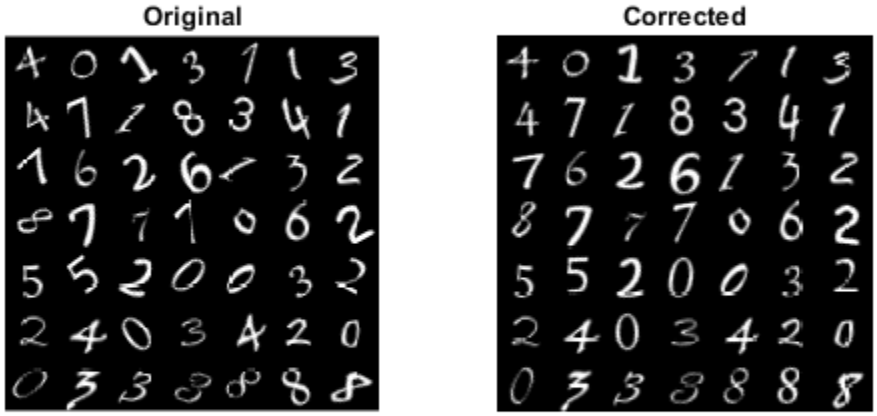
Correct Digit Rotations

You can use functions from Image Processing Toolbox to straighten the digits and display them together. Rotate 49 sample digits according to their predicted angles of rotation using `imrotate` (Image Processing Toolbox).

```
idx = randperm(numValidationImages,49);
for i = 1:numel(idx)
    image = XValidation(:,:,:,idx(i));
    predictedAngle = YPredicted(idx(i));
    imagesRotated(:,:,:,i) = imrotate(image,predictedAngle,'bicubic','crop');
end
```

Display the original digits with their corrected rotations. You can use `montage` (Image Processing Toolbox) to display the digits together in a single image.

```
figure
subplot(1,2,1)
montage(XValidation(:,:,:,idx))
title('Original')

subplot(1,2,2)
montage(imagesRotated)
title('Corrected')
```



See Also

`regressionLayer` | `classificationLayer`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Convert Classification Network into Regression Network” on page 3-70

Train Network with Multiple Outputs

This example shows how to train a deep learning network with multiple outputs that predict both labels and angles of rotations of handwritten digits.

To train a network with multiple outputs, you must train the network using a custom training loop.

Load Training Data

The `digitTrain4DArrayData` function loads the images, their digit labels, and their angles of rotation from the vertical. Create an `arrayDatastore` object for the images, labels, and the angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and number of nondiscrete responses.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsYTrain = arrayDatastore(YTrain);
dsAnglesTrain = arrayDatastore(anglesTrain);

dsTrain = combine(dsXTrain,dsYTrain,dsAnglesTrain);

classNames = categories(YTrain);
numClasses = numel(classNames);
numObservations = numel(YTrain);
```

View some images from the training data.

```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

Define Deep Learning Model

Define the following network that predicts both labels and angles of rotation.

- A convolution-batchnorm-ReLU block with 16 5-by-5 filters.
- Two convolution-batchnorm-ReLU blocks each with 32 3-by-3 filters.
- A skip connection around the previous two blocks containing a convolution-batchnorm-ReLU block with 32 1-by-1 convolutions.
- Merge the skip connection using addition.
- For classification output, a branch with a fully connected operation of size 10 (the number of classes) and a softmax operation.
- For the regression output, a branch with a fully connected operation of size 1 (the number of responses).

Define the main block of layers as a layer graph.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','in')

    convolution2dLayer(5,16,'Padding','same','Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')

    convolution2dLayer(3,32,'Padding','same','Stride',2,'Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,32,'Padding','same','Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu4')

    additionLayer(2,'Name','addition')

    fullyConnectedLayer(numClasses,'Name','fc1')
    softmaxLayer('Name','softmax')];

lgraph = layerGraph(layers);
```

Add the skip connection.

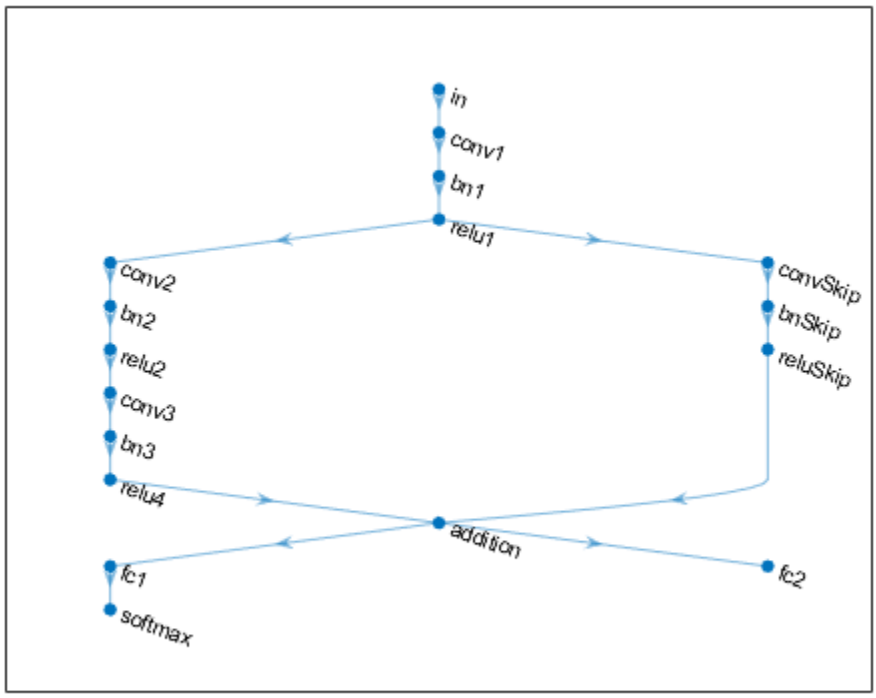
```
layers = [
    convolution2dLayer(1,32,'Stride',2,'Name','convSkip')
    batchNormalizationLayer('Name','bnSkip')
    reluLayer('Name','reluSkip')];

lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'relu1','convSkip');
lgraph = connectLayers(lgraph,'reluSkip','addition/in2');
```

Add the fully connected layer for regression.

```
layers = fullyConnectedLayer(1,'Name','fc2');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'addition','fc2');
```

View the layer graph in a plot.

```
figure
plot(lgraph)
```



Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [17×1 nnet.cnn.layer.Layer]
    Connections: [17×2 table]
     Learnables: [20×3 table]
          State: [8×3 table]
     InputNames: {'in'}
    OutputNames: {'softmax'  'fc2'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes as input, the `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

Specify Training Options

Specify the training options. Train for 30 epochs using a mini-batch size of 128.

```
numEpochs = 30;
miniBatchSize = 128;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

Train Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or angles.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessData,...
    'MiniBatchFormat',{'SSCB','',''});
```

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. At the end of each iteration, display the training progress. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

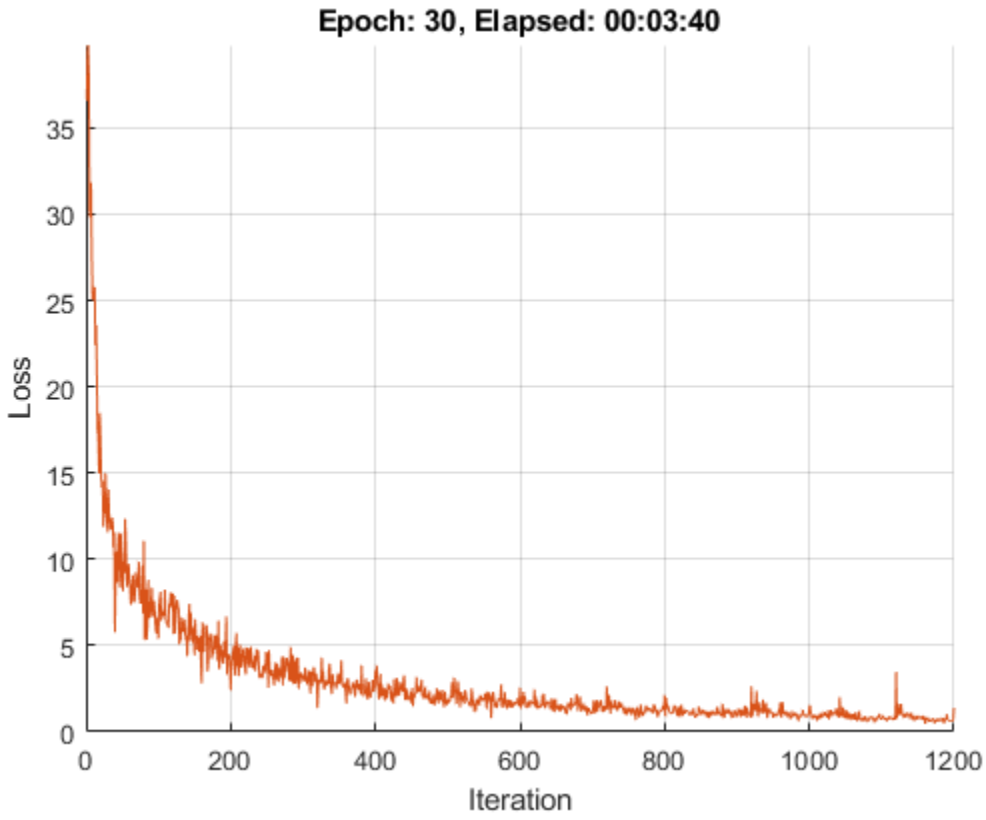
Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
```

```
        shuffle(mbq)

        % Loop over mini-batches
        while hasdata(mbq)

            iteration = iteration + 1;

            [dlX,dlY1,dlY2] = next(mbq);

            % Evaluate the model gradients, state, and loss using dlfeval and the
            % modelGradients function.
            [gradients,state,loss] = dlfeval(@modelGradients, dlnet, dlX, dlY1, dlY2);
            dlnet.State = state;

            % Update the network parameters using the Adam optimizer.
            [dlnet,trailingAvg,trailingAvgSq] = adamupdate(dlnet,gradients, ...
                trailingAvg,trailingAvgSq,iteration);

            % Display the training progress.
            if plots == "training-progress"
                D = duration(0,0,toc(start),'Format','hh:mm:ss');
                addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
                title("Epoch: " + epoch + ", Elapsed: " + string(D))
                drawnow
            end
        end
    end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels and angles. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
[XTest,Y1Test,anglesTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsYTest = arrayDatastore(Y1Test);
dsAnglesTest = arrayDatastore(anglesTest);

dsTest = combine(dsXTest,dsYTest,dsAnglesTest);

mbqTest = minibatchqueue(dsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessData,...
    'MiniBatchFormat',{'SSCB','',''});
```

To predict the labels and angles of the validation data, loop over the mini-batches and use the `predict` function. Store the predicted classes and angles. Compare the predicted and true classes and angles and store the results.

```
classesPredictions = [];
anglesPredictions = [];
classCorr = [];
angleDiff = [];

% Loop over mini-batches.
while hasdata(mbqTest)

    % Read mini-batch of data.
    [dlXTest,dlY1Test,dlY2Test] = next(mbqTest);

    % Make predictions using the predict function.
    [dlY1Pred,dlY2Pred] = predict(dlnet,dlXTest,'Outputs',["softmax" "fc2"]);

    % Determine predicted classes.
    Y1PredBatch = onehotdecode(dlY1Pred,classNames,1);
    classesPredictions = [classesPredictions Y1PredBatch];

    % Dermine predicted angles
    Y2PredBatch = extractdata(dlY2Pred);
    anglesPredictions = [anglesPredictions Y2PredBatch];

    % Compare predicted and true classes
    Y1Test = onehotdecode(dlY1Test,classNames,1);
    classCorr = [classCorr Y1PredBatch == Y1Test];

    % Compare predicted and true angles
    angleDiffBatch = Y2PredBatch - dlY2Test;
    angleDiff = [angleDiff extractdata(gather(angleDiffBatch))];

end
```

Evaluate the classification accuracy.

```
accuracy = mean(classCorr)
```

```
accuracy = 0.9814
```

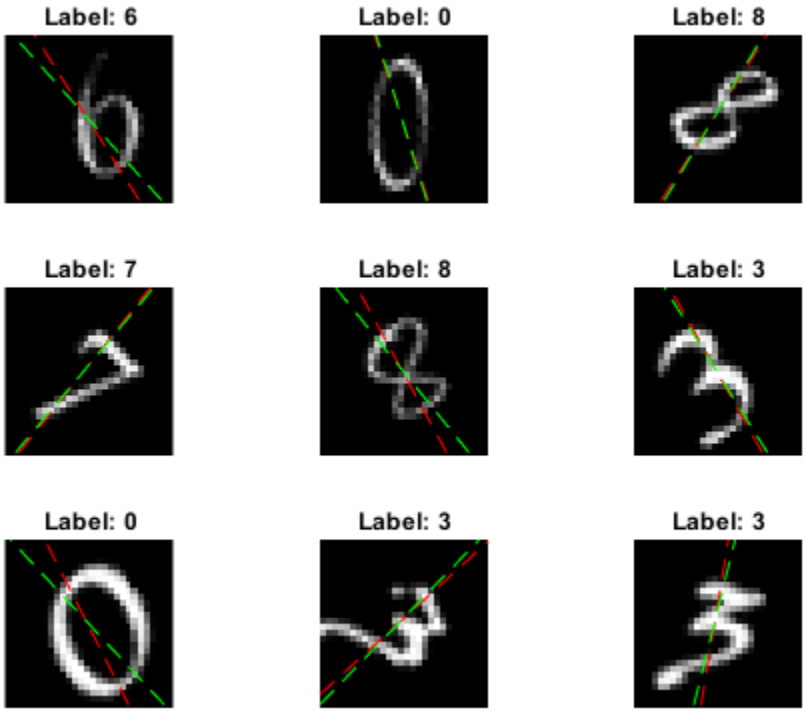
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean(angleDiff.^2))
```

```
angleRMSE = single
    7.7431
```

View some of the images with their predictions. Display the predicted angles in red and the correct labels in green.

```
idx = randperm(size(XTest,4),9);
figure
for i = 1:9
    subplot(3,3,i)
    I = XTest(:,:,:,idx(i));
    imshow(I)
    hold on

    sz = size(I,1);
    offset = sz/2;

    thetaPred = anglesPredictions(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    thetaValidation = anglesTest(idx(i));
    plot(offset*[1-tand(thetaValidation) 1+tand(thetaValidation)],[sz 0],'g--')

    hold off
    label = string(classesPredictions(idx(i)));
    title("Label: " + label)
end
```



Model Gradients Function

The `modelGradients` function, takes as input, the `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,T1,T2)

[dlY1,dlY2,state] = forward(dlnet,dlX,'Outputs',["softmax" "fc2"]);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,dlnet.Learnables);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.

2. Extract the label and angle data from the incoming cell arrays and concatenate along the second dimension into a categorical array and a numeric array, respectively.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,angle] = preprocessData(XCell,YCell,angleCell)

    % Extract image data from cell and concatenate
    X = cat(4,XCell{:});
    % Extract label data from cell and concatenate
    Y = cat(2,YCell{:});
    % Extract angle data from cell and concatenate
    angle = cat(2,angleCell{:});

    % One-hot encode labels
    Y = onehotencode(Y,1);

end
```

See Also

`dlarray` | `dlgradient` | `dlfeval` | `sgdmupdate` | `batchNormalizationLayer` | `convolution2dLayer` | `reluLayer` | `fullyConnectedLayer` | `softmaxLayer` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

- “Multiple-Input and Multiple-Output Networks” on page 1-19
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Assemble Multiple-Output Network for Prediction” on page 18-196
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Deep Learning Layers” on page 1-21

Convert Classification Network into Regression Network

This example shows how to convert a trained classification network into a regression network.

Pretrained image classification networks have been trained on over a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. The networks have learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. This example shows how to take a pretrained classification network and retrain it for regression tasks.

The example loads a pretrained convolutional neural network architecture for classification, replaces the layers for classification and retrains the network to predict angles of rotated handwritten digits. Optionally, you can use `imrotate` (Image Processing Toolbox™) to correct the image rotations using the predicted values.

Load Pretrained Network

Load the pretrained network from the supporting file `digitsNet.mat`. This file contains a classification network that classifies handwritten digits.

```
load digitsNet
layers = net.Layers
```

```
layers =
  15x1 Layer array with layers:

     1   'imageinput'    Image Input             28x28x1 images with 'zerocenter' normalization
     2   'conv_1'        Convolution             8 3x3x1 convolutions with stride [1  1] and pado
     3   'batchnorm_1'   Batch Normalization     Batch normalization with 8 channels
     4   'relu_1'        ReLU                    ReLU
     5   'maxpool_1'     Max Pooling             2x2 max pooling with stride [2  2] and padding
     6   'conv_2'        Convolution             16 3x3x8 convolutions with stride [1  1] and pad
     7   'batchnorm_2'   Batch Normalization     Batch normalization with 16 channels
     8   'relu_2'        ReLU                    ReLU
     9   'maxpool_2'     Max Pooling             2x2 max pooling with stride [2  2] and padding
    10   'conv_3'        Convolution             32 3x3x16 convolutions with stride [1  1] and pa
    11   'batchnorm_3'   Batch Normalization     Batch normalization with 32 channels
    12   'relu_3'        ReLU                    ReLU
    13   'fc'            Fully Connected         10 fully connected layer
    14   'softmax'       Softmax                 softmax
    15   'classoutput'   Classification Output   crossentropyex with '0' and 9 other classes
```

Load Data

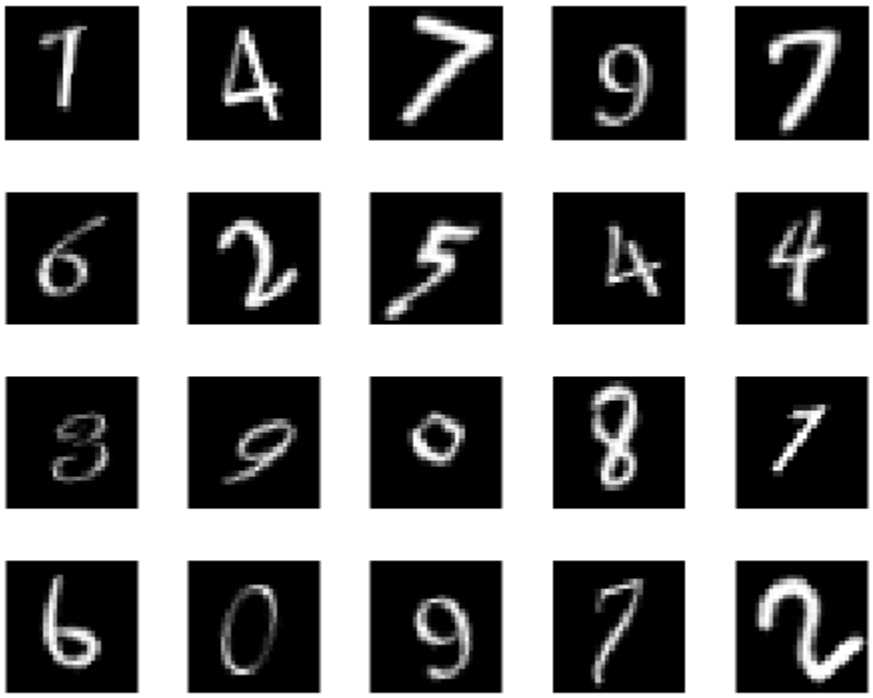
The data set contains synthetic images of handwritten digits together with the corresponding angles (in degrees) by which each image is rotated.

Load the training and validation images as 4-D arrays using `digitTrain4DArrayData` and `digitTest4DArrayData`. The outputs `YTrain` and `YValidation` are the rotation angles in degrees. The training and validation data sets each contain 5000 images.

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
[XValidation,~,YValidation] = digitTest4DArrayData;
```

Display 20 random training images using `imshow`.

```
numTrainImages = numel(YTrain);
figure
idx = randperm(numTrainImages,20);
for i = 1:numel(idx)
    subplot(4,5,i)
    imshow(XTrain(:,:,:,idx(i)))
end
```



Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, `'fc'` and `'classoutput'` in `digitsNet`, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network for regression, replace these two layers with new layers adapted to the task.

Replace the final fully connected layer, the softmax layer, and the classification output layer with a fully connected layer of size 1 (the number of responses) and a regression layer.

```
numResponses = 1;
layers = [
    layers(1:12)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

Freeze Initial Layers

The network is now ready to be retrained on the new data. Optionally, you can "freeze" the weights of earlier layers in the network by setting the learning rates in those layers to zero. During training, `trainNetwork` does not update the parameters of the frozen layers. Because the gradients of the frozen layers do not need to be computed, freezing the weights of many initial layers can significantly speed up network training. If the new data set is small, then freezing earlier network layers can also prevent those layers from overfitting to the new data set.

Use the supporting function `freezeWeights` to set the learning rates to zero in the first 12 layers.

```
layers(1:12) = freezeWeights(layers(1:12));
```

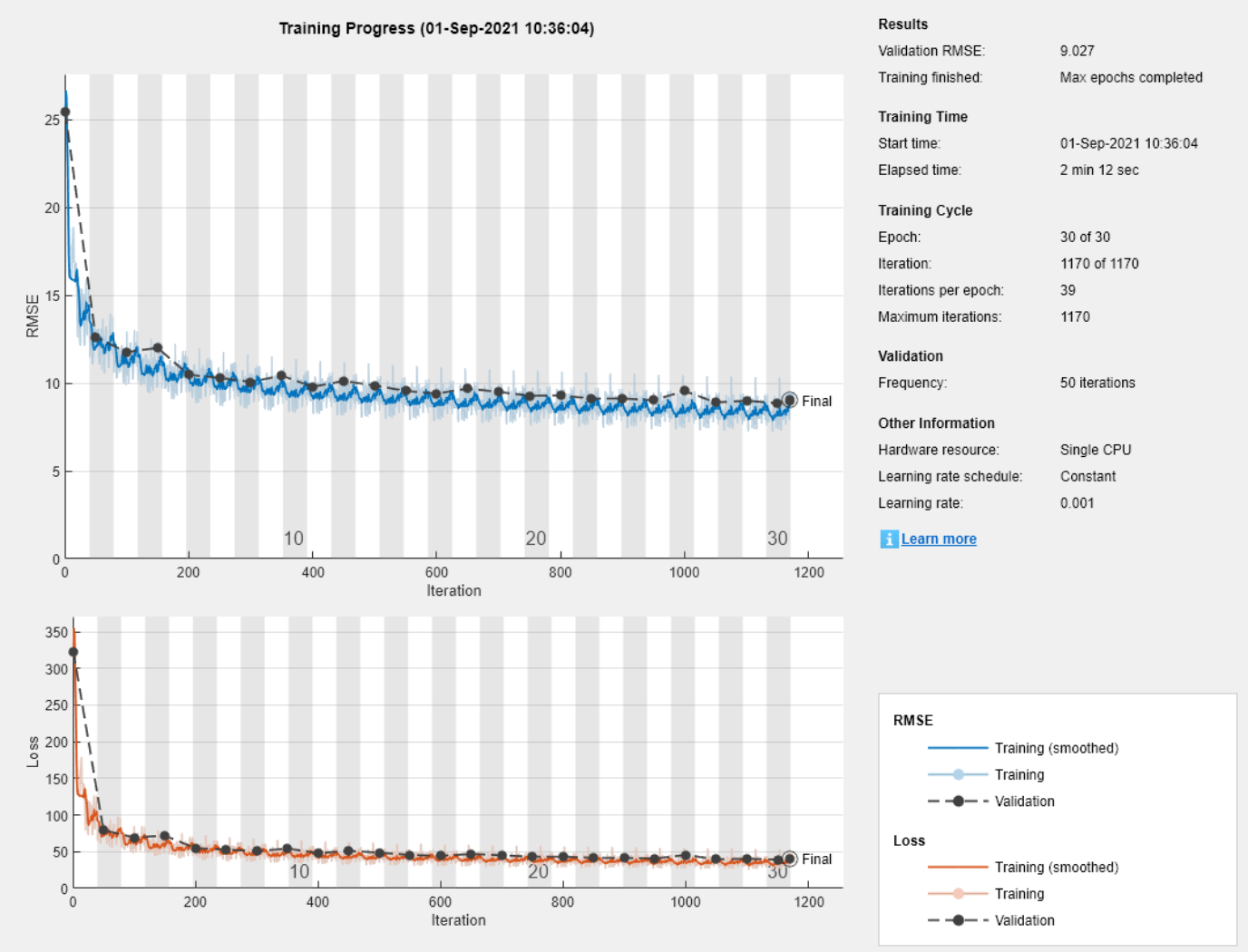
Train Network

Create the network training options. Set the initial learn rate to 0.001. Monitor the network accuracy during training by specifying validation data. Turn on the training progress plot, and turn off the command window output.

```
options = trainingOptions('sgdm',...
    'InitialLearnRate',0.001, ...
    'ValidationData',{XValidation,YValidation},...
    'Plots','training-progress',...
    'Verbose',false);
```

Create the network using `trainNetwork`. This command uses a compatible GPU if available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses the CPU.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Test Network

Test the performance of the network by evaluating the accuracy on the validation data.

Use `predict` to predict the angles of rotation of the validation images.

```
YPred = predict(net,XValidation);
```

Evaluate the performance of the model by calculating:

1. The percentage of predictions within an acceptable error margin
2. The root-mean-square error (RMSE) of the predicted and actual angles of rotation

Calculate the prediction error between the predicted and actual angles of rotation.

```
predictionError = YValidation - YPred;
```

Calculate the number of predictions within an acceptable error margin from the true angles. Set the threshold to be 10 degrees. Calculate the percentage of predictions within this threshold.

```
thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numImagesValidation = numel(YValidation);

accuracy = numCorrect/numImagesValidation
```

```
accuracy = 0.7532
```

Use the root-mean-square error (RMSE) to measure the differences between the predicted and actual angles of rotation.

```
 rmse = sqrt(mean(predictionError.^2))
```

```
rmse = single
    9.0271
```

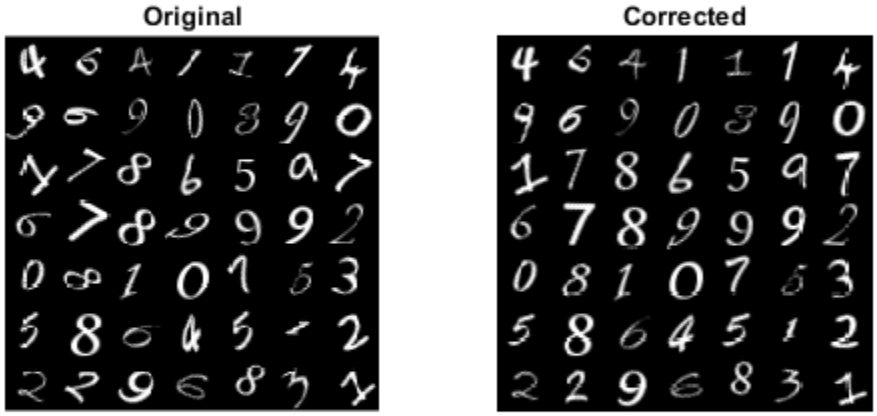
Correct Digit Rotations

You can use functions from Image Processing Toolbox to straighten the digits and display them together. Rotate 49 sample digits according to their predicted angles of rotation using `imrotate` (Image Processing Toolbox).

```
idx = randperm(numImagesValidation,49);
for i = 1:numel(idx)
    I = XValidation(:,:,:,idx(i));
    Y = YPred(idx(i));
    XValidationCorrected(:,:,:,i) = imrotate(I,Y,'bicubic','crop');
end
```

Display the original digits with their corrected rotations. Use `montage` (Image Processing Toolbox) to display the digits together in a single image.

```
figure
subplot(1,2,1)
montage(XValidation(:,:,:,idx))
title('Original')

subplot(1,2,2)
montage(XValidationCorrected)
title('Corrected')
```



See Also

`regressionLayer` | `classificationLayer`

Related Examples

- “Deep Learning in MATLAB” on page 1-2

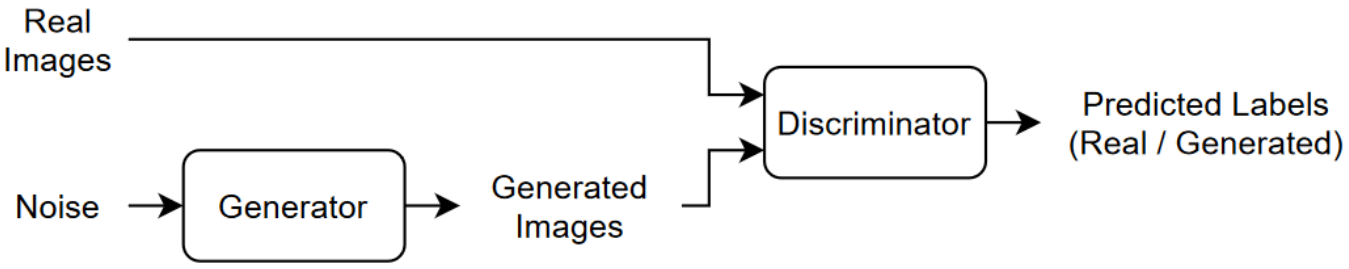
Train Generative Adversarial Network (GAN)

This example shows how to train a generative adversarial network to generate images.

A generative adversarial network (GAN) is a type of deep learning network that can generate data with similar characteristics as the input real data.

A GAN consists of two networks that train together:

1. Generator — Given a vector of random values (latent inputs) as input, this network generates data with the same structure as the training data.
2. Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as `"real"` or `"generated"`.



To train a GAN, train both networks simultaneously to maximize the performance of both:

- Train the generator to generate data that "fools" the discriminator.
- Train the discriminator to distinguish between real and generated data.

To optimize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate data that the discriminator classifies as `"real"`.

To optimize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. That is, the objective of the discriminator is to not be "fooled" by the generator.

Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.

Load Training Data

Download and extract the Flowers data set [1].

```
url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flowers data set (218 MB)...")
    websave(filename,url);
```

```
    untar(filename,downloadFolder)
end
```

Create an image datastore containing the photos of the flowers.

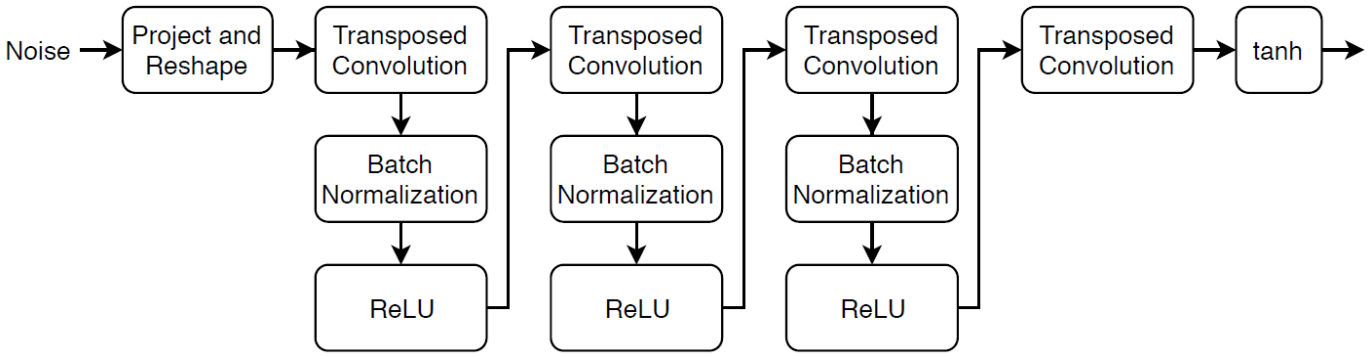
```
datasetFolder = fullfile(imageFolder);

imds = imageDatastore(datasetFolder,IncludeSubfolders=true);
```

Augment the data to include random horizontal flipping and resize the images to have size 64-by-64.

```
augmenter = imageDataAugmenter(RandXReflection=true);
augimds = augmentedImageDatastore([64 64],imds,DataAugmentation=augmenter);
```

Define Generator Network

Define the following network architecture, which generates images from random vectors of size 100.



This network:

- Converts the random vectors of size 100 to 7-by-7-by-128 arrays using a fully connected layer followed by a reshape operation.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers with batch normalization and ReLU layers.

Define this network architecture as a layer graph and specify the following network properties.

- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and cropping of the output on each edge.
- For the final transposed convolution layer, specify three 5-by-5 filters corresponding to the three RGB channels of the generated images, and the output size of the previous layer.
- At the end of the network, include a tanh layer.

To project and reshape the noise input, use a fully connected layer followed by a reshape operation specifed as a function layer with function given by the `feature2image` function, attached to this example as a supporting file. To access this function, open this example as a live script.

```
filterSize = 5;
numFilters = 64;
numLatentInputs = 100;

projectionSize = [4 4 512];
```

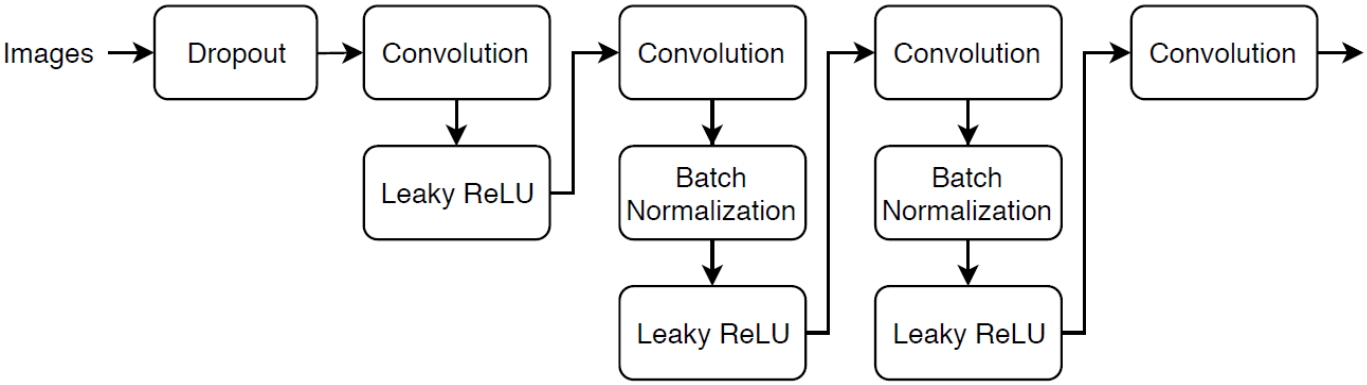
```
layersGenerator = [
    featureInputLayer(numLatentInputs)
    fullyConnectedLayer(prod(projectionSize))
    functionLayer(@(X) feature2image(X,projectionSize),Formattable=true)
    transposedConv2dLayer(filterSize,4*numFilters)
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,2*numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,3,Stride=2,Cropping="same")
    tanhLayer];

lgraphGenerator = layerGraph(layersGenerator);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Define Discriminator Network

Define the following network, which classifies real and generated 64-by-64 images.



Create a network that takes 64-by-64-by-3 images and returns a scalar prediction score using a series of convolution layers with batch normalization and leaky ReLU layers. Add noise to the input images using dropout.

- For the dropout layer, specify a dropout probability of 0.5.
- For the convolution layers, specify 5-by-5 filters with an increasing number of filters for each layer. Also specify a stride of 2 and padding of the output.
- For the leaky ReLU layers, specify a scale of 0.2.
- For the final layer, specify a convolutional layer with one 4-by-4 filter.

To output the probabilities in the range [0,1], use the `sigmoid` function in the `modelGradients` function, listed in the Model Gradients Function on page 3-0 section of the example.

```
dropoutProb = 0.5;
numFilters = 64;
```

```
scale = 0.2;

inputSize = [64 64 3];
filterSize = 5;

layersDiscriminator = [
    imageInputLayer(inputSize,Normalization="none")
    dropoutLayer(dropoutProb)
    convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same")
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(4,1)];

lgraphDiscriminator = layerGraph(layersDiscriminator);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

Define Model Gradients and Loss Functions

Create the function `modelGradients`, listed in the Model Gradients Function on page 3-0 section of the example, which takes as input the generator and discriminator networks, a mini-batch of input data, an array of random values, and the flip factor, and returns the gradients of the loss with respect to the learnable parameters in the networks and the scores of the two networks.

Specify Training Options

Train with a mini-batch size of 128 for 500 epochs. For larger data sets, you might not need to train for as many epochs.

```
numEpochs = 500;
miniBatchSize = 128;
```

Specify the options for Adam optimization. For both networks, specify:

- A learning rate of 0.0002
- A gradient decay factor of 0.5
- A squared gradient decay factor of 0.999

```
learnRate = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

If the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. To better balance the learning of the discriminator and the generator, add noise to the real data by randomly flipping the labels.

Specify a `flipFactor` value of 0.3 to flip 30% of the real labels (15% of the total labels). Note that this does not impair the generator as all the generated images are still labeled correctly.

```
flipFactor = 0.3;
```

Display the generated validation images every 100 iterations.

```
validationFrequency = 100;
```

Train Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to rescale the images in the range `[-1,1]`.
- Discard any partial mini-batches with less than 128 observations.
- Format the image data with the dimension labels `"SSCB"` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Train on a GPU if one is available. When the `OutputEnvironment` option of `minibatchqueue` is `"auto"`, `minibatchqueue` converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
augimds.MiniBatchSize = miniBatchSize;

executionEnvironment = "auto";

mbq = minibatchqueue(augimds,...
    MiniBatchSize=miniBatchSize,...
    PartialMiniBatch="discard",...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat="SSCB",...
    OutputEnvironment=executionEnvironment);
```

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display a batch of generated images using a held-out array of random values to input to the generator as well as a plot of the scores.

Initialize the parameters for Adam.

```
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

To monitor the training progress, display a batch of generated images using a held-out batch of fixed random vectors fed into the generator and plot the network scores.

Create an array of held-out random values.

```
numValidationImages = 25;
ZValidation = randn(numLatentInputs,numValidationImages,"single");
```

Convert the data to `dlarray` objects and specify the dimension labels `"CB"` (channel, batch).

```
dlZValidation = dlarray(ZValidation,"CB");
```

For GPU training, convert the data to `gpuArray` objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZValidation = gpuArray(dlZValidation);
end
```

Initialize the training progress plots. Create a figure and resize it to have twice the width.

```
f = figure;
f.Position(3) = 2*f.Position(3);
```

Create a subplot for the generated images and the network scores.

```
imageAxes = subplot(1,2,1);
scoreAxes = subplot(1,2,2);
```

Initialize the animated lines for the scores plot.

```
lineScoreGenerator = animatedline(scoreAxes,Color=[0 0.447 0.741]);
lineScoreDiscriminator = animatedline(scoreAxes,Color=[0.85 0.325 0.098]);
legend("Generator","Discriminator");
ylim([0 1])
xlabel("Iteration")
ylabel("Score")
grid on
```

Train the GAN. For each epoch, shuffle the datastore and loop over mini-batches of data.

For each mini-batch:

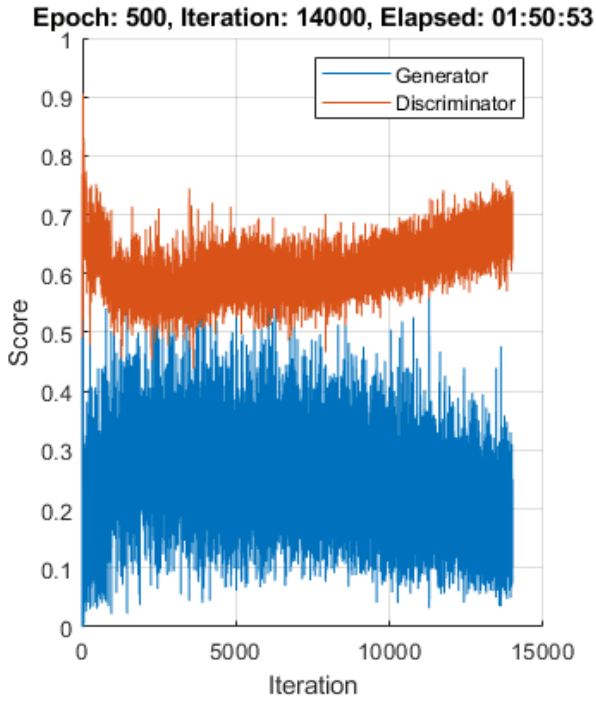
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.
- Plot the scores of the two networks.
- After every `validationFrequency` iterations, display a batch of generated images for a fixed held-out generator input.

Training can take some time to run.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Reset and shuffle datastore.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        dlX = next(mbq);

        % Generate latent inputs for the generator network. Convert to
        % dlarray and specify the dimension labels "CB" (channel, batch).
        % If training on a GPU, then convert latent inputs to gpuArray.
```

```
Z = randn(numLatentInputs,miniBatchSize,"single");
dlZ = dlarray(Z,"CB");

if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
end

% Evaluate the model gradients and the generator state using
% dlfeval and the modelGradients function listed at the end of the
% example.
[gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscri
    dlfeval(@modelGradients, dlnetGenerator, dlnetDiscriminator, dlX, dlZ, flipFactor);
dlnetGenerator.State = stateGenerator;

% Update the discriminator network parameters.
[dlnetDiscriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
    adamupdate(dlnetDiscriminator, gradientsDiscriminator, ...
    trailingAvgDiscriminator, trailingAvgSqDiscriminator, iteration, ...
    learnRate, gradientDecayFactor, squaredGradientDecayFactor);

% Update the generator network parameters.
[dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
    adamupdate(dlnetGenerator, gradientsGenerator, ...
    trailingAvgGenerator, trailingAvgSqGenerator, iteration, ...
    learnRate, gradientDecayFactor, squaredGradientDecayFactor);

% Every validationFrequency iterations, display batch of generated images using the
% held-out generator input.
if mod(iteration,validationFrequency) == 0 || iteration == 1
    % Generate images using the held-out generator input.
    dlXGeneratedValidation = predict(dlnetGenerator,dlZValidation);

    % Tile and rescale the images in the range [0 1].
    I = imtile(extractdata(dlXGeneratedValidation));
    I = rescale(I);

    % Display the images.
    subplot(1,2,1);
    image(imageAxes,I)
    xticklabels([]);
    yticklabels([]);
    title("Generated Images");
end

% Update the scores plot.
subplot(1,2,2)
addpoints(lineScoreGenerator,iteration,...
    double(gather(extractdata(scoreGenerator))));

addpoints(lineScoreDiscriminator,iteration,...
    double(gather(extractdata(scoreDiscriminator))));

% Update the title with training progress information.
D = duration(0,0,toc(start),Format="hh:mm:ss");
title(...
    "Epoch: " + epoch + ", " + ...
    "Iteration: " + iteration + ", " + ...
    "Elapsed: " + string(D))
```

```
        drawnow
    end
end
```



Here, the discriminator has learned a strong feature representation that identifies real images among generated images. In turn, the generator has learned a similarly strong feature representation that allows it to generate images similar to the training data.

The training plot shows the scores of the generator and discriminator networks. To learn more about how to interpret the network scores, see “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182.

Generate New Images

To generate new images, use the `predict` function on the generator with a `dlarray` object containing a batch of random vectors. To display the images together, use the `imtile` function and rescale the images using the `rescale` function.

Create a `dlarray` object containing a batch of 25 random vectors to input to the generator network.

```
numObservations = 25;
ZNew = randn(numLatentInputs,numObservations,"single");
dlZNew = dlarray(ZNew,"CB");
```

To generate images using the GPU, also convert the data to `gpuArray` objects.

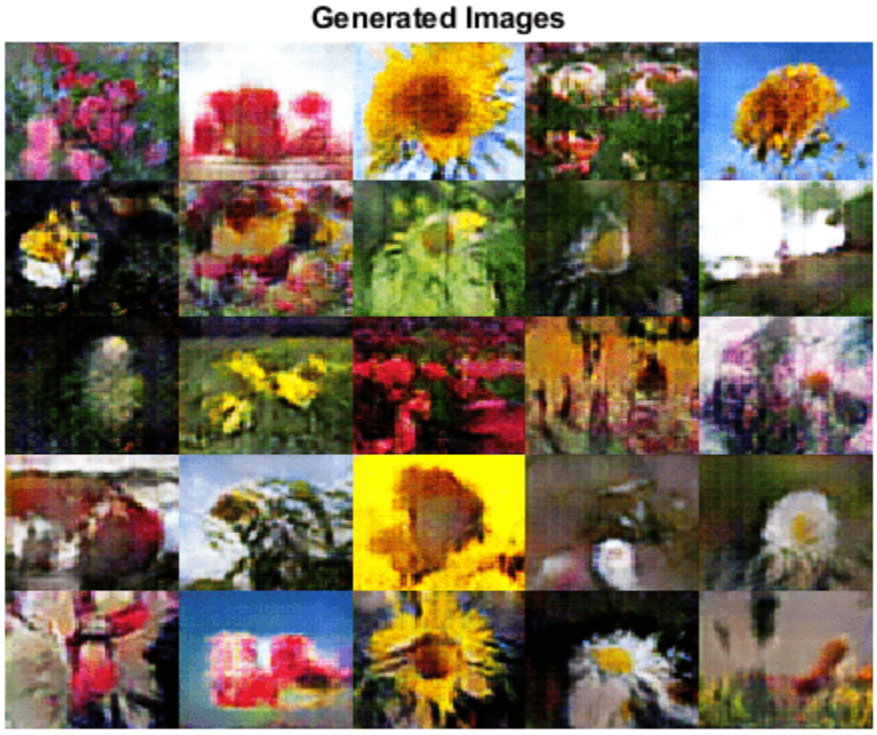
```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
end
```

Generate new images using the `predict` function with the generator and the input data.

```
dlXGeneratedNew = predict(dlnetGenerator,dlZNew);
```

Display the images.

```
I = imtile(extractdata(dlXGeneratedNew));
I = rescale(I);
figure
image(I)
axis off
title("Generated Images")
```



Model Gradients Function

The function `modelGradients` takes as input the generator and discriminator `dlnetwork` objects `dlnetGenerator` and `dlnetDiscriminator`, a mini-batch of input data `dlX`, an array of random values `dlZ`, and the percentage of real labels to flip `flipFactor`, and returns the gradients of the loss with respect to the learnable parameters in the networks, the generator state, and the scores of the two networks. Because the discriminator output is not in the range [0,1], the `modelGradients` function applies the sigmoid function to convert it into probabilities.

```
function [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscr
    modelGradients(dlnetGenerator, dlnetDiscriminator, dlX, dlZ, flipFactor)

% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetDiscriminator, dlX);

% Calculate the predictions for generated data with the discriminator network.
```

```
[dlXGenerated,stateGenerator] = forward(dlnetGenerator,dlZ);
dlYPredGenerated = forward(dlnetDiscriminator, dlXGenerated);

% Convert the discriminator outputs to probabilities.
probGenerated = sigmoid(dlYPredGenerated);
probReal = sigmoid(dlYPred);

% Calculate the score of the discriminator.
scoreDiscriminator = (mean(probReal) + mean(1-probGenerated)) / 2;

% Calculate the score of the generator.
scoreGenerator = mean(probGenerated);

% Randomly flip a fraction of the labels of the real images.
numObservations = size(probReal,4);
idx = randperm(numObservations,floor(flipFactor * numObservations));

% Flip the labels.
probReal(:,:,:,idx) = 1 - probReal(:,:,:,idx);

% Calculate the GAN loss.
[lossGenerator, lossDiscriminator] = ganLoss(probReal,probGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsGenerator = dlgradient(lossGenerator, dlnetGenerator.Learnables,RetainData=true);
gradientsDiscriminator = dlgradient(lossDiscriminator, dlnetDiscriminator.Learnables);

end
```

GAN Loss Function and Scores

The objective of the generator is to generate data that the discriminator classifies as `"real"`. To maximize the probability that images from the generator are classified as real by the discriminator, minimize the negative log likelihood function.

Given the output Y of the discriminator:

- $\widehat{Y} = \sigma(Y)$ is the probability that the input image belongs to the class `"real"`.
- $1 - \widehat{Y}$ is the probability that the input image belongs to the class `"generated"`.

Note that the sigmoid operation σ happens in the `modelGradients` function. The loss function for the generator is given by

$$\text{lossGenerator} = -\text{mean}\left(\log\left(\widehat{Y}_{\text{Generated}}\right)\right),$$

where $\widehat{Y}_{Generated}$ contains the discriminator output probabilities for the generated images.

The objective of the discriminator is to not be "fooled" by the generator. To maximize the probability that the discriminator successfully discriminates between the real and generated images, minimize the sum of the corresponding negative log likelihood functions.

The loss function for the discriminator is given by

$$\text{lossDiscriminator} = -\text{mean}\left(\log\left(\widehat{Y}_{\text{Real}}\right)\right) - \text{mean}\left(\log\left(1 - \widehat{Y}_{\text{Generated}}\right)\right),$$

where $\widehat{Y}_{Real}$ contains the discriminator output probabilities for the real images.

To measure on a scale from 0 to 1 how well the generator and discriminator achieve their respective goals, you can use the concept of score.

The generator score is the average of the probabilities corresponding to the discriminator output for the generated images:

$$\text{scoreGenerator} = \text{mean}\left(\widehat{Y}_{\text{Generated}}\right).$$

The discriminator score is the average of the probabilities corresponding to the discriminator output for both the real and generated images:

$$\text{scoreDiscriminator} = \frac{1}{2}\text{mean}\left(\widehat{Y}_{\text{Real}}\right) + \frac{1}{2}\text{mean}\left(1 - \widehat{Y}_{\text{Generated}}\right).$$

The score is inversely proportional to the loss but effectively contains the same information.

```
function [lossGenerator, lossDiscriminator] = ganLoss(probReal,probGenerated)

% Calculate the loss for the discriminator network.
lossDiscriminator = -mean(log(probReal)) - mean(log(1-probGenerated));

% Calculate the loss for the generator network.
lossGenerator = -mean(log(probGenerated));

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array.
2. Rescale the images to be in the range `[-1,1]`.

```
function X = preprocessMiniBatch(data)

% Concatenate mini-batch
X = cat(4,data{:});

% Rescale the images in the range [-1 1].
X = rescale(X,-1,1,InputMin=0,InputMax=255);

end
```

References

1. The TensorFlow Team. *Flowers* http://download.tensorflow.org/example_images/flower_photos.tgz
2. Radford, Alec, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.” Preprint, submitted November 19, 2015. http://arxiv.org/abs/1511.06434.

See Also

`dlnetwork` | `forward` | `predict` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate` | `minibatchqueue`

More About

- “Train Conditional Generative Adversarial Network (CGAN)” on page 3-88
- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182
- “Train Fast Style Transfer Network” on page 3-113
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

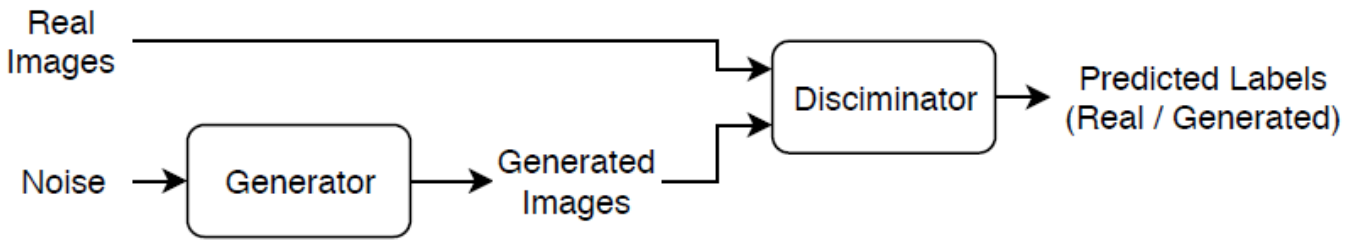
Train Conditional Generative Adversarial Network (CGAN)

This example shows how to train a conditional generative adversarial network to generate images.

A generative adversarial network (GAN) is a type of deep learning network that can generate data with similar characteristics as the input training data.

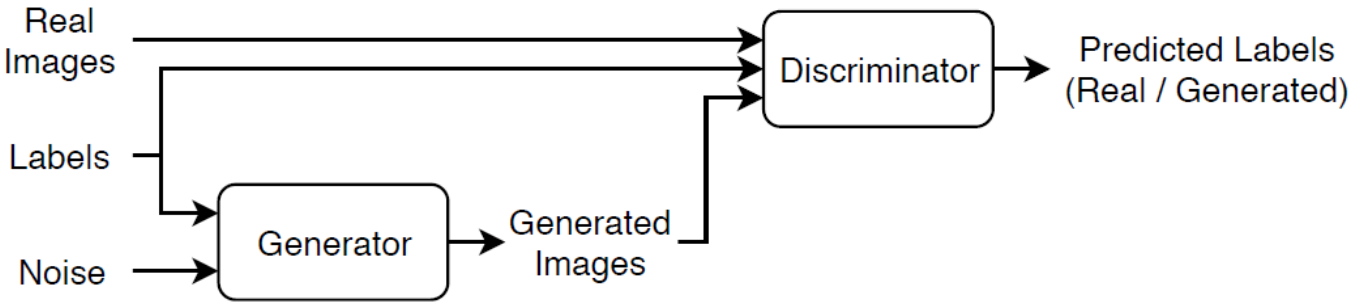
A GAN consists of two networks that train together:

1. Generator — Given a vector of random values as input, this network generates data with the same structure as the training data.
2. Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as `"real"` or `"generated"`.



A *conditional* generative adversarial network (CGAN) is a type of GAN that also takes advantage of labels during the training process.

1. Generator — Given a label and random array as input, this network generates data with the same structure as the training data observations corresponding to the same label.
2. Discriminator — Given batches of labeled data containing observations from both the training data and generated data from the generator, this network attempts to classify the observations as `"real"` or `"generated"`.



To train a conditional GAN, train both networks simultaneously to maximize the performance of both:

- Train the generator to generate data that "fools" the discriminator.
- Train the discriminator to distinguish between real and generated data.

To maximize the performance of the generator, maximize the loss of the discriminator when given generated labeled data. That is, the objective of the generator is to generate labeled data that the discriminator classifies as `"real"`.

To maximize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated labeled data. That is, the objective of the discriminator is to not be "fooled" by the generator.

Ideally, these strategies result in a generator that generates convincingly realistic data that corresponds to the input labels and a discriminator that has learned strong feature representations that are characteristic of the training data for each label.

Load Training Data

Download and extract the Flowers data set [1].

```
url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flowers data set (218 MB)...")
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

Create an image datastore containing the photos of the flowers.

```
datasetFolder = fullfile(imageFolder);

imds = imageDatastore(datasetFolder,IncludeSubfolders=true,LabelSource="foldernames");
```

View the number of classes.

```
classes = categories(imds.Labels);
numClasses = numel(classes)
```

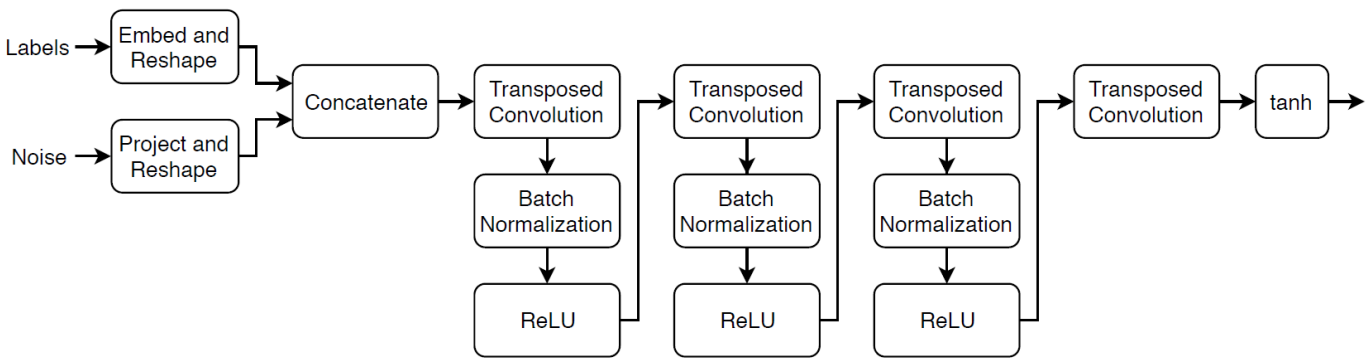
```
numClasses = 5
```

Augment the data to include random horizontal flipping and resize the images to have size 64-by-64.

```
augmenter = imageDataAugmenter(RandXReflection=true);
augimds = augmentedImageDatastore([64 64],imds,DataAugmentation=augmenter);
```

Define Generator Network

Define the following two-input network, which generates images given random vectors of size 100 and corresponding labels.



This network:

- Converts the random vectors of size 100 to 4-by-4-by-1024 arrays using a fully connected layer followed by a reshape operation.
- Converts the categorical labels to embedding vectors and reshapes them to a 4-by-4 array.
- Concatenates the resulting images from the two inputs along the channel dimension. The output is a 4-by-4-by-1025 array.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers with batch normalization and ReLU layers.

Define this network architecture as a layer graph and specify the following network properties.

- For the categorical inputs, use an embedding dimension of 50.
- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and `"same"` cropping of the output.
- For the final transposed convolution layer, specify a three 5-by-5 filter corresponding to the three RGB channels of the generated images.
- At the end of the network, include a tanh layer.

To project and reshape the noise input, use a fully connected layer followed by a reshape operation specifed as a function layer with function given by the `feature2image` function, attached to this example as a supporting file. To embed the categorical labels, use the custom layer `embeddingLayer` attached to this example as a supporting file. To access these supporting files, open the example as a live script.

```
numLatentInputs = 100;
embeddingDimension = 50;
numFilters = 64;

filterSize = 5;
projectionSize = [4 4 1024];

layersGenerator = [
    featureInputLayer(numLatentInputs)
    fullyConnectedLayer(prod(projectionSize))
    functionLayer(@(X) feature2image(X,projectionSize),Formattable=true)
    concatenationLayer(3,2,Name="cat");
    transposedConv2dLayer(filterSize,4*numFilters)
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,2*numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,numFilters,Stride=2,Cropping="same")
    batchNormalizationLayer
    reluLayer
    transposedConv2dLayer(filterSize,3,Stride=2,Cropping="same")
    tanhLayer];

lgraphGenerator = layerGraph(layersGenerator);

layers = [
    featureInputLayer(1)
    embeddingLayer(embeddingDimension,numClasses)
```

```
    fullyConnectedLayer(prod(projectionSize(1:2)))
    functionLayer(@(X) feature2image(X,[projectionSize(1:2) 1]),Formattable=true,Name="emb_reshap
```

```
lgraphGenerator = addLayers(lgraphGenerator,layers);
lgraphGenerator = connectLayers(lgraphGenerator,"emb_reshape","cat/in2");
```

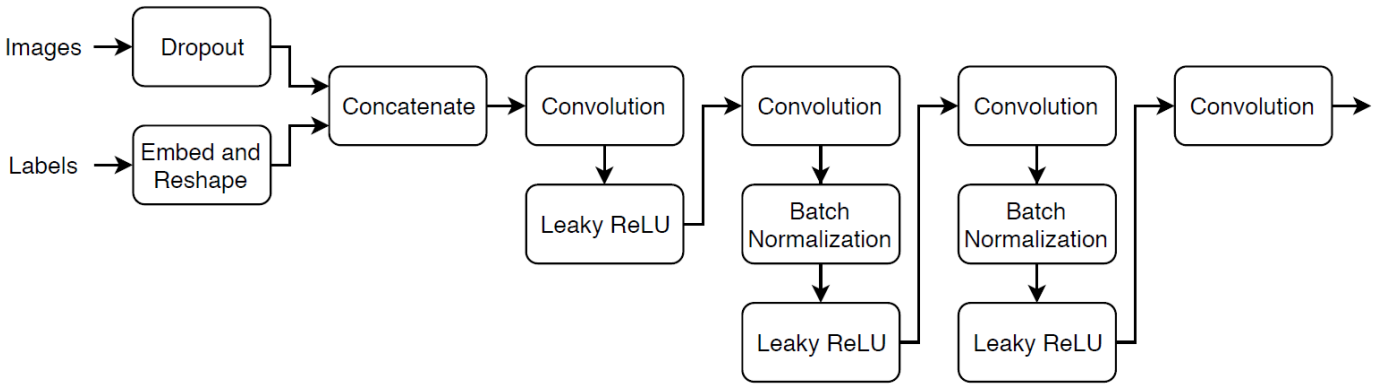
To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator)
```

```
dlnetGenerator =
  dlnetwork with properties:

         Layers: [19×1 nnet.cnn.layer.Layer]
    Connections: [18×2 table]
     Learnables: [19×3 table]
          State: [6×3 table]
     InputNames: {'input'  'input_1'}
    OutputNames: {'layer_2'}
    Initialized: 1
```

Define Discriminator Network

Define the following two-input network, which classifies real and generated 64-by-64 images given a set of images and the corresponding labels.



Create a network that takes as input 64-by-64-by-1 images and the corresponding labels and outputs a scalar prediction score using a series of convolution layers with batch normalization and leaky ReLU layers. Add noise to the input images using dropout.

- For the dropout layer, specify a dropout probability of 0.75.
- For the convolution layers, specify 5-by-5 filters with an increasing number of filters for each layer. Also specify a stride of 2 and padding of the output on each edge.
- For the leaky ReLU layers, specify a scale of 0.2.
- For the final layer, specify a convolution layer with one 4-by-4 filter.

```
dropoutProb = 0.75;
numFilters = 64;
scale = 0.2;
```

```
inputSize = [64 64 3];
filterSize = 5;

layersDiscriminator = [
    imageInputLayer(inputSize,Normalization="none")
    dropoutLayer(dropoutProb)
    concatenationLayer(3,2,Name="cat")
    convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same")
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same")
    batchNormalizationLayer
    leakyReluLayer(scale)
    convolution2dLayer(4,1)];

lgraphDiscriminator = layerGraph(layersDiscriminator);

layers = [
    featureInputLayer(1)
    embeddingLayer(embeddingDimension,numClasses)
    fullyConnectedLayer(prod(inputSize(1:2)))
    functionLayer(@(X) feature2image(X,[inputSize(1:2) 1]),Formattable=true,Name="emb_reshape")];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,"emb_reshape","cat/in2");
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator)
```

```
dlnetDiscriminator =
  dlnetwork with properties:

         Layers: [19×1 nnet.cnn.layer.Layer]
    Connections: [18×2 table]
     Learnables: [19×3 table]
          State: [6×3 table]
     InputNames: {'imageinput'  'input'}
    OutputNames: {'conv_5'}
    Initialized: 1
```

Define Model Gradients and Loss Functions

Create the function `modelGradients`, listed in the Model Gradients Function on page 3-0 section of the example, which takes as input the generator and discriminator networks, a mini-batch of input data, and an array of random values, and returns the gradients of the loss with respect to the learnable parameters in the networks and an array of generated images.

Specify Training Options

Train with a mini-batch size of 128 for 500 epochs.

```
numEpochs = 500;
miniBatchSize = 128;
```

Specify the options for Adam optimization. For both networks, use:

- A learning rate of 0.0002
- A gradient decay factor of 0.5
- A squared gradient decay factor of 0.999

```
learnRate = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Update the training progress plots every 100 iterations.

```
validationFrequency = 100;
```

If the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. To better balance the learning of the discriminator and the generator, randomly flip the labels of a proportion of the real images. Specify a flip factor of 0.5.

```
flipFactor = 0.5;
```

Train Model

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display a batch of generated images using a held-out array of random values to input into the generator and the network scores.

Use `minibatchqueue` to process and manage the mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to rescale the images in the range `[-1,1]`.
- Discard any partial mini-batches with less than 128 observations.
- Format the image data with the dimension labels `"SSCB"` (spatial, spatial, channel, batch).
- Format the label data with the dimension labels `"BC"` (batch, channel).
- Train on a GPU if one is available. When the `OutputEnvironment` option of `minibatchqueue` is `"auto"`, `minibatchqueue` converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.

```
augimds.MiniBatchSize = miniBatchSize;
executionEnvironment = "auto";

mbq = minibatchqueue(augimds, ...
    MiniBatchSize=miniBatchSize, ...
    PartialMiniBatch="discard", ...
    MiniBatchFcn=@preprocessData, ...
    MiniBatchFormat=["SSCB" "BC"], ...
    OutputEnvironment=executionEnvironment);
```

Initialize the parameters for the Adam optimizer.

```
velocityDiscriminator = [];
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

Initialize the plot of the training progress. Create a figure and resize it to have twice the width.

```
f = figure;
f.Position(3) = 2*f.Position(3);
```

Create subplots of the generated images and of the scores plot.

```
imageAxes = subplot(1,2,1);
scoreAxes = subplot(1,2,2);
```

Initialize animated lines for the scores plot.

```
lineScoreGenerator = animatedline(scoreAxes,Color=[0 0.447 0.741]);
lineScoreDiscriminator = animatedline(scoreAxes,Color=[0.85 0.325 0.098]);
```

Customize the appearance of the plots.

```
legend("Generator","Discriminator");
ylim([0 1])
xlabel("Iteration")
ylabel("Score")
grid on
```

To monitor training progress, create a held-out batch of 25 random vectors and a corresponding set of labels 1 through 5 (corresponding to the classes) repeated five times.

```
numValidationImagesPerClass = 5;
ZValidation = randn(numLatentInputs,numValidationImagesPerClass*numClasses,"single");

TValidation = single(repmat(1:numClasses,[1 numValidationImagesPerClass]));
```

Convert the data to `dlarray` objects and specify the dimension labels `"CB"` (channel, batch).

```
dlZValidation = dlarray(ZValidation,"CB");
dlTValidation = dlarray(TValidation,"CB");
```

For GPU training, convert the data to `gpuArray` objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZValidation = gpuArray(dlZValidation);
    dlTValidation = gpuArray(dlTValidation);
end
```

Train the conditional GAN. For each epoch, shuffle the data and loop over mini-batches of data.

For each mini-batch:

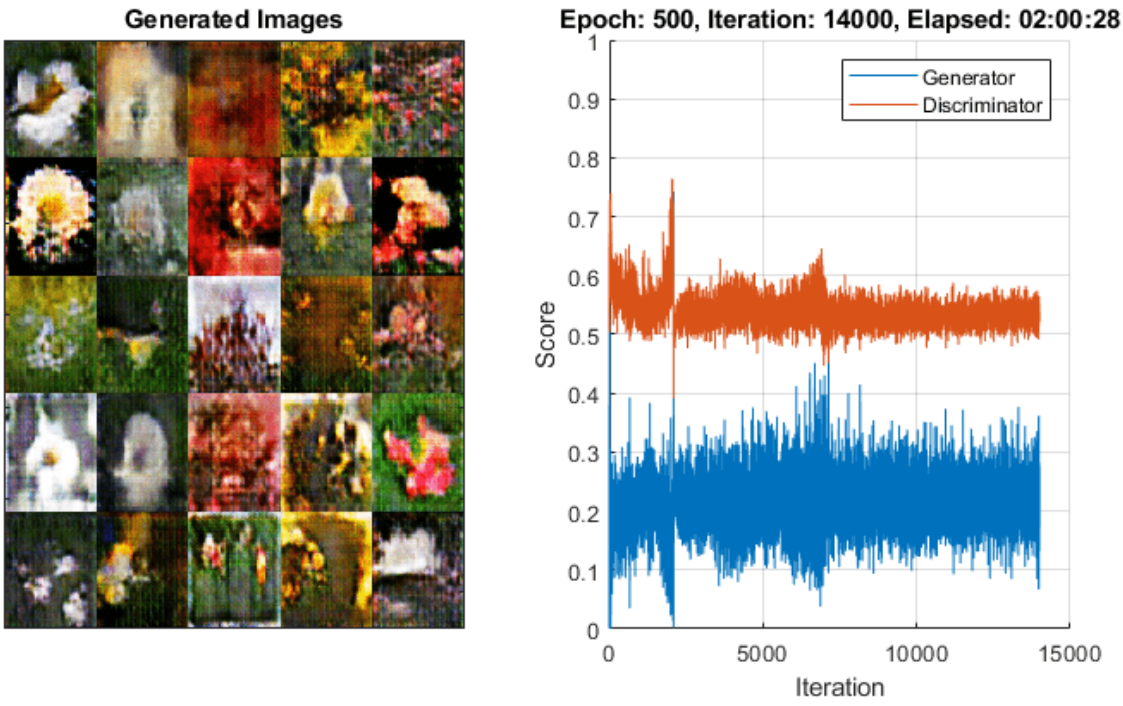
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

- Plot the scores of the two networks.
- After every `validationFrequency` iterations, display a batch of generated images for a fixed held-out generator input.

Training can take some time to run.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Reset and shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,dlT] = next(mbq);

        % Generate latent inputs for the generator network. Convert to
        % dlarray and specify the dimension labels "CB" (channel, batch).
        % If training on a GPU, then convert latent inputs to gpuArray.
        Z = randn(numLatentInputs,miniBatchSize,"single");
        dlZ = dlarray(Z,"CB");
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlZ = gpuArray(dlZ);
        end

        % Evaluate the model gradients and the generator state using
        % dlfeval and the modelGradients function listed at the end of the
        % example.
        [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscri
            dlfeval(@modelGradients, dlnetGenerator, dlnetDiscriminator, dlX, dlT, dlZ, flipFacto
        dlnetGenerator.State = stateGenerator;

        % Update the discriminator network parameters.
        [dlnetDiscriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
            adamupdate(dlnetDiscriminator, gradientsDiscriminator, ...
            trailingAvgDiscriminator, trailingAvgSqDiscriminator, iteration, ...
            learnRate, gradientDecayFactor, squaredGradientDecayFactor);

        % Update the generator network parameters.
        [dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
            adamupdate(dlnetGenerator, gradientsGenerator, ...
            trailingAvgGenerator, trailingAvgSqGenerator, iteration, ...
            learnRate, gradientDecayFactor, squaredGradientDecayFactor);

        % Every validationFrequency iterations, display batch of generated images using the
        % held-out generator input.
        if mod(iteration,validationFrequency) == 0 || iteration == 1

            % Generate images using the held-out generator input.
            dlXGeneratedValidation = predict(dlnetGenerator,dlZValidation,dlTValidation);
```

```
            % Tile and rescale the images in the range [0 1].
            I = imtile(extractdata(dlXGeneratedValidation), ...
                GridSize=[numValidationImagesPerClass numClasses]);
            I = rescale(I);

            % Display the images.
            subplot(1,2,1);
            image(imageAxes,I)
            xticklabels([]);
            yticklabels([]);
            title("Generated Images");
        end

        % Update the scores plot.
        subplot(1,2,2)
        addpoints(lineScoreGenerator,iteration,...
            double(gather(extractdata(scoreGenerator))));

        addpoints(lineScoreDiscriminator,iteration,...
            double(gather(extractdata(scoreDiscriminator))));

        % Update the title with training progress information.
        D = duration(0,0,toc(start),Format="hh:mm:ss");
        title(...
            "Epoch: " + epoch + ", " + ...
            "Iteration: " + iteration + ", " + ...
            "Elapsed: " + string(D))

        drawnow
    end
end
```



Here, the discriminator has learned a strong feature representation that identifies real images among generated images. In turn, the generator has learned a similarly strong feature representation that allows it to generate images similar to the training data. Each column corresponds to a single class.

The training plot shows the scores of the generator and discriminator networks. To learn more about how to interpret the network scores, see “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182.

Generate New Images

To generate new images of a particular class, use the `predict` function on the generator with a `dlarray` object containing a batch of random vectors and an array of labels corresponding to the desired classes. Convert the data to `dlarray` objects and specify the dimension labels `"CB"` (channel, batch). For GPU prediction, convert the data to `gpuArray` objects. To display the images together, use the `imtile` function and rescale the images using the `rescale` function.

Create an array of 36 vectors of random values corresponding to the first class.

```
numObservationsNew = 36;
idxClass = 1;
Z = randn(numLatentInputs,numObservationsNew,"single");
T = repmat(single(idxClass),[1 numObservationsNew]);
```

Convert the data to `dlarray` objects with the dimension labels `"SSCB"` (spatial, spatial, channels, batch).

```
dlZ = dlarray(Z,"CB");
dlT = dlarray(T,"CB");
```

To generate images using the GPU, also convert the data to `gpuArray` objects.

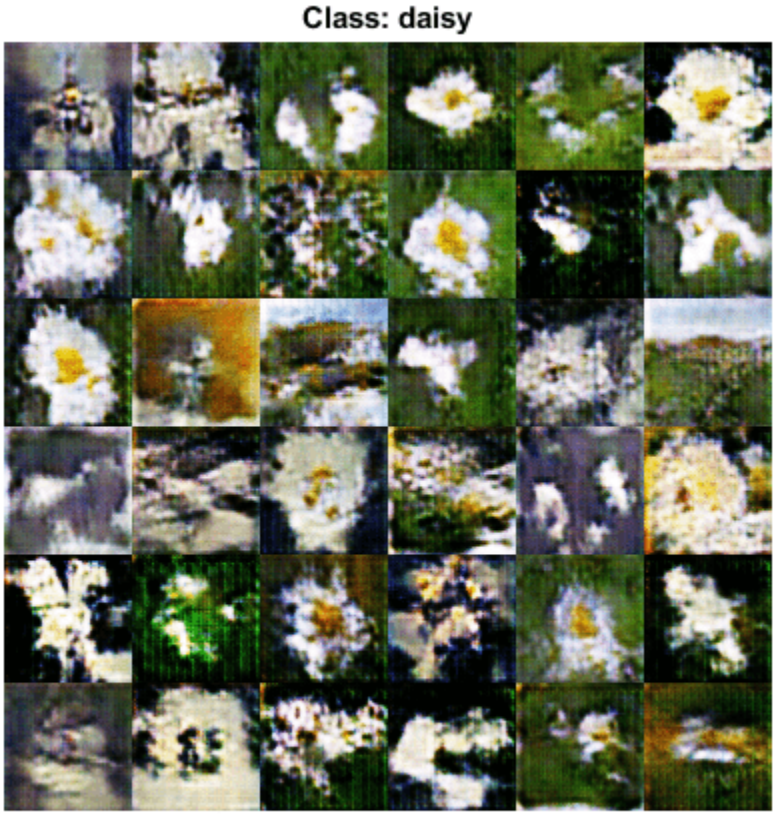
```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
    dlT = gpuArray(dlT);
end
```

Generate images using the `predict` function with the generator network.

```
dlXGenerated = predict(dlnetGenerator,dlZ,dlT);
```

Display the generated images in a plot.

```
figure
I = imtile(extractdata(dlXGenerated));
I = rescale(I);
imshow(I)
title("Class: " + classes(idxClass))
```



Here, the generator network generates images conditioned on the specified class.

Model Gradients Function

The function `modelGradients` takes as input the generator and discriminator `dlnetwork` objects `dlnetGenerator` and `dlnetDiscriminator`, a mini-batch of input data `dlX`, the corresponding labels `dlT`, and an array of random values `dlZ`, and returns the gradients of the loss with respect to the learnable parameters in the networks, the generator state, and the network scores.

If the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. To better balance the learning of the discriminator and the generator, randomly flip the labels of a proportion of the real images.

```
function [gradientsGenerator, gradientsDiscriminator, stateGenerator, scoreGenerator, scoreDiscr
    modelGradients(dlnetGenerator, dlnetDiscriminator, dlX, dlT, dlZ, flipFactor)

% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetDiscriminator, dlX, dlT);

% Calculate the predictions for generated data with the discriminator network.
[dlXGenerated,stateGenerator] = forward(dlnetGenerator, dlZ, dlT);
dlYPredGenerated = forward(dlnetDiscriminator, dlXGenerated, dlT);
```

```
% Calculate probabilities.
probGenerated = sigmoid(dlYPredGenerated);
probReal = sigmoid(dlYPred);

% Calculate the generator and discriminator scores.
scoreGenerator = mean(probGenerated);
scoreDiscriminator = (mean(probReal) + mean(1-probGenerated)) / 2;

% Flip labels.
numObservations = size(dlYPred,4);
idx = randperm(numObservations,floor(flipFactor * numObservations));
probReal(:,:,:,idx) = 1 - probReal(:,:,:,idx);

% Calculate the GAN loss.
[lossGenerator, lossDiscriminator] = ganLoss(probReal, probGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsGenerator = dlgradient(lossGenerator, dlnetGenerator.Learnables,RetainData=true);
gradientsDiscriminator = dlgradient(lossDiscriminator, dlnetDiscriminator.Learnables);

end
```

GAN Loss Function

The objective of the generator is to generate data that the discriminator classifies as `"real"`. To maximize the probability that images from the generator are classified as real by the discriminator, minimize the negative log likelihood function.

Given the output Y of the discriminator:

- $\widehat{Y} = \sigma(Y)$ is the probability that the input image belongs to the class `"real"`.
- $1 - \widehat{Y}$ is the probability that the input image belongs to the class `"generated"`.

Note the sigmoid operation σ happens in the `modelGradients` function. The loss function for the generator is given by

$$\text{lossGenerator} = -\text{mean}\left(\log\left(\widehat{Y}_{\text{Generated}}\right)\right),$$

where $\widehat{Y}_{Generated}$ contains the discriminator output probabilities for the generated images.

The objective of the discriminator is to not be "fooled" by the generator. To maximize the probability that the discriminator successfully discriminates between the real and generated images, minimize the sum of the corresponding negative log likelihood functions. The loss function for the discriminator is given by

$$\text{lossDiscriminator} = -\text{mean}\left(\log\left(\widehat{Y}_{\text{Real}}\right)\right) - \text{mean}\left(\log\left(1 - \widehat{Y}_{\text{Generated}}\right)\right),$$

where $\widehat{Y}_{Real}$ contains the discriminator output probabilities for the real images.

```
function [lossGenerator, lossDiscriminator] = ganLoss(scoresReal,scoresGenerated)

% Calculate losses for the discriminator network.
lossGenerated = -mean(log(1 - scoresGenerated));
lossReal = -mean(log(scoresReal));
```

```
% Combine the losses for the discriminator network.
lossDiscriminator = lossReal + lossGenerated;

% Calculate the loss for the generator network.
lossGenerator = -mean(log(scoresGenerated));

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image and label data from the input cell arrays and concatenate them into numeric arrays.
2. Rescale the images to be in the range `[-1,1]`.

```
function [X,T] = preprocessData(XCell,TCell)

% Extract image data from cell and concatenate
X = cat(4,XCell{:});

% Extract label data from cell and concatenate
T = cat(1,TCell{:});

% Rescale the images in the range [-1 1].
X = rescale(X,-1,1,InputMin=0,InputMax=255);

end
```

References

1. The TensorFlow Team. *Flowers* http://download.tensorflow.org/example_images/flower_photos.tgz

See Also

`dlnetwork` | `forward` | `predict` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182
- “Train Fast Style Transfer Network” on page 3-113
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

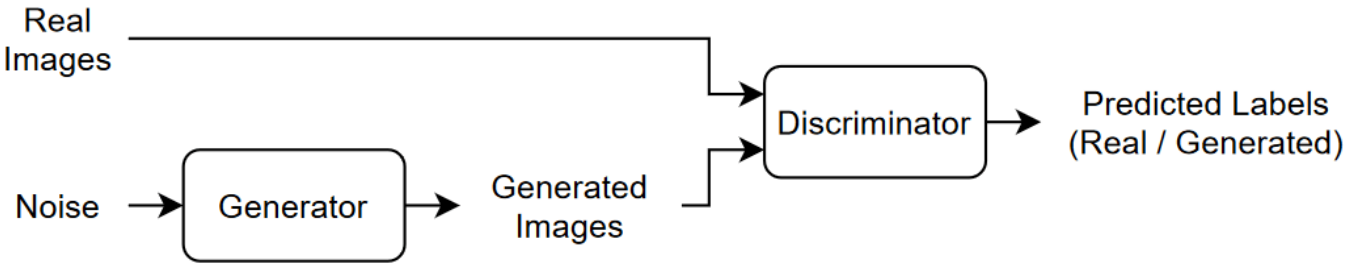
Train Wasserstein GAN with Gradient Penalty (WGAN-GP)

This example shows how to train a Wasserstein generative adversarial network with a gradient penalty (WGAN-GP) to generate images.

A generative adversarial network (GAN) is a type of deep learning network that can generate data with similar characteristics as the input real data.

A GAN consists of two networks that train together:

1. Generator — Given a vector of random values (latent inputs) as input, this network generates data with the same structure as the training data.
2. Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as "real" or "generated".



To train a GAN, train both networks simultaneously to maximize the performance of both:

- Train the generator to generate data that "fools" the discriminator.
- Train the discriminator to distinguish between real and generated data.

To optimize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate data that the discriminator classifies as "real". To optimize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. That is, the objective of the discriminator is to not be "fooled" by the generator.

Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data. However, [2] argues that the divergences which GANs typically minimize are potentially not continuous with respect to the generator's parameters, leading to training difficulty and introduces the Wasserstein GAN (WGAN) model that uses the Wasserstein loss to help stabilize training. A WGAN model can still produce poor samples or fail to converge because interactions between the weight constraint and the cost function can result in vanishing or exploding gradients. To address these issues, [3] introduces a gradient penalty which improves stability by penalizing gradients with large norm values at the cost of longer computational time. This type of model is known as a WGAN-GP model.

This example shows how to train a WGAN-GP model that can generate images with similar characteristics to a training set of images.

Load Training Data

Download and extract the Flowers data set [1].

```
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'flower_dataset.tgz');

imageFolder = fullfile(downloadFolder,'flower_photos');
if ~exist(imageFolder,'dir')
    disp('Downloading Flowers data set (218 MB)...')
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

Create an image datastore containing the photos of the flowers.

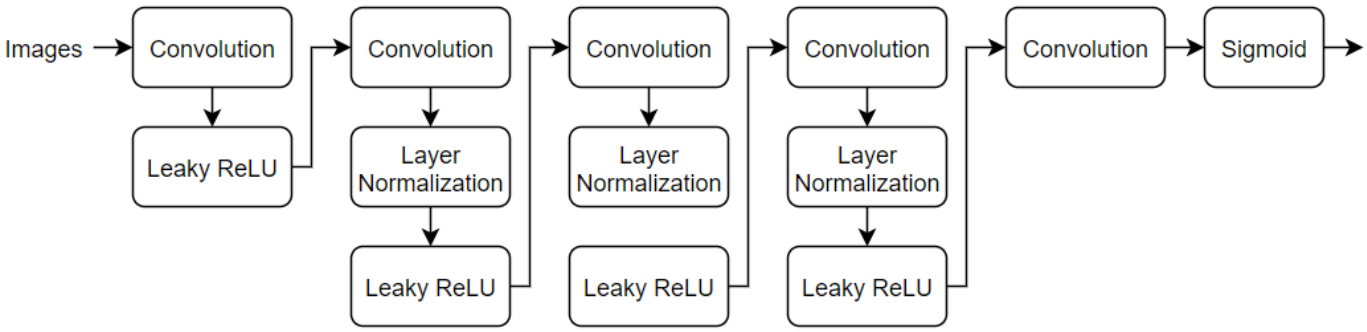
```
datasetFolder = fullfile(imageFolder);

imds = imageDatastore(datasetFolder, ...
    'IncludeSubfolders',true);
```

Augment the data to include random horizontal flipping and resize the images to have size 64-by-64.

```
augmenter = imageDataAugmenter('RandXReflection',true);
augimds = augmentedImageDatastore([64 64],imds,'DataAugmentation',augmenter);
```

Define Discriminator Network

Define the following network, which classifies real and generated 64-by-64 images.



Create a network that takes 64-by-64-by-3 images and returns a scalar prediction score using a series of convolution layers with batch normalization and leaky ReLU layers. To output the probabilities in the range [0,1], use a sigmoid layer.

- For the convolution layers, specify 5-by-5 filters with an increasing number of filters for each layer. Also specify a stride of 2 and padding of the output.
- For the leaky ReLU layers, specify a scale of 0.2.
- For the final convolution layer, specify one 4-by-4 filter.

```
numFilters = 64;
scale = 0.2;

inputSize = [64 64 3];
filterSize = 5;

layersD = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

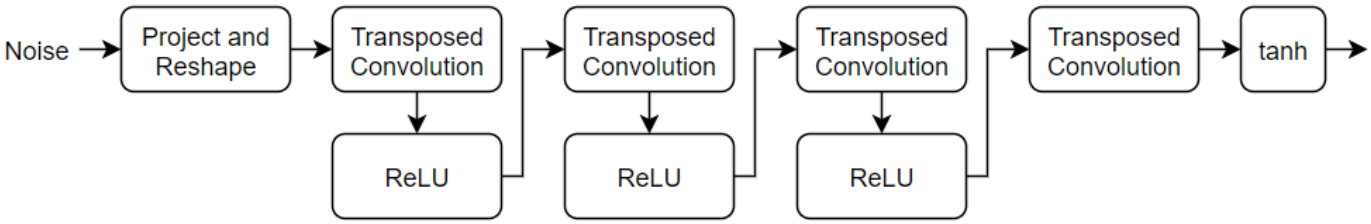
```
    convolution2dLayer(filterSize,numFilters,'Stride',2,'Padding','same','Name','conv1')
    leakyReluLayer(scale,'Name','lrelu1')
    convolution2dLayer(filterSize,2*numFilters,'Stride',2,'Padding','same','Name','conv2')
    layerNormalizationLayer('Name','bn2')
    leakyReluLayer(scale,'Name','lrelu2')
    convolution2dLayer(filterSize,4*numFilters,'Stride',2,'Padding','same','Name','conv3')
    layerNormalizationLayer('Name','bn3')
    leakyReluLayer(scale,'Name','lrelu3')
    convolution2dLayer(filterSize,8*numFilters,'Stride',2,'Padding','same','Name','conv4')
    layerNormalizationLayer('Name','bn4')
    leakyReluLayer(scale,'Name','lrelu4')
    convolution2dLayer(4,1,'Name','conv5')
    sigmoidLayer('Name','sigmoid')];

lgraphD = layerGraph(layersD);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetD = dlnetwork(lgraphD);
```

Define Generator Network

Define the following network architecture, which generates images from 1-by-1-by-100 arrays of random values:



This network:

- Converts the random vectors of size 100 to 7-by-7-by-128 arrays using a *project and reshape* layer.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers and ReLU layers.

Define this network architecture as a layer graph and specify the following network properties.

- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and cropping of the output on each edge.
- For the final transposed convolution layer, specify three 5-by-5 filters corresponding to the three RGB channels of the generated images, and the output size of the previous layer.
- At the end of the network, include a tanh layer.

To project and reshape the noise input, use the custom layer `projectAndReshapeLayer`, attached to this example as a supporting file. The `projectAndReshape` layer upscales the input using a fully connected operation and reshapes the output to the specified size.

```
filterSize = 5;
numFilters = 64;
numLatentInputs = 100;
```

```
projectionSize = [4 4 512];

layersG = [
    featureInputLayer(numLatentInputs,'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'Name','proj');
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','same','Name','tconv2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(filterSize,numFilters,'Stride',2,'Cropping','same','Name','tconv3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(filterSize,3,'Stride',2,'Cropping','same','Name','tconv4')
    tanhLayer('Name','tanh')];

lgraphG = layerGraph(layersG);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetG = dlnetwork(lgraphG);
```

Define Model Gradients Functions

Create the functions `modelGradientsD` and `modelGradientsG` listed in the Model Gradients Function on page 3-0 section of the example, that calculate the gradients of the discriminator and generator loss with respect to the learnable parameters of the discriminator and generator networks, respectively.

The function `modelGradientsD` takes as input the generator and discriminator `dlnetG` and `dlnetD`, a mini-batch of input data `dlX`, an array of random values `dlZ`, and the `lambda` value used for the gradient penalty, and returns the gradients of the loss with respect to the learnable parameters in the discriminator, and the loss.

The function `modelGradientsG` takes as input the generator and discriminator `dlnetG` and `dlnetD`, and an array of random values `dlZ`, and returns the gradients of the loss with respect to the learnable parameters in the generator, and the loss.

Specify Training Options

To train a WGAN-GP model, you must train the discriminator for more iterations than the generator. In other words, for each generator iteration, you must train the discriminator for multiple iterations.

Train with a mini-batch size of 64 for 10,000 generator iterations. For larger datasets, you might need to train for more iterations.

```
miniBatchSize = 64;
numIterationsG = 10000;
```

For each generator iteration, train the discriminator for 5 iterations.

```
numIterationsDPerG = 5;
```

For the WGAN-GP loss, specify a lambda value of 10. The lambda value controls the magnitude of the gradient penalty added to the discriminator loss.

```
lambda = 10;
```

Specify the options for Adam optimization:

- For the discriminator network, specify a learning rate of 0.0002.
- For the generator network, specify a learning rate of 0.001.
- For both networks, specify a gradient decay factor of 0 and a squared gradient decay factor of 0.9.

```
learnRateD = 2e-4;
learnRateG = 1e-3;
gradientDecayFactor = 0;
squaredGradientDecayFactor = 0.9;
```

Display the generated validation images every 20 generator iterations.

```
validationFrequency = 20;
```

Train Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to rescale the images in the range `[-1,1]`.
- Discard any partial mini-batches.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- Train on a GPU if one is available. When the `'OutputEnvironment'` option of `minibatchqueue` is `'auto'`, `minibatchqueue` converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox) (Parallel Computing Toolbox).

The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.

```
augimds.MiniBatchSize = miniBatchSize;
executionEnvironment = "auto";

mbq = minibatchqueue(augimds,...
    'MiniBatchSize',miniBatchSize,...
    'PartialMiniBatch','discard',...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat','SSCB',...
    'OutputEnvironment',executionEnvironment);
```

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display a batch of generated images using a held-out array of random values to input into the generator as well as a plot of the scores.

Initialize the parameters for Adam.

```
trailingAvgD = [];
trailingAvgSqD = [];
trailingAvgG = [];
trailingAvgSqG = [];
```

To monitor training progress, display a batch of generated images using a held-out batch of fixed arrays of random values fed into the generator and plot the network scores.

Create an array of held-out random values.

```
numValidationImages = 25;
ZValidation = randn(numLatentInputs,numValidationImages,'single');
```

Convert the data to `dlarray` objects and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).

```
dlZValidation = dlarray(ZValidation,'CB');
```

For GPU training, convert the data to `gpuArray` objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZValidation = gpuArray(dlZValidation);
end
```

Initialize the training progress plots. Create a figure and resize it to have twice the width.

```
f = figure;
f.Position(3) = 2*f.Position(3);
```

Create a subplot for the generated images and the network scores.

```
imageAxes = subplot(1,2,1);
scoreAxes = subplot(1,2,2);
```

Initialize the animated lines for the loss plot.

```
C = colororder;
lineLossD = animatedline(scoreAxes,'Color',C(1,:));
lineLossDUnregularized = animatedline(scoreAxes,'Color',C(2,:));
legend('With Gradient Penalty','Unregularized')
xlabel("Generator Iteration")
ylabel("Discriminator Loss")
grid on
```

Train the WGAN-GP model by looping over mini-batches of data.

For `numIterationsDPerG` iterations, train the discriminator only. For each mini-batch:

- Evaluate the discriminator model gradients using `dlfeval` and the `modelGradientsD` function.
- Update the discriminator network parameters using the `adamupdate` function.

After training the discriminator for `numIterationsDPerG` iterations, train the generator on a single mini-batch.

- Evaluate the generator model gradients using `dlfeval` and the `modelGradientsG` function.
- Update the generator network parameters using the `adamupdate` function.

After updating the generator network:

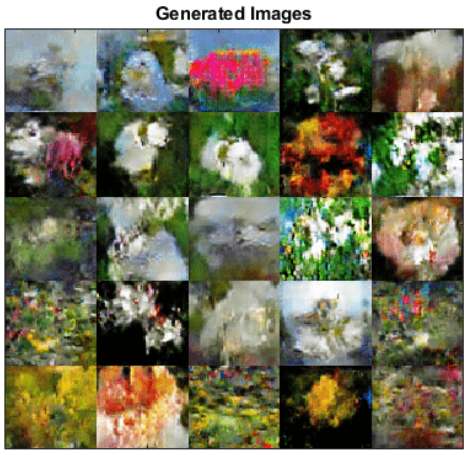
- Plot the losses of the two networks.
- After every `validationFrequency` generator iterations, display a batch of generated images for a fixed held-out generator input.

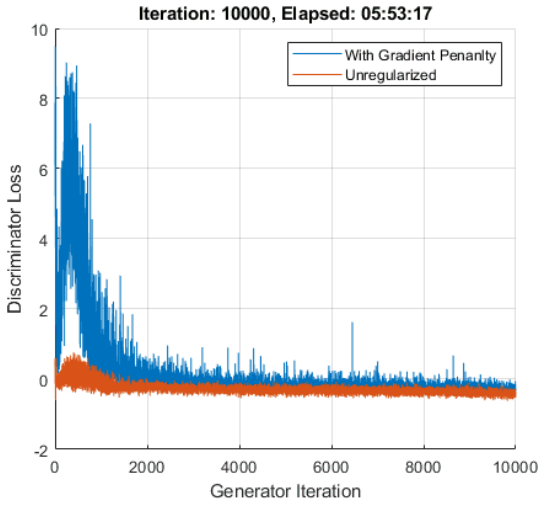
After passing through the data set, shuffle the mini-batch queue.

Training can take some time to run and may require many iterations to output good images.

```
iterationG = 0;
iterationD = 0;
start = tic;

% Loop over mini-batches
while iterationG < numIterationsG
    iterationG = iterationG + 1;

    % Train discriminator only
    for n = 1:numIterationsDPerG
        iterationD = iterationD + 1;

        % Reset and shuffle mini-batch queue when there is no more data.
        if ~hasdata(mbq)
            shuffle(mbq);
        end

        % Read mini-batch of data.
        dlX = next(mbq);

        % Generate latent inputs for the generator network. Convert to
        % dlarray and specify the dimension labels 'CB' (channel, batch).
        Z = randn([numLatentInputs size(dlX,4)],'like',dlX);
        dlZ = dlarray(Z,'CB');

        % Evaluate the discriminator model gradients using dlfeval and the
        % modelGradientsD function listed at the end of the example.
        [gradientsD, lossD, lossDUnregularized] = dlfeval(@modelGradientsD, dlnetD, dlnetG, dlX,

        % Update the discriminator network parameters.
        [dlnetD,trailingAvgD,trailingAvgSqD] = adamupdate(dlnetD, gradientsD, ...
            trailingAvgD, trailingAvgSqD, iterationD, ...
            learnRateD, gradientDecayFactor, squaredGradientDecayFactor);
    end

    % Generate latent inputs for the generator network. Convert to dlarray
    % and specify the dimension labels 'CB' (channel, batch).
    Z = randn([numLatentInputs size(dlX,4)],'like',dlX);
    dlZ = dlarray(Z,'CB');

    % Evaluate the generator model gradients using dlfeval and the
    % modelGradientsG function listed at the end of the example.
    gradientsG = dlfeval(@modelGradientsG, dlnetG, dlnetD, dlZ);

    % Update the generator network parameters.
    [dlnetG,trailingAvgG,trailingAvgSqG] = adamupdate(dlnetG, gradientsG, ...
        trailingAvgG, trailingAvgSqG, iterationG, ...
        learnRateG, gradientDecayFactor, squaredGradientDecayFactor);

    % Every validationFrequency generator iterations, display batch of
    % generated images using the held-out generator input
    if mod(iterationG,validationFrequency) == 0 || iterationG == 1
        % Generate images using the held-out generator input.
        dlXGeneratedValidation = predict(dlnetG,dlZValidation);

        % Tile and rescale the images in the range [0 1].
```

```
        I = imtile(extractdata(dlXGeneratedValidation));
        I = rescale(I);

        % Display the images.
        subplot(1,2,1);
        image(imageAxes,I)
        xticklabels([]);
        yticklabels([]);
        title("Generated Images");
    end

    % Update the scores plot
    subplot(1,2,2)

    lossD = double(gather(extractdata(lossD)));
    lossDUnregularized = double(gather(extractdata(lossDUnregularized)));
    addpoints(lineLossD,iterationG,lossD);
    addpoints(lineLossDUnregularized,iterationG,lossDUnregularized);

    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    title( ...
        "Iteration: " + iterationG + ", " + ...
        "Elapsed: " + string(D))
    drawnow
end
```





Here, the discriminator has learned a strong feature representation that identifies real images among generated images. In turn, the generator has learned a similarly strong feature representation that allows it to generate images similar to the training data.

Generate New Images

To generate new images, use the `predict` function on the generator with a `dlarray` object containing a batch of random vectors. To display the images together, use the `imtile` function and rescale the images using the `rescale` function.

Create a `dlarray` object containing a batch of 25 random vectors to input to the generator network.

```
ZNew = randn(numLatentInputs,25,'single');
dlZNew = dlarray(ZNew,'CB');
```

To generate images using the GPU, also convert the data to `gpuArray` objects.

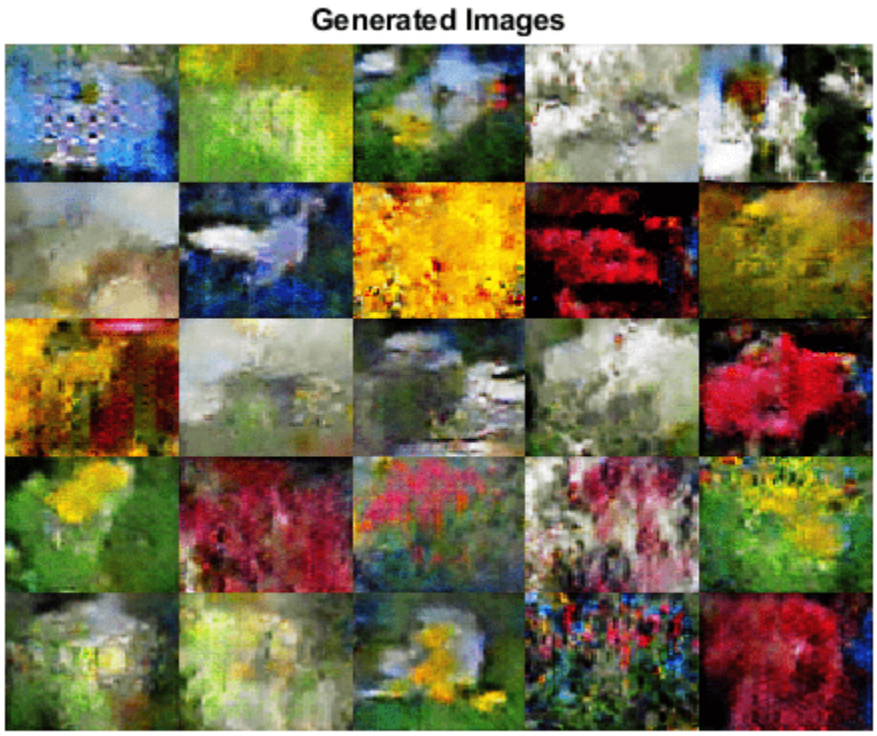
```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
end
```

Generate new images using the `predict` function with the generator and the input data.

```
dlXGeneratedNew = predict(dlnetG,dlZNew);
```

Display the images.

```
I = imtile(extractdata(dlXGeneratedNew));
I = rescale(I);
figure
image(I)
axis off
title("Generated Images")
```



Discriminator Model Gradients Function

The function `modelGradientsD` takes as input the generator and discriminator `dlnetwork` objects `dlnetG` and `dlnetD`, a mini-batch of input data `dlX`, an array of random values `dlZ`, and the `lambda` value used for the gradient penalty, and returns the gradients of the loss with respect to the learnable parameters in the discriminator, and the loss.

Given an image X, a generated image $\widetilde{X}$, define $\widehat{X} = \epsilon X + (1-\epsilon)\widetilde{X}$ for some random $\epsilon \in U(0,1)$.

For the WGAN-GP model, given the lambda value λ, the discriminator loss is given by

$$\text{loss}_D = \widetilde{Y} - Y + \lambda\left(\left\|\nabla_{\widehat{X}}\widehat{Y}\right\|_2 - 1\right)^2,$$

where Y, $\widetilde{Y}$, and $\widehat{Y}$ denote the output of the discriminator for the inputs X, $\widetilde{X}$, and $\widehat{X}$, respectively, and $\nabla_{\widehat{X}}\widehat{Y}$ denotes the gradients of the output $\widehat{Y}$ with respect to $\widehat{X}$. For a mini-batch of data, use a different value of ϵ for each obersvation and calculate the mean loss.

The gradient penalty $\lambda\left(\left\|\nabla_{\widehat{X}}\widehat{Y}\right\|_2 - 1\right)^2$ improves stability by penalizing gradients with large norm values. The lambda value controls the magnitude of the gradient penalty added to the discriminator loss.

```
function [gradientsD, lossD, lossDUnregularized] = modelGradientsD(dlnetD, dlnetG, dlX, dlZ, lamb

% Calculate the predictions for real data with the discriminator network.
dlYPred = forward(dlnetD, dlX);

% Calculate the predictions for generated data with the discriminator
% network.
dlXGenerated = forward(dlnetG,dlZ);
dlYPredGenerated = forward(dlnetD, dlXGenerated);

% Calculate the loss.
lossDUnregularized = mean(dlYPredGenerated - dlYPred);

% Calculate and add the gradient penalty.
epsilon = rand([1 1 1 size(dlX,4)],'like',dlX);
dlXHat = epsilon.*dlX + (1-epsilon).*dlXGenerated;
dlYHat = forward(dlnetD, dlXHat);

% Calculate gradients. To enable computing higher-order derivatives, set
% 'EnableHigherDerivatives' to true.
gradientsHat = dlgradient(sum(dlYHat),dlXHat,'EnableHigherDerivatives',true);
gradientsHatNorm = sqrt(sum(gradientsHat.^2,1:3) + 1e-10);
gradientPenalty = lambda.*mean((gradientsHatNorm - 1).^2);

% Penalize loss.
lossD = lossDUnregularized + gradientPenalty;

% Calculate the gradients of the penalized loss with respect to the
% learnable parameters.
gradientsD = dlgradient(lossD, dlnetD.Learnables);

end
```

Generator Model Gradients Function

The function `modelGradientsG` takes as input the generator and discriminator `dlnetwork` objects `dlnetG` and `dlnetD`, and an array of random values `dlZ`, and returns the gradients of the loss with respect to the learnable parameters in the generator.

Given a generated image $\widetilde{X}$, the loss for the generator network is given by

$$\text{loss}_G = -\widetilde{Y},$$

where $\widetilde{Y}$ denotes the output of the discriminator for the generated image $\widetilde{X}$. For a mini-batch of generated images, calculate the mean loss.

```
function gradientsG =  modelGradientsG(dlnetG, dlnetD, dlZ)

% Calculate the predictions for generated data with the discriminator
% network.
dlXGenerated = forward(dlnetG,dlZ);
dlYPredGenerated = forward(dlnetD, dlXGenerated);

% Calculate the loss.
lossG = -mean(dlYPredGenerated);

% Calculate the gradients of the loss with respect to the learnable
% parameters.
gradientsG = dlgradient(lossG, dlnetG.Learnables);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the input cell array and concatenate into a numeric array.
2. Rescale the images to be in the range `[-1,1]`.

```
function X = preprocessMiniBatch(data)

% Concatenate mini-batch
X = cat(4,data{:});

% Rescale the images in the range [-1 1].
X = rescale(X,-1,1,'InputMin',0,'InputMax',255);

end
```

References

1. The TensorFlow Team. *Flowers* http://download.tensorflow.org/example_images/flower_photos.tgz
2. Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein GAN." *arXiv preprint arXiv:1701.07875* (2017).
3. Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C. Courville. "Improved training of Wasserstein GANs." In *Advances in neural information processing systems*, pp. 5767-5777. 2017.

See Also

`dlnetwork` | `forward` | `predict` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

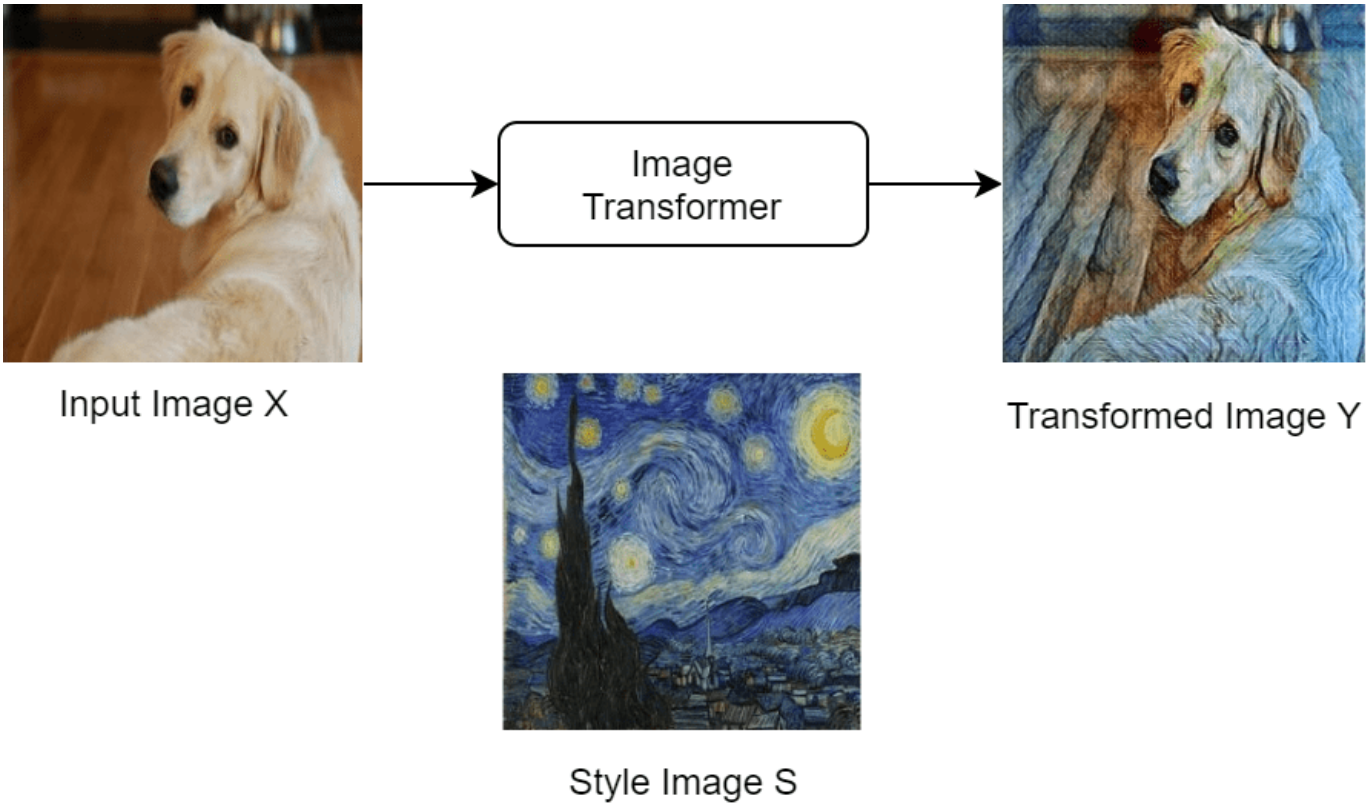
- “Train Generative Adversarial Network (GAN)” on page 3-76

- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182
- “Train Fast Style Transfer Network” on page 3-113
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

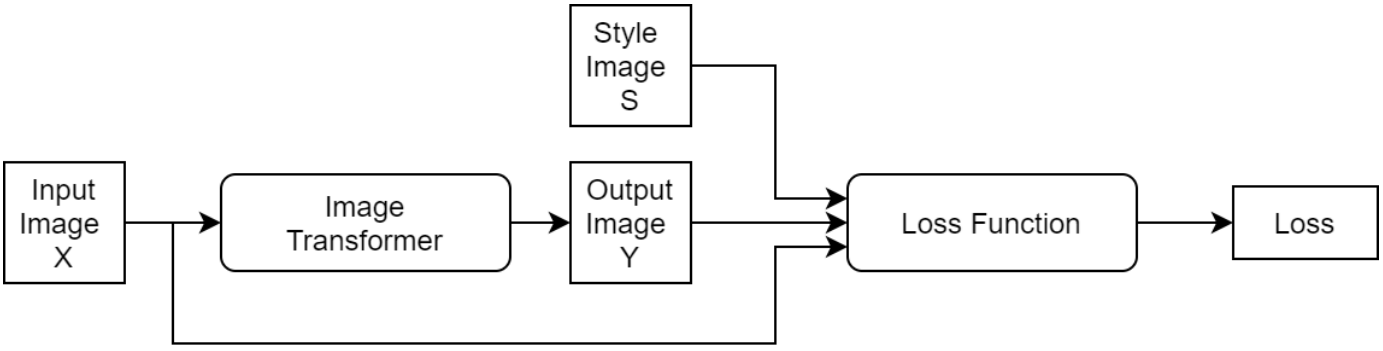
Train Fast Style Transfer Network

This example shows how to train a network to transfer the style of an image to a second image. It is based on the architecture defined in [1].

This example is similar to “Neural Style Transfer Using Deep Learning” on page 9-121, but it works faster once you have trained the network on a style image S. This is because, to obtain the stylized image Y you only need to do a forward pass of the input image X to the network.



Find a high-level diagram of the training algorithm below. This uses three images to calculate the loss: the input image X, the transformed image Y and the style image S.



Note that the loss function uses the pretrained network VGG-16 to extract features from the images. You can find its implementation and mathematical definition in the Style Transfer Loss on page 3-0 section of this example.

Load Training Data

Download and extract the COCO 2014 train images and captions from https://cocodataset.org/#download by clicking the "2014 Train images". Save the data in the folder specified by `imageFolder`. Extract the images into `imageFolder`. The COCO 2014 was collected by the Coco Consortium.

Create directories to store the COCO data set.

```
imageFolder = fullfile(tempdir,"coco");
if ~exist(imageFolder,'dir')
    mkdir(imageFolder);
end
```

Create an image datastore containing the COCO images.

```
imds = imageDatastore(imageFolder,'IncludeSubfolders',true);
```

Training can take a long time to run. If you want to decrease the training time at the cost of accuracy of the resulting network, then select a subset of the image datastore by setting `fraction` to a smaller value.

```
fraction = 1;
numObservations = numel(imds.Files);
imds = subset(imds,1:floor(numObservations*fraction));
```

To resize the images and convert them all to RGB, create an augmented image datastore.

```
augimds = augmentedImageDatastore([256 256],imds,'ColorPreprocessing',"gray2rgb");
```

Read the style image.

```
styleImage = imread('starryNight.jpg');
styleImage = imresize(styleImage,[256 256]);
```

Display the chosen style image.

```
figure
imshow(styleImage)
title("Style Image")
```

Define Image Transformer Network

Define the image transformer network. This is an image-to-image network. The network consists of 3 parts:

1. The first part of the network takes as input an RGB image of size [256x256x3] and downsamples it to a feature map of size [64x64x128].
2. The second part of the network consists of five identical residual blocks defined in the supporting function `residualBlock`.
3. The third and final part of the network upsamples the feature map to the original size of the image and returns the transformed image. This last part uses the `upsampleLayer`, which is a custom layer attached to this example as a supporting file.

```
layers = [

    % First part.
    imageInputLayer([256 256 3], 'Name', 'input', 'Normalization','none')

    convolution2dLayer([9 9], 32, 'Padding','same','Name', 'conv1')
    groupNormalizationLayer('channel-wise','Name','norm1')
    reluLayer('Name', 'relu1')

    convolution2dLayer([3 3], 64, 'Stride', 2,'Padding','same','Name', 'conv2')
    groupNormalizationLayer('channel-wise' ,'Name','norm2')
    reluLayer('Name', 'relu2')

    convolution2dLayer([3 3], 128, 'Stride', 2, 'Padding','same','Name', 'conv3')
    groupNormalizationLayer('channel-wise' ,'Name','norm3')
    reluLayer('Name', 'relu3')
```

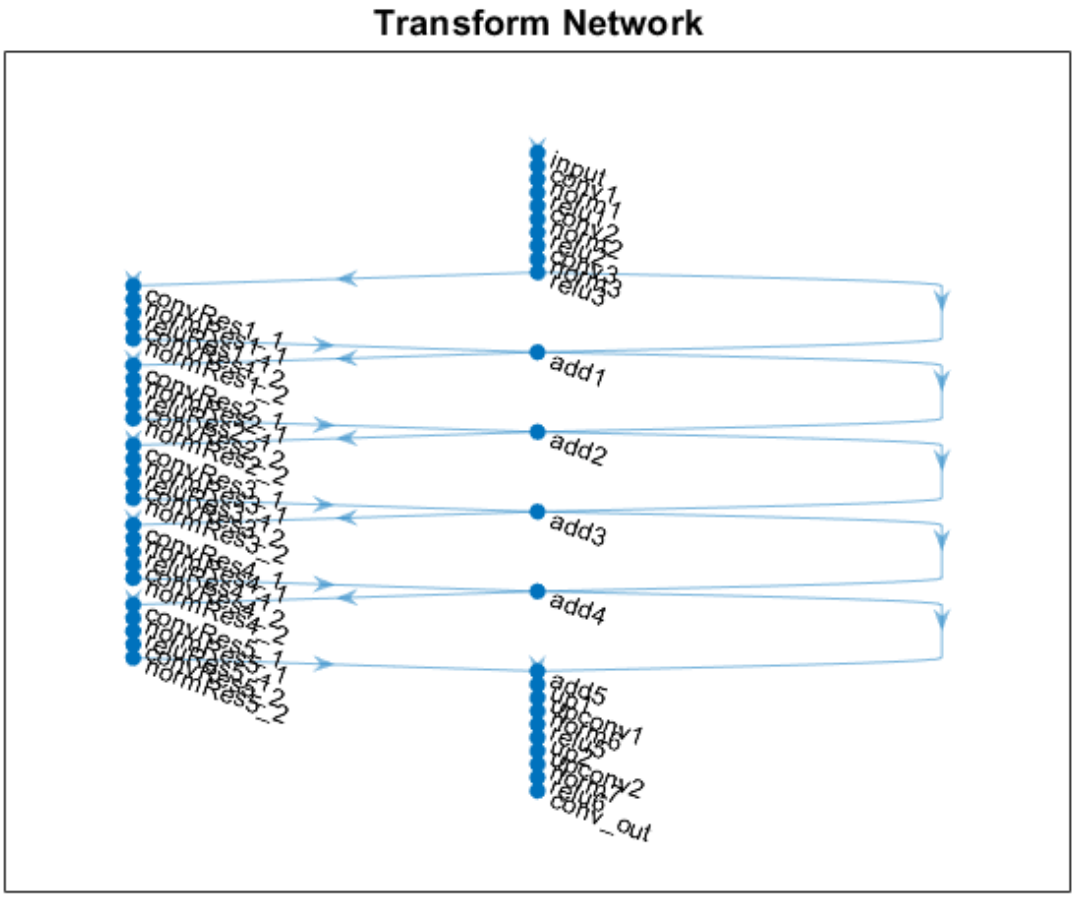
```
    % Second part.
    residualBlock("1")
    residualBlock("2")
    residualBlock("3")
    residualBlock("4")
    residualBlock("5")

    % Third part.
    upsampleLayer('up1')
    convolution2dLayer([3 3], 64, 'Stride', 1, 'Padding','same','Name', 'upconv1')
    groupNormalizationLayer('channel-wise' ,'Name','norm6')
    reluLayer('Name', 'relu5')

    upsampleLayer('up2')
    convolution2dLayer([3 3], 32, 'Stride', 1, 'Padding','same','Name', 'upconv2')
    groupNormalizationLayer('channel-wise' ,'Name','norm7')
    reluLayer('Name', 'relu6')

    convolution2dLayer(9,3,'Padding','same','Name','conv_out')];

lgraph = layerGraph(layers);
```

Add missing connections in residual blocks.

```
lgraph = connectLayers(lgraph,"relu3","add1/in2");
lgraph = connectLayers(lgraph,"add1","add2/in2");
lgraph = connectLayers(lgraph,"add2","add3/in2");
lgraph = connectLayers(lgraph,"add3","add4/in2");
lgraph = connectLayers(lgraph,"add4","add5/in2");
```

Visualize the image transformer network in a plot.

```
figure
plot(lgraph)
title("Transform Network")
```



Create a `dlnetwork` object from the layer graph.

```
dlnetTransform = dlnetwork(lgraph);
```

Style Loss Network

This example uses a pretrained VGG-16 deep neural network to extract the features of the content and style images at different layers. These multilayer features are used to compute respective content and style losses.

To get a pretrained VGG-16 network, use the `vgg16` function. If you do not have the required support packages installed, then the software provides a download link.

```
netLoss = vgg16;
```

To extract the feature necessary to calculate the loss you need the first 24 layers only. Extract and convert to a layer graph.

```
lossLayers = netLoss.Layers(1:24);
lgraph = layerGraph(lossLayers);
```

Convert to a `dlnetwork`.

```
dlnetLoss = dlnetwork(lgraph);
```

Define the Loss Function and Gram Matrix

Create the `styleTransferLoss` function defined in the Style Transfer Loss on page 3-0 section of this example.

The function `styleTransferLoss` takes as input the loss network `dlnetLoss`, a mini-batch of input transformed images `dlX`, a mini-batch of transformed images `dlY`, an array containing the Gram matrices of the style image `dlSGram`, the weight associated with the content loss `contentWeight` and the weight associated with the style loss `styleWeight`. The function returns the total loss `loss` and the individual components: the content loss `lossContent` and the style loss `lossStyle`.

The `styleTransferLoss` function uses the supporting function `createGramMatrix` in the computation of the style loss.

The `createGramMatrix` function takes as an input the features extracted by the loss network and returns a stylistic representation for each image in a mini-batch. You can find the implementation and mathematical definition of the Gram matrix in the section Gram Matrix on page 3-0 .

Define the Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 3-0 section of the example. This function takes as input the loss network `dlnetLoss`, the image transformer network `dlnetTransform`, a mini-batch of input images `dlX`, an array containing the Gram matrices of the style image `dlSGram`, the weight associated with the content loss `contentWeight` and the weight associated with the style loss `styleWeight`. The function returns the `gradients` of the loss with respect to the learnable parameters of the image transformer, the state of the image transformer network, the transformed images `dlY`, the total loss `loss`, the loss associated with the content `lossContent` and the loss associated with the style `lossStyle`.

Specify Training Options

Train with a mini-batch size of 4 for 2 epochs as in [1].

```
numEpochs = 2;
miniBatchSize = 4;
```

Set the read size of the augmented image datastore to the mini-batch size.

```
augimds.MiniBatchSize = miniBatchSize;
```

Specify the options for ADAM optimization. Specify a learn rate of 0.001 with a gradient decay factor of 0.01, and a squared gradient decay factor of 0.999.

```
learnRate = 0.001;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Specify the weight given to the style loss and the one given to the content loss in the calculation of the total loss.

Note that, in order to find a good balance between content and style loss, you might need to experiment with different combinations of weights.

```
weightContent = 1e-4;
weightStyle = 3e-8;
```

Choose the plot frequency of the training progress. This specifies how many iterations there are between each plot update.

```
plotFrequency = 10;
```

Train Model

In order to be able to compute the loss during training, calculate the Gram matrices for the style image.

Convert the style image to `dlarray`.

```
dlS = dlarray(single(styleImage),'SSC');
```

In order to calculate the Gram matrix, feed the style image to the VGG-16 network and extract the activations at four different layers.

```
[dlSActivations1,dlSActivations2,dlSActivations3,dlSActivations4] = forward(dlnetLoss,dlS, ...
    'Outputs',["relu1_2" "relu2_2" "relu3_3" "relu4_3"]);
```

Calculate the Gram matrix for each set of activations using the supporting function `createGramMatrix`.

```
dlSGram{1} = createGramMatrix(dlSActivations1);
dlSGram{2} = createGramMatrix(dlSActivations2);
dlSGram{3} = createGramMatrix(dlSActivations3);
dlSGram{4} = createGramMatrix(dlSActivations4);
```

The training plots consists of two figures:

1. A figure showing a plot of the losses during training
2. A figure containing an input and an output image of the image transformer network

Initialize the training plots. You can check the details of the initialization in the supporting function `initializeFigures.` This function returns: the axis `ax1` where you plot the loss, the axis `ax2` where you plot the validation images, the animated line `lineLossContent` which contains the content loss, the animated line `lineLossStyle` which contains the style loss and the animated line `lineLossTotal` which contains the total loss.

```
[ax1,ax2,lineLossContent,lineLossStyle,lineLossTotal]=initializeStyleTransferPlots();
```

Initialize the average gradient and average squared gradient hyperparameters for the ADAM optimizer.

```
averageGrad = [];
averageSqGrad = [];
```

Calculate total number of training iterations.

```
numIterations = floor(augimds.NumObservations*numEpochs/miniBatchSize);
```

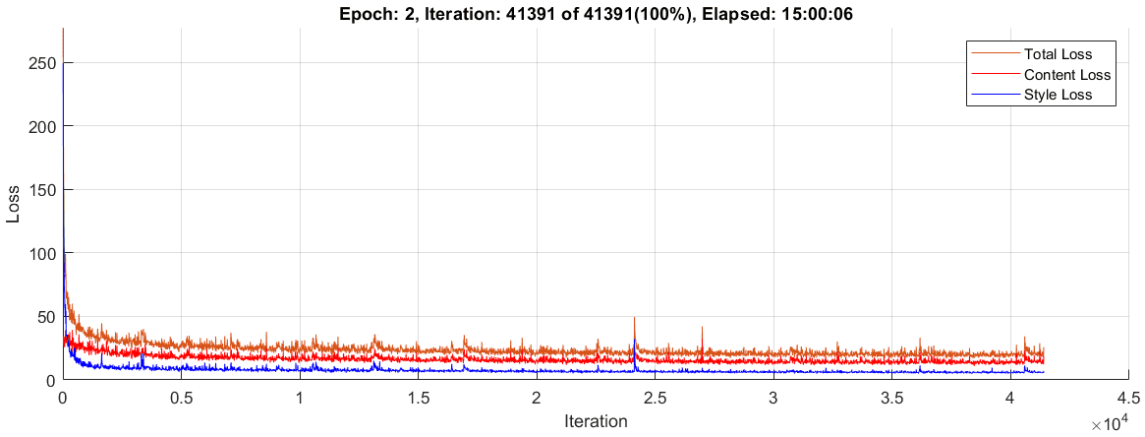
Initialize iteration number and timer before training.

```
iteration = 0;
start = tic;
```

Train the model. This could take a long time to run.

```
% Loop over epochs.
for i = 1:numEpochs

    % Reset and shuffle datastore.
    reset(augimds);
    augimds = shuffle(augimds);

    % Loop over mini-batches.
    while hasdata(augimds)
        iteration = iteration + 1;

        % Read mini-batch of data.
        data = read(augimds);

        % Ignore last partial mini-batch of epoch.
        if size(data,1) < miniBatchSize
            continue
        end

        % Extract the images from data store into a cell array.
        images = data{:,1};

        % Concatenate the images along the 4th dimension.
        X = cat(4,images{:});
        X = single(X);

        % Convert mini-batch of data to dlarray and specify the dimension labels
        % 'SSCB' (spatial, spatial, channel, batch).
        dlX = dlarray(X, 'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients and the network state using
        % dlfeval and the modelGradients function listed at the end of the
        % example.
        [gradients,state,dlY,loss,lossContent,lossStyle] = dlfeval(@modelGradients, ...
            dlnetLoss,dlnetTransform,dlX,dlSGram,weightContent,weightStyle);

        dlnetTransform.State = state;

        % Update the network parameters.
        [dlnetTransform,averageGrad,averageSqGrad] = ...
            adamupdate(dlnetTransform,gradients,averageGrad,averageSqGrad,iteration,...
            learnRate, gradientDecayFactor, squaredGradientDecayFactor);
```

```
        % Every plotFequency iterations, plot the training progress.
        if mod(iteration,plotFrequency) == 0
            addpoints(lineLossTotal,iteration,double(gather(extractdata(loss))))
            addpoints(lineLossContent,iteration,double(gather(extractdata(lossContent))))
            addpoints(lineLossStyle,iteration,double(gather(extractdata(lossStyle))))

            % Use the first image of the mini-batch as a validation image.
            dlV = dlX(:,:,:,1);
            % Use the transformed validation image computed previously.
            dlVY = dlY(:,:,:,1);

            % To use the function imshow, convert to uint8.
            validationImage = uint8(gather(extractdata(dlV)));
            transformedValidationImage = uint8(gather(extractdata(dlVY)));

            % Plot the input image and the output image and increase size
            imshow(imtile({validationImage,transformedValidationImage}),'Parent',ax2);
        end

        % Display time elapsed since start of training and training completion percentage.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        completionPercentage = round(iteration/numIterations*100,2);
        title(ax1,"Epoch: " + i + ", Iteration: " + iteration +" of "+ numIterations + "(" + comp
        drawnow

    end
end
```



Stylize an Image

Once training has finished, you can use the image transformer on any image of your choice.

Load the image you would like to transform.

```
imFilename = 'peppers.png';
im = imread(imFilename);
```

Resize the input image to the input dimensions of the image transformer.

```
im = imresize(im,[256,256]);
```

Convert it to `dlarray`.

```
dlX = dlarray(single(im),'SSCB');
```

To use the GPU convert to `gpuArray` if one is available.

```
if canUseGPU
    dlX = gpuArray(dlX);
end
```

To apply the style to the image, forward pass it to the image transformer using the function `predict.`

```
dlY = predict(dlnetTransform,dlX);
```

Rescale the image into the range [0 255]. First, use the function `tanh` to rescale `dlY` to the range [-1 1]. Then, shift and scale the output to rescale into the [0 255] range.

```
Y = 255*(tanh(dlY)+1)/2;
```

Prepare `Y` for plotting. Use the function `extractdata` to extract the data from `dlarray.` Use the function gather to transfer Y from the GPU to the local workspace.

```
Y = uint8(gather(extractdata(Y)));
```

Show the input image (left) next to the stylized image (right).

```
figure
m = imtile({im,Y});
imshow(m)
```

Model Gradients Function

The function `modelGradients` takes as input the loss network `dlnetLoss`, the image transformer network `dlnetTransform`, a mini-batch of input images `dlX`, an array containing the Gram matrices of the style image `dlSGram`, the weight associated with the content loss `contentWeight` and the weight associated with the style loss `styleWeight`. It returns the `gradients` of the loss with respect to the learnable parameters of the image transformer, the state of the image transformer network, the transformed images `dlY`, the total loss `loss`, the loss associated with the content `lossContent` and the loss associated with the style `lossStyle.`

```
function [gradients,state,dlY,loss,lossContent,lossStyle] = ...
    modelGradients(dlnetLoss,dlnetTransform,dlX,dlSGram,contentWeight,styleWeight)

[dlY,state] = forward(dlnetTransform,dlX);

dlY = 255*(tanh(dlY)+1)/2;

[loss,lossContent,lossStyle] = styleTransferLoss(dlnetLoss,dlY,dlX,dlSGram,contentWeight,styleWe

gradients = dlgradient(loss,dlnetTransform.Learnables);

end
```

Style Transfer Loss

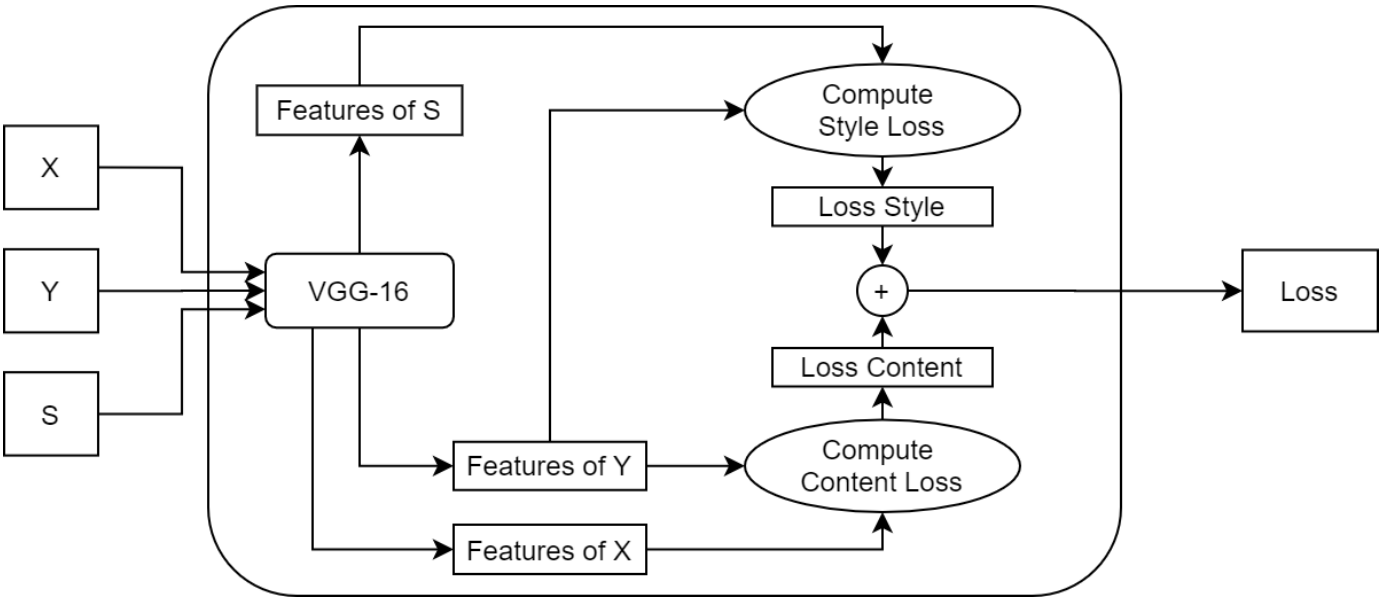
The function `styleTransferLoss` takes as input the loss network `dlnetLoss`, a mini-batch of input images `dlX`, a mini-batch of transformed images `dlY`, an array containing the Gram matrices of the style image `dlSGram`, the weights associated with the content and style `contentWeight` and `styleWeight`, respectively. It returns the total loss `loss` and the individual components: the content loss `lossContent` and the style loss `lossStyle`.

The content loss is a measure of how much difference in spatial structure there is between the input image X and the output images Y.

On the other hand, the style loss tells you how much difference in the stylistic appearance there is between the style image S and the output image Y.

The graph below explains the algorithm that `styleTransferLoss` implements to calculate the total loss.

First, the function passes the input images X, the transformed images Y and the style image S to the pretrained network VGG-16. This pretrained network extracts several features from these images. The algorithm then calculates the content loss by using the spatial features of the input image X and of the output image Y. Moreover, it calculates the style loss by using the stylistic features of the output image Y and of the style image S. Finally, it obtains the total loss by adding the content and style losses.



Content Loss

For each image in the mini-batch, the content loss function compares the features of the original image and of the transformed image output by the layer `relu_3_3`. In particular, it calculates the mean square error between the activations and returns the average loss for the mini-batch:

$$\text{lossContent} = \frac{1}{N}\sum_{n=1}^{N} \text{mean}([\phi(X_n) - \phi(Y_n)]^2),$$

where X contains the input images, Y contains the transformed images, N is the mini-batch size, and $\phi()$ represents the activations extracted at layer `relu_3_3`.

Style Loss

To calculate the style loss, for each single image in the mini-batch:

1. Extract the activations at the layers `relu1_2`, `relu2_2`, `relu3_3` and `relu4_3`.
2. For each of the four activations ϕ_j compute the Gram matrix $G(\phi_j)$.
3. Calculate the squared difference between the corresponding Gram matrices.
4. Add up the four outputs for each layer j from the previous step.

To obtain the style loss for the whole mini-batch, compute the average of the style loss for each image n in the mini-batch:

$$\text{lossStyle} = \frac{1}{N}\sum_{n=1}^{N}\sum_{j=1}^{4} [G(\phi_j(X_n)) - G(\phi_j(S))]^2,$$

where j is the index of the layer, and $G()$ is the Gram Matrix.

Total Loss

```
function [loss,lossContent,lossStyle] = styleTransferLoss(dlnetLoss,dlY,dlX, ...
    dlSGram,weightContent,weightStyle)
```

```
% Extract activations.
dlYActivations = cell(1,4);
[dlYActivations{1},dlYActivations{2},dlYActivations{3},dlYActivations{4}] = ...
    forward(dlnetLoss,dlY,'Outputs',["relu1_2" "relu2_2" "relu3_3" "relu4_3"]);

dlXActivations = forward(dlnetLoss,dlX,'Outputs','relu3_3');

% Calculate the mean square error between activations.
lossContent = mean((dlYActivations{3} - dlXActivations).^2,'all');

% Add up the losses for all the four activations.
lossStyle = 0;
for j = 1:4
    G = createGramMatrix(dlYActivations{j});
    lossStyle = lossStyle + sum((G - dlSGram{j}).^2,'all');
end

% Average the loss over the mini-batch.
miniBatchSize = size(dlX,4);
lossStyle = lossStyle/miniBatchSize;

% Apply weights.
lossContent = weightContent * lossContent;
lossStyle = weightStyle * lossStyle;

% Calculate the total loss.
loss = lossContent + lossStyle;

end
```

Residual Block

The `residualBlock` function returns an array of six layers. It consists of convolution layers, instance normalization layers, a ReLu layer and an addition layer. Note that `groupNormalizationLayer('channel-wise')` is simply an instance normalization layer.

```
function layers = residualBlock(name)

layers = [
    convolution2dLayer([3 3], 128, 'Stride', 1,'Padding','same','Name', "convRes"+name+"_1")
    groupNormalizationLayer('channel-wise','Name',"normRes"+name+"_1")
    reluLayer('Name', "reluRes"+name+"_1")
    convolution2dLayer([3 3], 128, 'Stride', 1,'Padding','same', 'Name', "convRes"+name+"_2")
    groupNormalizationLayer('channel-wise','Name',"normRes"+name+"_2")
    additionLayer(2,'Name',"add"+name)];

end
```

Gram Matrix

The function `createGramMatrix` takes as an input the activations of a single layer and returns a stylistic representation for each image in a mini-batch. The input is a feature map of size [H, W, C, N], where H is the height, W is the width, C is the number of channels and N is the mini-batch size. The function outputs an array `G` of size [C,C,N]. Each subarray `G(:,:,k)` is the Gram matrix corresponding to the k^{th} image in the mini-batch. Each entry $G(i, j, k)$ of the Gram matrix represents

the correlation between channels c_i and c_j, because each entry in channel c_i multiplies the entry in the corresponding position in channel c_j:

$$G(i,j,k) = \frac{1}{C \times H \times W} \sum_{h=1}^{H} \sum_{w=1}^{W} \phi_k(h,w,c_i)\phi_k(h,w,c_j),$$

where ϕ_k are the activations for the k^{th} image in the mini-batch.

The Gram matrix contains information about which features activate together but has no information about where the features occur in the image. This is because the summation over height and width loses the information about the spatial structure. The loss function uses this matrix as a stylistic representation of the image.

```
function G = createGramMatrix(activations)

[h,w,numChannels] = size(activations,1:3);

features = reshape(activations,h*w,numChannels,[]);
featuresT = permute(features,[2 1 3]);

G = dlmtimes(featuresT,features) / (h*w*numChannels);

end
```

References

1 Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. "Perceptual losses for real-time style transfer and super-resolution." *European conference on computer vision*. Springer, Cham, 2016.

See Also

`dlnetwork` | `forward` | `predict` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Train a Siamese Network to Compare Images

This example shows how to train a Siamese network to identify similar images of handwritten characters.

A Siamese network is a type of deep learning network that uses two or more identical subnetworks that have the same architecture and share the same parameters and weights. Siamese networks are typically used in tasks that involve finding the relationship between two comparable things. Some common applications for Siamese networks include facial recognition, signature verification [1], or paraphrase identification [2]. Siamese networks perform well in these tasks because their shared weights mean there are fewer parameters to learn during training and they can produce good results with a relatively small amount of training data.

Siamese networks are particularly useful in cases where there are large numbers of classes with small numbers of observations of each. In such cases, there is not enough data to train a deep convolutional neural network to classify images into these classes. Instead, the Siamese network can determine if two images are in the same class.

This example use the Omniglot dataset [3] to train a Siamese network to compare images of handwritten characters [4]. The Omniglot dataset contains character sets for 50 alphabets, divided into 30 used for training and 20 for testing. Each alphabet contains a number of characters from 14 for Ojibwe (Canadia Aboriginal Sullabics) to 55 for Tifinagh. Finally, each character has 20 handwritten observations. This example trains a network to identify whether two handwritten observations are different instances of the same character.

You can also use Siamese networks to identify similar images using dimensionality reduction. For an example, see “Train a Siamese Network for Dimensionality Reduction” on page 3-142.

Load and Preprocess Training Data

Download and extract the Omniglot training dataset.

```
url = "https://github.com/brendenlake/omniglot/raw/master/python/images_background.zip";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"images_background.zip");

dataFolderTrain = fullfile(downloadFolder,'images_background');
if ~exist(dataFolderTrain,"dir")
    disp("Downloading Omniglot training data (4.5 MB)...")
    websave(filename,url);
    unzip(filename,downloadFolder);
end
disp("Training data downloaded.")
```

```
Training data downloaded.
```

Load the training data as a image datastore using the `imageDatastore` function. Specify the labels manually by extracting the labels from the file names and setting the `Labels` property.

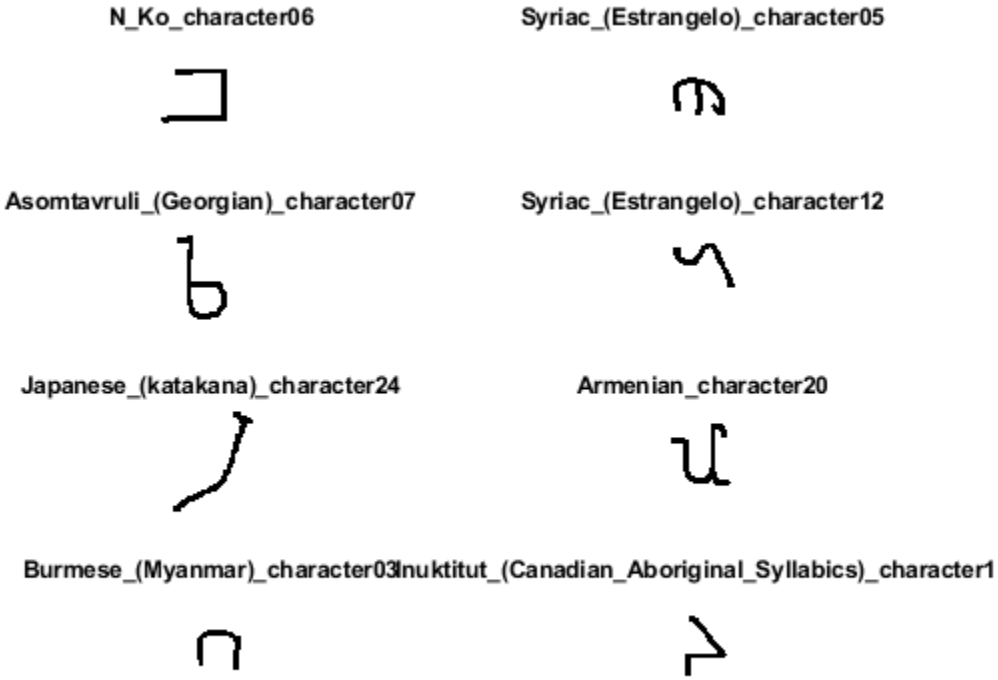
```
imdsTrain = imageDatastore(dataFolderTrain, ...
    'IncludeSubfolders',true, ...
    'LabelSource','none');

files = imdsTrain.Files;
parts = split(files,filesep);
```

```
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTrain.Labels = categorical(labels);
```

The Omniglot training dataset consists of black and white handwritten characters from 30 alphabets, with 20 observations of each character. The images are of size 105-by-105-by-1, and the values of each pixel are between `0` and `1`.

Display a random selection of the images.

```
idxs = randperm(numel(imdsTrain.Files),8);

for i = 1:numel(idxs)
    subplot(4,2,i)
    imshow(readimage(imdsTrain,idxs(i)))
    title(imdsTrain.Labels(idxs(i)), "Interpreter","none");
end
```



Create Pairs of Similar and Dissimilar Images

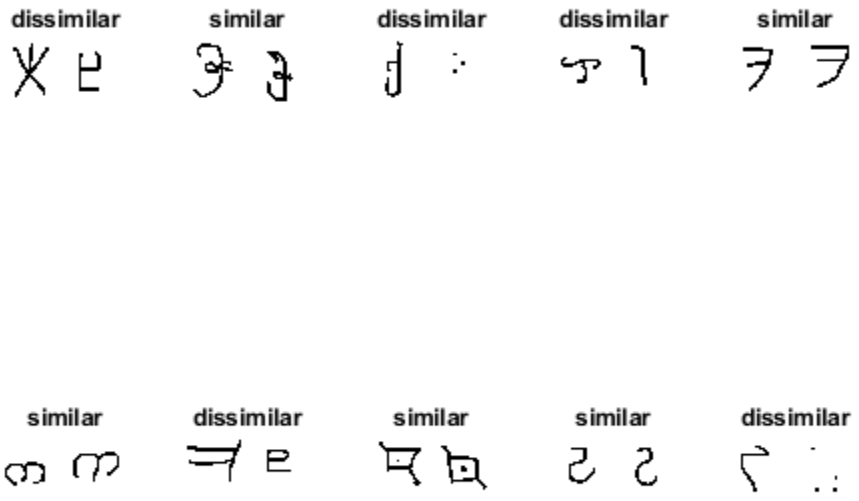
To train the network, the data must be grouped into pairs of images that are either similar or dissimilar. Here, similar images are different handwritten instances of the same character, which have the same label, while dissimilar images of different characters have different labels. The function `getSiameseBatch` (defined in the Supporting Functions on page 3-0 section of this example) creates randomized pairs of similar or dissimilar images, `pairImage1` and `pairImage2`. The function also returns the label `pairLabel`, which identifies if the pair of images is similar or dissimilar to each other. Similar pairs of images have `pairLabel = 1`, while dissimilar pairs have `pairLabel = 0`.

As an example, create a small representative set of five pairs of images

```
batchSize = 10;
[pairImage1,pairImage2,pairLabel] = getSiameseBatch(imdsTrain,batchSize);
```

Display the generated pairs of images.

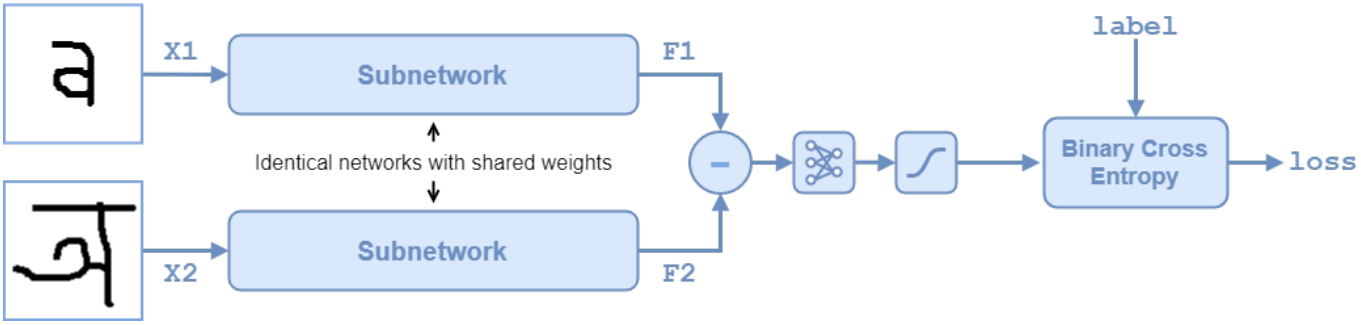
```
for i = 1:batchSize
    if pairLabel(i) == 1
        s = "similar";
    else
        s = "dissimilar";
    end
    subplot(2,5,i)
    imshow([pairImage1(:,:,:,i) pairImage2(:,:,:,i)]);
    title(s)
end
```



In this example, a new batch of 180 paired images is created for each iteration of the training loop. This ensures that the network is trained on a large number of random pairs of images with approximately equal proportions of similar and dissimilar pairs.

Define Network Architecture

The Siamese network architecture is illustrated in the following diagram.



To compare two images, each image is passed through one of two identical subnetworks that share weights. The subnetworks convert each 105-by-105-by-1 image to a 4096-dimensional feature vector. Images of the same class have similar 4096-dimensional representations. The output feature vectors from each subnetwork are combined through subtraction and the result is passed through a `fullyconnect` operation with a single output. A `sigmoid` operation converts this value to a probability between `0` and `1`, indicating the network's prediction of whether the images are similar or dissimilar. The binary cross-entropy loss between the network prediction and the true label is used to update the network during training.

In this example, the two identical subnetworks are defined as a `dlnetwork` object. The final `fullyconnect` and `sigmoid` operations are performed as functional operations on the subnetwork outputs.

Create the subnetwork as a series of layers that accepts 105-by-105-by-1 images and outputs a feature vector of size 4096.

For the `convolution2dLayer` objects, use the narrow normal distribution to initialize the weights and bias.

For the `maxPooling2dLayer` objects, set the stride to 2.

For the final `fullyConnectedLayer` object, specify an output size of 4096 and use the narrow normal distribution to initialize the weights and bias.

```
layers = [
    imageInputLayer([105 105 1],'Name','input1','Normalization','none')
    convolution2dLayer(10,64,'Name','conv1','WeightsInitializer','narrow-normal','BiasInitialize
    reluLayer('Name','relu1')
    maxPooling2dLayer(2,'Stride',2,'Name','maxpool1')
    convolution2dLayer(7,128,'Name','conv2','WeightsInitializer','narrow-normal','BiasInitialize
    reluLayer('Name','relu2')
    maxPooling2dLayer(2,'Stride',2,'Name','maxpool2')
    convolution2dLayer(4,128,'Name','conv3','WeightsInitializer','narrow-normal','BiasInitialize
    reluLayer('Name','relu3')
    maxPooling2dLayer(2,'Stride',2,'Name','maxpool3')
    convolution2dLayer(5,256,'Name','conv4','WeightsInitializer','narrow-normal','BiasInitialize
    reluLayer('Name','relu4')
    fullyConnectedLayer(4096,'Name','fc1','WeightsInitializer','narrow-normal','BiasInitializer'
```

```
lgraph = layerGraph(layers);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnet = dlnetwork(lgraph);
```

Create the weights for the final `fullyconnect` operation. Initialize the weights by sampling a random selection from a narrow normal distribution with standard deviation of 0.01.

```
fcWeights = dlarray(0.01*randn(1,4096));
fcBias = dlarray(0.01*randn(1,1));

fcParams = struct(...
    "FcWeights",fcWeights,...
    "FcBias",fcBias);
```

To use the network, create the function `forwardSiamese` (defined in the Supporting Functions on page 3-0 section of this example) that defines how the two subnetworks and the subtraction, `fullyconnect`, and `sigmoid` operations are combined. The function `forwardSiamese` accepts the network, the structure containing the parameters for the `fullyconnect` operation, and two training images. The `forwardSiamese` function outputs a prediction about the similarity of the two images.

Define Model Gradients Function

Create the function `modelGradients` (defined in the Supporting Functions on page 3-0 section of this example). The `modelGradients` function takes the Siamese subnetwork `dlnet`, the parameter structure for the `fullyconnect` operation, and a mini-batch of input data `X1` and `X2` with their labels `pairLabels`. The function returns the loss values and the gradients of the loss with respect to the learnable parameters of the network.

The objective of the Siamese network is to discriminate between the two inputs `X1` and `X2`. The output of the network is a probability between `0` and `1`, where a value closer to `0` indicates a prediction that the images are dissimilar, and a value closer to `1` that the images are similar. The loss is given by the binary cross-entropy between the predicted score and the true label value:

$$\text{loss} = -t\log(y) - (1-t)\log(1-y),$$

where the true label t can be 0 or 1 and y is the predicted label.

Specify Training Options

Specify the options to use during training. Train for 10000 iterations.

```
numIterations = 10000;
miniBatchSize = 180;
```

Specify the options for ADAM optimization:

- Set the learning rate to `0.00006`.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]` for both `dlnet` and `fcParams`.
- Set the gradient decay factor to `0.9` and the squared gradient decay factor to `0.99`.

```
learningRate = 6e-5;
trailingAvgSubnet = [];
trailingAvgSqSubnet = [];
trailingAvgParams = [];
trailingAvgSqParams = [];
gradDecay = 0.9;
gradDecaySq = 0.99;
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel

Computing Toolbox). To automatically detect if you have a GPU available and place the relevant data on the GPU, set the value of `executionEnvironment` to `"auto"`. If you don't have a GPU, or don't want to use one for training, set the value of `executionEnvironment` to `"cpu"`. To ensure you use a GPU for training, set the value of `executionEnvironment` to `"gpu"`.

```
executionEnvironment = "auto";
```

To monitor the training progress, you can plot the training loss after each iteration. Create the variable `plots` that contains `"training-progress"`. If you don't want to plot the training progress, set this value to `"none"`.

```
plots = "training-progress";
```

Initialize the plot parameters for the training loss progress plot.

```
plotRatio = 16/9;

if plots == "training-progress"
    trainingPlot = figure;
    trainingPlot.Position(3) = plotRatio*trainingPlot.Position(4);
    trainingPlot.Visible = 'on';

    trainingPlotAxes = gca;

    lineLossTrain = animatedline(trainingPlotAxes);
    xlabel(trainingPlotAxes,"Iteration")
    ylabel(trainingPlotAxes,"Loss")
    title(trainingPlotAxes,"Loss During Training")
end
```

Train Model

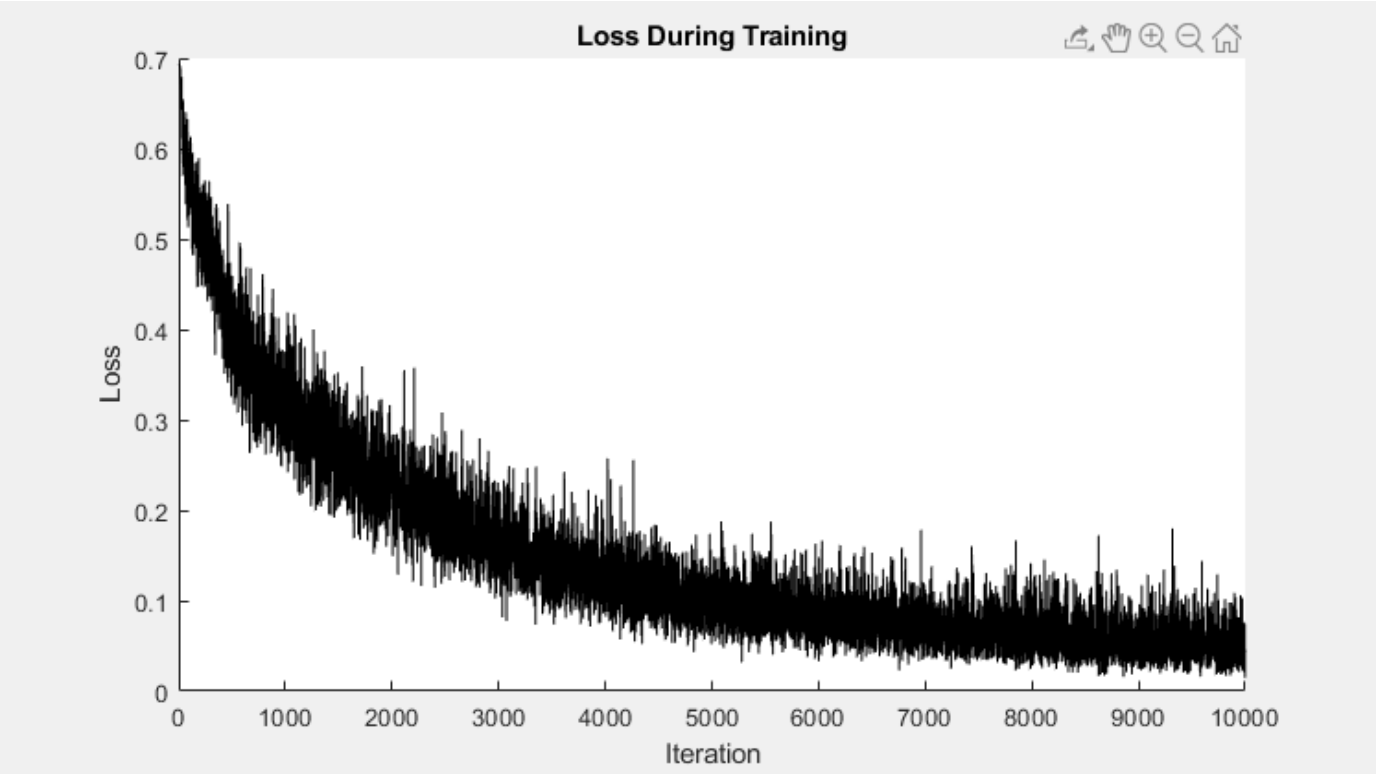
Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

For each iteration:

- Extract a batch of image pairs and labels using the `getSiameseBatch` function defined in the section Create Batches of Image Pairs on page 3-0 .
- Convert the data to `dlarray` objects with underlying type `single` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch) for the image data and `'CB'` (channel, batch) for the labels.
- For GPU training, convert the data to `gpuArray` objects.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

```
% Loop over mini-batches.
for iteration = 1:numIterations

    % Extract mini-batch of image pairs and pair labels
    [X1,X2,pairLabels] = getSiameseBatch(imdsTrain,miniBatchSize);

    % Convert mini-batch of data to dlarray. Specify the dimension labels
    % 'SSCB' (spatial, spatial, channel, batch) for image data
    dlX1 = dlarray(single(X1),'SSCB');
    dlX2 = dlarray(single(X2),'SSCB');
```

```
        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX1 = gpuArray(dlX1);
            dlX2 = gpuArray(dlX2);
        end

        % Evaluate the model gradients and the generator state using
        % dlfeval and the modelGradients function listed at the end of the
        % example.
        [gradientsSubnet, gradientsParams,loss] = dlfeval(@modelGradients,dlnet,fcParams,dlX1,dlX2,pa
        lossValue = double(gather(extractdata(loss)));

        % Update the Siamese subnetwork parameters.
        [dlnet,trailingAvgSubnet,trailingAvgSqSubnet] = ...
            adamupdate(dlnet,gradientsSubnet, ...
            trailingAvgSubnet,trailingAvgSqSubnet,iteration,learningRate,gradDecay,gradDecaySq);

        % Update the fullyconnect parameters.
        [fcParams,trailingAvgParams,trailingAvgSqParams] = ...
            adamupdate(fcParams,gradientsParams, ...
            trailingAvgParams,trailingAvgSqParams,iteration,learningRate,gradDecay,gradDecaySq);

        % Update the training loss progress plot.
        if plots == "training-progress"
            addpoints(lineLossTrain,iteration,lossValue);
        end
        drawnow;
    end
```



Evaluate the Accuracy of the Network

Download and extract the Omniglot test dataset.

```
url = 'https://github.com/brendenlake/omniglot/raw/master/python/images_evaluation.zip';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'images_evaluation.zip');

dataFolderTest = fullfile(downloadFolder,'images_evaluation');
if ~exist(dataFolderTest,'dir')
    disp('Downloading Omniglot test data (3.2 MB)...')
    websave(filename,url);
    unzip(filename,downloadFolder);
end
disp("Test data downloaded.")
```

```
Test data downloaded.
```

Load the test data as a image datastore using the `imageDatastore` function. Specify the labels manually by extracting the labels from the file names and setting the `Labels` property.

```
imdsTest = imageDatastore(dataFolderTest, ...
    'IncludeSubfolders',true, ...
    'LabelSource','none');

files = imdsTest.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTest.Labels = categorical(labels);
```

The test dataset contains 20 alphabets that are different to those that the network was trained on. In total, there 659 different classes in the test dataset.

```
numClasses = numel(unique(imdsTest.Labels))
```

```
numClasses = 659
```

To calculate the accuracy of the network, create a set of five random mini-batches of test pairs. Use the `predictSiamese` function (defined in the Supporting Functions on page 3-0 section of this example) to evaluate the network predictions and calculate the average accuracy over the mini-batches.

```
accuracy = zeros(1,5);
accuracyBatchSize = 150;

for i = 1:5

    % Extract mini-batch of image pairs and pair labels
    [XAcc1,XAcc2,pairLabelsAcc] = getSiameseBatch(imdsTest,accuracyBatchSize);

    % Convert mini-batch of data to dlarray. Specify the dimension labels
    % 'SSCB' (spatial, spatial, channel, batch) for image data.
    dlXAcc1 = dlarray(single(XAcc1),'SSCB');
    dlXAcc2 = dlarray(single(XAcc2),'SSCB');

    % If using a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
       dlXAcc1 = gpuArray(dlXAcc1);
```

```
        dlXAcc2 = gpuArray(dlXAcc2);
    end

    % Evaluate predictions using trained network
    dlY = predictSiamese(dlnet,fcParams,dlXAcc1,dlXAcc2);

    % Convert predictions to binary 0 or 1
    Y = gather(extractdata(dlY));
    Y = round(Y);

    % Compute average accuracy for the minibatch
    accuracy(i) = sum(Y == pairLabelsAcc)/accuracyBatchSize;
end

% Compute accuracy over all minibatches
averageAccuracy = mean(accuracy)*100
```

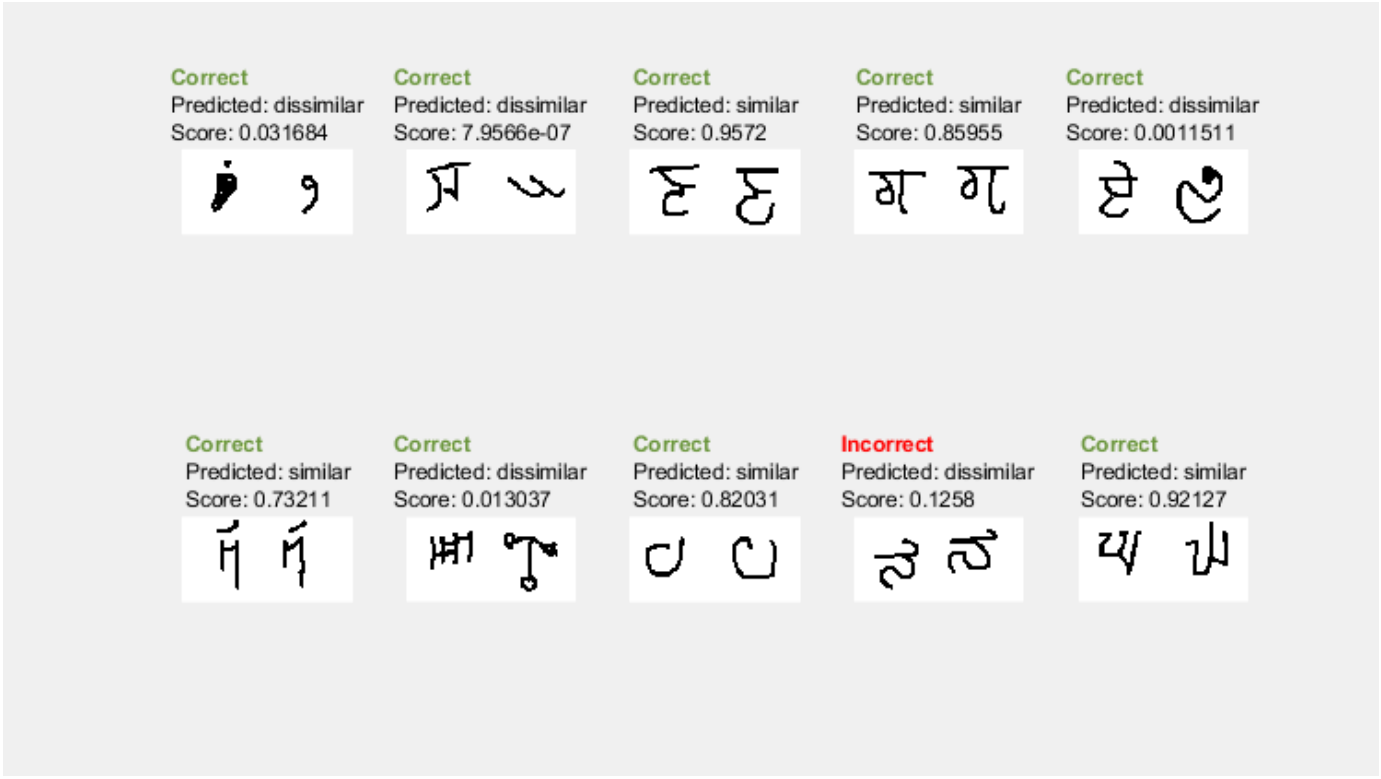
```
averageAccuracy = 88.6667
```

Display a Test Set of Images with Predictions

To visually check if the network correctly identifies similar and dissimilar pairs, create a small batch of image pairs to test. Use the predictSiamese function to get the prediction for each test pair. Display the pair of images with the prediction, the probability score, and a label indicating whether the prediction was correct or incorrect.

```
testBatchSize = 10;

[XTest1,XTest2,pairLabelsTest] = getSiameseBatch(imdsTest,testBatchSize);

% Convert test batch of data to dlarray. Specify the dimension labels
% 'SSCB' (spatial, spatial, channel, batch) for image data and 'CB'
% (channel, batch) for labels
dlXTest1 = dlarray(single(XTest1),'SSCB');
dlXTest2 = dlarray(single(XTest2),'SSCB');

% If using a GPU, then convert data to gpuArray
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
   dlXTest1 = gpuArray(dlXTest1);
   dlXTest2 = gpuArray(dlXTest2);
end

% Calculate the predicted probability
dlYScore = predictSiamese(dlnet,fcParams,dlXTest1,dlXTest2);
YScore = gather(extractdata(dlYScore));

% Convert predictions to binary 0 or 1
YPred = round(YScore);

% Extract data to plot
XTest1 = extractdata(dlXTest1);
XTest2 = extractdata(dlXTest2);

% Plot images with predicted label and predicted score
testingPlot = figure;
testingPlot.Position(3) = plotRatio*testingPlot.Position(4);
testingPlot.Visible = 'on';
```

```
for i = 1:numel(pairLabelsTest)

    if YPred(i) == 1
        predLabel = "similar";
    else
        predLabel = "dissimilar" ;
    end

    if pairLabelsTest(i) == YPred(i)
        testStr = "\bf\color{darkgreen}Correct\rm\newline";

    else
        testStr = "\bf\color{red}Incorrect\rm\newline";
    end

    subplot(2,5,i)
    imshow([XTest1(:,:,:,i) XTest2(:,:,:,i)]);

    title(testStr + "\color{black}Predicted: " + predLabel + "\newlineScore: " + YScore(i));
end
```



The network is able to compare the test images to determine their similarity, even though none of these images were in the training dataset.

Supporting Functions

Model Functions for Training and Prediction

The function `forwardSiamese` is used during network training. The function defines how the subnetworks and the `fullyconnect` and `sigmoid` operations combine to form the complete

Siamese network. forwardSiamese accepts the network structure and two training images and outputs a prediction about the similarity of the two images. Within this example, the function `forwardSiamese` is introduced in the section Define Network Architecture on page 3-0 .

```
function Y = forwardSiamese(dlnet,fcParams,dlX1,dlX2)
% forwardSiamese accepts the network and pair of training images, and returns a
% prediction of the probability of the pair being similar (closer to 1) or
% dissimilar (closer to 0). Use forwardSiamese during training.

    % Pass the first image through the twin subnetwork
    F1 = forward(dlnet,dlX1);
    F1 = sigmoid(F1);

    % Pass the second image through the twin subnetwork
    F2 = forward(dlnet,dlX2);
    F2 = sigmoid(F2);

    % Subtract the feature vectors
    Y = abs(F1 - F2);

    % Pass the result through a fullyconnect operation
    Y = fullyconnect(Y,fcParams.FcWeights,fcParams.FcBias);

    % Convert to probability between 0 and 1.
    Y = sigmoid(Y);
end
```

The function `predictSiamese` uses the trained network to make predictions about the similarity of two images. The function is similar to the function `forwardSiamese`, defined previously. However, `predictSiamese` uses the `predict` function with the network instead of the `forward` function, because some deep learning layers behave differently during training and prediction. Within this example, the function `predictSiamese` is introduced in the section Evaluate the Accuracy of the Network on page 3-0 .

```
function Y = predictSiamese(dlnet,fcParams,dlX1,dlX2)
% predictSiamese accepts the network and pair of images, and returns a
% prediction of the probability of the pair being similar (closer to 1)
% or dissimilar (closer to 0). Use predictSiamese during prediction.

    % Pass the first image through the twin subnetwork
    F1 = predict(dlnet,dlX1);
    F1 = sigmoid(F1);

    % Pass the second image through the twin subnetwork
    F2 = predict(dlnet,dlX2);
    F2 = sigmoid(F2);

    % Subtract the feature vectors
    Y = abs(F1 - F2);

    % Pass result through a fullyconnect operation
    Y = fullyconnect(Y,fcParams.FcWeights,fcParams.FcBias);

    % Convert to probability between 0 and 1.
    Y = sigmoid(Y);
end
```

Model Gradients Function

The function `modelGradients` takes the Siamese `dlnetwork` object `net`, a pair of mini-batch input data `X1` and `X2`, and the label indicating whether they are similar or dissimilar. The function returns the gradients of the loss with respect to the learnable parameters in the network and the binary cross-entropy loss between the prediction and the ground truth. Within this example, the function `modelGradients` is introduced in the section Define Model Gradients Function on page 3-0 .

```
function [gradientsSubnet,gradientsParams,loss] = modelGradients(dlnet,fcParams,dlX1,dlX2,pairLa
% The modelGradients function calculates the binary cross-entropy loss between the
% paired images and returns the loss and the gradients of the loss with respect to
% the network learnable parameters

    % Pass the image pair through the network
    Y = forwardSiamese(dlnet,fcParams,dlX1,dlX2);

    % Calculate binary cross-entropy loss
    loss = binarycrossentropy(Y,pairLabels);

    % Calculate gradients of the loss with respect to the network learnable
    % parameters
    [gradientsSubnet,gradientsParams] = dlgradient(loss,dlnet.Learnables,fcParams);
end

function loss = binarycrossentropy(Y,pairLabels)
    % binarycrossentropy accepts the network's prediction Y, the true
    % label, and pairLabels, and returns the binary cross-entropy loss value.

    % Get precision of prediction to prevent errors due to floating
    % point precision
    precision = underlyingType(Y);

    % Convert values less than floating point precision to eps.
    Y(Y < eps(precision)) = eps(precision);
    %convert values between 1-eps and 1 to 1-eps.
    Y(Y > 1 - eps(precision)) = 1 - eps(precision);

    % Calculate binary cross-entropy loss for each pair
    loss = -pairLabels.*log(Y) - (1 - pairLabels).*log(1 - Y);

    % Sum over all pairs in minibatch and normalize.
    loss = sum(loss)/numel(pairLabels);
end
```

Create Batches of Image Pairs

The following functions create randomized pairs of images that are similar or dissimilar, based on their labels. Within this example, the function `getSiameseBatch` is introduced in the section Create Pairs of Similar and Dissimilar Images. on page 3-0

```
function [X1,X2,pairLabels] = getSiameseBatch(imds,miniBatchSize)
% getSiameseBatch returns a randomly selected batch or paired images. On
% average, this function produces a balanced set of similar and dissimilar
% pairs.

    pairLabels = zeros(1,miniBatchSize);
    imgSize = size(readimage(imds,1));
    X1 = zeros([imgSize 1 miniBatchSize]);
```

```
        X2 = zeros([imgSize 1 miniBatchSize]);

        for i = 1:miniBatchSize
            choice = rand(1);
            if choice < 0.5
                [pairIdx1,pairIdx2,pairLabels(i)] = getSimilarPair(imds.Labels);
            else
                [pairIdx1,pairIdx2,pairLabels(i)] = getDissimilarPair(imds.Labels);
            end
            X1(:,:,:,i) = imds.readimage(pairIdx1);
            X2(:,:,:,i) = imds.readimage(pairIdx2);
        end
    end

    function [pairIdx1,pairIdx2,pairLabel] = getSimilarPair(classLabel)
    % getSimilarSiamesePair returns a random pair of indices for images
    % that are in the same class and the similar pair label = 1.

        % Find all unique classes.
        classes = unique(classLabel);

        % Choose a class randomly which will be used to get a similar pair.
        classChoice = randi(numel(classes));

        % Find the indices of all the observations from the chosen class.
        idxs = find(classLabel==classes(classChoice));

        % Randomly choose two different images from the chosen class.
        pairIdxChoice = randperm(numel(idxs),2);
        pairIdx1 = idxs(pairIdxChoice(1));
        pairIdx2 = idxs(pairIdxChoice(2));
        pairLabel = 1;
    end

    function  [pairIdx1,pairIdx2,label] = getDissimilarPair(classLabel)
    % getDissimilarSiamesePair returns a random pair of indices for images
    % that are in different classes and the dissimilar pair label = 0.

        % Find all unique classes.
        classes = unique(classLabel);

        % Choose two different classes randomly which will be used to get a dissimilar pair.
        classesChoice = randperm(numel(classes),2);

        % Find the indices of all the observations from the first and second classes.
        idxs1 = find(classLabel==classes(classesChoice(1)));
        idxs2 = find(classLabel==classes(classesChoice(2)));

        % Randomly choose one image from each class.
        pairIdx1Choice = randi(numel(idxs1));
        pairIdx2Choice = randi(numel(idxs2));
        pairIdx1 = idxs1(pairIdx1Choice);
        pairIdx2 = idxs2(pairIdx2Choice);
        label = 0;
    end
```

References

[1] Bromley, J., I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. "*Signature Verification using a "Siamese" Time Delay Neural Network*." In Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS 1993), 1994, pp737-744. Available at Signature Verification using a "Siamese" Time Delay Neural Network on the NIPS Proceedings website.

[2] Wenpeg, Y., and H Schütze. "*Convolutional Neural Network for Paraphrase Identification.*" In Proceedings of 2015 Conference of the North American Chapter of the ACL, 2015, pp901-911. Available at Convolutional Neural Network for Paraphrase Identification on the ACL Anthology website

[3] Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. "*Human-level concept learning through probabilistic program induction*." Science, 350(6266), (2015) pp1332-1338.

[4] Koch, G., Zemel, R., and Salakhutdinov, R. (2015). "*Siamese neural networks for one-shot image recognition*". In Proceedings of the 32nd International Conference on Machine Learning, 37 (2015). Available at Siamese Neural Networks for One-shot Image Recognition on the ICML'15 website.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `adamupdate`

More About

- “Train a Siamese Network for Dimensionality Reduction” on page 3-142
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Functions with dlarray Support” on page 18-423

Train a Siamese Network for Dimensionality Reduction

This example shows how to train a Siamese network to compare handwritten digits using dimensionality reduction.

A Siamese network is a type of deep learning network that uses two or more identical subnetworks that have the same architecture and share the same parameters and weights. Siamese networks are typically used in tasks that involve finding the relationship between two comparable things. Some common applications for Siamese networks include facial recognition, signature verification [1], or paraphrase identification [2]. Siamese networks perform well in these tasks because their shared weights mean there are fewer parameters to learn during training and they can produce good results with a relatively small amount of training data.

Siamese networks are particularly useful in cases where there are large numbers of classes with small numbers of observations of each. In such cases, there is not enough data to train a deep convolutional neural network to classify images into these classes. Instead, the Siamese network can determine if two images are in the same class. The network does this by reducing the dimensionality of the training data and using a distance-based cost function to differentiate between the classes.

This example uses a Siamese network for dimensionality reduction of a collection of images of handwritten digits. The Siamese architecture reduces the dimensionality by mapping images with the same class to nearby points in a low-dimensional space. The reduced-feature representation is then used to extract images from the dataset that are most similar to a test image. The training data in this example are images of size 28-by-28-by-1, giving an initial feature dimensionality of 784. The Siamese network reduces the dimensionality of the input images to two features and is trained to output similar reduced features for images with the same label.

You can also use Siamese networks to identify similar images by directly comparing them. For an example, see “Train a Siamese Network to Compare Images” on page 3-128.

Load and Preprocess Training Data

Load the training data, which consists of images of handwritten digits. The function `digitTrain4DArrayData` loads the digit images and their labels.

```
[XTrain,YTrain] = digitTrain4DArrayData;
```

`XTrain` is a 28-by-28-by-1-by-5000 array containing 5000 single-channel images, each of size 28-by-28. The values of each pixel are between `0` and `1`. `YTrain` is a categorical vector containing the labels for each observation, which are the numbers from 0 to 9 corresponding to the value of the written digit.

Display a random selection of the images.

```
perm = randperm(numel(YTrain), 9);
imshow(imtile(XTrain(:,:,:,perm),"ThumbnailSize",[100 100]));
```

Create Pairs of Similar and Dissimilar Images

To train the network, the data must be grouped into pairs of images that are either similar or dissimilar. Here, similar images are defined as having the same label, while dissimilar images have different labels. The function `getSiameseBatch` (defined in the Supporting Functions on page 3-0 section of this example) creates randomized pairs of similar or dissimilar images, `pairImage1` and `pairImage2`. The function also returns the label `pairLabel`, which identifies if the pair of images is similar or dissimilar to each other. Similar pairs of images have `pairLabel = 1`, while dissimilar pairs have `pairLabel = 0`.

As an example, create a small representative set of five pairs of images

```
batchSize = 10;
[pairImage1,pairImage2,pairLabel] = getSiameseBatch(XTrain,YTrain,batchSize);
```

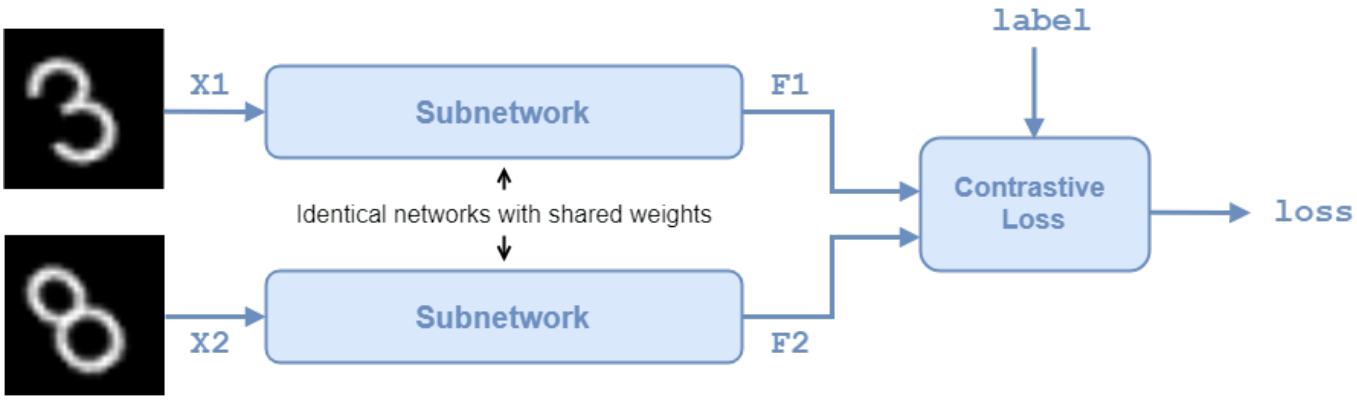
Display the generated pairs of images.

```
for i = 1:batchSize
subplot(2,5,i)
imshow([pairImage1(:,:,:,i) pairImage2(:,:,:,i)]);
if pairLabel(i) == 1
    s = "similar";
else
    s = "dissimilar";
end
title(s)
end
```

In this example, a new batch of 180 paired images is created for each iteration of the training loop. This ensures that the network is trained on a large number of random pairs of images with approximately equal proportions of similar and dissimilar pairs.

Define Network Architecture

The Siamese network architecture is illustrated in the following diagram.



In this example, the two identical subnetworks are defined as a series of fully connected layers with ReLU layers. Create a network that accepts 28-by-28-by-1 images and outputs the two feature vectors used for the reduced feature representation. The network reduces the dimensionality of the input images to two, a value that is easier to plot and visualize than the initial dimensionality of 784.

For the first two fully connected layers, specify an output size of 1024 and use the He weight initializer.

For the final fully connected layer, specify an output size of two and use the He weights initializer.

```
layers = [
    imageInputLayer([28 28],'Name','input1','Normalization','none')
    fullyConnectedLayer(1024,'Name','fc1','WeightsInitializer','he')
    reluLayer('Name','relu1')
    fullyConnectedLayer(1024,'Name','fc2','WeightsInitializer','he')
    reluLayer('Name','relu2')
    fullyConnectedLayer(2,'Name','fc3','WeightsInitializer','he')];

lgraph = layerGraph(layers);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnet = dlnetwork(lgraph);
```

Define Model Gradients Function

Create the function `modelGradients` (defined in the Supporting Functions on page 3-0 section of this example). The `modelGradients` function takes the Siamese `dlnetwork` object `dlnet` and a mini-batch of input data `dlX1` and `dlX2` with their labels `pairLabels`. The function returns the loss values and the gradients of the loss with respect to the learnable parameters of the network.

The objective of the Siamese network is to output a feature vector for each image such that the feature vectors are similar for similar images, and notably different for dissimilar images. In this way, the network can discriminate between the two inputs.

Find the contrastive loss between the outputs from the last fully connected layer, the feature vectors `features1` and `features1` from `pairImage1` and `pairImage2`, respectively. The contrastive loss for a pair is given by [3]

$$\text{loss} = \frac{1}{2} y d^2 + \frac{1}{2}\left(1 - y\right)\max(margin - d, 0)^2,$$

where y is the value of the pair label ($y = 1$ for similar images;$y = 0$ for dissimilar images), and d is the Euclidean distance between two features vectors $f1$ and $f2$: $d = \|f1 - f2\|_2$.

The *margin* parameter is used for constraint: if two images in a pair are dissimilar, then their distance should be at least *margin*, or a loss will be incurred.

The contrastive loss has two terms, but only one is ever non-zero for a given image pair. In the case of similar images, the first term can be non-zero and is minimized by reducing the distance between the image features $f1$ and $f2$. In the case of dissimilar images, the second term can be non-zero, and is minimized by increasing the distance between the image features, to at least a distance of *margin*. The smaller the value of *margin*, the less constraining it is over how close a dissimilar pair can be before a loss is incurred.

Specify Training Options

Specify the value of *margin* to use during training.

```
margin = 0.3;
```

Specify the options to use during training. Train for 3000 iterations.

```
numIterations = 3000;
miniBatchSize = 180;
```

Specify the options for ADAM optimization:

- Set the learning rate to `0.0001`.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.
- Set the gradient decay factor to `0.9` and the squared gradient decay factor to `0.99`.

```
learningRate = 1e-4;
trailingAvg = [];
trailingAvgSq = [];
gradDecay = 0.9;
gradDecaySq = 0.99;
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). To automatically detect if you have a GPU available and place the relevant data on the GPU, set the value of `executionEnvironment` to `"auto"`. If you don't have a GPU, or don't want to use one for training, set the value of `executionEnvironment` to `"cpu"`. To ensure you use a GPU for training, set the value of `executionEnvironment` to `"gpu"`.

```
executionEnvironment = "auto";
```

To monitor the training progress, you can plot the training loss after each iteration. Create the variable `plots` that contains `"training-progress"`. If you don't want to plot the training progress, set this value to `"none"`.

```
plots = "training-progress";
```

Initialize the plot parameters for the training loss progress plot.

```
plotRatio = 16/9;

if plots == "training-progress"
    trainingPlot = figure;
    trainingPlot.Position(3) = plotRatio*trainingPlot.Position(4);
    trainingPlot.Visible = 'on';

    trainingPlotAxes = gca;

    lineLossTrain = animatedline(trainingPlotAxes);
    xlabel(trainingPlotAxes,"Iteration")
    ylabel(trainingPlotAxes,"Loss")
    title(trainingPlotAxes,"Loss During Training")
end
```

To evaluate how well the network is doing at dimensionality reduction, compute and plot the reduced features of a set of test data after each iteration. Load the test data, which consists of images of handwritten digits similar to the training data. Convert the test data to `dlarray` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch). If you are using a GPU, convert the test data to `gpuArray`.

```
[XTest,YTest] = digitTest4DArrayData;
dlXTest = dlarray(single(XTest),'SSCB');
```

```
% If training on a GPU, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

Initialize the plot parameters for the reduced-feature plot of the test data.

```
dimensionPlot = figure;
dimensionPlot.Position(3) = plotRatio*dimensionPlot.Position(4);
dimensionPlot.Visible = 'on';

dimensionPlotAxes = gca;

uniqueGroups = unique(YTest);
colors = hsv(length(uniqueGroups));
```

Initialize a counter to keep track of the total number of iterations.

```
iteration = 1;
```

Train Model

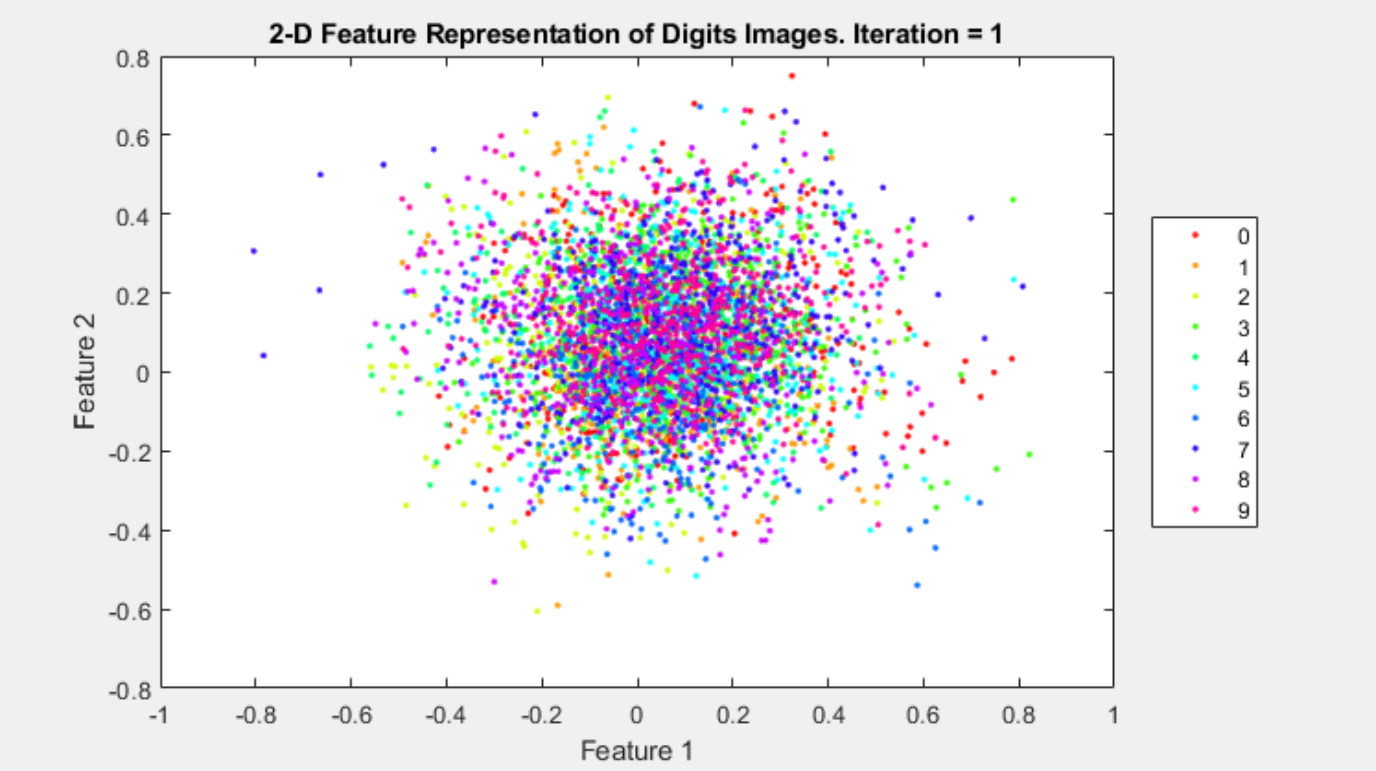
Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

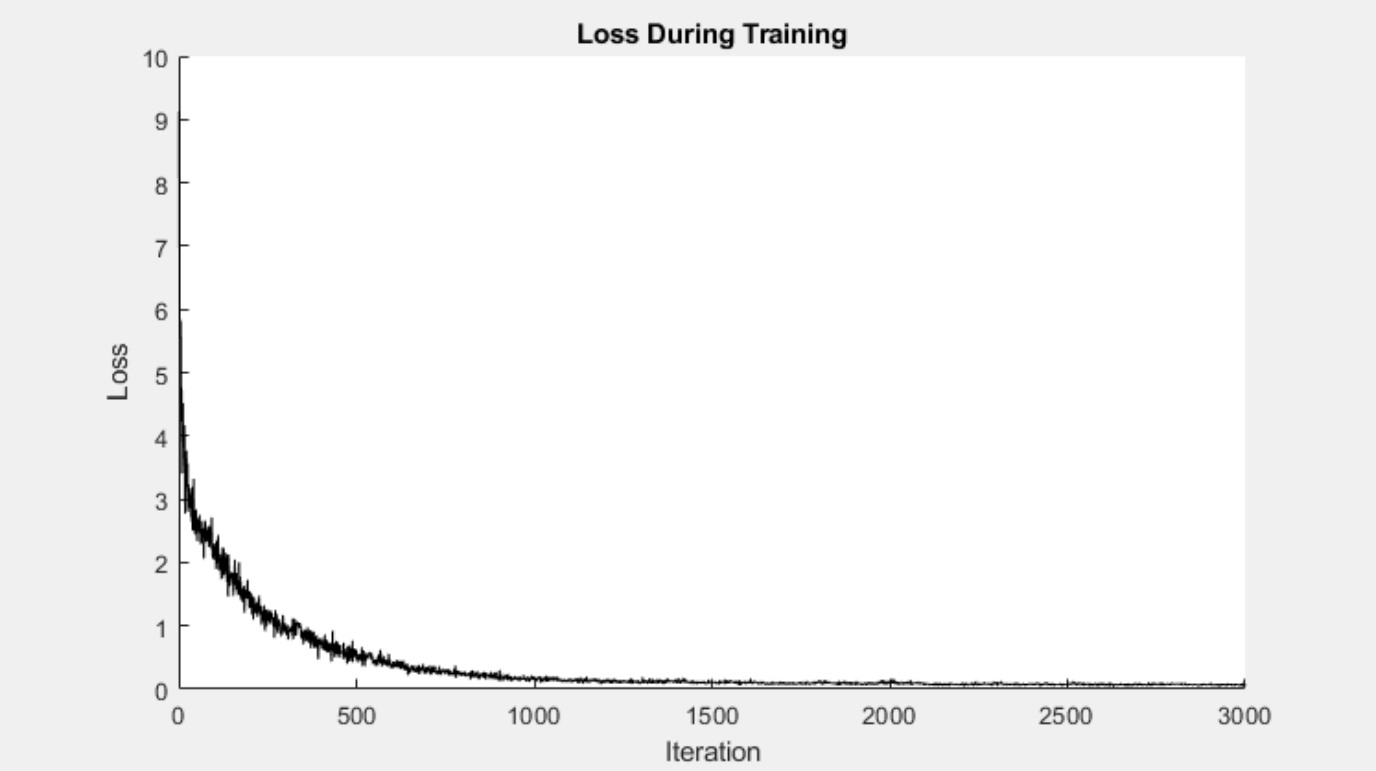
For each iteration:

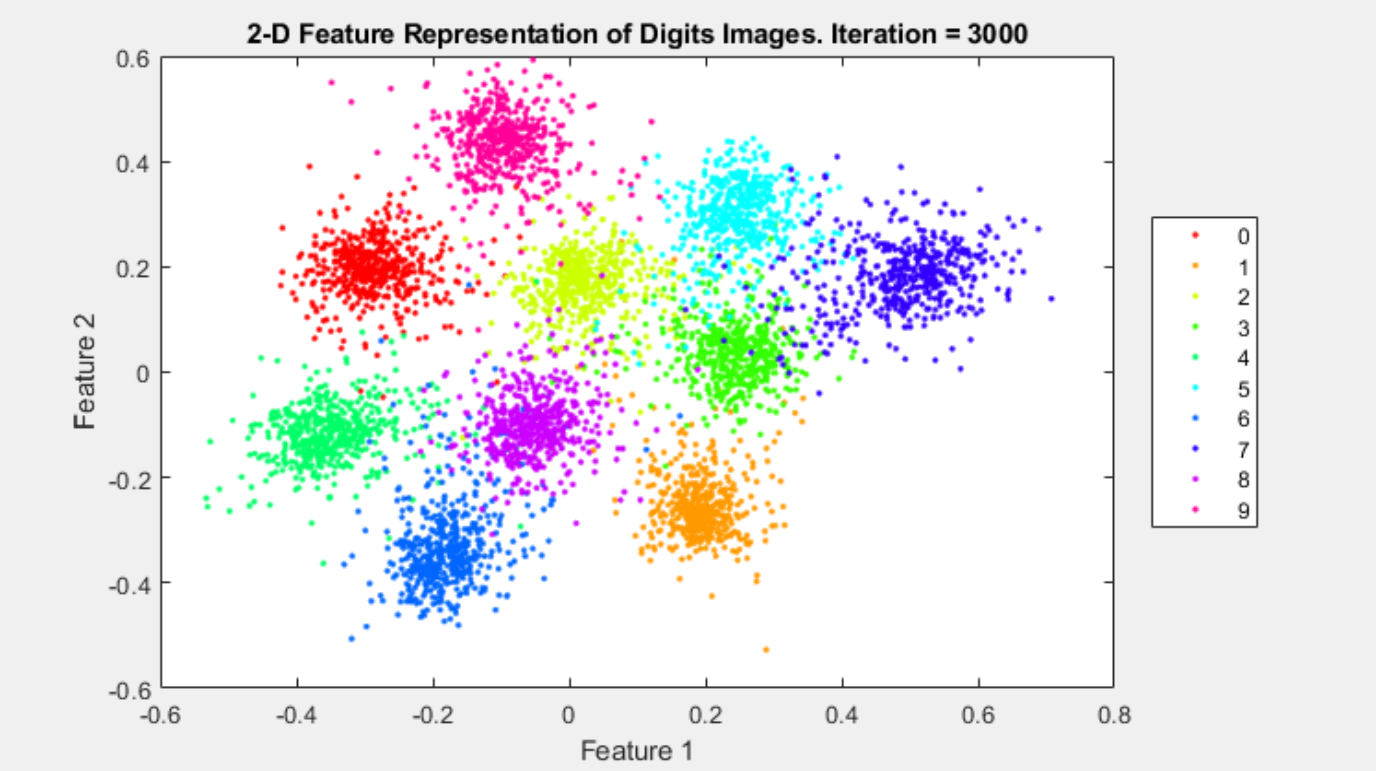
- Extract a batch of image pairs and labels using the `getSiameseBatch` function defined in the section Create Batches of Image Pairs on page 3-0 .
- Convert the image data to `dlarray` objects with underlying type `single` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the image data to `gpuArray` objects.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

```
% Loop over mini-batches.
for iteration = 1:numIterations

    % Extract mini-batch of image pairs and pair labels
    [X1,X2,pairLabels] = getSiameseBatch(XTrain,YTrain,miniBatchSize);

    % Convert mini-batch of data to dlarray. Specify the dimension labels
    % 'SSCB' (spatial, spatial, channel, batch) for image data
    dlX1 = dlarray(single(X1),'SSCB');
    dlX2 = dlarray(single(X2),'SSCB');

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dlX1 = gpuArray(dlX1);
        dlX2 = gpuArray(dlX2);
    end

    % Evaluate the model gradients and the generator state using
    % dlfeval and the modelGradients function listed at the end of the
    % example.
```

```
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX1,dlX2,pairLabels,margin);
        lossValue = double(gather(extractdata(loss)));

        % Update the Siamese network parameters.
        [dlnet.Learnables,trailingAvg,trailingAvgSq] = ...
            adamupdate(dlnet.Learnables,gradients, ...
            trailingAvg,trailingAvgSq,iteration,learningRate,gradDecay,gradDecaySq);

        % Update the training loss progress plot.
        if plots == "training-progress"
            addpoints(lineLossTrain,iteration,lossValue);
        end

        % Update the reduced-feature plot of the test data.
        % Compute reduced features of the test data:
        dlFTest = predict(dlnet,dlXTest);
        FTest = extractdata(dlFTest);

        figure(dimensionPlot);
        for k = 1:length(uniqueGroups)
            % Get indices of each image in test data with the same numeric
            % label (defined by the unique group):
            ind = YTest==uniqueGroups(k);
            % Plot this group:
            plot(dimensionPlotAxes,gather(FTest(1,ind)'),gather(FTest(2,ind)'),'.','color',...
                colors(k,:));
            hold on
        end

        legend(uniqueGroups)

        % Update title of reduced-feature plot with training progress information.
        title(dimensionPlotAxes,"2-D Feature Representation of Digits Images. Iteration = " +...
            iteration);
        legend(dimensionPlotAxes,'Location','eastoutside');
        xlabel(dimensionPlotAxes,"Feature 1")
        ylabel(dimensionPlotAxes,"Feature 2")

        hold off
        drawnow
    end
```


2-D Feature Representation of Digits Images. Iteration = 1
Feature 2
Feature 1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
0
1
2
3
4
5
6
7
8
9


Loss During Training
Loss
Iteration
10
9
8
7
6
5
4
3
2
1
0
0
500
1000
1500
2000
2500
3000



The network has now learned to represent each image as a 2-D vector. As you can see from the reduced-feature plot of the test data, images of similar digits are clustered close to each other in this 2-D representation.

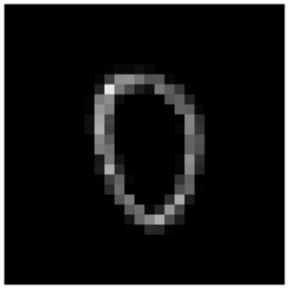
Use the Trained Network to Find Similar Images

You can use the trained network to find a selection of images that are similar to each other out of a group. In this case, use the test data as the group of images. Convert the group of images to `dlarray` objects and `gpuArray` objects, if you are using a GPU.

```
groupX = XTest;

dlGroupX = dlarray(single(groupX),'SSCB');

if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlGroupX = gpuArray(dlGroupX);
end
```

Extract a single test image from the group and display it. Remove the test image from the group so that it does not appear in the set of similar images.

```
testIdx = randi(5000);
testImg = dlGroupX(:,:,:,testIdx);

trialImgDisp = extractdata(testImg);

figure
imshow(trialImgDisp, 'InitialMagnification', 500);
```



```
dlGroupX(:,:,:,testIdx) = [];
```

Find the reduced features of the test image using `predict`.

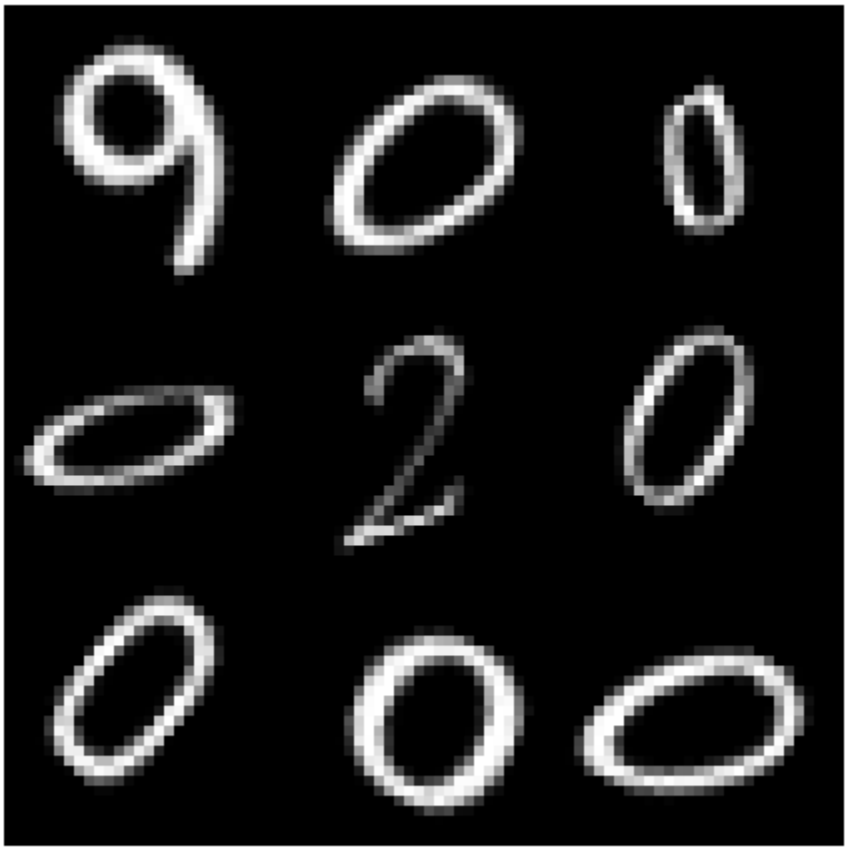
```
trialF = predict(dlnet,testImg);
```

Find the 2-D reduced feature representation of each of the images in the group using the trained network.

```
FGroupX = predict(dlnet,dlGroupX);
```

Use the reduced feature representation to find the nine images in the group that are closest to the test image, using the Euclidean distance metric. Display the images.

```
distances = vecnorm(extractdata(trialF - FGroupX));
[~,idx] = sort(distances);
sortedImages = groupX(:,:,:,idx);

figure
imshow(imtile(sortedImages(:,:,:,1:9)), 'InitialMagnification', 500);
```



By reducing the images to a lower dimensionality, the network is able to identify images that are similar to the trial image. The reduced feature representation allows the network to discriminate between images that are similar and dissimilar. Siamese networks are often used in the context of facial or signature recognition. For example, you can train a Siamese network to accept an image of a face as an input, and return a set of the most similar faces from a database.

Supporting Functions

Model Gradients Function

The function `modelGradients` takes the Siamese `dlnetwork` object `dlnet`, a pair of mini-batch input data `X1` and `X2`, and the label `pairLabels`. The function returns the gradients of the loss with respect to the learnable parameters in the network as well as the contrastive loss between the reduced dimensionality features of the paired images. Within this example, the function `modelGradients` is introduced in the section Define Model Gradients Function on page 3-0 .

```
function [gradients, loss] = modelGradients(net,X1,X2,pairLabel,margin)
% The modelGradients function calculates the contrastive loss between the
% paired images and returns the loss and the gradients of the loss with
% respect to the network learnable parameters
```

```
    % Pass first half of image pairs forward through the network
    F1 = forward(net,X1);
    % Pass second set of image pairs forward through the network
    F2 = forward(net,X2);

    % Calculate contrastive loss
    loss = contrastiveLoss(F1,F2,pairLabel,margin);

    % Calculate gradients of the loss with respect to the network learnable
    % parameters
    gradients = dlgradient(loss, net.Learnables);

end

function loss = contrastiveLoss(F1,F2,pairLabel,margin)
% The contrastiveLoss function calculates the contrastive loss between
% the reduced features of the paired images

    % Define small value to prevent taking square root of 0
    delta = 1e-6;

    % Find Euclidean distance metric
    distances = sqrt(sum((F1 - F2).^2,1) + delta);

    % label(i) = 1 if features1(:,i) and features2(:,i) are features
    % for similar images, and 0 otherwise
    lossSimilar = pairLabel.*(distances.^2);

    lossDissimilar = (1 - pairLabel).*(max(margin - distances, 0).^2);

    loss = 0.5*sum(lossSimilar + lossDissimilar,'all');
end
```

Create Batches of Image Pairs

The following functions create randomized pairs of images that are similar or dissimilar, based on their labels. Within this example, the function `getSiameseBatch` is introduced in the section Create Pairs of Similar and Dissimilar Images on page 3-0 .

```
function [X1,X2,pairLabels] = getSiameseBatch(X,Y,miniBatchSize)
% getSiameseBatch returns a randomly selected batch of paired images.
% On average, this function produces a balanced set of similar and
% dissimilar pairs.
    pairLabels = zeros(1, miniBatchSize);
    imgSize = size(X(:,:,:,1));
    X1 = zeros([imgSize 1 miniBatchSize]);
    X2 = zeros([imgSize 1 miniBatchSize]);

    for i = 1:miniBatchSize
        choice = rand(1);
        if choice < 0.5
            [pairIdx1, pairIdx2, pairLabels(i)] = getSimilarPair(Y);
        else
            [pairIdx1, pairIdx2, pairLabels(i)] = getDissimilarPair(Y);
        end
        X1(:,:,:,i) = X(:,:,:,pairIdx1);
        X2(:,:,:,i) = X(:,:,:,pairIdx2);
```

```
    end

end

function [pairIdx1,pairIdx2,pairLabel] = getSimilarPair(classLabel)
% getSimilarPair returns a random pair of indices for images
% that are in the same class and the similar pair label = 1.

    % Find all unique classes.
    classes = unique(classLabel);

    % Choose a class randomly which will be used to get a similar pair.
    classChoice = randi(numel(classes));

    % Find the indices of all the observations from the chosen class.
    idxs = find(classLabel==classes(classChoice));

    % Randomly choose two different images from the chosen class.
    pairIdxChoice = randperm(numel(idxs),2);
    pairIdx1 = idxs(pairIdxChoice(1));
    pairIdx2 = idxs(pairIdxChoice(2));
    pairLabel = 1;
end

function  [pairIdx1,pairIdx2,pairLabel] = getDissimilarPair(classLabel)
% getDissimilarPair returns a random pair of indices for images
% that are in different classes and the dissimilar pair label = 0.

    % Find all unique classes.
    classes = unique(classLabel);

    % Choose two different classes randomly which will be used to get a dissimilar pair.
    classesChoice = randperm(numel(classes), 2);

    % Find the indices of all the observations from the first and second classes.
    idxs1 = find(classLabel==classes(classesChoice(1)));
    idxs2 = find(classLabel==classes(classesChoice(2)));

    % Randomly choose one image from each class.
    pairIdx1Choice = randi(numel(idxs1));
    pairIdx2Choice = randi(numel(idxs2));
    pairIdx1 = idxs1(pairIdx1Choice);
    pairIdx2 = idxs2(pairIdx2Choice);
    pairLabel = 0;
end
```

References

[1] Bromley, J., I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. "*Signature Verification using a "Siamese" Time Delay Neural Network*." In Proceedings of the 6th International Conference on Neural Information Processing Systems (NIPS 1993), 1994, pp737-744. Available at Signature Verification using a "Siamese" Time Delay Neural Network on the NIPS Proceedings website.

[2] Wenpeg, Y., and H Schütze. "*Convolutional Neural Network for Paraphrase Identification*." In Proceedings of 2015 Conference of the North American Cahapter of the ACL, 2015, pp901-911. Available at Convolutional Neural Network for Paraphrase Identification on the ACL Anthology website.

[3] Hadsell, R., S. Chopra, and Y. LeCun. "*Dimensionality Reduction by Learning an Invariant Mapping*." In Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006), 2006, pp1735-1742.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `adamupdate`

More About

- “Train a Siamese Network to Compare Images” on page 3-128
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Functions with dlarray Support” on page 18-423

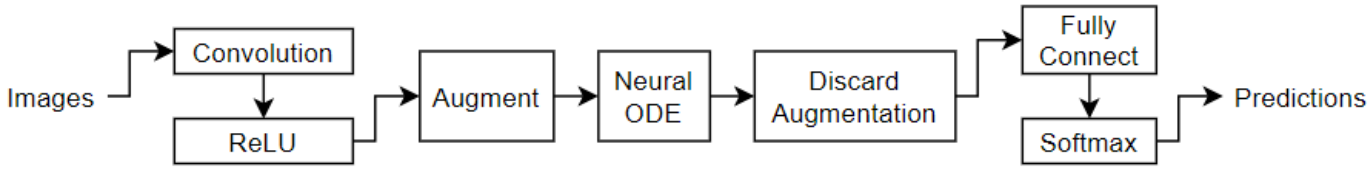
Train Neural ODE Network

This example shows how to train an augmented neural ordinary differential equation (ODE) network.

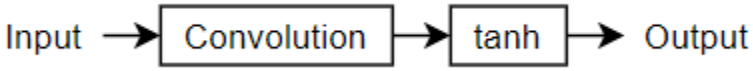
A neural ODE [1 on page 3-0] is a deep learning operation that returns the solution of an ODE. In particular, given an input, a neural ODE operation outputs the numerical solution of the ODE $y' = f(t, y, \theta)$ for the time horizon (t_0, t_1) and the initial condition $y(t_0) = y_0$, where t and y denote the ODE function inputs and θ is a set of learnable parameters. Typically, the initial condition y_0 is either the network input or, as in the case of this example, the output of another deep learning operation.

An *augmented* neural ODE [2 on page 3-0] operation improves upon a standard neural ODE by augmenting the input data with extra channels and then discarding the augmentation after the neural ODE operation. Empirically, augmented neural ODEs are more stable, generalize better, and have a lower computational cost than neural ODEs.

This example trains a simple convolutional neural network with an augmented neural ODE operation.



The ODE function can be a collection of deep learning operations. In this example, the model uses a convolution-tanh block as the ODE function:



The example shows how to train a neural network to classify images of digits using an augmented neural ODE operation.

Load Training Data

Load the training images and labels using the `digitTrain4DArrayData` function.

```
[XTrain,TTrain] = digitTrain4DArrayData;
```

View the number of classes of the training data.

```
classNames = categories(TTrain);
numClasses = numel(classNames)
```

```
numClasses = 10
```

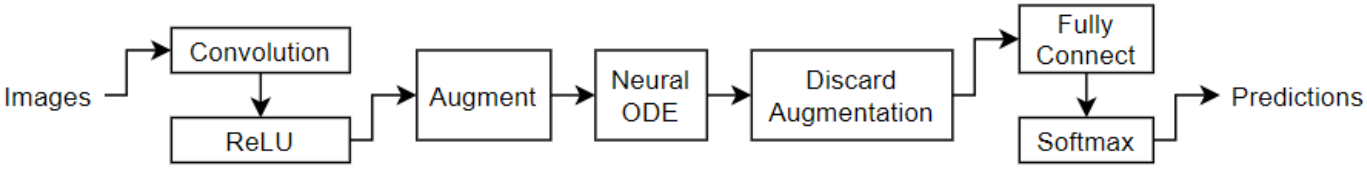
View some images from the training data.

```
numObservations = size(XTrain,4);
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

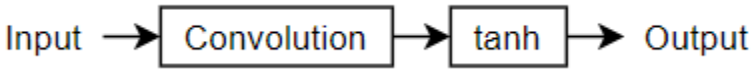
Define Deep Learning Model

Define the following network, which classifies images.

- A convolution-ReLU block with 8 3-by-3 filters with a stride of 2
- An augmentation step that concatenates an array of zeros to the input such that the number of channels is doubled
- A neural ODE operation with ODE function containing a convolution-tanh block with 16 3-by-3 filters
- For classification output, a fully connect operation of size 10 (the number of classes) and a softmax operation



A neural ODE operation outputs the solution of a specified ODE function. For this example, specify a convolution-tanh block as the ODE function.



That is, specify the ODE function given by $y' = f(t, y, \theta)$, where f denotes the convolution-tanh operation, y is the input data, and θ contains the learnable parameters for the convolution operation. In this case, the variable t is unused.

Define and Initialize Model Parameters

Define the learnable parameters for each of the operations and include them in a structure. Use the format `parameters.OperationName.ParameterName`, where `parameters` is the structure, `OperationName` is the name of the operation (for example, "conv1"), and `ParameterName` is the name of the parameter (for example, "Weights"). Initialize the learnable layer weights and biases using the `initializeGlorot` and `initializeZeros` example functions, respectively. The initialization example functions are attached to this example as supporting files. To access these functions, open this example as a live script. For more information about initializing learnable parameters for model functions, see “Initialize Learnable Parameters for Model Function” on page 18-292.

Initialize the parameters structure.

```
parameters = struct;
```

Initialize the parameters for the first convolutional layer. Specify 8 3-by-3 filters. If you change these dimensions, then you must manually calculate the input size of the fully connect operation for its Glorot weights initialization.

```
filterSize = [3 3];
numFilters = 8;

numChannels = size(XTrain,3);
sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv1.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters for the convolution operation used in the neural ODE function. Because the augmentation step augments the input data with an array of zeros, the number of input channels is given by `numFilters + numExtraChannels`, where `numExtraChannels` is the number of channels in the augmentation. Similarly, because the model discards channels of the output of the neural ODE operation corresponding to the augmentation, the convolution operation in the neural ODE must have (`numChannels + numExtraChannels`) filters, where `numChannels` is the desired number of output channels.

Specify the same number of filters as the first convolution layer and a matching augmentation size.

```
numChannels = numFilters;
numExtraChannels = numFilters;

numFiltersAugmented = numChannels + numExtraChannels;
sz = [filterSize numFiltersAugmented numFiltersAugmented];

numOut = prod(filterSize) * numFiltersAugmented;
numIn = prod(filterSize) * numFiltersAugmented;

parameters.neuralode.Weights = initializeGlorot(sz,numOut,numIn);
parameters.neuralode.Bias = initializeZeros([numFiltersAugmented 1]);
```

Initialize the parameters for the fully connect operation. To initialize the weights of the fully connect operation using the Glorot initializer, first calculate the number of input elements to the operation.

For each operation in the model that changes the size of the data flowing through, consider the output sizes when you pass 28-by-28 images through the model:

- The first convolution has 8 filters with `"same"` padding and a stride of 2. This operation outputs 14-by-14 images with 8 channels.
- The model then augments the data with an 8-channel array of zeros. This operation outputs 14-by-14 images with 16 channels.
- The neural ODE operation has a convolution operation with 16 filters and `"same"` padding. This operation outputs 14-by-14 images with 16 channels.
- The model then discards the channels corresponding to the augmentation. This operation outputs 14-by-14 images with 8 channels.

This means that the number of input elements to the fully connect operation is 14*14*8 = 1568.

```
sz =  [14 14];
inputSize = prod(sz)*numChannels;
outputSize = numClasses;

sz = [outputSize inputSize];
numOut = outputSize;
numIn = inputSize;

parameters.fc1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc1.Bias = initializeZeros([outputSize 1]);
```

View the structure of parameters.

```
parameters
```

```
parameters = struct with fields:
        conv1: [1×1 struct]
    neuralode: [1×1 struct]
          fc1: [1×1 struct]
```

View the parameters for the neural ODE operation.

```
parameters.neuralode
```

```
ans = struct with fields:
    Weights: [3×3×16×16 dlarray]
       Bias: [16×1 dlarray]
```

Define Model Hyper Parameters

Define the hyperparameters for the operations and include them in a structure. Use the format `hyperparameters.OperationName.ParameterName` where `hyperparameters` is the structure, `OperationName` is the name of the operation (for example "neuralode") and `ParameterName` is the name of the hyperparameter (for example, "tspan").

Initialize the hyperparameters structure.

```
hyperparameters = struct;
```

For the neural ODE, specify an interval of integration of [0 0.1].

```
hyperparameters.neuralode.tspan = [0 0.1];
```

Define Neural ODE Function

Create the function `odeModel`, listed in the ODE Function on page 3-0 section of the example, which takes as input the time input (unused), the initial conditions, and the ODE function parameters. The function applies a convolution operation followed by a tanh operation to the input data using the weights and biases given by the parameters.

Define Model Function

Create the function `model`, listed in the Model Function on page 3-0 section of the example, which computes the outputs of the deep learning model. The function `model` takes as input the model parameters and the input data. The function outputs the predictions for the labels.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients on page 3-0 section of the example, which takes as input the model parameters and a mini-batch of input data with corresponding targets containing the labels, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Specify Training Options

Specify the training options. Train with a mini-batch size of 64 for 30 epochs.

```
miniBatchSize = 64;
numEpochs = 30;
```

Train Model

Train the model using a custom training loop.

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. To create a `minibatchqueue` object, first create a datastore that returns the images and labels by creating array datastores and then combining them.

```
dsXTrain = arrayDatastore(XTrain,IterationDimension=4);
dsTTrain = arrayDatastore(TTrain);
dsTrain = combine(dsXTrain,dsTTrain);
```

Create the mini-batch queue. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch`, defined in the Mini-Batch Preprocessing Function on page 3-0 section of the example, to convert the labels to one-hot encoded variables.
- Format the image data with the dimension labels `"SSCB"` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Discard partial mini-batches.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatch, ...
    MiniBatchFormat=["SSCB" "CB"]);
```

Initialize the moving average of the parameter gradients and the element-wise squares of the gradients used by the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Initialize the training plot.

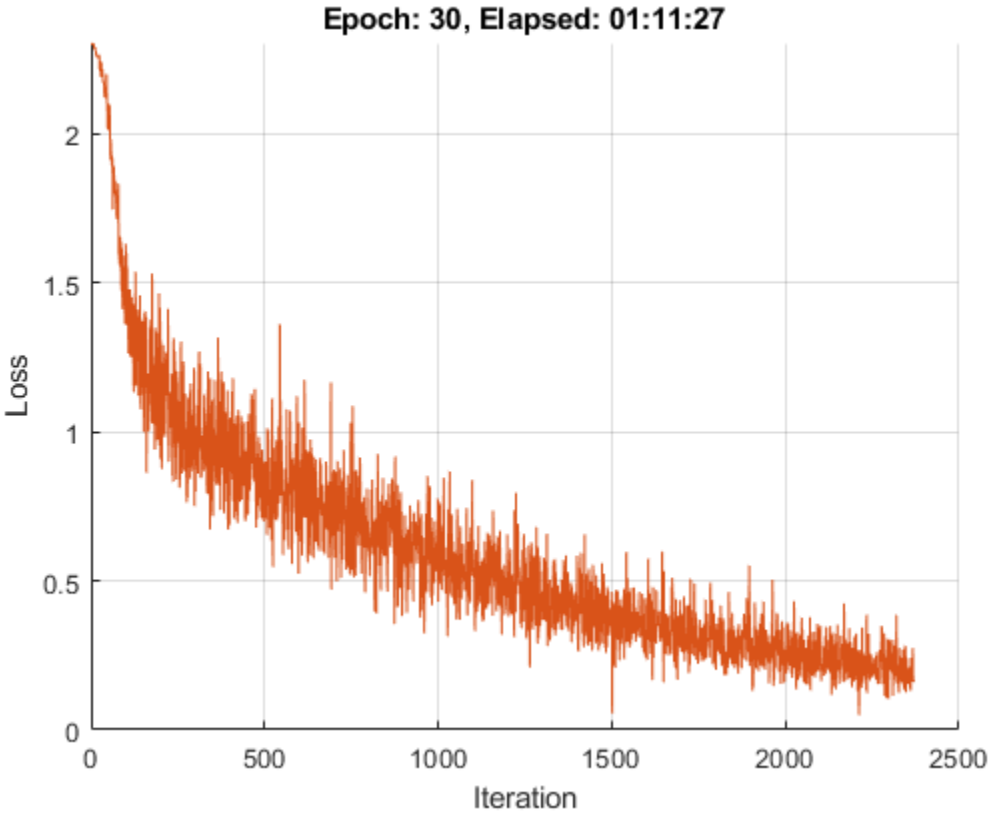
```
figure
C = colororder;
lineLossTrain = animatedline(Color=C(2,:));
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Train the model using a custom training loop. For each epoch, shuffle the data. For each mini-batch:

- Evaluate the model gradients using the `dlfeval` and `modelGradients` functions.
- Update the network parameters using the `adamupdate` function.
- Update the training progress plot.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,dlT] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss] = dlfeval(@modelGradients, parameters, dlX, dlT, hyperparameters);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);

        % Display the training progress.
        D = duration(0,0,toc(start),Format="hh:mm:ss");
        loss = double(gather(extractdata(loss)));
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
```

```
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a held-out test set with the true labels.

Load the test data.

```
[XTest,TTest] = digitTest4DArrayData;
```

After training, making predictions on new data does not require the labels. Create a `minibatchqueue` object containing only the predictors of the test data:

- Set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessPredictors` function, listed in the Mini-Batch Predictors Preprocessing Function on page 3-0 section of the example.
- For the single output of the datastore, specify the mini-batch format `"SSCB"` (spatial, spatial, channel, batch).

```
dsTest = arrayDatastore(XTest,IterationDimension=4);

mbqTest = minibatchqueue(dsTest,1, ...
    MiniBatchSize=miniBatchSize, ...
```

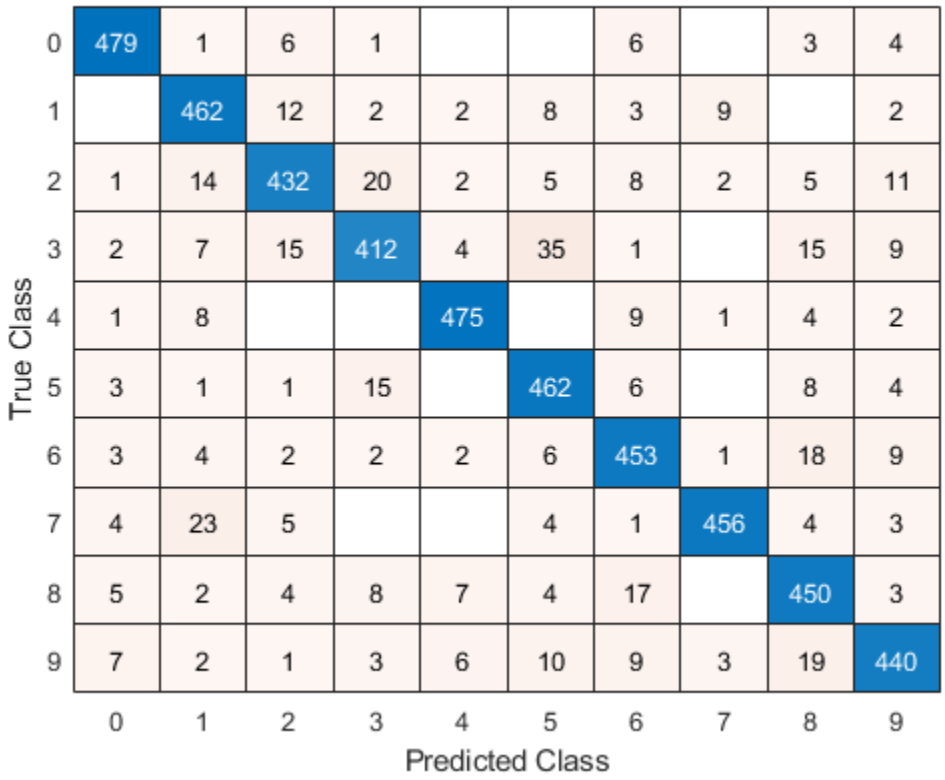
```
    MiniBatchFormat="SSCB", ...
    MiniBatchFcn=@preprocessPredictors);
```

Loop over the mini-batches and classify the sequences using `modelPredictions` function, listed in the Model Predictions Function on page 3-0 section of the example.

```
YPred = modelPredictions(parameters,hyperparameters,mbqTest,classNames);
```

Visualize the predictions in a confusion matrix.

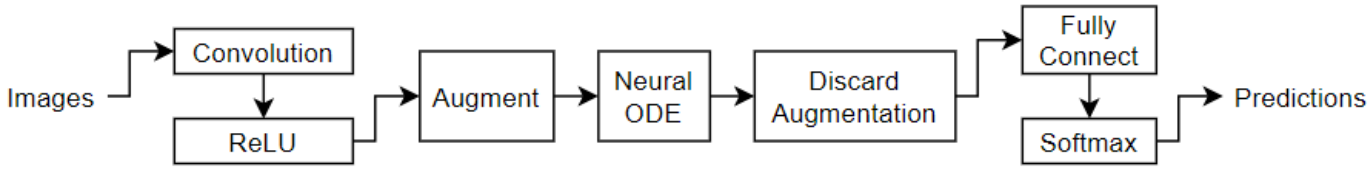
```
figure
confusionchart(TTest,YPred)
```



Model Function

The function `model` takes as input the model parameters, the input data `dlX`, the model hyperparameters, and outputs the predictions for the labels.

This diagram outlines the model structure.

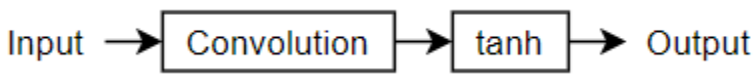


For the neural ODE operation, use the `dlode45` function and specify the `odeModel` function, listed in the ODE Function on page 3-0 section of the example. Increase the absolute and relative tolerance using the `AbsoluteTolerance` and `RelativeTolerance` name-value arguments, respectively. To calculate the gradients by solving the associated adjoint ODE system, set the `GradientMode` option to `"adjoint"`.

```
function dlY = model(parameters,dlX,hyperparameters)

% Convolution, ReLU.
weights = parameters.conv1.Weights;
bias = parameters.conv1.Bias;
dlY = dlconv(dlX,weights,bias,Padding="same",Stride=2);

dlY = relu(dlY);

% Augment.
weights = parameters.neuralode.Weights;

numChannels = size(dlY,3);
szAugmented = size(dlY);
szAugmented(3) = size(weights,3) - numChannels;

dlY0 = cat(3, dlY, zeros(szAugmented,"like",dlY));

% Neural ODE.
tspan = hyperparameters.neuralode.tspan;
dlY = dlode45(@odeModel,tspan,dlY0,parameters.neuralode, ...
    GradientMode="adjoint", ...
    AbsoluteTolerance=1e-3, ...
    RelativeTolerance=1e-4);

% Discard augmentation.
dlY(:,:,numChannels+1:end,:) = [];

% Fully connect, softmax.
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY = fullyconnect(dlY,weights,bias);

dlY = softmax(dlY);

end
```

ODE Function

The neural ODE operation consists of a convolution operation followed by a tanh operation.



The ODE function `odeModel` takes as input the function inputs `t` (unused) and `y` and the ODE function parameters `p` containing the convolution weights and biases, and returns the output of the convolution-tanh block operation.

```
function z = odeModel(t,y,p)
```

```
weights = p.Weights;
bias = p.Bias;

z = dlconv(y,weights,bias,Padding="same");
z = tanh(z);

end
```

Model Gradients Function

The `modelGradients` function takes as input the model parameters, a mini-batch of input data `dlX` with corresponding targets `dlT`, and model hyperparameters, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. To compute the gradients using automatic differentiation, use the `dlgradient` function.

```
function [gradients,loss] = modelGradients(parameters,dlX,dlT,hyperparameters)

dlY = model(parameters,dlX,hyperparameters);

loss = crossentropy(dlY,dlT);

gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes as input the model parameters, model hyperparameters, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted classes with the highest score.

```
function predictions = modelPredictions(parameters,hyperparameters,mbq,classNames)

predictions = [];

while hasdata(mbq)
    dlX = next(mbq);
    dlYPred = model(parameters,dlX,hyperparameters);
    YPred = onehotdecode(dlYPred,classNames,1)';
    predictions = [predictions; YPred];
end

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Preprocess the images using the `preprocessPredictors` function.
2. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)
```

```
% Preprocess predictors.
X = preprocessPredictors(XCell);

% Extract label data from cell and concatenate.
Y = cat(2,YCell{:});

% One-hot encode labels.
Y = onehotencode(Y,1);

end
```

Predictors Preprocessing Function

The `preprocessPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenating the data into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image to use as a singleton channel dimension.

```
function X = preprocessPredictors(XCell)

X = cat(4,XCell{:});

end
```

Bibliography

1. Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. “Neural Ordinary Differential Equations.” Preprint, submitted June 19, 2018. https://arxiv.org/abs/1806.07366.
2. Dupont, Emilien, Arnaud Doucet, and Yee Whye Teh. “Augmented Neural ODEs.” Preprint, submitted October 26, 2019. https://arxiv.org/abs/1904.01681.

See Also

`dlode45` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

- “Dynamical System Modeling Using Neural ODE” on page 18-368
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Functions with dlarray Support” on page 18-423

Train Variational Autoencoder (VAE) to Generate Images

This example shows how to create a variational autoencoder (VAE) in MATLAB to generate digit images. The VAE generates hand-drawn digits in the style of the MNIST data set.

VAEs differ from regular autoencoders in that they do not use the encoding-decoding process to reconstruct an input. Instead, they impose a probability distribution on the latent space, and learn the distribution so that the distribution of outputs from the decoder matches that of the observed data. Then, they sample from this distribution to generate new data.

In this example, you construct a VAE network, train it on the MNIST data set, and generate new images that closely resemble those in the data set.

Load Data

Download the MNIST files from http://yann.lecun.com/exdb/mnist/ and load the MNIST data set into the workspace [1]. Call the `processImagesMNIST` and `processLabelsMNIST` helper functions attached to this example to load the data from the files into MATLAB arrays.

Because the VAE compares the reconstructed digits against the inputs and not against the categorical labels, you do not need to use the training labels in the MNIST data set.

```
trainImagesFile = 'train-images-idx3-ubyte.gz';
testImagesFile = 't10k-images-idx3-ubyte.gz';
testLabelsFile = 't10k-labels-idx1-ubyte.gz';

XTrain = processImagesMNIST(trainImagesFile);
```

```
Read MNIST image data...
Number of images in the dataset:  60000 ...
```

```
numTrainImages = size(XTrain,4);
XTest = processImagesMNIST(testImagesFile);
```

```
Read MNIST image data...
Number of images in the dataset:  10000 ...
```

```
YTest = processLabelsMNIST(testLabelsFile);
```

```
Read MNIST label data...
Number of labels in the dataset:  10000 ...
```

Construct Network

Autoencoders have two parts: the encoder and the decoder. The encoder takes an image input and outputs a compressed representation (the encoding), which is a vector of size `latentDim`, equal to 20 in this example. The decoder takes the compressed representation, decodes it, and recreates the original image.

To make calculations more numerically stable, increase the range of possible values from [0,1] to [-inf, 0] by making the network learn from the logarithm of the variances. Define two vectors of size `latent_dim`: one for the means μ and one for the logarithm of the variances $\log\left(\sigma^2\right)$. Then use these two vectors to create the distribution to sample from.

Use 2-D convolutions followed by a fully connected layer to downsample from the 28-by-28-by-1 MNIST image to the encoding in the latent space. Then, use transposed 2-D convolutions to scale up the 1-by-1-by-20 encoding back into a 28-by-28-by-1 image.

```
latentDim = 20;
imageSize = [28 28 1];

encoderLG = layerGraph([
    imageInputLayer(imageSize,'Name','input_encoder','Normalization','none')
    convolution2dLayer(3, 32, 'Padding','same', 'Stride', 2, 'Name', 'conv1')
    reluLayer('Name','relu1')
    convolution2dLayer(3, 64, 'Padding','same', 'Stride', 2, 'Name', 'conv2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(2 * latentDim, 'Name', 'fc_encoder')
    ]);

decoderLG = layerGraph([
    imageInputLayer([1 1 latentDim],'Name','i','Normalization','none')
    transposedConv2dLayer(7, 64, 'Cropping', 'same', 'Stride', 7, 'Name', 'transpose1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(3, 64, 'Cropping', 'same', 'Stride', 2, 'Name', 'transpose2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(3, 32, 'Cropping', 'same', 'Stride', 2, 'Name', 'transpose3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(3, 1, 'Cropping', 'same', 'Name', 'transpose4')
    ]);
```

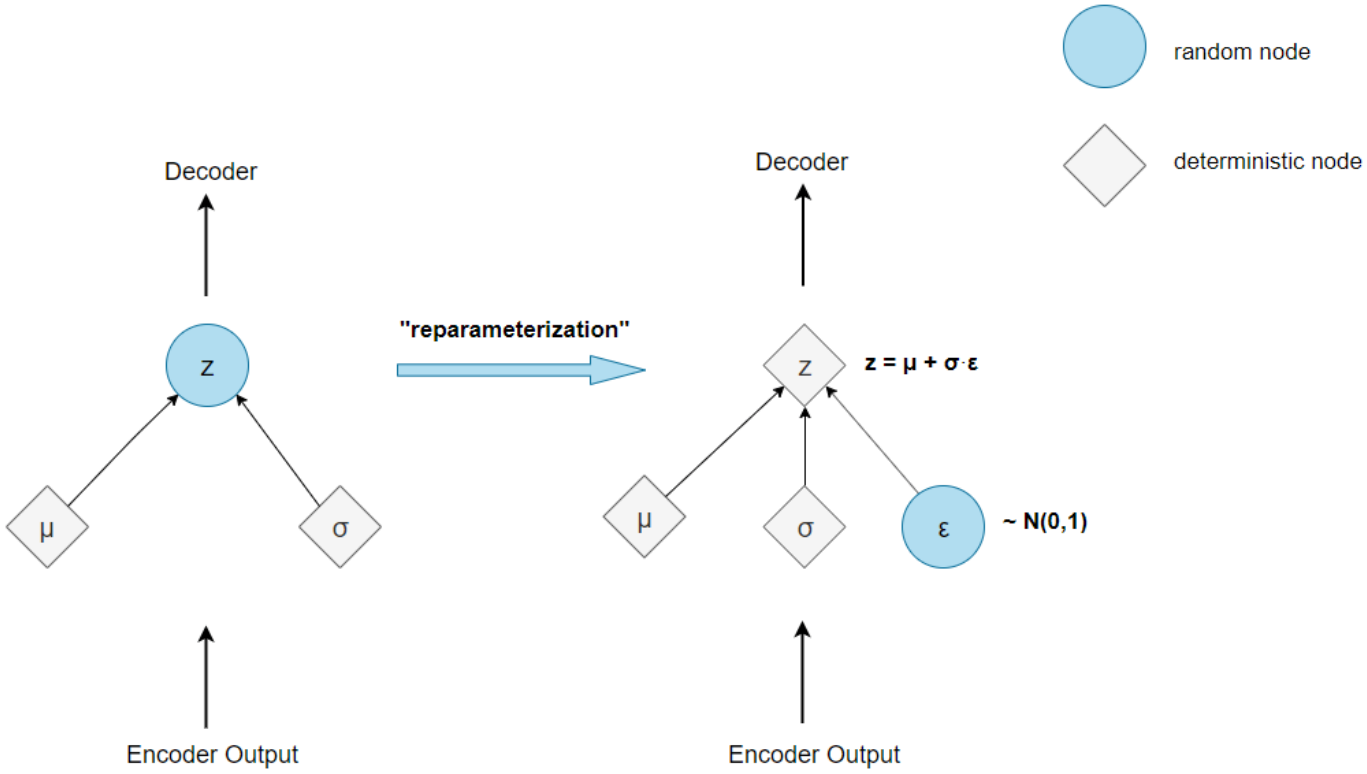
To train both networks with a custom training loop and enable automatic differentiation, convert the layer graphs to `dlnetwork` objects.

```
encoderNet = dlnetwork(encoderLG);
decoderNet = dlnetwork(decoderLG);
```

Define Model Gradients Function

The helper function modelGradients on page 3-0 takes in the encoder and decoder `dlnetwork` objects and a mini-batch of input data `X`, and returns the gradients of the loss with respect to the learnable parameters in the networks. This helper function is defined at the end of this example.

The function performs this process in two steps: sampling and loss on page 3-0 . The sampling step samples the mean and the variance vectors to create the final encoding to be passed to the decoder network. However, because backpropagation through a random sampling operation is not possible, you must use the *reparameterization trick*. This trick moves the random sampling operation to an auxiliary variable ε, which is then shifted by the mean μ_i and scaled by the standard deviation σ_i. The idea is that sampling from $N\left(\mu_i, \sigma_i^2\right)$ is the same as sampling from $\mu_i + \varepsilon \cdot \sigma_i$, where $\varepsilon \sim N(0, 1)$. The following figure depicts this idea graphically.



The loss step passes the encoding generated by the sampling step through the decoder network, and determines the loss, which is then used to compute the gradients. The loss in VAEs, also called the evidence lower bound (ELBO) loss, is defined as a sum of two separate loss terms:

ELBO loss = reconstruction loss + KL loss.

The *reconstruction loss* measures how close the decoder output is to the original input by using the mean-squared error (MSE):

reconstruction loss = MSE(decoder output, original image).

The *KL loss*, or Kullback–Leibler divergence, measures the difference between two probability distributions. Minimizing the KL loss in this case means ensuring that the learned means and variances are as close as possible to those of the target (normal) distribution. For a latent dimension of size n, the KL loss is obtained as

$$\text{KL loss} = -0.5 \cdot \sum_{i=1}^{n} \left(1 + \log\left(\sigma_i^2\right) - \mu_i^2 - \sigma_i^2\right).$$

The practical effect of including a KL loss term is to pack the clusters learned due to the reconstruction loss tightly around the center of the latent space, forming a continuous space to sample from.

Specify Training Options

Train on a GPU if one is available (requires Parallel Computing Toolbox™).

```
executionEnvironment = "auto";
```

Set the training options for the network. When using the Adam optimizer, you need to initialize for each network the trailing average gradient and the trailing average gradient-square decay rates with empty arrays.

```
numEpochs = 50;
miniBatchSize = 512;
lr = 1e-3;
numIterations = floor(numTrainImages/miniBatchSize);
iteration = 0;

avgGradientsEncoder = [];
avgGradientsSquaredEncoder = [];
avgGradientsDecoder = [];
avgGradientsSquaredDecoder = [];
```

Train Model

Train the model using a custom training loop.

For each iteration in an epoch:

- Obtain the next mini-batch from the training set.
- Convert the mini-batch to a `dlarray` object, making sure to specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the `dlarray` to a `gpuArray` object.
- Evaluate the model gradients using the `dlfeval` and `modelGradients` functions.
- Update the network learnables and the average gradients for both networks, using the `adamupdate` function.

At the end of each epoch, pass the test set images through the autoencoder, and display the loss and the training time for that epoch.

```
for epoch = 1:numEpochs
    tic;
    for i = 1:numIterations
        iteration = iteration + 1;
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        XBatch = XTrain(:,:,:,idx);
        XBatch = dlarray(single(XBatch), 'SSCB');

        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            XBatch = gpuArray(XBatch);
        end

        [infGrad, genGrad] = dlfeval(...
            @modelGradients, encoderNet, decoderNet, XBatch);

        [decoderNet.Learnables, avgGradientsDecoder, avgGradientsSquaredDecoder] = ...
            adamupdate(decoderNet.Learnables, ...
                genGrad, avgGradientsDecoder, avgGradientsSquaredDecoder, iteration, lr);
        [encoderNet.Learnables, avgGradientsEncoder, avgGradientsSquaredEncoder] = ...
            adamupdate(encoderNet.Learnables, ...
                infGrad, avgGradientsEncoder, avgGradientsSquaredEncoder, iteration, lr);
    end
    elapsedTime = toc;
```

```
    [z, zMean, zLogvar] = sampling(encoderNet, XTest);
    xPred = sigmoid(forward(decoderNet, z));
    elbo = ELBOloss(XTest, xPred, zMean, zLogvar);
    disp("Epoch : "+epoch+" Test ELBO loss = "+gather(extractdata(elbo))+...
        ". Time taken for epoch = "+ elapsedTime + "s")
end
```

```
Epoch : 1 Test ELBO loss = 28.0145. Time taken for epoch = 28.0573s
Epoch : 2 Test ELBO loss = 24.8995. Time taken for epoch = 8.797s
Epoch : 3 Test ELBO loss = 23.2756. Time taken for epoch = 8.8824s
Epoch : 4 Test ELBO loss = 21.151. Time taken for epoch = 8.5979s
Epoch : 5 Test ELBO loss = 20.5335. Time taken for epoch = 8.8472s
Epoch : 6 Test ELBO loss = 20.232. Time taken for epoch = 8.5068s
Epoch : 7 Test ELBO loss = 19.9988. Time taken for epoch = 8.4356s
Epoch : 8 Test ELBO loss = 19.8955. Time taken for epoch = 8.4015s
Epoch : 9 Test ELBO loss = 19.7991. Time taken for epoch = 8.8089s
Epoch : 10 Test ELBO loss = 19.6773. Time taken for epoch = 8.4269s
Epoch : 11 Test ELBO loss = 19.5181. Time taken for epoch = 8.5771s
Epoch : 12 Test ELBO loss = 19.4532. Time taken for epoch = 8.4227s
Epoch : 13 Test ELBO loss = 19.3771. Time taken for epoch = 8.5807s
Epoch : 14 Test ELBO loss = 19.2893. Time taken for epoch = 8.574s
Epoch : 15 Test ELBO loss = 19.1641. Time taken for epoch = 8.6434s
Epoch : 16 Test ELBO loss = 19.2175. Time taken for epoch = 8.8641s
Epoch : 17 Test ELBO loss = 19.158. Time taken for epoch = 9.1083s
Epoch : 18 Test ELBO loss = 19.085. Time taken for epoch = 8.6674s
Epoch : 19 Test ELBO loss = 19.1169. Time taken for epoch = 8.6357s
Epoch : 20 Test ELBO loss = 19.0791. Time taken for epoch = 8.5512s
Epoch : 21 Test ELBO loss = 19.0395. Time taken for epoch = 8.4674s
Epoch : 22 Test ELBO loss = 18.9556. Time taken for epoch = 8.3943s
Epoch : 23 Test ELBO loss = 18.9469. Time taken for epoch = 10.2924s
Epoch : 24 Test ELBO loss = 18.924. Time taken for epoch = 9.8302s
Epoch : 25 Test ELBO loss = 18.9124. Time taken for epoch = 9.9603s
Epoch : 26 Test ELBO loss = 18.9595. Time taken for epoch = 10.9887s
Epoch : 27 Test ELBO loss = 18.9256. Time taken for epoch = 10.1402s
Epoch : 28 Test ELBO loss = 18.8708. Time taken for epoch = 9.9109s
Epoch : 29 Test ELBO loss = 18.8602. Time taken for epoch = 10.3075s
Epoch : 30 Test ELBO loss = 18.8563. Time taken for epoch = 10.474s
Epoch : 31 Test ELBO loss = 18.8127. Time taken for epoch = 9.8779s
Epoch : 32 Test ELBO loss = 18.7989. Time taken for epoch = 9.6963s
Epoch : 33 Test ELBO loss = 18.8. Time taken for epoch = 9.8848s
Epoch : 34 Test ELBO loss = 18.8095. Time taken for epoch = 10.3168s
Epoch : 35 Test ELBO loss = 18.7601. Time taken for epoch = 10.8058s
Epoch : 36 Test ELBO loss = 18.7469. Time taken for epoch = 9.9365s
Epoch : 37 Test ELBO loss = 18.7049. Time taken for epoch = 10.0343s
Epoch : 38 Test ELBO loss = 18.7084. Time taken for epoch = 10.3214s
Epoch : 39 Test ELBO loss = 18.6858. Time taken for epoch = 10.3985s
Epoch : 40 Test ELBO loss = 18.7284. Time taken for epoch = 10.9685s
Epoch : 41 Test ELBO loss = 18.6574. Time taken for epoch = 10.5241s
Epoch : 42 Test ELBO loss = 18.6388. Time taken for epoch = 10.2392s
Epoch : 43 Test ELBO loss = 18.7133. Time taken for epoch = 9.8177s
Epoch : 44 Test ELBO loss = 18.6846. Time taken for epoch = 9.6858s
Epoch : 45 Test ELBO loss = 18.6001. Time taken for epoch = 9.5588s
Epoch : 46 Test ELBO loss = 18.5897. Time taken for epoch = 10.4554s
Epoch : 47 Test ELBO loss = 18.6184. Time taken for epoch = 10.0317s
Epoch : 48 Test ELBO loss = 18.6389. Time taken for epoch = 10.311s
Epoch : 49 Test ELBO loss = 18.5918. Time taken for epoch = 10.4506s
Epoch : 50 Test ELBO loss = 18.5081. Time taken for epoch = 9.9671s
```

Visualize Results

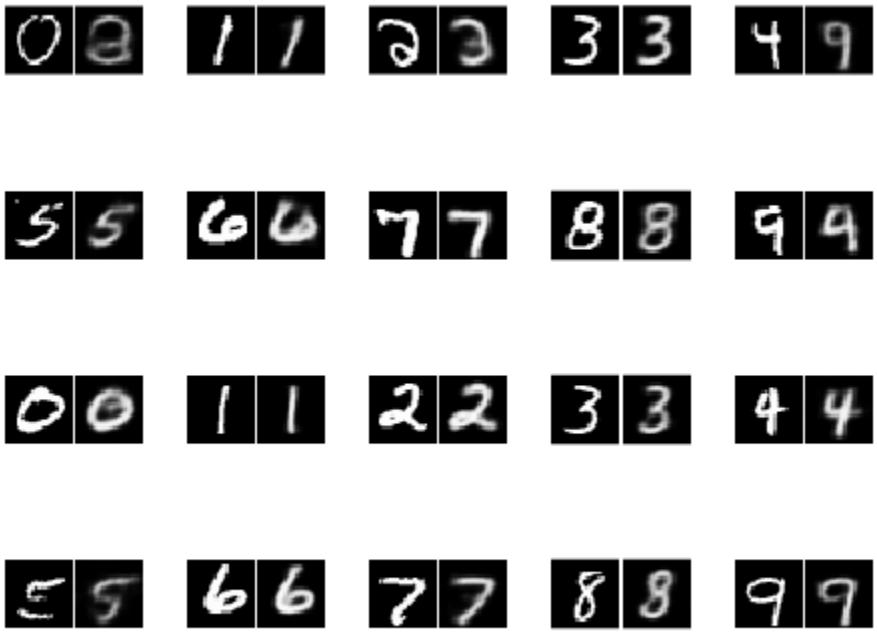
To visualize and interpret the results, use the helper Visualization functions on page 3-0 . These helper functions are defined at the end of this example.

The `VisualizeReconstruction` function shows a randomly chosen digit from each class accompanied by its reconstruction after passing through the autoencoder.

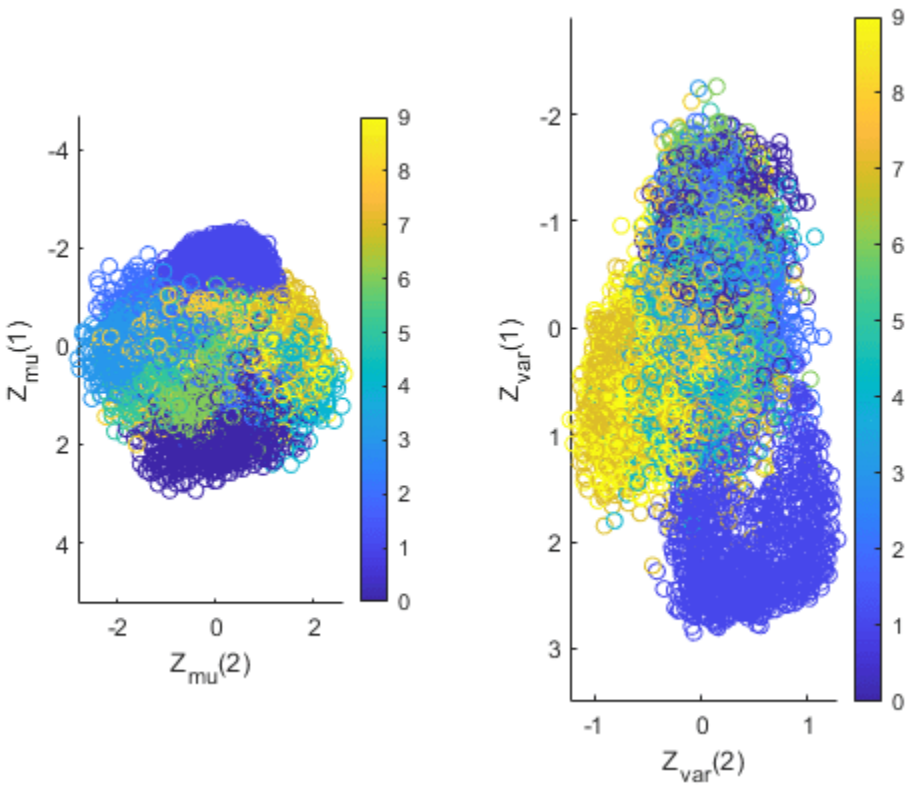
The `VisualizeLatentSpace` function takes the mean and the variance encodings (each of dimension 20) generated after passing the test images through the encoder network, and performs principal component analysis (PCA) on the matrix containing the encodings for each of the images. You can then visualize the latent space defined by the means and the variances in the two dimensions characterized by the two first principal components.

The `Generate` function initializes new encodings sampled from a normal distribution, and outputs the images generated when these encodings pass through the decoder network.

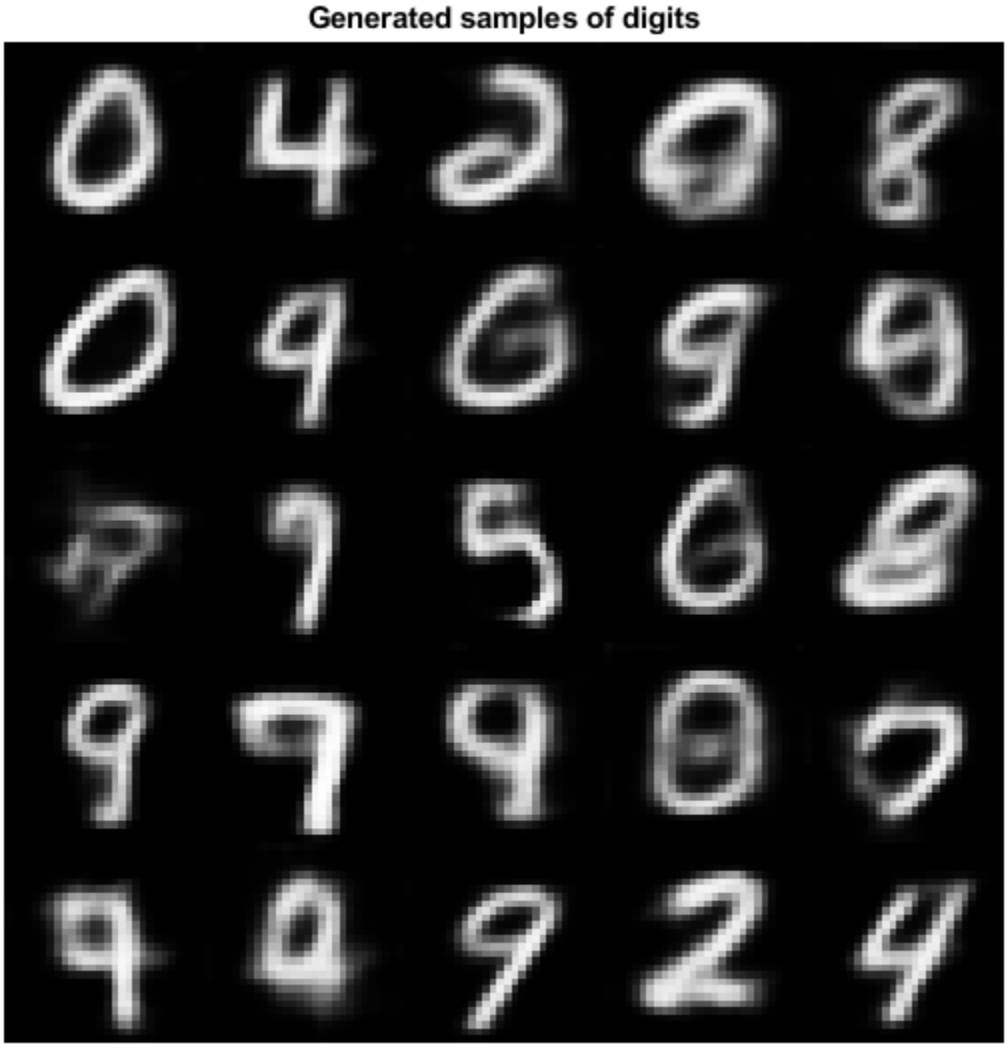
```
visualizeReconstruction(XTest, YTest, encoderNet, decoderNet)
```



```
visualizeLatentSpace(XTest, YTest, encoderNet)
```




```
generate(decoderNet, latentDim)
```



Next Steps

Variational autoencoders are only one of the many available models used to perform generative tasks. They work well on data sets where the images are small and have clearly defined features (such as MNIST). For more complex data sets with larger images, generative adversarial networks (GANs) tend to perform better and generate images with less noise. For an example showing how to implement GANs to generate 64-by-64 RGB images, see “Train Generative Adversarial Network (GAN)” on page 3-76.

References

1 LeCun, Y., C. Cortes, and C. J. C. Burges. "The MNIST Database of Handwritten Digits." http://yann.lecun.com/exdb/mnist/.

Helper Functions

Model Gradients Function

The `modelGradients` function takes the encoder and decoder `dlnetwork` objects and a mini-batch of input data `X`, and returns the gradients of the loss with respect to the learnable parameters in the networks. The function performs three operations:

1. Obtain the encodings by calling the `sampling` function on the mini-batch of images that passes through the encoder network.
2. Obtain the loss by passing the encodings through the decoder network and calling the `ELBOloss` function.
3. Compute the gradients of the loss with respect to the learnable parameters of both networks by calling the `dlgradient` function.

```
function [infGrad, genGrad] = modelGradients(encoderNet, decoderNet, x)
[z, zMean, zLogvar] = sampling(encoderNet, x);
xPred = sigmoid(forward(decoderNet, z));
loss = ELBOloss(x, xPred, zMean, zLogvar);
[genGrad, infGrad] = dlgradient(loss, decoderNet.Learnables, ...
    encoderNet.Learnables);
end
```

Sampling and Loss Functions

The `sampling` function obtains encodings from input images. Initially, it passes a mini-batch of images through the encoder network and splits the output of size `(2*latentDim)*miniBatchSize` into a matrix of means and a matrix of variances, each of size `latentDim*batchSize`. Then, it uses these matrices to implement the reparameterization trick and to compute the encoding. Finally, it converts this encoding to a `dlarray` object in SSCB format.

```
function [zSampled, zMean, zLogvar] = sampling(encoderNet, x)
compressed = forward(encoderNet, x);
d = size(compressed,1)/2;
zMean = compressed(1:d,:);
zLogvar = compressed(1+d:end,:);

sz = size(zMean);
epsilon = randn(sz);
sigma = exp(.5 * zLogvar);
z = epsilon .* sigma + zMean;
z = reshape(z, [1,1,sz]);
zSampled = dlarray(z, 'SSCB');
end
```

The `ELBOloss` function takes the encodings of the means and the variances returned by the `sampling` function, and uses them to compute the ELBO loss.

```
function elbo = ELBOloss(x, xPred, zMean, zLogvar)
squares = 0.5*(xPred-x).^2;
reconstructionLoss  = sum(squares, [1,2,3]);

KL = -.5 * sum(1 + zLogvar - zMean.^2 - exp(zLogvar), 1);

elbo = mean(reconstructionLoss + KL);
end
```

Visualization Functions

The `VisualizeReconstruction` function randomly chooses two images for each digit of the MNIST data set, passes them through the VAE, and plots the reconstruction side by side with the original input. Note that to plot the information contained inside a `dlarray` object, you need to extract it first using the `extractdata` and `gather` functions.

```
function visualizeReconstruction(XTest,YTest, encoderNet, decoderNet)
f = figure;
figure(f)
title("Example ground truth image vs. reconstructed image")
for i = 1:2
    for c=0:9
        idx = iRandomIdxOfClass(YTest,c);
        X = XTest(:,:,:,idx);

        [z, ~, ~] = sampling(encoderNet, X);
        XPred = sigmoid(forward(decoderNet, z));

        X = gather(extractdata(X));
        XPred = gather(extractdata(XPred));

        comparison = [X, ones(size(X,1),1), XPred];
        subplot(4,5,(i-1)*10+c+1), imshow(comparison,[]),
    end
end
end

function idx = iRandomIdxOfClass(T,c)
idx = T == categorical(c);
idx = find(idx);
idx = idx(randi(numel(idx),1));
end
```

The `VisualizeLatentSpace` function visualizes the latent space defined by the mean and the variance matrices that form the output of the encoder network, and locates the clusters formed by the latent space representations of each digit.

The function starts by extracting the mean and the variance matrices from the `dlarray` objects. Because transposing a matrix with channel/batch dimensions (C and B) is not possible, the function calls `stripdims` before transposing the matrices. Then, it carries out a principal component analysis (PCA) on both matrices. To visualize the latent space in two dimensions, the function keeps the first two principal components and plots them against each other. Finally, the function colors the digit classes so that you can observe clusters.

```
function visualizeLatentSpace(XTest, YTest, encoderNet)
[~, zMean, zLogvar] = sampling(encoderNet, XTest);

zMean = stripdims(zMean)';
zMean = gather(extractdata(zMean));

zLogvar = stripdims(zLogvar)';
zLogvar = gather(extractdata(zLogvar));

[~,scoreMean] = pca(zMean);
[~,scoreLogvar] = pca(zLogvar);
```

```
c = parula(10);
f1 = figure;
figure(f1)
title("Latent space")

ah = subplot(1,2,1);
scatter(scoreMean(:,2),scoreMean(:,1),[],c(double(YTest),:));
ah.YDir = 'reverse';
axis equal
xlabel("Z_m_u(2)")
ylabel("Z_m_u(1)")
cb = colorbar; cb.Ticks = 0:(1/9):1; cb.TickLabels = string(0:9);

ah = subplot(1,2,2);
scatter(scoreLogvar(:,2),scoreLogvar(:,1),[],c(double(YTest),:));
ah.YDir = 'reverse';
xlabel("Z_v_a_r(2)")
ylabel("Z_v_a_r(1)")
cb = colorbar;  cb.Ticks = 0:(1/9):1; cb.TickLabels = string(0:9);
axis equal
end
```

The `generate` function tests the generative capabilities of the VAE. It initializes a `dlarray` object containing 25 randomly generated encodings, passes them through the decoder network, and plots the outputs.

```
function generate(decoderNet, latentDim)
randomNoise = dlarray(randn(1,1,latentDim,25),'SSCB');
generatedImage = sigmoid(predict(decoderNet, randomNoise));
generatedImage = extractdata(generatedImage);

f3 = figure;
figure(f3)
imshow(imtile(generatedImage, "ThumbnailSize", [100,100]))
title("Generated samples of digits")
drawnow
end
```

See Also

`dlnetwork` | `layerGraph` | `dlarray` | `adamupdate` | `dlfeval` | `dlgradient` | `sigmoid`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Automatic Differentiation Background” on page 18-200

Lane and Vehicle Detection in Simulink Using Deep Learning

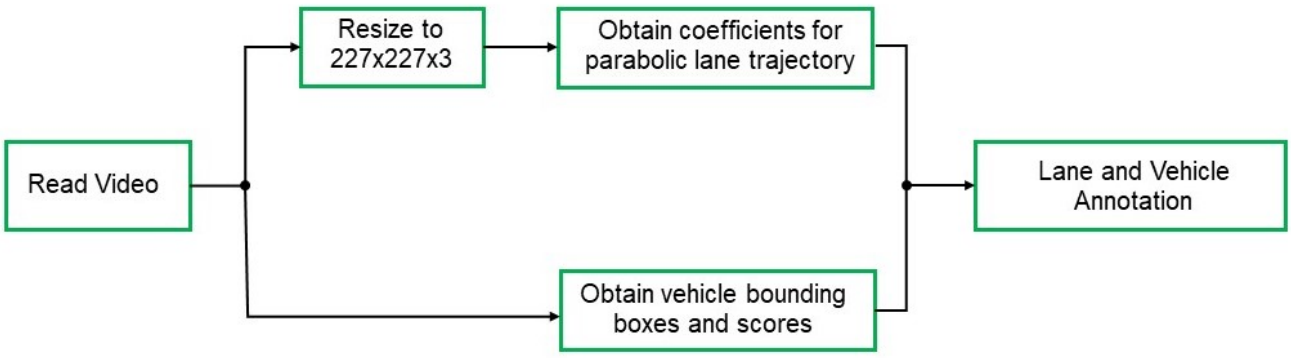
This example shows how to use deep convolutional neural networks inside a Simulink® model to perform lane and vehicle detection. This example takes the frames from a traffic video as an input, outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle, and detects vehicles in the frame.

This example uses the pretrained lane detection network from the *Lane Detection Optimized with GPU Coder* example of the GPU Coder Toolbox™. For more information, see "Lane Detection Optimized with GPU Coder" (GPU Coder).

This example also uses the pretrained vehicle detection network from the *Object Detection Using YOLO v2 Deep Learning* example of the Computer Vision toolbox™. For more information, see "Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox).

Algorithmic Workflow

The block diagram for the algorithmic workflow of the Simulink model is shown.



Get Pretrained Lane and Vehicle Detection Networks

This example uses the `trainedLaneNet` and `yolov2ResNet50VehicleExample` MAT-files containing the pretrained networks. The files are approximately 143MB and 98MB in size, respectively. Download the files from the MathWorks website.

```
lanenetFile = matlab.internal.examples.downloadSupportFile('gpucoder/cnn_models/lane_detection',
vehiclenetFile = matlab.internal.examples.downloadSupportFile('vision/data','yolov2ResNet50Vehic
```

Download Test Traffic Video

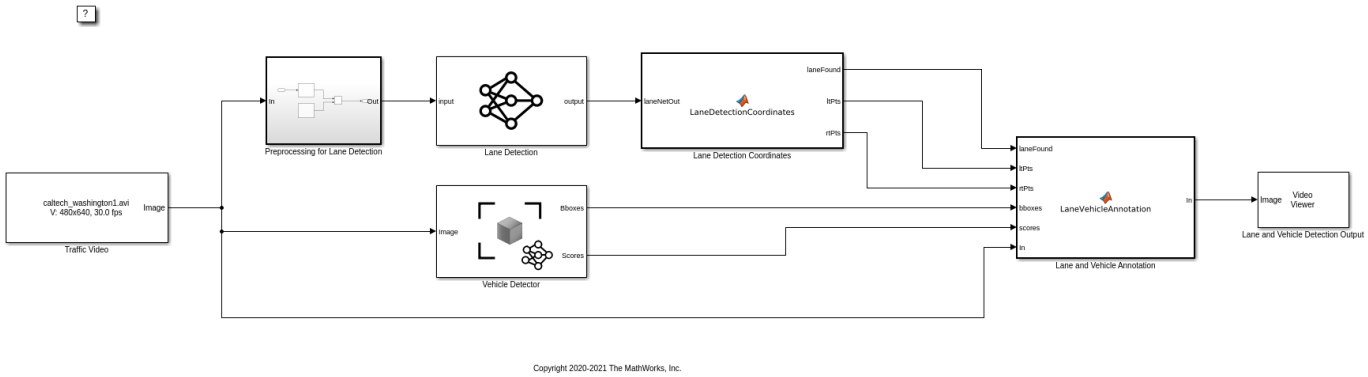
To test the model, the example uses the Caltech lanes dataset. The file is approximately 16 MB in size. Download the files from the MathWorks website.

```
mediaFile = matlab.internal.examples.downloadSupportFile('gpucoder/media','caltech_washington1.av
```

Lane and Vehicle Detection Simulink Model

The Simulink model for performing lane and vehicle detection on the traffic video is shown. When the model runs, the `Video Viewer` block displays the traffic video with lane and vehicle annotations.

```
open_system('laneAndVehicleDetectionMDL');
```



Set the file paths of the dowloaded network model in the predict and detector blocks of the Simulink model. Set the location of the test video to be loaded by the Simulink model.

```
set_param('laneAndVehicleDetectionMDL/Lane Detection','NetworkFilePath',lanenetFile)
set_param('laneAndVehicleDetectionMDL/Vehicle Detector','DetectorFilePath',vehiclenetFile)
set_param('laneAndVehicleDetectionMDL/Traffic Video','inputFileName',mediaFile)
```

Lane Detection

For lane detection, the traffic video is preprocessed by resizing each frame of the video to 227-by-227-by-3 and then scaled by a factor of 255. The preprocessed frames are then input to the `trainedLaneNet.mat` network loaded in the `Predict` block from the Deep Learning Toolbox™. This network takes an image as an input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by the parabolic equation:

$$y = ax^2 + bx + c$$

Here y is the lateral offset and x is the longitudinal distance from the vehicle. The network outputs the three parameters a, b, and c per lane. The network architecture is similar to `AlexNet` except that the last few layers are replaced by a smaller fully connected layer and regression output layer. The `Lane Detection Coordinates` MATLAB function block defines a function `lane_detection_coordinates` that takes the output from the predict block and outputs three parameters; `laneFound`, `ltPts`, and `rtPts`. Thresholding is used to determine if both left and right lane boundaries are both found. If both are found, `laneFound` is set to be true and the trajectories of the boundaries are calculated and stored in `ltPts` and `rtPts` respectively.

```
type lane_detection_coordinates
```

```
function [laneFound,ltPts,rtPts] = lane_detection_coordinates(laneNetOut)

% Copyright 2020-2021 The MathWorks, Inc.

persistent laneCoeffMeans;
if isempty(laneCoeffMeans)
    laneCoeffMeans = [-0.0002,0.0002,1.4740,-0.0002,0.0045,-1.3787];
end

persistent laneCoeffStds;
if isempty(laneCoeffStds)
    laneCoeffStds = [0.0030,0.0766,0.6313,0.0026,0.0736,0.9846];
end
```

```
params = laneNetOut .* laneCoeffStds + laneCoeffMeans;

% 'c' should be more than 0.5 for it to be a right lane
isRightLaneFound = abs(params(6)) > 0.5;
isLeftLaneFound =  abs(params(3)) > 0.5;

persistent vehicleXPoints;
if isempty(vehicleXPoints)
    vehicleXPoints = 3:30; %meters, ahead of the sensor
end

ltPts = coder.nullcopy(zeros(28,2,'single'));
rtPts = coder.nullcopy(zeros(28,2,'single'));

if isRightLaneFound && isLeftLaneFound
    rtBoundary = params(4:6);
    rt_y = computeBoundaryModel(rtBoundary, vehicleXPoints);
    ltBoundary = params(1:3);
    lt_y = computeBoundaryModel(ltBoundary, vehicleXPoints);

    % Visualize lane boundaries of the ego vehicle
    tform = get_tformToImage;
    % Map vehicle to image coordinates
    ltPts =  tform.transformPointsInverse([vehicleXPoints', lt_y']);
    rtPts =  tform.transformPointsInverse([vehicleXPoints', rt_y']);
    laneFound = true;
else
    laneFound = false;
end

end
```

Vehicle Detection

This example uses a YOLO v2 based network for vehicle detection. A YOLO v2 object detection network is composed of two subnetworks: a feature extraction network followed by a detection network. This pretrained network uses a `ResNet-50` for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to YOLO v2.

The Simulink model performs vehicle detection using the `Object Detector` block from the Computer Vision Toolbox. This block takes an image as input and outputs the bounding box coordinates along with the confidence scores for vehicles in the image.

Annotation of Vehicle Bounding Boxes and Lane Trajectory in Traffic Video

The `Lane and Vehicle Annotation` MATLAB function block defines a function `lane_vehicle_annotation` which annotates the vehicle bounding boxes along with the confidence scores. If `laneFound` is true, then the left and right lane boundaries stored in `ltPts` and `rtPts` are overlayed on the traffic video.

```
type lane_vehicle_annotation
```

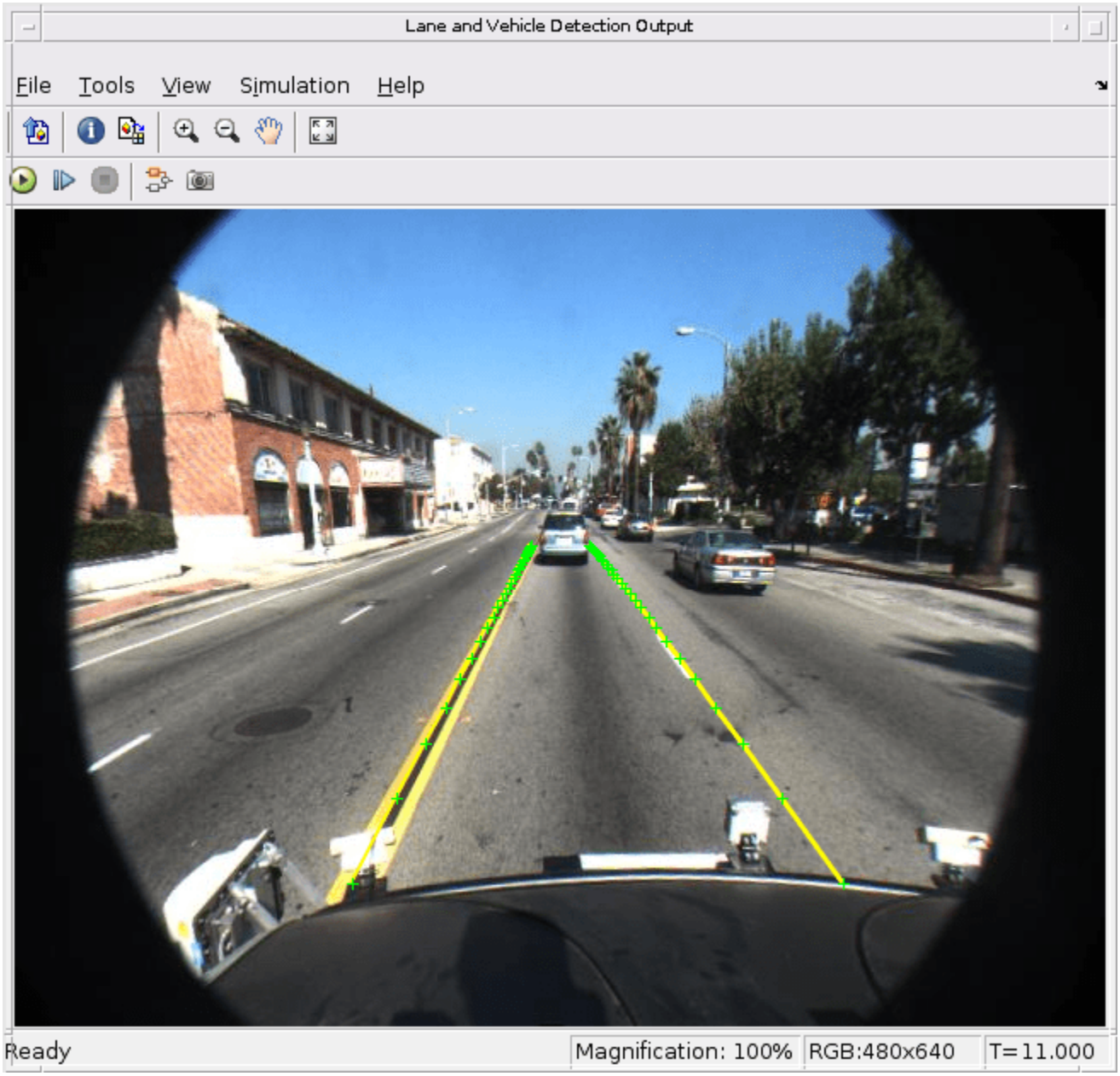
```
function In = lane_vehicle_annotation(laneFound,ltPts,rtPts,bboxes,scores,In)
```

```
% Copyright 2020-2021 The MathWorks, Inc.

if ~isempty(bboxes)
    In = insertObjectAnnotation(In, 'rectangle',bboxes,scores);
end

pts = coder.nullcopy(zeros(28, 4, 'single'));
if laneFound
    prevpt =  [ltPts(1,1) ltPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt ltPts(k,1) ltPts(k,2)];
        prevpt = [ltPts(k,1) ltPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    prevpt =  [rtPts(1,1) rtPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt rtPts(k,1) rtPts(k,2)];
        prevpt = [rtPts(k,1) rtPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    In = insertMarker(In, ltPts);
    In = insertMarker(In, rtPts);
end

end
```

Run the Simulation

To verify the lane and vehicle detection algorithms and display the lane trajectories, vehicle bounding boxes and scores for the traffic video loaded in the Simulink model, run the simulation.

```
set_param('laneAndVehicleDetectionMDL', 'SimulationMode', 'Normal');
sim('laneAndVehicleDetectionMDL');
```



On Windows®, the maximum path length of 260 characters can cause `"File not found"` errors when running the simulation. In such cases, move the example folder to a different location or enable long paths in Windows. For more information, see Maximum Path Length Limitation (Microsoft).

Use Deep Learning Accelerator Libraries

If you have an Intel® CPU that supports AVX2 instructions, you can use the *MATLAB Coder Interface for Deep Learning Libraries* to accelerate the simulation using Intel MKL-DNN libraries. In the Model Configuration Parameters window, on the Simulation Target pane, set the Language to `C++` and the Target library to `MKL-DNN`.

Code Generation

With GPU Coder, you can accelerate the execution of model on NVIDIA® GPUs and generate CUDA® code for model. For more information, see “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” (GPU Coder).

Classify ECG Signals in Simulink Using Deep Learning

This example shows how to use wavelet transforms and a deep learning network within a Simulink (R) model to classify ECG signals. This example uses the pretrained convolutional neural network from the *Classify Time Series Using Wavelet Analysis and Deep Learning* example of the Wavelet Toolbox™ to classify ECG signals based on images from the CWT of the time series data. For information on training, see “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox).

ECG Data Description

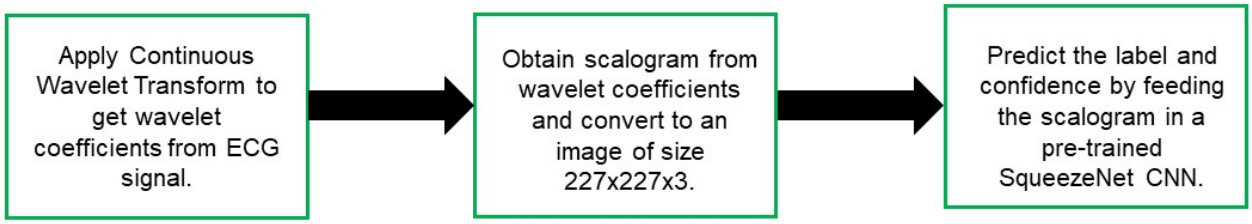
This example uses ECG data from PhysioNet database. It contains data from three groups of people:

1. Persons with cardiac arrhythmia (ARR)
2. Persons with congestive heart failure (CHF)
3. Persons with normal sinus rhythms (NSR)

It includes 96 recordings from persons with ARR, 30 recordings from persons with CHF, and 36 recordings from persons with NSR. The `ecg_signals` MAT-file contains the test ECG data in time series format. The image classifier in this example distinguishes between ARR, CHF, and NSR.

Algorithmic Workflow

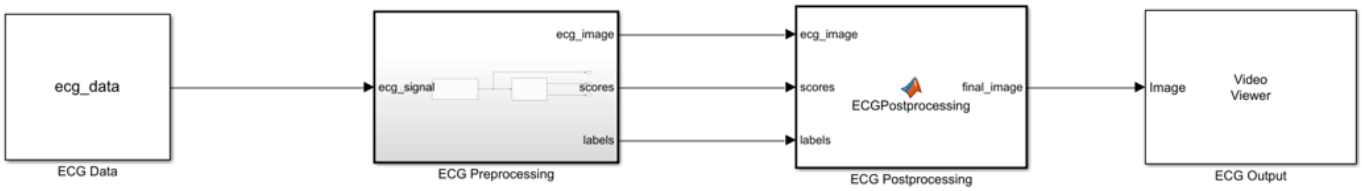
The block diagram for the algorithmic workflow of the Simulink model is shown.



ECG Deep Learning Simulink Model

The Simulink model for classifying the ECG signals is shown. When the model runs, the `Video Viewer` block displays the classified ECG signal.

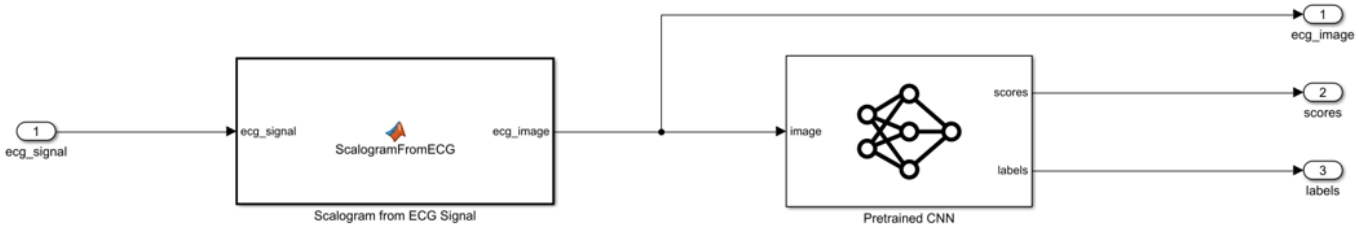
```
open_system('ecg_dl_cwtMDL');
```



ECG Preprocessing Subsystem

The `ECG Preprocessing` subsystem contains a `MATLAB Function` block that performs CWT to obtain scalogram of the ECG signal and then processes the scalogram to obtain an image. It also contains an `Image Classifier` block from the Deep Learning Toolbox™ that loads the pretrained network from `trainedNet.mat` and performs prediction for image classification based on SqueezeNet deep learning CNN.

```
open_system('ecg_dl_cwtMDL/ECG Preprocessing');
```



The `ScalogramFromECG` function block defines a function called `ecg_to_scalogram` that:

- Uses 65536 samples of double-precision ECG data as input.
- Create time frequency representation from the ECG data by applying Wavelet transform.
- Obtain scalogram from the wavelet coefficients.
- Convert the scalogram to image of size (227-by-227-by-3).

The function signature of `ecg_to_scalogram` is shown.

```
type ecg_to_scalogram
```

```
function ecg_image  = ecg_to_scalogram(ecg_signal)

% Copyright 2020 The MathWorks, Inc.

persistent jetdata;
if(isempty(jetdata))
    jetdata = colourmap(128,'single');
end
% Obtain wavelet coefficients from ECG signal
cfs = cwt_ecg(ecg_signal);
% Obtain scalogram from wavelet coefficients
image = ind2rgb(im2uint8(rescale(cfs)),jetdata);
ecg_image = im2uint8(imresize(image,[227,227]));

end
```

ECG Postprocessing

The `ECG Postprocessing` MATLAB function block defines the `label_prob_image` function that finds the label for the scalogram image based on the highest score from the scores outputed by the image classifier. It outputs the scalogram image with the label and confidence overlayed.

```
type label_prob_image
```

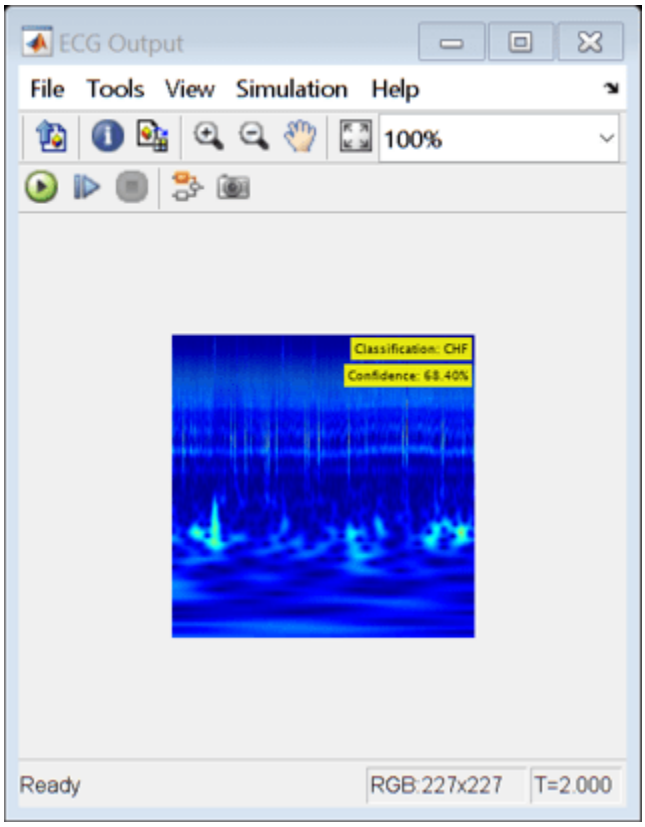
```
function final_image = label_prob_image(ecg_image, scores, labels)
```

```
% Copyright 2020 The MathWorks, Inc.

scores = double(scores);
% Obtain maximum confidence
[prob,index] = max(scores);
confidence = prob*100;
% Obtain label corresponding to maximum confidence
label = erase(char(labels(index)),'_label');
text = cell(2,1);
text{1} = ['Classification: ' label];
text{2} = ['Confidence: ' sprintf('%0.2f',confidence) '%'];
position = [135 20 0 0; 130 40 0 0];
final_image = insertObjectAnnotation(ecg_image,'rectangle',position,text,'TextBoxOpacity',0.9,'Fo

end
```

Run the Simulation

To verify the algorithm and display the labels and confidence score of the test ECG signal loaded in the workspace, run the simulation.

```
set_param('ecg_dl_cwtMDL', 'SimulationMode', 'Normal');
sim('ecg_dl_cwtMDL');
```



Code Generation

With GPU Coder™, you can accelerate the execution of model on NVIDIA® GPUs and generate CUDA® code for model. See the “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” (GPU Coder) for more details.

Cleanup

Close the Simulink model.

```
close_system('ecg_dl_cwtMDL/ECG Preprocessing');
close_system('ecg_dl_cwtMDL');
```

Classify Images in Simulink Using GoogLeNet

This example shows how to classify an image in Simulink® using the `Image Classifier` block. The example uses the pretrained deep convolutional neural network GoogLeNet to perform the classification.

Pretrained GoogLeNet Network

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

```
net = googlenet;
inputSize = net.Layers(1).InputSize;
classNames = net.Layers(end).ClassNames;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
    {'speedboat'    }
    {'window screen'}
    {'isopod'       }
    {'wooden spoon' }
    {'lipstick'     }
    {'drake'        }
    {'hyena'        }
    {'dumbbell'     }
    {'strawberry'   }
    {'custard apple'}
```

Read and Resize Image

Read and show the image that you want to classify.

```
I = imread('peppers.png');
figure
imshow(I)
```

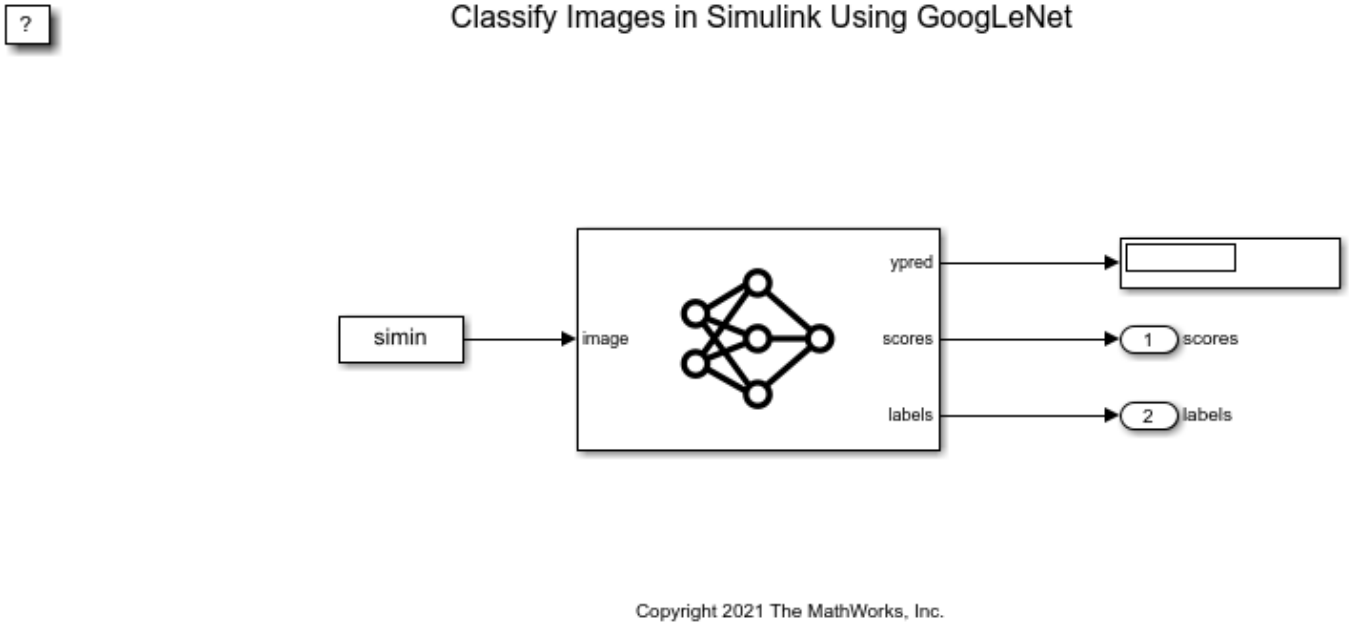
To import this data into the Simulink model, specify a structure variable containing the input image data and an empty time vector.

```
simin.time = [];
simin.signals.values = I;
simin.signals.dimensions = size(I);
```

Simulink Model for Prediction

The Simulink model for classifying images is shown. The model uses a `From Workspace` block to load the input image, an `Image Classifier` block from the Deep Neural Networks library that classifies the input, and `Display` block to show the predicted output.

```
model = 'googlenet_classifier';
open_system(model);
```

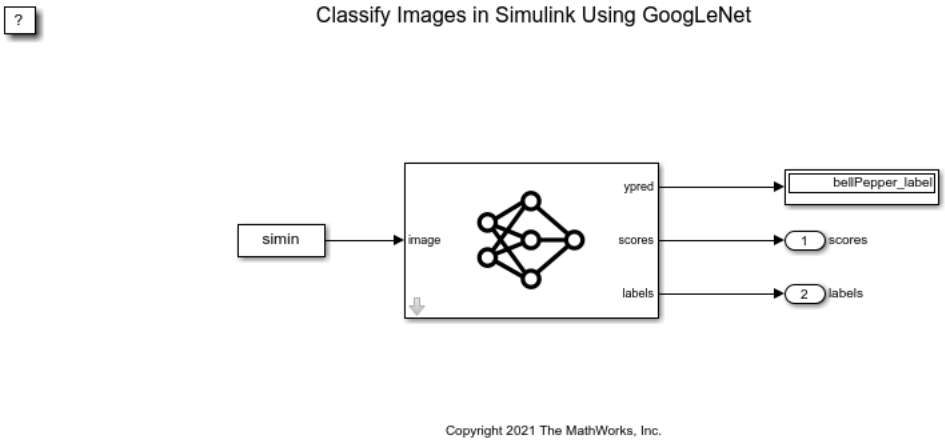


Run the Simulation

To validate the Simulink model, run the simulation.

```
set_param(model,'SimulationMode','Normal');
sim(model);
```

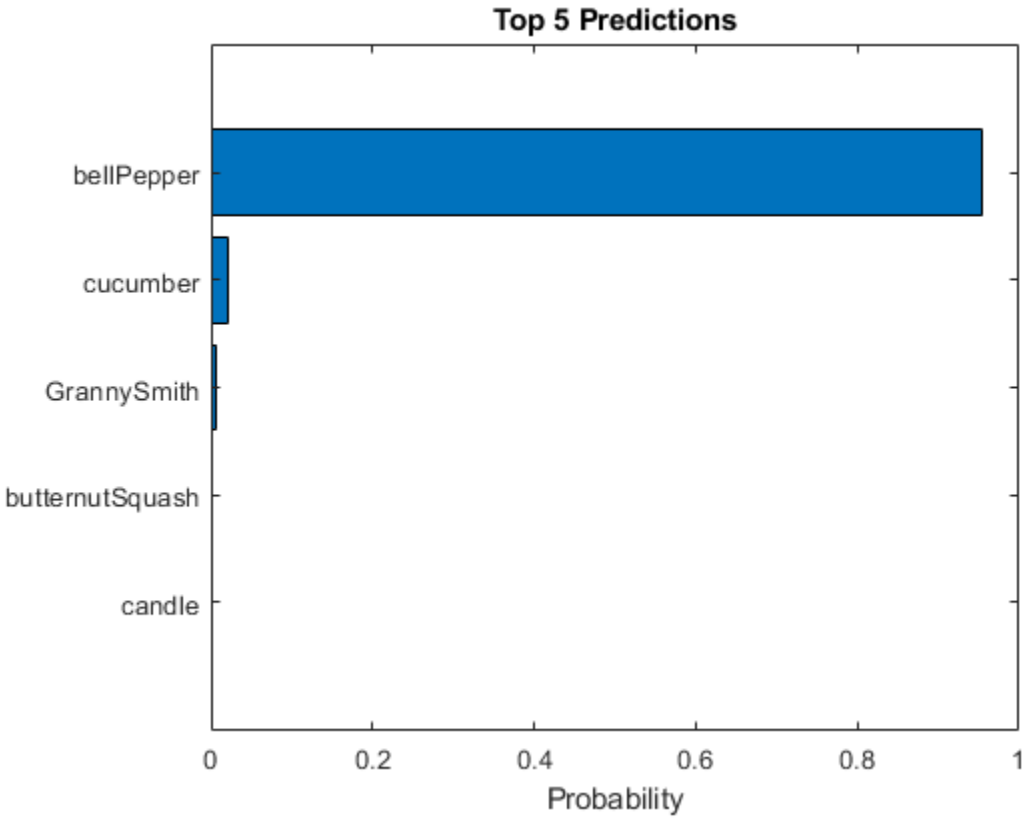
The network classifies the image as a bell pepper.



Display Top Predictions

Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
scores = yout.signals(1).values(:,:,1);
labels = yout.signals(2).values(:,:,1);
[~,idx] = sort(scores,'descend');
idx = idx(5:-1:1);
scoresTop = scores(idx);
labelsTop = split(string(labels(idx)),'_');
labelsTop = labelsTop(:,:,1);

figure
imshow(I)
title(labelsTop(5) + ", " + num2str(100*scoresTop(5) + "%"));

figure
barh(scoresTop)
xlim([0 1])
title('Top 5 Predictions')
xlabel('Probability')
yticklabels(labelsTop)
```


Top 5 Predictions
bellPepper
cucumber
GrannySmith
butternutSquash
candle
0
0.2
0.4
0.6
0.8
1
Probability

Deep Learning with Time Series, Sequences, and Text

Sequence Classification Using Deep Learning

This example shows how to classify sequence data using a long short-term memory (LSTM) network.

To train a deep neural network to classify sequence data, you can use an LSTM network. An LSTM network enables you to input sequence data into a network, and make predictions based on the individual time steps of the sequence data.

This example uses the Japanese Vowels data set as described in [1] and [2]. This example trains an LSTM network to recognize the speaker given time series data representing two Japanese vowels spoken in succession. The training data contains time series data for nine speakers. Each sequence has 12 features and varies in length. The data set contains 270 training observations and 370 test observations.

Load Sequence Data

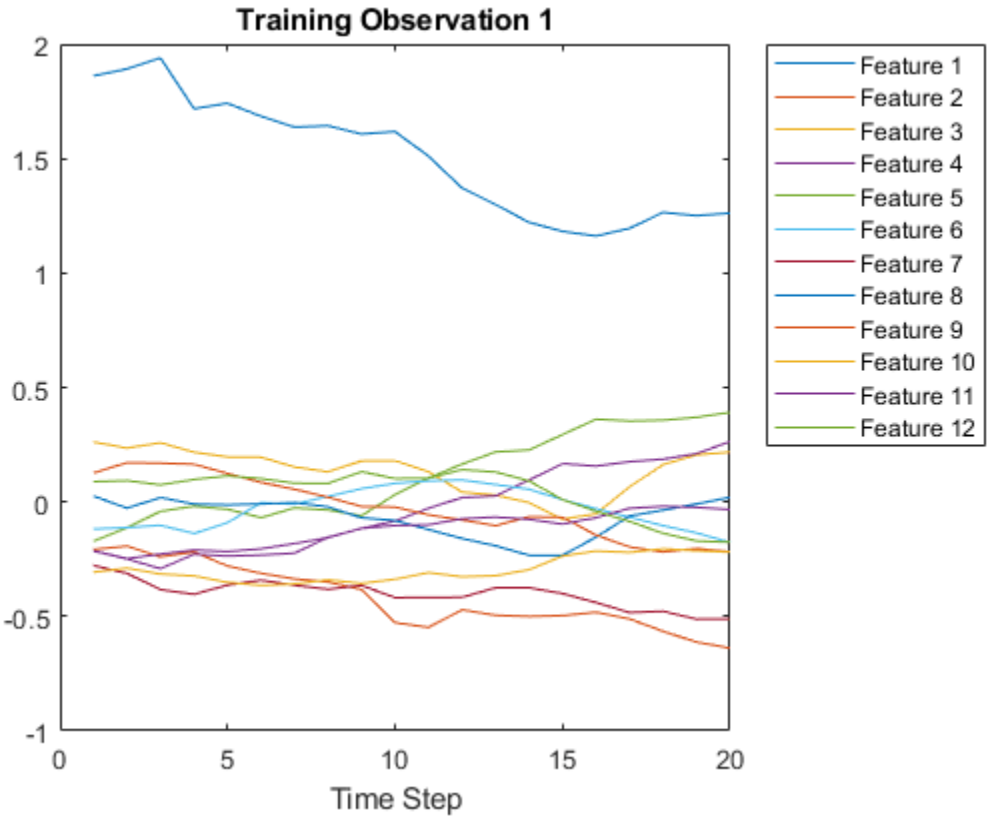
Load the Japanese Vowels training data. `XTrain` is a cell array containing 270 sequences of dimension 12 of varying length. `Y` is a categorical vector of labels "1","2",...,"9", which correspond to the nine speakers. The entries in `XTrain` are matrices with 12 rows (one row for each feature) and varying number of columns (one column for each time step).

```
[XTrain,YTrain] = japaneseVowelsTrainData;
XTrain(1:5)
```

```
ans=5×1 cell array
    {12x20 double}
    {12x26 double}
    {12x22 double}
    {12x20 double}
    {12x21 double}
```

Visualize the first time series in a plot. Each line corresponds to a feature.

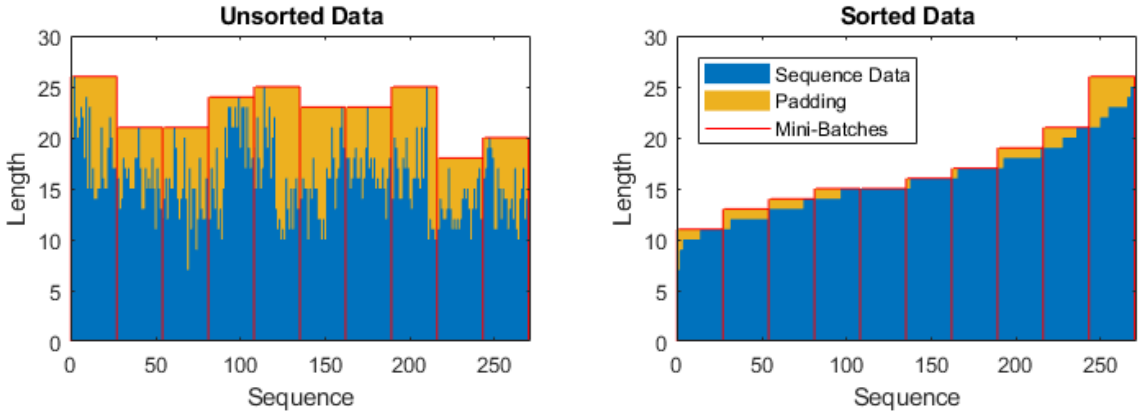
```
figure
plot(XTrain{1}')
xlabel("Time Step")
title("Training Observation 1")
numFeatures = size(XTrain{1},1);
legend("Feature " + string(1:numFeatures),'Location','northeastoutside')
```



Prepare Data for Padding

During training, by default, the software splits the training data into mini-batches and pads the sequences so that they have the same length. Too much padding can have a negative impact on the network performance.

To prevent the training process from adding too much padding, you can sort the training data by sequence length, and choose a mini-batch size so that sequences in a mini-batch have a similar length. The following figure shows the effect of padding sequences before and after sorting data.



Get the sequence lengths for each observation.

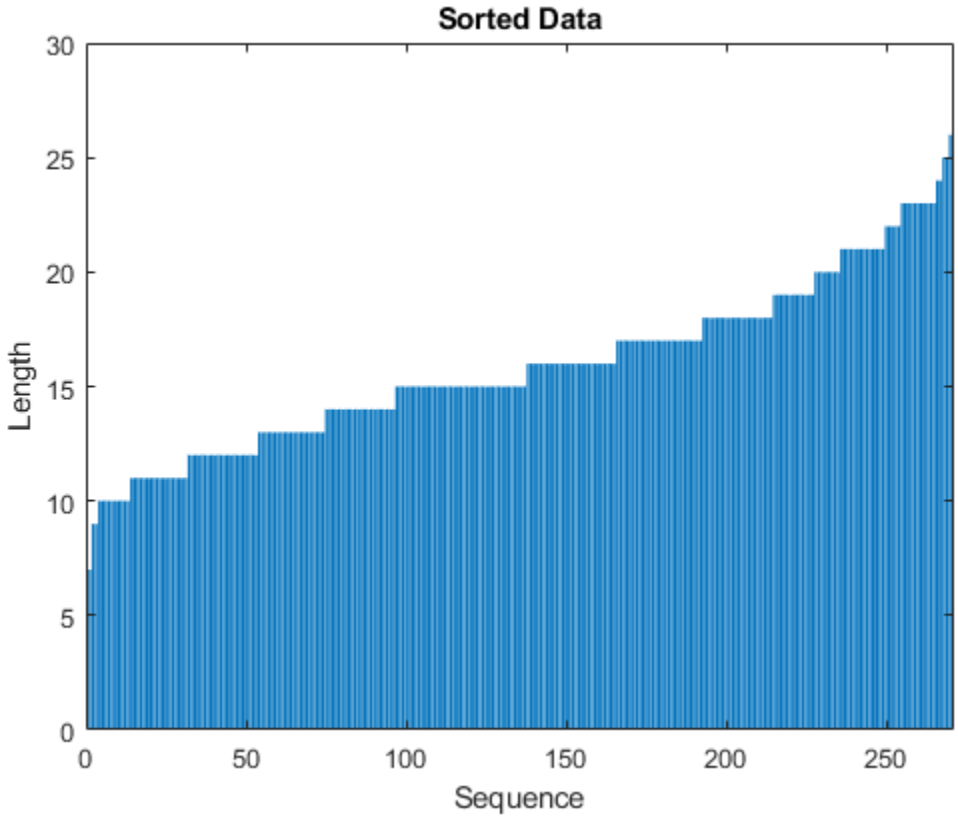
```
numObservations = numel(XTrain);
for i=1:numObservations
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end
```

Sort the data by sequence length.

```
[sequenceLengths,idx] = sort(sequenceLengths);
XTrain = XTrain(idx);
YTrain = YTrain(idx);
```

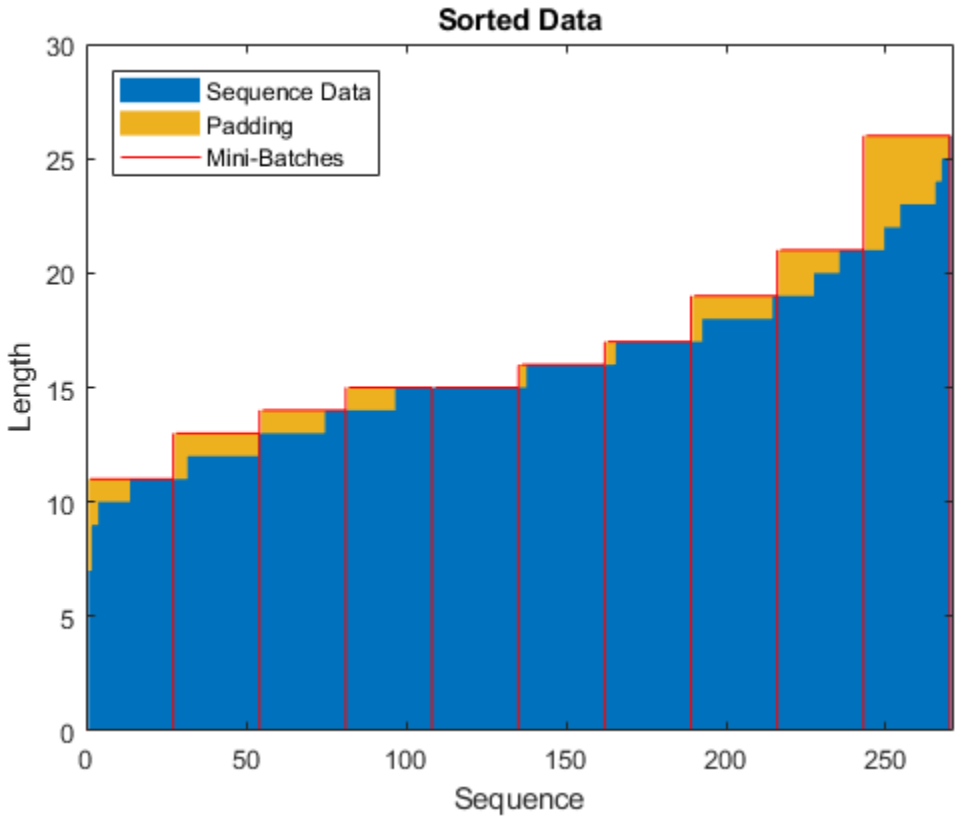
View the sorted sequence lengths in a bar chart.

```
figure
bar(sequenceLengths)
ylim([0 30])
xlabel("Sequence")
ylabel("Length")
title("Sorted Data")
```



Choose a mini-batch size of 27 to divide the training data evenly and reduce the amount of padding in the mini-batches. The following figure illustrates the padding added to the sequences.

```
miniBatchSize = 27;
```



Define LSTM Network Architecture

Define the LSTM network architecture. Specify the input size to be sequences of size 12 (the dimension of the input data). Specify an bidirectional LSTM layer with 100 hidden units, and output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

If you have access to full sequences at prediction time, then you can use a bidirectional LSTM layer in your network. A bidirectional LSTM layer learns from the full sequence at each time step. If you do not have access to the full sequence at prediction time, for example, if you are forecasting values or predicting one time step at a time, then use an LSTM layer instead.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(inputSize)
    bilstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 12 dimensions
```

```
     2   ''   BiLSTM                  BiLSTM with 100 hidden units
     3   ''   Fully Connected         9 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

Now, specify the training options. Specify the solver to be `'adam'`, the gradient threshold to be 1, and the maximum number of epochs to be 100. To reduce the amount of padding in the mini-batches, choose a mini-batch size of 27. To pad the data to have the same length as the longest sequences, specify the sequence length to be `'longest'`. To ensure that the data remains sorted by sequence length, specify to never shuffle the data.

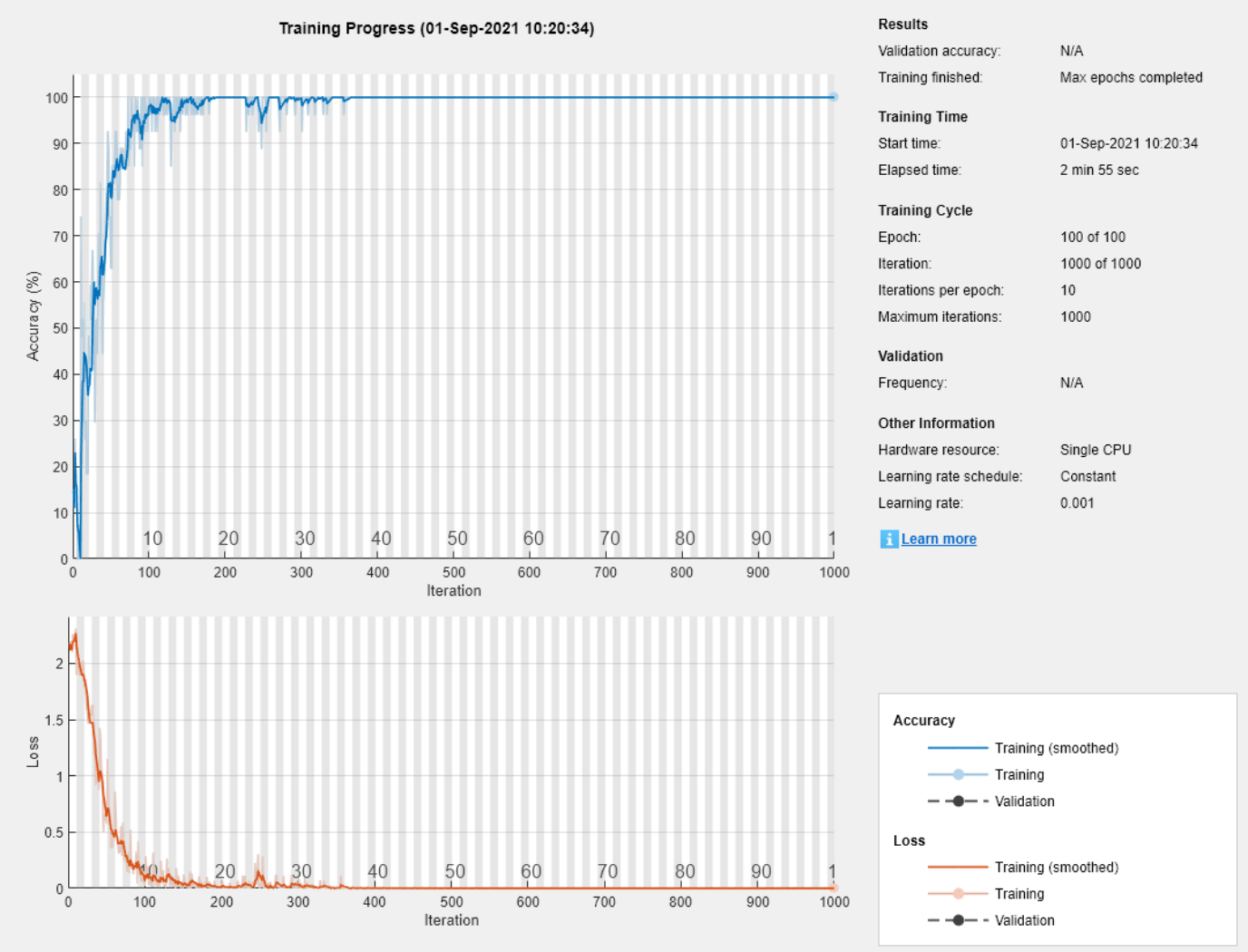
Since the mini-batches are small with short sequences, training is better suited for the CPU. Specify `'ExecutionEnvironment'` to be `'cpu'`. To train on a GPU, if available, set `'ExecutionEnvironment'` to `'auto'` (this is the default value).

```
maxEpochs = 100;
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'GradientThreshold',1, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','longest', ...
    'Shuffle','never', ...
    'Verbose',0, ...
    'Plots','training-progress');
```

Train LSTM Network

Train the LSTM network with the specified training options by using `trainNetwork`.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Test LSTM Network

Load the test set and classify the sequences into speakers.

Load the Japanese Vowels test data. `XTest` is a cell array containing 370 sequences of dimension 12 of varying length. `YTest` is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
XTest(1:3)
```

```
ans=3×1 cell array
    {12x19 double}
    {12x17 double}
    {12x19 double}
```

The LSTM network `net` was trained using mini-batches of sequences of similar length. Ensure that the test data is organized in the same way. Sort the test data by sequence length.

```
numObservationsTest = numel(XTest);
for i=1:numObservationsTest
    sequence = XTest{i};
    sequenceLengthsTest(i) = size(sequence,2);
end
[sequenceLengthsTest,idx] = sort(sequenceLengthsTest);
XTest = XTest(idx);
YTest = YTest(idx);
```

Classify the test data. To reduce the amount of padding introduced by the classification process, set the mini-batch size to 27. To apply the same padding as the training data, specify the sequence length to be `'longest'`.

```
miniBatchSize = 27;
YPred = classify(net,XTest, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','longest');
```

Calculate the classification accuracy of the predictions.

```
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9730
```

References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `bilstmLayer` | `sequenceInputLayer`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Sequence Classification Using 1-D Convolutions

This example shows how to classify sequence data using a 1-D convolutional neural network.

To train a deep neural network to classify sequence data, you can use a 1-D convolutional neural network. A 1-D convolutional layer learns features by applying sliding convolutional filters to 1-D input. Using 1-D convolutional layers can be faster than using recurrent layers because convolutional layers can process the input with a single operation. By contrast, recurrent layers must iterate over the time steps of the input. However, depending on the network architecture and filter sizes, 1-D convolutional layers might not perform as well as recurrent layers, which can learn long-term dependencies between time steps.

This example uses the Japanese Vowels data set described in [1] and [2]. This example trains a 1-D convolutional neural network to recognize the speaker given time series data representing two Japanese vowels spoken in succession. The training data contains time series data for nine speakers. Each sequence has 12 features and varies in length. The data set contains 270 training observations and 370 test observations.

Load Sequence Data

Load the Japanese Vowels training data. The predictor data is a cell array containing sequences of varying length with 12 features. The target data is a categorical vector of labels "1","2",...,"9", which correspond to the nine speakers. The predictor sequences are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

```
[XTrain,TTrain] = japaneseVowelsTrainData;
[XValidation,TValidation] = japaneseVowelsTestData;
```

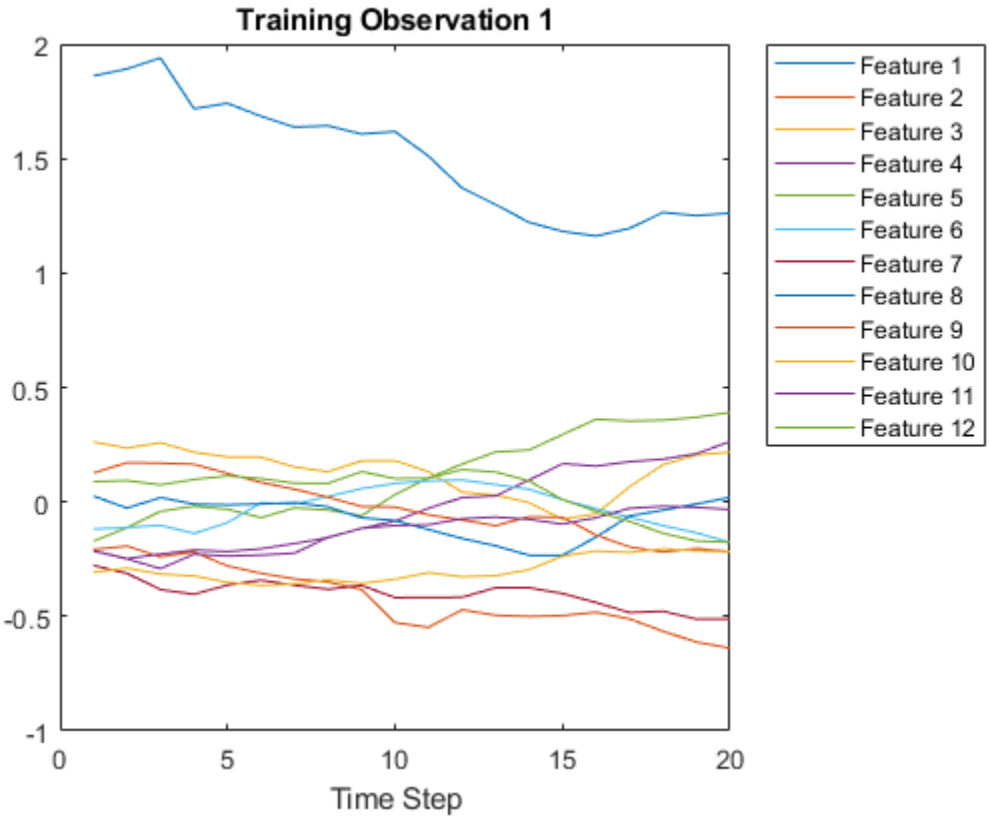
View the first few training sequences.

```
XTrain(1:5)
```

```
ans=5×1 cell array
    {12x20 double}
    {12x26 double}
    {12x22 double}
    {12x20 double}
    {12x21 double}
```

Visualize the first time series in a plot. Each line corresponds to a feature.

```
figure
plot(XTrain{1}')
xlabel("Time Step")
title("Training Observation 1")
numFeatures = size(XTrain{1},1);
legend("Feature " + string(1:numFeatures),Location="northeastoutside")
```



View the number of classes in the training data.

```
classes = categories(TTrain);
numClasses = numel(classes)
```

```
numClasses = 9
```

Define 1-D Convolutional Network Architecture

Define the 1-D convolutional neural network architecture.

- Specify the input size as the number of features of the input data.
- Specify two blocks of 1-D convolution, ReLU, and layer normalization layers, where the convolutional layer has a filter size of 3. Specify 32 and 64 filters for the first and second convolutional layers, respectively. For both convolutional layers, left-pad the inputs such that the outputs have the same length (causal padding).
- To reduce the output of the convolutional layers to a single vector, use a 1-D global average pooling layer.
- To map the output to a vector of probabilities, specify a fully connected layer with an output size matching the number of classes, followed by a softmax layer and a classification layer.

```
filterSize = 3;
numFilters = 32;

layers = [ ...
    sequenceInputLayer(numFeatures)
```

```
    convolution1dLayer(filterSize,numFilters,Padding="causal")
    reluLayer
    layerNormalizationLayer
    convolution1dLayer(filterSize,2*numFilters,Padding="causal")
    reluLayer
    layerNormalizationLayer
    globalAveragePooling1dLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify Training Options

Specify the training options:

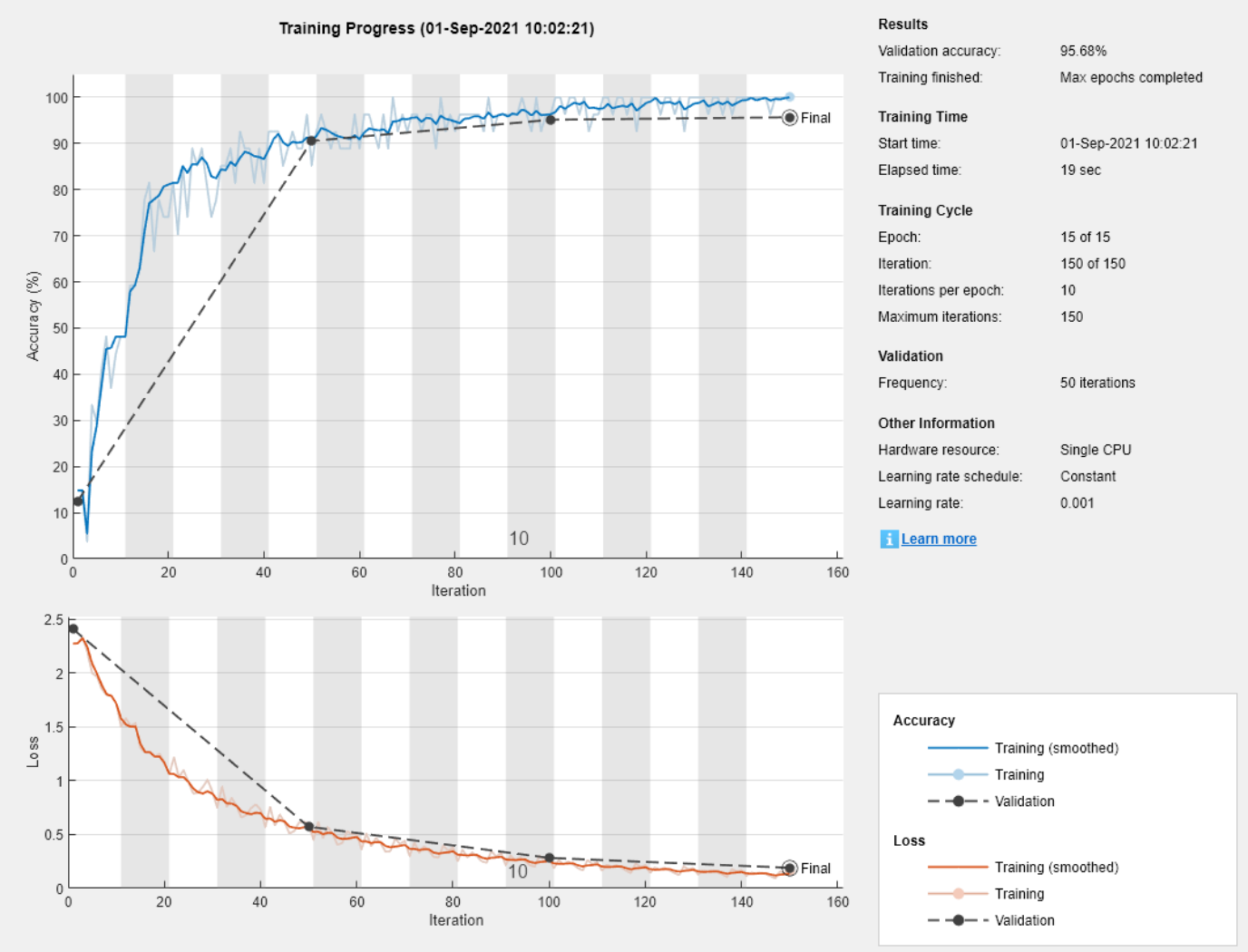
- Train using the Adam optimizer.
- Train with a mini-batch size of 27 for 15 epochs.
- Left-pad the sequences.
- Validate the network using the validation data.
- Monitor the training progress in a plot and suppress the verbose output.

```
miniBatchSize = 27;

options = trainingOptions("adam", ...
    MiniBatchSize=miniBatchSize, ...
    MaxEpochs=15, ...
    SequencePaddingDirection="left", ...
    ValidationData={XValidation,TValidation}, ...
    Plots="training-progress", ...
    Verbose=0);
```

Train Network

Train the network with the specified training options using the `trainNetwork` function.

```
net = trainNetwork(XTrain,TTrain,layers,options);
```



Test Network

Classify the validation data using the same mini-batch size and sequence padding options used for training.

```
YPred = classify(net,XValidation, ...
    MiniBatchSize=miniBatchSize, ...
    SequencePaddingDirection="left");
```

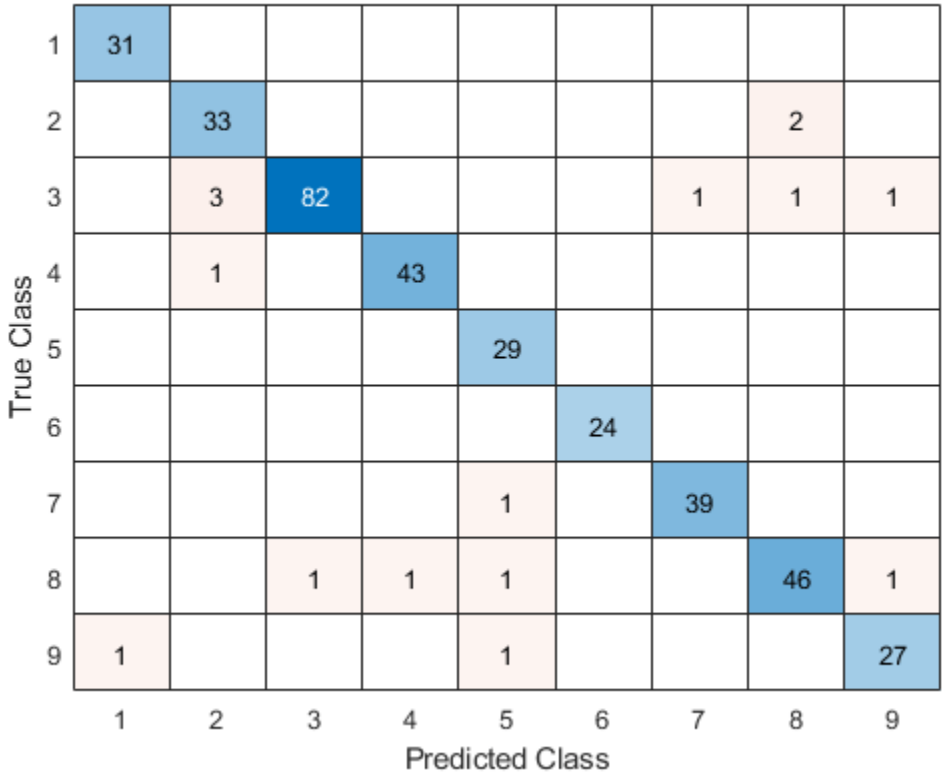
Calculate the classification accuracy of the predictions.

```
acc = mean(YPred == TValidation)
```

```
acc = 0.9568
```

Visualize the predictions in a confusion matrix.

```
confusionchart(TValidation,YPred)
```



References

[1] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. “Multidimensional Curve Classification Using Passing-through Regions.” Pattern Recognition Letters 20, no. 11–13 (November 1999): 1103–11. https://doi.org/10.1016/S0167-8655(99)00077-X

[2] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Japanese Vowels Data Set." Distributed by UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`convolution1dLayer` | `trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Related Examples

- “Sequence-to-Sequence Classification Using 1-D Convolutions” on page 4-79
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75

- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Time Series Forecasting Using Deep Learning

This example shows how to forecast time series data using a long short-term memory (LSTM) network.

To forecast the values of future time steps of a sequence, you can train a sequence-to-sequence regression LSTM network, where the responses are the training sequences with values shifted by one time step. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step.

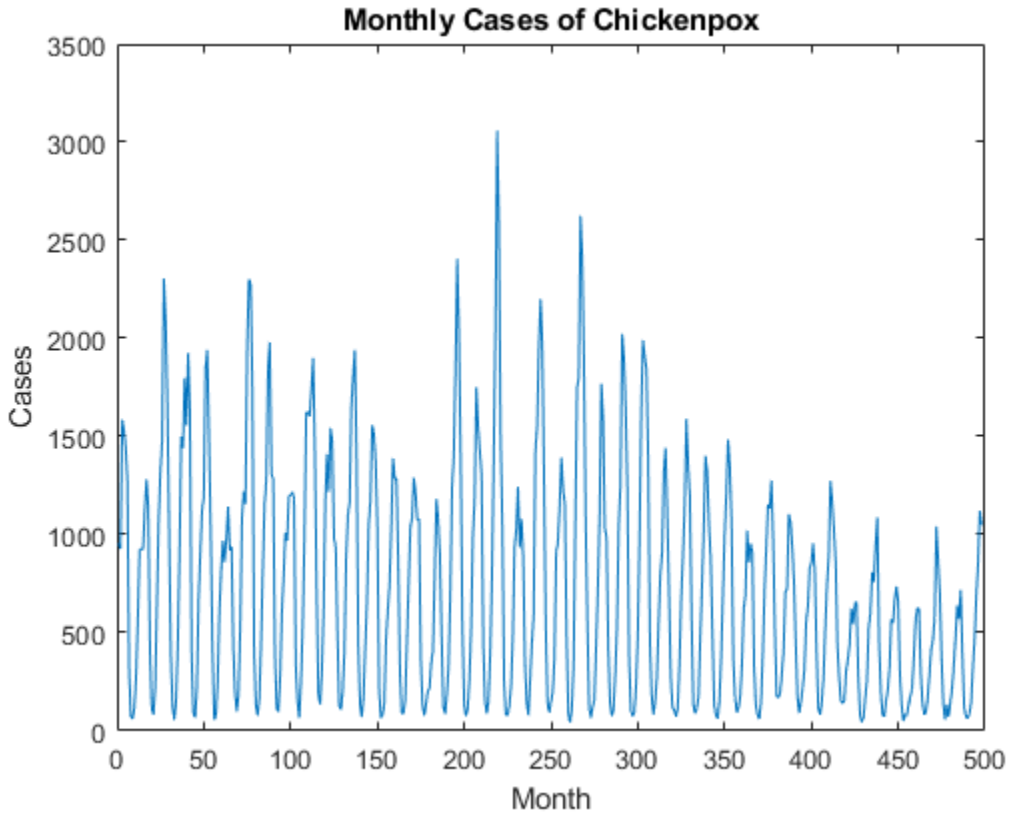
To forecast the values of multiple time steps in the future, use the `predictAndUpdateState` function to predict time steps one at a time and update the network state at each prediction.

This example uses the data set `chickenpox_dataset`. The example trains an LSTM network to forecast the number of chickenpox cases given the number of cases in previous months.

Load Sequence Data

Load the example data. `chickenpox_dataset` contains a single time series, with time steps corresponding to months and values corresponding to the number of cases. The output is a cell array, where each element is a single time step. Reshape the data to be a row vector.

```
data = chickenpox_dataset;
data = [data{:}];

figure
plot(data)
xlabel("Month")
ylabel("Cases")
title("Monthly Cases of Chickenpox")
```



Partition the training and test data. Train on the first 90% of the sequence and test on the last 10%.

```
numTimeStepsTrain = floor(0.9*numel(data));

dataTrain = data(1:numTimeStepsTrain+1);
dataTest = data(numTimeStepsTrain+1:end);
```

Standardize Data

For a better fit and to prevent the training from diverging, standardize the training data to have zero mean and unit variance. At prediction time, you must standardize the test data using the same parameters as the training data.

```
mu = mean(dataTrain);
sig = std(dataTrain);

dataTrainStandardized = (dataTrain - mu) / sig;
```

Prepare Predictors and Responses

To forecast the values of future time steps of a sequence, specify the responses to be the training sequences with values shifted by one time step. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step. The predictors are the training sequences without the final time step.

```
XTrain = dataTrainStandardized(1:end-1);
YTrain = dataTrainStandardized(2:end);
```

Define LSTM Network Architecture

Create an LSTM regression network. Specify the LSTM layer to have 200 hidden units.

```
numFeatures = 1;
numResponses = 1;
numHiddenUnits = 200;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

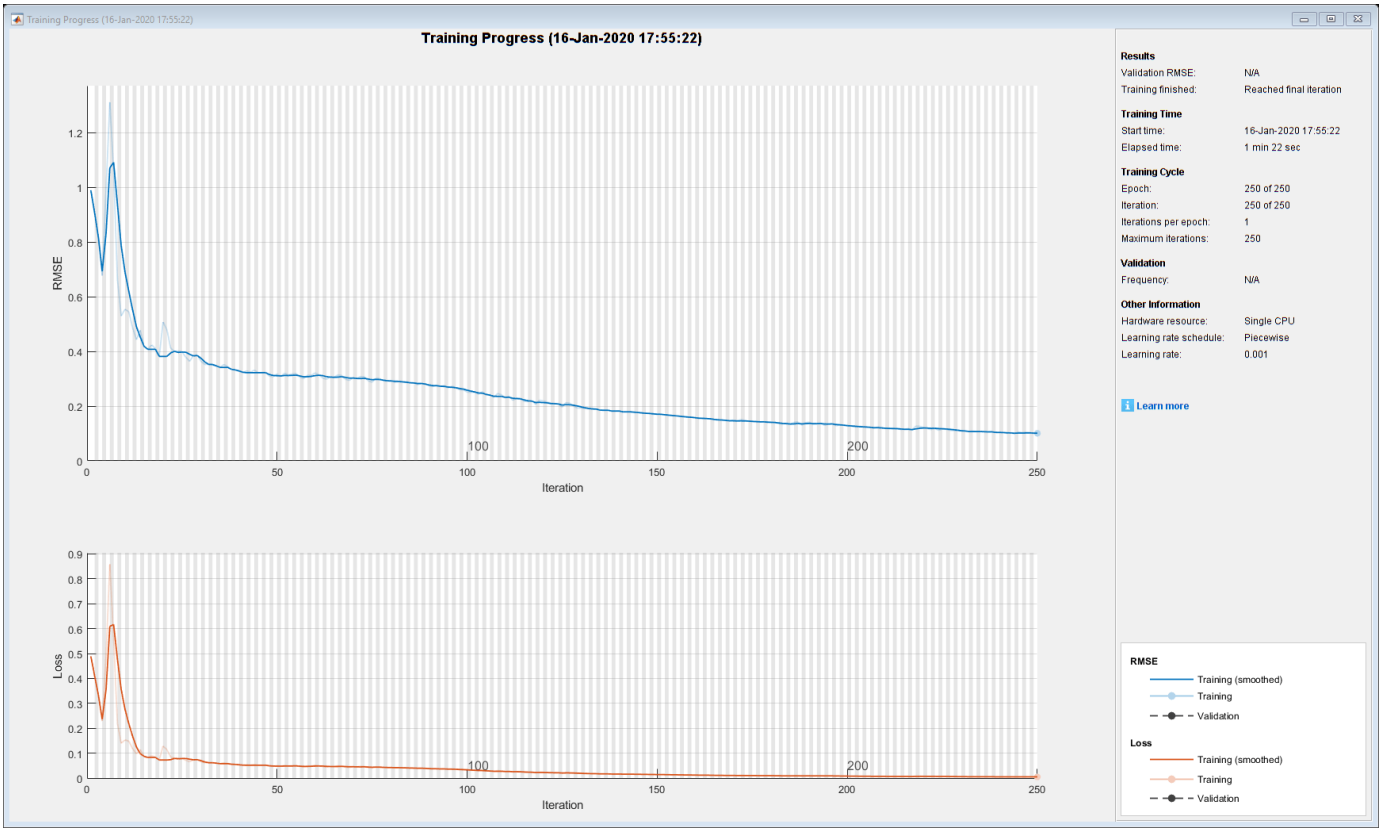
Specify the training options. Set the solver to `'adam'` and train for 250 epochs. To prevent the gradients from exploding, set the gradient threshold to 1. Specify the initial learn rate 0.005, and drop the learn rate after 125 epochs by multiplying by a factor of 0.2.

```
options = trainingOptions('adam', ...
    'MaxEpochs',250, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.005, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',125, ...
    'LearnRateDropFactor',0.2, ...
    'Verbose',0, ...
    'Plots','training-progress');
```

Train LSTM Network

Train the LSTM network with the specified training options by using `trainNetwork`.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Forecast Future Time Steps

To forecast the values of multiple time steps in the future, use the `predictAndUpdateState` function to predict time steps one at a time and update the network state at each prediction. For each prediction, use the previous prediction as input to the function.

Standardize the test data using the same parameters as the training data.

```
dataTestStandardized = (dataTest - mu) / sig;
XTest = dataTestStandardized(1:end-1);
```

To initialize the network state, first predict on the training data `XTrain`. Next, make the first prediction using the last time step of the training response `YTrain(end)`. Loop over the remaining predictions and input the previous prediction to `predictAndUpdateState`.

For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
net = predictAndUpdateState(net,XTrain);
[net,YPred] = predictAndUpdateState(net,YTrain(end));

numTimeStepsTest = numel(XTest);
for i = 2:numTimeStepsTest
    [net,YPred(:,i)] = predictAndUpdateState(net,YPred(:,i-1),'ExecutionEnvironment','cpu');
end
```

Unstandardize the predictions using the parameters calculated earlier.

```
YPred = sig*YPred + mu;
```

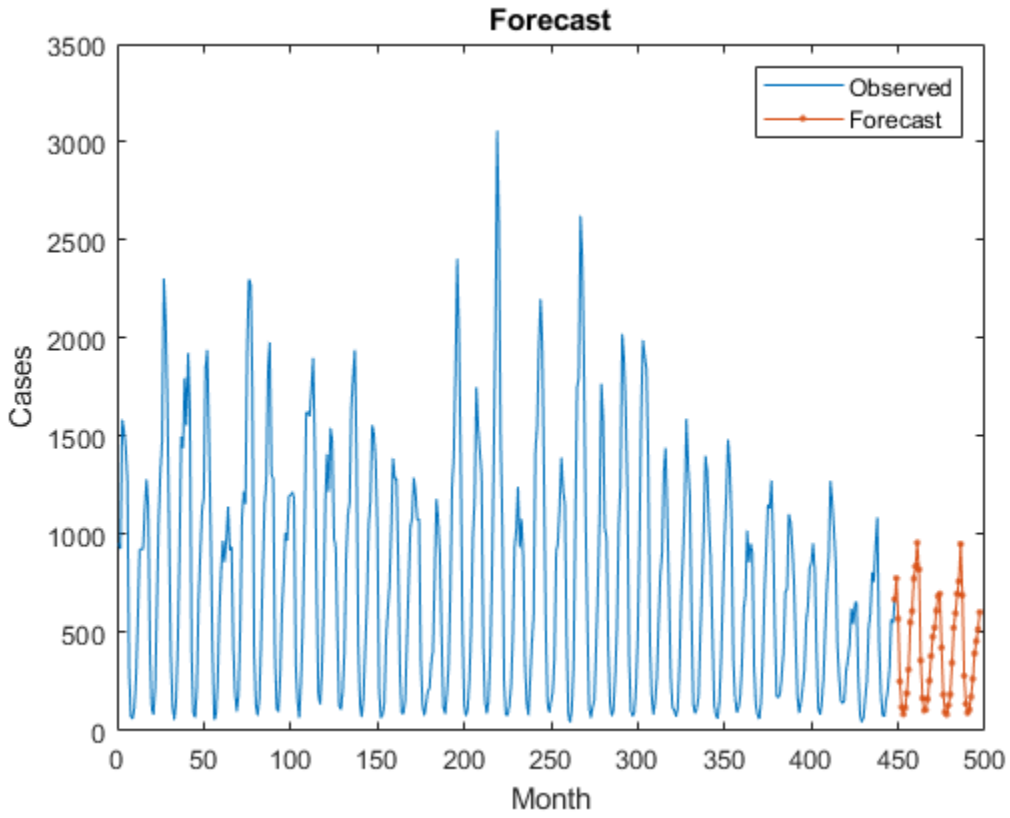
The training progress plot reports the root-mean-square error (RMSE) calculated from the standardized data. Calculate the RMSE from the unstandardized predictions.

```
YTest = dataTest(2:end);
rmse = sqrt(mean((YPred-YTest).^2))
```

```
rmse = single
    248.5531
```

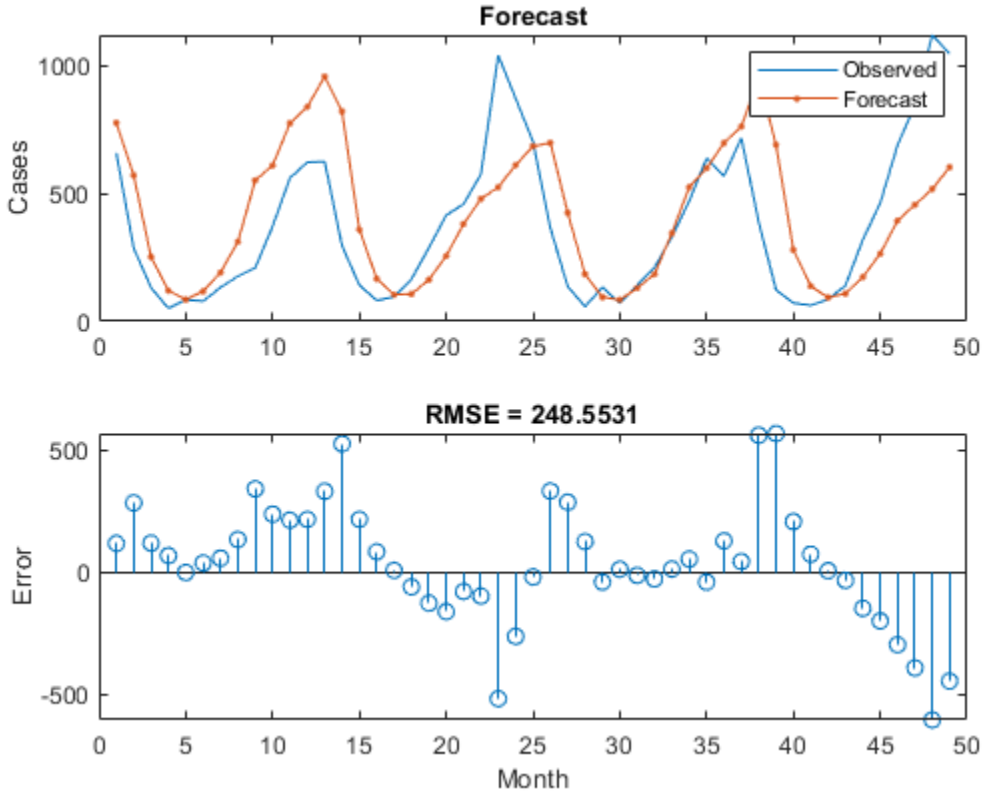
Plot the training time series with the forecasted values.

```
figure
plot(dataTrain(1:end-1))
hold on
idx = numTimeStepsTrain:(numTimeStepsTrain+numTimeStepsTest);
plot(idx,[data(numTimeStepsTrain) YPred],'.-')
hold off
xlabel("Month")
ylabel("Cases")
title("Forecast")
legend(["Observed" "Forecast"])
```



Compare the forecasted values with the test data.

```
figure
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
legend(["Observed" "Forecast"])
ylabel("Cases")
title("Forecast")

subplot(2,1,2)
stem(YPred - YTest)
xlabel("Month")
ylabel("Error")
title("RMSE = " + rmse)
```



Update Network State with Observed Values

If you have access to the actual values of time steps between predictions, then you can update the network state with the observed values instead of the predicted values.

First, initialize the network state. To make predictions on a new sequence, reset the network state using `resetState`. Resetting the network state prevents previous predictions from affecting the predictions on the new data. Reset the network state, and then initialize the network state by predicting on the training data.

```
net = resetState(net);
net = predictAndUpdateState(net,XTrain);
```

Predict on each time step. For each prediction, predict the next time step using the observed value of the previous time step. Set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
YPred = [];
numTimeStepsTest = numel(XTest);
for i = 1:numTimeStepsTest
    [net,YPred(:,i)] = predictAndUpdateState(net,XTest(:,i),'ExecutionEnvironment','cpu');
end
```

Unstandardize the predictions using the parameters calculated earlier.

```
YPred = sig*YPred + mu;
```

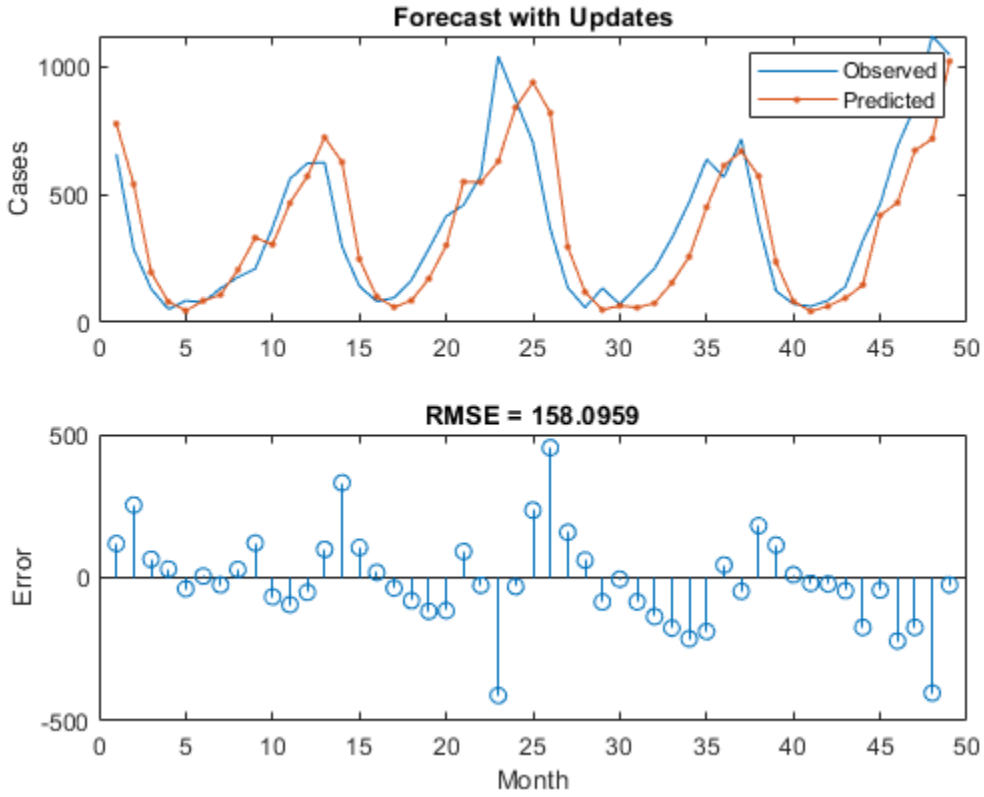
Calculate the root-mean-square error (RMSE).

```
rmse = sqrt(mean((YPred-YTest).^2))
```

```
rmse = 158.0959
```

Compare the forecasted values with the test data.

```
figure
subplot(2,1,1)
plot(YTest)
hold on
plot(YPred,'.-')
hold off
legend(["Observed" "Predicted"])
ylabel("Cases")
title("Forecast with Updates")

subplot(2,1,2)
stem(YPred - YTest)
xlabel("Month")
ylabel("Error")
title("RMSE = " + rmse)
```



Here, the predictions are more accurate when updating the network state with the observed values instead of the predicted values.

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer`

Related Examples

- “Generate Text Using Deep Learning” on page 4-180
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Speech Command Recognition Using Deep Learning

This example shows how to train a deep learning model that detects the presence of speech commands in audio. The example uses the Speech Commands Dataset [1] to train a convolutional neural network to recognize a given set of commands.

To train a network from scratch, you must first download the data set. If you do not want to download the data set or train the network, then you can load a pretrained network provided with this example and execute the next two sections of the example: *Recognize Commands with a Pre-Trained Network* and *Detect Commands Using Streaming Audio from Microphone*.

Recognize Commands with a Pre-Trained Network

Before going into the training process in detail, you will use a pre-trained speech recognition network to identify speech commands.

Load the pre-trained network.

```
load('commandNet.mat')
```

The network is trained to recognize the following speech commands:

- "yes"
- "no"
- "up"
- "down"
- "left"
- "right"
- "on"
- "off"
- "stop"
- "go"

Load a short speech signal where a person says "stop".

```
[x,fs] = audioread('stop_command.flac');
```

Listen to the command.

```
sound(x,fs)
```

The pre-trained network takes auditory-based spectrograms as inputs. You will first convert the speech waveform to an auditory-based spectrogram.

Use the function `extractAuditoryFeature` to compute the auditory spectrogram. You will go through the details of feature extraction later in the example.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the command based on its auditory spectrogram.

```
command = classify(trainedNet,auditorySpect)
```

```
command =

  categorical

     stop
```

The network is trained to classify words not belonging to this set as "unknown".

You will now classify a word ("play") that was not included in the list of command to identify.

Load the speech signal and listen to it.

```
x = audioread('play_command.flac');
sound(x,fs)
```

Compute the auditory spectrogram.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the signal.

```
command = classify(trainedNet,auditorySpect)
```

```
command =

  categorical

     unknown
```

The network is trained to classify background noise as "background".

Create a one-second signal consisting of random noise.

```
x = pinknoise(16e3);
```

Compute the auditory spectrogram.

```
auditorySpect = helperExtractAuditoryFeatures(x,fs);
```

Classify the background noise.

```
command = classify(trainedNet,auditorySpect)
```

```
command =

  categorical

     background
```

Detect Commands Using Streaming Audio from Microphone

Test your pre-trained command detection network on streaming audio from your microphone. Try saying one of the commands, for example, *yes*, *no*, or *stop*. Then, try saying one of the unknown words such as *Marvin*, *Sheila*, *bed*, *house*, *cat*, *bird*, or any number from zero to nine.

Specify the classification rate in Hz and create an audio device reader that can read audio from your microphone.

```
classificationRate = 20;
adr = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',floor(fs/classificationRate));
```

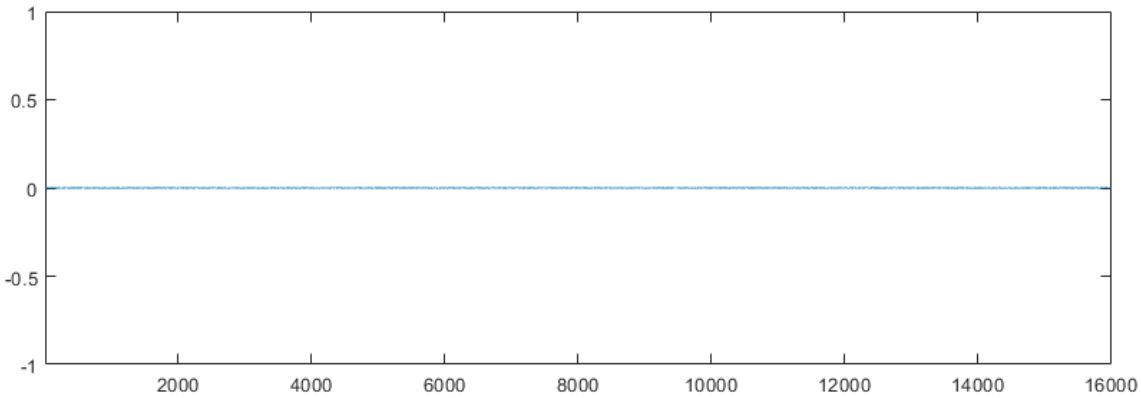
Initialize a buffer for the audio. Extract the classification labels of the network. Initialize buffers of half a second for the labels and classification probabilities of the streaming audio. Use these buffers to compare the classification results over a longer period of time and by that build 'agreement' over when a command is detected. Specify thresholds for the decision logic.

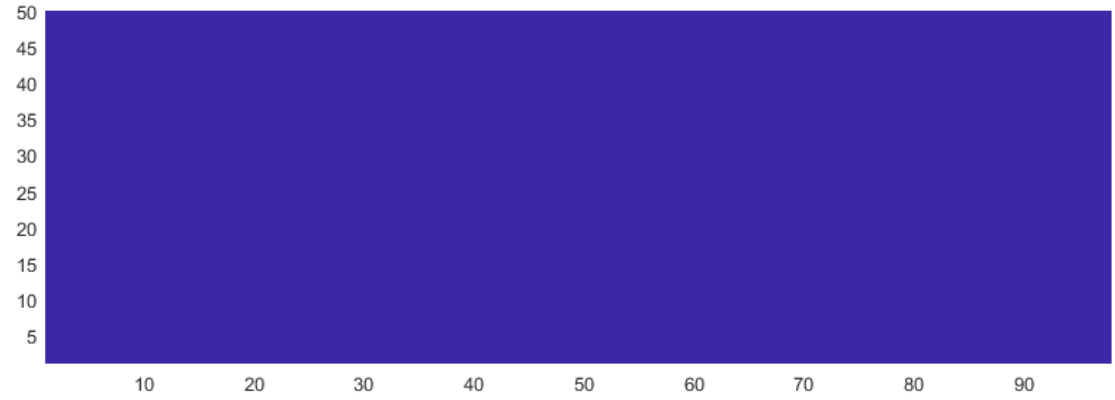
```
audioBuffer = dsp.AsyncBuffer(fs);

labels = trainedNet.Layers(end).Classes;
YBuffer(1:classificationRate/2) = categorical("background");

probBuffer = zeros([numel(labels),classificationRate/2]);

countThreshold = ceil(classificationRate*0.2);
probThreshold = 0.7;
```

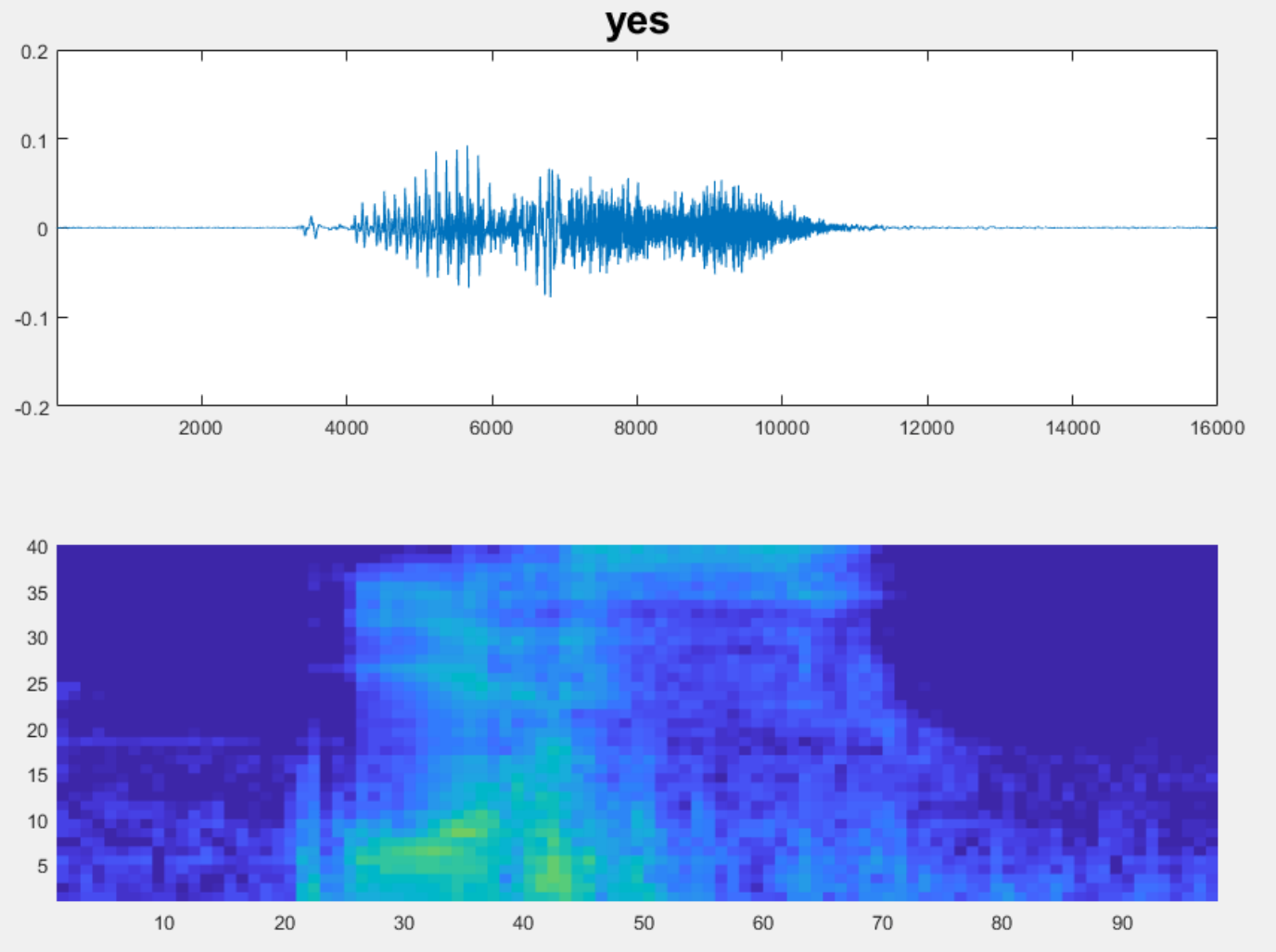
Create a figure and detect commands as long as the created figure exists. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the live detection, simply close the figure.

```
h = figure('Units','normalized','Position',[0.2 0.1 0.6 0.8]);

timeLimit = 20;

tic
while ishandle(h) && toc < timeLimit

    % Extract audio samples from the audio device and add the samples to
    % the buffer.
    x = adr();
    write(audioBuffer,x);
    y = read(audioBuffer,fs,fs-adr.SamplesPerFrame);

    spec = helperExtractAuditoryFeatures(y,fs);

    % Classify the current spectrogram, save the label to the label buffer,
    % and save the predicted probabilities to the probability buffer.
    [YPredicted,probs] = classify(trainedNet,spec,'ExecutionEnvironment','cpu');
    YBuffer = [YBuffer(2:end),YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

    % Plot the current waveform and spectrogram.
    subplot(2,1,1)
    plot(y)
    axis tight
    ylim([-1,1])
```

```
        subplot(2,1,2)
        pcolor(spec')
        caxis([-4 2.6445])
        shading flat

        % Now do the actual command detection by performing a very simple
        % thresholding operation. Declare a detection and display it in the
        % figure title if all of the following hold: 1) The most common label
        % is not background. 2) At least countThreshold of the latest frame
        % labels agree. 3) The maximum probability of the predicted label is at
        % least probThreshold. Otherwise, do not declare a detection.
        [YMode,count] = mode(YBuffer);

        maxProb = max(probBuffer(labels == YMode,:));
        subplot(2,1,1)
        if YMode == "background" || count < countThreshold || maxProb < probThreshold
            title(" ")
        else
            title(string(YMode),'FontSize',20)
        end

        drawnow
    end
```


1
0.5
0
-0.5
-1
2000
4000
6000
8000
10000
12000
14000
16000


50
45
40
35
30
25
20
15
10
5
10
20
30
40
50
60
70
80
90



Load Speech Commands Data Set

This example uses the Google Speech Commands Dataset [1]. Download the dataset and untar the downloaded file. Set PathToDatabase to the location of the data.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'google_speech');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (1.4 GB) ...')
    unzip(url,downloadFolder)
end
```

Create Training Datastore

Create an `audioDatastore` (Audio Toolbox) that points to the training data set.

```
ads = audioDatastore(fullfile(dataFolder, 'train'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames')
```

```
ads =

  audioDatastore with properties:

                      Files: {
                             ' ...\AppData\Local\Temp\google_speech\train\bed\00176480_nohash_0
                             ' ...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_0
                             ' ...\AppData\Local\Temp\google_speech\train\bed\004ae714_nohash_1
                              ... and 51085 more
                             }
                    Folders: {
                             'C:\Users\jibrahim\AppData\Local\Temp\google_speech\train'
                             }
                     Labels: [bed; bed; bed ... and 51085 more categorical]
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Choose Words to Recognize

Specify the words that you want your model to recognize as commands. Label all words that are not commands as `unknown`. Labeling words that are not commands as `unknown` creates a group of words that approximates the distribution of all words other than the commands. The network uses this group to learn the difference between commands and all other words.

To reduce the class imbalance between the known and unknown words and speed up processing, only include a fraction of the unknown words in the training set.

Use `subset` (Audio Toolbox) to create a datastore that contains only the commands and the subset of unknown words. Count the number of examples belonging to each category.

```
commands = categorical(["yes","no","up","down","left","right","on","off","stop","go"]);

isCommand = ismember(ads.Labels,commands);
isUnknown = ~isCommand;

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsTrain = subset(ads,isCommand|isUnknown);
countEachLabel(adsTrain)
```

```
ans =

  11×2 table

     Label     Count
    _______    _____

    down       1842
    go         1861
    left       1839
```

```
    no         1853
    off        1839
    on         1864
    right      1852
    stop       1885
    unknown    6483
    up         1843
    yes        1860
```

Create Validation Datastore

Create an `audioDatastore` (Audio Toolbox) that points to the validation data set. Follow the same steps used to create the training datastore.

```
ads = audioDatastore(fullfile(dataFolder, 'validation'), ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames')

isCommand = ismember(ads.Labels,commands);
isUnknown = ~isCommand;

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsValidation = subset(ads,isCommand|isUnknown);
countEachLabel(adsValidation)
```

```
ads =

  audioDatastore with properties:

                      Files: {
                             ' ...\AppData\Local\Temp\google_speech\validation\bed\026290a7_noha
                             ' ...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha
                             ' ...\AppData\Local\Temp\google_speech\validation\bed\060cd039_noha
                              ... and 6795 more
                             }
                    Folders: {
                             'C:\Users\jibrahim\AppData\Local\Temp\google_speech\validation'
                             }
                     Labels: [bed; bed; bed ... and 6795 more categorical]
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ans =

  11×2 table

     Label     Count
    _______    _____
```

```
    down         264
    go           260
    left         247
    no           270
    off          256
    on           257
    right        256
    stop         246
    unknown      850
    up           260
    yes          261
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = false;
if reduceDataset
    numUniqueLabels = numel(unique(adsTrain.Labels));
    % Reduce the dataset by a factor of 20
    adsTrain = splitEachLabel(adsTrain,round(numel(adsTrain.Files) / numUniqueLabels / 20));
    adsValidation = splitEachLabel(adsValidation,round(numel(adsValidation.Files) / numUniqueLabe
end
```

Compute Auditory Spectrograms

To prepare the data for efficient training of a convolutional neural network, convert the speech waveforms to auditory-based spectrograms.

Define the parameters of the feature extraction. `segmentDuration` is the duration of each speech clip (in seconds). `frameDuration` is the duration of each frame for spectrum calculation. `hopDuration` is the time step between each spectrum. `numBands` is the number of filters in the auditory spectrogram.

Create an `audioFeatureExtractor` (Audio Toolbox) object to perform the feature extraction.

```
fs = 16e3; % Known sample rate of the data set.

segmentDuration = 1;
frameDuration = 0.025;
hopDuration = 0.010;

segmentSamples = round(segmentDuration*fs);
frameSamples = round(frameDuration*fs);
hopSamples = round(hopDuration*fs);
overlapSamples = frameSamples - hopSamples;

FFTLength = 512;
numBands = 50;

afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',FFTLength, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
    'barkSpectrum',true);
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);
```

Read a file from the dataset. Training a convolutional neural network requires input to be a consistent size. Some files in the data set are less than 1 second long. Apply zero-padding to the front and back of the audio signal so that it is of length `segmentSamples`.

```
x = read(adsTrain);

numSamples = size(x,1);

numToPadFront = floor( (segmentSamples - numSamples)/2 );
numToPadBack = ceil( (segmentSamples - numSamples)/2 );

xPadded = [zeros(numToPadFront,1,'like',x);x;zeros(numToPadBack,1,'like',x)];
```

To extract audio features, call `extract`. The output is a Bark spectrum with time across rows.

```
features = extract(afe,xPadded);
[numHops,numFeatures] = size(features)
```

```
numHops =

    98


numFeatures =

    50
```

In this example, you post-process the auditory spectrogram by applying a logarithm. Taking a log of small numbers can lead to roundoff error.

To speed up processing, you can distribute the feature extraction across multiple workers using `parfor`.

First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel')) && ~reduceDataset
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end
```

For each partition, read from the datastore, zero-pad the signal, and then extract the features.

```
parfor ii = 1:numPar
    subds = partition(adsTrain,numPar,ii);
    XTrain = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size
        XTrain(:,:,:,idx) = extract(afe,xPadded);
    end
    XTrainC{ii} = XTrain;
end
```

Convert the output to a 4-dimensional array with auditory spectrograms along the fourth dimension.

```
XTrain = cat(4,XTrainC{:});

[numHops,numBands,numChannels,numSpec] = size(XTrain)
```

```
numHops =

    98


numBands =

    50


numChannels =

     1


numSpec =

       25021
```

Scale the features by the window power and then take the log. To obtain data with a smoother distribution, take the logarithm of the spectrograms using a small offset.

```
epsil = 1e-6;
XTrain = log10(XTrain + epsil);
```

Perform the feature extraction steps described above to the validation set.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(adsValidation,pool);
else
    numPar = 1;
end
parfor ii = 1:numPar
    subds = partition(adsValidation,numPar,ii);
    XValidation = zeros(numHops,numBands,1,numel(subds.Files));
    for idx = 1:numel(subds.Files)
        x = read(subds);
        xPadded = [zeros(floor((segmentSamples-size(x,1))/2),1);x;zeros(ceil((segmentSamples-size
        XValidation(:,:,:,idx) = extract(afe,xPadded);
    end
    XValidationC{ii} = XValidation;
end
XValidation = cat(4,XValidationC{:});
XValidation = log10(XValidation + epsil);
```

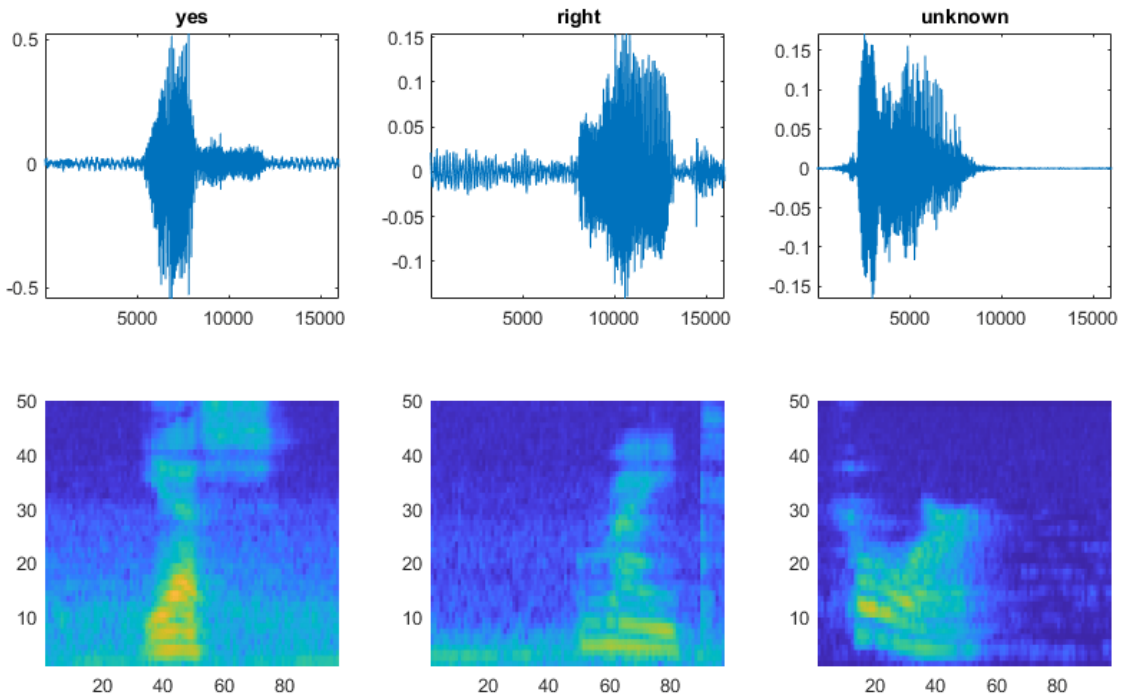
Isolate the train and validation labels. Remove empty categories.

```
YTrain = removecats(adsTrain.Labels);
YValidation = removecats(adsValidation.Labels);
```

Visualize Data

Plot the waveforms and auditory spectrograms of a few training samples. Play the corresponding audio clips.

```
specMin = min(XTrain,[],'all');
specMax = max(XTrain,[],'all');
idx = randperm(numel(adsTrain.Files),3);
figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
for i = 1:3
    [x,fs] = audioread(adsTrain.Files{idx(i)});
    subplot(2,3,i)
    plot(x)
    axis tight
    title(string(adsTrain.Labels(idx(i))))

    subplot(2,3,i+3)
    spect = (XTrain(:,:,1,idx(i))');
    pcolor(spect)
    caxis([specMin specMax])
    shading flat

    sound(x,fs)
    pause(2)
end
```



Add Background Noise Data

The network must be able not only to recognize different spoken words but also to detect if the input contains silence or background noise.

Use the audio files in the `_background_` folder to create samples of one-second clips of background noise. Create an equal number of background clips from each background noise file. You can also create your own recordings of background noise and add them to the `_background_` folder. Before calculating the spectrograms, the function rescales each audio clip with a factor sampled from a log-uniform distribution in the range given by `volumeRange`.

```
adsBkg = audioDatastore(fullfile(dataFolder, 'background'))
numBkgClips = 4000;
if reduceDataset
    numBkgClips = numBkgClips/20;
end
volumeRange = log10([1e-4,1]);

numBkgFiles = numel(adsBkg.Files);
numClipsPerFile = histcounts(1:numBkgClips,linspace(1,numBkgClips,numBkgFiles+1));
Xbkg = zeros(size(XTrain,1),size(XTrain,2),1,numBkgClips,'single');
bkgAll = readall(adsBkg);
ind = 1;

for count = 1:numBkgFiles
    bkg = bkgAll{count};
    idxStart = randi(numel(bkg)-fs,numClipsPerFile(count),1);
    idxEnd = idxStart+fs-1;
    gain = 10.^((volumeRange(2)-volumeRange(1))*rand(numClipsPerFile(count),1) + volumeRange(1))
    for j = 1:numClipsPerFile(count)

        x = bkg(idxStart(j):idxEnd(j))*gain(j);

        x = max(min(x,1),-1);

        Xbkg(:,:,:,ind) = extract(afe,x);

        if mod(ind,1000)==0
            disp("Processed " + string(ind) + " background clips out of " + string(numBkgClips))
        end
        ind = ind + 1;
    end
end
Xbkg = log10(Xbkg + epsil);
```

```
adsBkg =

  audioDatastore with properties:

                      Files: {
                             ' ...\AppData\Local\Temp\google_speech\background\doing_the_dishes
                             ' ...\AppData\Local\Temp\google_speech\background\dude_miaowing.wav
                             ' ...\AppData\Local\Temp\google_speech\background\exercise_bike.wav
                              ... and 3 more
                             }
                    Folders: {
                             'C:\Users\jibrahim\AppData\Local\Temp\google_speech\background'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "mp4"    "m4a"]
```

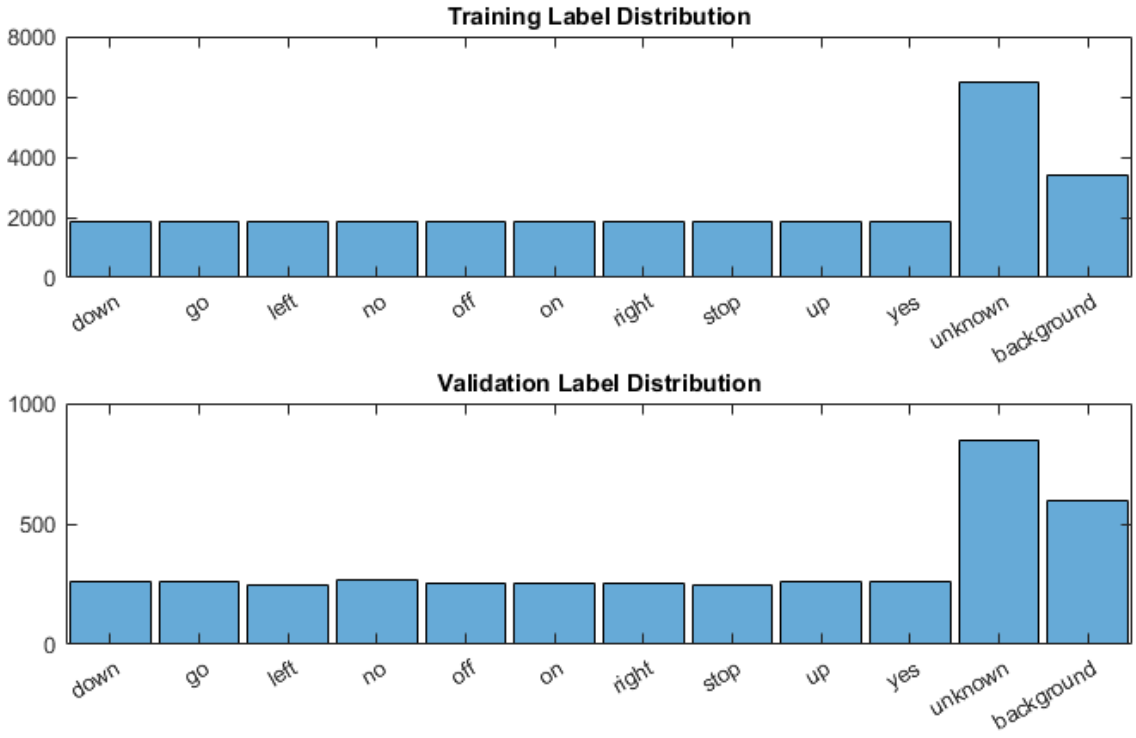
```
         DefaultOutputFormat: "wav"

Processed 1000 background clips out of 4000
Processed 2000 background clips out of 4000
Processed 3000 background clips out of 4000
Processed 4000 background clips out of 4000
```

Split the spectrograms of background noise between the training, validation, and test sets. Because the `_background_noise_` folder contains only about five and a half minutes of background noise, the background samples in the different data sets are highly correlated. To increase the variation in the background noise, you can create your own background files and add them to the folder. To increase the robustness of the network to noise, you can also try mixing background noise into the speech files.

```
numTrainBkg = floor(0.85*numBkgClips);
numValidationBkg = floor(0.15*numBkgClips);

XTrain(:,:,:,end+1:end+numTrainBkg) = Xbkg(:,:,:,1:numTrainBkg);
YTrain(end+1:end+numTrainBkg) = "background";

XValidation(:,:,:,end+1:end+numValidationBkg) = Xbkg(:,:,:,numTrainBkg+1:end);
YValidation(end+1:end+numValidationBkg) = "background";
```

Plot the distribution of the different class labels in the training and validation sets.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5])

subplot(2,1,1)
histogram(YTrain)
title("Training Label Distribution")

subplot(2,1,2)
histogram(YValidation)
title("Validation Label Distribution")
```



Define Neural Network Architecture

Create a simple network architecture as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps "spatially" (that is, in time and frequency) using max pooling layers. Add a final max pooling layer that pools the input feature map globally over time. This enforces (approximate) time-translation invariance in the input spectrograms, allowing the network to perform the same classification independent of the exact position of the speech in time. Global pooling also significantly reduces the number of parameters in the final fully connected layer. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

The network is small, as it has only five convolutional layers with few filters. `numF` controls the number of filters in the convolutional layers. To increase the accuracy of the network, try increasing the network depth by adding identical blocks of convolutional, batch normalization, and ReLU layers. You can also try increasing the number of convolutional filters by increasing `numF`.

Use a weighted cross entropy classification loss. `weightedClassificationLayer(classWeights)` creates a custom classification layer that calculates the cross entropy loss with observations weighted by `classWeights`. Specify the class weights in the same order as the classes appear in `categories(YTrain)`. To give each class equal total weight in the loss, use class weights that are inversely proportional to the number of training examples in each class. When using the Adam optimizer to train the network, the training algorithm is independent of the overall normalization of the class weights.

```
classWeights = 1./countcats(YTrain);
classWeights = classWeights'/mean(classWeights);
numClasses = numel(categories(YTrain));
```

```
timePoolSize = ceil(numHops/8);

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer([numHops numBands])

    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer([timePoolSize,1])

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    weightedClassificationLayer(classWeights)];
```

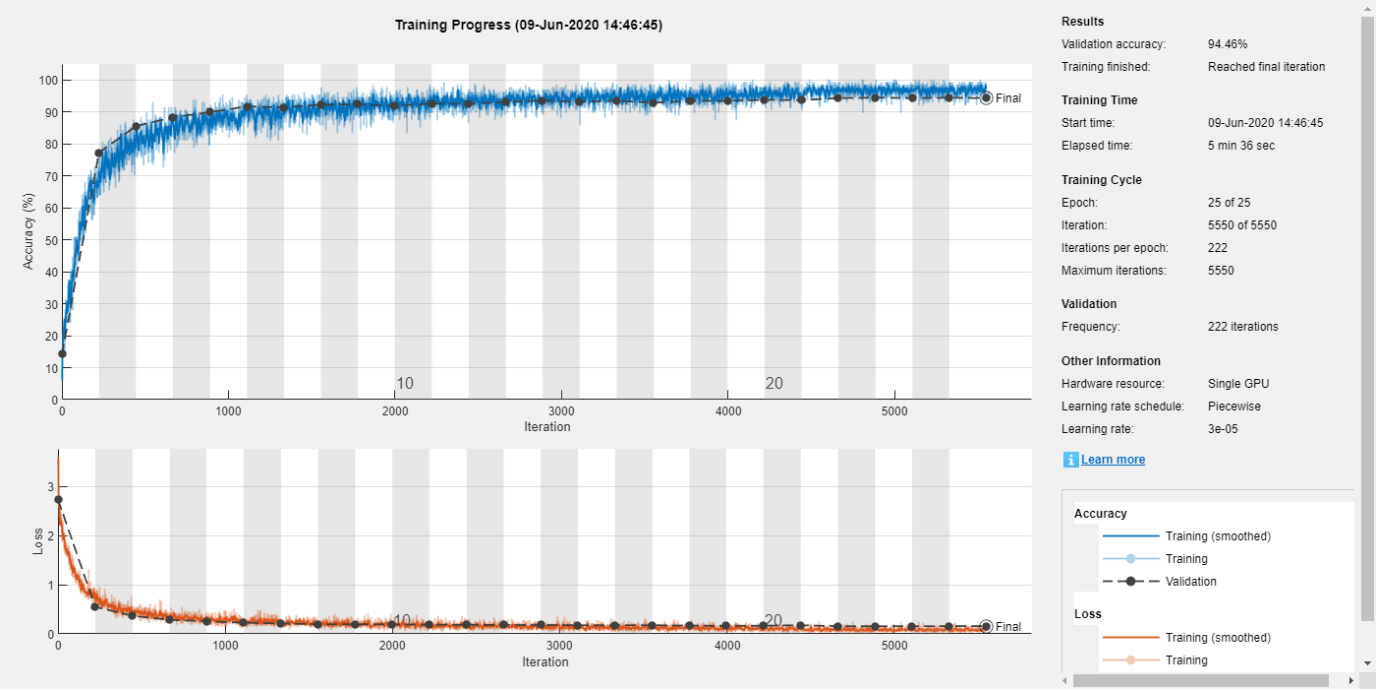
Train Network

Specify the training options. Use the Adam optimizer with a mini-batch size of 128. Train for 25 epochs and reduce the learning rate by a factor of 10 after 20 epochs.

```
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'MaxEpochs',25, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',validationFrequency, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',20);
```

Train the network. If you do not have a GPU, then training the network can take time.

```
trainedNet = trainNetwork(XTrain,YTrain,layers,options);
```



Evaluate Trained Network

Calculate the final accuracy of the network on the training set (without data augmentation) and validation set. The network is very accurate on this data set. However, the training, validation, and test data all have similar distributions that do not necessarily reflect real-world environments. This limitation particularly applies to the `unknown` category, which contains utterances of only a small number of words.

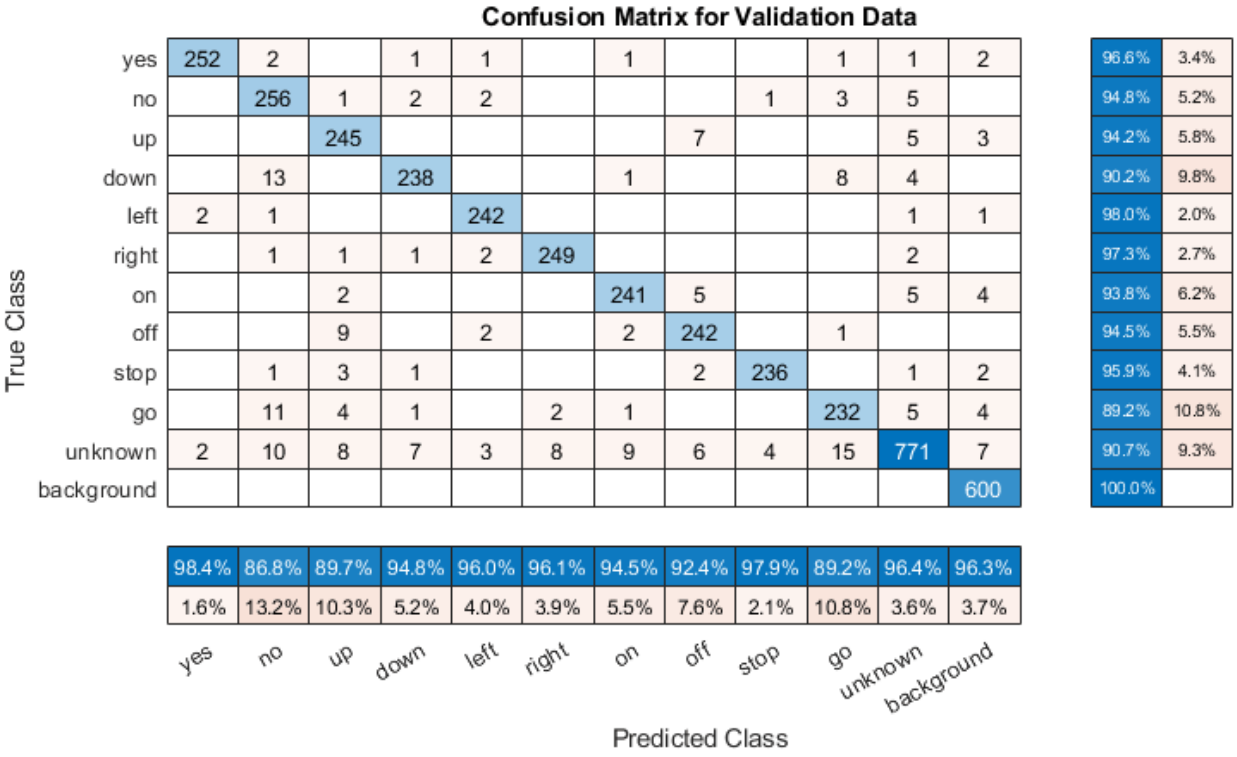
```
if reduceDataset
    load('commandNet.mat','trainedNet');
end
YValPred = classify(trainedNet,XValidation);
validationError = mean(YValPred ~= YValidation);
YTrainPred = classify(trainedNet,XTrain);
trainError = mean(YTrainPred ~= YTrain);
disp("Training error: " + trainError*100 + "%")
disp("Validation error: " + validationError*100 + "%")
```

```
Training error: 1.907%
Validation error: 5.5376%
```

Plot the confusion matrix. Display the precision and recall for each class by using column and row summaries. Sort the classes of the confusion matrix. The largest confusion is between unknown words and commands, *up* and *off*, *down* and *no*, and *go* and *no*.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
cm = confusionchart(YValidation,YValPred);
cm.Title = 'Confusion Matrix for Validation Data';
cm.ColumnSummary = 'column-normalized';
```

```
cm.RowSummary = 'row-normalized';
sortClasses(cm, [commands,"unknown","background"])
```



When working on applications with constrained hardware resources such as mobile applications, consider the limitations on available memory and computational resources. Compute the total size of the network in kilobytes and test its prediction speed when using a CPU. The prediction time is the time for classifying a single input image. If you input multiple images to the network, these can be classified simultaneously, leading to shorter prediction times per image. When classifying streaming audio, however, the single-image prediction time is the most relevant.

```
info = whos('trainedNet');
disp("Network size: " + info.bytes/1024 + " kB")

for i = 1:100
    x = randn([numHops,numBands]);
    tic
    [YPredicted,probs] = classify(trainedNet,x,"ExecutionEnvironment",'cpu');
    time(i) = toc;
end
disp("Single-image prediction time on CPU: " + mean(time(11:end))*1000 + " ms")

Network size: 286.7402 kB
Single-image prediction time on CPU: 2.5119 ms
```

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed

under the Creative Commons Attribution 4.0 license, available here: https://creativecommons.org/licenses/by/4.0/legalcode.

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license, available here: https://creativecommons.org/licenses/by/4.0/legalcode.

See Also

`trainNetwork` | `classify` | `analyzeNetwork`

More About

- “Deep Learning in MATLAB” on page 1-2

Sequence-to-Sequence Classification Using Deep Learning

This example shows how to classify each time step of sequence data using a long short-term memory (LSTM) network.

To train a deep neural network to classify each time step of sequence data, you can use a *sequence-to-sequence LSTM network*. A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of the sequence data.

This example uses sensor data obtained from a smartphone worn on the body. The example trains an LSTM network to recognize the activity of the wearer given time series data representing accelerometer readings in three different directions. The training data contains time series data for seven people. Each sequence has three features and varies in length. The data set contains six training observations and one test observation.

Load Sequence Data

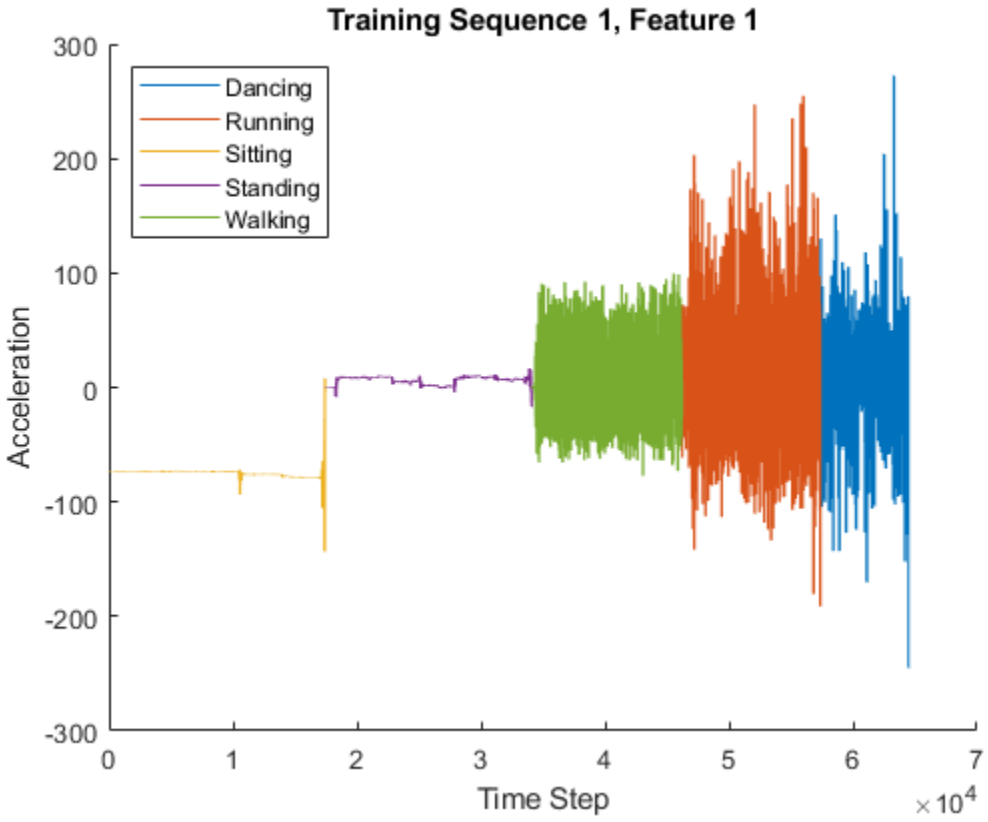
Load the human activity recognition data. The data contains seven time series of sensor data obtained from a smartphone worn on the body. Each sequence has three features and varies in length. The three features correspond to the accelerometer readings in three different directions.

```
load HumanActivityTrain
XTrain
```

```
XTrain=6×1 cell array
    {3×64480 double}
    {3×53696 double}
    {3×56416 double}
    {3×50688 double}
    {3×51888 double}
    {3×54256 double}
```

Visualize one training sequence in a plot. Plot the first feature of the first training sequence and color the plot according to the corresponding activity.

```
X = XTrain{1}(1,:);
classes = categories(YTrain{1});

figure
for j = 1:numel(classes)
    label = classes(j);
    idx = find(YTrain{1} == label);
    hold on
    plot(idx,X(idx))
end
hold off

xlabel("Time Step")
ylabel("Acceleration")
title("Training Sequence 1, Feature 1")
legend(classes,'Location','northwest')
```



Define LSTM Network Architecture

Define the LSTM network architecture. Specify the input to be sequences of size 3 (the number of features of the input data). Specify an LSTM layer with 200 hidden units, and output the full sequence. Finally, specify five classes by including a fully connected layer of size 5, followed by a softmax layer and a classification layer.

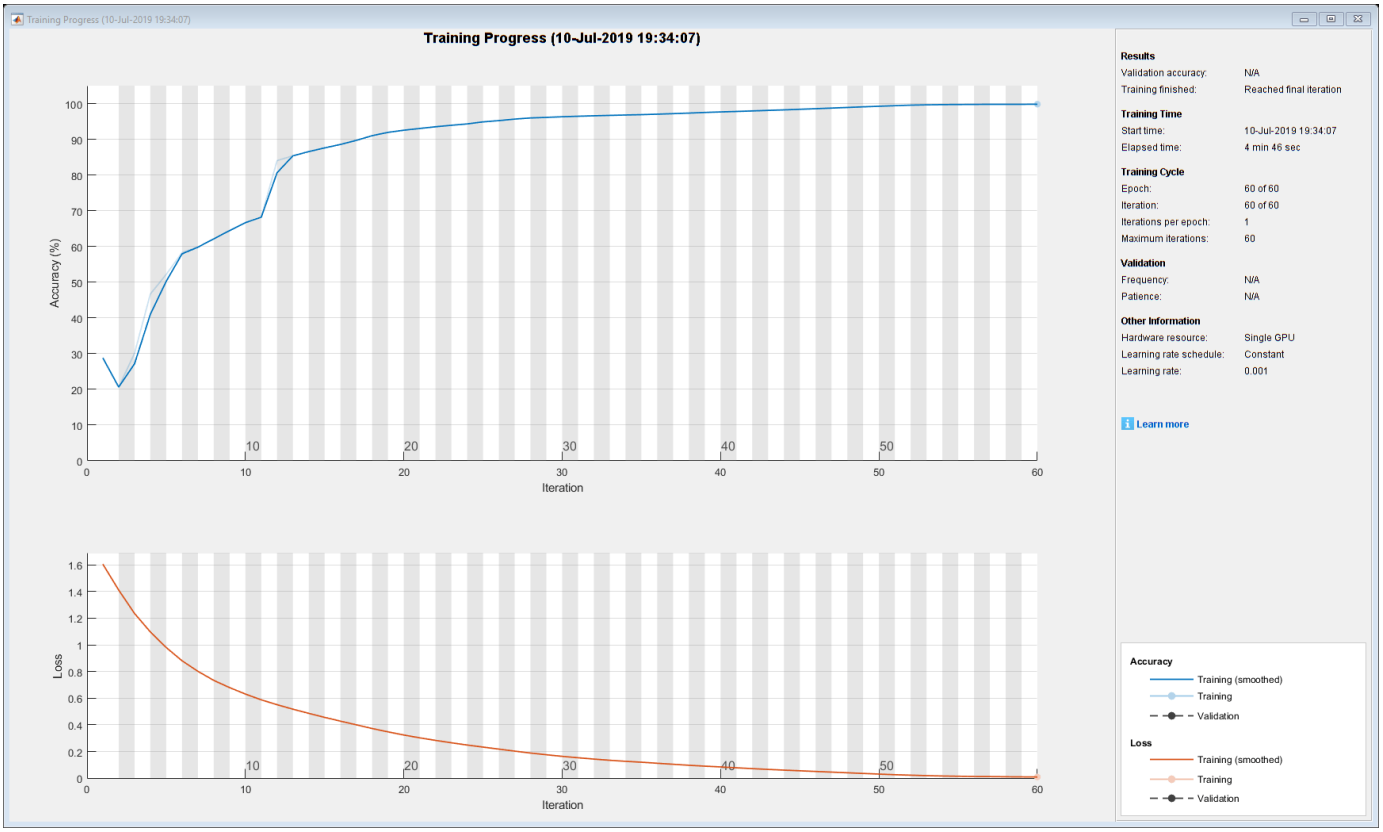
```
numFeatures = 3;
numHiddenUnits = 200;
numClasses = 5;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Set the solver to `'adam'`. Train for 60 epochs. To prevent the gradients from exploding, set the gradient threshold to 2.

```
options = trainingOptions('adam', ...
    'MaxEpochs',60, ...
    'GradientThreshold',2, ...
    'Verbose',0, ...
    'Plots','training-progress');
```

Train the LSTM network with the specified training options using `trainNetwork`. Each mini-batch contains the whole training set, so the plot is updated once per epoch. The sequences are very long, so it might take some time to process each mini-batch and update the plot.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```

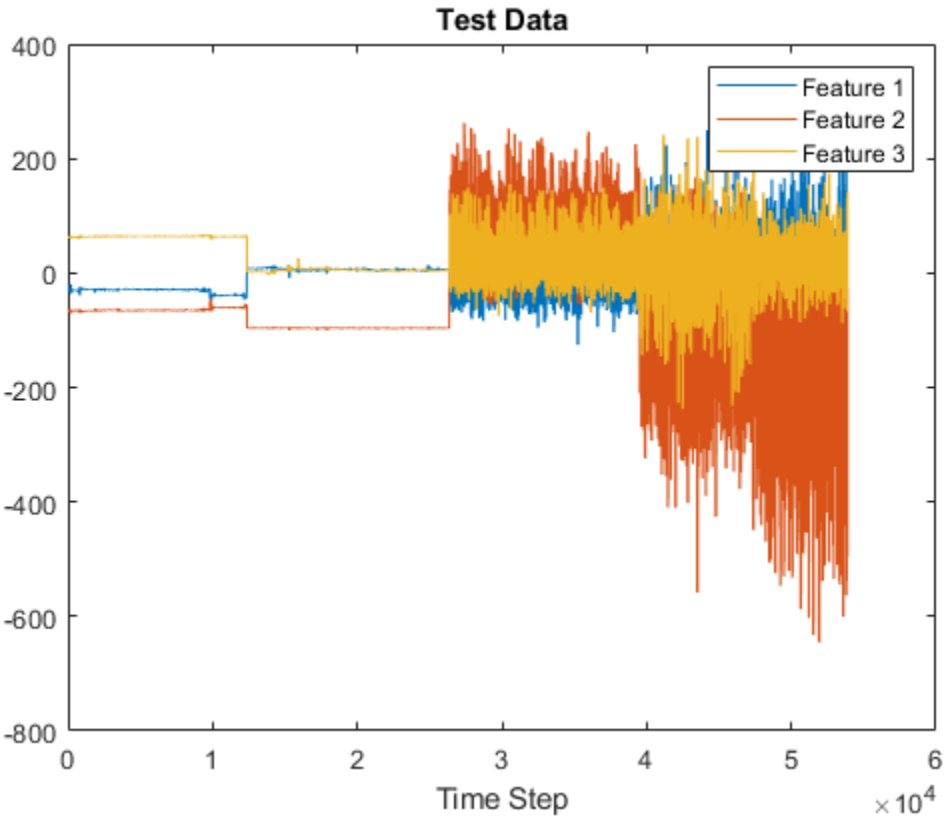


Test LSTM Network

Load the test data and classify the activity at each time step.

Load the human activity test data. `XTest` contains a single sequence of dimension 3. `YTest` is contains sequence of categorical labels corresponding to the activity at each time step.

```
load HumanActivityTest
figure
plot(XTest{1}')
xlabel("Time Step")
legend("Feature " + (1:numFeatures))
title("Test Data")
```



Classify the test data using `classify`.

```
YPred = classify(net,XTest{1});
```

Alternatively, you can make predictions one time step at a time by using `classifyAndUpdateState`. This is useful when you have the values of the time steps arriving in a stream. Usually, it is faster to make predictions on full sequences when compared to making predictions one time step at a time. For an example showing how to forecast future time steps by updating the network between single time step predictions, see “Time Series Forecasting Using Deep Learning” on page 4-15.

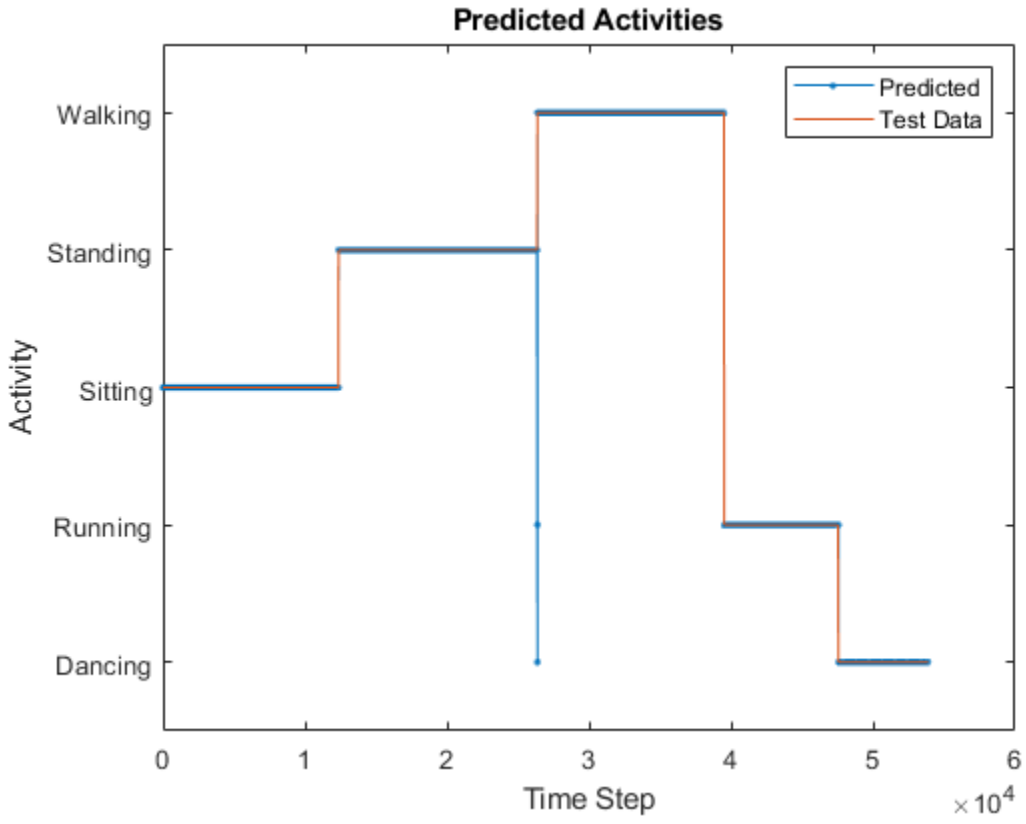
Calculate the accuracy of the predictions.

```
acc = sum(YPred == YTest{1})./numel(YTest{1})
```

```
acc = 0.9998
```

Compare the predictions with the test data by using a plot.

```
figure
plot(YPred,'.-')
hold on
plot(YTest{1})
hold off

xlabel("Time Step")
ylabel("Activity")
title("Predicted Activities")
legend(["Predicted" "Test Data"])
```



See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Sequence-to-Sequence Regression Using Deep Learning

This example shows how to predict the remaining useful life (RUL) of engines by using deep learning.

To train a deep neural network to predict numeric values from time series or sequence data, you can use a long short-term memory (LSTM) network.

This example uses the Turbofan Engine Degradation Simulation Data Set as described in [1]. The example trains an LSTM network to predict the remaining useful life of an engine (predictive maintenance), measured in cycles, given time series data representing various sensors in the engine. The training data contains simulated time series data for 100 engines. Each sequence varies in length and corresponds to a full run to failure (RTF) instance. The test data contains 100 partial sequences and corresponding values of the remaining useful life at the end of each sequence.

The data set contains 100 training observations and 100 test observations.

Download Data

Download and unzip the Turbofan Engine Degradation Simulation Data Set from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ [2].

Each time series of the Turbofan Engine Degradation Simulation data set represents a different engine. Each engine starts with unknown degrees of initial wear and manufacturing variation. The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure.

The data contains a ZIP-compressed text files with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, and each column is a different variable. The columns correspond to the following:

- Column 1 – Unit number
- Column 2 – Time in cycles
- Columns 3–5 – Operational settings
- Columns 6–26 – Sensor measurements 1–21

Create a directory to store the Turbofan Engine Degradation Simulation data set.

```
dataFolder = fullfile(tempdir,"turbofan");
if ~exist(dataFolder,'dir')
    mkdir(dataFolder);
end
```

Download and extract the Turbofan Engine Degradation Simulation Data Set from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/.

Unzip the data from the file `CMAPSSData.zip`.

```
filename = "CMAPSSData.zip";
unzip(filename,dataFolder)
```

Prepare Training Data

Load the data using the function `processTurboFanDataTrain` attached to this example. The function `processTurboFanDataTrain` extracts the data from `filenamePredictors` and returns the cell arrays `XTrain` and `YTrain`, which contain the training predictor and response sequences.

```
filenamePredictors = fullfile(dataFolder,"train_FD001.txt");
[XTrain,YTrain] = processTurboFanDataTrain(filenamePredictors);
```

Remove Features with Constant Values

Features that remain constant for all time steps can negatively impact the training. Find the rows of data that have the same minimum and maximum values, and remove the rows.

```
m = min([XTrain{:}],[],2);
M = max([XTrain{:}],[],2);
idxConstant = M == m;

for i = 1:numel(XTrain)
    XTrain{i}(idxConstant,:) = [];
end
```

View the number of remaining features in the sequences.

```
numFeatures = size(XTrain{1},1)
```

```
numFeatures = 17
```

Normalize Training Predictors

Normalize the training predictors to have zero mean and unit variance. To calculate the mean and standard deviation over all observations, concatenate the sequence data horizontally.

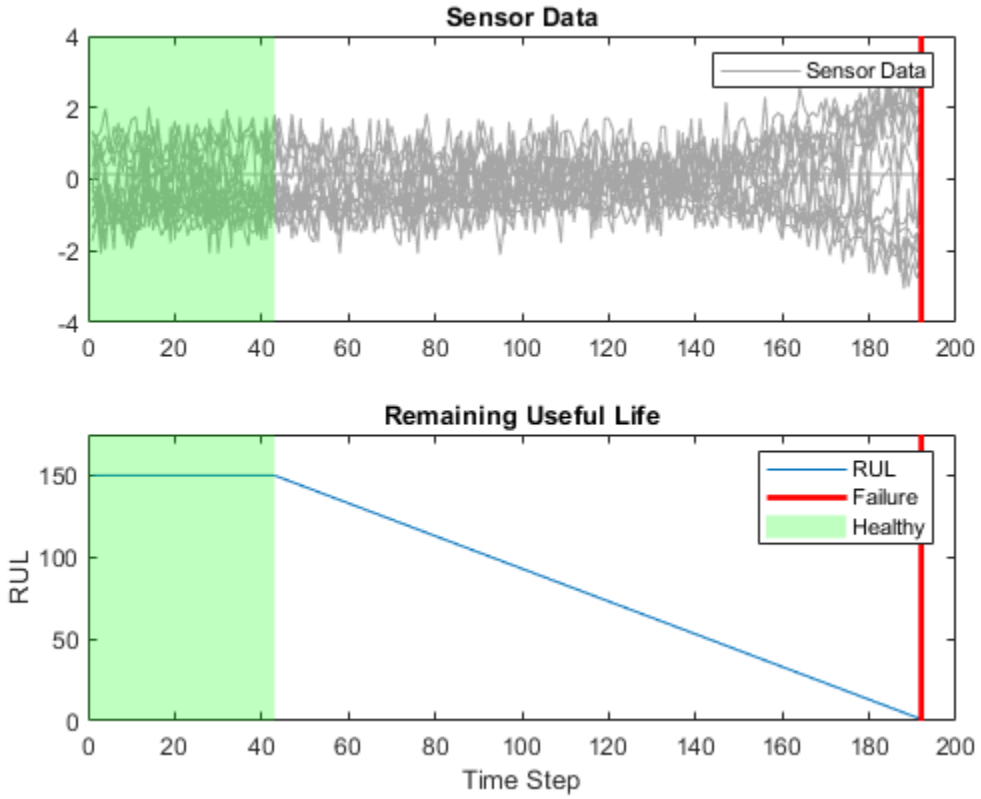
```
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);

for i = 1:numel(XTrain)
    XTrain{i} = (XTrain{i} - mu) ./ sig;
end
```

Clip Responses

To learn more from the sequence data when the engines are close to failing, clip the responses at the threshold 150. This makes the network treat instances with higher RUL values as equal.

```
thr = 150;
for i = 1:numel(YTrain)
    YTrain{i}(YTrain{i} > thr) = thr;
end
```

This figure shows the first observation and the corresponding clipped response.



Prepare Data for Padding

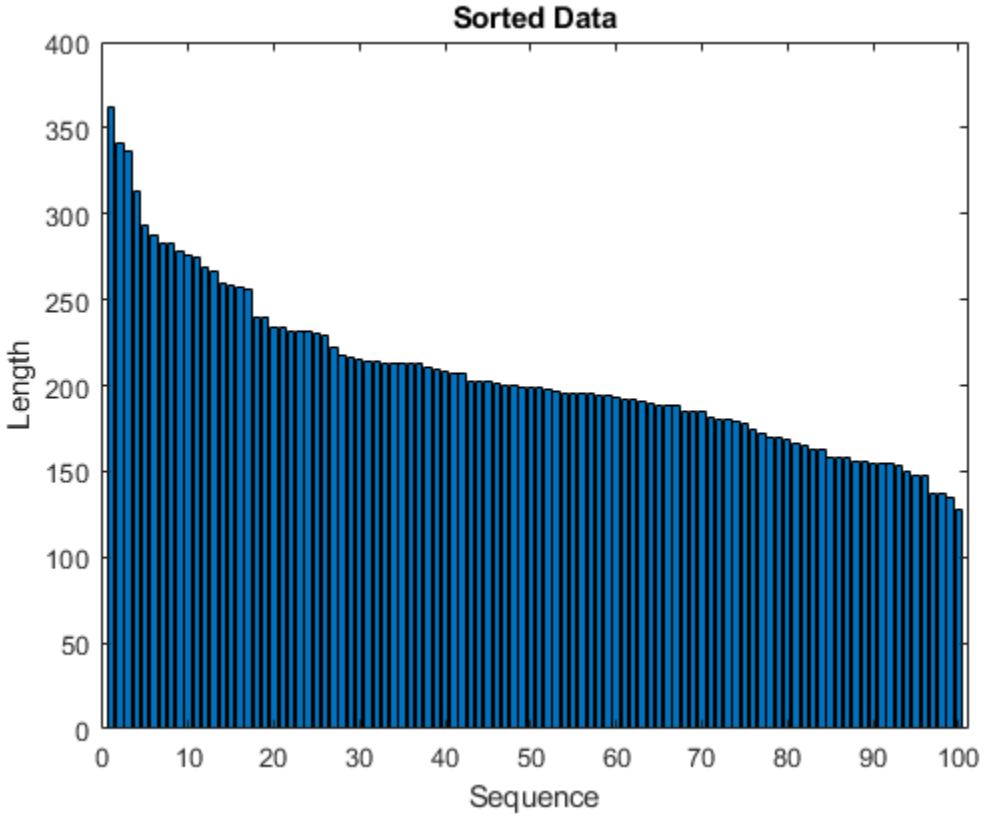
To minimize the amount of padding added to the mini-batches, sort the training data by sequence length. Then, choose a mini-batch size which divides the training data evenly and reduces the amount of padding in the mini-batches.

Sort the training data by sequence length.

```
for i=1:numel(XTrain)
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end

[sequenceLengths,idx] = sort(sequenceLengths,'descend');
XTrain = XTrain(idx);
YTrain = YTrain(idx);
```

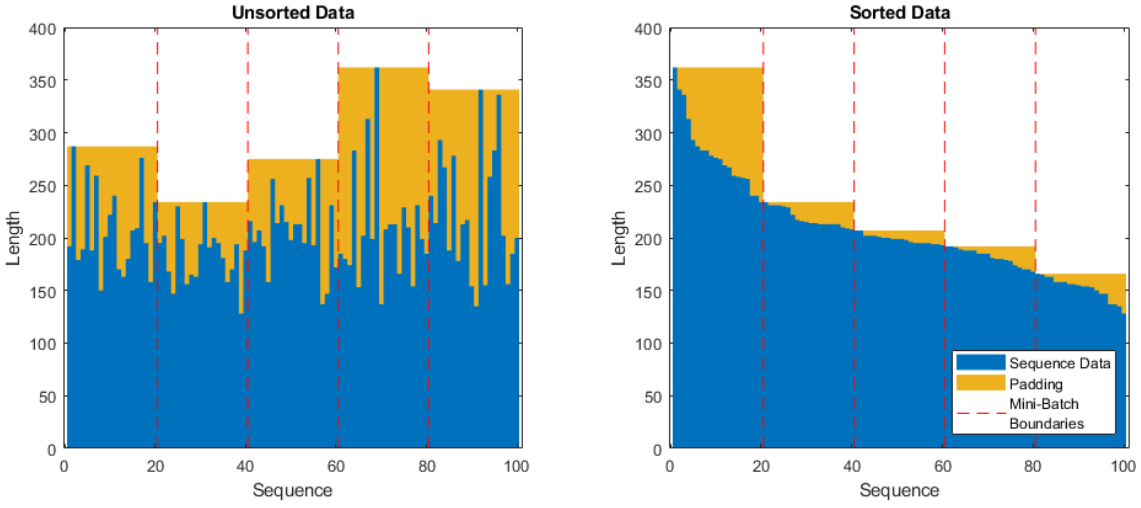
View the sorted sequence lengths in a bar chart.

```
figure
bar(sequenceLengths)
xlabel("Sequence")
ylabel("Length")
title("Sorted Data")
```



Choose a mini-batch size which divides the training data evenly and reduces the amount of padding in the mini-batches. Specify a mini-batch size of 20. This figure illustrates the padding added to the unsorted and sorted sequences.

```
miniBatchSize = 20;
```



Define Network Architecture

Define the network architecture. Create an LSTM network that consists of an LSTM layer with 200 hidden units, followed by a fully connected layer of size 50 and a dropout layer with dropout probability 0.5.

```
numResponses = size(YTrain{1},1);
numHiddenUnits = 200;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(50)
    dropoutLayer(0.5)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

Specify the training options. Train for 60 epochs with mini-batches of size 20 using the solver `'adam'`. Specify the learning rate 0.01. To prevent the gradients from exploding, set the gradient threshold to 1. To keep the sequences sorted by length, set `'Shuffle'` to `'never'`.

```
maxEpochs = 60;
miniBatchSize = 20;

options = trainingOptions('adam', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'InitialLearnRate',0.01, ...
    'GradientThreshold',1, ...
    'Shuffle','never', ...
    'Plots','training-progress',...
    'Verbose',0);
```

Train the Network

Train the network using `trainNetwork`.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```

Test the Network

Prepare the test data using the function `processTurboFanDataTest` attached to this example. The function `processTurboFanDataTest` extracts the data from `filenamePredictors` and `filenameResponses` and returns the cell arrays `XTest` and `YTest`, which contain the test predictor and response sequences, respectively.

```
filenamePredictors = fullfile(dataFolder,"test_FD001.txt");
filenameResponses = fullfile(dataFolder,"RUL_FD001.txt");
[XTest,YTest] = processTurboFanDataTest(filenamePredictors,filenameResponses);
```

Remove features with constant values using `idxConstant` calculated from the training data. Normalize the test predictors using the same parameters as in the training data. Clip the test responses at the same threshold used for the training data.

```
for i = 1:numel(XTest)
    XTest{i}(idxConstant,:) = [];
    XTest{i} = (XTest{i} - mu) ./ sig;
```

```
    YTest{i}(YTest{i} > thr) = thr;
end
```

Make predictions on the test data using `predict`. To prevent the function from adding padding to the data, specify the mini-batch size 1.

```
YPred = predict(net,XTest,'MiniBatchSize',1);
```

The LSTM network makes predictions on the partial sequence one time step at a time. At each time step, the network predicts using the value at this time step, and the network state calculated from the previous time steps only. The network updates its state between each prediction. The `predict` function returns a sequence of these predictions. The last element of the prediction corresponds to the predicted RUL for the partial sequence.

Alternatively, you can make predictions one time step at a time by using `predictAndUpdateState`. This is useful when you have the values of the time steps arriving in a stream. Usually, it is faster to make predictions on full sequences when compared to making predictions one time step at a time. For an example showing how to forecast future time steps by updating the network between single time step predictions, see “Time Series Forecasting Using Deep Learning” on page 4-15.

Visualize some of the predictions in a plot.

```
idx = randperm(numel(YPred),4);
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YTest{idx(i)},'--')
    hold on
    plot(YPred{idx(i)},'.-')
    hold off

    ylim([0 thr + 25])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southeast')
```

For a given partial sequence, the predicted current RUL is the last element of the predicted sequences. Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
for i = 1:numel(YTest)
    YTestLast(i) = YTest{i}(end);
    YPredLast(i) = YPred{i}(end);
end
figure
rmse = sqrt(mean((YPredLast - YTestLast).^2))
histogram(YPredLast - YTestLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```

References

1. Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pp. 1-9. IEEE, 2008.
2. Saxena, Abhinav, Kai Goebel. "Turbofan Engine Degradation Simulation Data Set." *NASA Ames Prognostics Data Repository* https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/, NASA Ames Research Center, Moffett Field, CA

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer` | `predictAndUpdateState`

See Also

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Classify Videos Using Deep Learning

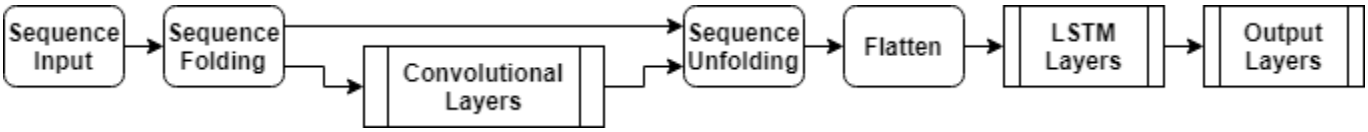
This example shows how to create a network for video classification by combining a pretrained image classification model and an LSTM network.

To create a deep learning network for video classification:

1. Convert videos to sequences of feature vectors using a pretrained convolutional neural network, such as GoogLeNet, to extract features from each frame.
2. Train an LSTM network on the sequences to predict the video labels.
3. Assemble a network that classifies videos directly by combining layers from both networks.

The following diagram illustrates the network architecture.

- To input image sequences to the network, use a sequence input layer.
- To use convolutional layers to extract features, that is, to apply the convolutional operations to each frame of the videos independently, use a sequence folding layer followed by the convolutional layers.
- To restore the sequence structure and reshape the output to vector sequences, use a sequence unfolding layer and a flatten layer.
- To classify the resulting vector sequences, include the LSTM layers followed by the output layers.



Load Pretrained Convolutional Network

To convert frames of videos to feature vectors, use the activations of a pretrained network.

Load a pretrained GoogLeNet model using the `googlenet` function. This function requires the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package. If this support package is not installed, then the function provides a download link.

```
netCNN = googlenet;
```

Load Data

Download the HMBD51 data set from HMDB: a large human motion database and extract the RAR file into a folder named `"hmdb51_org"`. The data set contains about 2 GB of video data for 7000 clips over 51 classes, such as `"drink"`, `"run"`, and `"shake_hands"`.

After extracting the RAR files, use the supporting function `hmdb51Files` to get the file names and the labels of the videos.

```
dataFolder = "hmdb51_org";
[files,labels] = hmdb51Files(dataFolder);
```

Read the first video using the `readVideo` helper function, defined at the end of this example, and view the size of the video. The video is a *H*-by-*W*-by-*C*-by-*S* array, where *H*, *W*, *C*, and *S* are the height, width, number of channels, and number of frames of the video, respectively.

```
idx = 1;
filename = files(idx);
video = readVideo(filename);
size(video)
```

```
ans = 1×4

   240   320     3   409
```

View the corresponding label.

```
labels(idx)
```

```
ans = categorical
     brush_hair
```

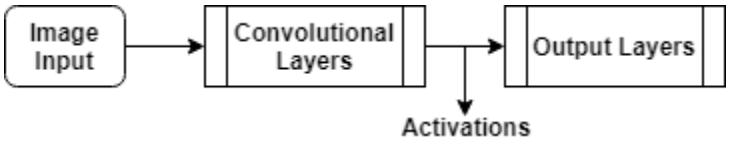
To view the video, use the `implay` function (requires Image Processing Toolbox™). This function expects data in the range [0,1], so you must first divide the data by 255. Alternatively, you can loop over the individual frames and use the `imshow` function.

```
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    imshow(frame/255);
    drawnow
end
```

Convert Frames to Feature Vectors

Use the convolutional network as a feature extractor by getting the activations when inputting the video frames to the network. Convert the videos to sequences of feature vectors, where the feature vectors are the output of the `activations` function on the last pooling layer of the GoogLeNet network (`"pool5-7x7_s1"`).

This diagram illustrates the data flow through the network.



To read the video data and resize it to match the input size of the GoogLeNet network, use the `readVideo` and `centerCrop` helper functions, defined at the end of this example. This step can take a long time to run. After converting the videos to sequences, save the sequences in a MAT-file in the `tempdir` folder. If the MAT file already exists, then load the sequences from the MAT-file without reconverting them.

```
inputSize = netCNN.Layers(1).InputSize(1:2);
layerName = "pool5-7x7_s1";

tempFile = fullfile(tempdir,"hmdb51_org.mat");

if exist(tempFile,'file')
    load(tempFile,"sequences")
```

```
else
    numFiles = numel(files);
    sequences = cell(numFiles,1);

    for i = 1:numFiles
        fprintf("Reading file %d of %d...\n", i, numFiles)

        video = readVideo(files(i));
        video = centerCrop(video,inputSize);

        sequences{i,1} = activations(netCNN,video,layerName,'OutputAs','columns');
    end

    save(tempFile,"sequences","-v7.3");
end
```

View the sizes of the first few sequences. Each sequence is a *D*-by-*S* array, where *D* is the number of features (the output size of the pooling layer) and *S* is the number of frames of the video.

```
sequences(1:10)
```

```
ans = 10×1 cell array
    {1024×409 single}
    {1024×395 single}
    {1024×323 single}
    {1024×246 single}
    {1024×159 single}
    {1024×137 single}
    {1024×359 single}
    {1024×191 single}
    {1024×439 single}
    {1024×528 single}
```

Prepare Training Data

Prepare the data for training by partitioning the data into training and validation partitions and removing any long sequences.

Create Training and Validation Partitions

Partition the data. Assign 90% of the data to the training partition and 10% to the validation partition.

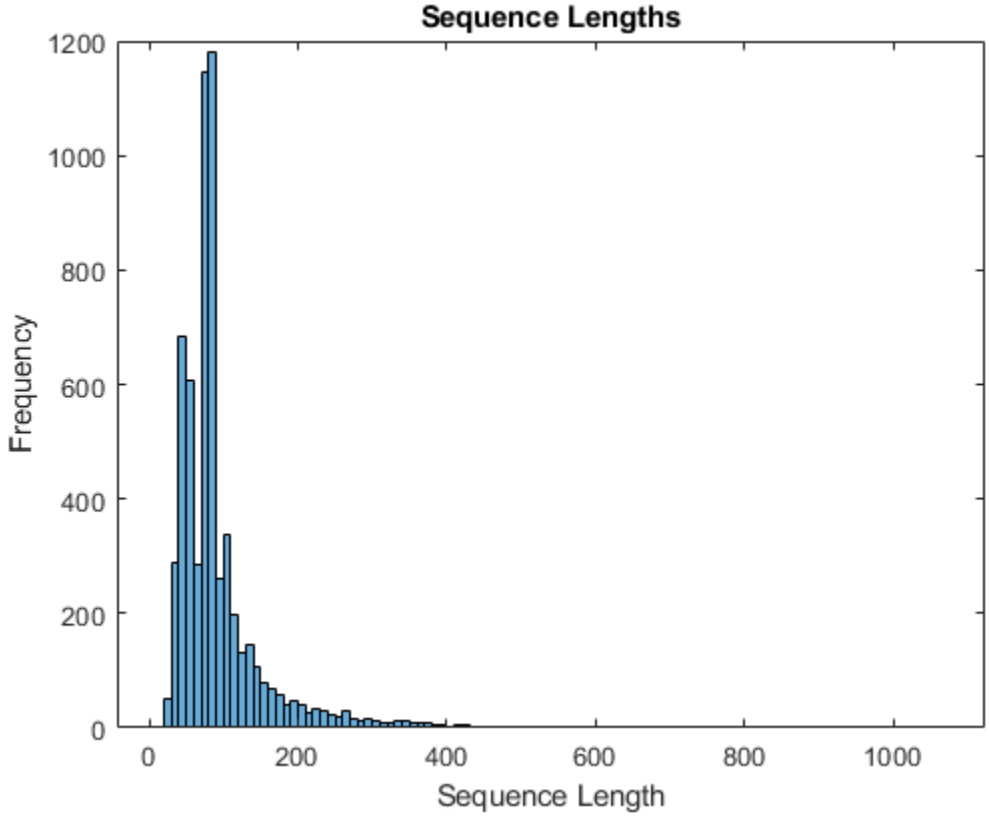
```
numObservations = numel(sequences);
idx = randperm(numObservations);
N = floor(0.9 * numObservations);

idxTrain = idx(1:N);
sequencesTrain = sequences(idxTrain);
labelsTrain = labels(idxTrain);

idxValidation = idx(N+1:end);
sequencesValidation = sequences(idxValidation);
labelsValidation = labels(idxValidation);
```

Remove Long Sequences

Sequences that are much longer than typical sequences in the networks can introduce lots of padding into the training process. Having too much padding can negatively impact the classification accuracy.

Get the sequence lengths of the training data and visualize them in a histogram of the training data.

```
numObservationsTrain = numel(sequencesTrain);
sequenceLengths = zeros(1,numObservationsTrain);

for i = 1:numObservationsTrain
    sequence = sequencesTrain{i};
    sequenceLengths(i) = size(sequence,2);
end

figure
histogram(sequenceLengths)
title("Sequence Lengths")
xlabel("Sequence Length")
ylabel("Frequency")
```



Only a few sequences have more than 400 time steps. To improve the classification accuracy, remove the training sequences that have more than 400 time steps along with their corresponding labels.

```
maxLength = 400;
idx = sequenceLengths > maxLength;
sequencesTrain(idx) = [];
labelsTrain(idx) = [];
```

Create LSTM Network

Next, create an LSTM network that can classify the sequences of feature vectors representing the videos.

Define the LSTM network architecture. Specify the following network layers.

- A sequence input layer with an input size corresponding to the feature dimension of the feature vectors
- A BiLSTM layer with 2000 hidden units with a dropout layer afterwards. To output only one label for each sequence by setting the `'OutputMode'` option of the BiLSTM layer to `'last'`
- A fully connected layer with an output size corresponding to the number of classes, a softmax layer, and a classification layer.

```
numFeatures = size(sequencesTrain{1},1);
numClasses = numel(categories(labelsTrain));

layers = [
    sequenceInputLayer(numFeatures,'Name','sequence')
    bilstmLayer(2000,'OutputMode','last','Name','bilstm')
    dropoutLayer(0.5,'Name','drop')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classification')];
```

Specify Training Options

Specify the training options using the `trainingOptions` function.

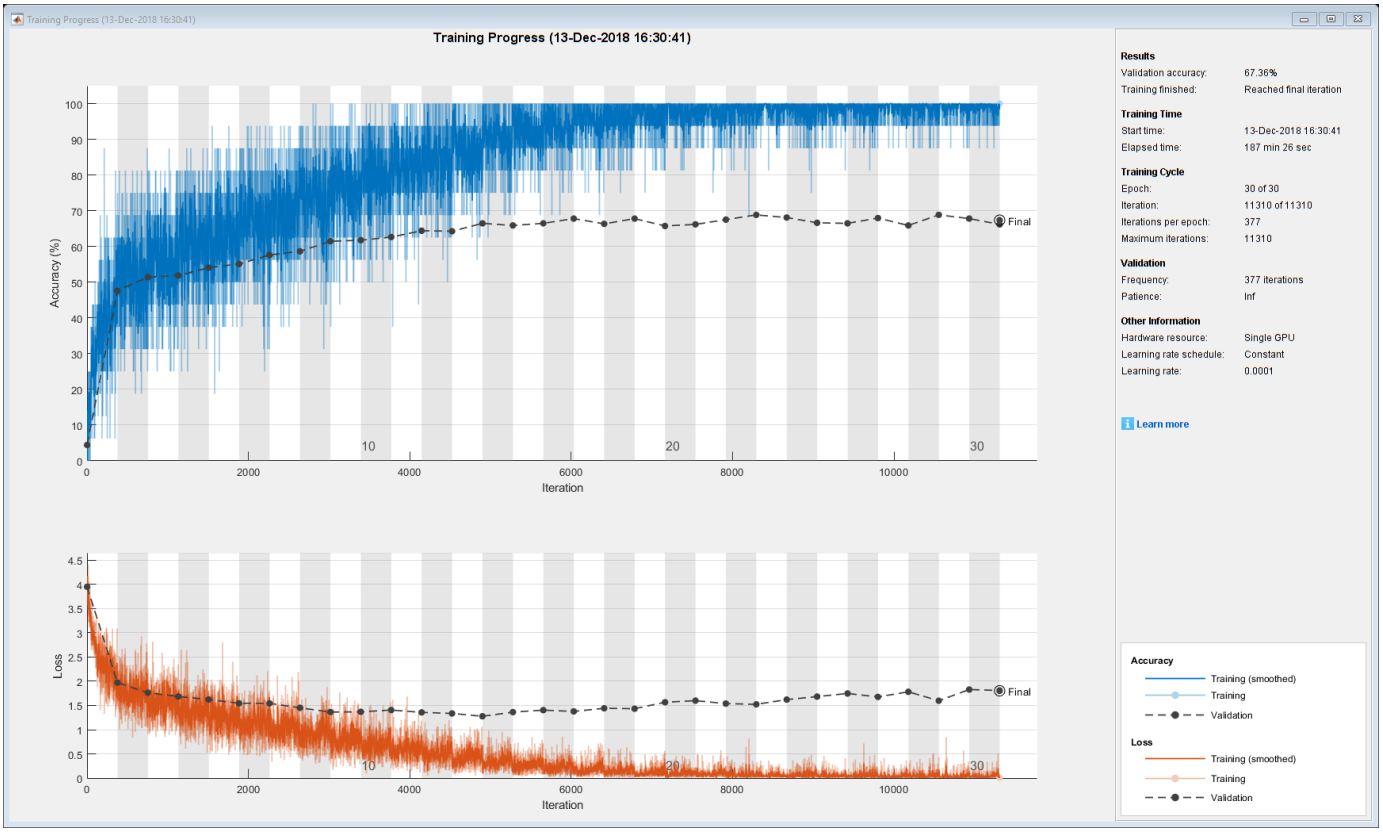
- Set a mini-batch size 16, an initial learning rate of 0.0001, and a gradient threshold of 2 (to prevent the gradients from exploding).
- Shuffle the data every epoch.
- Validate the network once per epoch.
- Display the training progress in a plot and suppress verbose output.

```
miniBatchSize = 16;
numObservations = numel(sequencesTrain);
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'InitialLearnRate',1e-4, ...
    'GradientThreshold',2, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{sequencesValidation,labelsValidation}, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train LSTM Network

Train the network using the `trainNetwork` function. This can take a long time to run.

```
[netLSTM,info] = trainNetwork(sequencesTrain,labelsTrain,layers,options);
```



Calculate the classification accuracy of the network on the validation set. Use the same mini-batch size as for the training options.

```
YPred = classify(netLSTM,sequencesValidation,'MiniBatchSize',miniBatchSize);
YValidation = labelsValidation;
accuracy = mean(YPred == YValidation)
```

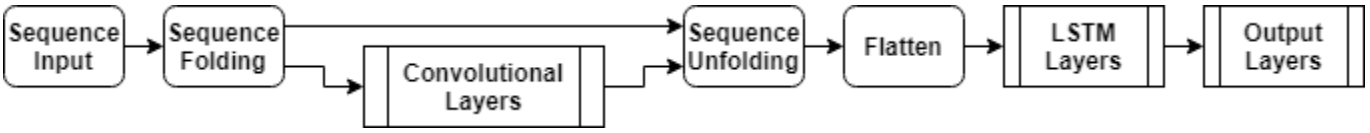
```
accuracy = 0.6647
```

Assemble Video Classification Network

To create a network that classifies videos directly, assemble a network using layers from both of the created networks. Use the layers from the convolutional network to transform the videos into vector sequences and the layers from the LSTM network to classify the vector sequences.

The following diagram illustrates the network architecture.

- To input image sequences to the network, use a sequence input layer.
- To use convolutional layers to extract features, that is, to apply the convolutional operations to each frame of the videos independently, use a sequence folding layer followed by the convolutional layers.
- To restore the sequence structure and reshape the output to vector sequences, use a sequence unfolding layer and a flatten layer.
- To classify the resulting vector sequences, include the LSTM layers followed by the output layers.



Add Convolutional Layers

First, create a layer graph of the GoogLeNet network.

```
cnnLayers = layerGraph(netCNN);
```

Remove the input layer (`"data"`) and the layers after the pooling layer used for the activations (`"pool5-drop_7x7_s1"`, `"loss3-classifier"`, `"prob"`, and `"output"`).

```
layerNames = ["data" "pool5-drop_7x7_s1" "loss3-classifier" "prob" "output"];
cnnLayers = removeLayers(cnnLayers,layerNames);
```

Add Sequence Input Layer

Create a sequence input layer that accepts image sequences containing images of the same input size as the GoogLeNet network. To normalize the images using the same average image as the GoogLeNet network, set the `'Normalization'` option of the sequence input layer to `'zerocenter'` and the `'Mean'` option to the average image of the input layer of GoogLeNet.

```
inputSize = netCNN.Layers(1).InputSize(1:2);
averageImage = netCNN.Layers(1).Mean;

inputLayer = sequenceInputLayer([inputSize 3], ...
    'Normalization','zerocenter', ...
    'Mean',averageImage, ...
    'Name','input');
```

Add the sequence input layer to the layer graph. To apply the convolutional layers to the images of the sequences independently, remove the sequence structure of the image sequences by including a sequence folding layer between the sequence input layer and the convolutional layers. Connect the output of the sequence folding layer to the input of the first convolutional layer (`"conv1-7x7_s2"`).

```
layers = [
    inputLayer
    sequenceFoldingLayer('Name','fold')];

lgraph = addLayers(cnnLayers,layers);
lgraph = connectLayers(lgraph,"fold/out","conv1-7x7_s2");
```

Add LSTM Layers

Add the LSTM layers to the layer graph by removing the sequence input layer of the LSTM network. To restore the sequence structure removed by the sequence folding layer, include a sequence unfolding layer after the convolution layers. The LSTM layers expect sequences of vectors. To reshape the output of the sequence unfolding layer to vector sequences, include a flatten layer after the sequence unfolding layer.

Take the layers from the LSTM network and remove the sequence input layer.

```
lstmLayers = netLSTM.Layers;
lstmLayers(1) = [];
```

Add the sequence unfolding layer, the flatten layer, and the LSTM layers to the layer graph. Connect the last convolutional layer (`"pool5-7x7_s1"`) to the input of the sequence unfolding layer (`"unfold/in"`).

```
layers = [
    sequenceUnfoldingLayer('Name','unfold')
    flattenLayer('Name','flatten')
    lstmLayers];

lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,"pool5-7x7_s1","unfold/in");
```

To enable the unfolding layer to restore the sequence structure, connect the `"miniBatchSize"` output of the sequence folding layer to the corresponding input of the sequence unfolding layer.

```
lgraph = connectLayers(lgraph,"fold/miniBatchSize","unfold/miniBatchSize");
```

Assemble Network

Check that the network is valid using the `analyzeNetwork` function.

```
analyzeNetwork(lgraph)
```

Assemble the network so that it is ready for prediction using the `assembleNetwork` function.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [148×1 nnet.cnn.layer.Layer]
    Connections: [175×2 table]
```

Classify Using New Data

Read and center-crop the video `"pushup.mp4"` using the same steps as before.

```
filename = "pushup.mp4";
video = readVideo(filename);
```

To view the video, use the `implay` function (requires Image Processing Toolbox). This function expects data in the range [0,1], so you must first divide the data by 255. Alternatively, you can loop over the individual frames and use the `imshow` function.

```
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    imshow(frame/255);
    drawnow
end
```

Classify the video using the assembled network. The `classify` function expects a cell array containing the input videos, so you must input a 1-by-1 cell array containing the video.

```
video = centerCrop(video,inputSize);
YPred = classify(net,{video})
```

```
YPred = categorical
     pushup
```

Helper Functions

The `readVideo` function reads the video in `filename` and returns an `H`-by-`W`-by-`C`-by-`S` array, where `H`, `W`, `C`, and `S` are the height, width, number of channels, and number of frames of the video, respectively.

```
function video = readVideo(filename)

vr = VideoReader(filename);
H = vr.Height;
W = vr.Width;
C = 3;

% Preallocate video array
numFrames = floor(vr.Duration * vr.FrameRate);
video = zeros(H,W,C,numFrames);

% Read frames
i = 0;
while hasFrame(vr)
```

```
    i = i + 1;
    video(:,:,:,i) = readFrame(vr);
end

% Remove unallocated frames
if size(video,4) > i
    video(:,:,:,i+1:end) = [];
end

end
```

The `centerCrop` function crops the longest edges of a video and resizes it have size `inputSize`.

```
function videoResized = centerCrop(video,inputSize)

sz = size(video);

if sz(1) < sz(2)
    % Video is landscape
    idx = floor((sz(2) - sz(1))/2);
    video(:,1:(idx-1),:,:) = [];
    video(:,(sz(1)+1):end,:,:) = [];

elseif sz(2) < sz(1)
    % Video is portrait
    idx = floor((sz(1) - sz(2))/2);
    video(1:(idx-1),:,:,:) = [];
    video((sz(2)+1):end,:,:,:) = [];
end

videoResized = imresize(video,inputSize(1:2));

end
```

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer` | `sequenceFoldingLayer` | `sequenceUnfoldingLayer` | `flattenLayer`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Classify Videos Using Deep Learning with Custom Training Loop” on page 4-64
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Classify Videos Using Deep Learning with Custom Training Loop

This example shows how to create a network for video classification by combining a pretrained image classification model and a sequence classification network.

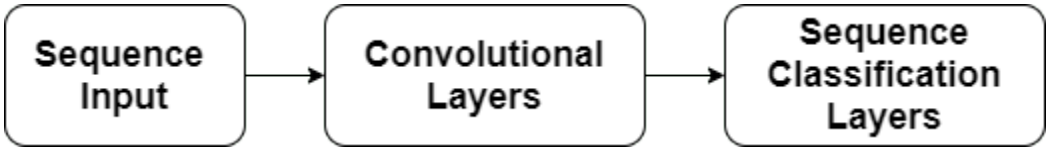
You can perform video classification without using a custom training loop by using the `trainNetwork` function. For an example, see “Classify Videos Using Deep Learning” on page 4-54. However, If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop as shown in this example.

To create a deep learning network for video classification:

1. Convert videos to sequences of feature vectors using a pretrained convolutional neural network, such as GoogLeNet, to extract features from each frame.
2. Train a sequence classification network on the sequences to predict the video labels.
3. Assemble a network that classifies videos directly by combining layers from both networks.

The following diagram illustrates the network architecture:

- To input image sequences to the network, use a sequence input layer.
- To extract features from the image sequences, use convolutional layers from the pretrained GoogLeNet network.
- To classify the resulting vector sequences, include the sequence classification layers.



When training this type of network with the `trainNetwork` function (not done in this example), you must use sequence folding and unfolding layers to process the video frames independently. When you train this type of network with a `dlnetwork` object and a custom training loop (as in this example), sequence folding and unfolding layers are not required because the network uses dimension information given by the `dlarray` dimension labels.

Load Pretrained Convolutional Network

To convert frames of videos to feature vectors, use the activations of a pretrained network.

Load a pretrained GoogLeNet model using the `googlenet` function. This function requires the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package. If this support package is not installed, then the function provides a download link.

```
netCNN = googlenet;
```

Load Data

Download the HMBD51 data set from HMDB: a large human motion database and extract the RAR file into a folder named `"hmdb51_org"`. The data set contains about 2 GB of video data for 7000 clips over 51 classes, such as `"drink"`, `"run"`, and `"shake_hands"`.

After extracting the RAR file, make sure that the folder `hmdb51_org` contains subfolders named after the body motions. If it contains RAR files, you need to extract them as well. Use the supporting function `hmdb51Files` to get the file names and the labels of the videos. To speed up training at the cost of accuracy, specify a fraction in the range [0 1] to read only a random subset of files from the database. If the `fraction` input argument is not specified, the function `hmdb51Files` reads the full dataset without changing the order of the files.

```
dataFolder = "hmdb51_org";
fraction = 1;
[files,labels] = hmdb51Files(dataFolder,fraction);
```

Read the first video using the `readVideo` helper function, defined at the end of this example, and view the size of the video. The video is an *H*-by-*W*-by-*C*-by-*T* array, where *H*, *W*, *C*, and *T* are the height, width, number of channels, and number of frames of the video, respectively.

```
idx = 1;
filename = files(idx);
video = readVideo(filename);
size(video)
```

```
ans = 1×4

   240   352     3   115
```

View the corresponding label.

```
labels(idx)
```

```
ans = categorical
     shoot_ball
```

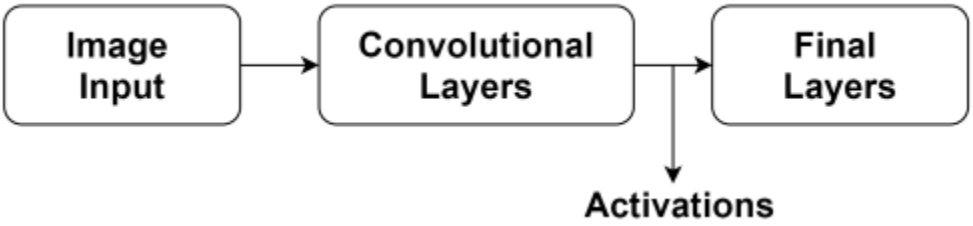
To view the video, loop over the individual frames and use the `image` function. Alternatively, you can use the `implay` function (requires Image Processing Toolbox).

```
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    image(frame);
    xticklabels([]);
    yticklabels([]);
    drawnow
end
```

Convert Frames to Feature Vectors

Use the convolutional network as a feature extractor: input video frames to the network and extract the activations. Convert the videos to sequences of feature vectors, where the feature vectors are the output of the `activations` function on the last pooling layer of the GoogLeNet network (`"pool5-7x7_s1"`).

This diagram illustrates the data flow through the network.



Read the video data using the `readVideo` function, defined at the end of this example, and resize it to match the input size of the GoogLeNet network. Note that this step can take a long time to run. After converting the videos to sequences, save the sequences and corresponding labels in a MAT file in the `tempdir` folder. If the MAT file already exists, then load the sequences and labels from the MAT file directly. In case a MAT file already exists but you want to overwrite it, set the variable `overwriteSequences` to `true`.

```
inputSize = netCNN.Layers(1).InputSize(1:2);
layerName = "pool5-7x7_s1";

tempFile = fullfile(tempdir,"hmdb51_org.mat");

overwriteSequences = false;

if exist(tempFile,'file') && ~overwriteSequences
    load(tempFile)
else
    numFiles = numel(files);
    sequences = cell(numFiles,1);

    for i = 1:numFiles
        fprintf("Reading file %d of %d...\n", i, numFiles)

        video = readVideo(files(i));
        video = imresize(video,inputSize);
        sequences{i,1} = activations(netCNN,video,layerName,'OutputAs','columns');

    end
    % Save the sequences and the labels associated with them.
    save(tempFile,"sequences","labels","-v7.3");
end
```

View the sizes of the first few sequences. Each sequence is a *D*-by-*T* array, where *D* is the number of features (the output size of the pooling layer) and *T* is the number of frames of the video.

```
sequences(1:10)
```

```
ans=10×1 cell array
    {1024×115 single}
    {1024×227 single}
    {1024×180 single}
    {1024×40  single}
    {1024×60  single}
    {1024×156 single}
    {1024×83  single}
    {1024×42  single}
    {1024×82  single}
```

```
    {1024×110 single}
```

Prepare Training Data

Prepare the data for training by partitioning the data into training and validation partitions and removing any long sequences.

Create Training and Validation Partitions

Partition the data. Assign 90% of the data to the training partition and 10% to the validation partition.

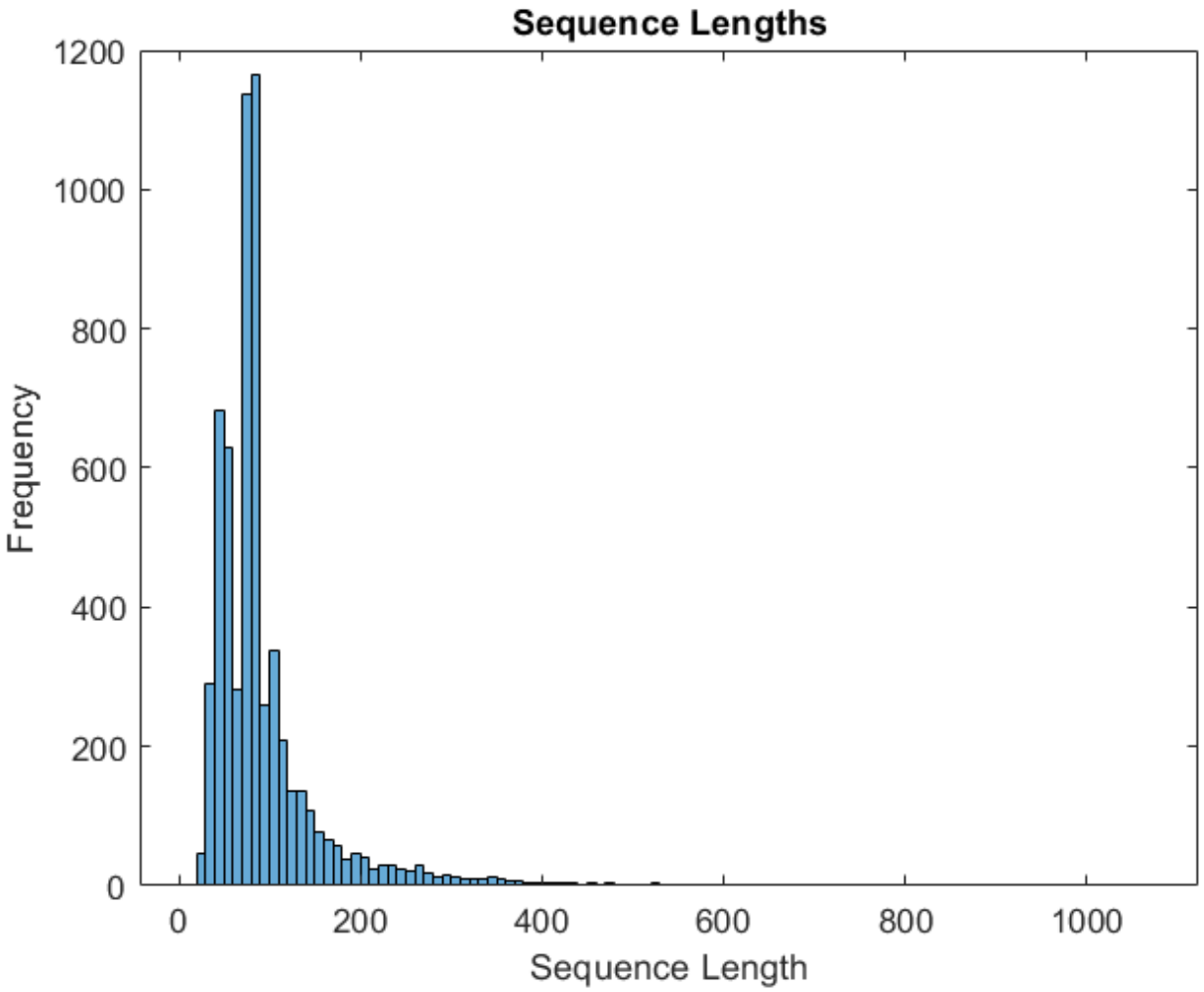
```
numObservations = numel(sequences);
idx = randperm(numObservations);
N = floor(0.9 * numObservations);

idxTrain = idx(1:N);
sequencesTrain = sequences(idxTrain);
labelsTrain = labels(idxTrain);

idxValidation = idx(N+1:end);
sequencesValidation = sequences(idxValidation);
labelsValidation = labels(idxValidation);
```

Remove Long Sequences

Sequences that are much longer than typical sequences in the networks can introduce lots of padding into the training process. Having too much padding can negatively impact the classification accuracy.

Get the sequence lengths of the training data and visualize them in a histogram of the training data.

```
numObservationsTrain = numel(sequencesTrain);
sequenceLengths = zeros(1,numObservationsTrain);

for i = 1:numObservationsTrain
    sequence = sequencesTrain{i};
    sequenceLengths(i) = size(sequence,2);
end

figure
histogram(sequenceLengths)
title("Sequence Lengths")
xlabel("Sequence Length")
ylabel("Frequency")
```



Only a few sequences have more than 400 time steps. To improve the classification accuracy, remove the training sequences that have more than 400 time steps along with their corresponding labels.

```
maxLength = 400;
idx = sequenceLengths > maxLength;
sequencesTrain(idx) = [];
labelsTrain(idx) = [];
```

Create Datastore for Data

Create an `arrayDatastore` object for the sequences and the labels, and then combine them into a single datastore.

```
dsXTrain = arrayDatastore(sequencesTrain,'OutputType','same');
dsYTrain = arrayDatastore(labelsTrain,'OutputType','cell');

dsTrain = combine(dsXTrain,dsYTrain);
```

Determine the classes in the training data.

```
classes = categories(labelsTrain);
```

Create Sequence Classification Network

Next, create a sequence classification network that can classify the sequences of feature vectors representing the videos.

Define the sequence classification network architecture. Specify the following network layers:

- A sequence input layer with an input size corresponding to the feature dimension of the feature vectors.
- A BiLSTM layer with 2000 hidden units with a dropout layer afterwards. To output only one label for each sequence, set the `'OutputMode'` option of the BiLSTM layer to `'last'`.
- A dropout layer with a probability of 0.5.
- A fully connected layer with an output size corresponding to the number of classes and a softmax layer.

```
numFeatures = size(sequencesTrain{1},1);
numClasses = numel(categories(labelsTrain));

layers = [
    sequenceInputLayer(numFeatures,'Name','sequence')
    bilstmLayer(2000,'OutputMode','last','Name','bilstm')
    dropoutLayer(0.5,'Name','drop')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')
    ];
```

Convert the layers to a `layerGraph` object.

```
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph);
```

Specify Training Options

Train for 15 epochs and specify a mini-batch size of 16.

```
numEpochs = 15;
miniBatchSize = 16;
```

Specify the options for Adam optimization. Specify an initial learning rate of `1e-4` with a decay of 0.001, a gradient decay of 0.9, and a squared gradient decay of 0.999.

```
initialLearnRate = 1e-4;
decay = 0.001;
gradDecay = 0.9;
sqGradDecay = 0.999;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

Train Sequence Classification Network

Create a `minibatchqueue` object that processes and manages mini-batches of sequences during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessLabeledSequences` (defined at the end of this example) to convert the labels to dummy variables.
- Format the vector sequence data with the dimension labels `'CTB'` (channel, time, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` object if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessLabeledSequences,...
    'MiniBatchFormat',{'CTB',''});
```

Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Initialize the average gradient and average squared gradient parameters for the Adam solver.

```
averageGrad = [];
averageSqGrad = [];
```

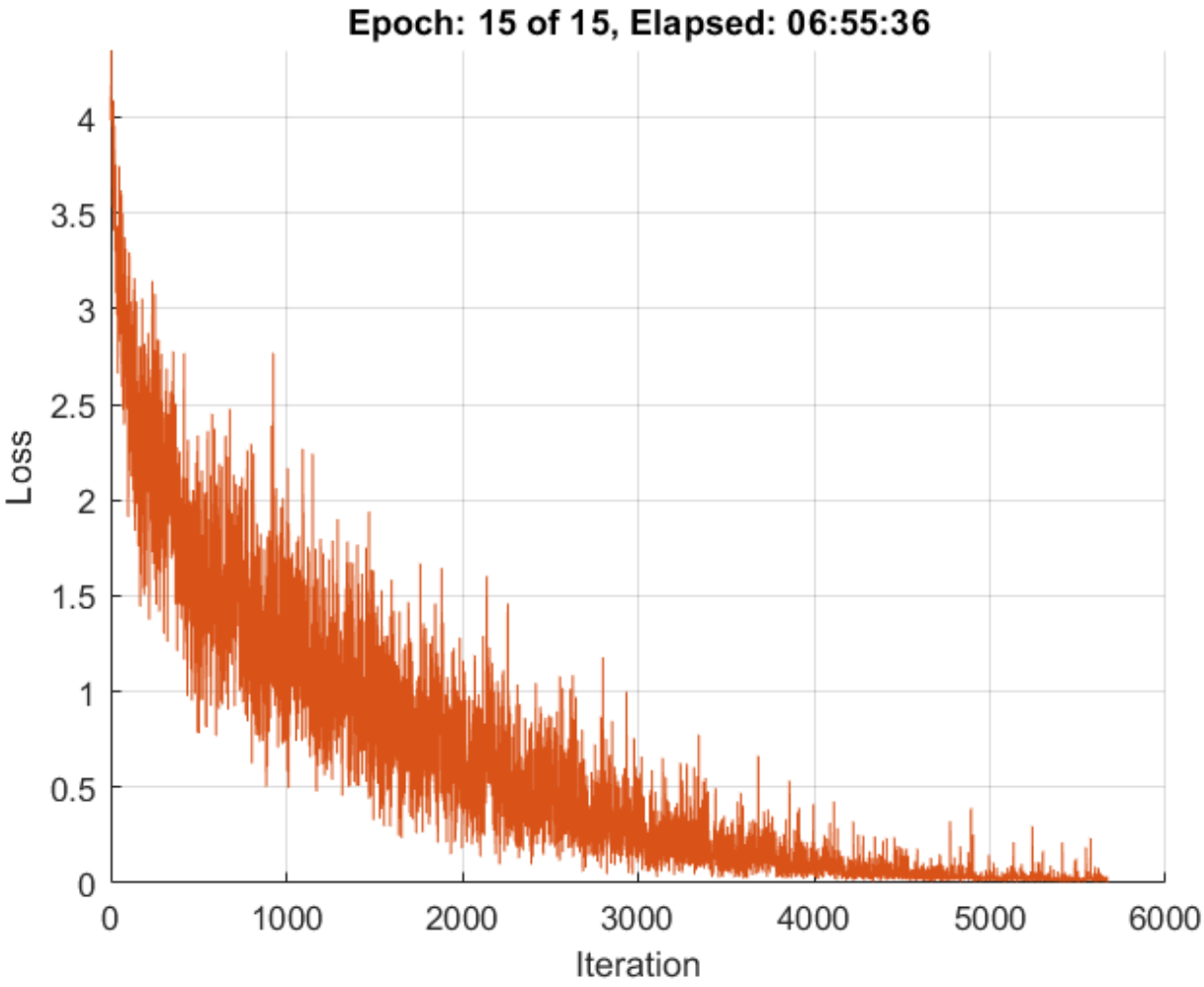
Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using `dlfeval` and the `modelGradients` function and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + k\,t}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.
- Update the network parameters using the `adamupdate` function.
- Display the training progress.

Note that training can take a long time to run.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
```

```
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX, dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the Adam optimizer.
        [dlnet,averageGrad,averageSqGrad] = adamupdate(dlnet,gradients,averageGrad,averageSqGrad
            iteration,learnRate,gradDecay,sqGradDecay);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + " of " + numEpochs + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

After training is complete, making predictions on new data does not require the labels.

To create a `minibatchqueue` object for testing:

- Create an array datastore containing only the predictors of the test data.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessUnlabeledSequences` helper function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'CTB'` (channel, time, batch).

```
dsXValidation = arrayDatastore(sequencesValidation,'OutputType','same');
mbqTest = minibatchqueue(dsXValidation, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessUnlabeledSequences, ...
    'MiniBatchFormat','CTB');
```

Loop over the mini-batches and classify the images using the `modelPredictions` helper function, listed at the end of the example.

```
predictions = modelPredictions(dlnet,mbqTest,classes);
```

Evaluate the classification accuracy by comparing the predicted labels to the true validation labels.

```
accuracy = mean(predictions == labelsValidation)
```

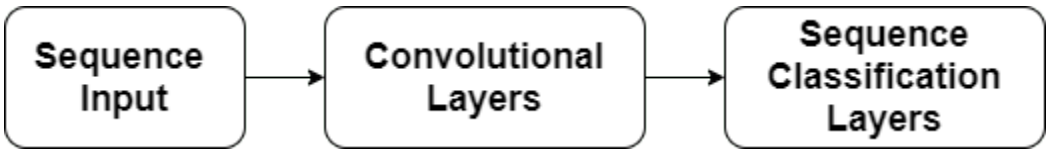
```
accuracy = 0.6721
```

Assemble Video Classification Network

To create a network that classifies videos directly, assemble a network using layers from both of the created networks. Use the layers from the convolutional network to transform the videos into vector sequences and the layers from the sequence classification network to classify the vector sequences.

The following diagram illustrates the network architecture:

- To input image sequences to the network, use a sequence input layer.
- To use convolutional layers to extract features, that is, to apply the convolutional operations to each frame of the videos independently, use the GoogLeNet convolutional layers.
- To classify the resulting vector sequences, include the sequence classification layers.



When training this type of network with the `trainNetwork` function (not done in this example), you have to use sequence folding and unfolding layers to process the video frames independently. When training this type of network with a `dlnetwork` object and a custom training loop (as in this example), sequence folding and unfolding layers are not required because the network uses dimension information given by the `dlarray` dimension labels.

Add Convolutional Layers

First, create a layer graph of the GoogLeNet network.

```
cnnLayers = layerGraph(netCNN);
```

Remove the input layer (`"data"`) and the layers after the pooling layer used for the activations (`"pool5-drop_7x7_s1"`, `"loss3-classifier"`, `"prob"`, and `"output"`).

```
layerNames = ["data" "pool5-drop_7x7_s1" "loss3-classifier" "prob" "output"];
cnnLayers = removeLayers(cnnLayers,layerNames);
```

Add Sequence Input Layer

Create a sequence input layer that accepts image sequences containing images of the same input size as the GoogLeNet network. To normalize the images using the same average image as the GoogLeNet network, set the `'Normalization'` option of the sequence input layer to `'zerocenter'` and the `'Mean'` option to the average image of the input layer of GoogLeNet.

```
inputSize = netCNN.Layers(1).InputSize(1:2);
averageImage = netCNN.Layers(1).Mean;
```

```
inputLayer = sequenceInputLayer([inputSize 3], ...
    'Normalization','zerocenter', ...
    'Mean',averageImage, ...
    'Name','input');
```

Add the sequence input layer to the layer graph. Connect the output of the input layer to the input of the first convolutional layer (`"conv1-7x7_s2"`).

```
lgraph = addLayers(cnnLayers,inputLayer);
lgraph = connectLayers(lgraph,"input","conv1-7x7_s2");
```

Add Sequence Classification Layers

Add the previously trained sequence classification network layers to the layer graph and connect them.

Take the layers from the sequence classification network and remove the sequence input layer.

```
lstmLayers = dlnet.Layers;
lstmLayers(1) = [];
```

Add the sequence classification layers to the layer graph. Connect the last convolutional layer `pool5-7x7_s1` to the `bilstm` layer.

```
lgraph = addLayers(lgraph,lstmLayers);
lgraph = connectLayers(lgraph,"pool5-7x7_s1","bilstm");
```

Convert to `dlnetwork`

To be able to do predictions, convert the layer graph to a `dlnetwork` object.

```
dlnetAssembled = dlnetwork(lgraph)
```

```
dlnetAssembled =
  dlnetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
     Learnables: [119×3 table]
          State: [2×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
    Initialized: 1
```

Classify Using New Data

Unzip the file `pushup_mathworker.zip`.

```
unzip("pushup_mathworker.zip")
```

The extracted `pushup_mathworker` folder contains a video of a push-up. Create a file datastore for this folder. Use a custom read function to read the videos.

```
ds = fileDatastore("pushup_mathworker", ...
    'ReadFcn',@readVideo);
```

Read the first video from the datastore. To be able to read the video again, reset the datastore.

```
video = read(ds);
reset(ds);
```

To view the video, loop over the individual frames and use the `image` function. Alternatively, you can use the `implay` function (requires Image Processing Toolbox).

```
numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:,:,i);
    image(frame);
    xticklabels([]);
    yticklabels([]);
    drawnow
end
```

To preprocess the videos to have the input size expected by the network, use the `transform` function and apply the `imresize` function to each image in the datastore.

```
dsXTest = transform(ds,@(x) imresize(x,inputSize));
```

To manage and process the unlabeled videos, create a `minibatchqueue`:

- Specify a mini-batch size of 1.
- Preprocess the videos using the `preprocessUnlabeledVideos` helper function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCTB'` (spatial, spatial, channel, time, batch).

```
mbqTest = minibatchqueue(dsXTest,...
    'MiniBatchSize',1,...
    'MiniBatchFcn', @preprocessUnlabeledVideos,...
    'MiniBatchFormat',{'SSCTB'});
```

Classify the videos using the `modelPredictions` helper function, defined at the end of this example. The function expects three inputs: a `dlnetwork` object, a `minibatchqueue` object, and a cell array containing the network classes.

```
[predictions] = modelPredictions(dlnetAssembled,mbqTest,classes)
```

```
predictions = categorical
     pushup
```

Helper Functions

Video Reading Function

The `readVideo` function reads the video in `filename` and returns an *H*-by-*W*-by-*C*-by-*T* array, where *H*, *W*, *C*, and *T* are the height, width, number of channels, and number of frames of the video, respectively.

```
function video = readVideo(filename)

vr = VideoReader(filename);
H = vr.Height;
W = vr.Width;
C = 3;

% Preallocate video array
numFrames = floor(vr.Duration * vr.FrameRate);
video = zeros(H,W,C,numFrames,'uint8');

% Read frames
i = 0;
while hasFrame(vr)
    i = i + 1;
    video(:,:,:,i) = readFrame(vr);
end

% Remove unallocated frames
if size(video,4) > i
    video(:,:,:,i+1:end) = [];
end

end
```

Model Gradients Function

The `modelGradients` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,Y)

    [dlYPred,state] = forward(dlnet,dlX);

    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);

end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` object of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the mini-batch queue. The function uses the `onehotdecode` function to find the predicted class with the highest score. The function returns the predicted labels.

```
function [predictions] = modelPredictions(dlnet,mbq,classes)
    predictions = [];

    while hasdata(mbq)

        % Extract a mini-batch from the minibatchqueue and pass it to the
        % network for predictions
        [dlXTest] = next(mbq);
        dlYPred = predict(dlnet,dlXTest);

        % To obtain categorical labels, one-hot decode the predictions
        YPred = onehotdecode(dlYPred,classes,1)';
        predictions = [predictions; YPred];
    end
end
```

Labeled Sequence Data Preprocessing Function

The `preprocessLabeledSequences` function preprocesses the sequence data using the following steps:

1. Use the `padsequences` function to pad the sequences in the time dimension and concatenate them in the batch dimension.
2. Extract the label data from the incoming cell array and concatenate into a categorical array.
3. One-hot encode the categorical labels into numeric arrays.
4. Transpose the array of one-hot encoded labels to match the shape of the network output.

```
function [X, Y] = preprocessLabeledSequences(XCell,YCell)
    % Pad the sequences with zeros in the second dimension (time) and concatenate along the third
    % dimension (batch)
    X = padsequences(XCell,2);

    % Extract label data from cell and concatenate
```

```
    Y = cat(1,YCell{1:end});

    % One-hot encode labels
    Y = onehotencode(Y,2);

    % Transpose the encoded labels to match the network output
    Y = Y';
end
```

Unlabeled Sequence Data Preprocessing Function

The `preprocessUnlabeledSequences` function preprocesses the sequence data using the `padsequences` function. This function pads the sequences with zeros in the time dimension and concatenates the result in the batch dimension.

```
function [X] = preprocessUnlabeledSequences(XCell)
    % Pad the sequences with zeros in the second dimension (time) and concatenate along the third
    % dimension (batch)
    X = padsequences(XCell,2);
end
```

Unlabeled Video Data Preprocessing Function

The `preprocessUnlabeledVideos` function preprocesses unlabeled video data using the `padsequences` function. This function pads the videos with zero in the time dimension and concatenates the result in the batch dimension.

```
function [X] = preprocessUnlabeledVideos(XCell)
    % Pad the sequences with zeros in the fourth dimension (time) and
    % concatenate along the fifth dimension (batch)
    X = padsequences(XCell,4);
end
```

See Also

`lstmLayer` | `sequenceInputLayer` | `dlfeval` | `dlgradient` | `dlarray`

Related Examples

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Classify Videos Using Deep Learning” on page 4-54
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Sequence-to-Sequence Classification Using 1-D Convolutions

This example shows how to classify each time step of sequence data using a generic temporal convolutional network (TCN).

While sequence-to-sequence tasks are commonly solved with recurrent neural network architectures, Bai et al. [1] show that convolutional neural networks can match the performance of recurrent networks on typical sequence modeling tasks or even outperform them. Potential benefits of using convolutional networks are better parallelism, better control over the receptive field size, better control of the memory footprint of the network during training, and more stable gradients. Just like recurrent networks, convolutional networks can operate on variable length input sequences and can be used to model sequence-to-sequence or sequence-to-one tasks.

In this example, you train a TCN to recognize the activity of person wearing a smartphone on the body. You train the network using time series data representing accelerometer readings in three directions.

Load Training Data

Load the Human Activity Recognition data. The data contains seven time series of sensor data obtained from a smartphone worn on the body. Each sequence has three features and varies in length. The three features correspond to accelerometer readings in three directions.

```
s = load("HumanActivityTrain.mat");
XTrain = s.XTrain;
TTrain = s.YTrain;
```

View the number of observations in the training data.

```
numObservations = numel(XTrain)
```

```
numObservations = 6
```

View the number of classes in the training data.

```
classes = categories(TTrain{1});
numClasses = numel(classes)
```

```
numClasses = 5
```

View the number of features of the training data.

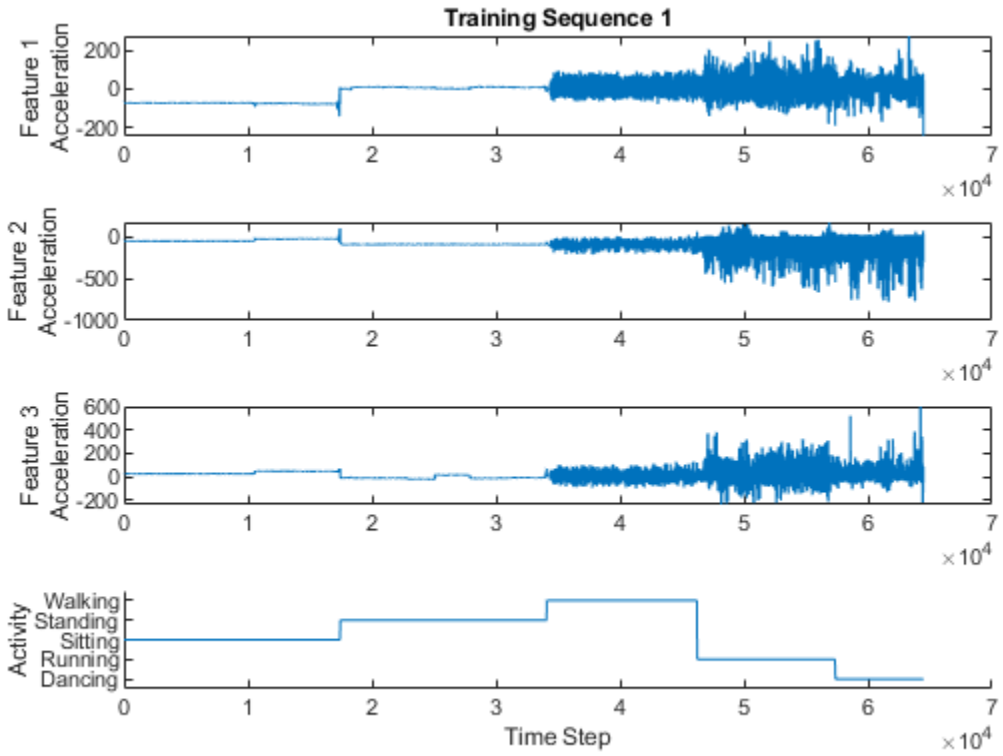
```
numFeatures = size(s.XTrain{1},1)
```

```
numFeatures = 3
```

Visualize one of the training sequences in a plot. Plot the features of the first training sequence and the corresponding activity.

```
figure
for i = 1:3
    X = s.XTrain{1}(i,:);

    subplot(4,1,i)
    plot(X)
    ylabel("Feature " + i + newline + "Acceleration")
end
```

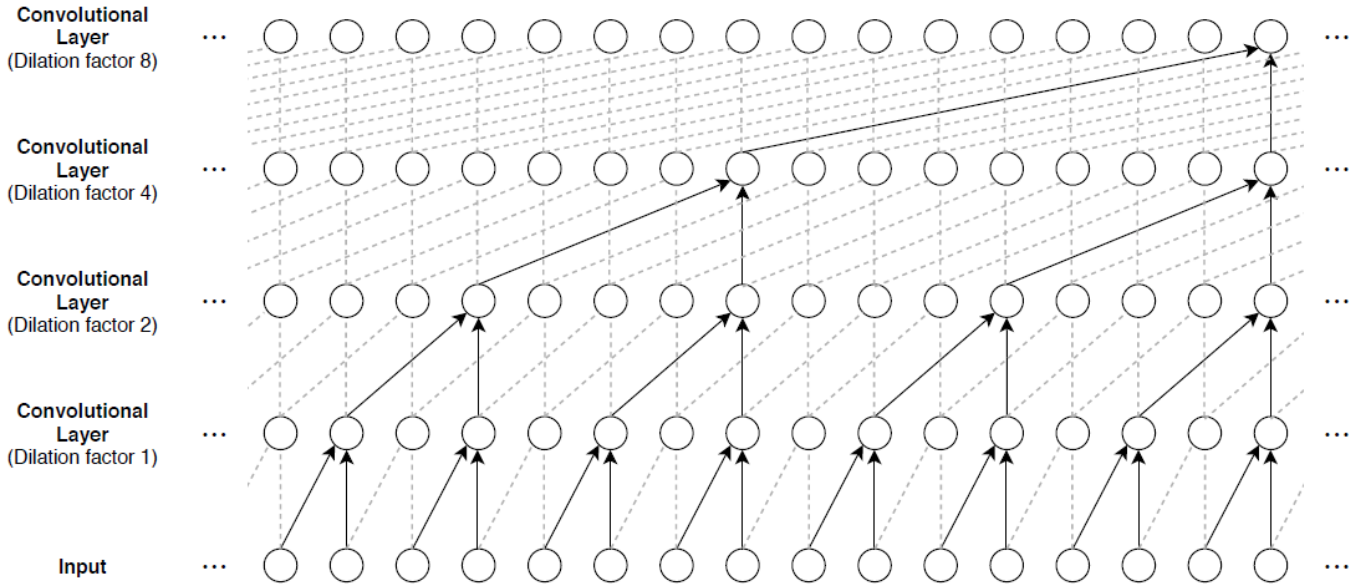
```
subplot(4,1,4)

hold on
plot(s.YTrain{1})
hold off

xlabel("Time Step")
ylabel("Activity")

subplot(4,1,1)
title("Training Sequence 1")
```



Define Deep Learning Model

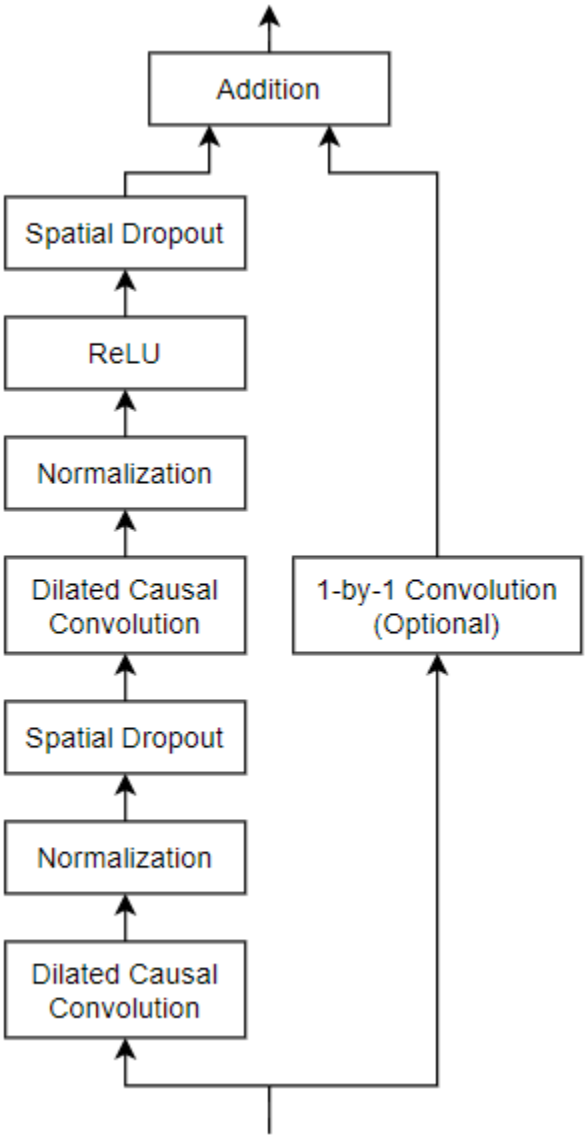
The main building block of a TCN is a dilated causal convolution layer, which operates over the time steps of each sequence. In this context, "causal" means that the activations computed for a particular time step cannot depend on activations from future time steps.

To build up context from previous time steps, multiple convolutional layers are typically stacked on top of each other. To achieve large receptive field sizes, the dilation factor of subsequent convolution layers is increased exponentially, as shown in the following image. Assuming that the dilation factor of the k-th convolutional layer is $2^{(k-1)}$ and the stride is 1, then the receptive field size of such a network can be computed as $R = (f-1)\left(2^K - 1\right) + 1$, where f is the filter size and K is the number of convolutional layers. Change the filter size and number of layers to easily adjust the receptive field size and the number of learnable parameters as necessary for the data and task at hand.



One of the disadvantages of TCNs compared to recurrent networks is that they have a larger memory footprint during inference. The entire raw sequence is required to compute the next time step. To reduce inference time and memory consumption, especially for step-ahead predictions, train with the smallest sensible receptive field size R and perform prediction only with the last R time steps of the input sequence.

The general TCN architecture (as described in [1]) consists of multiple residual blocks, each containing two sets of dilated causal convolution layers with the same dilation factor, followed by normalization, ReLU activation, and spatial dropout layers. The network adds the input of each block to the output of the block (including a 1-by-1 convolution on the input when the number of channels between the input and output do not match) and applies a final activation function.



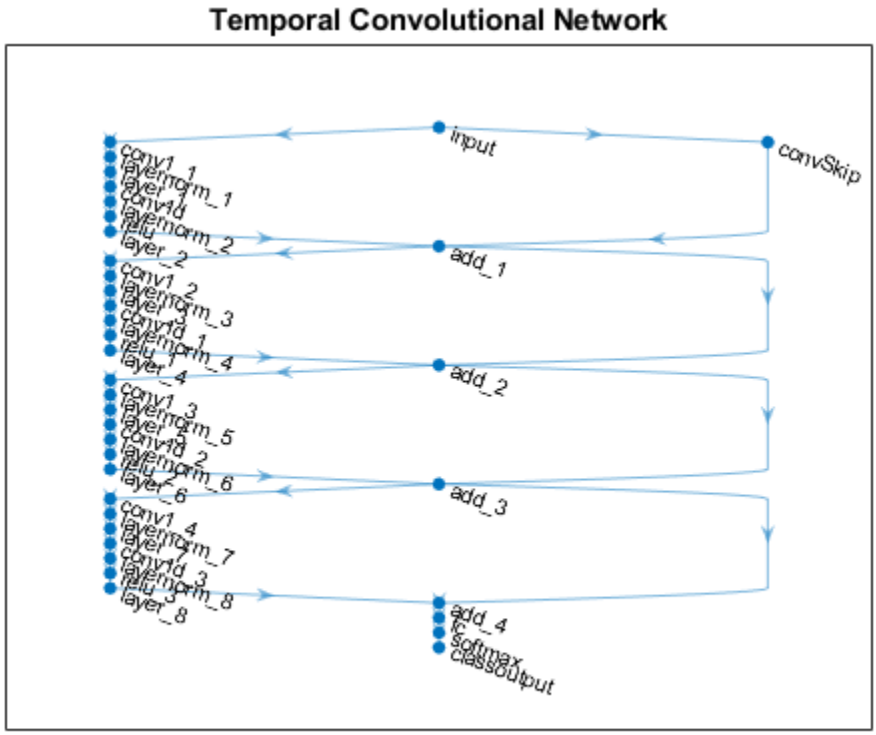
Define a network containing four of these residual blocks in series, each with double the dilation factor of the previous layer, starting with a dilation factor of 1. For the residual blocks, specify 64 filters for the 1-D convolutional layers with a filter size of 5 and a dropout factor of 0.005 for the spatial dropout layers. For spatial dropout, use the custom layer `spatialDropoutLayer`, attached to this example as a supporting file. To access this layer, open this example as a live script.

```
numFilters = 64;
filterSize = 5;
dropoutFactor = 0.005;
numBlocks = 4;

layer = sequenceInputLayer(numFeatures,Normalization="rescale-symmetric",Name="input");
lgraph = layerGraph(layer);

outputName = layer.Name;
```

```
for i = 1:numBlocks
    dilationFactor = 2^(i-1);

    layers = [
        convolution1dLayer(filterSize,numFilters,DilationFactor=dilationFactor,Padding="causal",N
        layerNormalizationLayer
        spatialDropoutLayer(dropoutFactor)
        convolution1dLayer(filterSize,numFilters,DilationFactor=dilationFactor,Padding="causal")
        layerNormalizationLayer
        reluLayer
        spatialDropoutLayer(dropoutFactor)
        additionLayer(2,Name="add_"+i)];

    % Add and connect layers.
    lgraph = addLayers(lgraph,layers);
    lgraph = connectLayers(lgraph,outputName,"conv1_"+i);

    % Skip connection.
    if i == 1
        % Include convolution in first skip connection.
        layer = convolution1dLayer(1,numFilters,Name="convSkip");

        lgraph = addLayers(lgraph,layer);
        lgraph = connectLayers(lgraph,outputName,"convSkip");
        lgraph = connectLayers(lgraph,"convSkip","add_" + i + "/in2");
    else
        lgraph = connectLayers(lgraph,outputName,"add_" + i + "/in2");
    end

    % Update layer output name.
    outputName = "add_" + i;
end

layers = [
    fullyConnectedLayer(numClasses,Name="fc")
    softmaxLayer
    classificationLayer];
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,outputName,"fc");
```

View the network in a plot.

```
figure
plot(lgraph)
title("Temporal Convolutional Network")
```



Specify Training Options

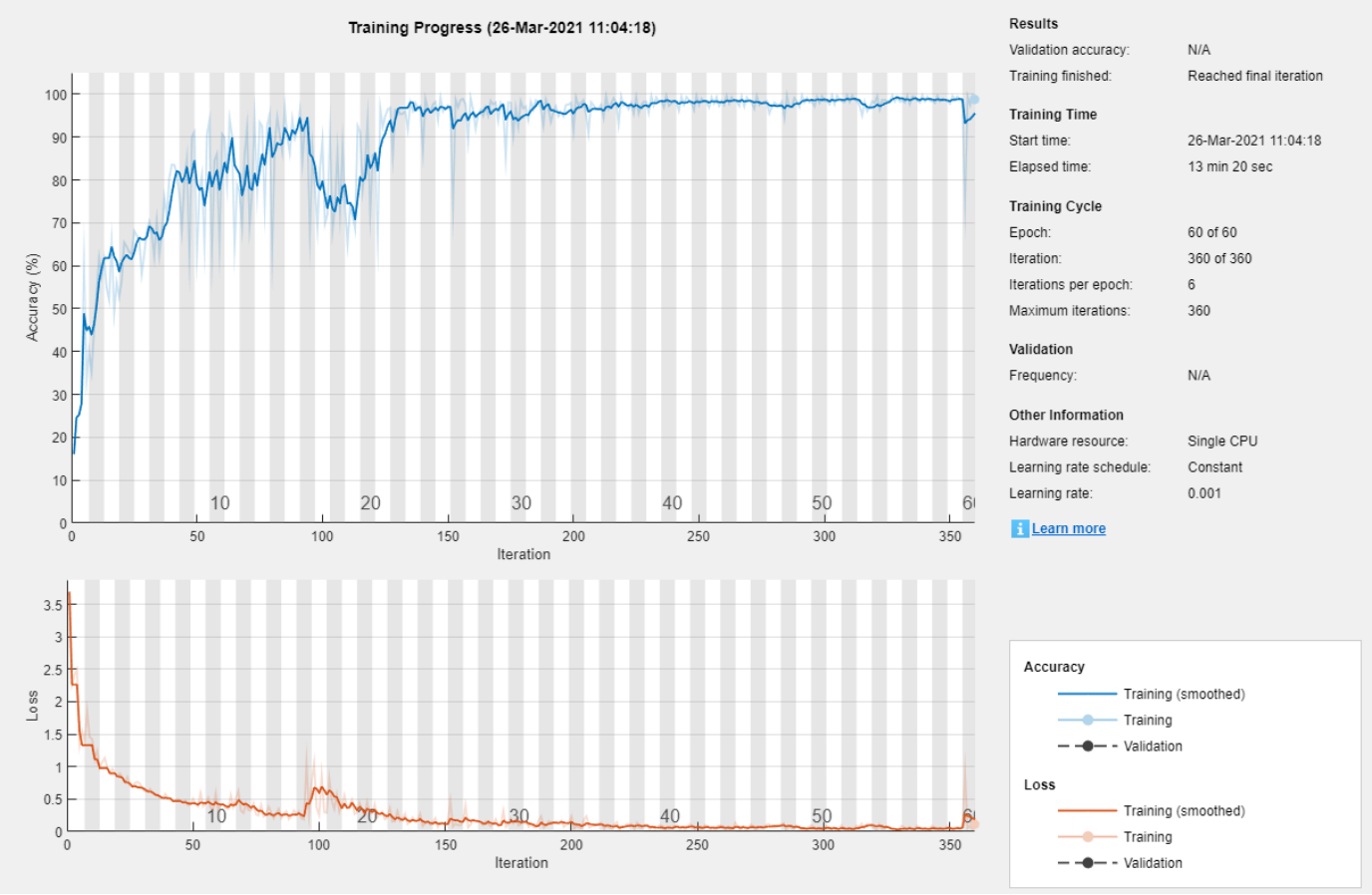
Specify a set of options used for training.

- Train for 60 epochs with a mini-batch size of 1.
- Train with a learning rate of 0.001.
- Display the training progress in a plot and suppress the verbose output.

```
options = trainingOptions("adam", ...
    MaxEpochs=60, ...
    miniBatchSize=1, ...
    Plots="training-progress", ...
    Verbose=0);
```

Train Model

```
net = trainNetwork(XTrain,TTrain,lgraph,options);
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a held-out test set with the true labels for each time step.

Load the test data.

```
s = load("HumanActivityTest.mat");
XTest = s.XTest;
TTest = s.YTest;
```

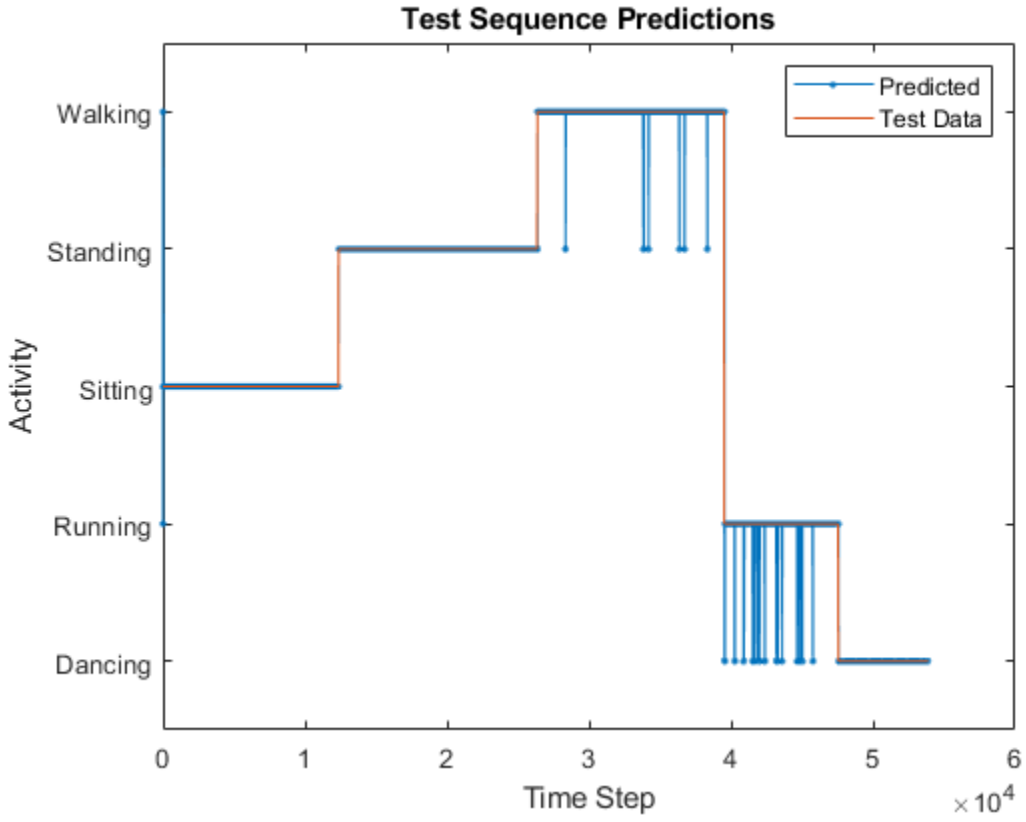
Use the trained network to make predictions by using the `classify` function.

```
YPred = classify(net,XTest);
```

Compare the predictions with the corresponding test data in a plot.

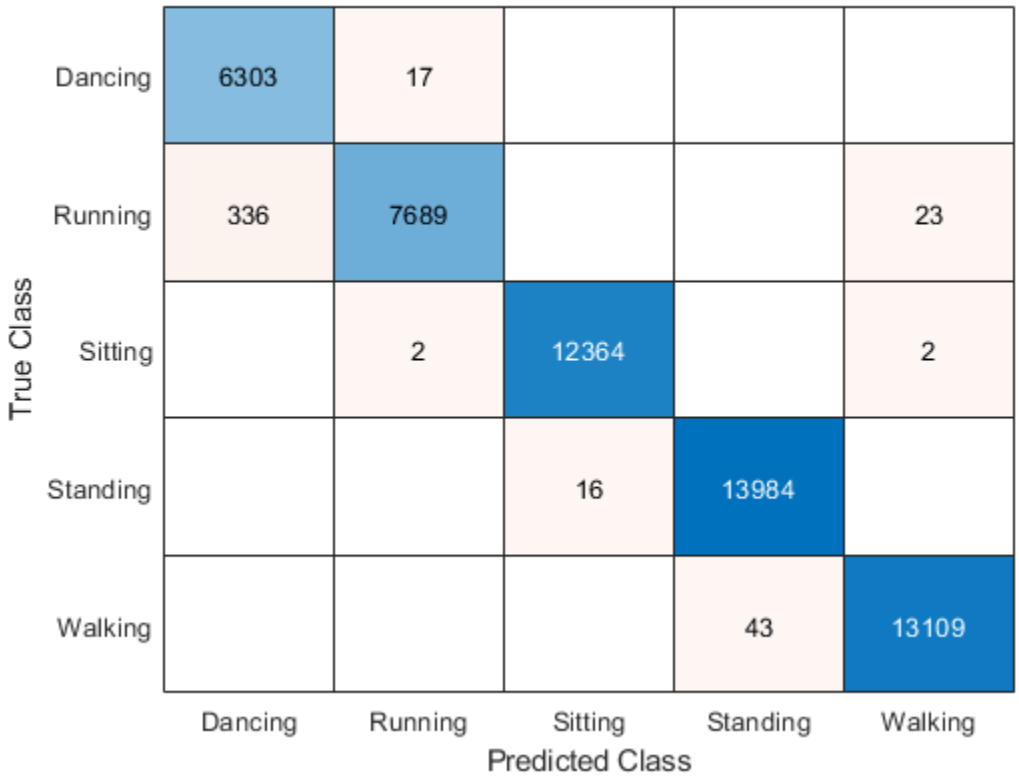
```
figure
plot(YPred{1},".-")
hold on
plot(TTest{1})
hold off

xlabel("Time Step")
ylabel("Activity")
```

```
legend(["Predicted" "Test Data"],Location="northeast")
title("Test Sequence Predictions")
```



Visualize the predictions in a confusion matrix.

```
figure
confusionchart(TTest{1},YPred{1})
```



Evaluate the classification accuracy by comparing the predictions to the test labels.

```
accuracy = mean(YPred{1} == TTest{1})
```

```
accuracy = 0.9919
```

References

[1] Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling.” Preprint, submitted April 19, 2018. https://arxiv.org/abs/1803.01271.

[2] Oord, Aaron van den, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. “WaveNet: A Generative Model for Raw Audio.” Preprint, submitted September 12, 2016. https://arxiv.org/abs/1609.03499.

[3] Tompson, Jonathan, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. “Efficient Object Localization Using Convolutional Networks.” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 648–56. https://doi.org/10.1109/CVPR.2015.7298664.

See Also

`convolution1dLayer` | `trainingOptions` | `trainNetwork` | `sequenceInputLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Related Examples

- “Sequence Classification Using 1-D Convolutions” on page 4-9
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

See Also

`dlnetwork` | `dlarray` | `adamupdate` | `dlfeval` | `dlgradient` | `crossentropy` | `minibatchqueue` | `convolution1dLayer` | `sequenceInputLayer` | `maxPooling1dLayer` | `averagePooling1dLayer` | `globalMaxPooling1dLayer` | `globalAveragePooling1dLayer`

Related Examples

- “Sequence Classification Using 1-D Convolutions” on page 4-9
- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Classify Text Data Using Deep Learning

This example shows how to classify text data using a deep learning long short-term memory (LSTM) network.

Text data is naturally sequential. A piece of text is a sequence of words, which might have dependencies between them. To learn and use long-term dependencies to classify sequence data, use an LSTM neural network. An LSTM network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data.

To input text to an LSTM network, first convert the text data into numeric sequences. You can achieve this using a word encoding which maps documents to sequences of numeric indices. For better results, also include a word embedding layer in the network. Word embeddings map words in a vocabulary to numeric vectors rather than scalar indices. These embeddings capture semantic details of the words, so that words with similar meanings have similar vectors. They also model relationships between words through vector arithmetic. For example, the relationship "*Rome is to Italy as* Paris *is to France*" is described by the equation Italy – *Rome* + *Paris* = *France*.

There are four steps in training and using the LSTM network in this example:

- Import and preprocess the data.
- Convert the words to numeric sequences using a word encoding.
- Create and train an LSTM network with a word embedding layer.
- Classify new text data using the trained LSTM network.

Import Data

Import the factory reports data. This data contains labeled textual descriptions of factory events. To import the text data as strings, specify the text type to be `'string'`.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
head(data)
```

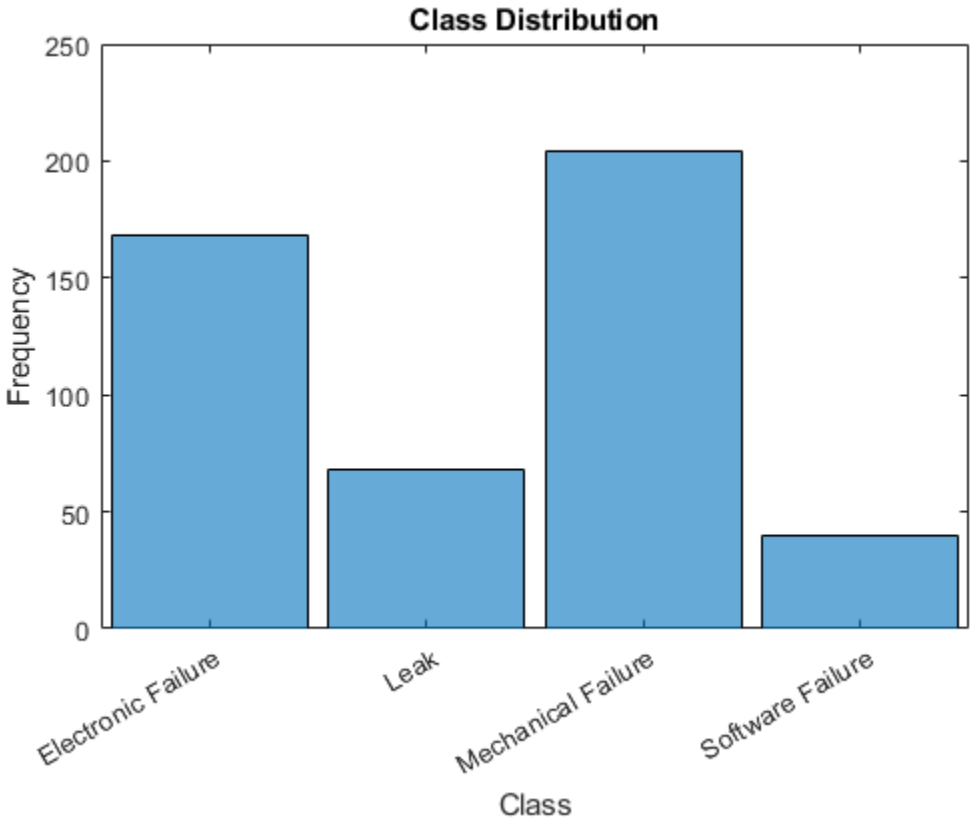
```
ans=8×5 table
                                Description                                 Category
    ____________________________________________________________________    ____________

    "Items are occasionally getting stuck in the scanner spools."           "Mechanical Failure"
    "Loud rattling and banging sounds are coming from assembler pistons."   "Mechanical Failure"
    "There are cuts to the power when starting the plant."                  "Electronic Failure"
    "Fried capacitors in the assembler."                                    "Electronic Failure"
    "Mixer tripped the fuses."                                              "Electronic Failure"
    "Burst pipe in the constructing agent is spraying coolant."             "Leak"
    "A fuse is blown in the mixer."                                         "Electronic Failure"
    "Things continue to tumble off of the belt."                            "Mechanical Failure"
```

The goal of this example is to classify events by the label in the `Category` column. To divide the data into classes, convert these labels to categorical.

```
data.Category = categorical(data.Category);
```

View the distribution of the classes in the data using a histogram.

```
figure
histogram(data.Category);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



The next step is to partition it into sets for training and validation. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 20%.

```
cvp = cvpartition(data.Category,'Holdout',0.2);
dataTrain = data(training(cvp),:);
dataValidation = data(test(cvp),:);
```

Extract the text data and labels from the partitioned tables.

```
textDataTrain = dataTrain.Description;
textDataValidation = dataValidation.Description;
YTrain = dataTrain.Category;
YValidation = dataValidation.Category;
```

To check that you have imported the data correctly, visualize the training text data using a word cloud.

```
figure
wordcloud(textDataTrain);
title("Training Data")
```

Preprocess Text Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

Preprocess the training data and the validation data using the `preprocessText` function.

```
documentsTrain = preprocessText(textDataTrain);
documentsValidation = preprocessText(textDataValidation);
```

View the first few preprocessed training documents.

```
documentsTrain(1:5)
```

```
ans =
  5×1 tokenizedDocument:

     9 tokens: items are occasionally getting stuck in the scanner spools
    10 tokens: loud rattling and banging sounds are coming from assembler pistons
    10 tokens: there are cuts to the power when starting the plant
     5 tokens: fried capacitors in the assembler
     4 tokens: mixer tripped the fuses
```

Convert Document to Sequences

To input the documents into an LSTM network, use a word encoding to convert the documents into sequences of numeric indices.

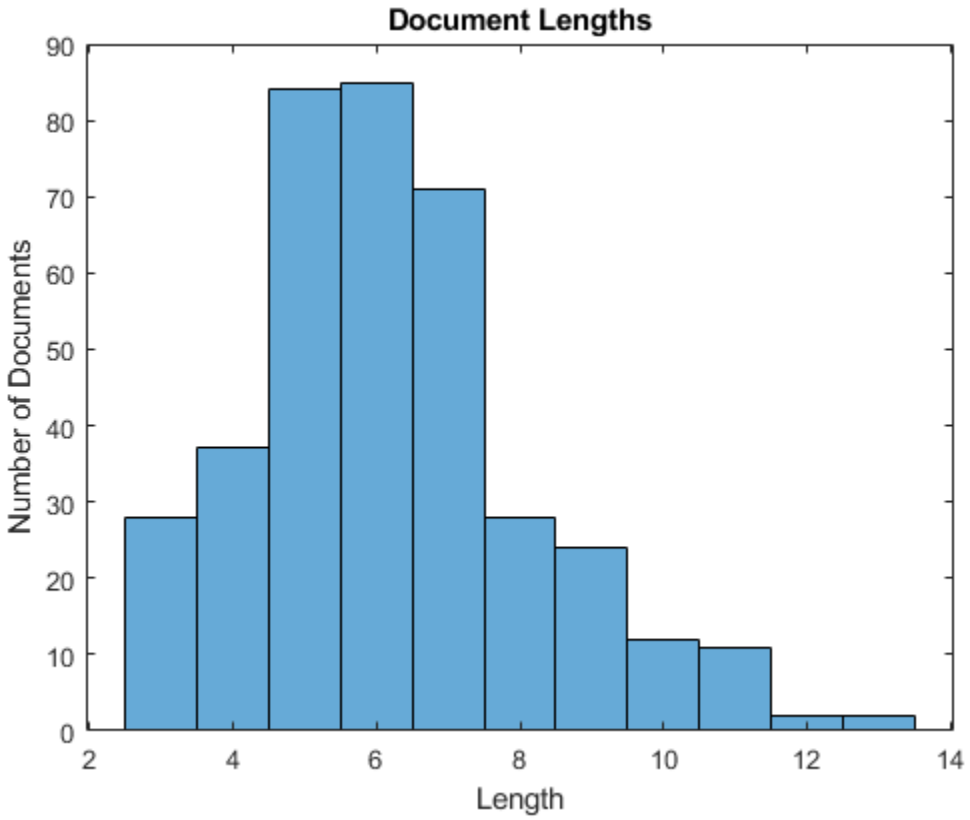
To create a word encoding, use the `wordEncoding` function.

```
enc = wordEncoding(documentsTrain);
```

The next conversion step is to pad and truncate documents so they are all the same length. The `trainingOptions` function provides options to pad and truncate input sequences automatically. However, these options are not well suited for sequences of word vectors. Instead, pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

To pad and truncate the documents, first choose a target length, and then truncate documents that are longer than it and left-pad documents that are shorter than it. For best results, the target length should be short without discarding large amounts of data. To find a suitable target length, view a histogram of the training document lengths.

```
documentLengths = doclength(documentsTrain);
figure
histogram(documentLengths)
title("Document Lengths")
xlabel("Length")
ylabel("Number of Documents")
```



Most of the training documents have fewer than 10 tokens. Use this as your target length for truncation and padding.

Convert the documents to sequences of numeric indices using `doc2sequence`. To truncate or left-pad the sequences to have length 10, set the `'Length'` option to 10.

```
sequenceLength = 10;
XTrain = doc2sequence(enc,documentsTrain,'Length',sequenceLength);
XTrain(1:5)
```

```
ans=5×1 cell array
    {1×10 double}
    {1×10 double}
    {1×10 double}
    {1×10 double}
    {1×10 double}
```

Convert the validation documents to sequences using the same options.

```
XValidation = doc2sequence(enc,documentsValidation,'Length',sequenceLength);
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 50 and the same number of words as the word encoding. Next, include an LSTM layer and set the number of hidden units to 80. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with the same size as the number of classes, a softmax layer, and a classification layer.

```
inputSize = 1;
embeddingDimension = 50;
numHiddenUnits = 80;

numWords = enc.NumWords;
numClasses = numel(categories(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension,numWords)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  6x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 1 dimensions
     2   ''   Word Embedding Layer    Word embedding layer with 50 dimensions and 423 unique word
     3   ''   LSTM                   LSTM with 80 hidden units
     4   ''   Fully Connected         4 fully connected layer
     5   ''   Softmax                softmax
     6   ''   Classification Output   crossentropyex
```

Specify Training Options

Specify the training options:

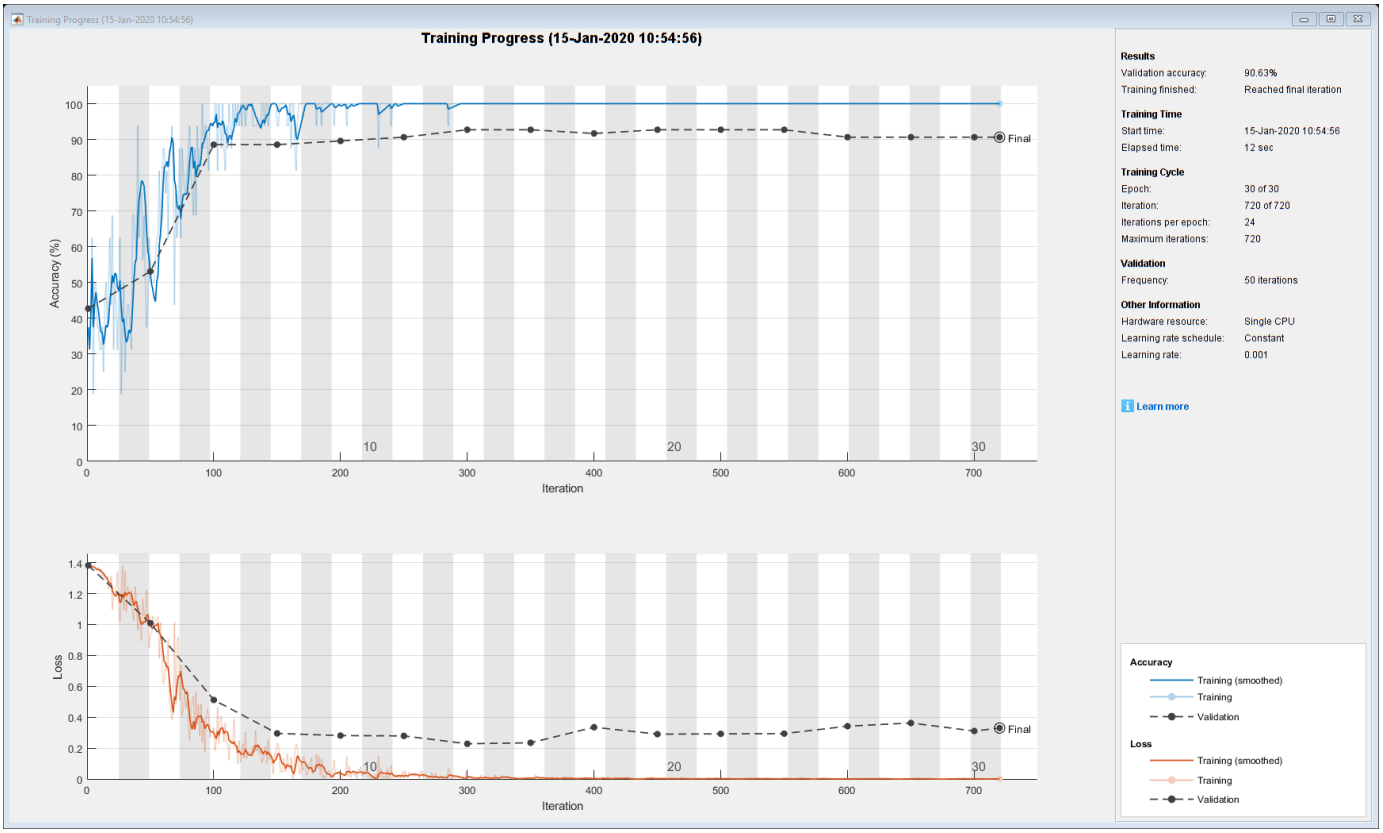
- Train using the Adam solver.
- Specify a mini-batch size of 16.
- Shuffle the data every epoch.
- Monitor the training progress by setting the `'Plots'` option to `'training-progress'`.
- Specify the validation data using the `'ValidationData'` option.
- Suppress verbose output by setting the `'Verbose'` option to `false`.

By default, `trainNetwork` uses a GPU if one is available. Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU. Training with a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
options = trainingOptions('adam', ...
    'MiniBatchSize',16, ...
    'GradientThreshold',2, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{XValidation,YValidation}, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
```

Convert the text data to sequences using `doc2sequence` with the same options as when creating the training sequences.

```
XNew = doc2sequence(enc,documentsNew,'Length',sequenceLength);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Convert to lowercase.
documents = lower(documents);

% Erase punctuation.
documents = erasePunctuation(documents);

end
```

See Also

`fastTextWordEmbedding` | `wordEmbeddingLayer` | `tokenizedDocument` | `lstmLayer` | `trainNetwork` | `trainingOptions` | `doc2sequence` | `sequenceInputLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” on page 4-180

- “Word-By-Word Text Generation Using Deep Learning” (Text Analytics Toolbox)
- “Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore” (Text Analytics Toolbox)
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Classify Text Data Using Convolutional Neural Network

This example shows how to classify text data using a convolutional neural network.

To classify text data using convolutions, you must convert the text data into images. To do this, pad or truncate the observations to have constant length S and convert the documents into sequences of word vectors of length C using a word embedding. You can then represent a document as a 1-by-S-by-C image (an image with height 1, width S, and C channels).

To convert text data from a CSV file to images, create a `tabularTextDatastore` object. The convert the data read from the `tabularTextDatastore` object to images for deep learning by calling `transform` with a custom transformation function. The `transformTextData` function, listed at the end of the example, takes data read from the datastore and a pretrained word embedding, and converts each observation to an array of word vectors.

This example trains a network with 1-D convolutional filters of varying widths. The width of each filter corresponds the number of words the filter can see (the n-gram length). The network has multiple branches of convolutional layers, so it can use different n-gram lengths.

Load Pretrained Word Embedding

Load the pretrained fastText word embedding. This function requires the Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Load Data

Create a tabular text datastore from the data in `factoryReports.csv`. Read the data from the `"Description"` and `"Category"` columns only.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelName]);
```

Preview the datastore.

```
ttdsTrain.ReadSize = 8;
preview(ttdsTrain)
```

```
ans=8×2 table
                                Description                                                Category
    ____________________________________________________________________________    ____________________

    {'Items are occasionally getting stuck in the scanner spools.'        }    {'Mechanical Failu
    {'Loud rattling and banging sounds are coming from assembler pistons.'}    {'Mechanical Failu
    {'There are cuts to the power when starting the plant.'               }    {'Electronic Failu
    {'Fried capacitors in the assembler.'                                 }    {'Electronic Failu
    {'Mixer tripped the fuses.'                                           }    {'Electronic Failu
    {'Burst pipe in the constructing agent is spraying coolant.'          }    {'Leak'
    {'A fuse is blown in the mixer.'                                      }    {'Electronic Failu
    {'Things continue to tumble off of the belt.'                         }    {'Mechanical Failu
```

Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The `transformTextData` function, listed at the end of the example, takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are 1-by-`sequenceLength`-by-*C* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension. The responses are categorical labels over the classes in `classNames`.

Read the labels from the training data using the `readLabels` function, listed at the end of the example, and find the unique class names.

```
labels = readLabels(ttdsTrain,labelName);
classNames = unique(labels);
numObservations = numel(labels);
```

Transform the datastore using `transformTextData` function and specify a sequence length of 14.

```
sequenceLength = 14;
tdsTrain = transform(ttdsTrain, @(data) transformTextData(data,sequenceLength,emb,classNames))
```

```
tdsTrain =
  TransformedDatastore with properties:

       UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@(data)transformTextData(data,sequenceLength,emb,classNames)}
              IncludeInfo: 0
```

Preview the transformed datastore. The predictors are 1-by-*S*-by-*C* arrays, where *S* is the sequence length and *C* is the number of features (the embedding dimension). The responses are the categorical labels.

```
preview(tdsTrain)
```

```
ans=8×2 table
       Predictors            Responses
    _________________    _________________

    {1×14×300 single}    Mechanical Failure
    {1×14×300 single}    Mechanical Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Leak
    {1×14×300 single}    Electronic Failure
    {1×14×300 single}    Mechanical Failure
```

Define Network Architecture

Define the network architecture for the classification task.

The following steps describe the network architecture.

- Specify an input size of 1-by-*S*-by-*C*, where *S* is the sequence length and *C* is the number of features (the embedding dimension).
- For the n-gram lengths 2, 3, 4, and 5, create blocks of layers containing a convolutional layer, a batch normalization layer, a ReLU layer, a dropout layer, and a max pooling layer.

- For each block, specify 200 convolutional filters of size 1-by-*N* and pooling regions of size 1-by-*S*, where *N* is the n-gram length.
- Connect the input layer to each block and concatenate the outputs of the blocks using a depth concatenation layer.
- To classify the outputs, include a fully connected layer with output size *K*, a softmax layer, and a classification layer, where *K* is the number of classes.

First, in a layer array, specify the input layer, the first block for unigrams, the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
numFeatures = emb.Dimension;
inputSize = [1 sequenceLength numFeatures];
numFilters = 200;

ngramLengths = [2 3 4 5];
numBlocks = numel(ngramLengths);

numClasses = numel(classNames);
```

Create a layer graph containing the input layer. Set the normalization option to `'none'` and the layer name to `'input'`.

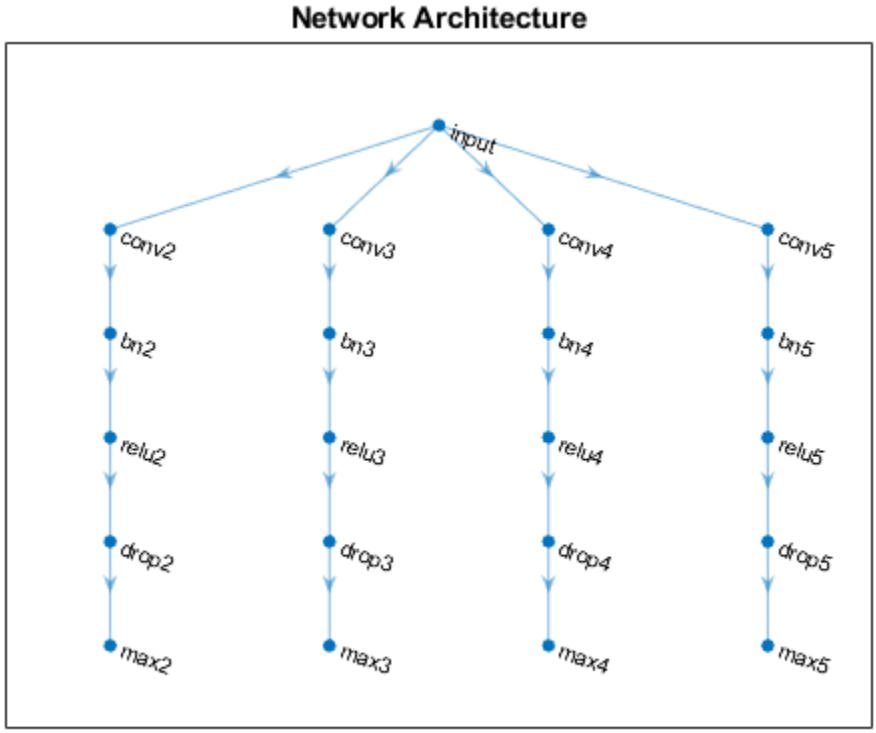
```
layer = imageInputLayer(inputSize,'Normalization','none','Name','input');
lgraph = layerGraph(layer);
```

For each of the n-gram lengths, create a block of convolution, batch normalization, ReLU, dropout, and max pooling layers. Connect each block to the input layer.

```
for j = 1:numBlocks
    N = ngramLengths(j);

    block = [
        convolution2dLayer([1 N],numFilters,'Name',"conv"+N,'Padding','same')
        batchNormalizationLayer('Name',"bn"+N)
        reluLayer('Name',"relu"+N)
        dropoutLayer(0.2,'Name',"drop"+N)
        maxPooling2dLayer([1 sequenceLength],'Name',"max"+N)];

    lgraph = addLayers(lgraph,block);
    lgraph = connectLayers(lgraph,'input',"conv"+N);
end
```

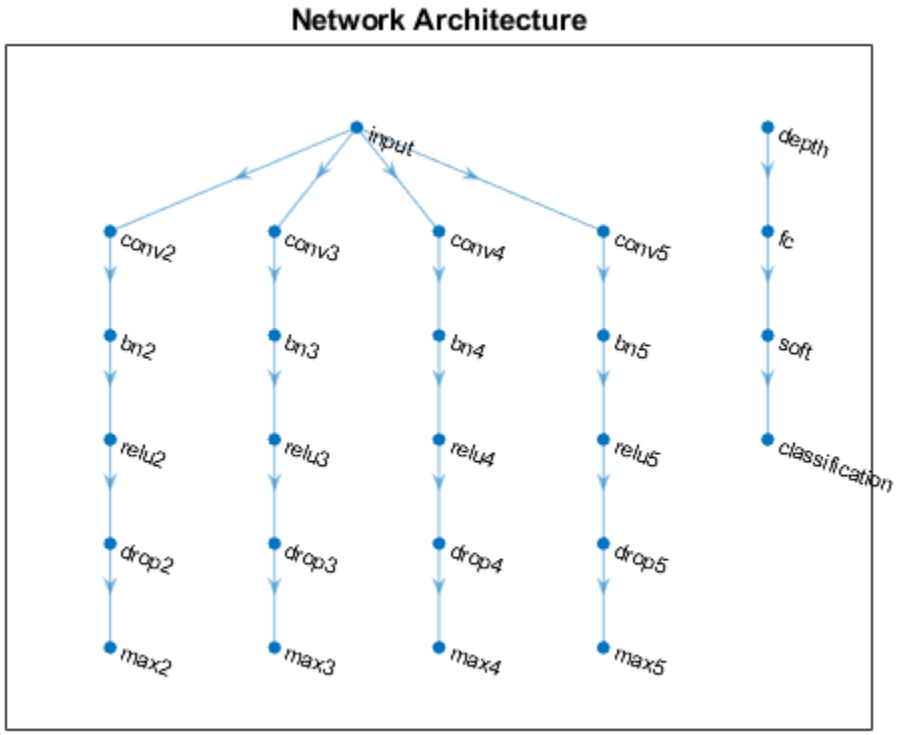
View the network architecture in a plot.

```
figure
plot(lgraph)
title("Network Architecture")
```



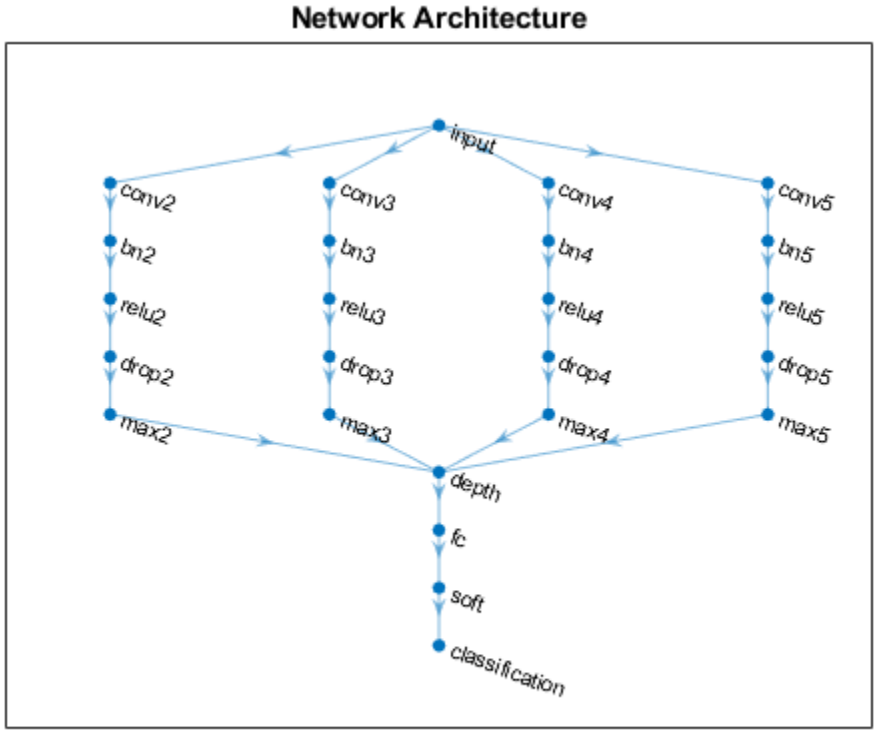
Add the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
layers = [
    depthConcatenationLayer(numBlocks,'Name','depth')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','soft')
    classificationLayer('Name','classification')];

lgraph = addLayers(lgraph,layers);

figure
plot(lgraph)
title("Network Architecture")
```



Connect the max pooling layers to the depth concatenation layer and view the final network architecture in a plot.

```
for j = 1:numBlocks
    N = ngramLengths(j);
    lgraph = connectLayers(lgraph,"max"+N,"depth/in"+j);
end

figure
plot(lgraph)
title("Network Architecture")
```



Train Network

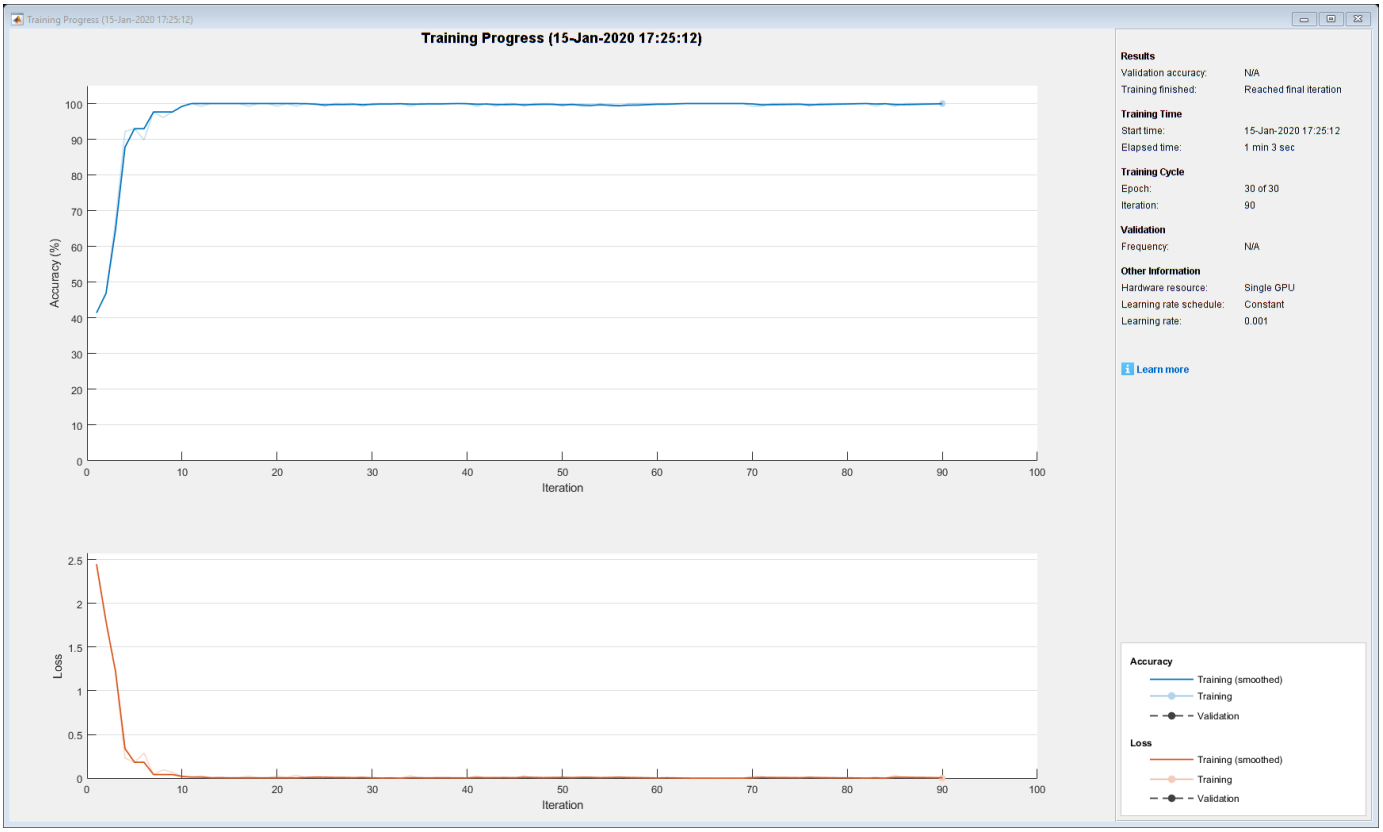
Specify the training options:

- Train with a mini-batch size of 128.
- Do not shuffle the data because the datastore is not shuffleable.
- Display the training progress plot and suppress the verbose output.

```
miniBatchSize = 128;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,lgraph,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
XNew = preprocessText(reportsNew,sequenceLength,emb);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Read Labels Function

The `readLabels` function creates a copy of the `tabularTextDatastore` object `ttds` and reads the labels from the `labelName` column.

```
function labels = readLabels(ttds,labelName)
```

```
ttdsNew = copy(ttds);
ttdsNew.SelectedVariableNames = labelName;
tbl = readall(ttdsNew);
labels = tbl.(labelName);

end
```

Transform Text Data Function

The `transformTextData` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are 1-by-`sequenceLength`-by-*C* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension. The responses are categorical labels over the classes in `classNames`.

```
function dataTransformed = transformTextData(data,sequenceLength,emb,classNames)

% Preprocess documents.
textData = data{:,1};

% Prepocess text
dataTransformed = preprocessText(textData,sequenceLength,emb);

% Read labels.
labels = data{:,2};
responses = categorical(labels,classNames);

% Convert data to table.
dataTransformed.Responses = responses;

end
```

Preprocess Text Function

The `preprocessTextData` function takes text data, a sequence length, and a word embedding and performs these steps:

1. Tokenize the text.
2. Convert the text to lowercase.
3. Converts the documents to sequences of word vectors of the specified length using the embedding.
4. Reshapes the word vector sequences to input into the network.

```
function tbl = preprocessText(textData,sequenceLength,emb)

documents = tokenizedDocument(textData);
documents = lower(documents);

% Convert documents to embeddingDimension-by-sequenceLength-by-1 images.
predictors = doc2sequence(emb,documents,'Length',sequenceLength);

% Reshape images to be of size 1-by-sequenceLength-embeddingDimension.
predictors = cellfun(@(X) permute(X,[3 2 1]),predictors,'UniformOutput',false);

tbl = table;
tbl.Predictors = predictors;
```

```
end
```

See Also

`fastTextWordEmbedding` | `wordcloud` | `wordEmbedding` | `layerGraph` | `convolution2dLayer` | `batchNormalizationLayer` | `trainingOptions` | `trainNetwork` | `doc2sequence` | `tokenizedDocument` | `transform`

Related Examples

- “Classify Text Data Using Deep Learning” (Text Analytics Toolbox)
- “Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore” (Text Analytics Toolbox)
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Datastores for Deep Learning” on page 19-2
- “Deep Learning in MATLAB” on page 1-2

Multilabel Text Classification Using Deep Learning

This example shows how to classify text data that has multiple independent labels.

For classification tasks where there can be multiple independent labels for each observation—for example, tags on an scientific article—you can train a deep learning model to predict probabilities for each independent class. To enable a network to learn multilabel classification targets, you can optimize the loss of each class independently using binary cross-entropy loss.

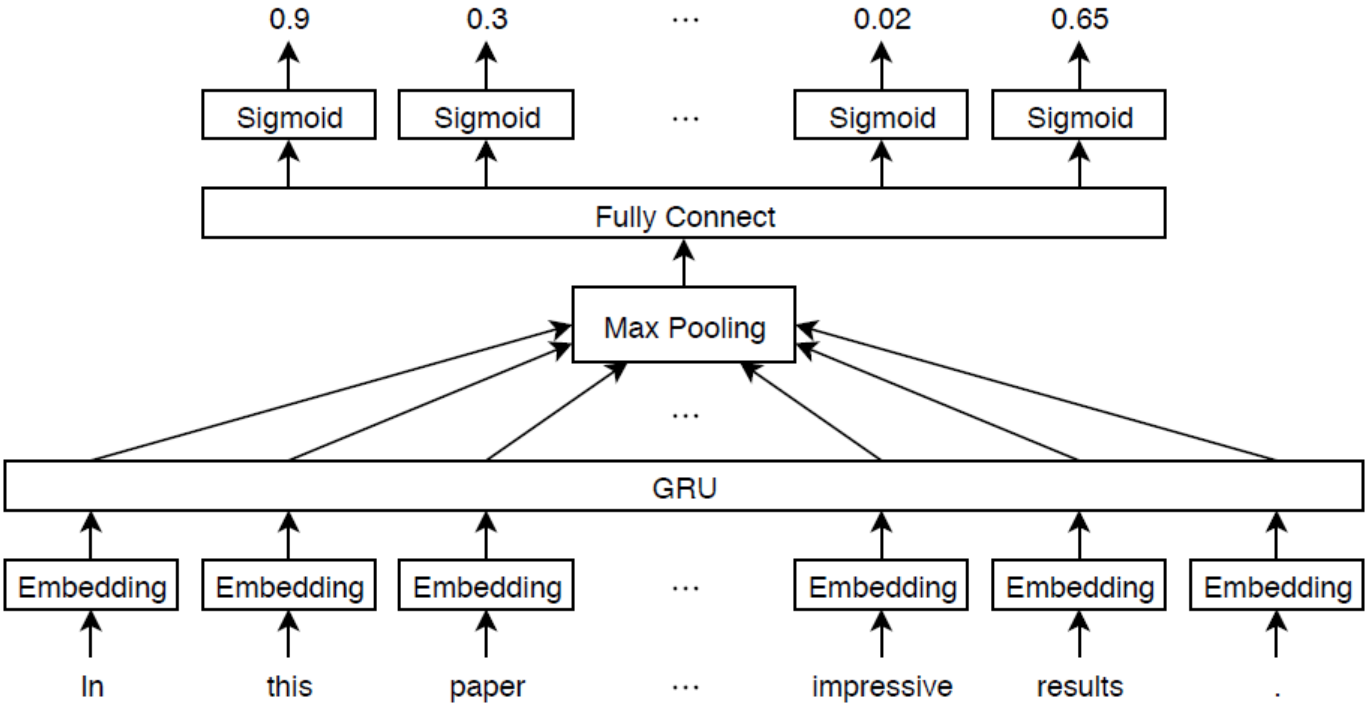
This example defines a deep learning model that classifies subject areas given the abstracts of mathematical papers collected using the arXiv API [1]. The model consists of a word embedding and GRU, max pooling operation, fully connected, and sigmoid operations.

To measure the performance of multilabel classification, you can use the labeling F-score [2]. The labeling F-score evaluates multilabel classification by focusing on per-text classification with partial matches. The measure is the normalized proportion of matching labels against the total number of true and predicted labels.

This example defines the following model:

- A word embedding that maps a sequence of words to a sequence of numeric vectors.
- A GRU operation that learns dependencies between the embedding vectors.
- A max pooling operation that reduces a sequence of feature vectors to a single feature vector.
- A fully connected layer that maps the features to the binary outputs.
- A sigmoid operation for learning the binary cross entropy loss between the outputs and the target labels.

This diagram shows a piece of text propagating through the model architecture and outputting a vector of probabilities. The probabilities are independent, so they need not sum to one.



Import Text Data

Import a set of abstracts and category labels from math papers using the arXiv API. Specify the number of records to import using the `importSize` variable.

```
importSize = 50000;
```

Create a URL that queries records with set `"math"` and metadata prefix `"arXiv"`.

```
url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
    "&set=math" + ...
    "&metadataPrefix=arXiv";
```

Extract the abstract text, category labels, and the resumption token returned by the query URL using the `parseArXivRecords` function which is attached to this example as a supporting file. To access this file, open this example as a live script. Note that the arXiv API is rate limited and requires waiting between multiple requests.

```
[textData,labelsAll,resumptionToken] = parseArXivRecords(url);
```

Iteratively import more chunks of records until the required amount is reached, or there are no more records. To continue importing records from where you left off, use the resumption token from the previous result in the query URL. To adhere to the rate limits imposed by the arXiv API, add a delay of 20 seconds before each query using the `pause` function.

```
while numel(textData) < importSize

    if resumptionToken == ""
        break
    end
```

```
    url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
        "&resumptionToken=" + resumptionToken;

    pause(20)
    [textDataNew,labelsNew,resumptionToken] = parseArXivRecords(url);

    textData = [textData; textDataNew];
    labelsAll = [labelsAll; labelsNew];
end
```

Preprocess Text Data

Tokenize and preprocess the text data using the `preprocessText` function, listed at the end of the example.

```
documentsAll = preprocessText(textData);
documentsAll(1:5)
```

```
ans =
  5×1 tokenizedDocument:

    72 tokens: describe new algorithm $(k,\ell)$ pebble game color obtain characterization family
    22 tokens: show determinant stirling cycle number count unlabeled acyclic singlesource automa
    18 tokens: paper show compute $\lambda_{\alpha}$ norm alpha dyadic grid result consequence de
    62 tokens: partial cube isometric subgraphs hypercubes structure graph define mean semicubes
    29 tokens: paper present algorithm compute hecke eigensystems hilbertsiegel cusp form real qu
```

Remove labels that do not belong to the `"math"` set.

```
for i = 1:numel(labelsAll)
    labelsAll{i} = labelsAll{i}(startsWith(labelsAll{i},"math."));
end
```

Visualize some of the classes in a word cloud. Find the documents corresponding to the following:

- Abstracts tagged with "Combinatorics" and not tagged with `"Statistics Theory"`
- Abstracts tagged with "Statistics Theory" and not tagged with `"Combinatorics"`
- Abstracts tagged with both `"Combinatorics"` and `"Statistics Theory"`

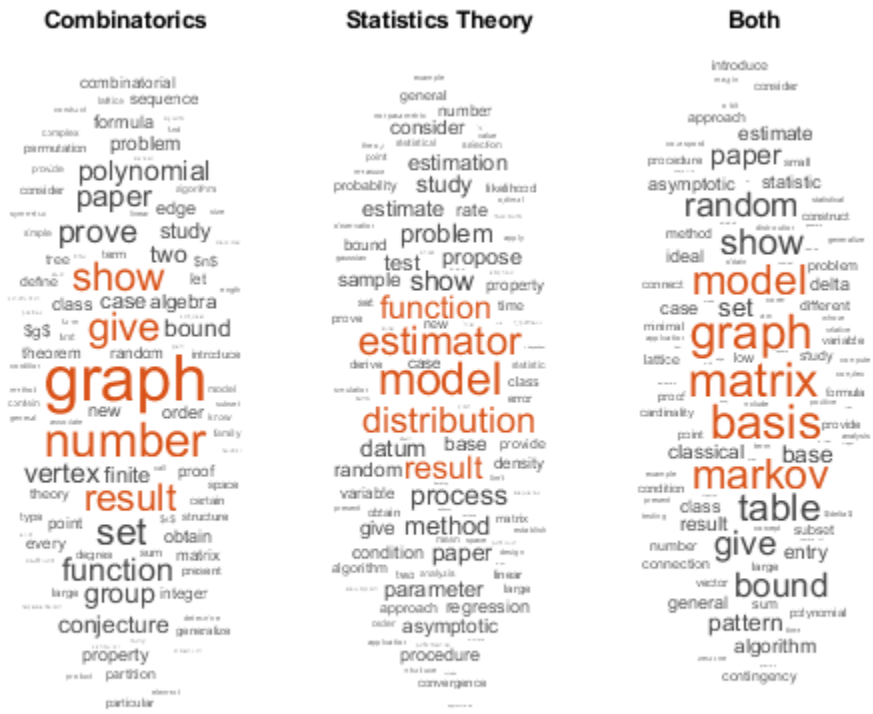
Find the document indices for each of the groups using the `ismember` function.

```
idxCO = cellfun(@(lbls) ismember("math.CO",lbls) && ~ismember("math.ST",lbls),labelsAll);
idxST = cellfun(@(lbls) ismember("math.ST",lbls) && ~ismember("math.CO",lbls),labelsAll);
idxCOST = cellfun(@(lbls) ismember("math.CO",lbls) && ismember("math.ST",lbls),labelsAll);
```

Visualize the documents for each group in a word cloud.

```
figure
subplot(1,3,1)
wordcloud(documentsAll(idxCO));
title("Combinatorics")

subplot(1,3,2)
wordcloud(documentsAll(idxST));
title("Statistics Theory")

subplot(1,3,3)
```

```
wordcloud(documentsAll(idxCOST));
title("Both")
```



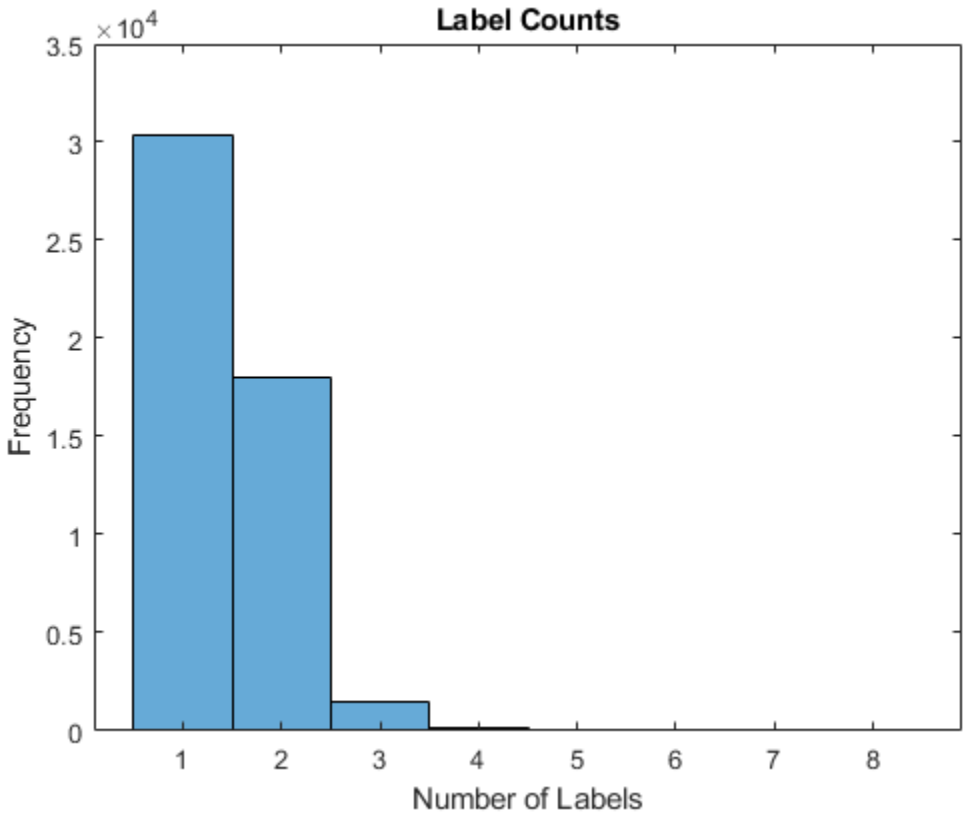
View the number of classes.

```
classNames = unique(cat(1,labelsAll{:}));
numClasses = numel(classNames)
```

```
numClasses = 32
```

Visualize the number of per-document labels using a histogram.

```
labelCounts = cellfun(@numel, labelsAll);
figure
histogram(labelCounts)
xlabel("Number of Labels")
ylabel("Frequency")
title("Label Counts")
```



Prepare Text Data for Deep Learning

Partition the data into training and validation partitions using the `cvpartition` function. Hold out 10% of the data for validation by setting the `'HoldOut'` option to 0.1.

```
cvp = cvpartition(numel(documentsAll),'HoldOut',0.1);
documentsTrain = documentsAll(training(cvp));
documentsValidation = documentsAll(test(cvp));

labelsTrain = labelsAll(training(cvp));
labelsValidation = labelsAll(test(cvp));
```

Create a word encoding object that encodes the training documents as sequences of word indices. Specify a vocabulary of the 5000 words by setting the `'Order'` option to `'frequency'`, and the `'MaxNumWords'` option to 5000.

```
enc = wordEncoding(documentsTrain,'Order','frequency','MaxNumWords',5000)
```

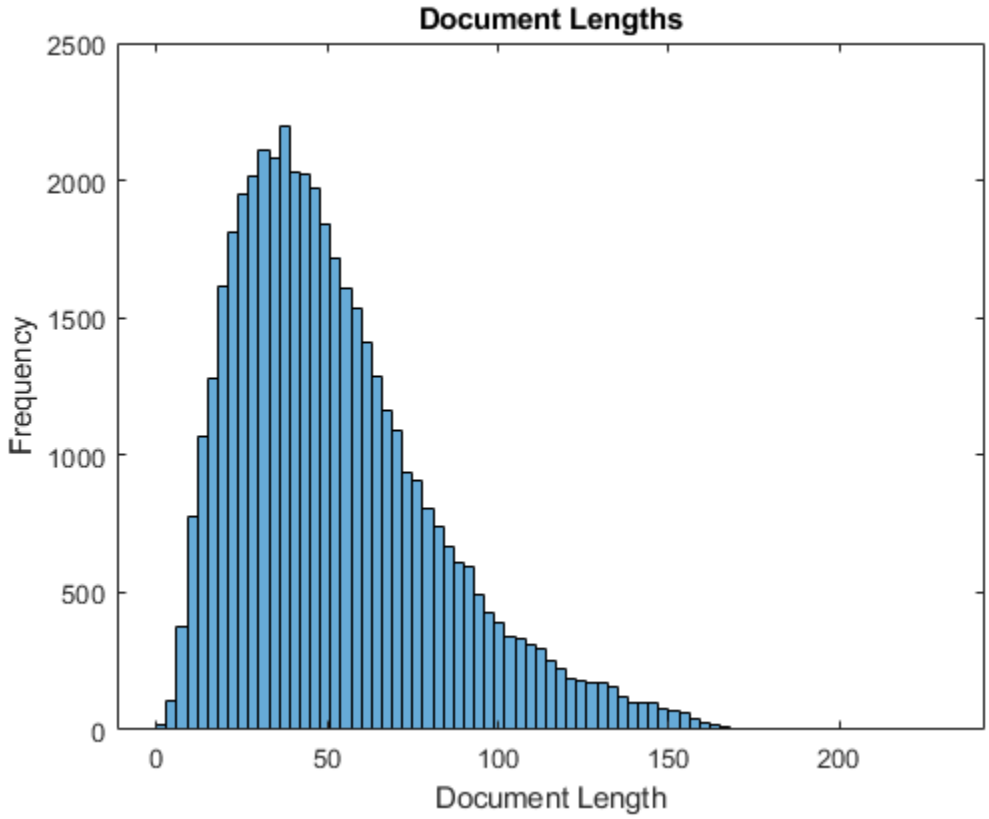
```
enc =
  wordEncoding with properties:

      NumWords: 5000
    Vocabulary: [1×5000 string]
```

To improve training, use the following techniques:

1. When training, truncate the documents to a length that reduces the amount of padding used and does not does discard too much data.
2. Train for one epoch with the documents sorted by length in ascending order, then shuffle the data each epoch. This technique is known as *sortagrad*.

To choose a sequence length for truncation, visualize the document lengths in a histogram and choose a value that captures most of the data.

```
documentLengths = doclength(documentsTrain);

figure
histogram(documentLengths)
xlabel("Document Length")
ylabel("Frequency")
title("Document Lengths")
```



Most of the training documents have fewer than 175 tokens. Use 175 tokens as the target length for truncation and padding.

```
maxSequenceLength = 175;
```

To use the sortagrad technique, sort the documents by length in ascending order.

```
[~,idx] = sort(documentLengths);
documentsTrain = documentsTrain(idx);
labelsTrain = labelsTrain(idx);
```

Define and Initialize Model Parameters

Define the parameters for each of the operations and include them in a struct. Use the format `parameters.OperationName.ParameterName`, where `parameters` is the struct, `OperationName` is the name of the operation (for example `"fc"`), and `ParameterName` is the name of the parameter (for example, `"Weights"`).

Create a struct `parameters` containing the model parameters. Initialize the bias with zeros. Use the following weight initializers for the operations:

- For the embedding, initialize the weights with random normal values.
- For the GRU operation, initialize the weights using the `initializeGlorot` function, listed at the end of the example.
- For the fully connect operation, initialize the weights using the `initializeGaussian` function, listed at the end of the example.

```
embeddingDimension = 300;
numHiddenUnits = 250;
inputSize = enc.NumWords + 1;

parameters = struct;
parameters.emb.Weights = dlarray(randn([embeddingDimension inputSize]));

parameters.gru.InputWeights = dlarray(initializeGlorot(3*numHiddenUnits,embeddingDimension));
parameters.gru.RecurrentWeights = dlarray(initializeGlorot(3*numHiddenUnits,numHiddenUnits));
parameters.gru.Bias = dlarray(zeros(3*numHiddenUnits,1,'single'));

parameters.fc.Weights = dlarray(initializeGaussian([numClasses,numHiddenUnits]));
parameters.fc.Bias = dlarray(zeros(numClasses,1,'single'));
```

View the `parameters` struct.

```
parameters
```

```
parameters = struct with fields:
    emb: [1×1 struct]
    gru: [1×1 struct]
     fc: [1×1 struct]
```

View the parameters for the GRU operation.

```
parameters.gru
```

```
ans = struct with fields:
        InputWeights: [750×300 dlarray]
    RecurrentWeights: [750×250 dlarray]
                Bias: [750×1 dlarray]
```

Define Model Function

Create the function `model`, listed at the end of the example, which computes the outputs of the deep learning model described earlier. The function `model` takes as input the input data `dlX` and the model parameters `parameters`. The network outputs the predictions for the labels.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, which takes as input a mini-batch of input data `dlX` and the corresponding targets `T` containing the labels, and returns the gradients of the loss with respect to the learnable parameters, the corresponding loss, and the network outputs.

Specify Training Options

Train for 5 epochs with a mini-batch size of 256.

```
numEpochs = 5;
miniBatchSize = 256;
```

Train using the Adam optimizer, with a learning rate of 0.01, and specify gradient decay and squared gradient decay factors of 0.5 and 0.999, respectively.

```
learnRate = 0.01;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Clip the gradients with a threshold of 1 using L_2 norm gradient clipping.

```
gradientThreshold = 1;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

To convert a vector of probabilities to labels, use the labels with probabilities higher than a specified threshold. Specify a label threshold of 0.5.

```
labelThreshold = 0.5;
```

Validate the network every epoch.

```
numObservationsTrain = numel(documentsTrain);
numIterationsPerEpoch = floor(numObservationsTrain/miniBatchSize);
validationFrequency = numIterationsPerEpoch;
```

Train on a GPU if one is available. This requires Parallel Computing Toolbox™. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Train Model

Train the model using a custom training loop.

For each epoch, loop over mini-batches of data. At the end of each epoch, shuffle the data. At the end of each iteration, update the training progress plot.

For each mini-batch:

- Convert the documents to sequences of word indices and convert the labels to dummy variables.
- Convert the sequences to `dlarray` objects with underlying type single and specify the dimension labels `'BTC'` (batch, time, channel).

- For GPU training, convert to `gpuArray` objects.
- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Clip the gradients.
- Update the network parameters using the `adamupdate` function.
- If necessary, validate the network using the `modelPredictions` function, listed at the end of the example.
- Update the training plot.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure

    % Labeling F-Score.
    subplot(2,1,1)
    lineFScoreTrain = animatedline('Color',[0 0.447 0.741]);
    lineFScoreValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 1])
    xlabel("Iteration")
    ylabel("Labeling F-Score")
    grid on

    % Loss.
    subplot(2,1,2)
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    lineLossValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Initialize parameters for the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Prepare the validation data. Create a one-hot encoded matrix where non-zero entries correspond to the labels of each observation.

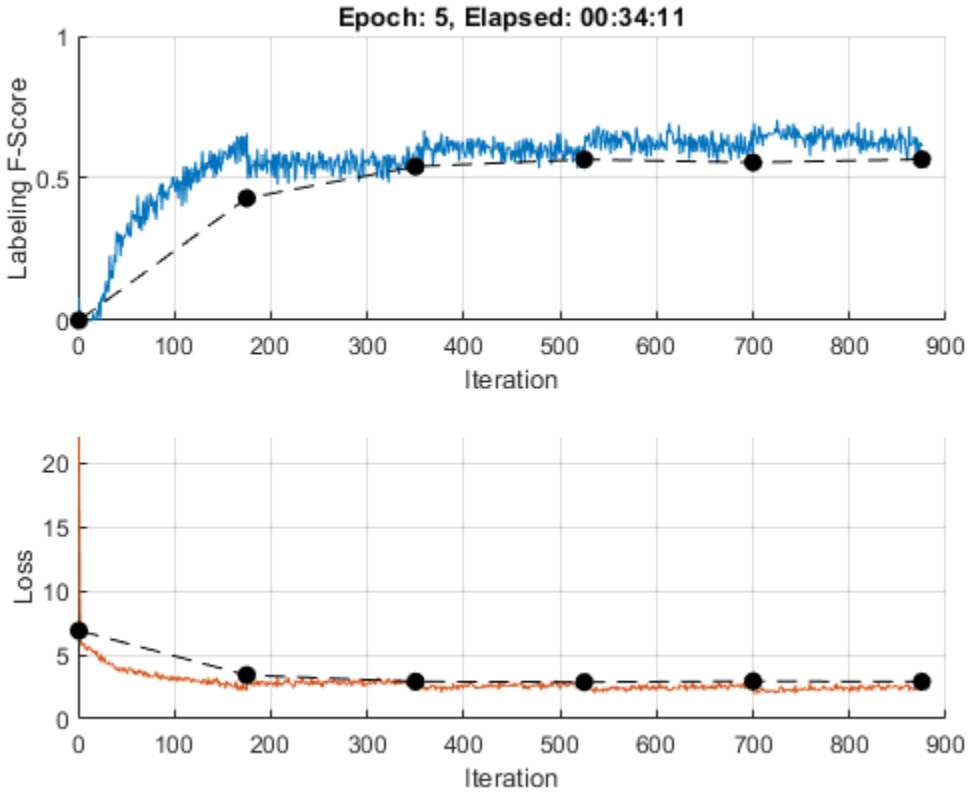
```
numObservationsValidation = numel(documentsValidation);
TValidation = zeros(numClasses, numObservationsValidation, 'single');
for i = 1:numObservationsValidation
    [~,idx] = ismember(labelsValidation{i},classNames);
    TValidation(idx,i) = 1;
end
```

Train the model.

```
iteration = 0;
start = tic;
```

```
% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;

        % Read mini-batch of data and convert the labels to dummy
        % variables.
        documents = documentsTrain(idx);
        labels = labelsTrain(idx);

        % Convert documents to sequences.
        len = min(maxSequenceLength,max(doclength(documents)));
        X = doc2sequence(enc,documents, ...
            'PaddingValue',inputSize, ...
            'Length',len);
        X = cat(1,X{:});

        % Dummify labels.
        T = zeros(numClasses, miniBatchSize, 'single');
        for j = 1:miniBatchSize
            [~,idx2] = ismember(labels{j},classNames);
            T(idx2,j) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(X,'BTC');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,dlYPred] = dlfeval(@modelGradients, dlX, T, parameters);

        % Gradient clipping.
        gradients = dlupdate(@(g) thresholdL2Norm(g, gradientThreshold),gradients);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration,learnRate,gradientDecayFactor,squaredGradientDeca

        % Display the training progress.
        if plots == "training-progress"
            subplot(2,1,1)
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            title("Epoch: " + epoch + ", Elapsed: " + string(D))

            % Loss.
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))

            % Labeling F-score.
            YPred = extractdata(dlYPred) > labelThreshold;
```

```
            score = labelingFScore(YPred,T);
            addpoints(lineFScoreTrain,iteration,double(gather(score)))

            drawnow

            % Display validation metrics.
            if iteration == 1 || mod(iteration,validationFrequency) == 0
                dlYPredValidation = modelPredictions(parameters,enc,documentsValidation,miniBatch

                % Loss.
                lossValidation = crossentropy(dlYPredValidation,TValidation, ...
                    'TargetCategories','independent', ...
                    'DataFormat','CB');
                addpoints(lineLossValidation,iteration,double(gather(extractdata(lossValidation)

                % Labeling F-score.
                YPredValidation = extractdata(dlYPredValidation) > labelThreshold;
                score = labelingFScore(YPredValidation,TValidation);
                addpoints(lineFScoreValidation,iteration,double(gather(score)))

                drawnow
            end
        end
    end

    % Shuffle data.
    idx = randperm(numObservationsTrain);
    documentsTrain = documentsTrain(idx);
    labelsTrain = labelsTrain(idx);
end
```



Test Model

To make predictions on a new set of data, use the `modelPredictions` function, listed at the end of the example. The `modelPredictions` function takes as input the model parameters, a word encoding, and an array of tokenized documents, and outputs the model predictions corresponding to the specified mini-batch size and the maximum sequence length.

```
dlYPredValidation = modelPredictions(parameters,enc,documentsValidation,miniBatchSize,maxSequence
```

To convert the network outputs to an array of labels, find the labels with scores higher than the specified label threshold.

```
YPredValidation = extractdata(dlYPredValidation) > labelThreshold;
```

To evaluate the performance, calculate the labeling F-score using the `labelingFScore` function, listed at the end of the example. The labeling F-score evaluates multilabel classification by focusing on per-text classification with partial matches.

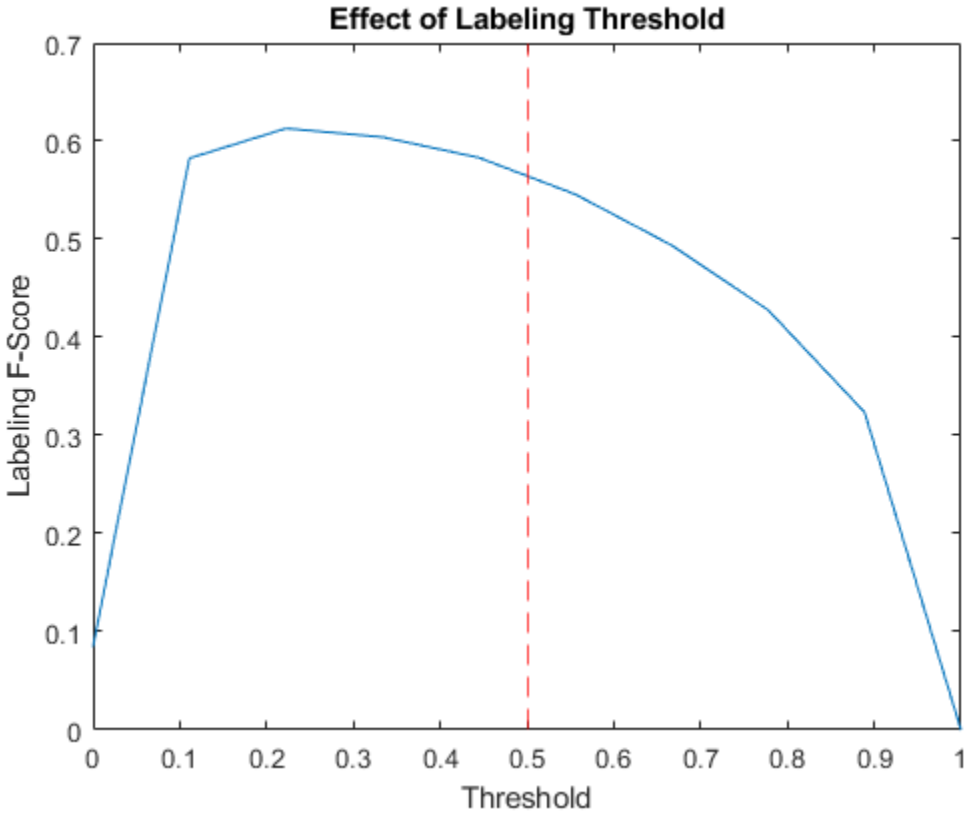
```
score = labelingFScore(YPredValidation,TValidation)
```

```
score = single
    0.5663
```

View the effect of the labeling threshold on the labeling F-score by trying a range of values for the threshold and comparing the results.

```
thr = linspace(0,1,10);
score = zeros(size(thr));
```

```
for i = 1:numel(thr)
    YPredValidationThr = extractdata(dlYPredValidation) >= thr(i);
    score(i) = labelingFScore(YPredValidationThr,TValidation);
end

figure
plot(thr,score)
xline(labelThreshold,'r--');
xlabel("Threshold")
ylabel("Labeling F-Score")
title("Effect of Labeling Threshold")
```



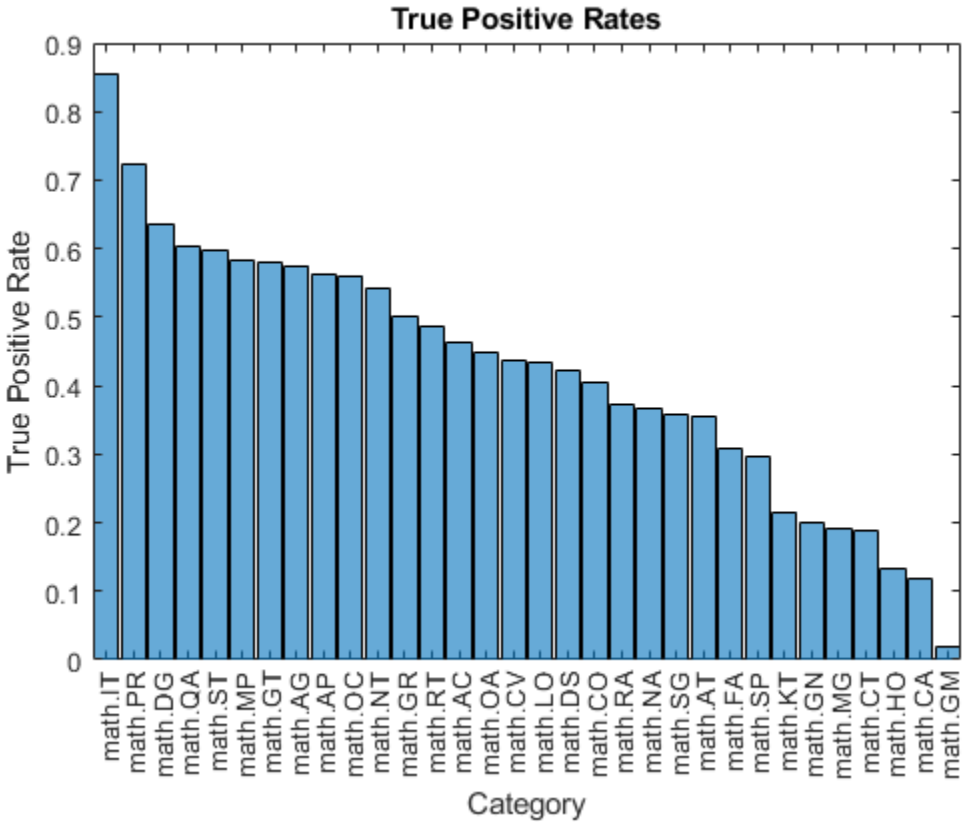
Visualize Predictions

To visualize the correct predictions of the classifier, calculate the numbers of true positives. A true positive is an instance of a classifier correctly predicting a particular class for an observation.

```
Y = YPredValidation;
T = TValidation;

numTruePositives = sum(T & Y,2);

numObservationsPerClass = sum(T,2);
truePositiveRates = numTruePositives ./ numObservationsPerClass;
```

Visualize the numbers of true positives for each class in a histogram.

```
figure
[~,idx] = sort(truePositiveRates,'descend');
histogram('Categories',classNames(idx),'BinCounts',truePositiveRates(idx))
xlabel("Category")
ylabel("True Positive Rate")
title("True Positive Rates")
```



Visualize the instances where the classifier predicts incorrectly by showing the distribution of true positives, false positives, and false negatives. A false positive is an instance of a classifier assigning a particular incorrect class to an observation. A false negative is an instance of a classifier failing to assign a particular correct class to an observation.

Create a confusion matrix showing the true positive, false positive, and false negative counts:

- For each class, display the true positive counts on the diagonal.
- For each pair of classes (i,j), display the number of instances of a false positive for j when the instance is also a false negative for i.

That is, the confusion matrix with elements given by:

$$\mathrm{TPFN}_{ij} = \begin{cases} \text{numTruePositives}(i), & \text{if } i = j \\ \text{numFalsePositives}(j \mid i \text{ is a false negative}), & \text{if } i \neq j \end{cases}$$

Calculate the false negatives and false positives.

```
falseNegatives = T & ~Y;
falsePositives = ~T & Y;
```

Calculate the off-diagonal elements.

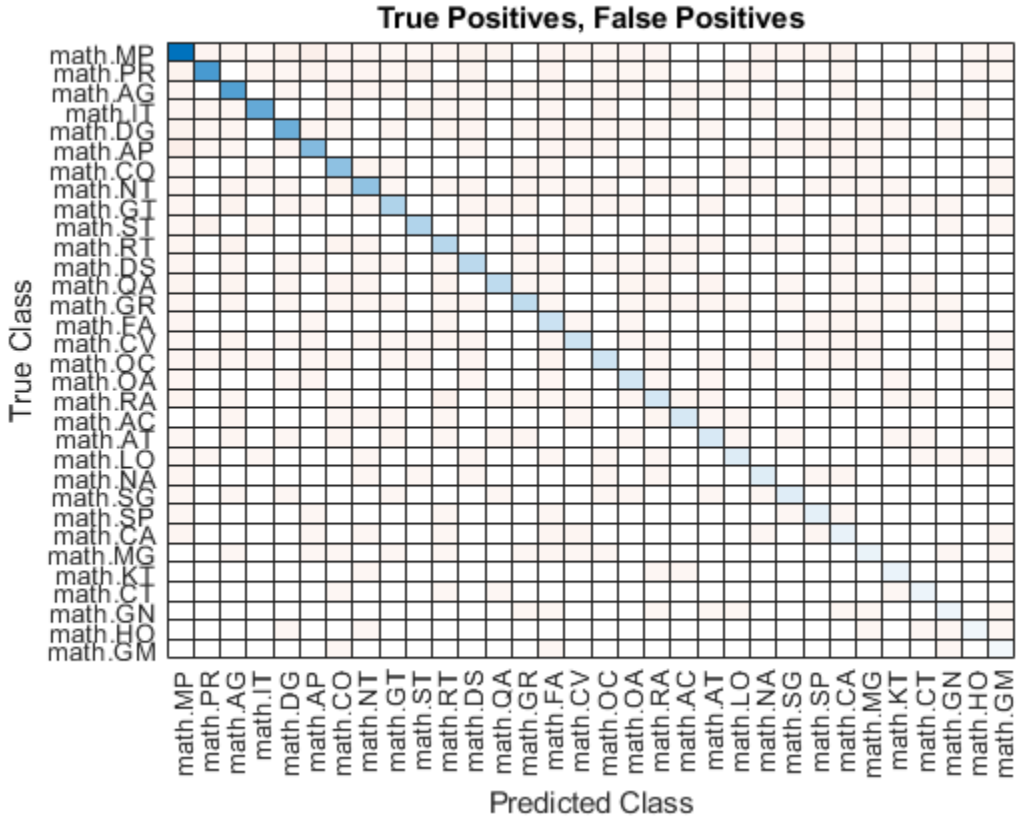
```
falseNegatives = permute(falseNegatives,[3 2 1]);
numConditionalFalsePositives = sum(falseNegatives & falsePositives, 2);
numConditionalFalsePositives = squeeze(numConditionalFalsePositives);

tpfnMatrix = numConditionalFalsePositives;
```

Set the diagonal elements to the true positive counts.

```
idxDiagonal = 1:numClasses+1:numClasses^2;
tpfnMatrix(idxDiagonal) = numTruePositives;
```

Visualize the true positive and false positive counts in a confusion matrix using the `confusionchart` function and sort the matrix such that the elements on the diagonal are in descending order.

```
figure
cm = confusionchart(tpfnMatrix,classNames);
sortClasses(cm,"descending-diagonal");
title("True Positives, False Positives")
```



To view the matrix in more detail, open this example as a live script and open the figure in a new window.

Preprocess Text Function

The `preprocessText` function tokenizes and preprocesses the input text data using the following steps:

1. Tokenize the text using the `tokenizedDocument` function. Extract mathematical equations as a single token using the `'RegularExpressions'` option by specifying the regular expression `"\$.*?\$"`, which captures text appearing between two "$" symbols.
2. Erase the punctuation using the `erasePunctuation` function.
3. Convert the text to lowercase using the `lower` function.
4. Remove the stop words using the `removeStopWords` function.
5. Lemmatize the text using the `normalizeWords` function with the `'Style'` option set to `'lemma'`.

```
function documents = preprocessText(textData)

% Tokenize the text.
regularExpressions = table;
regularExpressions.Pattern = "\$.*?\$";
regularExpressions.Type = "equation";

documents = tokenizedDocument(textData,'RegularExpressions',regularExpressions);

% Erase punctuation.
documents = erasePunctuation(documents);

% Convert to lowercase.
documents = lower(documents);

% Lemmatize.
documents = addPartOfSpeechDetails(documents);
documents = normalizeWords(documents,'Style','Lemma');

% Remove stop words.
documents = removeStopWords(documents);

% Remove short words.
documents = removeShortWords(documents,2);

end
```

Model Function

The function `model` takes as input the input data `dlX` and the model parameters `parameters`, and returns the predictions for the labels.

```
function dlY = model(dlX,parameters)

% Embedding
weights = parameters.emb.Weights;
dlX = embedding(dlX, weights);

% GRU
inputWeights = parameters.gru.InputWeights;
recurrentWeights = parameters.gru.RecurrentWeights;
bias = parameters.gru.Bias;

numHiddenUnits = size(inputWeights,1)/3;
hiddenState = dlarray(zeros([numHiddenUnits 1]));

dlY = gru(dlX, hiddenState, inputWeights, recurrentWeights, bias,'DataFormat','CBT');
```

```
% Max pooling along time dimension
dlY = max(dlY,[],3);

% Fully connect
weights = parameters.fc.Weights;
bias = parameters.fc.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CB');

% Sigmoid
dlY = sigmoid(dlY);

end
```

Model Gradients Function

The `modelGradients` function takes as input a mini-batch of input data `dlX` with corresponding targets `T` containing the labels and returns the gradients of the loss with respect to the learnable parameters, the corresponding loss, and the network outputs.

```
function [gradients,loss,dlYPred] = modelGradients(dlX,T,parameters)

dlYPred = model(dlX,parameters);

loss = crossentropy(dlYPred,T,'TargetCategories','independent','DataFormat','CB');

gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes as input the model parameters, a word encoding, an array of tokenized documents, a mini-batch size, and a maximum sequence length, and returns the model predictions by iterating over mini-batches of the specified size.

```
function dlYPred = modelPredictions(parameters,enc,documents,miniBatchSize,maxSequenceLength)

inputSize = enc.NumWords + 1;

numObservations = numel(documents);
numIterations = ceil(numObservations / miniBatchSize);

numFeatures = size(parameters.fc.Weights,1);
dlYPred = zeros(numFeatures,numObservations,'like',parameters.fc.Weights);

for i = 1:numIterations

    idx = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);

    len = min(maxSequenceLength,max(doclength(documents(idx))));
    X = doc2sequence(enc,documents(idx), ...
        'PaddingValue',inputSize, ...
        'Length',len);
    X = cat(1,X{:});

    dlX = dlarray(X,'BTC');
```

```
    dlYPred(:,idx) = model(dlX,parameters);
end

end
```

Labeling F-Score Function

The labeling F-score function [2] evaluates multilabel classification by focusing on per-text classification with partial matches. The measure is the normalized proportion of matching labels against the total number of true and predicted labels given by

$$\frac{1}{N}\sum_{n=1}^{N}\left(\frac{2\sum_{c=1}^{C}Y_{nc}T_{nc}}{\sum_{c=1}^{C}(Y_{nc}+T_{nc})}\right),$$

where N and C correspond to the number of observations and classes, respectively, and Y and T correspond to the predictions and targets, respectively.

```
function score = labelingFScore(Y,T)

numObservations = size(T,2);

scores = (2 * sum(Y .* T)) ./ sum(Y + T);
score = sum(scores) / numObservations;

end
```

Glorot Weights Initialization Function

The `initializeGlorot` function generates an array of weights according to Glorot initialization.

```
function weights = initializeGlorot(numOut, numIn)

varWeights = sqrt( 6 / (numIn + numOut) );
weights = varWeights * (2 * rand([numOut, numIn], 'single') - 1);

end
```

Gaussian Weights Initialization Function

The `initializeGaussian` function samples weights from a Gaussian distribution with mean 0 and standard deviation 0.01.

```
function parameter = initializeGaussian(sz)

parameter = randn(sz,'single') .* 0.01;

end
```

Embedding Function

The `embedding` function maps numeric indices to the corresponding vector given by the input weights.

```
function Z = embedding(X, weights)
% Reshape inputs into a vector.
[N, T] = size(X, 2:3);
X = reshape(X, N*T, 1);
```

```
% Index into embedding matrix.
Z = weights(:, X);

% Reshape outputs by separating batch and sequence dimensions.
Z = reshape(Z, [], N, T);
end
```

L_2 Norm Gradient Clipping Function

The `thresholdL2Norm` function scales the input gradients so that their L_2 norm values equal the specified gradient threshold when the L_2 norm value of the gradient of a learnable parameter is larger than the specified threshold.

```
function gradients = thresholdL2Norm(gradients,gradientThreshold)

gradientNorm = sqrt(sum(gradients(:).^2));
if gradientNorm > gradientThreshold
    gradients = gradients * (gradientThreshold / gradientNorm);
end

end
```

References

1. arXiv. "arXiv API." Accessed January 15, 2020. https://arxiv.org/help/api
2. Sokolova, Marina, and Guy Lapalme. "A Sytematic Analysis of Performance Measures for Classification Tasks." *Information Processing & Management* 45, no. 4 (2009): 427–437.

See Also

`tokenizedDocument` | `fullyconnect` | `gru` | `dlupdate` | `adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `wordEncoding` | `doc2sequence` | `extractHTMLText` | `htmlTree`

Related Examples

- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Sequence-to-Sequence Translation Using Attention” on page 4-164
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Classify Text Data Using Deep Learning” on page 4-89
- “Deep Learning Tips and Tricks” on page 1-67
- “Automatic Differentiation Background” on page 18-200

Classify Text Data Using Custom Training Loop

This example shows how to classify text data using a deep learning bidirectional long short-term memory (BiLSTM) network with a custom training loop.

When training a deep learning network using the `trainNetwork` function, if `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation. For an example showing how to classify text data using the `trainNetwork` function, see “Classify Text Data Using Deep Learning” on page 4-89.

This example trains a network to classify text data with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + kt}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.

Import Data

Import the factory reports data. This data contains labeled textual descriptions of factory events. To import the text data as strings, specify the text type to be `'string'`.

```
filename = "factoryReports.csv";
data = readtable(filename,'TextType','string');
head(data)
```

```
ans=8×5 table
                                Description                                 
    ____________________________________________________________________    

    "Items are occasionally getting stuck in the scanner spools."           
    "Loud rattling and banging sounds are coming from assembler pistons."   
    "There are cuts to the power when starting the plant."                  
    "Fried capacitors in the assembler."                                    
    "Mixer tripped the fuses."                                              
    "Burst pipe in the constructing agent is spraying coolant."             
    "A fuse is blown in the mixer."                                         
    "Things continue to tumble off of the belt."                            
```

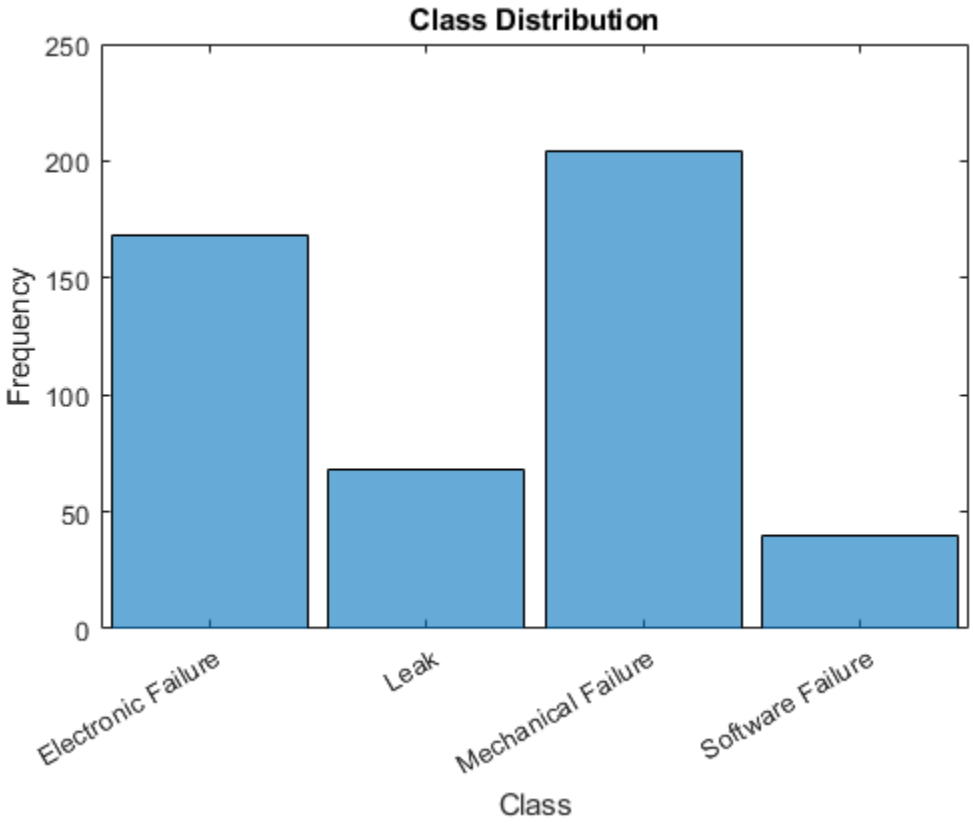
```
                Category         
    _____________________________

    "Mechanical Failure"
    "Mechanical Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Electronic Failure"
    "Leak"
    "Electronic Failure"
    "Mechanical Failure"
```

The goal of this example is to classify events by the label in the `Category` column. To divide the data into classes, convert these labels to categorical.

```
data.Category = categorical(data.Category);
```

View the distribution of the classes in the data using a histogram.

```
figure
histogram(data.Category);
xlabel("Class")
ylabel("Frequency")
title("Class Distribution")
```



The next step is to partition it into sets for training and validation. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 20%.

```
cvp = cvpartition(data.Category,'Holdout',0.2);
dataTrain = data(training(cvp),:);
dataValidation = data(test(cvp),:);
```

Extract the text data and labels from the partitioned tables.

```
textDataTrain = dataTrain.Description;
textDataValidation = dataValidation.Description;
YTrain = dataTrain.Category;
YValidation = dataValidation.Category;
```

To check that you have imported the data correctly, visualize the training text data using a word cloud.

```
figure
wordcloud(textDataTrain);
title("Training Data")
```

View the number of classes.

```
classes = categories(YTrain);
numClasses = numel(classes)
```

```
numClasses = 4
```

Preprocess Text Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

Preprocess the training data and the validation data using the `preprocessText` function.

```
documentsTrain = preprocessText(textDataTrain);
documentsValidation = preprocessText(textDataValidation);
```

View the first few preprocessed training documents.

```
documentsTrain(1:5)
```

```
ans =
  5×1 tokenizedDocument:
```

```
     9 tokens: items are occasionally getting stuck in the scanner spools
    10 tokens: loud rattling and banging sounds are coming from assembler pistons
     5 tokens: fried capacitors in the assembler
     4 tokens: mixer tripped the fuses
     9 tokens: burst pipe in the constructing agent is spraying coolant
```

Create a single datastore that contains both the documents and the labels by creating `arrayDatastore` objects, then combining them using the `combine` function.

```
dsDocumentsTrain = arrayDatastore(documentsTrain,'OutputType','cell');
dsYTrain = arrayDatastore(YTrain,'OutputType','cell');
dsTrain = combine(dsDocumentsTrain,dsYTrain);
```

Create a datastore for the validation data using the same steps.

```
dsDocumentsValidation = arrayDatastore(documentsValidation,'OutputType','cell');
dsYValidation = arrayDatastore(YValidation,'OutputType','cell');
dsValidation = combine(dsDocumentsValidation,dsYValidation);
```

Create Word Encoding

To input the documents into a BiLSTM network, use a word encoding to convert the documents into sequences of numeric indices.

To create a word encoding, use the `wordEncoding` function.

```
enc = wordEncoding(documentsTrain)
```

```
enc =
  wordEncoding with properties:

      NumWords: 421
    Vocabulary: [1×421 string]
```

Define Network

Define the BiLSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 25 and the same number of words as the word encoding. Next, include a BiLSTM layer and set the number of hidden units to 40. To use the BiLSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with the same size as the number of classes, and a softmax layer.

```
inputSize = 1;
embeddingDimension = 25;
numHiddenUnits = 40;

numWords = enc.NumWords;

layers = [
    sequenceInputLayer(inputSize,'Name','in')
    wordEmbeddingLayer(embeddingDimension,numWords,'Name','emb')
    bilstmLayer(numHiddenUnits,'OutputMode','last','Name','bilstm')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','sm')]
```

```
layers =
  5×1 Layer array with layers:

     1   'in'       Sequence Input         Sequence input with 1 dimensions
     2   'emb'      Word Embedding Layer   Word embedding layer with 25 dimensions and 421 unique
     3   'bilstm'   BiLSTM                 BiLSTM with 40 hidden units
     4   'fc'       Fully Connected        4 fully connected layer
     5   'sm'       Softmax                softmax
```

Convert the layer array to a layer graph and create a `dlnetwork` object.

```
lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [5×1 nnet.cnn.layer.Layer]
    Connections: [4×2 table]
     Learnables: [6×3 table]
          State: [2×3 table]
     InputNames: {'in'}
    OutputNames: {'sm'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object, a mini-batch of input data with corresponding labels, and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Specify Training Options

Train for 30 epochs with a mini-batch size of 16.

```
numEpochs = 30;
miniBatchSize = 16;
```

Specify the options for Adam optimization. Specify an initial learn rate of 0.001 with a decay of 0.01, gradient decay factor 0.9, and squared gradient decay factor 0.999.

```
initialLearnRate = 0.001;
decay = 0.01;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train Model

Train the model using a custom training loop.

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);

lineLossValidation = animatedline( ...
    'LineStyle','--', ...
    'Marker','o', ...
```

```
    'MarkerFaceColor','black');

ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Create a `minibatchqueue` object that processes and manages the mini-batches of data. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to convert documents to sequences and one-hot encode the labels. To pass the word encoding to the mini-batch, create an anonymous function that takes two inputs.
- Format the predictors with the dimension labels `'BTC'` (batch, time, channel). The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.
- Train on a GPU if one is available. The `minibatchqueue` object, by default, converts each output to `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain, ...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @(X,Y) preprocessMiniBatch(X,Y,enc), ...
    'MiniBatchFormat',{'BTC',''});
```

Create a `minibatchqueue` object for the validation data using the same options and also specify to return partial mini-batches.

```
mbqValidation = minibatchqueue(dsValidation, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn', @(X,Y) preprocessMiniBatch(X,Y,enc), ...
    'MiniBatchFormat',{'BTC',''}, ...
    'PartialMiniBatch','return');
```

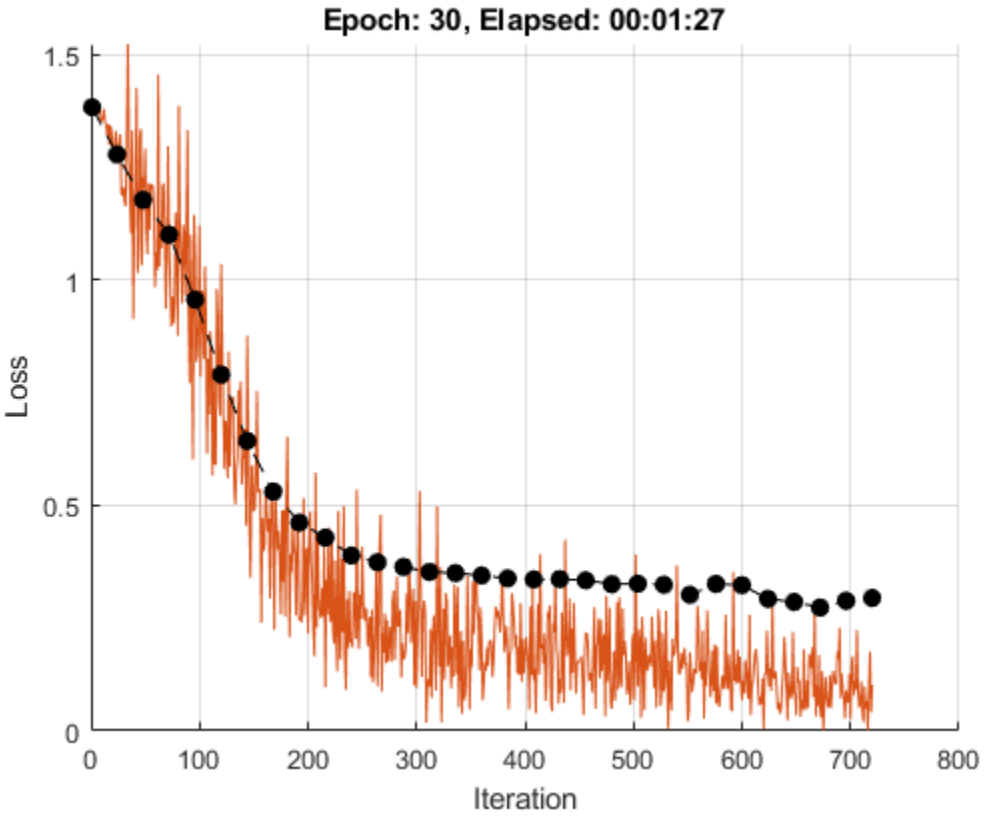
Train the network. For each epoch, shuffle the data and loop over mini-batches of data. At the end of each iteration, display the training progress. At the end of each epoch, validate the network using the validation data.

For each mini-batch:

- Convert the documents to sequences of integers and one-hot encode the labels.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'BTC'` (batch, time, channel).
- For GPU training, convert to `gpuArray` objects.
- Evaluate the model gradients, state, and loss using `dlfeval` and the `modelGradients` function and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule.

- Update the network parameters using the `adamupdate` function.
- Update the training plot.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX, dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the Adam optimizer.
        [dlnet,trailingAvg,trailingAvgSq] = adamupdate(dlnet, gradients, ...
            trailingAvg, trailingAvgSq, iteration, learnRate, ...
            gradientDecayFactor, squaredGradientDecayFactor);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow

        % Validate network.
        if iteration == 1 || ~hasdata(mbq)
            % Validation predictions.
            [~,lossValidation] = modelPredictions(dlnet,mbqValidation,classes);

            % Update plot.
            addpoints(lineLossValidation,iteration,lossValidation)
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

Classify the validation data using `modelPredictions` function, listed at the end of the example.

```
dlYPred = modelPredictions(dlnet,mbqValidation,classes);
YPred = onehotdecode(dlYPred,classes,1)';
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred == YValidation)
```

```
accuracy = 0.9167
```

Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
dsNew = arrayDatastore(documentsNew,'OutputType','cell');
```

Create a `minibatchqueue` object that processes and manages the mini-batches of data. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatchPredictors` (defined at the end of this example) to convert documents to sequences. This preprocessing function does not require label data. To pass the word encoding to the mini-batch, create an anonymous function that takes one input only.
- Format the predictors with the dimension labels `'BTC'` (batch, time, channel). The `minibatchqueue` object, by default, converts the data to `dlarray` objects with underlying type `single`.
- To make predictions for all observations, return any partial mini-batches.

```
mbqNew = minibatchqueue(dsNew, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@(X) preprocessMiniBatchPredictors(X,enc), ...
    'MiniBatchFormat','BTC', ...
    'PartialMiniBatch','return');
```

Classify the text data using `modelPredictions` function, listed at the end of the example and find the classes with the highest scores.

```
dlYPred = modelPredictions(dlnet,mbqNew,classes);
YPred = onehotdecode(dlYPred,classes,1)'
```

```
YPred = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Text Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

% Tokenize the text.
documents = tokenizedDocument(textData);

% Convert to lowercase.
documents = lower(documents);

% Erase punctuation.
documents = erasePunctuation(documents);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function converts a mini-batch of documents to sequences of integers and one-hot encodes label data.

```
function [X, Y] = preprocessMiniBatch(documentsCell,labelsCell,enc)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(documentsCell,enc);

% Extract labels from cell and concatenate.
Y = cat(1,labelsCell{1:end});

% One-hot encode labels.
Y = onehotencode(Y,2);

% Transpose the encoded labels to match the network output.
Y = Y';

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function converts a mini-batch of documents to sequences of integers.

```
function X = preprocessMiniBatchPredictors(documentsCell,enc)

% Extract documents from cell and concatenate.
documents = cat(4,documentsCell{1:end});

% Convert documents to sequences of integers.
X = doc2sequence(enc,documents);
X = cat(1,X{:});

end
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding target labels `T` and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,loss] = modelGradients(dlnet,dlX,T)

dlYPred = forward(dlnet,dlX);

loss = crossentropy(dlYPred,T);
gradients = dlgradient(loss,dlnet.Learnables);

loss = double(gather(extractdata(loss)));

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, a mini-batch queue, and outputs the model predictions by iterating over mini-batches in the queue. To evaluate validation data, this function optionally calcualtes the loss when given a mini-batch queue with two outputs.

```
function [dlYPred,loss] = modelPredictions(dlnet,mbq,classes)
```

```
% Initialize predictions.
numClasses = numel(classes);
outputCast = mbq.OutputCast{1};
dlYPred = dlarray(zeros(numClasses,0,outputCast),'CB');

% Reset mini-batch queue.
reset(mbq);

% For mini-batch queues with two ouputs, also compute the loss.
if mbq.NumOutputs == 1

    % Loop over mini-batches.
    while hasdata(mbq)

        % Make predictions.
        dlX = next(mbq);
        dlY = predict(dlnet,dlX);
        dlYPred = [dlYPred dlY];
    end

else
    % Initialize loss.
    numObservations = 0;
    loss = 0;

    % Loop over mini-batches.
    while hasdata(mbq)

        % Make predictions.
        [dlX,dlT] = next(mbq);
        dlY = predict(dlnet,dlX);
        dlYPred = [dlYPred dlY];

        % Calculate unnormalized loss.
        miniBatchSize = size(dlX,2);
        loss = loss + miniBatchSize * crossentropy(dlY, dlT);

        % Count observations.
        numObservations = numObservations + miniBatchSize;
    end

    % Normalize loss.
    loss = loss / numObservations;

    % Convert to double.
    loss = double(gather(extractdata(loss)));
end

end
```

See Also

wordEmbeddingLayer | tokenizedDocument | lstmLayer | doc2sequence | sequenceInputLayer | wordcloud | dlfeval | dlgradient | dlarray

Related Examples

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Classify Text Data Using Deep Learning” on page 4-89
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Generate Text Using Autoencoders

This example shows how to generate text data using autoencoders.

An autoencoder is a type of deep learning network that is trained to replicate its input. An autoencoder consists of two smaller networks: and encoder and a decoder. The encoder maps the input data to a feature vector in some latent space. The decoder reconstructs data using vectors in this latent space.

The training process is unsupervised. In other words, the model does not require labeled data. To generate text, you can use the decoder to reconstruct text from arbitrary input.

This example trains an autoencoder to generate text. The encoder uses a word embedding and an LSTM operation to map the input text into latent vectors. The decoder uses an LSTM operation and the same embedding to reconstruct the text from the latent vectors.

Load Data

The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.

Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.

```
filename = "sonnets.txt";
textData = fileread(filename);
```

The sonnets are indented by two whitespace characters. Remove the indentations using `replace` and split the text into separate lines using the `split` function. Remove the header from the first nine elements and the short sonnet titles.

```
textData = replace(textData,"  ","");
textData = split(textData,newline);
textData(1:9) = [];
textData(strlength(textData)<5) = [];
```

Prepare Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

Preprocess the text data and specify the start and stop tokens `"<start>"` and `"<stop>"`, respectively.

```
startToken = "<start>";
stopToken = "<stop>";
documents = preprocessText(textData,startToken,stopToken);
```

Create a word encoding object from the tokenized documents.

```
enc = wordEncoding(documents);
```

When training a deep learning model, the input data must be a numeric array containing sequences of a fixed length. Because the documents have different lengths, you must pad the shorter sequences with a padding value.

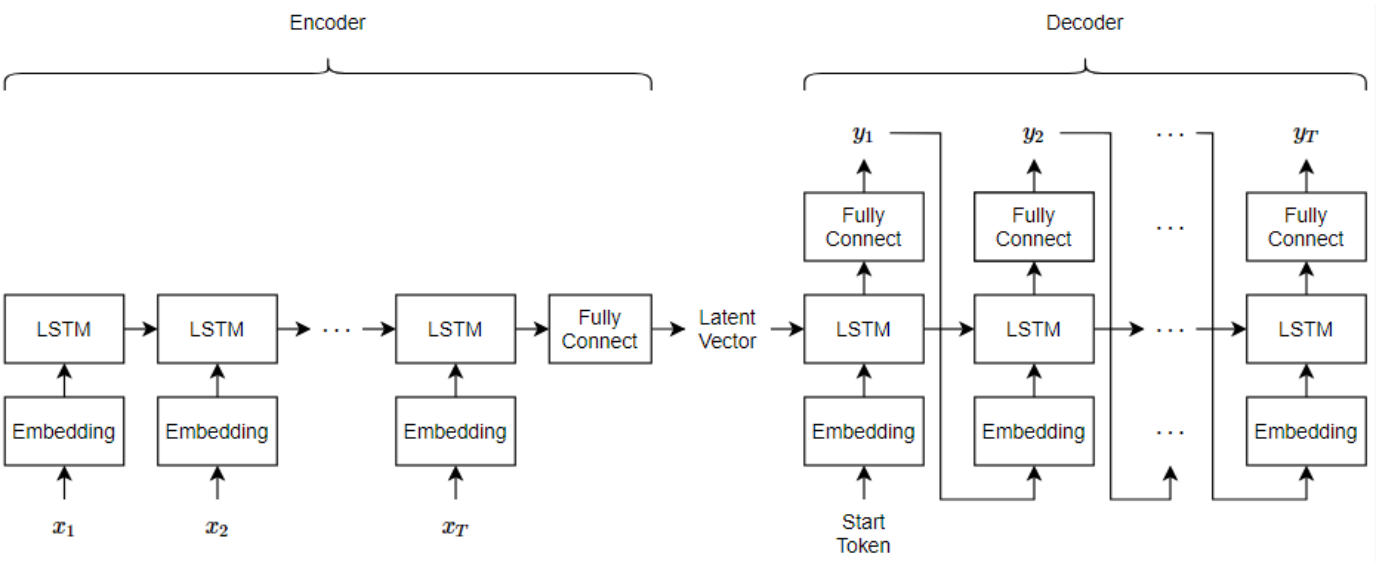
Recreate the word encoding to also include a padding token and determine the index of that token.

```
paddingToken = "<pad>";
newVocabulary = [enc.Vocabulary paddingToken];
enc = wordEncoding(newVocabulary);
paddingIdx = word2ind(enc,paddingToken)
```

```
paddingIdx = 3595
```

Initialize Model Parameters

Initialize the parameters for the following model.



Here, T is the sequence length, $x_1, \cdots, x_T$ is the input sequence of word indices, and $y_1, \cdots, y_T$ is the reconstructed sequence.

The encoder maps sequences of word indices to a latent vector by converting the input to sequences of word vectors using an embedding, inputting the word vector sequences into an LSTM operation, and applying a fully connected operation to the last time step of the LSTM output. The decoder reconstructs the input using an LSTM initialized the encoder output. For each time step, the decoder predicts the next time step and uses the output for the next time-step predictions. Both the encoder and the decoder use the same embedding.

Specify the dimensions of the parameters.

```
embeddingDimension = 100;
numHiddenUnits = 150;
latentDimension = 75;
vocabularySize = enc.NumWords;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” on page 18-299.

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” on page 18-296.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” on page 18-300.
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” on page 18-301.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmEncoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmEncoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits])
parameters.lstmEncoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” on page 18-302.

```
sz = [latentDimension numHiddenUnits];
numOut = latentDimension;
numIn = numHiddenUnits;

parameters.fcEncoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcEncoder.Bias = initializeZeros([latentDimension 1]);
```

Initialize the learnable parameters for the decoder LSTM operation:

- Initialize the input weights with the Glorot initializer.
- Initialize the recurrent weights with the orthogonal initializer.
- Initialize the bias with the unit forget gate initializer.

```
sz = [4*latentDimension embeddingDimension];
numOut = 4*latentDimension;
numIn = embeddingDimension;

parameters.lstmDecoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmDecoder.RecurrentWeights = initializeOrthogonal([4*latentDimension latentDimensio
parameters.lstmDecoder.Bias = initializeZeros([4*latentDimension 1]);
```

Initialize the learnable parameters for the decoder fully connected operation:

- Initialize the weights with the Glorot initializer.

- Initialize the bias with zeros.

```
sz = [vocabularySize latentDimension];
numOut = vocabularySize;
numIn = latentDimension;

parameters.fcDecoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcDecoder.Bias = initializeZeros([vocabularySize 1]);
```

To learn more about weight initialization, see “Initialize Learnable Parameters for Model Function” on page 18-292.

Define Model Encoder Function

Create the function `modelEncoder`, listed in the Encoder Model Function on page 4-0 section of the example, that computes the output of the encoder model. The `modelEncoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector. To learn more about defining a model encoder function, see “Define Text Encoder Model Function” on page 4-150.

Define Model Decoder Function

Create the function `modelDecoder`, listed in the Decoder Model Function on page 4-0 section of the example, that computes the output of the decoder model. The `modelDecoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector. To learn more about defining a model decoder function, see “Define Text Decoder Model Function” on page 4-157.

Define Model Gradients Function

The `modelGradients` function, listed in the Model Gradients Function on page 4-0 section of the example, takes as input the model learnable parameters, the input data `dlX`, and a vector of sequence lengths for masking, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. To learn more about defining a model gradients function, see “Define Model Gradients Function for Custom Training Loop” on page 18-231.

Specify Training Options

Specify the options for training.

Train for 100 epochs with a mini-batch size of 128.

```
miniBatchSize = 128;
numEpochs = 100;
```

Train with a learning rate of 0.01.

```
learnRate = 0.01;
```

Display the training progress in a plot.

```
plots = "training-progress";
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Train Network

Train the network using a custom training loop.

Initialize the parameters for the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

Initialize the training progress plot. Create an animated line that plots the loss against the corresponding iteration.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    xlabel("Iteration")
    ylabel("Loss")
    ylim([0 inf])
    grid on
end
```

Train the model. For the first epoch, shuffle the data and loop over mini-batches of data.

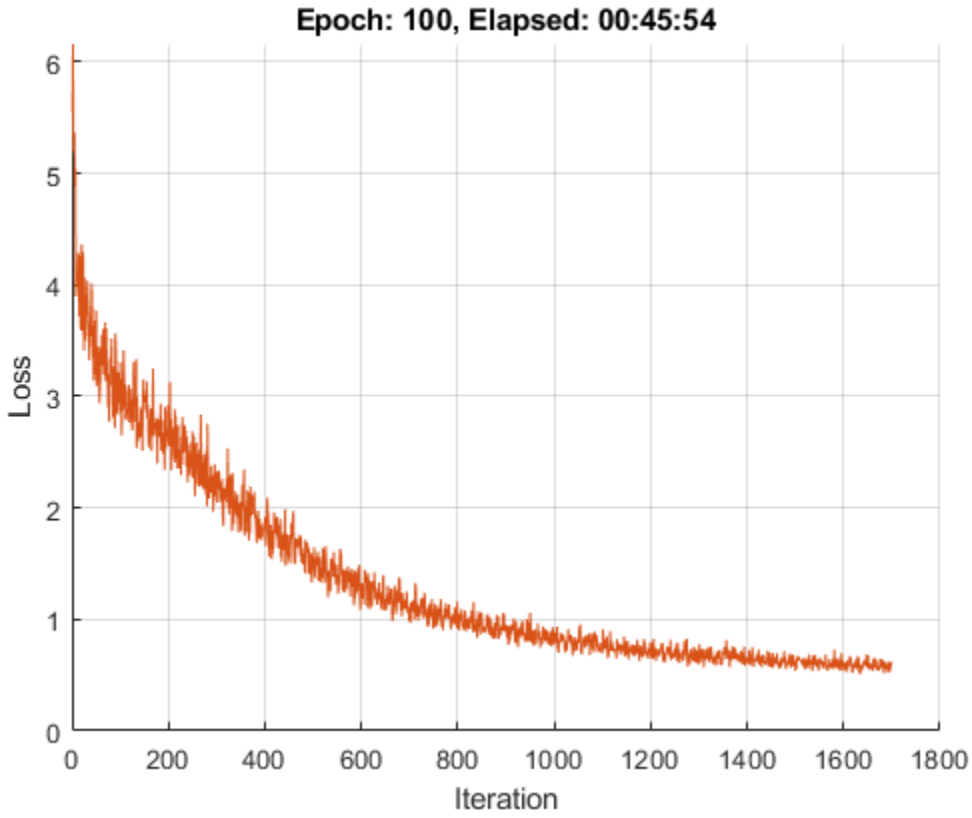
For each mini-batch:

- Convert the text data to sequences of word indices.
- Convert the data to `dlarray`.
- For GPU training, convert the data to `gpuArray` objects.
- Compute loss and gradients.
- Update the learnable parameters using the `adamupdate` function.
- Update the training progress plot.

Training can take some time to run.

```
numObservations = numel(documents);
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

iteration = 0;
start = tic;

for epoch = 1:numEpochs

    % Shuffle.
    idx = randperm(numObservations);
    documents = documents(idx);

    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        documentsBatch = documents(idx);

        % Convert to sequences.
```

```
            X = doc2sequence(enc,documentsBatch, ...
                'PaddingDirection','right', ...
                'PaddingValue',paddingIdx);
            X = cat(1,X{:});

            % Convert to dlarray.
            dlX = dlarray(X,'BTC');

            % If training on a GPU, then convert data to gpuArray.
            if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
                dlX = gpuArray(dlX);
            end

            % Calculate sequence lengths.
            sequenceLengths = doclength(documentsBatch);

            % Evaluate model gradients.
            [gradients,loss] = dlfeval(@modelGradients, parameters, dlX, sequenceLengths);

            % Update learnable parameters.
            [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
                trailingAvg,trailingAvgSq,iteration,learnRate);

            % Display the training progress.
            if plots == "training-progress"
                D = duration(0,0,toc(start),'Format','hh:mm:ss');
                addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
                title("Epoch: " + epoch + ", Elapsed: " + string(D))

                drawnow
            end
        end
    end
```



Generate Text

Generate text using closed loop generation by initializing the decoder with different random states. Closed loop generation is when the model generates data one time-step at a time and uses the previous prediction as input for the next prediction.

Specify to generate 3 sequences of length 16.

```
numGenerations = 3;
sequenceLength = 16;
```

Create an array of random values to initialize the decoder state.

```
dlZ = dlarray(randn(latentDimension,numGenerations),'CB');
```

If predicting on a GPU, then convert data to `gpuArray`.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
end
```

Make predictions using the `modelPredictions` function, listed at the end of the example. The `modelPredictions` function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

```
dlY = modelDecoderPredictions(parameters,dlZ,sequenceLength,enc,startToken,miniBatchSize);
```

Find the word indices with the highest scores.

```
[~,idx] = max(dlY,[],1);
idx = squeeze(idx);
```

Convert the numeric indices to words and join them using the `join` function.

```
strGenerated = join(enc.Vocabulary(idx));
```

Extract the text before the first stop token using the `extractBefore` function. To prevent the function from returning missing when there are no stop tokens, append a stop token to the end of each sequence.

```
strGenerated = extractBefore(strGenerated+stopToken,stopToken);
```

Remove padding tokens.

```
strGenerated = erase(strGenerated,paddingToken);
```

The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.

Remove the spaces that appear before the specified punctuation characters.

```
punctuationCharacters = ["." "," "’" ")" ":" ";" "?" "!"];
strGenerated = replace(strGenerated," " + punctuationCharacters,punctuationCharacters);
```

Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
strGenerated = replace(strGenerated,punctuationCharacters + " ",punctuationCharacters);
```

Remove leading and trailing white space using the `strip` function and view the generated text.

```
strGenerated = strip(strGenerated)
```

```
strGenerated = 3×1 string
    "love's thou rest light best ill mistake show seeing farther cross enough by me"
    "as before his bending sickle's compass come look find."
    "summer's lays? truth once lead mayst take,"
```

Encoder Model Function

The `modelEncoder` function, takes as input the model parameters, sequences of word indices, and the sequence lengths, and returns the corresponding latent feature vector.

Because the input data contains padded sequences of different lengths, the padding can have adverse effects on loss calculations. For the LSTM operation, instead of returning the output of the last time step of the sequence (which likely corresponds to the LSTM state after processing lots of padding values), determine the actual last time step given by the `sequenceLengths` input.

```
function dlZ = modelEncoder(parameters,dlX,sequenceLengths)

% Embedding.
weights = parameters.emb.Weights;
dlZ = embedding(dlX,weights);

% LSTM.
```

```
inputWeights = parameters.lstmEncoder.InputWeights;
recurrentWeights = parameters.lstmEncoder.RecurrentWeights;
bias = parameters.lstmEncoder.Bias;

numHiddenUnits = size(recurrentWeights,2);
hiddenState = zeros(numHiddenUnits,1,'like',dlX);
cellState = zeros(numHiddenUnits,1,'like',dlX);

dlZ1 = lstm(dlZ,hiddenState,cellState,inputWeights,recurrentWeights,bias,'DataFormat','CBT');

% Output mode 'last' with masking.
miniBatchSize = size(dlZ1,2);
dlZ = zeros(numHiddenUnits,miniBatchSize,'like',dlZ1);

for n = 1:miniBatchSize
    t = sequenceLengths(n);
    dlZ(:,n) = dlZ1(:,n,t);
end

% Fully connect.
weights = parameters.fcEncoder.Weights;
bias = parameters.fcEncoder.Bias;
dlZ = fullyconnect(dlZ,weights,bias,'DataFormat','CB');

end
```

Decoder Model Function

The `modelDecoder` function, takes as input the model parameters, sequences of word indices, and the network state, and returns the decoded sequences.

Because the `lstm` function is *stateful* (when given a time series as input, the function propagates and updates the state between each time step) and that the `embedding` and `fullyconnect` functions are time-distributed by default (when given a time series as input, the functions operate on each time step independently), the `modelDecoder` function supports both sequence and single time-step inputs.

```
function [dlY,state] = modelDecoder(parameters,dlX,state)

% Embedding.
weights = parameters.emb.Weights;
dlX = embedding(dlX,weights);

% LSTM.
inputWeights = parameters.lstmDecoder.InputWeights;
recurrentWeights = parameters.lstmDecoder.RecurrentWeights;
bias = parameters.lstmDecoder.Bias;

hiddenState = state.HiddenState;
cellState = state.CellState;

[dlY,hiddenState,cellState] = lstm(dlX,hiddenState,cellState, ...
    inputWeights,recurrentWeights,bias,'DataFormat','CBT');

state.HiddenState = hiddenState;
state.CellState = cellState;

% Fully connect.
```

```
weights = parameters.fcDecoder.Weights;
bias = parameters.fcDecoder.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

% Softmax.
dlY = softmax(dlY,'DataFormat','CBT');

end
```

Model Gradients Function

The `modelGradients` function that takes as input the model learnable parameters, the input data `dlX`, and a vector of sequence lengths for masking, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

To calculate the masked loss, the model gradients function uses the `maskedCrossEntropy` loss function, listed at the end of the example. To train the decoder to predict the next time-step of the sequence, specify the targets to be the input sequences shifted by one time-step.

To learn more about defining a model gradients function, see “Define Model Gradients Function for Custom Training Loop” on page 18-231.

```
function [gradients, loss] = modelGradients(parameters,dlX,sequenceLengths)

% Model encoder.
dlZ = modelEncoder(parameters,dlX,sequenceLengths);

% Initialize LSTM state.
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);

% Teacher forcing.
dlY = modelDecoder(parameters,dlX,state);

% Loss.
dlYPred = dlY(:,:,1:end-1);
dlT = dlX(:,:,2:end);
loss = mean(maskedCrossEntropy(dlYPred,dlT,sequenceLengths));

% Gradients.
gradients = dlgradient(loss,parameters);

% Normalize loss for plotting.
sequenceLength = size(dlX,3);
loss = loss / sequenceLength;

end
```

Model Predictions Function

The `modelPredictions` function returns the output scores of the decoder given the model parameters, decoder initial state, maximum sequence length, word encoding, start token, and mini-batch size.

```
function dlY = modelDecoderPredictions(parameters,dlZ,maxLength,enc,startToken,miniBatchSize)

numObservations = size(dlZ,2);
```

```
numIterations = ceil(numObservations / miniBatchSize);

startTokenIdx = word2ind(enc,startToken);
vocabularySize = enc.NumWords;

dlY = zeros(vocabularySize,numObservations,maxLength,'like',dlZ);

% Loop over mini-batches.
for i = 1:numIterations
    idxMiniBatch = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);
    miniBatchSize = numel(idxMiniBatch);

    % Initialize state.
    state = struct;
    state.HiddenState = dlZ(:,idxMiniBatch);
    state.CellState = zeros(size(dlZ(:,idxMiniBatch)),'like',dlZ);

    % Initialize decoder input.
    decoderInput = dlarray(repmat(startTokenIdx,[1 miniBatchSize]),'CBT');

    % Loop over time steps.
    for t = 1:maxLength
        % Predict next time step.
        [dlY(:,idxMiniBatch,t), state] = modelDecoder(parameters,decoderInput,state);

        % Closed loop generation.
        [~,idx] = max(dlY(:,idxMiniBatch,t));
        decoderInput = idx;
    end
end

end
```

Masked Cross Entropy Loss Function

The `maskedCrossEntropy` function calculates the loss between the specified input sequences and target sequences ignoring any time steps containing padding using the specified vector of sequence lengths.

```
function maskedLoss = maskedCrossEntropy(dlY,T,sequenceLengths)

numClasses = size(dlY,1);
miniBatchSize = size(dlY,2);
sequenceLength = size(dlY,3);

maskedLoss = zeros(sequenceLength,miniBatchSize,'like',dlY);

for t = 1:sequenceLength
    T1 = single(oneHot(T(:,:,t),numClasses));

    mask = (t<=sequenceLengths)';

    maskedLoss(t,:) = mask .* crossentropy(dlY(:,:,t),T1,'DataFormat','CBT');
end

maskedLoss = sum(maskedLoss,1);

end
```

Text Preprocessing Function

The function `preprocessText` performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

```
function documents = preprocessText(textData,startToken,stopToken)

% Add start and stop tokens.
textData = startToken + textData + stopToken;

% Tokenize the text.
documents = tokenizedDocument(textData,'CustomTokens',[startToken stopToken]);

end
```

Embedding Function

The `embedding` function maps sequences of indices to vectors using the given weights.

```
function Z = embedding(X, weights)

% Reshape inputs into a vector.
[N, T] = size(X, 2:3);
X = reshape(X, N*T, 1);

% Index into embedding matrix.
Z = weights(:, X);

% Reshape outputs by separating out batch and sequence dimensions.
Z = reshape(Z, [], N, T);

end
```

One-Hot Encoding Function

The `oneHot` function converts an array of numeric indices to one-hot encoded vectors.

```
function oh = oneHot(idx, outputSize)

miniBatchSize = numel(idx);
oh = zeros(outputSize,miniBatchSize);

for n = 1:miniBatchSize
    c = idx(n);
    oh(c,n) = 1;
end

end
```

See Also

`dlfeval` | `dlgradient` | `dlarray`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Text Encoder Model Function” on page 4-150
- “Define Text Decoder Model Function” on page 4-157

Define Text Encoder Model Function

This example shows how to define a text encoder model function.

In the context of deep learning, an encoder is the part of a deep learning network that maps the input to some latent space. You can use these vectors for various tasks. For example,

- Classification by applying a softmax operation to the encoded data and using cross entropy loss.
- Sequence-to-sequence translation by using the encoded vector as a context vector.

Load Data

The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.

Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.

```
filename = "sonnets.txt";
textData = fileread(filename);
```

The sonnets are indented by two whitespace characters. Remove the indentations using `replace` and split the text into separate lines using the `split` function. Remove the header from the first nine elements and the short sonnet titles.

```
textData = replace(textData,"  ","");
textData = split(textData,newline);
textData(1:9) = [];
textData(strlength(textData)<5) = [];
```

Prepare Data

Create a function that tokenizes and preprocesses the text data. The function `preprocessText`, listed at the end of the example, performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

Preprocess the text data and specify the start and stop tokens `"<start>"` and `"<stop>"`, respectively.

```
startToken = "<start>";
stopToken = "<stop>";
documents = preprocessText(textData,startToken,stopToken);
```

Create a word encoding object from the tokenized documents.

```
enc = wordEncoding(documents);
```

When training a deep learning model, the input data must be a numeric array containing sequences of a fixed length. Because the documents have different lengths, you must pad the shorter sequences with a padding value.

Recreate the word encoding to also include a padding token and determine the index of that token.

```
paddingToken = "<pad>";
newVocabulary = [enc.Vocabulary paddingToken];
enc = wordEncoding(newVocabulary);
paddingIdx = word2ind(enc,paddingToken)
```

```
paddingIdx = 3595
```

Initialize Model Parameters

The goal of the encoder is to map sequences of word indices to vectors in some latent space.

Initialize the parameters for the following model.

This model uses three operations:

- The embedding maps word indices in the range 1 though `vocabularySize` to vectors of dimension `embeddingDimension`, where `vocabularySize` is the number of words in the encoding vocabulary and `embeddingDimension` is the number of components learned by the embedding.
- The LSTM operation takes as input sequences of word vectors and outputs 1-by-`numHiddenUnits` vectors, where `numHiddenUnits` is the number of hidden units in the LSTM operation.
- The fully connected operation multiplies the input by a weight matrix adding bias and outputs vectors of size `latentDimension`, where `latentDimension` is the dimension of the latent space.

Specify the dimensions of the parameters.

```
embeddingDimension = 100;
numHiddenUnits = 150;
latentDimension = 50;
vocabularySize = enc.NumWords;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” on page 18-299.

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” on page 18-296.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” on page 18-300.

- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” on page 18-301.

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the LSTM operation are sequences of word vectors from the embedding operation, the number of input channels is `embeddingDimension`.

- The input weight matrix has size `4*numHiddenUnits`-by-`inputSize`, where `inputSize` is the dimension of the input data.
- The recurrent weight matrix has size `4*numHiddenUnits`-by-`numHiddenUnits`.
- The bias vector has size `4*numHiddenUnits`-by-1.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmEncoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmEncoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits])
parameters.lstmEncoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” on page 18-302.

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the fully connected operation are the outputs of the LSTM operation, the number of input channels is `numHiddenUnits`. To make the fully connected operation output vectors with size `latentDimension`, specify an output size of `latentDimension`.

- The weights matrix has size `outputSize`-by-`inputSize`, where `outputSize` and `inputSize` correspond to the output and input dimensions, respectively.
- The bias vector has size `outputSize`-by-1.

```
sz = [latentDimension numHiddenUnits];
numOut = latentDimension;
numIn = numHiddenUnits;

parameters.fcEncoder.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fcEncoder.Bias = initializeZeros([latentDimension 1]);
```

Define Model Encoder Function

Create the function `modelEncoder`, listed in the Encoder Model Function on page 4-0 section of the example, that computes the output of the encoder model. The `modelEncoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector.

Prepare Mini-Batch of Data

To train the model using a custom training loop, you must iterate over mini-batches of data and convert it into the format required for the encoder model and the model gradients functions. This

section of the example illustrates the steps needed for preparing a mini-batch of data inside the custom training loop.

Prepare an example mini-batch of data. Select a mini-batch of 32 documents from `documents`. This represents the mini-batch of data used in an iteration of a custom training loop.

```
miniBatchSize = 32;
idx = 1:miniBatchSize;
documentsBatch = documents(idx);
```

Convert the documents to sequences using the `doc2sequence` function and specify to right-pad the sequences with the word index corresponding to the padding token.

```
X = doc2sequence(enc,documentsBatch, ...
    'PaddingDirection','right', ...
    'PaddingValue',paddingIdx);
```

The output of the `doc2sequence` function is a cell array, where each element is a row vector of word indices. Because the encoder model function requires numeric input, concatenate the rows of the data using the `cat` function and specify to concatenate along the first dimension. The output has size `miniBatchSize`-by-`sequenceLength`, where `sequenceLength` is the length of the longest sequence in the mini-batch.

```
X = cat(1,X{:});
size(X)
```

```
ans = 1×2

    32    14
```

Convert the data to a `dlarray` with format `'BTC'` (batch, time, channel). The software automatically rearranges the output to have format `'CTB'` so the output has size `1`-by-`miniBatchSize`-by-`sequenceLength`.

```
dlX = dlarray(X,'BTC');
size(dlX)
```

```
ans = 1×3

     1    32    14
```

For masking, calculate the unpadded sequence lengths of the input data using the `doclength` function with the mini-batch of documents as input.

```
sequenceLengths = doclength(documentsBatch);
```

This code snippet shows an example of preparing a mini-batch in a custom training loop.

```
iteration = 0;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
```

```
        iteration = iteration + 1;

        % Read mini-batch.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        documentsBatch = documents(idx);

        % Convert to sequences.
        X = doc2sequence(enc,documentsBatch, ...
            'PaddingDirection','right', ...
            'PaddingValue',paddingIdx);
        X = cat(1,X{:});

        % Convert to dlarray.
        dlX = dlarray(X,'BTC');

        % Calculate sequence lengths.
        sequenceLengths = doclength(documentsBatch);

        % Evaluate model gradients.
        % ...

        % Update learnable parameters.
        % ...
    end
end
```

Use Model Function in Model Gradients Function

When training a deep learning model with a custom training loop, you must calculate the gradients of the loss with respect to the learnable parameters. This calculation depends on the output of a forward pass of the model function.

To perform a forward pass of the encoder, use the `modelEncoder` function directly with the parameters, data, and sequence lengths as input. The output is a `latentDimension`-by-`miniBatchSize` matrix.

```
dlZ = modelEncoder(parameters,dlX,sequenceLengths);
size(dlZ)
```

```
ans = 1×2

    50    32
```

This code snippet shows an example of using a model encoder function inside the model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    dlZ = modelEncoder(parameters,dlX,sequenceLengths);

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

This code snippet shows an example of evaluating the model gradients in a custom training loop.

```
iteration = 0;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Prepare mini-batch.
        % ...

        % Evaluate model gradients.
        gradients = dlfeval(@modelGradients, parameters, dlX, sequenceLengths);

        % Update learnable parameters.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);
    end
end
```

Encoder Model Function

The `modelEncoder` function, takes as input the model parameters, sequences of word indices, and the sequence lengths, and returns the corresponding latent feature vector.

Because the input data contains padded sequences of different lengths, the padding can have adverse effects on loss calculations. For the LSTM operation, instead of returning the output of the last time step of the sequence (which likely corresponds to the LSTM state after processing lots of padding values), determine the actual last time step given by the `sequenceLengths` input.

```
function dlZ = modelEncoder(parameters,dlX,sequenceLengths)

% Embedding.
weights = parameters.emb.Weights;
dlZ = embed(dlX,weights);

% LSTM.
inputWeights = parameters.lstmEncoder.InputWeights;
recurrentWeights = parameters.lstmEncoder.RecurrentWeights;
bias = parameters.lstmEncoder.Bias;

numHiddenUnits = size(recurrentWeights,2);
hiddenState = zeros(numHiddenUnits,1,'like',dlX);
cellState = zeros(numHiddenUnits,1,'like',dlX);

dlZ1 = lstm(dlZ,hiddenState,cellState,inputWeights,recurrentWeights,bias);

% Output mode 'last' with masking.
miniBatchSize = size(dlZ1,2);
dlZ = zeros(numHiddenUnits,miniBatchSize,'like',dlZ1);
dlZ = dlarray(dlZ,'CB');

for n = 1:miniBatchSize
    t = sequenceLengths(n);
    dlZ(:,n) = dlZ1(:,n,t);
```

```
end

% Fully connect.
weights = parameters.fcEncoder.Weights;
bias = parameters.fcEncoder.Bias;
dlZ = fullyconnect(dlZ,weights,bias);

end
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Prepends and appends each input string with the specified start and stop tokens, respectively.
2. Tokenize the text using `tokenizedDocument`.

```
function documents = preprocessText(textData,startToken,stopToken)

% Add start and stop tokens.
textData = startToken + textData + stopToken;

% Tokenize the text.
documents = tokenizedDocument(textData,'CustomTokens',[startToken stopToken]);

end
```

See Also

`dlfeval` | `dlgradient` | `dlarray`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Generate Text Using Autoencoders” on page 4-137
- “Define Text Decoder Model Function” on page 4-157

Define Text Decoder Model Function

This example shows how to define a text decoder model function.

In the context of deep learning, a decoder is the part of a deep learning network that maps a latent vector to some sample space. You can use decode the vectors for various tasks. For example,

- Text generation by initializing a recurrent network with the encoded vector.
- Sequence-to-sequence translation by using the encoded vector as a context vector.
- Image captioning by using the encoded vector as a context vector.

Load Data

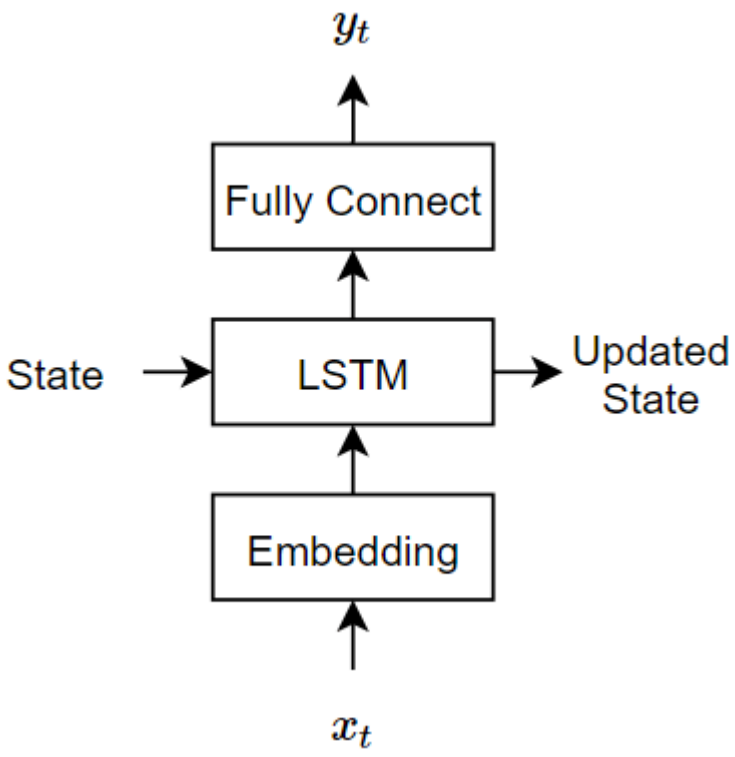
Load the encoded data from `sonnetsEncoded.mat`. This MAT file contains the word encoding, a mini-batch of sequences `dlX`, and the corresponding encoded data `dlZ` output by the encoder used in the example “Define Text Encoder Model Function” on page 4-150.

```
s = load("sonnetsEncoded.mat");
enc = s.enc;
dlX = s.dlX;
dlZ = s.dlZ;

[latentDimension,miniBatchSize] = size(dlZ,1:2);
```

Initialize Model Parameters

The goal of the decoder is to generate sequences given some initial input data and network state.

Initialize the parameters for the following model.



The decoder reconstructs the input using an LSTM initialized the encoder output. For each time step, the decoder predicts the next time step and uses the output for the next time-step predictions. Both the encoder and the decoder use the same embedding.

This model uses three operations:

- The embedding maps word indices in the range 1 though `vocabularySize` to vectors of dimension `embeddingDimension`, where `vocabularySize` is the number of words in the encoding vocabulary and `embeddingDimension` is the number of components learned by the embedding.
- The LSTM operation takes as input a single word vector and outputs 1-by-`numHiddenUnits` vector, where `numHiddenUnits` is the number of hidden units in the LSTM operation. The initial state of the LSTM network (the state at the first time-step) is the encoded vector, so the number of hidden units must match the latent dimension of the encoder.
- The fully connected operation multiplies the input by a weight matrix adding bias and outputs vectors of size `vocabularySize`.

Specify the dimensions of the parameters. The embedding sizes must match the encoder.

```
embeddingDimension = 100;
vocabularySize = enc.NumWords;
numHiddenUnits = latentDimension;
```

Create a struct for the parameters.

```
parameters = struct;
```

Initialize the weights of the embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” on page 18-299.

```
mu = 0;
sigma = 0.01;
parameters.emb.Weights = initializeGaussian([embeddingDimension vocabularySize],mu,sigma);
```

Initialize the learnable parameters for the decoder LSTM operation:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” on page 18-296.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” on page 18-300.
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” on page 18-301.

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the LSTM operation are sequences of word vectors from the embedding operation, the number of input channels is `embeddingDimension`.

- The input weight matrix has size `4*numHiddenUnits`-by-`inputSize`, where `inputSize` is the dimension of the input data.

- The recurrent weight matrix has size `4*numHiddenUnits`-by-`numHiddenUnits`.
- The bias vector has size `4*numHiddenUnits`-by-1.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.lstmDecoder.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.lstmDecoder.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits]);
parameters.lstmDecoder.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the encoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” on page 18-302.

The sizes of the learnable parameters depend on the size of the input. Because the inputs to the fully connected operation are the outputs of the LSTM operation, the number of input channels is `numHiddenUnits`. To make the fully connected operation output vectors with size `latentDimension`, specify an output size of `latentDimension`.

- The weights matrix has size `outputSize`-by-`inputSize`, where `outputSize` and `inputSize` correspond to the output and input dimensions, respectively.
- The bias vector has size `outputSize`-by-1.

To make the fully connected operation output vectors with size `vocabularySize`, specify an output size of `vocabularySize`.

```
inputSize = numHiddenUnits;
outputSize = vocabularySize;
parameters.fcDecoder.Weights = dlarray(randn(outputSize,inputSize,'single'));
parameters.fcDecoder.Bias = dlarray(zeros(outputSize,1,'single'));
```

Define Model Decoder Function

Create the function `modelDecoder`, listed in the Decoder Model Function on page 4-0 section of the example, that computes the output of the decoder model. The `modelDecoder` function, takes as input sequences of word indices, the model parameters, and the sequence lengths, and returns the corresponding latent feature vector.

Use Model Function in Model Gradients Function

When training a deep learning model with a custom training loop, you must calculate the gradients of the loss with respect to the learnable parameters. This calculation depends on the output of a forward pass of the model function.

There are two common approaches to generating text data with a decoder:

1. Closed loop — For each time step, make predictions using the previous prediction as input.
2. Open loop — For each time step, make predictions using inputs from an external source (for example, training targets).

Closed Loop Generation

Closed loop generation is when the model generates data one time-step at a time and uses the previous prediction as input for the next prediction. Unlike open loop generation, this process does

not require any input between predictions and is best suited for scenarios without supervision. For example, a language translation model that generates output text in one go.

To use closed loop

Initialize the hidden state of the LSTM network with the encoder output `dlZ`.

```
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);
```

For the first time step, use an array of start tokens as input for the decoder. For simplicity, extract an array of start tokens from the first time-step of the training data.

```
decoderInput = dlX(:,:,1);
```

Preallocate the decoder output to have size `numClasses`-by-`miniBatchSize`-by-`sequenceLength` with the same datatype as `dlX`, where `sequenceLength` is the desired length of the generation, for example, the length of the training targets. For this example, specify a sequence length of 16.

```
sequenceLength = 16;
dlY = zeros(vocabularySize,miniBatchSize,sequenceLength,'like',dlX);
dlY = dlarray(dlY,'CBT');
```

For each time step, predict the next time step of the sequence using the `modelDecoder` function. After each prediction, find the indices corresponding to the maximum values of the decoder output and use these indices as the decoder input for the next time step.

```
for t = 1:sequenceLength
    [dlY(:,:,t), state] = modelDecoder(parameters,decoderInput,state);

    [~,idx] = max(dlY(:,:,t));
    decoderInput = idx;
end
```

The output is a `vocabularySize`-by-`miniBatchSize`-by-`sequenceLength` array.

```
size(dlY)
```

```
ans = 1×3

        3595          32          16
```

This code snippet shows an example of performing closed loop generation in a model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    % Encode input.
    dlZ = modelEncoder(parameters,dlX,sequenceLengths);

    % Initialize LSTM state.
    state = struct;
    state.HiddenState = dlZ;
    state.CellState = zeros(size(dlZ),'like',dlZ);

    % Initialize decoder input.
```

```
    decoderInput = dlX(:,:,1);

    % Closed loop prediction.
    sequenceLength = size(dlX,3);
    dlY = zeros(numClasses,miniBatchSize,sequenceLength,'like',dlX);
    for t = 1:sequenceLength
        [dlY(:,:,t), state] = modelDecoder(parameters,decoderInput,state);

        [~,idx] = max(dlY(:,:,t));
        decoderInput = idx;
    end

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

Open Loop Generation: Teacher Forcing

When training with closed loop generation, predicting the most likely word for each step in the sequence can lead to suboptimal results. For example, in an image captioning workflow, if the decoder predicts the first word of a caption is "a" when given an image of an elephant, then the probability of predicting "elephant" for the next word becomes much more unlikely because of the extremely low probability of the phrase "a elephant" appearing in English text.

To help the network converge faster, you can use *teacher forcing:* use the target values as input to the decoder instead of the previous predictions. Using teacher forcing helps the network to learn characteristics from the later time steps of the sequences without having to wait for the network to correctly generate the earlier time steps of the sequences.

To perform teacher forcing, use the `modelEncoder` function directly with the target sequence as input.

Initialize the hidden state of the LSTM network with the encoder output `dlZ`.

```
state = struct;
state.HiddenState = dlZ;
state.CellState = zeros(size(dlZ),'like',dlZ);
```

Make predictions using the target sequence as input.

```
dlY = modelDecoder(parameters,dlX,state);
```

The output is a `vocabularySize`-by-`miniBatchSize`-by-`sequenceLength` array, where `sequenceLength` is the length of the input sequences.

```
size(dlY)
```

```
ans = 1×3

        3595          32          14
```

This code snippet shows an example of performing teacher forcing in a model gradients function.

```
function gradients = modelGradients(parameters,dlX,sequenceLengths)

    % Encode input.
    dlZ = modelEncoder(parameters,dlX,dlZ);

    % Initialize LSTM state.
    state = struct;
    state.HiddenState = dlZ;
    state.CellState = zeros(size(dlZ),'like',dlZ);

    % Teacher forcing.
    dlY = modelDecoder(parameters,dlX,state);

    % Calculate loss.
    % ...

    % Calculate gradients.
    % ...

end
```

Decoder Model Function

The `modelDecoder` function, takes as input the model parameters, sequences of word indices, and the network state, and returns the decoded sequences.

Because the `lstm` function is *stateful* (when given a time series as input, the function propagates and updates the state between each time step) and that the `embed` and `fullyconnect` functions are time-distributed by default (when given a time series as input, the functions operate on each time step independently), the `modelDecoder` function supports both sequence and single time-step inputs.

```
function [dlY,state] = modelDecoder(parameters,dlX,state)

% Embedding.
weights = parameters.emb.Weights;
dlX = embed(dlX,weights);

% LSTM.
inputWeights = parameters.lstmDecoder.InputWeights;
recurrentWeights = parameters.lstmDecoder.RecurrentWeights;
bias = parameters.lstmDecoder.Bias;

hiddenState = state.HiddenState;
cellState = state.CellState;

[dlY,hiddenState,cellState] = lstm(dlX,hiddenState,cellState, ...
    inputWeights,recurrentWeights,bias);

state.HiddenState = hiddenState;
state.CellState = cellState;

% Fully connect.
weights = parameters.fcDecoder.Weights;
bias = parameters.fcDecoder.Bias;
dlY = fullyconnect(dlY,weights,bias);
```

```
end
```

See Also

`dlfeval` | `dlgradient` | `dlarray`

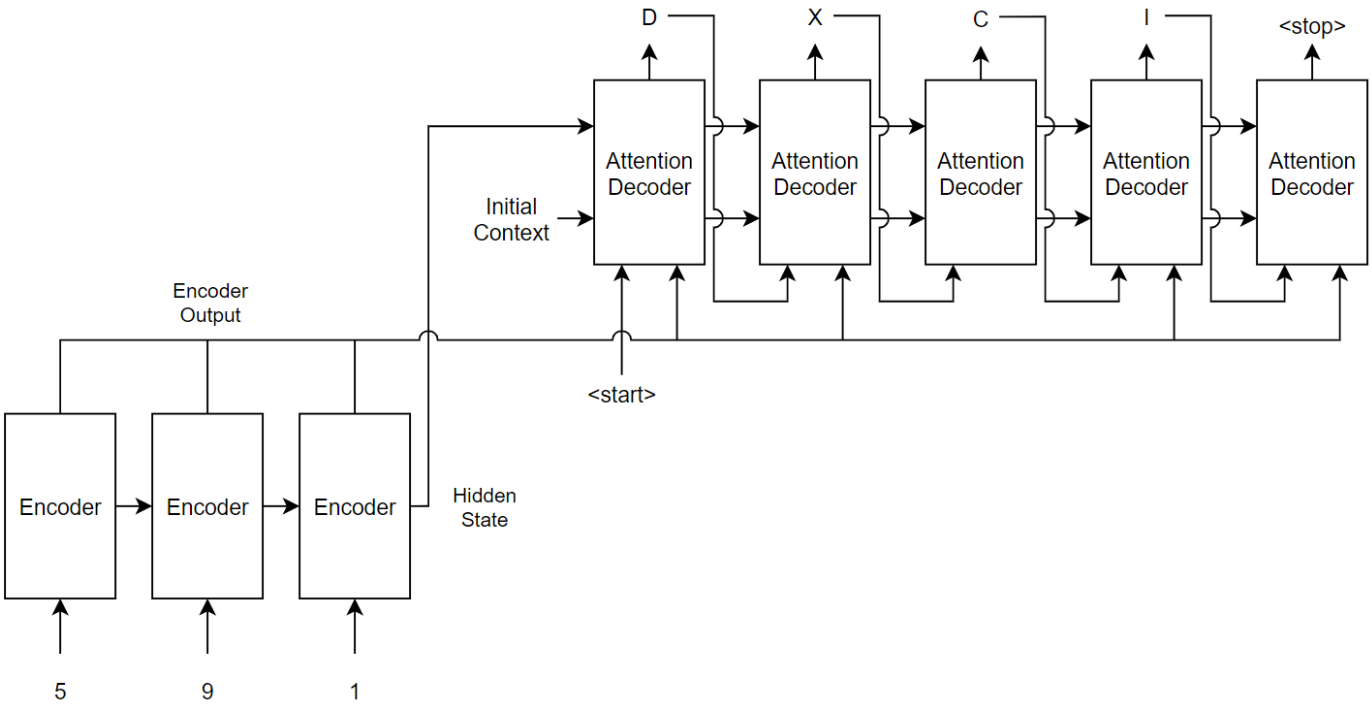
More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Generate Text Using Autoencoders” on page 4-137
- “Define Text Encoder Model Function” on page 4-150

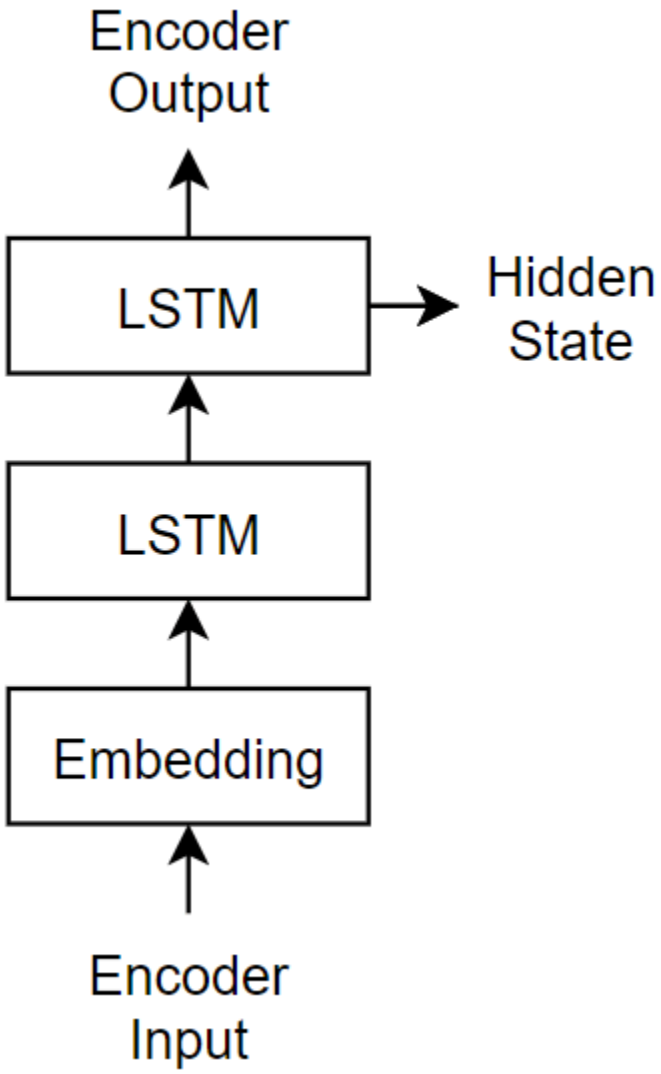
Sequence-to-Sequence Translation Using Attention

This example shows how to convert decimal strings to Roman numerals using a recurrent sequence-to-sequence encoder-decoder model with attention.

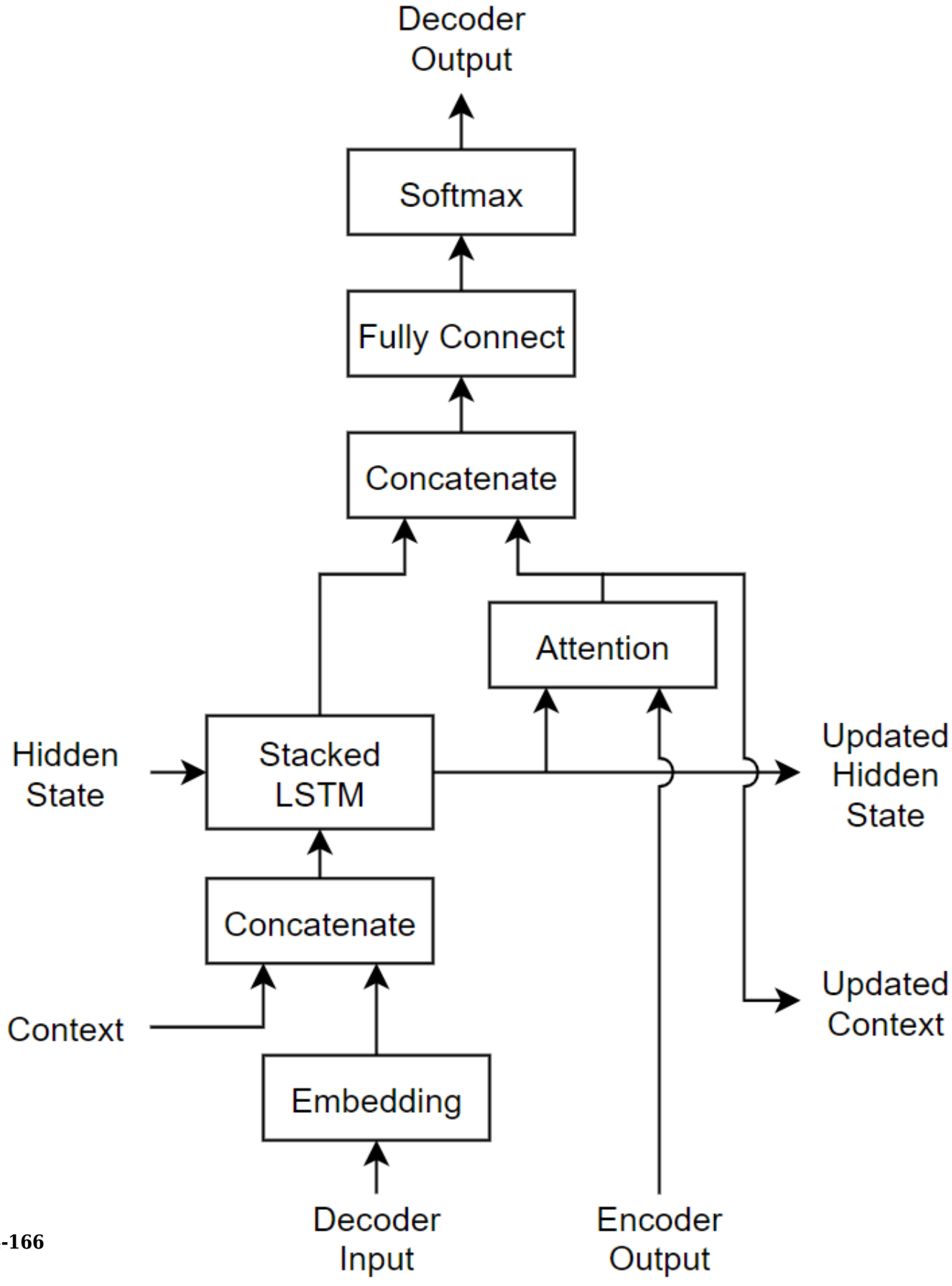
Recurrent encoder-decoder models have proven successful at tasks like abstractive text summarization and neural machine translation. The model consists of an *encoder* which typically processes input data with a recurrent layer such as LSTM, and a *decoder* which maps the encoded input into the desired output, typically with a second recurrent layer. Models that incorporate *attention mechanisms* into the models allows the decoder to focus on parts of the encoded input while generating the translation.



For the encoder model, this example uses a simple network consisting of an embedding followed by two LSTM operations. Embedding is a method of converting categorical tokens into numeric vectors.



For the decoder model, this example uses a network very similar to the encoder that contains two LSTMs. However, an important difference is that the decoder contains an attention mechanism. The attention mechanism allows the decoder to *attend* to specific parts of the encoder output.


Decoder Output
Softmax
Fully Connect
Concatenate
Attention
Hidden State
Stacked LSTM
Updated Hidden State
Concatenate
Context
Updated Context
Embedding
Decoder Input
Encoder Output

Load Training Data

Download the decimal-Roman numeral pairs from "romanNumerals.csv"

```
filename = fullfile("romanNumerals.csv");

options = detectImportOptions(filename, ...
    'TextType','string', ...
    'ReadVariableNames',false);
options.VariableNames = ["Source" "Target"];
options.VariableTypes = ["string" "string"];

data = readtable(filename,options);
```

Split the data into training and test partitions containing 50% of the data each.

```
idx = randperm(size(data,1),500);
dataTrain = data(idx,:);
dataTest = data;
dataTest(idx,:) = [];
```

View some of the decimal-Roman numeral pairs.

```
head(dataTrain)
```

```
ans=8×2 table
    Source     Target
    ______    ________

    "492"     "CDXCII"
    "911"     "CMXI"
    "965"     "CMLXV"
    "951"     "CMLI"
    "160"     "CLX"
    "662"     "DCLXII"
    "102"     "CII"
    "440"     "CDXL"
```

Preprocess Data

Preprocess the text data using the `transformText` function, listed at the end of the example. The `transformText` function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
startToken = "<start>";
stopToken = "<stop>";

strSource = dataTrain.Source;
documentsSource = transformText(strSource,startToken,stopToken);
```

Create a `wordEncoding` object that maps tokens to a numeric index and vice-versa using a vocabulary.

```
encSource = wordEncoding(documentsSource);
```

Using the word encoding, convert the source text data to numeric sequences.

```
sequencesSource = doc2sequence(encSource, documentsSource,'PaddingDirection','none');
```

Convert the target data to sequences using the same steps.

```
strTarget = dataTrain.Target;
documentsTarget = transformText(strTarget,startToken,stopToken);
encTarget = wordEncoding(documentsTarget);
sequencesTarget = doc2sequence(encTarget, documentsTarget,'PaddingDirection','none');
```

Sort the sequences by length. Training with the sequences sorted by increasing sequence length results in batches with sequences of approximately the same sequence length and ensures smaller sequence batches are used to update the model before longer sequence batches.

```
sequenceLengths = cellfun(@(sequence) size(sequence,2),sequencesSource);
[~,idx] = sort(sequenceLengths);
sequencesSource = sequencesSource(idx);
sequencesTarget = sequencesTarget(idx);
```

Create `arrayDatastore` objects containing the source and target data and combine them using the `combine` function.

```
sequencesSourceDs = arrayDatastore(sequencesSource,'OutputType','same');
sequencesTargetDs = arrayDatastore(sequencesTarget,'OutputType','same');

sequencesDs = combine(sequencesSourceDs,sequencesTargetDs);
```

Initialize Model Parameters

Initialize the model parameters. for both the encoder and decoder, specify an embedding dimension of 128, two LSTM layers with 200 hidden units, and dropout layers with random dropout with probability 0.05.

```
embeddingDimension = 128;
numHiddenUnits = 200;
dropout = 0.05;
```

Initialize Encoder Model Parameters

Initialize the weights of the encoding embedding using the Gaussian using the `initializeGaussian` function which is attached to this example as a supporting file. Specify a mean of 0 and a standard deviation of 0.01. To learn more, see “Gaussian Initialization” on page 18-299.

```
inputSize = encSource.NumWords + 1;
sz = [embeddingDimension inputSize];
mu = 0;
sigma = 0.01;
parameters.encoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize the learnable parameters for the encoder LSTM operations:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function which is attached to this example as a supporting file. To learn more, see “Glorot Initialization” on page 18-296.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function which is attached to this example as a supporting file. To learn more, see “Orthogonal Initialization” on page 18-300.

- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function which is attached to this example as a supporting file. To learn more, see “Unit Forget Gate Initialization” on page 18-301.

Initialize the learnable parameters for the first encoder LSTM operation.

```
sz = [4*numHiddenUnits embeddingDimension];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension;

parameters.encoder.lstm1.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.encoder.lstm1.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.encoder.lstm1.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the second encoder LSTM operation.

```
sz = [4*numHiddenUnits numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = numHiddenUnits;

parameters.encoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.encoder.lstm2.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.encoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize Decoder Model Parameters

Initialize the weights of the encoding embedding using the Gaussian using the `initializeGaussian` function. Specify a mean of 0 and a standard deviation of 0.01.

```
outputSize = encTarget.NumWords + 1;
sz = [embeddingDimension outputSize];
mu = 0;
sigma = 0.01;
parameters.decoder.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Initialize the weights of the attention mechanism using the Glorot initializer using the `initializeGlorot` function.

```
sz = [numHiddenUnits numHiddenUnits];
numOut = numHiddenUnits;
numIn = numHiddenUnits;
parameters.decoder.attn.Weights = initializeGlorot(sz,numOut,numIn);
```

Initialize the learnable parameters for the decoder LSTM operations:

- Initialize the input weights with the Glorot initializer using the `initializeGlorot` function.
- Initialize the recurrent weights with the orthogonal initializer using the `initializeOrthogonal` function.
- Initialize the bias with the unit forget gate initializer using the `initializeUnitForgetGate` function.

Initialize the learnable parameters for the first decoder LSTM operation.

```
sz = [4*numHiddenUnits embeddingDimension+numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = embeddingDimension + numHiddenUnits;
```

```
parameters.decoder.lstm1.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm1.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.decoder.lstm1.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the second decoder LSTM operation.

```
sz = [4*numHiddenUnits numHiddenUnits];
numOut = 4*numHiddenUnits;
numIn = numHiddenUnits;

parameters.decoder.lstm2.InputWeights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.lstm2.RecurrentWeights = initializeOrthogonal([4*numHiddenUnits numHiddenUnits
parameters.decoder.lstm2.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the learnable parameters for the decoder fully connected operation:

- Initialize the weights with the Glorot initializer.
- Initialize the bias with zeros using the `initializeZeros` function which is attached to this example as a supporting file. To learn more, see “Zeros Initialization” on page 18-302.

```
sz = [outputSize 2*numHiddenUnits];
numOut = outputSize;
numIn = 2*numHiddenUnits;

parameters.decoder.fc.Weights = initializeGlorot(sz,numOut,numIn);
parameters.decoder.fc.Bias = initializeZeros([outputSize 1]);
```

Define Model Functions

Create the functions `modelEncoder` and `modelDecoder`, listed at the end of the example, that compute the outputs of the encoder and decoder models, respectively.

The `modelEncoder` function, listed in the Encoder Model Function on page 4-0 section of the example, takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model outputs and the LSTM hidden state.

The `modelDecoder` function, listed in the Decoder Model Function on page 4-0 section of the example, takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 4-0 section of the example, that takes the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Specify Training Options

Train with a mini-batch size of 32 for 75 epochs with a learning rate of 0.002.

```
miniBatchSize = 32;
numEpochs = 75;
learnRate = 0.002;
```

Initialize the options from Adam.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train Model

Train the model using a custom training loop. Use `minibatchqueue` to process and manage mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to find the lengths of all sequence in the mini-batch and pad the sequences to the same length as the longest sequence, for the source and target sequences, respectively.
- Permute the second and third dimensions of the padded sequences.
- Return the mini-batch variables unformatted `dlarray` objects with underlying data type `single`. All other outputs are arrays of data type `single`.
- Train on a GPU if one is available. Return all mini-batch variables on the GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see GPU Support by Release.

The `minibatchqueue` object returns four output arguments for each mini-batch: the source sequences, the target sequences, the lengths of all source sequences in the mini-batch, and the sequence mask of the target sequences.

```
numMiniBatchOutputs = 4;

mbq = minibatchqueue(sequencesDs,numMiniBatchOutputs,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn',@(x,t) preprocessMiniBatch(x,t,inputSize,outputSize));
```

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])

xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the values for the `adamupdate` function.

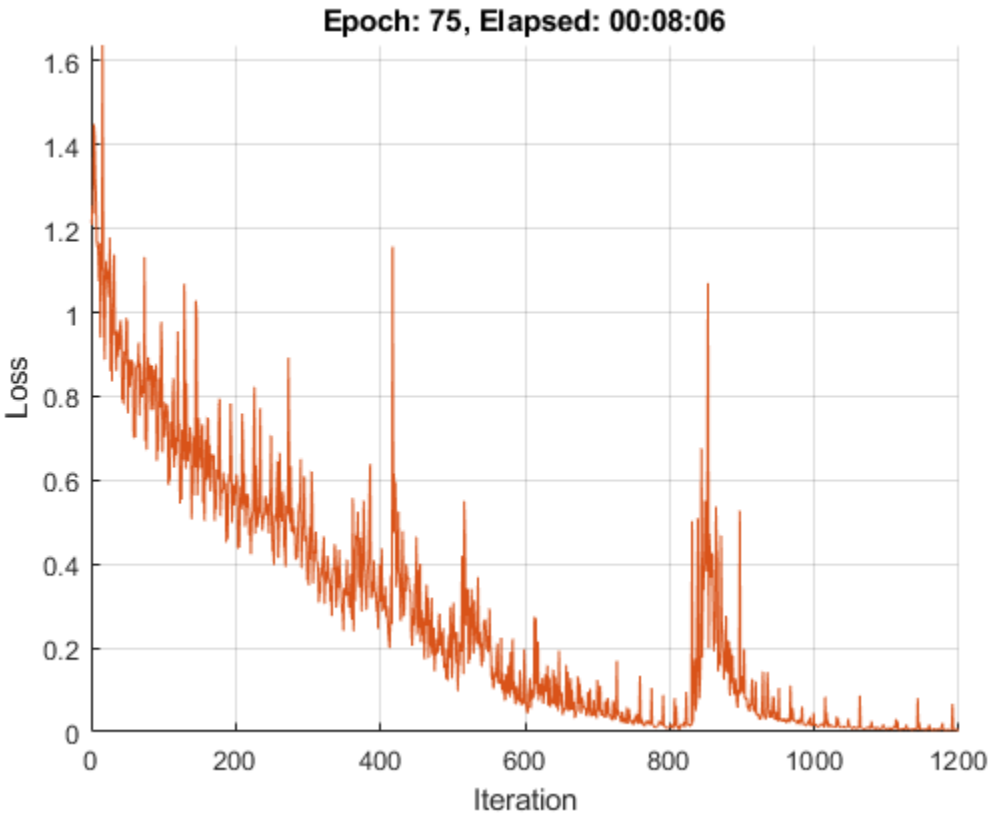
```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model. For each mini-batch:

- Read a mini-batch of padded sequences.
- Compute loss and gradients.
- Update the encoder and decoder model parameters using the `adamupdate` function.
- Update the training progress plot.

```
iteration = 0;
start = tic;
```

```
% Loop over epochs.
for epoch = 1:numEpochs

    reset(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,T,sequenceLengthsSource,maskSequenceTarget] = next(mbq);

        % Compute loss and gradients.
        [gradients,loss] = dlfeval(@modelGradients,parameters,dlX,T,sequenceLengthsSource,...
            maskSequenceTarget,dropout);

        % Update parameters using adamupdate.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients,trailingAvg,tra
            iteration,learnRate,gradientDecayFactor,squaredGradientDecayFactor);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,double(gather(loss)))
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
    end
end
```



Generate Translations

To generate translations for new data using the trained model, convert the text data to numeric sequences using the same steps as when training and input the sequences into the encoder-decoder model and convert the resulting sequences back into text using the token indices.

Preprocess the text data using the same steps as when training. Use the `transformText` function, listed at the end of the example, to split the text into characters and add the start and stop tokens.

```
strSource = dataTest.Source;
strTarget = dataTest.Target;
```

Translate the text using the `modelPredictions` function.

```
maxSequenceLength = 10;
delimiter = "";

strTranslated = translateText(parameters,strSource,maxSequenceLength,miniBatchSize, ...
    encSource,encTarget,startToken,stopToken,delimiter);
```

Create a table containing the test source text, target text, and translations.

```
tbl = table;
tbl.Source = strSource;
tbl.Target = strTarget;
tbl.Translated = strTranslated;
```

View a random selection of the translations.

```
idx = randperm(size(dataTest,1),miniBatchSize);
tbl(idx,:)
```

```
ans=32×3 table
    Source      Target       Translated
    ______    ___________    __________

    "108"     "CVIII"        "CVIIII"
    "651"     "DCLI"         "DCI"
    "147"     "CXLVII"       "CCLXVII"
    "850"     "DCCCL"        "DCCCDCC"
    "468"     "CDLXVIII"     "CCLXVIII"
    "168"     "CLXVIII"      "CDLXVIII"
    "220"     "CCXX"         "CCC"
    "832"     "DCCCXXXII"    "DCCCXXII"
    "162"     "CLXII"        "CDXLII"
    "659"     "DCLIX"        "DCXIX"
    "224"     "CCXXIV"       "CCXXIV"
    "760"     "DCCLX"        "DCCLDC"
    "304"     "CCCIV"        "CCCIV"
    "65"      "LXV"          "DCLV"
    "647"     "DCXLVII"      "DCXLVII"
    "596"     "DXCVI"        "DCXVI"
      ⋮
```

Text Transformation Function

The `transformText` function preprocesses and tokenizes the input text for translation by splitting the text into characters and adding start and stop tokens. To translate text by splitting the text into words instead of characters, skip the first step.

```
function documents = transformText(str,startToken,stopToken)

str = strip(replace(str,"","" ""));
str = startToken + str + stopToken;
documents = tokenizedDocument(str,'CustomTokens',[startToken stopToken]);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function, described in the Train Model section of the example, preprocesses the data for training. The function preprocesses the data using the following steps:

1. Determine the lengths of all source and target sequences in the mini-batch
2. Pad the sequences to the same length as the longest sequence in the mini-batch using the `padsequences` function.
3. Permute the last two dimensions of the sequences

```
function [X,T,sequenceLengthsSource,maskTarget] = preprocessMiniBatch(sequencesSource,sequencesTa

sequenceLengthsSource = cellfun(@(x) size(x,2),sequencesSource);

X = padsequences(sequencesSource,2,"PaddingValue",inputSize);
X = permute(X,[1 3 2]);

[T,maskTarget] = padsequences(sequencesTarget,2,"PaddingValue",outputSize);
T = permute(T,[1 3 2]);
maskTarget = permute(maskTarget,[1 3 2]);

end
```

Model Gradients Function

The `modelGradients` function takes the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

```
function [gradients,loss] = modelGradients(parameters,dlX,T,...
    sequenceLengthsSource,maskTarget,dropout)

% Forward through encoder.
[dlZ,hiddenState] = modelEncoder(parameters.encoder,dlX,sequenceLengthsSource);

% Decoder Output.
doTeacherForcing = rand < 0.5;
sequenceLength = size(T,3);
dlY = decoderPredictions(parameters.decoder,dlZ,T,hiddenState,dropout,...
     doTeacherForcing,sequenceLength);

% Masked loss.
```

```
dlY = dlY(:,:,1:end-1);
T = extractdata(gather(T(:,:,2:end)));
T = onehotencode(T,1,'ClassNames',1:size(dlY,1));

maskTarget = maskTarget(:,:,2:end);
maskTarget = repmat(maskTarget,[size(dlY,1),1,1]);

loss = crossentropy(dlY,T,'Mask',maskTarget,'Dataformat','CBT');

% Update gradients.
gradients = dlgradient(loss,parameters);

% For plotting, return loss normalized by sequence length.
loss = extractdata(loss) ./ sequenceLength;

end
```

Encoder Model Function

The function `modelEncoder` takes the input data, the model parameters, the optional mask that is used to determine the correct outputs for training and returns the model output and the LSTM hidden state.

If `sequenceLengths` is empty, then the function does not mask the output. Specify and empty value for `sequenceLengths` when using the `modelEncoder` function for prediction.

```
function [dlZ, hiddenState] = modelEncoder(parametersEncoder, dlX, sequenceLengths)

% Embedding.
weights = parametersEncoder.emb.Weights;
dlZ = embed(dlX,weights,'DataFormat','CBT');

% LSTM 1.
inputWeights = parametersEncoder.lstm1.InputWeights;
recurrentWeights = parametersEncoder.lstm1.RecurrentWeights;
bias = parametersEncoder.lstm1.Bias;

numHiddenUnits = size(recurrentWeights, 2);
initialHiddenState = dlarray(zeros([numHiddenUnits 1]));
initialCellState = dlarray(zeros([numHiddenUnits 1]));

dlZ = lstm(dlZ, initialHiddenState, initialCellState, inputWeights, ...
    recurrentWeights, bias, 'DataFormat', 'CBT');

% LSTM 2.
inputWeights = parametersEncoder.lstm2.InputWeights;
recurrentWeights = parametersEncoder.lstm2.RecurrentWeights;
bias = parametersEncoder.lstm2.Bias;

[dlZ, hiddenState] = lstm(dlZ,initialHiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Masking for training.
if ~isempty(sequenceLengths)
    miniBatchSize = size(dlZ,2);
    for n = 1:miniBatchSize
        hiddenState(:,n) = dlZ(:,n,sequenceLengths(n));
    end
```

```
end

end
```

Decoder Model Function

The function `modelDecoder` takes the input data, the model parameters, the context vector, the LSTM initial hidden state, the outputs of the encoder, and the dropout probability and outputs the decoder output, the updated context vector, the updated LSTM state, and the attention scores.

```
function [dlY, context, hiddenState, attentionScores] = modelDecoder(parametersDecoder, dlX, con
    hiddenState, dlZ, dropout)

% Embedding.
weights = parametersDecoder.emb.Weights;
dlX = embed(dlX, weights,'DataFormat','CBT');

% RNN input.
sequenceLength = size(dlX,3);
dlY = cat(1, dlX, repmat(context, [1 1 sequenceLength]));

% LSTM 1.
inputWeights = parametersDecoder.lstm1.InputWeights;
recurrentWeights = parametersDecoder.lstm1.RecurrentWeights;
bias = parametersDecoder.lstm1.Bias;

initialCellState = dlarray(zeros(size(hiddenState)));

dlY = lstm(dlY, hiddenState, initialCellState, inputWeights, recurrentWeights, bias, 'DataFormat

% Dropout.
mask = ( rand(size(dlY), 'like', dlY) > dropout );
dlY = dlY.*mask;

% LSTM 2.
inputWeights = parametersDecoder.lstm2.InputWeights;
recurrentWeights = parametersDecoder.lstm2.RecurrentWeights;
bias = parametersDecoder.lstm2.Bias;

[dlY, hiddenState] = lstm(dlY, hiddenState, initialCellState,inputWeights, recurrentWeights, bias

% Attention.
weights = parametersDecoder.attn.Weights;
[attentionScores, context] = attention(hiddenState, dlZ, weights);

% Concatenate.
dlY = cat(1, dlY, repmat(context, [1 1 sequenceLength]));

% Fully connect.
weights = parametersDecoder.fc.Weights;
bias = parametersDecoder.fc.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

% Softmax.
dlY = softmax(dlY,'DataFormat','CBT');

end
```

Attention Function

The `attention` function returns the attention scores according to Luong "general" scoring and the updated context vector. The energy at each time step is the dot product of the hidden state and the learnable attention weights times the encoder output.

```
function [attentionScores, context] = attention(hiddenState, encoderOutputs, weights)

% Initialize attention energies.
[miniBatchSize, sequenceLength] = size(encoderOutputs, 2:3);
attentionEnergies = zeros([sequenceLength miniBatchSize],'like',hiddenState);

% Attention energies.
hWX = hiddenState .* pagemtimes(weights,encoderOutputs);
for tt = 1:sequenceLength
    attentionEnergies(tt, :) = sum(hWX(:, :, tt), 1);
end

% Attention scores.
attentionScores = softmax(attentionEnergies, 'DataFormat', 'CB');

% Context.
encoderOutputs = permute(encoderOutputs, [1 3 2]);
attentionScores = permute(attentionScores,[1 3 2]);
context = pagemtimes(encoderOutputs,attentionScores);
context = squeeze(context);

end
```

Decoder Model Predictions Function

The `decoderModelPredictions` function returns the predicted sequence `dlY` given the input sequence, target sequence, hidden state, dropout probability, flag to enable teacher forcing, and the sequence length.

```
function dlY = decoderPredictions(parametersDecoder,dlZ,T,hiddenState,dropout, ...
    doTeacherForcing,sequenceLength)

% Convert to dlarray.
dlT = dlarray(T);

% Initialize context.
miniBatchSize = size(dlT,2);
numHiddenUnits = size(dlZ,1);
context = zeros([numHiddenUnits miniBatchSize],'like',dlZ);

if doTeacherForcing
    % Forward through decoder.
    dlY = modelDecoder(parametersDecoder, dlT, context, hiddenState, dlZ, dropout);
else
    % Get first time step for decoder.
    decoderInput = dlT(:,:,1);

    % Initialize output.
    numClasses = numel(parametersDecoder.fc.Bias);
    dlY = zeros([numClasses miniBatchSize sequenceLength],'like',decoderInput);

    % Loop over time steps.
```

```
    for t = 1:sequenceLength
        % Forward through decoder.
        [dlY(:,:,t), context, hiddenState] = modelDecoder(parametersDecoder, decoderInput, contex
            hiddenState, dlZ, dropout);

        % Update decoder input.
        [~, decoderInput] = max(dlY(:,:,t),[],1);
    end
end

end
```

Text Translation Function

The `translateText` function translates an array of text by iterating over mini-batches. The function takes as input the model parameters, the input string array, a maximum sequence length, the mini-batch size, the source and target word encoding objects, the start and stop tokens, and the delimiter for assembling the output.

```
function strTranslated = translateText(parameters,strSource,maxSequenceLength,miniBatchSize, ...
    encSource,encTarget,startToken,stopToken,delimiter)

% Transform text.
documentsSource = transformText(strSource,startToken,stopToken);
sequencesSource = doc2sequence(encSource,documentsSource, ...
    'PaddingDirection','right', ...
    'PaddingValue',encSource.NumWords + 1);

% Convert to dlarray.
X = cat(3,sequencesSource{:});
X = permute(X,[1 3 2]);
dlX = dlarray(X);

% Initialize output.
numObservations = numel(strSource);
strTranslated = strings(numObservations,1);

% Loop over mini-batches.
numIterations = ceil(numObservations / miniBatchSize);
for i = 1:numIterations
    idxMiniBatch = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);
    miniBatchSize = numel(idxMiniBatch);

    % Encode using model encoder.
    sequenceLengths = [];
    [dlZ, hiddenState] = modelEncoder(parameters.encoder, dlX(:,idxMiniBatch,:), sequenceLengths

    % Decoder predictions.
    doTeacherForcing = false;
    dropout = 0;
    decoderInput = repmat(word2ind(encTarget,startToken),[1 miniBatchSize]);
    decoderInput = dlarray(decoderInput);
    dlY = decoderPredictions(parameters.decoder,dlZ,decoderInput,hiddenState,dropout, ...
        doTeacherForcing,maxSequenceLength);
    [~, idxPred] = max(extractdata(dlY), [], 1);

    % Keep translating flag.
    idxStop = word2ind(encTarget,stopToken);
```

```
    keepTranslating = idxPred ~= idxStop;

    % Loop over time steps.
    t = 1;
    while t <= maxSequenceLength && any(keepTranslating(:,:,t))

        % Update output.
        newWords = ind2word(encTarget, idxPred(:,:,t))';
        idxUpdate = idxMiniBatch(keepTranslating(:,:,t));
        strTranslated(idxUpdate) = strTranslated(idxUpdate) + delimiter + newWords(keepTranslati

        t = t + 1;
    end
end

end
```

See Also

word2ind | tokenizedDocument | wordEncoding | dlarray | adamupdate | dlupdate | dlfeval | dlgradient | crossentropy | softmax | lstm | doc2sequence

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Multilabel Text Classification Using Deep Learning” on page 4-106
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Automatic Differentiation Background” on page 18-200

Generate Text Using Deep Learning

This example shows how to train a deep learning long short-term memory (LSTM) network to generate text.

To train a deep learning network for text generation, train a sequence-to-sequence LSTM network to predict the next character in a sequence of characters. To train the network to predict the next character, specify the input sequences shifted by one time step as the responses.

To input a sequence of characters into an LSTM network, convert each training observation to a sequence of characters represented by the vectors $x \in \mathbb{R}^D$, where D is the number of unique characters in the vocabulary. For each vector, $x_i = 1$ if x corresponds to the character with index i in a given vocabulary, and $x_j = 0$ for $j \neq i$.

Load Training Data

Extract the text data from the text file `sonnets.txt`.

```
filename = "sonnets.txt";
textData = fileread(filename);
```

The sonnets are indented by two whitespace characters and are separated by two newline characters. Remove the indentations using `replace` and split the text into separate sonnets using `split`. Remove the main title from the first three elements and the sonnet titles which appear before each sonnet.

```
textData = replace(textData,"  ","");
textData = split(textData,[newline newline]);
textData = textData(5:2:end);
```

View the first few observations.

```
textData(1:10)
```

```
ans = 10×1 cell array
    {'From fairest creatures we desire increase,↵That thereby beauty's rose might never die,↵But
    {'When forty winters shall besiege thy brow,↵And dig deep trenches in thy beauty's field,↵Thy
    {'Look in thy glass and tell the face thou viewest↵Now is the time that face should form anot
    {'Unthrifty loveliness, why dost thou spend↵Upon thy self thy beauty's legacy?↵Nature's beque
    {'Those hours, that with gentle work did frame↵The lovely gaze where every eye doth dwell,↵Wi
    {'Then let not winter's ragged hand deface,↵In thee thy summer, ere thou be distill'd:↵Make s
    {'Lo! in the orient when the gracious light↵Lifts up his burning head, each under eye↵Doth ho
    {'Music to hear, why hear'st thou music sadly?↵Sweets with sweets war not, joy delights in jo
    {'Is it for fear to wet a widow's eye,↵That thou consum'st thy self in single life?↵Ah! if th
    {'For shame! deny that thou bear'st love to any,↵Who for thy self art so unprovident.↵Grant,
```

Convert Text Data to Sequences

Convert the text data to sequences of vectors for the predictors and categorical sequences for the responses.

Create special characters to denote "start of text", "whitespace", "end of text" and "newline". Use the special characters `"\x0002"` (start of text), `"\x00B7"` ("·", middle dot), `"\x2403"` ("ETX", end of text), and `"\x00B6"` ("¶", pilcrow) respectively. To prevent ambiguity, you must choose special characters

that do not appear in the text. Because these characters do not appear in the training data, they can be used for this purpose.

```
startOfTextCharacter = compose("\x0002");
whitespaceCharacter = compose("\x00B7");
endOfTextCharacter = compose("\x2403");
newlineCharacter = compose("\x00B6");
```

For each observation, insert the start of text character at the beginning and replace the whitespace and newlines with the corresponding characters.

```
textData = startOfTextCharacter + textData;
textData = replace(textData,[" " newline],[whitespaceCharacter newlineCharacter]);
```

Create a vocabulary of the unique characters in the text.

```
uniqueCharacters = unique([textData{:}]);
numUniqueCharacters = numel(uniqueCharacters);
```

Loop over the text data and create a sequence of vectors representing the characters of each observation and a categorical sequence of characters for the responses. To denote the end of each observation, include the end of text character.

```
numDocuments = numel(textData);
XTrain = cell(1,numDocuments);
YTrain = cell(1,numDocuments);
for i = 1:numel(textData)
    characters = textData{i};
    sequenceLength = numel(characters);

    % Get indices of characters.
    [~,idx] = ismember(characters,uniqueCharacters);

    % Convert characters to vectors.
    X = zeros(numUniqueCharacters,sequenceLength);
    for j = 1:sequenceLength
        X(idx(j),j) = 1;
    end

    % Create vector of categorical responses with end of text character.
    charactersShifted = [cellstr(characters(2:end)')' endOfTextCharacter];
    Y = categorical(charactersShifted);

    XTrain{i} = X;
    YTrain{i} = Y;
end
```

View the first observation and the size of the corresponding sequence. The sequence is a *D*-by-*S* matrix, where *D* is the number of features (the number of unique characters) and *S* is the sequence length (the number of characters in the text).

```
textData{1}
```

```
ans =
'From·fairest·creatures·we·desire·increase,¶That·thereby·beauty's·rose·might·never·die,¶But·as·t
```

```
size(XTrain{1})
```

```
ans = 1×2

    62   611
```

View the corresponding response sequence. The sequence is a 1-by-*S* categorical vector of responses.

```
YTrain{1}
```

```
ans = 1×611 categorical array
     F      r      o      m      ·      f      a      i      r      e      s      t      ·      c
```

Create and Train LSTM Network

Define the LSTM architecture. Specify a sequence-to-sequence LSTM classification network with 200 hidden units. Set the feature dimension of the training data (the number of unique characters) as the input size, and the number of categories in the responses as the output size of the fully connected layer.

```
inputSize = size(XTrain{1},1);
numHiddenUnits = 200;
numClasses = numel(categories([YTrain{:}]));

layers = [
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

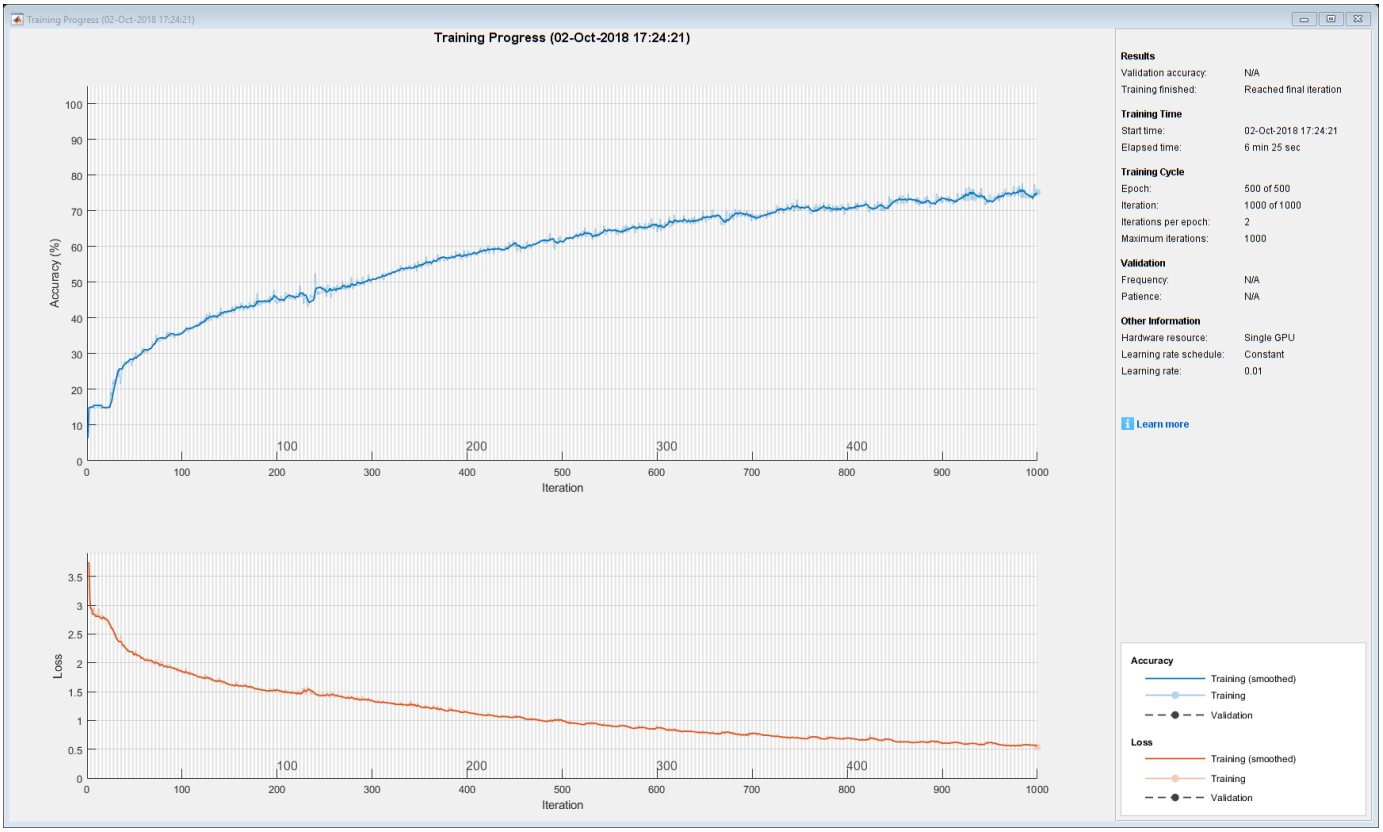
Specify the training options using the `trainingOptions` function. Specify the number of training epochs as 500 and the initial learn rate as 0.01. To prevent the gradients from exploding, set the gradient threshold to 2. Specify to shuffle the data every epoch by setting the `'Shuffle'` option to `'every-epoch'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

The mini-batch size option specifies the number of observations to process in a single iteration. Specify a mini-batch size that evenly divides the data to ensure that the function uses all observations for training. Otherwise, the function ignores observations that do not complete a mini-batch. Set the mini-batch size to 77.

```
options = trainingOptions('adam', ...
    'MaxEpochs',500, ...
    'InitialLearnRate',0.01, ...
    'GradientThreshold',2, ...
    'MiniBatchSize',77,...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Generate New Text

Use the `generateText` function, listed at the end of the example, to generate text using the trained network.

The `generateText` function generates text character by character, starting with the start of text character and reconstructs the text using the special characters. The function samples each character using the output prediction scores. The function stops predicting when the network predicts the end-of-text character or when the generated text is 500 characters long.

Generate text using the trained network.

```
generatedText = generateText(net,uniqueCharacters,startOfTextCharacter,newlineCharacter,whitespac
```

```
generatedText =
    "Look, that your lepperites of such soous toor men,
     Where than proud on your sweetest but lever ill lie.
     One of Death a deal doth teal hearts come,
     And that which gives did mistress one learn
     Made mens of tongue that hands hear,
     And all they with me, do I fortune to brief;
     And every peinted could with this right ampontion sorend
     By genilir'd lime thau hours, and wonder sposing,
     And night by day you waster'd then new;
     For ailling thuse borrowest vein fulse were of here spent,
     Since my heart morey "
```

Text Generation Function

The `generateText` function generates text character by character, starting with the start of text character and reconstructs the text using the special characters. The function samples each character using the output prediction scores. The function stops predicting when the network predicts the end-of-text character or when the generated text is 500 characters long.

```
function generatedText = generateText(net,uniqueCharacters,startOfTextCharacter,newlineCharacter
```

Create the vector of the start of text character by finding its index.

```
numUniqueCharacters = numel(uniqueCharacters);
X = zeros(numUniqueCharacters,1);
idx = strfind(uniqueCharacters,startOfTextCharacter);
X(idx) = 1;
```

Generate the text character by character using the trained LSTM network using `predictAndUpdateState` and `datasample`. Stop predicting when the network predicts the end-of-text character or when the generated text is 500 characters long. The `datasample` function requires Statistics and Machine Learning Toolbox™.

For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
generatedText = "";
vocabulary = string(net.Layers(end).Classes);

maxLength = 500;
while strlength(generatedText) < maxLength
    % Predict the next character scores.
    [net,characterScores] = predictAndUpdateState(net,X,'ExecutionEnvironment','cpu');

    % Sample the next character.
    newCharacter = datasample(vocabulary,1,'Weights',characterScores);

    % Stop predicting at the end of text.
    if newCharacter == endOfTextCharacter
        break
    end

    % Add the character to the generated text.
    generatedText = generatedText + newCharacter;

    % Create a new vector for the next input.
    X(:) = 0;
    idx = strfind(uniqueCharacters,newCharacter);
    X(idx) = 1;
end
```

Reconstruct the generated text by replacing the special characters with their corresponding whitespace and newline characters.

```
generatedText = replace(generatedText,[newlineCharacter whitespaceCharacter],[newline " "]);

end
```

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer`

Related Examples

- “Word-By-Word Text Generation Using Deep Learning” (Text Analytics Toolbox)
- “Pride and Prejudice and MATLAB” (Text Analytics Toolbox)
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Pride and Prejudice and MATLAB

This example shows how to train a deep learning LSTM network to generate text using character embeddings.

To train a deep learning network for text generation, train a sequence-to-sequence LSTM network to predict the next character in a sequence of characters. To train the network to predict the next character, specify the responses to be the input sequences shifted by one time step.

To use character embeddings, convert each training observation to a sequence of integers, where the integers index into a vocabulary of characters. Include a word embedding layer in the network which learns an embedding of the characters and maps the integers to vectors.

Load Training Data

Read the HTML code from The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen and parse it using `webread` and `htmlTree`.

```
url = "https://www.gutenberg.org/files/1342/1342-h/1342-h.htm";
code = webread(url);
tree = htmlTree(code);
```

Extract the paragraphs by finding the `p` elements. Specify to ignore paragraph elements with class `"toc"` using the CSS selector `':not(.toc)'`.

```
paragraphs = findElement(tree,'p:not(.toc)');
```

Extract the text data from the paragraphs using `extractHTMLText`. and remove the empty strings.

```
textData = extractHTMLText(paragraphs);
textData(textData == "") = [];
```

Remove strings shorter than 20 characters.

```
idx = strlength(textData) < 20;
textData(idx) = [];
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
title("Pride and Prejudice")
```

Convert Text Data to Sequences

Convert the text data to sequences of character indices for the predictors and categorical sequences for the responses.

The categorical function treats newline and whitespace entries as undefined. To create categorical elements for these characters, replace them with the special characters "¶" (pilcrow, `"\x00B6"`) and "·" (middle dot, `"\x00B7"`) respectively. To prevent ambiguity, you must choose special characters that do not appear in the text. These characters do not appear in the training data so can be used for this purpose.

```
newlineCharacter = compose("\x00B6");
whitespaceCharacter = compose("\x00B7");
textData = replace(textData,[newline " "],[newlineCharacter whitespaceCharacter]);
```

Loop over the text data and create a sequence of character indices representing the characters of each observation and a categorical sequence of characters for the responses. To denote the end of each observation, include the special character "ETX" (end of text, `"\x2403"`).

```
endOfTextCharacter = compose("\x2403");
numDocuments = numel(textData);
for i = 1:numDocuments
    characters = textData{i};
    X = double(characters);

    % Create vector of categorical responses with end of text character.
    charactersShifted = [cellstr(characters(2:end)')' endOfTextCharacter];
```

```
    Y = categorical(charactersShifted);

    XTrain{i} = X;
    YTrain{i} = Y;
end
```

During training, by default, the software splits the training data into mini-batches and pads the sequences so that they have the same length. Too much padding can have a negative impact on the network performance.

To prevent the training process from adding too much padding, you can sort the training data by sequence length, and choose a mini-batch size so that sequences in a mini-batch have a similar length.

Get the sequence lengths for each observation.

```
numObservations = numel(XTrain);
for i=1:numObservations
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end
```

Sort the data by sequence length.

```
[~,idx] = sort(sequenceLengths);
XTrain = XTrain(idx);
YTrain = YTrain(idx);
```

Create and Train LSTM Network

Define the LSTM architecture. Specify a sequence-to-sequence LSTM classification network with 400 hidden units. Set the input size to be the feature dimension of the training data. For sequences of character indices, the feature dimension is 1. Specify a word embedding layer with dimension 200 and specify the number of words (which correspond to characters) to be the highest character value in the input data. Set the output size of the fully connected layer to be the number of categories in the responses. To help prevent overfitting, include a dropout layer after the LSTM layer.

The word embedding layer learns an embedding of characters and maps each character to a 200-dimension vector.

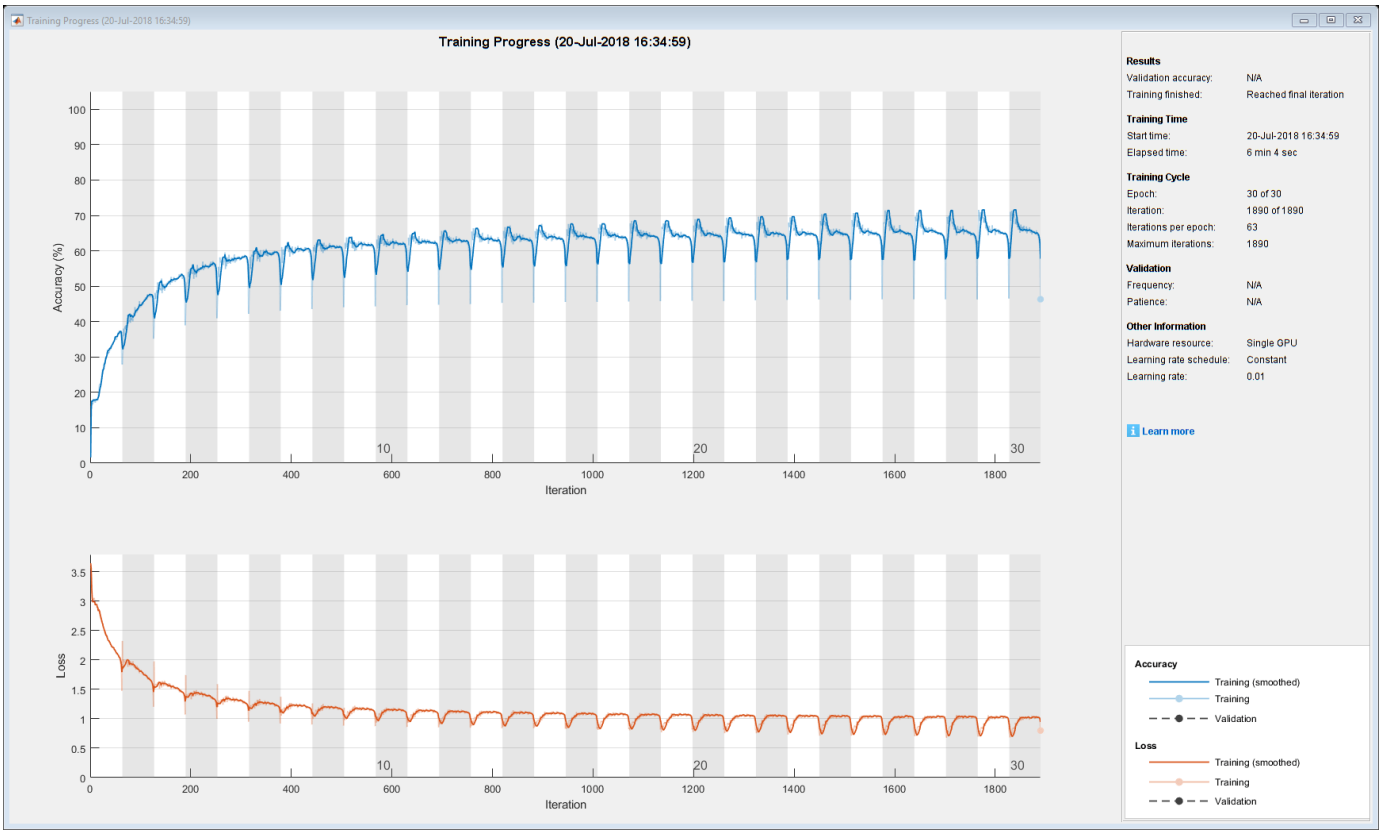
```
inputSize = size(XTrain{1},1);
numClasses = numel(categories([YTrain{:}]));
numCharacters = max([textData{:}]);

layers = [
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(200,numCharacters)
    lstmLayer(400,'OutputMode','sequence')
    dropoutLayer(0.2);
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Specify to train with a mini-batch size of 32 and initial learn rate 0.01. To prevent the gradients from exploding, set the gradient threshold to 1. To ensure the data remains sorted, set `'Shuffle'` to `'never'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

```
options = trainingOptions('adam', ...
    'MiniBatchSize',32,...
    'InitialLearnRate',0.01, ...
    'GradientThreshold',1, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



Generate New Text

Generate the first character of the text by sampling a character from a probability distribution according to the first characters of the text in the training data. Generate the remaining characters by using the trained LSTM network to predict the next sequence using the current sequence of generated text. Keep generating characters one-by-one until the network predicts the "end of text" character.

Sample the first character according to the distribution of the first characters in the training data.

```
initialCharacters = extractBefore(textData,2);
firstCharacter = datasample(initialCharacters,1);
generatedText = firstCharacter;
```

Convert the first character to a numeric index.

```
X = double(char(firstCharacter));
```

For the remaining predictions, sample the next character according to the prediction scores of the network. The prediction scores represent the probability distribution of the next character. Sample the characters from the vocabulary of characters given by the class names of the output layer of the network. Get the vocabulary from the classification layer of the network.

```
vocabulary = string(net.Layers(end).ClassNames);
```

Make predictions character by character using `predictAndUpdateState`. For each prediction, input the index of the previous character. Stop predicting when the network predicts the end of text character or when the generated text is 500 characters long. For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
maxLength = 500;
while strlength(generatedText) < maxLength
    % Predict the next character scores.
    [net,characterScores] = predictAndUpdateState(net,X,'ExecutionEnvironment','cpu');

    % Sample the next character.
    newCharacter = datasample(vocabulary,1,'Weights',characterScores);

    % Stop predicting at the end of text.
    if newCharacter == endOfTextCharacter
        break
    end

    % Add the character to the generated text.
    generatedText = generatedText + newCharacter;

    % Get the numeric index of the character.
    X = double(char(newCharacter));
end
```

Reconstruct the generated text by replacing the special characters with their corresponding whitespace and new line characters.

```
generatedText = replace(generatedText,[newlineCharacter whitespaceCharacter],[newline " "])
```

```
generatedText =
""“I wish Mr. Darcy, upon latter of my sort sincerely fixed in the regard to relanth. We were to
```

To generate multiple pieces of text, reset the network state between generations using `resetState`.

```
net = resetState(net);
```

See Also

`wordEmbeddingLayer` | `doc2sequence` | `tokenizedDocument` | `lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer` | `wordcloud` | `extractHTMLText` | `findElement` | `htmlTree`

Related Examples

- “Generate Text Using Deep Learning” on page 4-180

- “Word-By-Word Text Generation Using Deep Learning” (Text Analytics Toolbox)
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Word-By-Word Text Generation Using Deep Learning

This example shows how to train a deep learning LSTM network to generate text word-by-word.

To train a deep learning network for word-by-word text generation, train a sequence-to-sequence LSTM network to predict the next word in a sequence of words. To train the network to predict the next word, specify the responses to be the input sequences shifted by one time step.

This example reads text from a website. It reads and parses the HTML code to extract the relevant text, then uses a custom mini-batch datastore `documentGenerationDatastore` to input the documents to the network as mini-batches of sequence data. The datastore converts documents to sequences of numeric word indices. The deep learning network is an LSTM network that contains a word embedding layer.

A mini-batch datastore is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use mini-batch datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

You can adapt the custom mini-batch datastore specified by `documentGenerationDatastore.m` to your data by customizing the functions. This file is eattached to this example as a supporting file. To access this file, open the example as a live script. For an example showing how to create your own custom mini-batch datastore, see “Develop Custom Mini-Batch Datastore” on page 19-36.

Load Training Data

Load the training data. Read the HTML code from Alice's Adventures in Wonderland by Lewis Carroll from Project Gutenberg.

```
url = "https://www.gutenberg.org/files/11/11-h/11-h.htm";
code = webread(url);
```

Parse HTML Code

The HTML code contains the relevant text inside <p> (paragraph) elements. Extract the relevant text by parsing the HTML code using `htmlTree` and then finding all the elements with element name `"p"`.

```
tree = htmlTree(code);
selector = "p";
subtrees = findElement(tree,selector);
```

Extract the text data from the HTML subtrees using `extractHTMLText` and view the first 10 paragraphs.

```
textData = extractHTMLText(subtrees);
textData(1:10)
```

```
ans = 10×1 string
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
    "In another moment down went Alice after it, never once considering how in the world she was
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down,
    "Either the well was very deep, or she fell very slowly, for she had plenty of time as she we
    "“Well!” thought Alice to herself, “after such a fall as this, I shall think nothing of tumbl
```

```
    "Down, down, down. Would the fall never come to an end? “I wonder how many miles I’ve fallen
    "Presently she began again. “I wonder if I shall fall right through the earth! How funny it’l
    "Down, down, down. There was nothing else to do, so Alice soon began talking again. “Dinah’l
```

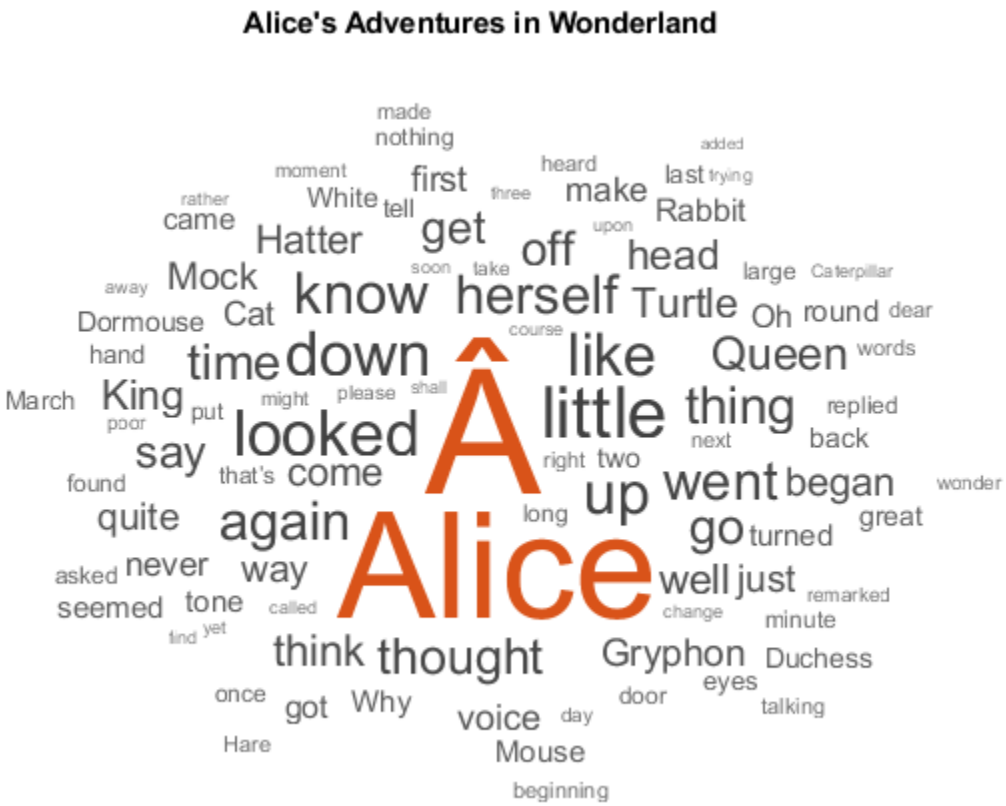
Remove the empty paragraphs and view the first 10 remaining paragraphs.

```
textData(textData == "") = [];
textData(1:10)
```

```
ans = 10×1 string
    "Alice was beginning to get very tired of sitting by her sister on the bank, and of having no
    "So she was considering in her own mind (as well as she could, for the hot day made her feel
    "There was nothing so very remarkable in that; nor did Alice think it so very much out of the
    "In another moment down went Alice after it, never once considering how in the world she was
    "The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down,
    "Either the well was very deep, or she fell very slowly, for she had plenty of time as she we
    "“Well!” thought Alice to herself, “after such a fall as this, I shall think nothing of tumbl
    "Down, down, down. Would the fall never come to an end? “I wonder how many miles I’ve fallen
    "Presently she began again. “I wonder if I shall fall right through the earth! How funny it’l
    "Down, down, down. There was nothing else to do, so Alice soon began talking again. “Dinah’l
```

Visualize the text data in a word cloud.

```
figure
wordcloud(textData);
title("Alice's Adventures in Wonderland")
```



Prepare Data for Training

Create a datastore that contains the data for training using `documentGenerationDatastore`. For the predictors, this datastore converts the documents into sequences of word indices using a word encoding. The first word index for each document corresponds to a "start of text" token. The "start of text" token is given by the string `"startOfText"`. For the responses, the datastore returns categorical sequences of the words shifted by one.

Tokenize the text data using `tokenizedDocument`.

```
documents = tokenizedDocument(textData);
```

Create a document generation datastore using the tokenized documents.

```
ds = documentGenerationDatastore(documents);
```

To reduce the amount of padding added to the sequences, sort the documents in the datastore by sequence length.

```
ds = sort(ds);
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to 1. Next, include a word embedding layer of dimension 100 and the same number of words as the word encoding. Next, include an LSTM layer and specify the hidden size to be 100. Finally, add a fully connected layer with the same size as the number of classes, a softmax layer, and a classification layer. The number of classes is the number of words in the vocabulary plus an extra class for the "end of text" class.

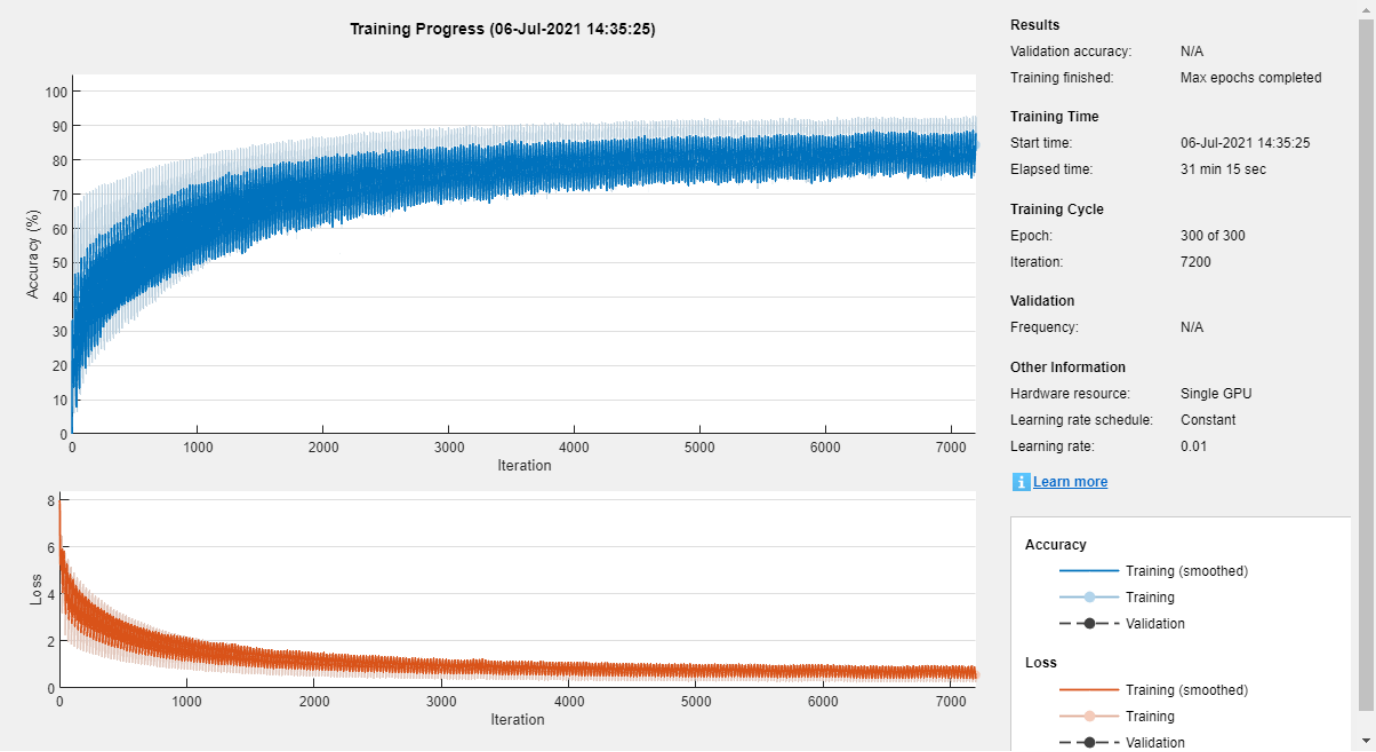
```
inputSize = 1;
embeddingDimension = 100;
numWords = numel(ds.Encoding.Vocabulary);
numClasses = numWords + 1;

layers = [
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension,numWords)
    lstmLayer(100)
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Specify the solver to be `'adam'`. Train for 300 epochs with learn rate 0.01. Set the mini-batch size to 32. To keep the data sorted by sequence length, set the `'Shuffle'` option to `'never'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

```
options = trainingOptions('adam', ...
    'MaxEpochs',300, ...
    'InitialLearnRate',0.01, ...
    'MiniBatchSize',32, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using `trainNetwork`.

```
net = trainNetwork(ds,layers,options);
```



Generate New Text

Generate the first word of the text by sampling a word from a probability distribution according to the first words of the text in the training data. Generate the remaining words by using the trained LSTM network to predict the next time step using the current sequence of generated text. Keep generating words one-by-one until the network predicts the "end of text" word.

To make the first prediction using the network, input the index that represents the "start of text" token. Find the index by using the `word2ind` function with the word encoding used by the document datastore.

```
enc = ds.Encoding;
wordIndex = word2ind(enc,"startOfText")
```

```
wordIndex = 1
```

For the remaining predictions, sample the next word according to the prediction scores of the network. The prediction scores represent the probability distribution of the next word. Sample the words from the vocabulary given by the class names of the output layer of the network.

```
vocabulary = string(net.Layers(end).Classes);
```

Make predictions word by word using `predictAndUpdateState`. For each prediction, input the index of the previous word. Stop predicting when the network predicts the end of text word or when the generated text is 500 characters long. For large collections of data, long sequences, or large networks, predictions on the GPU are usually faster to compute than predictions on the CPU. Otherwise, predictions on the CPU are usually faster to compute. For single time step predictions, use

the CPU. To use the CPU for prediction, set the `'ExecutionEnvironment'` option of `predictAndUpdateState` to `'cpu'`.

```
generatedText = "";
maxLength = 500;
while strlength(generatedText) < maxLength
    % Predict the next word scores.
    [net,wordScores] = predictAndUpdateState(net,wordIndex,'ExecutionEnvironment','cpu');

    % Sample the next word.
    newWord = datasample(vocabulary,1,'Weights',wordScores);

    % Stop predicting at the end of text.
    if newWord == "EndOfText"
        break
    end

    % Add the word to the generated text.
    generatedText = generatedText + " " + newWord;

    % Find the word index for the next input.
    wordIndex = word2ind(enc,newWord);
end
```

The generation process introduces whitespace characters between each prediction, which means that some punctuation characters appear with unnecessary spaces before and after. Reconstruct the generated text by removing the spaces before and after the appropriate punctuation characters.

Remove the spaces that appear before the specified punctuation characters.

```
punctuationCharacters = ["." "," "’" ")" ":" "?" "!"];
generatedText = replace(generatedText," " + punctuationCharacters,punctuationCharacters);
```

Remove the spaces that appear after the specified punctuation characters.

```
punctuationCharacters = ["(" "‘"];
generatedText = replace(generatedText,punctuationCharacters + " ",punctuationCharacters)
```

```
generatedText =
" “ Just about as much right, ” said the Duchess, “ and that’s all the least, ” said the Hatter.
```

To generate multiple pieces of text, reset the network state between generations using `resetState`.

```
net = resetState(net);
```

See Also

`wordEmbeddingLayer` | `doc2sequence` | `tokenizedDocument` | `lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer` | `wordcloud` | `extractHTMLText` | `findElement` | `htmlTree`

Related Examples

- “Generate Text Using Deep Learning” on page 4-180
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)

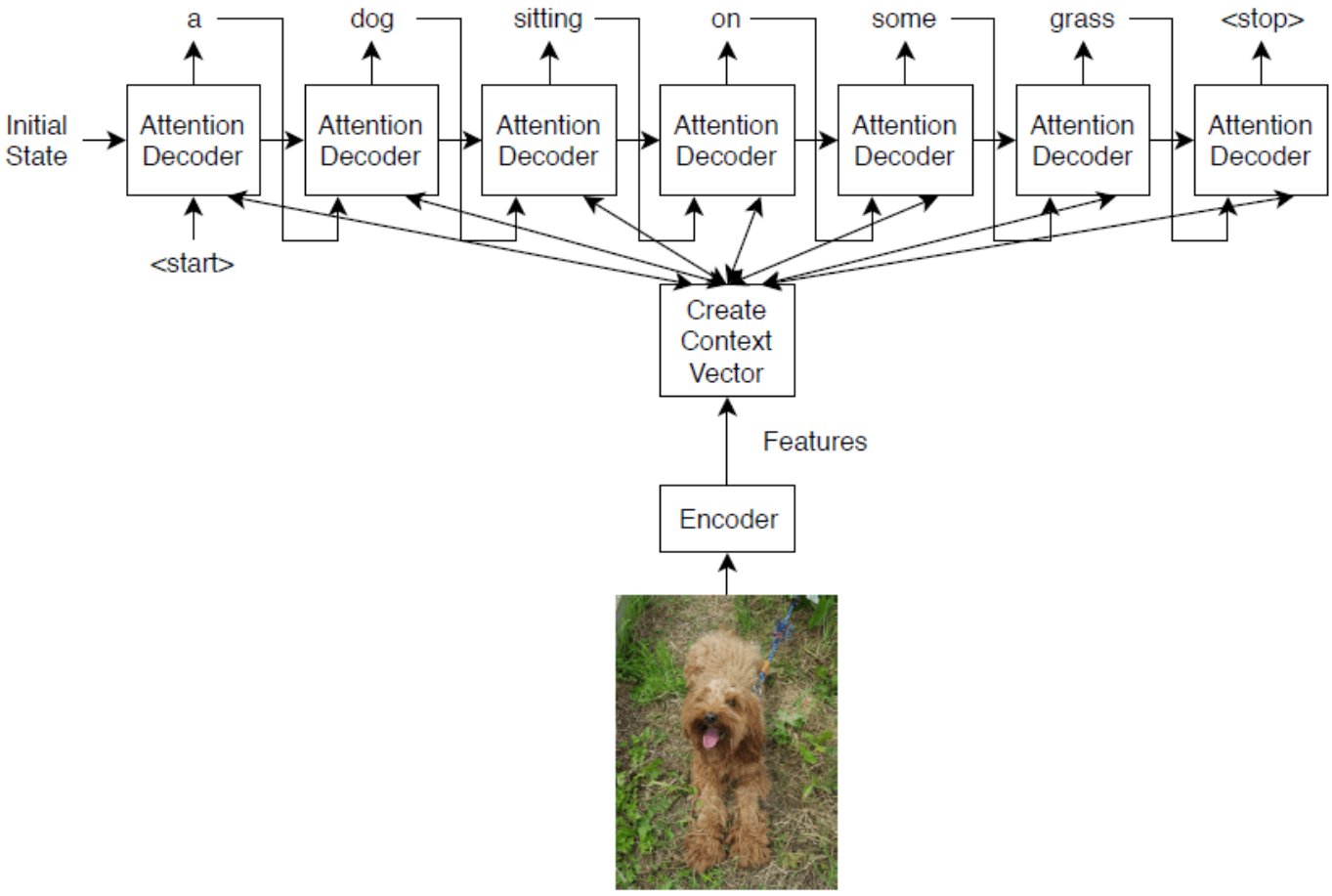
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Image Captioning Using Attention

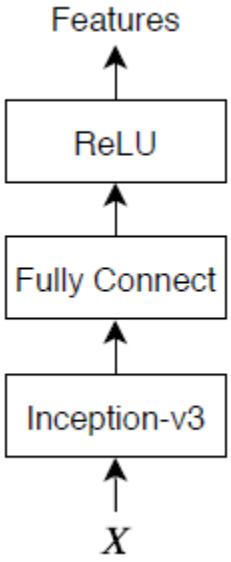
This example shows how to train a deep learning model for image captioning using attention.

Most pretrained deep learning networks are configured for single-label classification. For example, given an image of a typical office desk, the network might predict the single class "keyboard" or "mouse". In contrast, an image captioning model combines convolutional and recurrent operations to produce a textual description of what is in the image, rather than a single label.

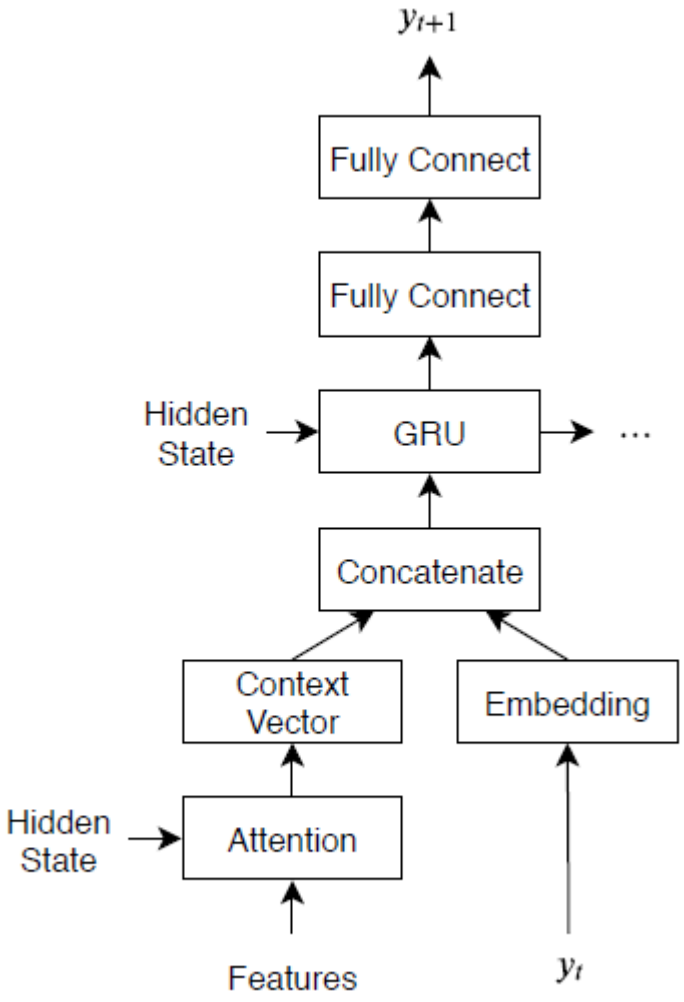
This model trained in this example uses an encoder-decoder architecture. The encoder is a pretrained Inception-v3 network used as a feature extractor. The decoder is a recurrent neural network (RNN) that takes the extracted features as input and generates a caption. The decoder incorporates an *attention mechanism* that allows the decoder to focus on parts of the encoded input while generating the caption.



The encoder model is a pretrained Inception-v3 model that extracts features from the `"mixed10"` layer, followed by fully connected and ReLU operations.



The decoder model consists of a word embedding, an attention mechanism, a gated recurrent unit (GRU), and two fully connected operations.



Load Pretrained Network

Load a pretrained Inception-v3 network. This step requires the Deep Learning Toolbox™ Model *for Inception-v3 Network* support package. If you do not have the required support package installed, then the software provides a download link.

```
net = inceptionv3;
inputSizeNet = net.Layers(1).InputSize;
```

Convert the network to a `dlnetwork` object for feature extraction and remove the last four layers, leaving the `"mixed10"` layer as the last layer.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,["avg_pool" "predictions" "predictions_softmax" "ClassificationLaye
```

View the input layer of the network. The Inception-v3 network uses symmetric-rescale normalization with a minimum value of 0 and a maximum value of 255.

```
lgraph.Layers(1)
```

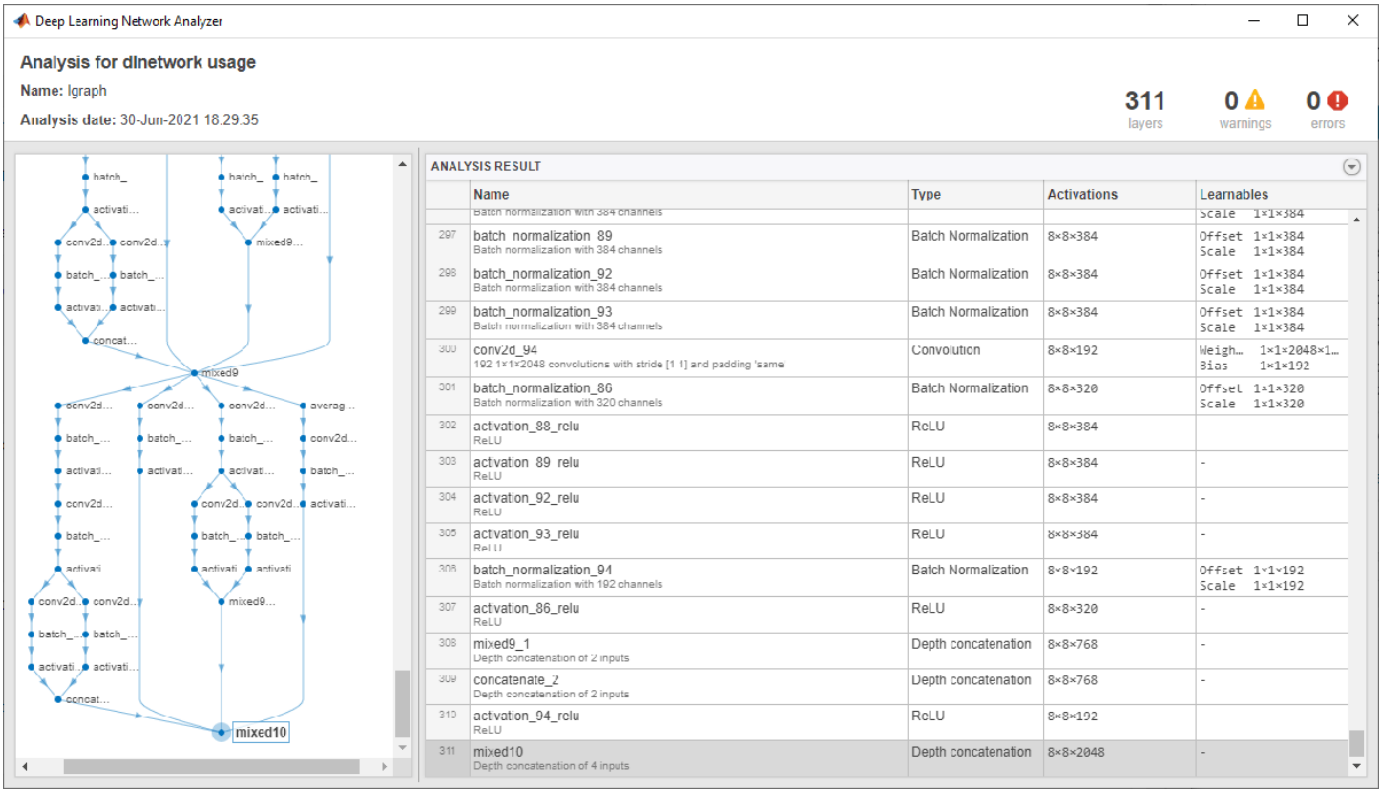
```
ans =
  ImageInputLayer with properties:

                     Name: 'input_1'
                InputSize: [299 299 3]

   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'rescale-symmetric'
    NormalizationDimension: 'auto'
                      Max: 255
                      Min: 0
```

Custom training does not support this normalization, so you must disable normalization in the network and perform the normalization in the custom training loop instead. Save the minimum and maximum values as doubles in variables named `inputMin` and `inputMax`, respectively, and replace the input layer with an image input layer without normalization.

```
inputMin = double(lgraph.Layers(1).Min);
inputMax = double(lgraph.Layers(1).Max);
layer = imageInputLayer(inputSizeNet,'Normalization','none','Name','input');
lgraph = replaceLayer(lgraph,'input_1',layer);
```

Determine the output size of the network. Use the `analyzeNetwork` function to see the activation sizes of the last layer. To analyze the network for custom training loop workflows, set the `TargetUsage` option to `'dlnetwork'`.

```
analyzeNetwork(lgraph,'TargetUsage','dlnetwork')
```



Create a variable named `outputSizeNet` containing the network output size.

```
outputSizeNet = [8 8 2048];
```

Convert the layer graph to a `dlnetwork` object and view the output layer. The output layer is the `"mixed10"` layer of the Inception-v3 network.

```
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [311×1 nnet.cnn.layer.Layer]
    Connections: [345×2 table]
     Learnables: [376×3 table]
          State: [188×3 table]
     InputNames: {'input'}
    OutputNames: {'mixed10'}
```

Import COCO Data Set

Download images and annotations from the data sets "2014 Train images" and "2014 Train/val annotations," respectively, from https://cocodataset.org/#download. Extract the images and annotations into a folder named `"coco"`. The COCO 2014 data set was collected by Coco Consortium.

Extract the captions from the file `"captions_train2014.json"` using the `jsondecode` function.

```
dataFolder = fullfile(tempdir,"coco");
filename = fullfile(dataFolder,"annotations_trainval2014","annotations","captions_train2014.json
str = fileread(filename);
data = jsondecode(str)
```

```
data = struct with fields:
           info: [1×1 struct]
         images: [82783×1 struct]
       licenses: [8×1 struct]
    annotations: [414113×1 struct]
```

The `annotations` field of the struct contains the data required for image captioning.

```
data.annotations
```

```
ans=414113×1 struct array with fields:
    image_id
    id
    caption
```

The data set contains multiple captions for each image. To ensure the same images do not appear in both training and validation sets, identify the unique images in the data set using the `unique` function by using the IDs in the `image_id` field of the annotations field of the data, then view the number of unique images.

```
numObservationsAll = numel(data.annotations)
```

```
numObservationsAll = 414113
```

```
imageIDs = [data.annotations.image_id];
imageIDsUnique = unique(imageIDs);
numUniqueImages = numel(imageIDsUnique)
```

```
numUniqueImages = 82783
```

Each image has at least five captions. Create a struct `annotationsAll` with these fields:

- `ImageID` — Image ID
- `Filename` — File name of the image
- `Captions` — String array of raw captions
- `CaptionIDs` — Vector of indices of the corresponding captions in `data.annotations`

To make merging easier, sort the annotations by the image IDs.

```
[~,idx] = sort([data.annotations.image_id]);
data.annotations = data.annotations(idx);
```

Loop over the annotations and merge multiple annotations when necessary.

```
i = 0;
j = 0;
imageIDPrev = 0;
while i < numel(data.annotations)
    i = i + 1;

    imageID = data.annotations(i).image_id;
    caption = string(data.annotations(i).caption);

    if imageID ~= imageIDPrev
        % Create new entry
        j = j + 1;
        annotationsAll(j).ImageID = imageID;
        annotationsAll(j).Filename = fullfile(dataFolder,"train2014","COCO_train2014_" + pad(str
        annotationsAll(j).Captions = caption;
        annotationsAll(j).CaptionIDs = i;
    else
        % Append captions
        annotationsAll(j).Captions = [annotationsAll(j).Captions; caption];
        annotationsAll(j).CaptionIDs = [annotationsAll(j).CaptionIDs; i];
    end

    imageIDPrev = imageID;
end
```

Partition the data into training and validation sets. Hold out 5% of the observations for testing.

```
cvp = cvpartition(numel(annotationsAll),'HoldOut',0.05);
idxTrain = training(cvp);
idxTest = test(cvp);
annotationsTrain = annotationsAll(idxTrain);
annotationsTest = annotationsAll(idxTest);
```

The struct contains three fields:

- `id` — Unique identifier for the caption

- `caption` — Image caption, specified as a character vector
- `image_id` — Unique identifier of the image corresponding to the caption

To view the image and the corresponding caption, locate the image file with file name `"train2014\COCO_train2014_XXXXXXXXXXX.jpg"`, where `"XXXXXXXXXXX"` corresponds to the image ID left-padded with zeros to have length 12.

```
imageID = annotationsTrain(1).ImageID;
captions = annotationsTrain(1).Captions;
filename = annotationsTrain(1).Filename;
```

To view the image, use the `imread` and `imshow` functions.

```
img = imread(filename);
figure
imshow(img)
title(captions)
```

Prepare Data for Training

Prepare the captions for training and testing. Extract the text from the `Captions` field of the struct containing both the training and test data (`annotationsAll`), erase the punctuation, and convert the text to lowercase.

```
captionsAll = cat(1,annotationsAll.Captions);
captionsAll = erasePunctuation(captionsAll);
captionsAll = lower(captionsAll);
```

In order to generate captions, the RNN decoder requires special start and stop tokens to indicate when to start and stop generating text, respectively. Add the custom tokens `"<start>"` and `"<stop>"` to the beginnings and ends of the captions, respectively.

```
captionsAll = "<start>" + captionsAll + "<stop>";
```

Tokenize the captions using the `tokenizedDocument` function and specify the start and stop tokens using the `'CustomTokens'` option.

```
documentsAll = tokenizedDocument(captionsAll,'CustomTokens',["<start>" "<stop>"]);
```

Create a `wordEncoding` object that maps words to numeric indices and back. Reduce the memory requirements by specifying a vocabulary size of 5000 corresponding to the most frequently observed words in the training data. To avoid bias, use only the documents corresponding to the training set.

```
enc = wordEncoding(documentsAll(idxTrain),'MaxNumWords',5000,'Order','frequency');
```

Create an augmented image datastore containing the images corresponding to the captions. Set the output size to match the input size of the convolutional network. To keep the images synchronized with the captions, specify a table of file names for the datastore by reconstructing the file names using the image ID. To return grayscale images as 3-channel RGB images, set the `'ColorPreprocessing'` option to `'gray2rgb'`.

```
tblFilenames = table(cat(1,annotationsTrain.Filename));
augimdsTrain = augmentedImageDatastore(inputSizeNet,tblFilenames,'ColorPreprocessing','gray2rgb'
```

```
augimdsTrain =
  augmentedImageDatastore with properties:
```

```
         NumObservations: 78644
           MiniBatchSize: 1
       DataAugmentation: 'none'
     ColorPreprocessing: 'gray2rgb'
             OutputSize: [299 299]
         OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

Initialize Model Parameters

Initialize the model parameters. Specify 512 hidden units with a word embedding dimension of 256.

```
embeddingDimension = 256;
numHiddenUnits = 512;
```

Initialize a struct containing the parameters for the encoder model.

- Initialize the weights of the fully connected operations using the Glorot initializer, specified by the `initializeGlorot` function, listed at the end of the example. Specify the output size to match the embedding dimension of the decoder (256) and an input size to match the number of output channels of the pretrained network. The `'mixed10'` layer of the Inception-v3 network outputs data with 2048 channels.

```
numFeatures = outputSizeNet(1) * outputSizeNet(2);
inputSizeEncoder = outputSizeNet(3);
parametersEncoder = struct;

% Fully connect
parametersEncoder.fc.Weights = dlarray(initializeGlorot(embeddingDimension,inputSizeEncoder));
parametersEncoder.fc.Bias = dlarray(zeros([embeddingDimension 1],'single'));
```

Initialize a struct containing parameters for the decoder model.

- Initialize the word embedding weights with the size given by the embedding dimension and the vocabulary size plus one, where the extra entry corresponds to the padding value.
- Initialize the weights and biases for the Bahdanau attention mechanism with sizes corresponding to the number of hidden units of the GRU operation.
- Initialize the weights and bias of the GRU operation.
- Initialize the weights and biases of two fully connected operations.

For the model decoder parameters, initialize each of the weighs and biases with the Glorot initializer and zeros, respectively.

```
inputSizeDecoder = enc.NumWords + 1;
parametersDecoder = struct;

% Word embedding
parametersDecoder.emb.Weights = dlarray(initializeGlorot(embeddingDimension,inputSizeDecoder));

% Attention
parametersDecoder.attention.Weights1 = dlarray(initializeGlorot(numHiddenUnits,embeddingDimensio
parametersDecoder.attention.Bias1 = dlarray(zeros([numHiddenUnits 1],'single'));
parametersDecoder.attention.Weights2 = dlarray(initializeGlorot(numHiddenUnits,numHiddenUnits));
parametersDecoder.attention.Bias2 = dlarray(zeros([numHiddenUnits 1],'single'));
parametersDecoder.attention.WeightsV = dlarray(initializeGlorot(1,numHiddenUnits));
```

```
parametersDecoder.attention.BiasV = dlarray(zeros(1,1,'single'));

% GRU
parametersDecoder.gru.InputWeights = dlarray(initializeGlorot(3*numHiddenUnits,2*embeddingDimens
parametersDecoder.gru.RecurrentWeights = dlarray(initializeGlorot(3*numHiddenUnits,numHiddenUnit
parametersDecoder.gru.Bias = dlarray(zeros(3*numHiddenUnits,1,'single'));

% Fully connect
parametersDecoder.fc1.Weights = dlarray(initializeGlorot(numHiddenUnits,numHiddenUnits));
parametersDecoder.fc1.Bias = dlarray(zeros([numHiddenUnits 1],'single'));

% Fully connect
parametersDecoder.fc2.Weights = dlarray(initializeGlorot(enc.NumWords+1,numHiddenUnits));
parametersDecoder.fc2.Bias = dlarray(zeros([enc.NumWords+1 1],'single'));
```

Define Model Functions

Create the functions `modelEncoder` and `modelDecoder`, listed at the end of the example, which compute the outputs of the encoder and decoder models, respectively.

The `modelEncoder` function, listed in the Encoder Model Function on page 4-0 section of the example, takes as input an array of activations `dlX` from the output of the pretrained network and passes it through a fully connected operation and a ReLU operation. Because the pretrained network does not need to be traced for automatic differentiation, extracting the features outside the encoder model function is more computationally efficient.

The `modelDecoder` function, listed in the Decoder Model Function on page 4-0 section of the example, takes as input a single input time-step corresponding to an input word, the decoder model parameters, the features from the encoder, and the network state, and returns the predictions for the next time step, the updated network state, and the attention weights.

Specify Training Options

Specify the options for training. Train for 30 epochs with a mini-batch size of 128 and display the training progress in a plot.

```
miniBatchSize = 128;
numEpochs = 30;
plots = "training-progress";
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Train Network

Train the network using a custom training loop.

At the beginning of each epoch, shuffle the input data. To keep the images in the augmented image datastore and the captions synchronized, create an array of shuffled indices that indexes into both data sets.

For each mini-batch:

- Rescale the images to the size that the pretrained network expects.
- For each image, select a random caption.
- Convert the captions to sequences of word indices. Specify right-padding of the sequences with the padding value corresponding to the index of the padding token.
- Convert the data to `dlarray` objects. For the images, specify dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the data to `gpuArray` objects.
- Extract the image features using the pretrained network and reshape them to the size the encoder expects.
- Evaluate the model gradients and loss using the `dlfeval` and `modelGradients` functions.
- Update the encoder and decoder model parameters using the `adamupdate` function.
- Display the training progress in a plot.

Initialize the parameters for the Adam optimizer.

```
trailingAvgEncoder = [];
trailingAvgSqEncoder = [];

trailingAvgDecoder = [];
trailingAvgSqDecoder = [];
```

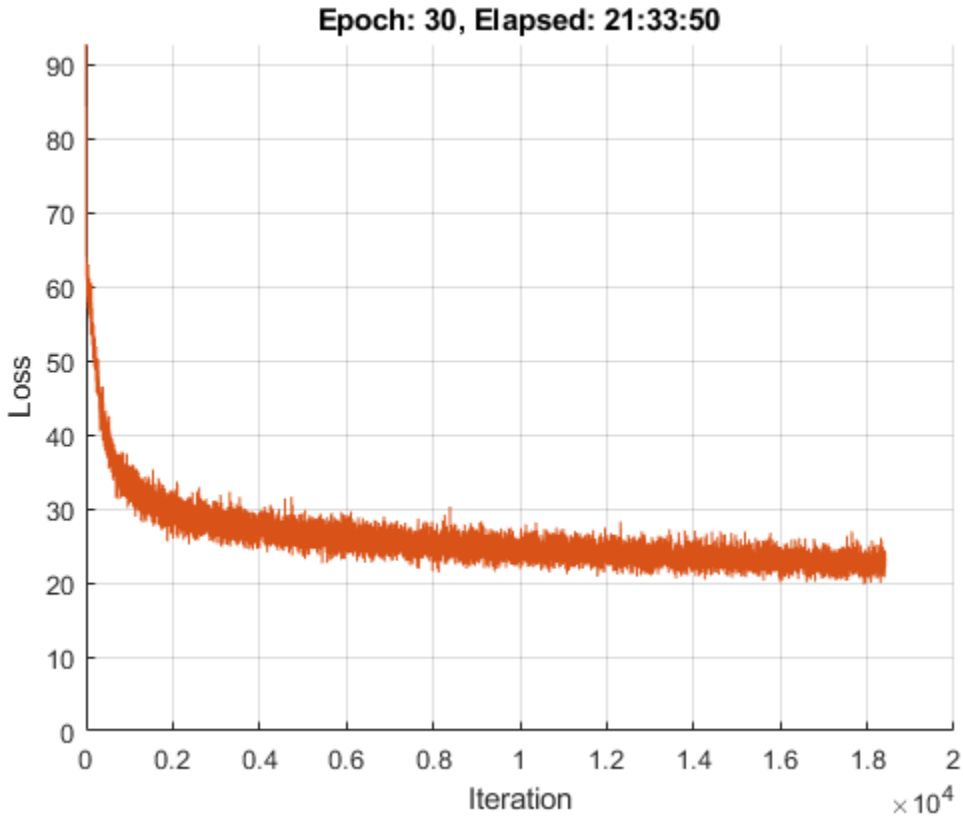
Initialize the training progress plot. Create an animated line that plots the loss against the corresponding iteration.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    xlabel("Iteration")
    ylabel("Loss")
    ylim([0 inf])
    grid on
end
```

Train the model.

```
iteration = 0;
numObservationsTrain = numel(annotationsTrain);
numIterationsPerEpoch = floor(numObservationsTrain / miniBatchSize);
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    idxShuffle = randperm(numObservationsTrain);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Determine mini-batch indices.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        idxMiniBatch = idxShuffle(idx);
```

```
            % Read mini-batch of data.
            tbl = readByIndex(augimdsTrain,idxMiniBatch);
            X = cat(4,tbl.input{:});
            annotations = annotationsTrain(idxMiniBatch);

            % For each image, select random caption.
            idx = cellfun(@(captionIDs) randsample(captionIDs,1),{annotations.CaptionIDs});
            documents = documentsAll(idx);

            % Create batch of data.
            [dlX, dlT] = createBatch(X,documents,dlnet,inputMin,inputMax,enc,executionEnvironment);

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients function.
            [gradientsEncoder, gradientsDecoder, loss] = dlfeval(@modelGradients, parametersEncoder,
                parametersDecoder, dlX, dlT);

            % Update encoder using adamupdate.
            [parametersEncoder, trailingAvgEncoder, trailingAvgSqEncoder] = adamupdate(parametersEnco
                gradientsEncoder, trailingAvgEncoder, trailingAvgSqEncoder, iteration);

            % Update decoder using adamupdate.
            [parametersDecoder, trailingAvgDecoder, trailingAvgSqDecoder] = adamupdate(parametersDeco
                gradientsDecoder, trailingAvgDecoder, trailingAvgSqDecoder, iteration);

            % Display the training progress.
            if plots == "training-progress"
                D = duration(0,0,toc(start),'Format','hh:mm:ss');
                addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
                title("Epoch: " + epoch + ", Elapsed: " + string(D))

                drawnow
            end
        end
    end
```



Predict New Captions

The caption generation process is different from the process for training. During training, at each time step, the decoder uses the true value of the previous time step as input. This is known as "teacher forcing". When making predictions on new data, the decoder uses the previous predicted values instead of the true values.

Predicting the most likely word for each step in the sequence can lead to suboptimal results. For example, if the decoder predicts the first word of a caption is "a" when given an image of an elephant, then the probability of predicting "elephant" for the next word becomes much more unlikely because of the extremely low probability of the phrase "a elephant" appearing in English text.

To address this issue, you can use the beam search algorithm: instead of taking the most likely prediction for each step in the sequence, take the top k predictions (the beam index) and for each following step, keep the top k predicted sequences so far according to the overall score.

Generate a caption of a new image by extracting the image features, inputting them into the encoder, and then using the `beamSearch` function, listed in the Beam Search Function on page 4-0 section of the example.

```
img = imread("laika_sitting.jpg");
dlX = extractImageFeatures(dlnet,img,inputMin,inputMax,executionEnvironment);

beamIndex = 3;
maxNumWords = 20;
[words,attentionScores] = beamSearch(dlX,beamIndex,parametersEncoder,parametersDecoder,enc,maxNu
caption = join(words)
```

```
caption =
"a dog is standing on a tile floor"
```

Display the image with the caption.

```
figure
imshow(img)
title(caption)
```

a dog is standing on a tile floor

Predict Captions for Data Set

To predict captions for a collection of images, loop over mini-batches of data in the datastore and extract the features from the images using the `extractImageFeatures` function. Then, loop over the images in the mini-batch and generate captions using the `beamSearch` function.

Create an augmented image datastore and set the output size to match the input size of the convolutional network. To output grayscale images as 3-channel RGB images, set the `'ColorPreprocessing'` option to `'gray2rgb'`.

```
tblFilenamesTest = table(cat(1,annotationsTest.Filename));
augimdsTest = augmentedImageDatastore(inputSizeNet,tblFilenamesTest,'ColorPreprocessing','gray2rg
```

```
augimdsTest =
  augmentedImageDatastore with properties:

         NumObservations: 4139
           MiniBatchSize: 1
        DataAugmentation: 'none'
      ColorPreprocessing: 'gray2rgb'
              OutputSize: [299 299]
          OutputSizeMode: 'resize'
    DispatchInBackground: 0
```

Generate captions for the test data. Predicting captions on a large data set can take some time. If you have Parallel Computing Toolbox™, then you can make predictions in parallel by generating captions inside a `parfor` look. If you do not have Parallel Computing Toolbox. then the `parfor` loop runs in serial.

```
beamIndex = 2;
maxNumWords = 20;

numObservationsTest = numel(annotationsTest);
numIterationsTest = ceil(numObservationsTest/miniBatchSize);

captionsTestPred = strings(1,numObservationsTest);
documentsTestPred = tokenizedDocument(strings(1,numObservationsTest));

for i = 1:numIterationsTest
    % Mini-batch indices.
    idxStart = (i-1)*miniBatchSize+1;
    idxEnd = min(i*miniBatchSize,numObservationsTest);
    idx = idxStart:idxEnd;

    sz = numel(idx);

    % Read images.
    tbl = readByIndex(augimdsTest,idx);

    % Extract image features.
    X = cat(4,tbl.input{:});
    dlX = extractImageFeatures(dlnet,X,inputMin,inputMax,executionEnvironment);

    % Generate captions.
    captionsPredMiniBatch = strings(1,sz);
    documentsPredMiniBatch = tokenizedDocument(strings(1,sz));
```

```
    parfor j = 1:sz
        words = beamSearch(dlX(:,:,j),beamIndex,parametersEncoder,parametersDecoder,enc,maxNumWo
        captionsPredMiniBatch(j) = join(words);
        documentsPredMiniBatch(j) = tokenizedDocument(words,'TokenizeMethod','none');
    end

    captionsTestPred(idx) = captionsPredMiniBatch;
    documentsTestPred(idx) = documentsPredMiniBatch;
end
```

```
Analyzing and transferring files to the workers ...done.
```

To view a test image with the corresponding caption, use the `imshow` function and set the title to the predicted caption.

```
idx = 1;
tbl = readByIndex(augimdsTest,idx);
img = tbl.input{1};
figure
imshow(img)
title(captionsTestPred(idx))
```

Evaluate Model Accuracy

To evaluate the accuracy of the captions using the BLEU score, calculate the BLEU score for each caption (the candidate) against the corresponding captions in the test set (the references) using the `bleuEvaluationScore` function. Using the `bleuEvaluationScore` function, you can compare a single candidate document to multiple reference documents.

The `bleuEvaluationScore` function, by default, scores similarity using n-grams of length one through four. As the captions are short, this behavior can lead to uninformative results as most scores are close to zero. Set the n-gram length to one through two by setting the `'NgramWeights'` option to a two-element vector with equal weights.

```
ngramWeights = [0.5 0.5];

for i = 1:numObservationsTest
    annotation = annotationsTest(i);

    captionIDs = annotation.CaptionIDs;
    candidate = documentsTestPred(i);
    references = documentsAll(captionIDs);

    score = bleuEvaluationScore(candidate,references,'NgramWeights',ngramWeights);

    scores(i) = score;
end
```

View the mean BLEU score.

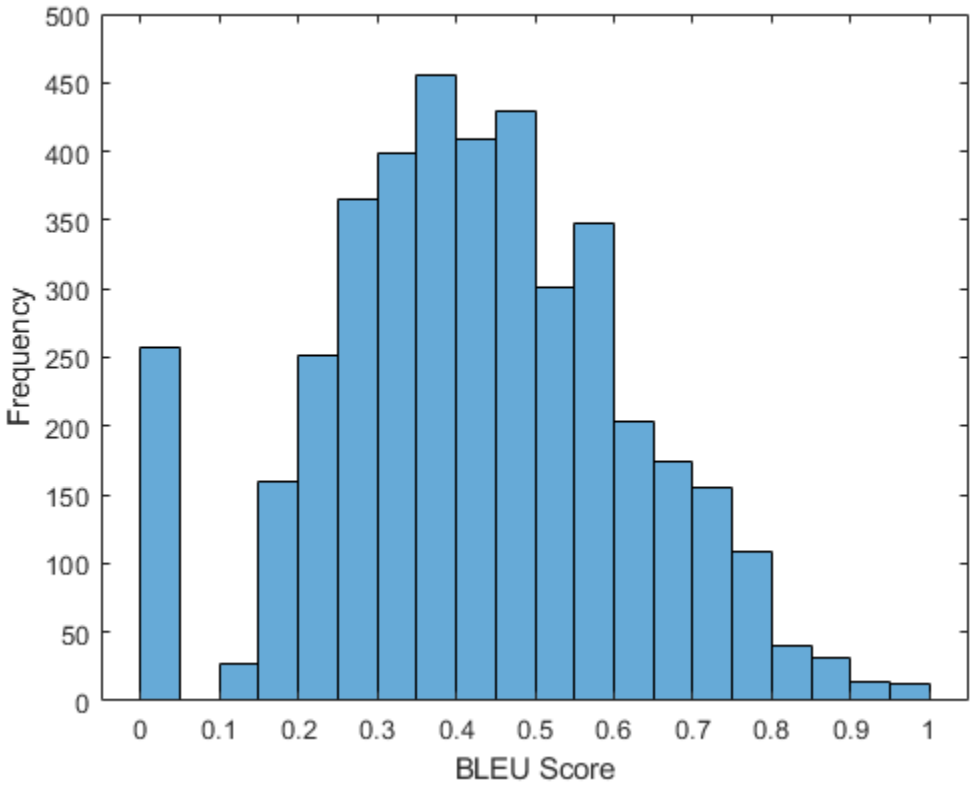
```
scoreMean = mean(scores)
```

```
scoreMean = 0.4224
```

Visualize the scores in a histogram.

```
figure
histogram(scores)
```

```
xlabel("BLEU Score")
ylabel("Frequency")
```



Attention Function

The `attention` function calculates the context vector and the attention weights using Bahdanau attention.

```
function [contextVector, attentionWeights] = attention(hidden,features,weights1, ...
    bias1,weights2,bias2,weightsV,biasV)

% Model dimensions.
[embeddingDimension,numFeatures,miniBatchSize] = size(features);
numHiddenUnits = size(weights1,1);

% Fully connect.
dlY1 = reshape(features,embeddingDimension, numFeatures*miniBatchSize);
dlY1 = fullyconnect(dlY1,weights1,bias1,'DataFormat','CB');
dlY1 = reshape(dlY1,numHiddenUnits,numFeatures,miniBatchSize);

% Fully connect.
dlY2 = fullyconnect(hidden,weights2,bias2,'DataFormat','CB');
dlY2 = reshape(dlY2,numHiddenUnits,1,miniBatchSize);

% Addition, tanh.
scores = tanh(dlY1 + dlY2);
scores = reshape(scores, numHiddenUnits, numFeatures*miniBatchSize);
```

```
% Fully connect, softmax.
attentionWeights = fullyconnect(scores,weightsV,biasV,'DataFormat','CB');
attentionWeights = reshape(attentionWeights,1,numFeatures,miniBatchSize);
attentionWeights = softmax(attentionWeights,'DataFormat','SCB');

% Context.
contextVector = attentionWeights .* features;
contextVector = squeeze(sum(contextVector,2));

end
```

Embedding Function

The `embedding` function maps an array of indices to a sequence of embedding vectors.

```
function Z = embedding(X, weights)

% Reshape inputs into a vector
[N, T] = size(X, 1:2);
X = reshape(X, N*T, 1);

% Index into embedding matrix
Z = weights(:, X);

% Reshape outputs by separating out batch and sequence dimensions
Z = reshape(Z, [], N, T);

end
```

Feature Extraction Function

The `extractImageFeatures` function takes as input a trained `dlnetwork` object, an input image, statistics for image rescaling, and the execution environment, and returns a `dlarray` containing the features extracted from the pretrained network.

```
function dlX = extractImageFeatures(dlnet,X,inputMin,inputMax,executionEnvironment)

% Resize and rescale.
inputSize = dlnet.Layers(1).InputSize(1:2);
X = imresize(X,inputSize);
X = rescale(X,-1,1,'InputMin',inputMin,'InputMax',inputMax);

% Convert to dlarray.
dlX = dlarray(X,'SSCB');

% Convert to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end

% Extract features and reshape.
dlX = predict(dlnet,dlX);
sz = size(dlX);
numFeatures = sz(1) * sz(2);
inputSizeEncoder = sz(3);
miniBatchSize = sz(4);
dlX = reshape(dlX,[numFeatures inputSizeEncoder miniBatchSize]);

end
```

Batch Creation Function

The `createBatch` function takes as input a mini-batch of data, tokenized captions, a pretrained network, statistics for image rescaling, a word encoding, and the execution environment, and returns a mini-batch of data corresponding to the extracted image features and captions for training.

```
function [dlX, dlT] = createBatch(X,documents,dlnet,inputMin,inputMax,enc,executionEnvironment)

dlX = extractImageFeatures(dlnet,X,inputMin,inputMax,executionEnvironment);

% Convert documents to sequences of word indices.
T = doc2sequence(enc,documents,'PaddingDirection','right','PaddingValue',enc.NumWords+1);
T = cat(1,T{:});

% Convert mini-batch of data to dlarray.
dlT = dlarray(T);

% If training on a GPU, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlT = gpuArray(dlT);
end

end
```

Encoder Model Function

The `modelEncoder` function takes as input an array of activations `dlX` and passes it through a fully connected operation and a ReLU operation. For the fully connected operation, operate on the channel dimension only. To apply the fully connected operation across the channel dimension only, flatten the other channels into a single dimension and specify this dimension as the batch dimension using the `'DataFormat'` option of the `fullyconnect` function.

```
function dlY = modelEncoder(dlX,parametersEncoder)

[numFeatures,inputSizeEncoder,miniBatchSize] = size(dlX);

% Fully connect
weights = parametersEncoder.fc.Weights;
bias = parametersEncoder.fc.Bias;
embeddingDimension = size(weights,1);

dlX = permute(dlX,[2 1 3]);
dlX = reshape(dlX,inputSizeEncoder,numFeatures*miniBatchSize);
dlY = fullyconnect(dlX,weights,bias,'DataFormat','CB');
dlY = reshape(dlY,embeddingDimension,numFeatures,miniBatchSize);

% ReLU
dlY = relu(dlY);

end
```

Decoder Model Function

The `modelDecoder` function takes as input a single time-step `dlX`, the decoder model parameters, the features from the encoder, and the network state, and returns the predictions for the next time step, the updated network state, and the attention weights.

```
function [dlY,state,attentionWeights] = modelDecoder(dlX,parametersDecoder,features,state)

hiddenState = state.gru.HiddenState;

% Attention
weights1 = parametersDecoder.attention.Weights1;
bias1 = parametersDecoder.attention.Bias1;
weights2 = parametersDecoder.attention.Weights2;
bias2 = parametersDecoder.attention.Bias2;
weightsV = parametersDecoder.attention.WeightsV;
biasV = parametersDecoder.attention.BiasV;
[contextVector, attentionWeights] = attention(hiddenState,features,weights1,bias1,weights2,bias2

% Embedding
weights = parametersDecoder.emb.Weights;
dlX = embedding(dlX,weights);

% Concatenate
dlY = cat(1,contextVector,dlX);

% GRU
inputWeights = parametersDecoder.gru.InputWeights;
recurrentWeights = parametersDecoder.gru.RecurrentWeights;
bias = parametersDecoder.gru.Bias;
[dlY, hiddenState] = gru(dlY, hiddenState, inputWeights, recurrentWeights, bias, 'DataFormat','CB

% Update state
state.gru.HiddenState = hiddenState;

% Fully connect
weights = parametersDecoder.fc1.Weights;
bias = parametersDecoder.fc1.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CB');

% Fully connect
weights = parametersDecoder.fc2.Weights;
bias = parametersDecoder.fc2.Bias;
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CB');

end
```

Model Gradients

The `modelGradients` function takes as input the encoder and decoder parameters, the encoder features `dlX`, and the target caption `dlT`, and returns the gradients of the encoder and decoder parameters with respect to the loss, the loss, and the predictions.

```
function [gradientsEncoder,gradientsDecoder,loss,dlYPred] = ...
    modelGradients(parametersEncoder,parametersDecoder,dlX,dlT)

miniBatchSize = size(dlX,3);
sequenceLength = size(dlT,2) - 1;
vocabSize = size(parametersDecoder.emb.Weights,2);

% Model encoder
features = modelEncoder(dlX,parametersEncoder);

% Initialize state
```

```
numHiddenUnits = size(parametersDecoder.attention.Weights1,1);
state = struct;
state.gru.HiddenState = dlarray(zeros([numHiddenUnits miniBatchSize],'single'));

dlYPred = dlarray(zeros([vocabSize miniBatchSize sequenceLength],'like',dlX));
loss = dlarray(single(0));

padToken = vocabSize;

for t = 1:sequenceLength
    decoderInput = dlT(:,t);

    dlYReal = dlT(:,t+1);

    [dlYPred(:,:,t),state] = modelDecoder(decoderInput,parametersDecoder,features,state);

    mask = dlYReal ~= padToken;

    loss = loss + sparseCrossEntropyAndSoftmax(dlYPred(:,:,t),dlYReal,mask);
end

% Calculate gradients
[gradientsEncoder,gradientsDecoder] = dlgradient(loss, parametersEncoder,parametersDecoder);

end
```

Sparse Cross Entropy and Softmax Loss Function

The `sparseCrossEntropyAndSoftmax` takes as input the predictions `dlY`, corresponding targets `dlT`, and sequence padding mask, and applies the `softmax` functions and returns the cross-entropy loss.

```
function loss = sparseCrossEntropyAndSoftmax(dlY, dlT, mask)

miniBatchSize = size(dlY, 2);

% Softmax.
dlY = softmax(dlY,'DataFormat','CB');

% Find rows corresponding to the target words.
idx = sub2ind(size(dlY), dlT', 1:miniBatchSize);
dlY = dlY(idx);

% Bound away from zero.
dlY = max(dlY, single(1e-8));

% Masked loss.
loss = log(dlY) .* mask';
loss = -sum(loss,'all') ./ miniBatchSize;

end
```

Beam Search Function

The `beamSearch` function takes as input the image features `dlX`, a beam index, the parameters for the encoder and decoder networks, a word encoding, and a maximum sequence length, and returns the caption words for the image using the beam search algorithm.

```
function [words,attentionScores] = beamSearch(dlX,beamIndex,parametersEncoder,parametersDecoder,
    enc,maxNumWords)

% Model dimensions
numFeatures = size(dlX,1);
numHiddenUnits = size(parametersDecoder.attention.Weights1,1);

% Extract features
features = modelEncoder(dlX,parametersEncoder);

% Initialize state
state = struct;
state.gru.HiddenState = dlarray(zeros([numHiddenUnits 1],'like',dlX));

% Initialize candidates
candidates = struct;
candidates.State = state;
candidates.Words = "<start>";
candidates.Score = 0;
candidates.AttentionScores = dlarray(zeros([numFeatures maxNumWords],'like',dlX));
candidates.StopFlag = false;

t = 0;

% Loop over words
while t < maxNumWords
    t = t + 1;

    candidatesNew = [];

    % Loop over candidates
    for i = 1:numel(candidates)

        % Stop generating when stop token is predicted
        if candidates(i).StopFlag
            continue
        end

        % Candidate details
        state = candidates(i).State;
        words = candidates(i).Words;
        score = candidates(i).Score;
        attentionScores = candidates(i).AttentionScores;

        % Predict next token
        decoderInput = word2ind(enc,words(end));
        [dlYPred,state,attentionScores(:,t)] = modelDecoder(decoderInput,parametersDecoder,featu

        dlYPred = softmax(dlYPred,'DataFormat','CB');
        [scoresTop,idxTop] = maxk(extractdata(dlYPred),beamIndex);
        idxTop = gather(idxTop);

        % Loop over top predictions
        for j = 1:beamIndex
            candidate = struct;

            candidateWord = ind2word(enc,idxTop(j));
            candidateScore = scoresTop(j);
```

```
            if candidateWord == "<stop>"
                candidate.StopFlag = true;
                attentionScores(:,t+1:end) = [];
            else
                candidate.StopFlag = false;
            end

            candidate.State = state;
            candidate.Words = [words candidateWord];
            candidate.Score = score + log(candidateScore);
            candidate.AttentionScores = attentionScores;

            candidatesNew = [candidatesNew candidate];
        end
    end

    % Get top candidates
    [~,idx] = maxk([candidatesNew.Score],beamIndex);
    candidates = candidatesNew(idx);

    % Stop predicting when all candidates have stop token
    if all([candidates.StopFlag])
        break
    end
end

% Get top candidate
words = candidates(1).Words(2:end-1);
attentionScores = candidates(1).AttentionScores;

end
```

Glorot Weight Initialization Function

The `initializeGlorot` function generates an array of weights according to Glorot initialization.

```
function weights = initializeGlorot(numOut, numIn)

varWeights = sqrt( 6 / (numIn + numOut) );
weights = varWeights * (2 * rand([numOut, numIn], 'single') - 1);

end
```

See Also

`word2ind` | `tokenizedDocument` | `wordEncoding` | `dlarray` | `adamupdate` | `dlupdate` | `dlfeval` | `dlgradient` | `crossentropy` | `softmax` | `lstm` | `doc2sequence` | `gru`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216

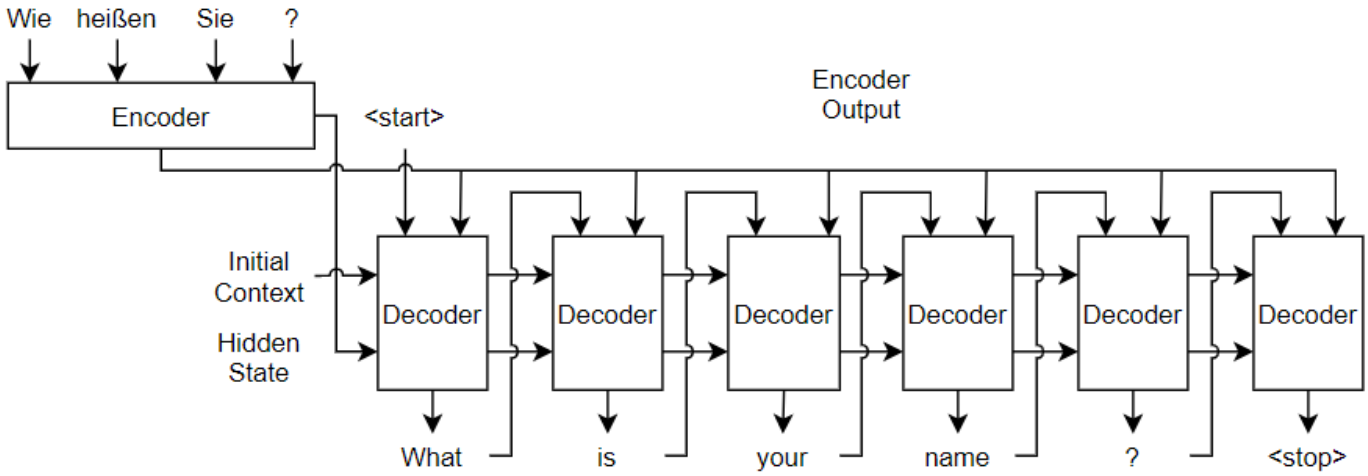
- “Multilabel Text Classification Using Deep Learning” on page 4-106
- “Automatic Differentiation Background” on page 18-200

Language Translation Using Deep Learning

This example shows how to train a German to English language translator using a recurrent sequence-to-sequence encoder-decoder model with attention.

Recurrent encoder-decoder models have proven successful at tasks such as abstractive text summarization and neural machine translation. These models consist of an *encoder*, which typically processes input data with a recurrent layer such as an LSTM layer, and a *decoder* which maps the encoded input into the desired output, typically also with a recurrent layer. Models that incorporate *attention mechanisms* into the models allow the decoder to focus on parts of the encoded input while generating the translation one time step at a time. This example implements *Bahdanau attention* [1] on page 4-0 using the custom layer `attentionLayer`, attached to this example as a supporting file. To access this layer, open this example as a live script.

This diagram shows the structure of a language translation model. The input text, specified as a sequence of words, is passed through the encoder, which outputs an encoded version of the input sequence and a hidden state used to initialize the decoder state. The decoder makes predictions one word at a time using previous prediction as input and also outputs update state and context values.



For more information and the details about the encoder and decoder networks used in this example, see the Define Encoder and Decoder Networks on page 4-0 section of the example.

Predicting the most likely word for each step in the sequence can lead to suboptimal results. Any incorrect predictions can cause even more incorrect predictions in later time steps. For example, for the target text `"An eagle flew by."`, if the decoder predicts the first word of a translation as `"A"`, then the probability of predicting "eagle" for the next word becomes much more unlikely because of the low probability of the phrase "a eagle" appearing in English text. The translation generation process differs for training and prediction. This example uses different approaches to stabilize training and the predictions:

- To stabilize training, you can randomly use the target values as inputs to the decoder. In particular, you can adjust the probability used to inject the target values as training progresses. For example, you can train using the target values at a much higher rate at the start of training, then decay the probability such that towards the end of training the model uses only the previous predictions. This technique is known as *scheduled sampling* [2] on page 4-0 . For more information, see the Decoder Predictions Function on page 4-0 section of the example.

- To improve the predictions at translation time, for each time step, you can consider the top K predictions for some positive integer K and explore different sequences of predictions to identify the best combination. This technique is known as *beam search*. For more information, see the Beam Search Function on page 4-0 section of the example.

This example shows how to load and preprocess text data to train a German to English language translator, define the encoder and decoder networks, train the model using a custom training loop, and generate translations using beam search.

Note: Language translation is a computationally intensive task. Training on the full data set used in this example can take many hours to run. To make the example run quicker, you can reduce training time at the cost of accuracy of predictions with previously unseen data by discarding a portion of the training data. Removing observations can speed up training because it reduces the amount of data to process in an epoch and reduces the vocabulary size of the training data.

To shorten the time it takes to run the example, discard 70% of the data. Note that discarding large amounts of data negatively affects the accuracy of the learned model. For more accurate results, reduce the amount of discarded data. To speed up the example, increase the amount of discarded data.

```
discardProp = 0.70;
```

Load Training Data

Download and extract the English-German Tab-delimited Bilingual Sentence Pairs data set. The data comes from http://www.manythings.org/anki and https://tatoeba.org, and is provided under the Tatoeba Terms of Use and the CC-BY license.

```
downloadFolder = tempdir;
url = "http://www.manythings.org/anki/deu-eng.zip";
filename = fullfile(downloadFolder,"deu-eng.zip");
dataFolder = fullfile(downloadFolder,"deu-eng");

if ~exist(dataFolder,"dir")
    fprintf("Downloading English-German Tab-delimited Bilingual Sentence Pairs data set (7.6 MB)
    websave(filename,url);
    unzip(filename,dataFolder);
    fprintf("Done.\n")
end
```

Create a table that contains the sentence pairs specified as strings. Read the tab-delimited sentences pairs using `readtable`. Specify the German text as the source and the English text as the target.

```
filename = fullfile(dataFolder,"deu.txt");

opts = delimitedTextImportOptions(...
    Delimiter="\t", ...
    VariableNames=["Target" "Source" "License"], ...
    SelectedVariableNames=["Source" "Target"], ...
    VariableTypes=["string" "string" "string"], ...
    Encoding="UTF-8");
```

View the first few sentence pairs in the data.

```
data = readtable(filename, opts);
head(data)
```

```
ans=8×2 table
        Source          Target
    ________________    _______

    "Geh."              "Go."
    "Hallo!"            "Hi."
    "Grüß Gott!"        "Hi."
    "Lauf!"             "Run!"
    "Lauf!"             "Run."
    "Potzdonner!"       "Wow!"
    "Donnerwetter!"     "Wow!"
    "Feuer!"            "Fire!"
```

Training on the full dataset can take a long time to run. To reduce training time at the cost of accuracy, you can discard a portion of the training data. Removing observations can speed up training because it reduces the amount of data to process in an epoch as well as reducing the vocabulary size of the training data.

Discard a portion of the data according to the `discardProp` variable defined at the start of the example. Note that discarding large amounts of data negatively affects the accuracy of the learned model. For more accurate results, reduce the amount of discarded data by setting `discardProp` to a lower value.

```
idx = size(data,1) - floor(discardProp*size(data,1)) + 1;
data(idx:end,:) = [];
```

View the number of remaining observations.

```
size(data,1)
```

```
ans = 68124
```

Split the data into training and test partitions containing 90% and 10% of the data, respectively.

```
trainingProp = 0.9;
idx = randperm(size(data,1),floor(trainingProp*size(data,1)));
dataTrain = data(idx,:);
dataTest = data;
dataTest(idx,:) = [];
```

View the first few rows of the training data.

```
head(dataTrain)
```

```
ans=8×2 table
                  Source                          Target
    ____________________________________    ____________________________

    "Tom erschoss Mary."                    "Tom shot Mary."
    "Ruf mich bitte an."                    "Call me, please."
    "Kann das einer nachprüfen?"            "Can someone check this?"
    "Das lasse ich mir nicht gefallen!"     "I won't stand for it."
    "Ich mag Englisch nicht."               "I don't like English."
    "Er ist auf dem Laufenden."             "He is up to date."
    "Sie sieht glücklich aus."              "She seems happy."
    "Wo wurden sie geboren?"                "Where were they born?"
```

View the number of training observations.

```
numObservationsTrain = size(dataTrain,1)
```

```
numObservationsTrain = 61311
```

Preprocess Data

Preprocess the text data using the `preprocessText` function, listed at the end of the example. The `preprocessText` function preprocesses and tokenizes the input text for translation by splitting the text into words and adding start and stop tokens.

```
documentsGerman = preprocessText(dataTrain.Source);
```

Create a `wordEncoding` object that maps tokens to a numeric index and vice versa using a vocabulary.

```
encGerman = wordEncoding(documentsGerman);
```

Convert the target data to sequences using the same steps.

```
documentsEnglish = preprocessText(dataTrain.Target);
encEnglish = wordEncoding(documentsEnglish);
```

View the vocabulary sizes of the source and target encodings.

```
numWordsGerman = encGerman.NumWords
```

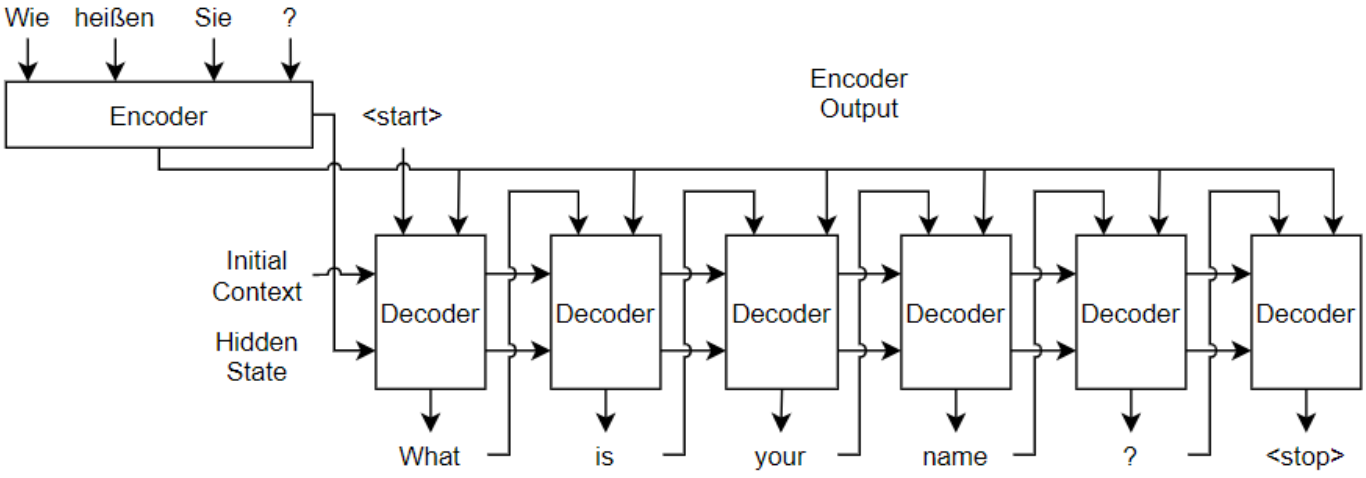
```
numWordsGerman = 12117
```

```
numWordsEnglish = encEnglish.NumWords
```

```
numWordsEnglish = 7226
```

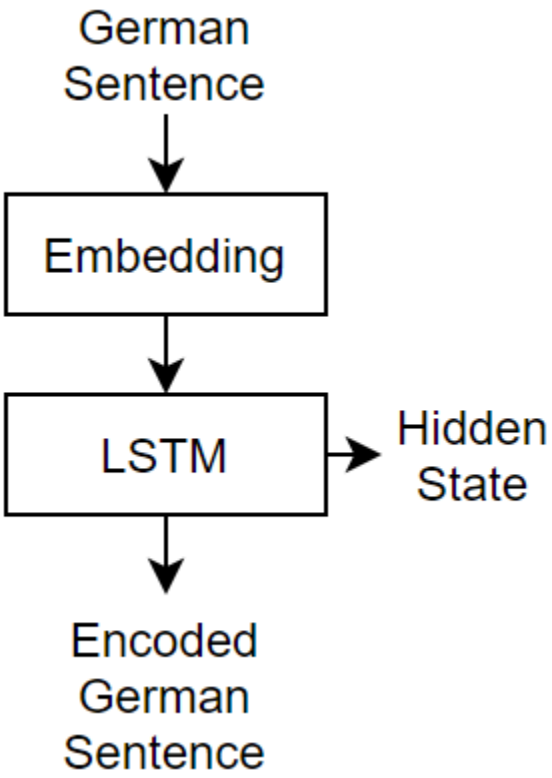
Define Encoder and Decoder Networks

This diagram shows the structure of a language translation model. The input text, specified as a sequence of words, is passed through the encoder, which outputs an encoded version of the input sequence and a hidden state used to initialize the decoder state. The decoder makes predictions one word at a time using previous the prediction as input and also outputs updated state and context values.

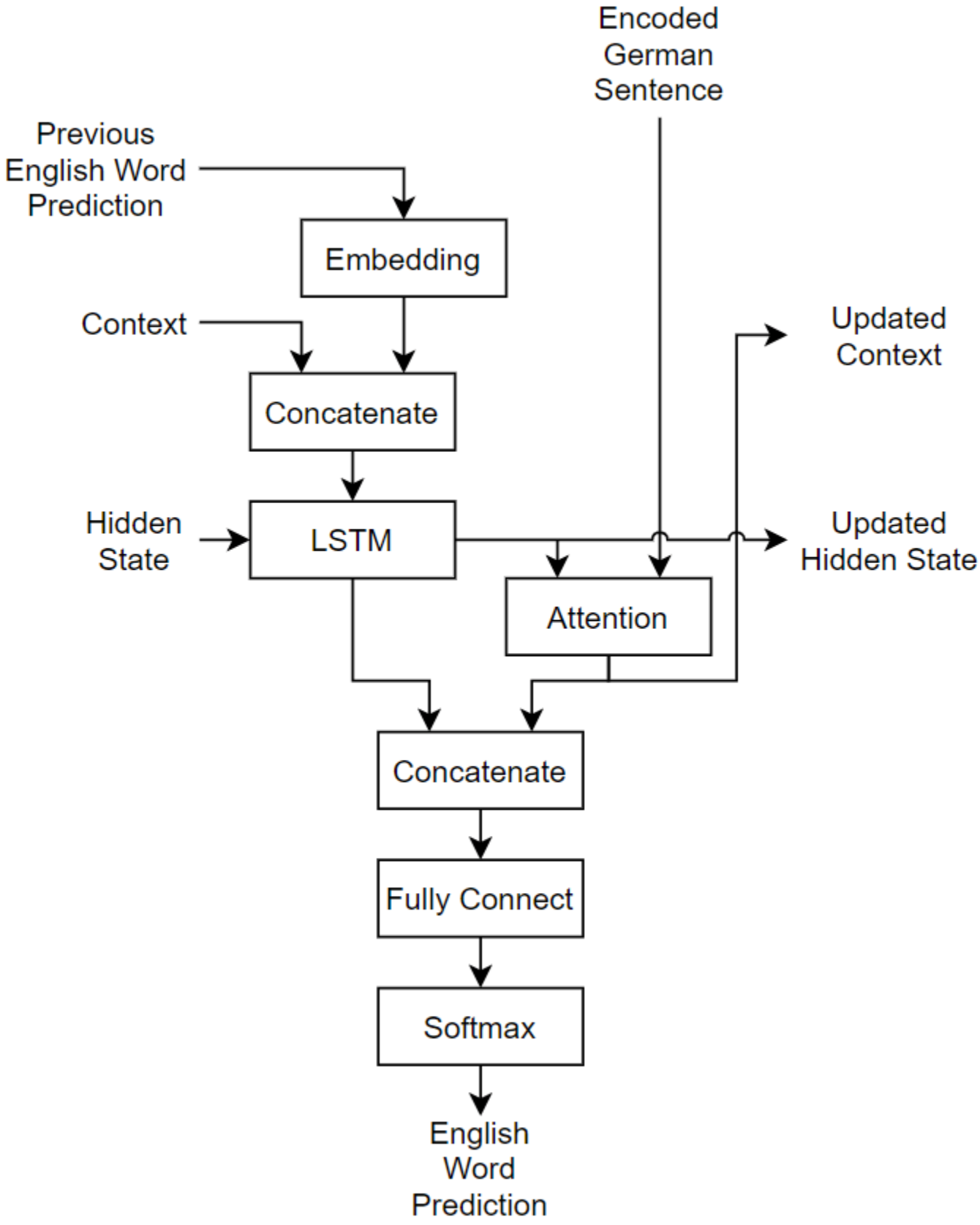


Create the encoder and decoder networks using the `languageTranslationLayers` function, attached to this example as a supporting file. To access this function, open the example as a live script.

For the encoder network, the `languageTranslationLayers` function defines a simple network consisting of an embedding layer followed by an LSTM layer. An embedding operation converts categorical tokens into numeric vectors, where the numeric vectors are learned by the network.



For the decoder network, the `languageTranslationLayers` function defines a network that passes the input data concatenated with the input context through an LSTM layer, and takes the updated hidden state and the encoder output and passes it through an attention mechanism to determine the context vector. The LSTM output and the context vector are then concatenated and passed through a fully connected and a softmax layer for classification.


Encoded
German
Sentence
Previous
English Word
Prediction
Embedding
Context
Updated
Context
Concatenate
Hidden
State
LSTM
Updated
Hidden State
Attention
Concatenate
Fully Connect
Softmax
English
Word
Prediction

Create the encoder and decoder networks using the `languageTranslationLayers` function, attached to this example as a supporting file. To access this function, open the example as a live script. Specify an embedding dimension of 128, and 128 hidden units in the LSTM layers.

```
embeddingDimension = 128;
numHiddenUnits = 128;

[lgraphEncoder,lgraphDecoder] = languageTranslationLayers(embeddingDimension,numHiddenUnits,numWo
```

To train the network in a custom training loop, convert the encoder and decoder networks to `dlnetwork` objects.

```
dlnetEncoder = dlnetwork(lgraphEncoder);
dlnetDecoder = dlnetwork(lgraphDecoder);
```

The decoder has multiple outputs including the context output of the attention layer, which is also passed to another layer. Specify the network outputs using the `OutputNames` property of the decoder `dlnetwork` object.

```
dlnetDecoder.OutputNames = ["softmax" "context" "lstm2/hidden" "lstm2/cell"];
```

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 4-0 section of the example, which takes as input the encoder and decoder model parameters, a mini-batch of input data and the padding masks corresponding to the input data, and the dropout probability and returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Specify Training Options

Train with a mini-batch size of 64 for 15 epochs and a learning rate of 0.005.

```
miniBatchSize = 64;
numEpochs = 15;
learnRate = 0.005;
```

Initialize the options for Adam optimization.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train using gradually decaying values of ϵ for scheduled sampling. Start with a value of $\epsilon = 0.5$ and linearly decay to end with a value of $\epsilon = 0$. For more information about scheduled sampling, see the Decoder Predictions Function on page 4-0 section of the example.

```
epsilonStart = 0.5;
epsilonEnd = 0;
```

Train using SortaGrad [3] on page 4-0 , which is a strategy to improve training of ragged sequences by training for one epoch with the sequences sorted by sequence then shuffling once per epoch thereafter.

Sort the training sequences by sequence length.

```
sequenceLengths = doclength(documentsGerman);
[~,idx] = sort(sequenceLengths);
```

```
documentsGerman = documentsGerman(idx);
documentsEnglish = documentsEnglish(idx);
```

Train Model

Train the model using a custom training loop.

Create array datastores for the source and target data using the `arrayDatastore` function. Combine the datastores using the `combine` function.

```
adsSource = arrayDatastore(documentsGerman);
adsTarget = arrayDatastore(documentsEnglish);
cds = combine(adsSource,adsTarget);
```

Create a mini-batch queue to automatically prepare mini-batches for training.

- Preprocess the training data using the `preprocessMiniBatch` function, which returns a mini-batch of source sequences, target sequences, the corresponding mask, and the initial start token.
- Output `dlarray` objects with the format `"CTB"` (channel, time, batch).
- Discard any partial mini-batches.

```
mbq = minibatchqueue(cds,4, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@(X,Y) preprocessMiniBatch(X,Y,encGerman,encEnglish), ...
    MiniBatchFormat=["CTB" "CTB" "CTB" "CTB"], ...
    PartialMiniBatch="discard");
```

Initialize the training progress plot.

```
figure
C = colororder;
lineLossTrain = animatedline(Color=C(2,:));

xlabel("Iteration")
ylabel("Loss")
ylim([0 inf])
grid on
```

For the encoder and decoder networks, initialize the values for Adam optimization.

```
trailingAvgEncoder = [];
trailingAvgSqEncoder = [];
trailingAvgDecder = [];
trailingAvgSqDecoder = [];
```

Create an array of ϵ values for scheduled sampling.

```
numIterationsPerEpoch = floor(numObservationsTrain/miniBatchSize);
numIterations = numIterationsPerEpoch * numEpochs;
epsilon = linspace(epsilonStart,epsilonEnd,numIterations);
```

Train the model. For each iteration:

- Read a mini-batch of data from the mini-batch queue.
- Compute the model gradients and loss.
- Update the encoder and decoder networks using the `adamupdate` function.

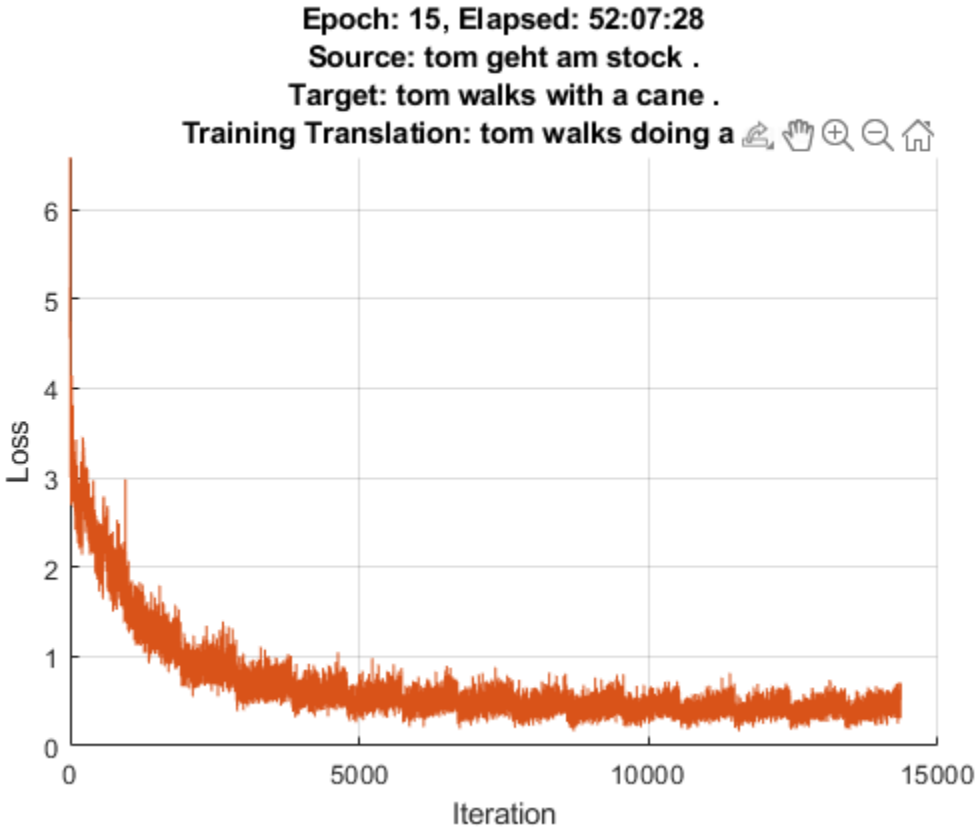
- Update the training progress plot and display an example translation using the `ind2str` function, attached to this example as a supporting file. To access this function, open this example as a live script.
- If the iteration yields the lowest training loss, then save the network.

At the end of each epoch, shuffle the mini-batch queue.

For large data sets, training can take many hours to run.

```
iteration = 0;
start = tic;
lossMin = inf;
reset(mbq)

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,dlT,maskT,decoderInput] = next(mbq);

        % Compute loss and gradients.
        [gradientsEncoder,gradientsDecoder,loss,dlYPred] = dlfeval(@modelGradients,dlnetEncoder,d

        % Update network learnable parameters using adamupdate.
        [dlnetEncoder, trailingAvgEncoder, trailingAvgSqEncoder] = adamupdate(dlnetEncoder,gradie
            iteration,learnRate,gradientDecayFactor,squaredGradientDecayFactor);

        [dlnetDecoder, trailingAvgDecder, trailingAvgSqDecoder] = adamupdate(dlnetDecoder,gradien
            iteration,learnRate,gradientDecayFactor,squaredGradientDecayFactor);

        % Generate translation for plot.
        if iteration == 1 || mod(iteration,10) == 0
            strGerman = ind2str(dlX(:,1,:),encGerman);
            strEnglish = ind2str(dlT(:,1,:),encEnglish,Mask=maskT);
            strTranslated = ind2str(dlYPred(:,1,:),encEnglish);
        end

        % Display training progress.
        D = duration(0,0,toc(start),Format="hh:mm:ss");
        loss = double(gather(extractdata(loss)));
        addpoints(lineLossTrain,iteration,loss)
        title( ...
            "Epoch: " + epoch + ", Elapsed: " + string(D) + newline + ...
            "Source: " + strGerman + newline + ...
            "Target: " + strEnglish + newline + ...
            "Training Translation: " + strTranslated)

        drawnow

        % Save best network.
        if loss < lossMin
            lossMin = loss;
            netBest.dlnetEncoder = dlnetEncoder;
```

```
            netBest.dlnetDecoder = dlnetDecoder;
            netBest.loss = loss;
            netBest.iteration = iteration;
            netBest.D = D;
        end
    end

    % Shuffle.
    shuffle(mbq);
end
```



The plot shows two translations of the source text. The target is the target translation provided by the training data that the network attempts to reproduce. The training translation is the predicted translation, which uses information from the target text via the scheduled sampling mechanism.

Add the word encodings to the `netBest` structure and save the structure in a MAT file.

```
netBest.encGerman = encGerman;
netBest.encEnglish = encEnglish;

D = datestr(now,'yyyy_mm_dd__HH_MM_SS');
filename = "dlnet_best__" + D + ".mat";
save(filename,"netBest");
```

Extract the best network from `netBest`.

```
dlnetEncoder = netBest.dlnetEncoder;
dlnetDecoder = netBest.dlnetDecoder;
```

Test Model

To evaluate the quality of the translations, use the BiLingual Evaluation Understudy (BLEU) scoring algorithm [4] on page 4-0 .

Translate the test data using the `translateText` function listed at the end of the example.

```
strTranslatedTest = translateText(dlnetEncoder,dlnetDecoder,encGerman,encEnglish,dataTest.Source)
```

View a random selection of the test source text, target text, and predicted translations in a table.

```
numObservationsTest = size(dataTest,1);
idx = randperm(numObservationsTest,8);
tbl = table;
tbl.Source = dataTest.Source(idx);
tbl.Target = dataTest.Target(idx);
tbl.Translated = strTranslatedTest(idx)
```

```
tbl=8×3 table
                    Source                                Target                          Translated
    ____________________________________    ____________________________    ____________________

    "Er sieht krank aus."                   "He seems ill."                 "he looks sick ."
    "Ich werde das Buch holen."             "I'll get the book."            "i'll get the book .
    "Ruhst du dich jemals aus?"             "Do you ever rest?"             "do you look out of ?
    "Was willst du?"                        "What are you after?"           "what do you want wan
    "Du hast keinen Beweis."                "You have no proof."            "you have no proof .
    "Macht es, wann immer ihr wollt."       "Do it whenever you want."      "do it you like it .
    "Tom ist gerade nach Hause gekommen."   "Tom has just come home."       "tom just came home ho
    "Er lügt nie."                          "He never tells a lie."         "he never lie lies .
```

To evaluate the quality of the translations using the BLEU similarity score, first preprocess the text data using the same steps as for training. Specify empty start and stop tokens, as these are not used in the translation.

```
candidates = preprocessText(strTranslatedTest,StartToken="",StopToken="");
references = preprocessText(dataTest.Target,StartToken="",StopToken="");
```

The `bleuEvaluationScore` function, by default, evaluates the similarity scores by comparing n-grams of length one through four (multiword phrases with four or fewer words or single words). If the candidate or reference documents have fewer than four tokens, then the resulting BLEU evaluation score is zero. To ensure that `bleuEvaluationScore` returns nonzero scores for these short candidate documents, set the n-gram weights to a vector with fewer elements than the number of words in `candidate`.

Determine the length of the shortest candidate document.

```
minLength = min([doclength(candidates); doclength(references)])
```

```
minLength = 2
```

If the shortest document has fewer than four tokens, then set the n-gram weights to a vector with a length matching the shortest document with equal weights that sum to one. Otherwise, specify n-gram weights of `[0.25 0.25 0.25 0.25]`. Note that if `minLength` is `1` (and consequently the n-gram weights is also `1`), then the `bleuEvaluationScore` function can return less meaningful results as it only compares individual words (unigrams) and does not compare any n-grams (multiword phrases).

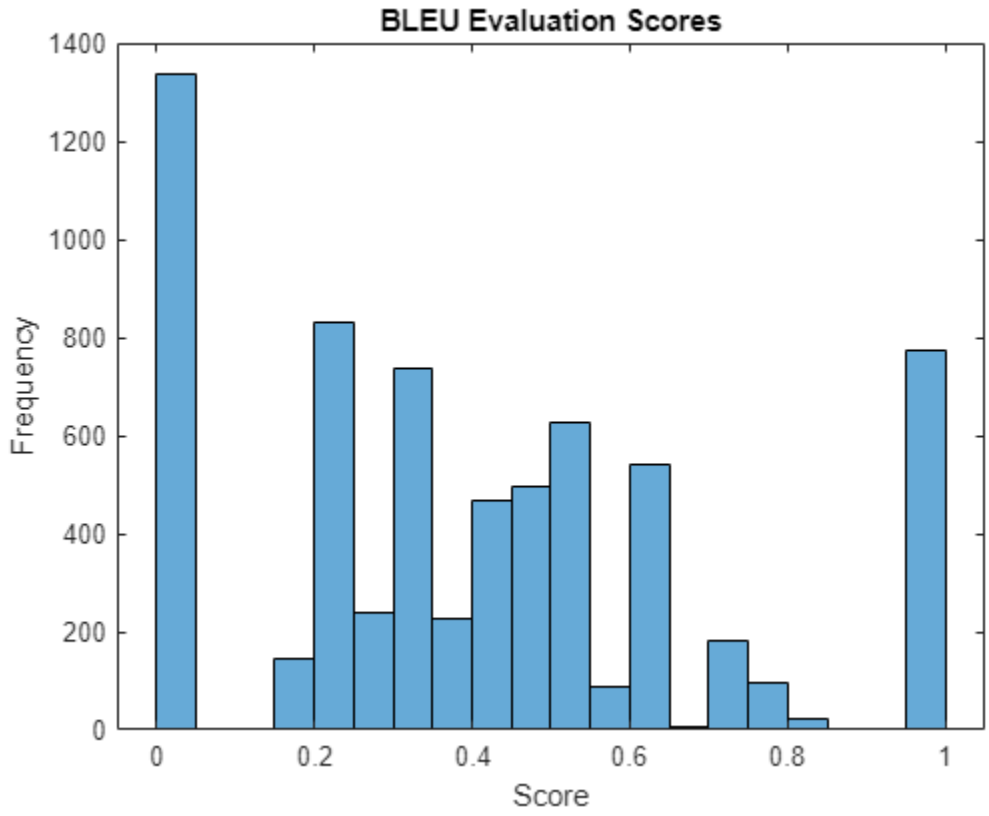
```
if minLength < 4
    ngramWeights = ones(1,minLength) / minLength;
else
    ngramWeights = [0.25 0.25 0.25 0.25];
end
```

Calculate the BLEU evaluation scores by iterating over the translations and using the `bleuEvaluationScore` function.

```
for i = 1:numObservationsTest
    score(i) = bleuEvaluationScore(candidates(i),references(i),NgramWeights=ngramWeights);
end
```

Visualize the BLEU evaluation scores in a histogram.

```
figure
histogram(score);
title("BLEU Evaluation Scores")
xlabel("Score")
ylabel("Frequency")
```



View a table of some of the best translations.

```
[~,idxSorted] = sort(score,"descend");
idx = idxSorted(1:8);
tbl = table;
tbl.Source = dataTest.Source(idx);
```

```
tbl.Target = dataTest.Target(idx);
tbl.Translated = strTranslatedTest(idx)
```

```
tbl=8×3 table
             Source                Target        Translated
    ___________________________    ___________    ____________

    "Legen Sie sich hin!"          "Lie low."     "lie low ."
    "Ich gähnte."                  "I yawned."    "i yawned ."
    "Küsse Tom!"                   "Kiss Tom."    "kiss tom ."
    "Küssen Sie Tom!"              "Kiss Tom."    "kiss tom ."
    "Nimm Tom."                    "Take Tom."    "take tom ."
    "Komm bald."                   "Come soon."   "come soon ."
    "Ich habe es geschafft."       "I made it."   "i made it ."
    "Ich sehe Tom."                "I see Tom."   "i see tom ."
```

View a table of some of the worst translations.

```
idx = idxSorted(end-7:end);
tbl = table;
tbl.Source = dataTest.Source(idx);
tbl.Target = dataTest.Target(idx);
tbl.Translated = strTranslatedTest(idx)
```

```
tbl=8×3 table
                                  Source                                                  Target
    ______________________________________________________________________    ____________________________

    "Diese Schnecken kann man essen."                                         "These snails are edible."
    "Sie stehen noch zu Verfügung."                                           "They're still available."
    "Diese Schraube passt zu dieser Mutter."                                  "This bolt fits this nut."
    "Diese Puppe gehört mir."                                                 "This doll belongs to me."
    "Das ist eine japanische Puppe."                                          "This is a Japanese doll."
    "Das ist eine Kreuzung, an der alle Fahrzeuge anhalten müssen."           "This is a four-way stop."
    "Diese Sendung ist eine Wiederholung."                                    "This program is a rerun."
    "Die heutige Folge ist eine Wiederholung."                                "Today's show is a rerun."
```

Generate Translations

Generate translations for new data using the `translateText` function.

```
strGermanNew = [
    "Wie geht es Dir heute?"
    "Wie heißen Sie?"
    "Das Wetter ist heute gut."];
```

Translate the text using the `translateText`, function listed at the end of the example.

```
strTranslatedNew = translateText(dlnetEncoder,dlnetDecoder,encGerman,encEnglish,strGermanNew)
```

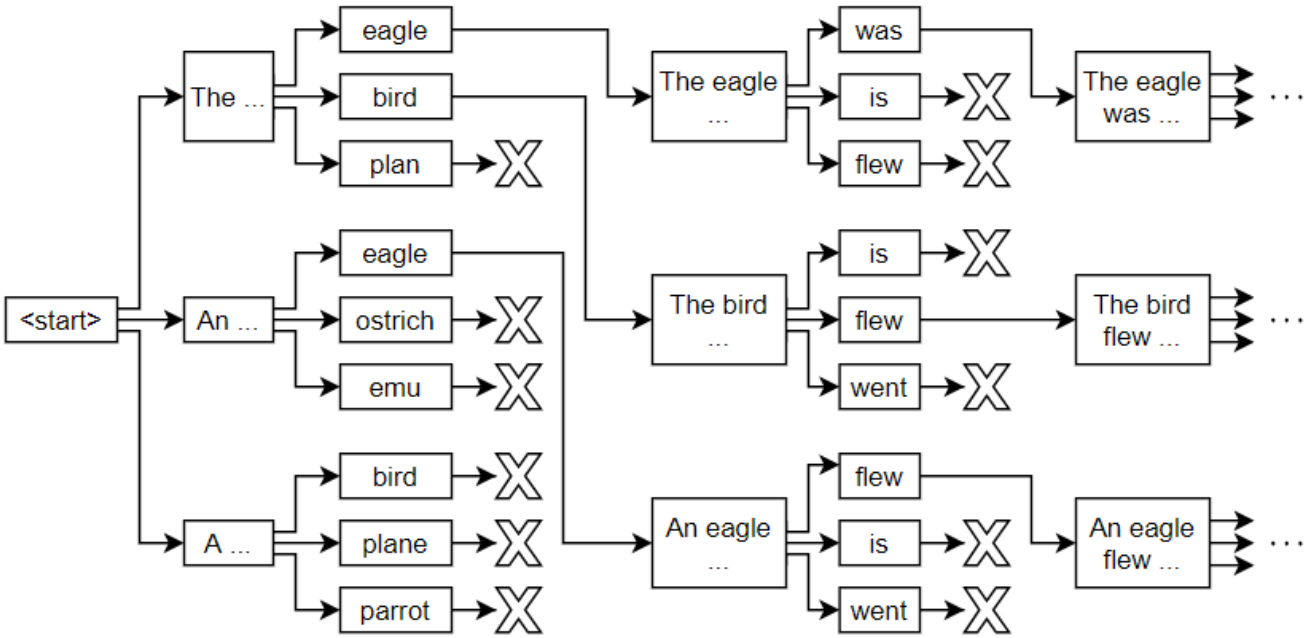
```
strTranslatedNew = 3×1 string
    "how do you feel today ?"
    "what's your your name ? ? ? ? ?"
    "the is is today . . today . ."
```

Prediction Functions

Beam Search Function

Beam search is technique for exploring different combinations of time-step predictions to help find the best prediction sequence. The premise of beam search is for each time-step prediction, identify the top K predictions for some positive integer K (also known as the *beam index* or the *beam width*), and maintain the top K predicted sequences so far at each time step.

This diagram shows the structure of an example beam search with beam index $K = 3$. For each prediction, the top three sequences are maintained.



The `beamSearch` function takes as input the input data `dlX`, the encoder and decoder networks, and the target word encoding, and returns the predicted translated words using the beam search algorithm with a beam index of 3 and a maximum sequence length of 10. You can also specify optional arguments using name-value arguments:

- `BeamIndex` — Beam index. The default is 3.
- `MaxNumWords` — Maximum sequence length. The default is 10.

```
function str = beamSearch(dlX,dlnetEncoder,dlnetDecoder,encEnglish,args)

% Parse input arguments.
arguments
    dlX
    dlnetEncoder
    dlnetDecoder
    encEnglish

    args.BeamIndex = 3;
    args.MaxNumWords = 10;
end
```

```
beamIndex = args.BeamIndex;
maxNumWords = args.MaxNumWords;
startToken = "<start>";
stopToken = "<stop>";

% Encoder predictions.
[dlZ, hiddenState, cellState] = predict(dlnetEncoder,dlX);

% Initialize context.
miniBatchSize = size(dlX,2);
numHiddenUnits = size(dlZ,1);
context = zeros([numHiddenUnits miniBatchSize],"like",dlZ);
context = dlarray(context,"CB");

% Initialize candidates.
candidates = struct;
candidates.Words = startToken;
candidates.Score = 0;
candidates.StopFlag = false;
candidates.HiddenState = hiddenState;
candidates.CellState = cellState;

% Loop over words.
t = 0;
while t < maxNumWords
    t = t + 1;

    candidatesNew = [];

    % Loop over candidates.
    for i = 1:numel(candidates)

        % Stop generating when stop token is predicted.
        if candidates(i).StopFlag
            continue
        end

        % Candidate details.
        words = candidates(i).Words;
        score = candidates(i).Score;
        hiddenState = candidates(i).HiddenState;
        cellState = candidates(i).CellState;

        % Predict next token.
        decoderInput = word2ind(encEnglish,words(end));
        decoderInput = dlarray(decoderInput,"CBT");

        [dlYPred,context,hiddenState,cellState] = predict(dlnetDecoder,decoderInput,hiddenState,
            Outputs=["softmax" "context" "lstm2/hidden" "lstm2/cell"]);

        % Find top predictions.
        [scoresTop,idxTop] = maxk(extractdata(dlYPred),beamIndex);
        idxTop = gather(idxTop);

        % Loop over top predictions.
        for j = 1:beamIndex
            candidate = struct;
```

```
            % Determine candidate word and score.
            candidateWord = ind2word(encEnglish,idxTop(j));
            candidateScore = scoresTop(j);

            % Set stop translating flag.
            if candidateWord == stopToken
                candidate.StopFlag = true;
            else
                candidate.StopFlag = false;
            end

            % Update candidate details.
            candidate.Words = [words candidateWord];
            candidate.Score = score + log(candidateScore);
            candidate.HiddenState = hiddenState;
            candidate.CellState = cellState;

            % Add to new candidates.
            candidatesNew = [candidatesNew candidate];
        end
    end

    % Get top candidates.
    [~,idx] = maxk([candidatesNew.Score],beamIndex);
    candidates = candidatesNew(idx);

    % Stop predicting when all candidates have stop token.
    if all([candidates.StopFlag])
        break
    end
end

% Get top candidate.
words = candidates(1).Words;

% Convert to string scalar.
words(ismember(words,[startToken stopToken])) = [];
str = join(words);

end
```

Translate Text Function

The `translateText` function takes as input the encoder and decoder networks, an input string, and source and target word encodings and returns the translated text.

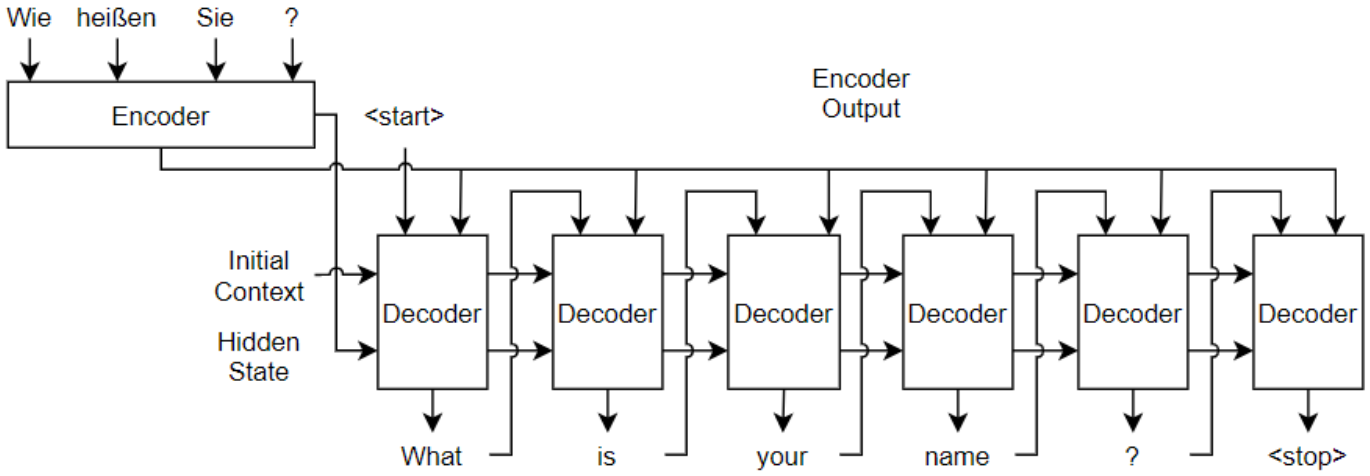
```
function strTranslated = translateText(dlnetEncoder,dlnetDecoder,encGerman,encEnglish,strGerman,a

% Parse input arguments.
arguments
    dlnetEncoder
    dlnetDecoder
    encGerman
    encEnglish
    strGerman

    args.BeamIndex = 3;
```

```
end

beamIndex = args.BeamIndex;

% Preprocess text.
documentsGerman = preprocessText(strGerman);
dlX = preprocessPredictors(documentsGerman,encGerman);
dlX = dlarray(dlX,"CTB");

% Loop over observations.
numObservations = numel(strGerman);
strTranslated = strings(numObservations,1);
for n = 1:numObservations

    % Translate text.
    strTranslated(n) = beamSearch(dlX(:,n,:),dlnetEncoder,dlnetDecoder,encEnglish,BeamIndex=beam
end

end
```

Model Functions

Model Gradients Function

The `modelGradients` function takes as input the encoder network, decoder network, mini-batches of predictors `dlX`, targets `dlT`, padding mask corresponding to the targets `maskT`, and ϵ value for scheduled sampling. The function returns the gradients of the loss with respect to the learnable parameters in the networks `gradientsE` and `gradientsD`, the corresponding loss, and the decoder predictions `dlYPred` encoded as sequences of one-hot vectors.



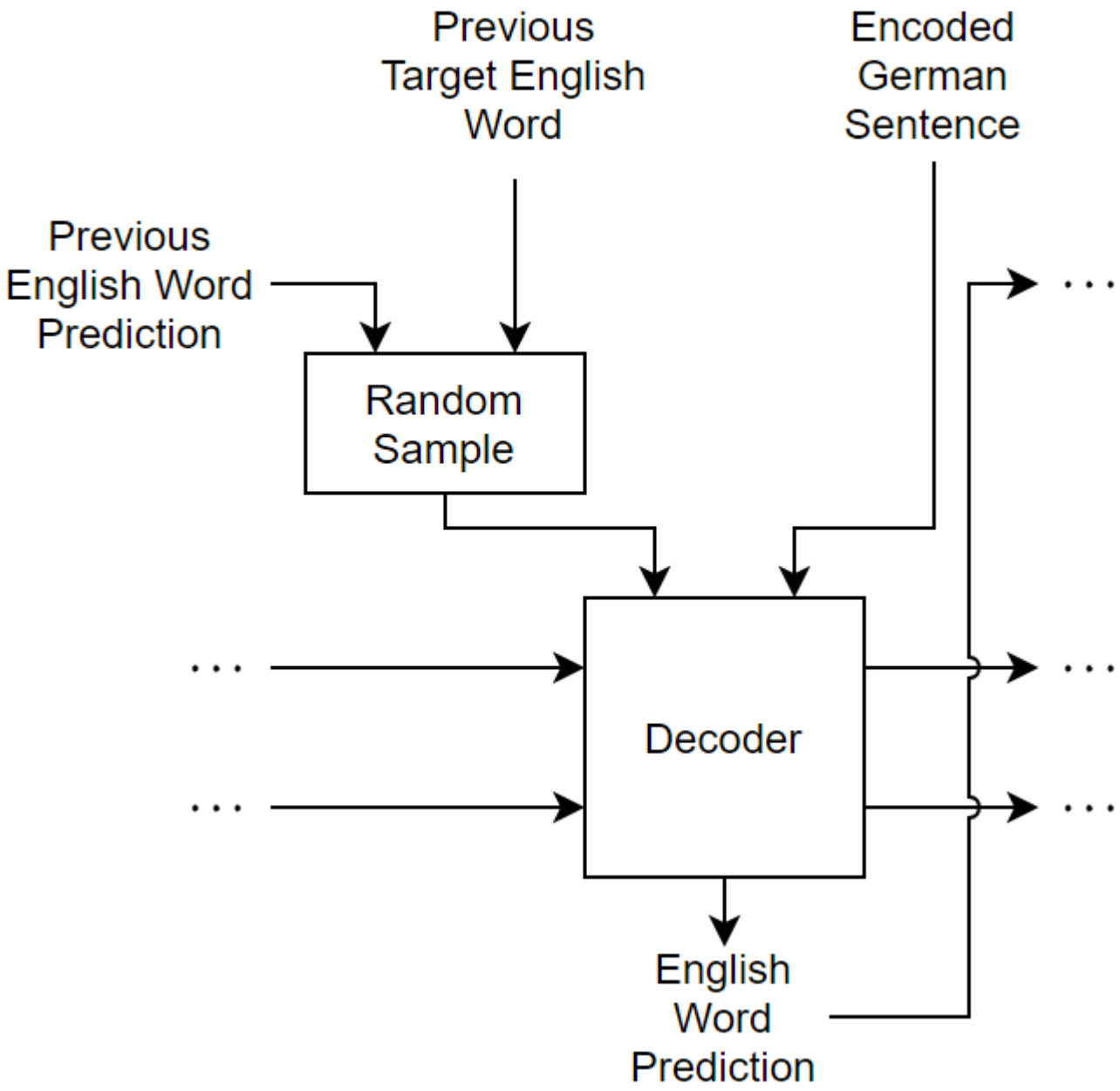

```
function [gradientsE,gradientsD,loss,dlYPred] = modelGradients(dlnetEncoder,dlnetDecoder,dlX,dlT

% Forward through encoder.
[dlZ, hiddenState, cellState] = forward(dlnetEncoder,dlX);

% Decoder output.
dlY = decoderPredictions(dlnetDecoder,dlZ,dlT,hiddenState,cellState,decoderInput,epsilon);

% Sparse cross-entropy loss.
loss = sparseCrossEntropy(dlY,dlT,maskT);
```

```
% Update gradients.
[gradientsE,gradientsD] = dlgradient(loss,dlnetEncoder.Learnables,dlnetDecoder.Learnables);

% For plotting, return loss normalized by sequence length.
sequenceLength = size(dlT,3);
loss = loss ./ sequenceLength;

% For plotting example translations, return the decoder output.
dlYPred = onehotdecode(dlY,1:size(dlY,1),1,"single");

end
```

Decoder Predictions Function

The `decoderPredictions` function takes as input, the decoder network, the encoder output `dlZ`, the targets `dlT`, the decoder input hidden and cell state values, and the ϵ value for scheduled sampling.

To stabilize training, you can randomly use the target values as inputs to the decoder. In particular, you can adjust the probability used to inject the target values as training progresses. For example, you can train using the target values at a much higher rate at the start of training, then decay the probability such that towards the end of training the model uses only the previous predictions. This technique is known as *scheduled sampling* [2] on page 4-0 . This diagram shows the sampling mechanism incorporated into one time step of a decoder prediction.



The decoder makes predictions one time step at a time. At each time step, the input is randomly selected according to the ϵ value for scheduled sampling. In particular, the function uses the target value as input with probability ϵ and uses the previous prediction otherwise.

```
function dlY = decoderPredictions(dlnetDecoder,dlZ,dlT,hiddenState,cellState,decoderInput,epsilo

% Initialize context.
numHiddenUnits = size(dlZ,1);
miniBatchSize = size(dlZ,2);
context = zeros([numHiddenUnits miniBatchSize],"like",dlZ);
context = dlarray(context,"CB");
```

```
% Initialize output.
idx = (dlnetDecoder.Learnables.Layer == "fc" & dlnetDecoder.Learnables.Parameter=="Bias");
numClasses = numel(dlnetDecoder.Learnables.Value{idx});
sequenceLength = size(dlT,3);
dlY = zeros([numClasses miniBatchSize sequenceLength],"like",dlZ);
dlY = dlarray(dlY,"CBT");

% Forward start token through decoder.
[dlY(:,:,1),context,hiddenState,cellState] = forward(dlnetDecoder,decoderInput,hiddenState,cellS

% Loop over remaining time steps.
for t = 2:sequenceLength

    % Scheduled sampling. Randomly select previous target or previous
    % prediction.
    if rand < epsilon
        % Use target value.
        decoderInput = dlT(:,:,t-1);
    else
        % Use previous prediction.
        [~,dlYhat] = max(dlY(:,:,t-1),[],1);
        decoderInput = dlYhat;
    end

    % Forward through decoder.
    [dlY(:,:,t),context,hiddenState,cellState] = forward(dlnetDecoder,decoderInput,hiddenState,ce
end

end
```

Sparse Cross-Entropy Loss

The `sparseCrossEntropy` function calculates the cross-entropy loss between the predictions `dlY` and targets `dlT` with the target mask `maskT`, where `dlY` is an array of probabilities and `dlT` is encoded as a sequence of integer values.

```
function loss = sparseCrossEntropy(dlY,dlT,maskT)

% Initialize loss.
[~,miniBatchSize,sequenceLength] = size(dlY);
loss = zeros([miniBatchSize sequenceLength],"like",dlY);

% To prevent calculating log of 0, bound away from zero.
precision = underlyingType(dlY);
dlY(dlY < eps(precision)) = eps(precision);

% Loop over time steps.
for n = 1:miniBatchSize
    for t = 1:sequenceLength
        idx = dlT(1,n,t);
        loss(n,t) = -log(dlY(idx,n,t));
    end
end

% Apply masking.
maskT = squeeze(maskT);
loss = loss .* maskT;
```

```
% Calculate sum and normalize.
loss = sum(loss,"all");
loss = loss / miniBatchSize;

end
```

Preprocessing Functions

Text Preprocessing Function

The `preprocessText` function preprocesses the input text for translation by converting the text to lowercase, adding start and stop tokens, and tokenizing.

```
function documents = preprocessText(str,args)

arguments
    str
    args.StartToken = "<start>";
    args.StopToken = "<stop>";
end

startToken = args.StartToken;
stopToken = args.StopToken;

str = lower(str);
str = startToken + str + stopToken;
documents = tokenizedDocument(str,CustomTokens=[startToken stopToken]);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses tokenized documents for training. The function encodes mini-batches of documents as sequences of numeric indices and pads the sequences to have the same length.

```
function [XSource,XTarget,mask,decoderInput] = preprocessMiniBatch(dataSource,dataTarget,encGerma

documentsGerman = cat(1,dataSource{:});
XSource = preprocessPredictors(documentsGerman,encGerman);

documentsEnglish = cat(1,dataTarget{:});
sequencesTarget = doc2sequence(encEnglish,documentsEnglish,PaddingDirection="none");

[XTarget,mask] = padsequences(sequencesTarget,2,PaddingValue=1);

decoderInput = XTarget(:,1,:);
XTarget(:,1,:) = [];
mask(:,1,:) = [];

end
```

Predictors Preprocessing Function

The `preprocessPredictors` function preprocesses source documents for training or prediction. The function encodes an array of tokenized documents as sequences of numeric indices.

```
function XSource = preprocessPredictors(documentsGerman,encGerman)
```

```
sequencesSource = doc2sequence(encGerman,documentsGerman,PaddingDirection="none");
XSource = padsequences(sequencesSource,2);

end
```

Bibliography

1. Chorowski, Jan, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. “Attention-Based Models for Speech Recognition.” Preprint, siubmitted June 24, 2015. https://arxiv.org/abs/1506.07503.
2. Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks.” Preprint, submitted September 23, 2015. https://arxiv.org/abs/1506.03099.
3. Amodei, Dario, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper et al. "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin." In <cite>Proceedings of Machine Learning Research</cite> 48 (2016): 173–182.
4. Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "BLEU: A Method for Automatic Evaluation of Machine Translation." In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (2002): 311–318.

See Also

`dlarray` | `dlfeval` | `dlgradient` | `lstm` | `minibatchqueue`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence Classification Using 1-D Convolutions” on page 4-9
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “List of Deep Learning Layers” on page 1-21
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225

Predict and Update Network State in Simulink

This example shows how to predict responses for a trained recurrent neural network in Simulink® by using the `Stateful Predict` block. This example uses a pretrained long short-term memory (LSTM) network.

Load Pretrained Network

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

```
ans =

  5x1 Layer array with layers:

     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load Test Data

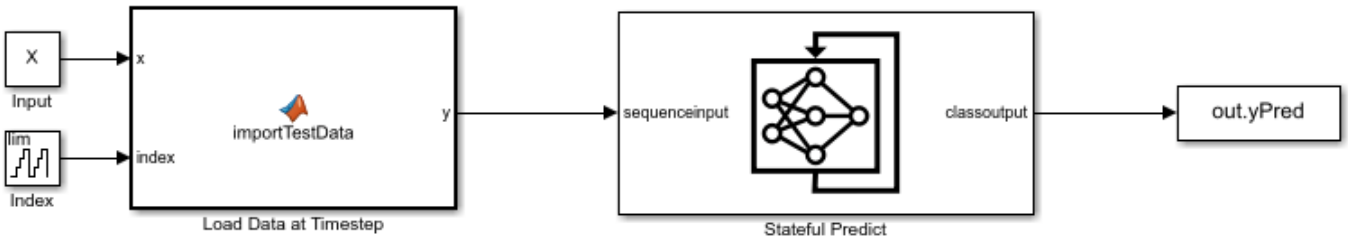
Load the Japanese Vowels test data. `XTest` is a cell array containing 370 sequences of dimension 12 of varying length. `YTest` is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
X = XTest{94};
numTimeSteps = size(X,2);
```

Simulink Model for Predicting Responses

The Simulink model for predicting responses contains a `Stateful Predict` block to predict the scores and `MATLAB Function` blocks to load the input data sequence over the time steps.

```
open_system('StatefulPredictExample');
```



Configure Model for Simulation

Set the model configuration parameters for the input blocks and the `Stateful Predict` block.

```
set_param('StatefulPredictExample/Input','Value','X');
set_param('StatefulPredictExample/Index','uplimit','numTimeSteps-1');
set_param('StatefulPredictExample/Stateful Predict','NetworkFilePath','JapaneseVowelsNet.mat');
set_param('StatefulPredictExample', 'SimulationMode', 'Normal');
```

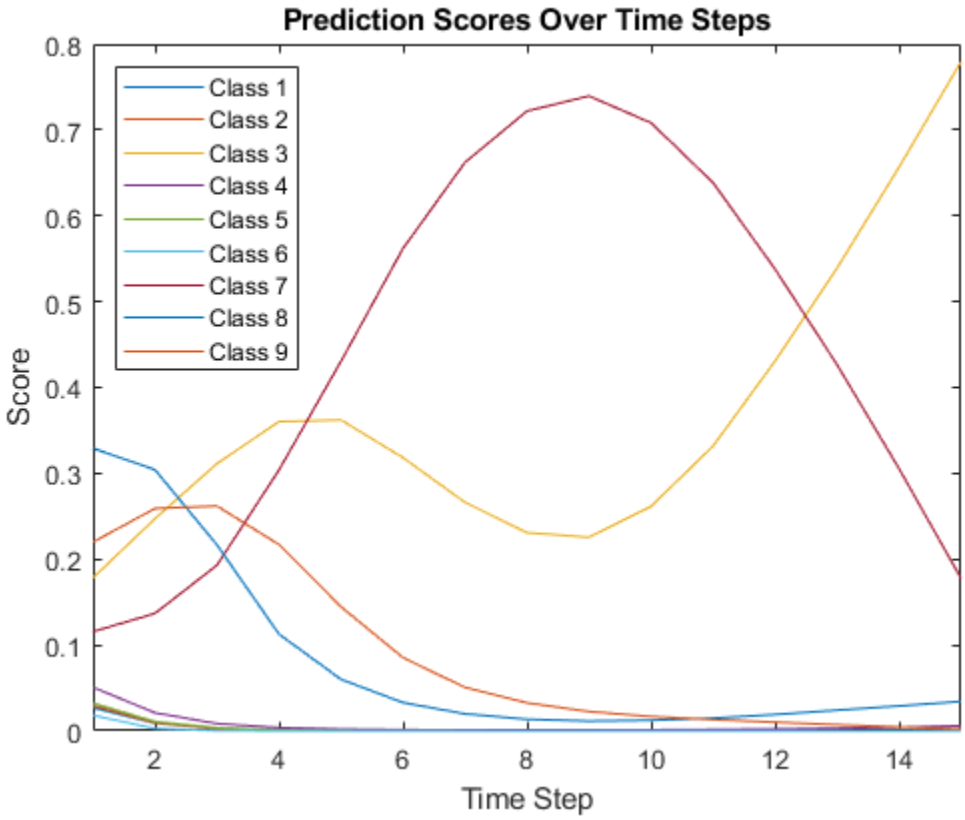
Run the Simulation

To compute responses for the `JapaneseVowelsNet` network, run the simulation. The prediction scores are saved in the MATLAB® workspace.

```
out = sim('StatefulPredictExample');
```

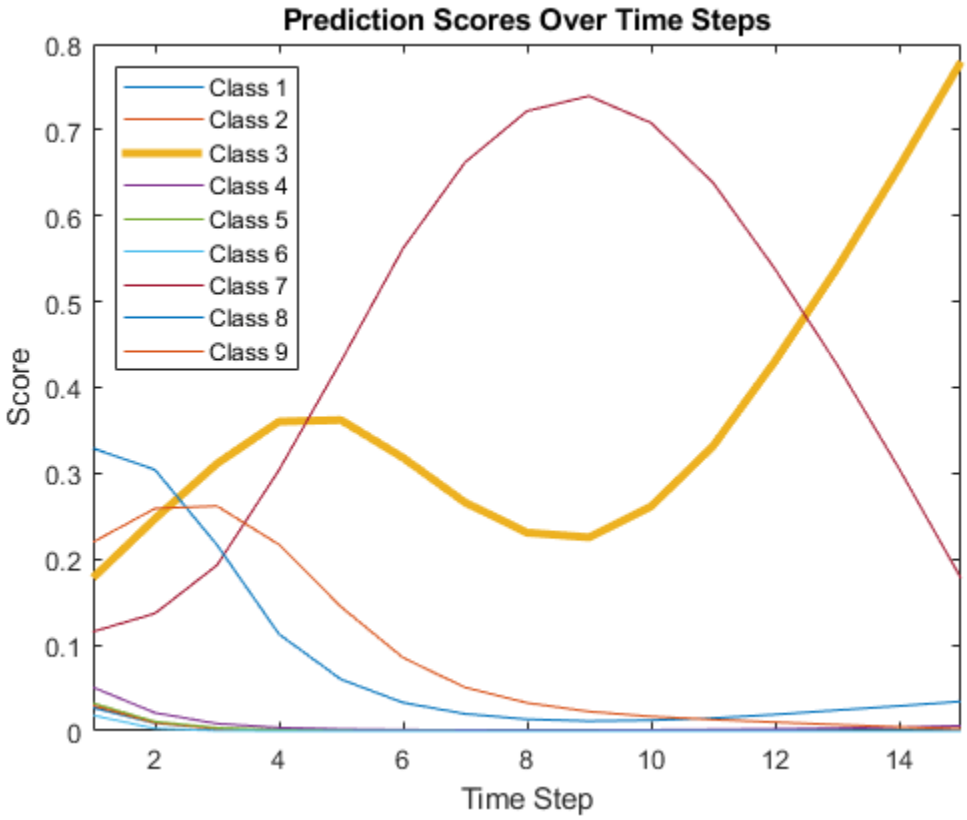
Plot the prediction scores. The plot shows how the prediction scores change between time steps.

```
scores = squeeze(out.yPred.Data(:,:,1:numTimeSteps));

classNames = string(net.Layers(end).Classes);
figure
lines = plot(scores');
xlim([1 numTimeSteps])
legend("Class " + classNames,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
title("Prediction Scores Over Time Steps")
```



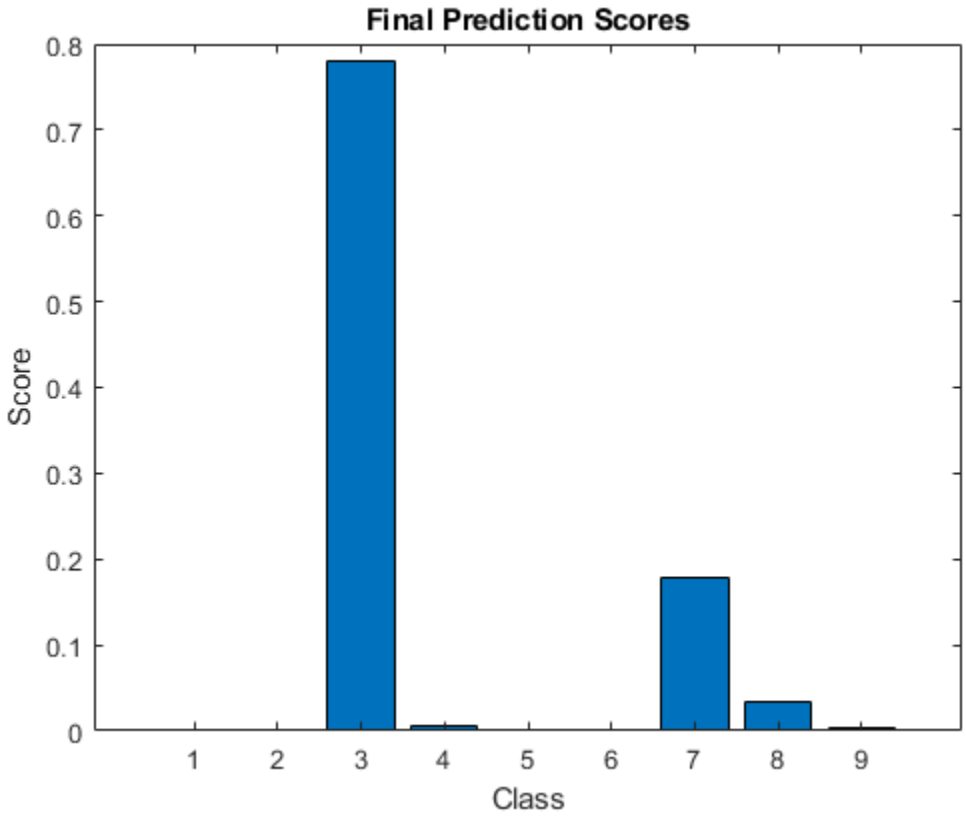
Highlight the prediction scores over time steps for the correct class.

```
trueLabel = YTest(94);
lines(trueLabel).LineWidth = 3;
```



Display the final time step prediction in a bar chart.

```
figure
bar(scores(:,end))
title("Final Prediction Scores")
xlabel("Class")
ylabel("Score")
```



References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

Stateful Predict | Stateful Classify | Predict | Image Classifier

Related Examples

- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Classify and Update Network State in Simulink” on page 4-249
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Classify and Update Network State in Simulink

This example shows how to classify data for a trained recurrent neural network in Simulink® by using the `Stateful Classify` block. This example uses a pretrained long short-term memory (LSTM) network.

Load Pretrained Network

Load `JapaneseVowelsNet`, a pretrained long short-term memory (LSTM) network trained on the Japanese Vowels data set as described in [1] and [2]. This network was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

```
ans =

  5x1 Layer array with layers:

     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load Test Data

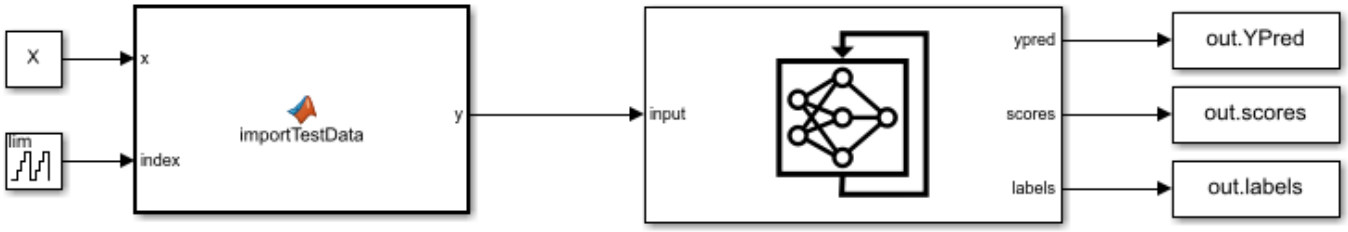
Load the Japanese Vowels test data. `XTest` is a cell array containing 370 sequences of dimension 12 of varying length. `YTest` is a categorical vector of labels "1","2",..."9", which correspond to the nine speakers.

```
[XTest,YTest] = japaneseVowelsTestData;
X = XTest{94};
numTimeSteps = size(X,2);
```

Simulink Model for Classifying Data

The Simulink model for classifying data contains a `Stateful Classify` block to predict the labels and `MATLAB Function` blocks to load the input data sequence over the time steps.

```
open_system('StatefulClassifyExample');
```



Configure Model for Simulation

Set the model configuration parameters for the input blocks and the `Stateful Classify` block.

```
set_param('StatefulClassifyExample/Input','Value','X');
set_param('StatefulClassifyExample/Index','uplimit','numTimeSteps-1');
set_param('StatefulClassifyExample/Stateful Classify','NetworkFilePath','JapaneseVowelsNet.mat')
set_param('StatefulClassifyExample','SimulationMode','Normal');
```

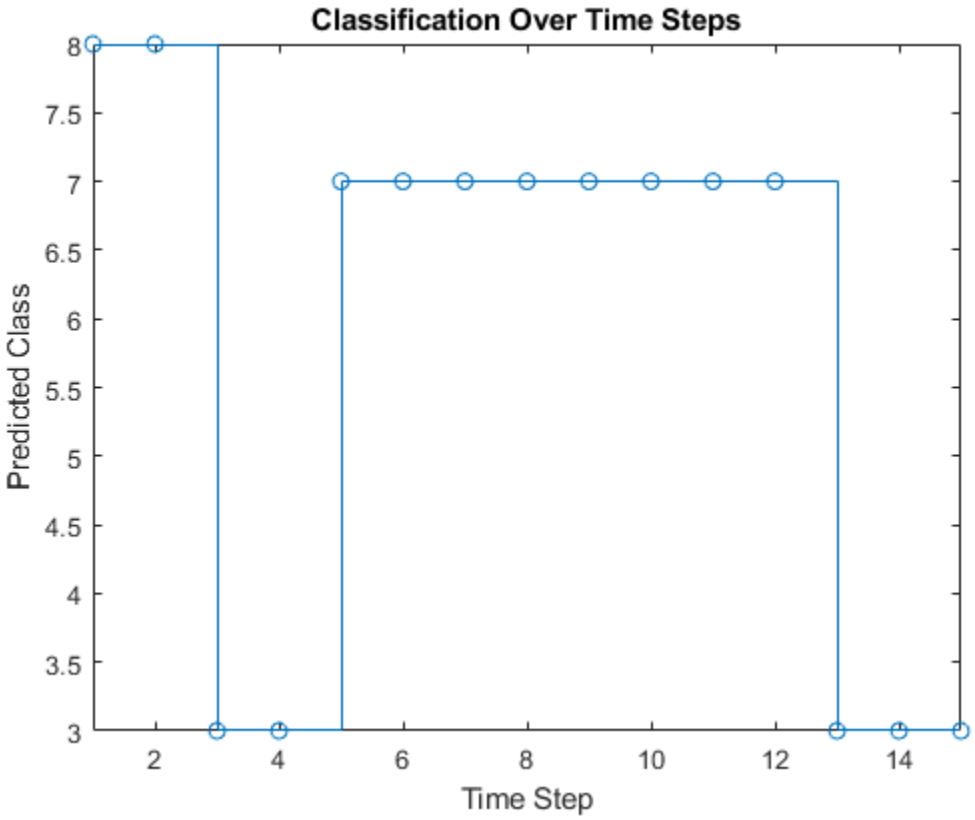
Run the Simulation

To compute responses for the `JapaneseVowelsNet` network, run the simulation. The prediction labels are saved in the MATLAB® workspace.

```
out = sim('StatefulClassifyExample');
```

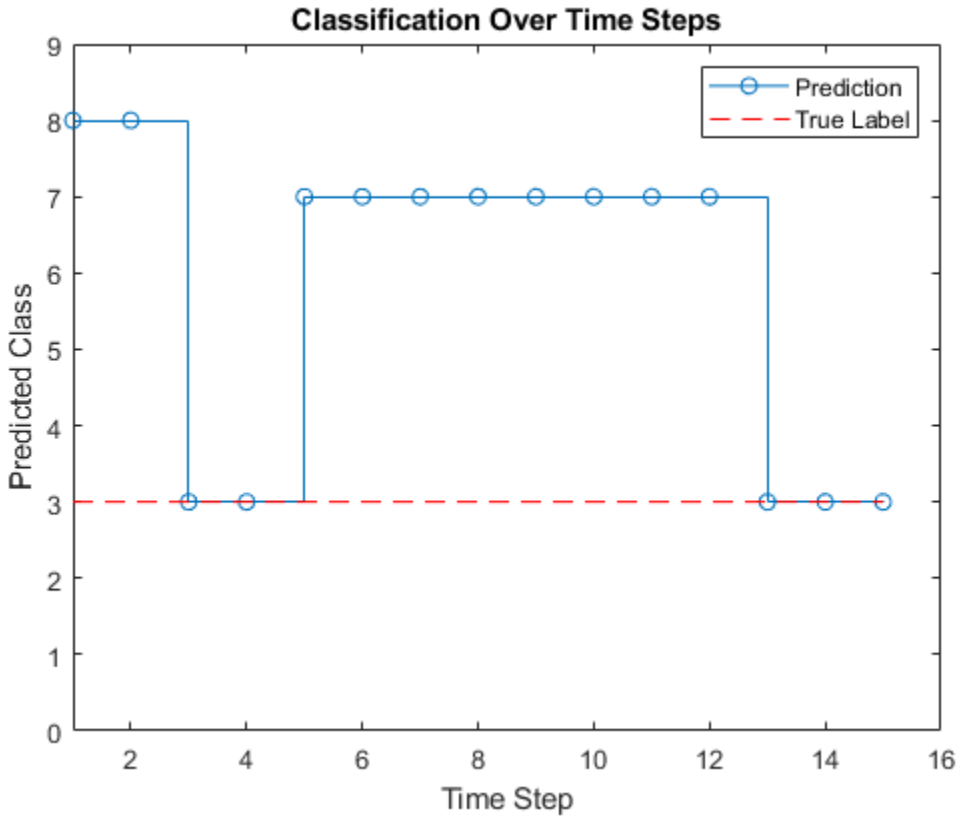
Plot the predicted labels in a stair plot. The plot shows how the predictions change between time steps.

```
labels = squeeze(out.YPred.Data(1:numTimeSteps,1));

figure
stairs(labels, '-o')
xlim([1 numTimeSteps])
xlabel("Time Step")
ylabel("Predicted Class")
title("Classification Over Time Steps")
```



Compare the predictions with the true label. Plot a horizontal line showing the true label of the observation.

```
trueLabel = double(YTest(94));
hold on
line([1 numTimeSteps],[trueLabel trueLabel], ...
    'Color','red', ...
    'LineStyle','--')
legend(["Prediction" "True Label"])
axis([1 numTimeSteps+1 0 9]);
```



References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

Stateful Predict | Stateful Classify | Predict | Image Classifier

Related Examples

- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Predict and Update Network State in Simulink” on page 4-244
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Time Series Prediction in Simulink Using Deep Learning Network

This example shows how to use an LSTM deep learning network inside a Simulink® model to predict the remaining useful life (RUL) of an engine. You include the network inside the Simulink model by using a `Stateful Predict` block, which predicts the RUL at every simulation step.

This example uses data from the *Turbofan Engine Degradation Simulation Data Set* as described in [1]. The example uses a trained LSTM network to predict the RUL of an engine (predictive maintenance), measured in cycles, given time series data representing various sensors in the engine. The data used to train the network contains simulated time series data for 100 engines. Each sequence has 17 features of varying length and corresponds to a full run to failure (RTF) instance. For more information on how to train the network, see “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47.

Download Data

Download and unzip the *Turbofan Engine Degradation Simulation Data Set* from the Prognostics Data Repository (NASA) [2].

Each time series represents a different engine. Each engine starts with unknown degrees of initial wear and manufacturing variation. The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure.

The data contains zip-compressed text files with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, and each column is a different variable. The columns correspond to the following:

- Column 1: Unit number
- Column 2: Time in cycles
- Columns 3–5: Operational settings
- Columns 6–26: Sensor measurements 1–21

```
filename = "CMAPSSData.zip";
dataFolder = "data";
unzip(filename,dataFolder)
```

Prepare Data

Load the data using the `processTurboFanDataTrain` helper function. The `processTurboFanDataTrain` function extracts the data from `filenamePredictors` and returns the cell array `XTrain`, which contains the training predictor data.

```
filenamePredictors = fullfile(dataFolder,"train_FD001.txt");
[XTrain] = processTurboFanDataTrain(filenamePredictors);
```

Remove Features with Constant Values

Since the network was trained with features that do not remain constant for all time steps, features with constant values for all time steps need to be removed for prediction. Find the rows of data that have the same minimum and maximum values, and remove the rows.

```
m = min([XTrain{:}],[],2);
M = max([XTrain{:}],[],2);
idxConstant = M == m;

for i = 1:numel(XTrain)
    XTrain{i}(idxConstant,:) = [];
end
```

Normalize Training Predictors

Normalize the training predictors to have zero mean and unit variance. To calculate the mean and standard deviation over all observations, concatenate the sequence data horizontally.

```
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);

for i = 1:numel(XTrain)
    XTrain{i} = (XTrain{i} - mu) ./ sig;
end
```

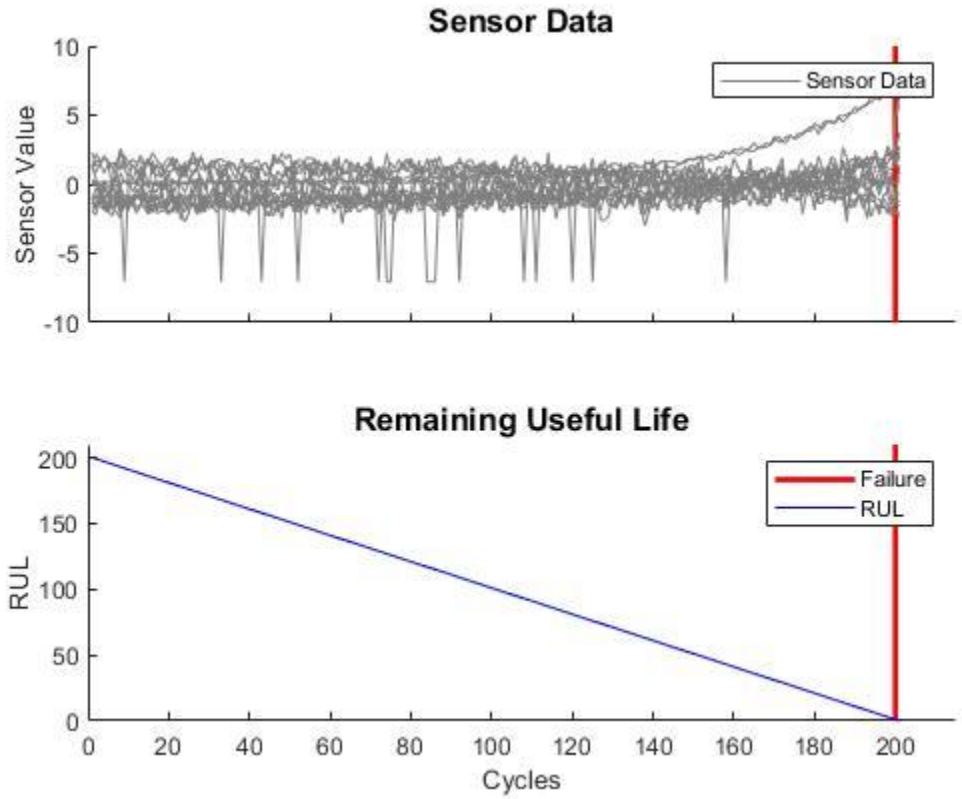
Extract data for one engine

In the Simulink model, we calculate the RUL for one engine only. In this example, we extract the 9th element of `XTrain` and store it in a variable named `SensorData`. You can choose any other engine from the `XTrain` cell array. `SensorData` is a double array of size 17-by-201. Every row corresponds to one feature and every column corresponds to the sensor readings at a given cycle.

```
SensorData = XTrain{9};
```

Simulink models have an associated simulation time, which in this example needs to be related to the engine cycles. For this reason, we define a timeseries named `EngineData`, which stores the sensor data as a timeseries object that can be loaded in the Simulink model. As the default simulation time in Simulink is `10.0` and the engine runs through 201 cycles, the `Time` field of `EngineData` needs to be an array of size 201-by-1 with values linearly increasing from 0 to 10.

```
Time = linspace(0,10,201)';
EngineData = timeseries(SensorData',Time);
```

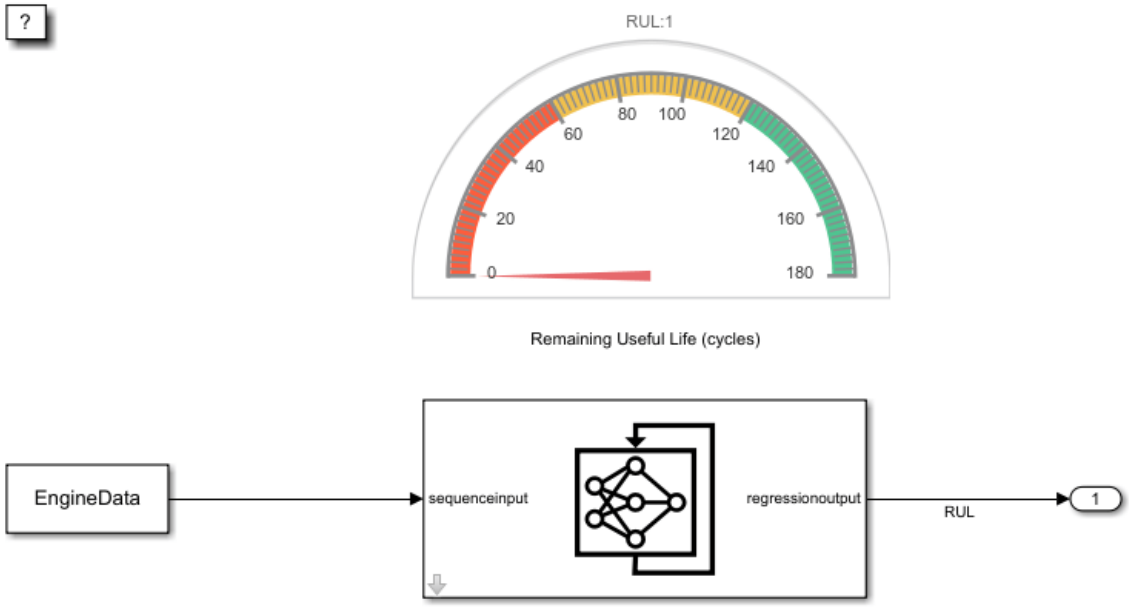


The top panel of this figure shows the sensor readings from each sensor at each cycle and the bottom panel shows the RUL of the engine in units of cycles. After 201 cycles, the engine stops operating. Note, if you select another engine from the `XTrain` data, then you need to adapt the `Time` field of `EngineData` accordingly, as each engine operates for a different number of cycles.

Simulink Model to Predict RUL

Load the Simulink model `RULPredictionLSTM.slx`.

```
modelName = 'RULPredictionLSTM';
open_system(modelName);
```



`EngineData` is loaded from the base workspace using a `From Workspace` block. In this example, the time step in `EngineData` is `0.05`. So, we set the sample time of the `From Workspace` block to 0.05. Hence, at the first step the block outputs the first row of `EngineData`, at the second step it outputs the second row - corresponding to the second engine cycle - and so on. If another engine is chosen from `XData`, then the sample time of the block needs to be updated accordingly.

```
set_param([modelName,'/From Workspace'],'SampleTime','0.05');
```

The `Stateful Predict` block loads the pretrained LSTM network in the `turbofanNet` MAT-file and returns the RUL at its output port. The `Stateful Predict` block updates the state of the network with every prediction, improving the prediction of the current RUL.The `Half Gauge` block shows the value of the calculated RUL (in units of engine cycles) during the simulation.

```
RUL_sigSpec = Simulink.HMI.SignalSpecification;
RUL_sigSpec.BlockPath = Simulink.BlockPath('RULPredictionLSTM/Stateful Predict');
set_param('RULPredictionLSTM/Half Gauge','Binding',RUL_sigSpec)
```

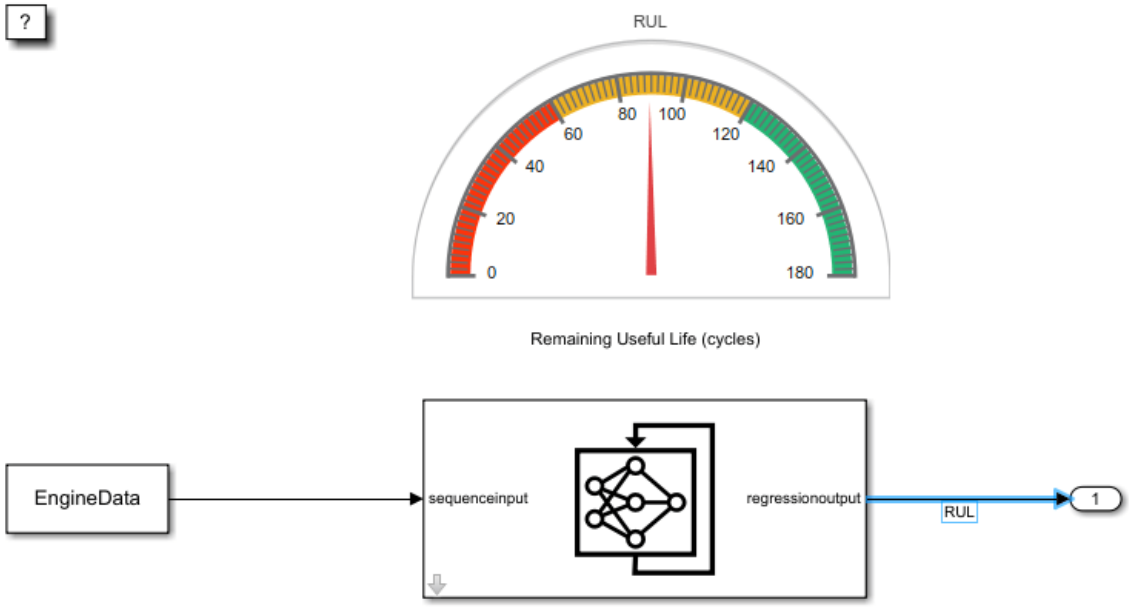
Run the Simulation

Because the simulation reads data from a MAT-file, it runs very quickly and can be difficult to follow. To slow down the simulation, set the Simulation Pacing option to 0.5.

```
set_param(modelName,'EnablePacing','on');
set_param(modelName,'PacingRate',0.5);
```

To compute the RUL, run the simulation.

```
sim(modelName);
```



The figure shows the model while it is running. The gauge shows the estimated RUL, corresponding in this case to 90 cycles. At the end of the simulation, the RUL is returned to the Base Workspace in the form of a single array, containing the values calculated at each simulation iteration.

You could integrate this system within a bigger framework, for example in a system that continuously monitors the status of an engine, and which adopts precautionary measures if the RUL falls below a given user-defined value.

References

1 Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pp. 1-9. IEEE, 2008.

2 Saxena, Abhinav, Kai Goebel. "Turbofan Engine Degradation Simulation Data Set." *NASA Ames Prognostics Data Repository* https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/, NASA Ames Research Center, Moffett Field, CA

See Also

Stateful Predict | Stateful Classify | Predict | Image Classifier

Related Examples

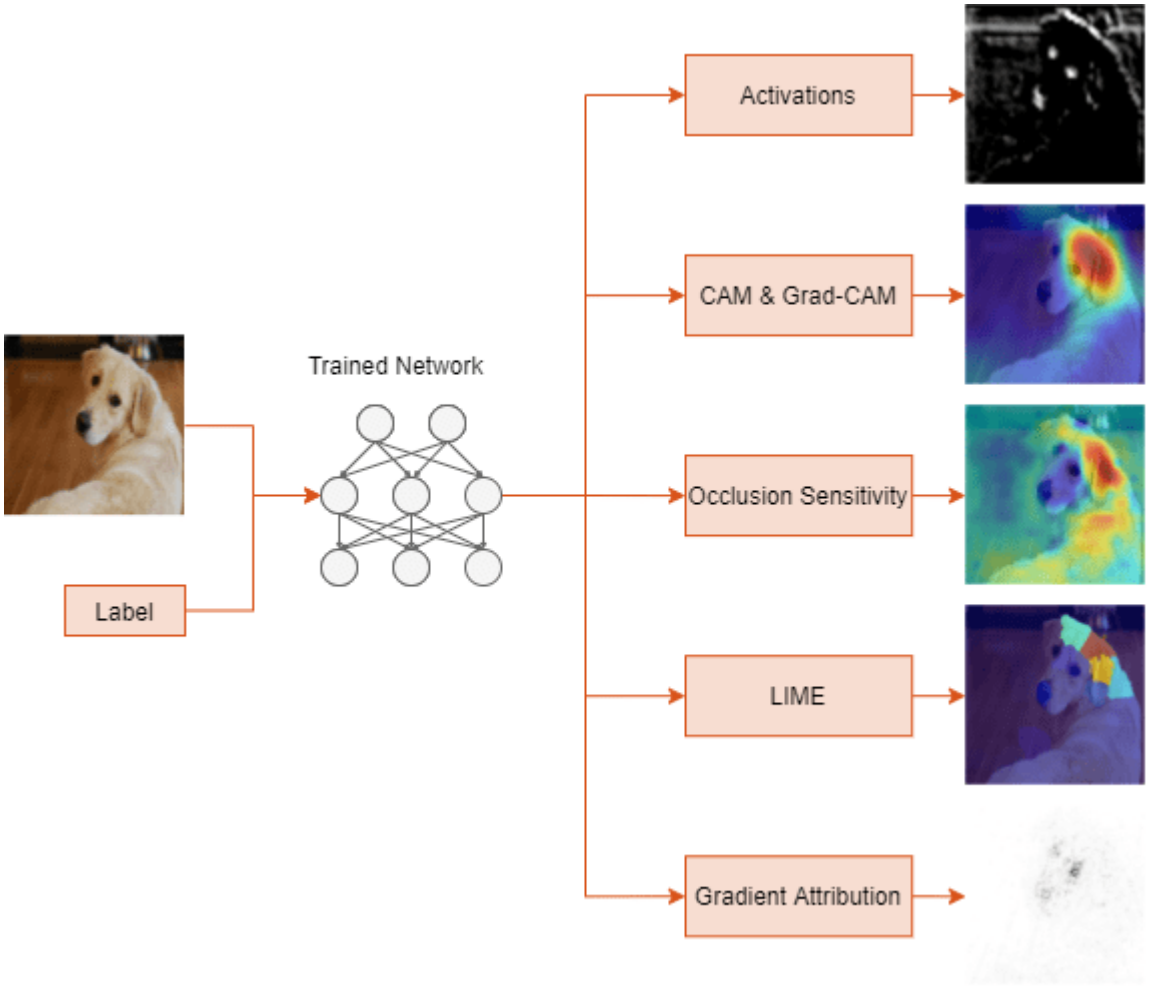
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Predict and Update Network State in Simulink” on page 4-244
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Deep Learning Tuning and Visualization

Explore Network Predictions Using Deep Learning Visualization Techniques

This example shows how to investigate network predictions using deep learning visualization techniques.

Deep learning networks are often described as "black boxes" because why a network makes a certain decision is not always obvious. You can use an interpretability technique to translate network behavior into output that a person can interpret. This interpretable output can then answer questions about the predictions of a network. This example focuses on visualization methods, which are interpretability techniques that explain network predictions using visual representations of what a network is “looking” at.



Load Pretrained Network

Load a pretrained image classification network. For this example, use GoogLeNet, a pretrained network that can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

```
net = googlenet;
```

Find the input size of the network and the class labels.

```
inputSize = net.Layers(1).InputSize(1:2);
classes = net.Layers(end).Classes;
```

Classify Image

Load a test image containing a picture of a golden retriever.

```
img = imread("sherlock.jpg");
img = imresize(img,inputSize);

figure
imshow(img)
```

Classify the image using the pretrained network.

```
[YPred,scores] = classify(net,img);
YPred
```

```
YPred = categorical
     golden retriever
```

The network correctly classifies the image as a golden retriever. Find the three classes with the highest scores.

```
[~,topIdx] = maxk(scores,3);
topScores = scores(topIdx)';
topClasses = classes(topIdx);
table(topClasses,topScores)
```

```
ans=3×2 table
        topClasses        topScores
    __________________    _________

    golden retriever       0.55419
```

```
Labrador retriever     0.39633
kuvasz                 0.02544
```

The classes with the top three scores are all dog breeds. The network outputs higher scores for the classes that share similar features with the true golden retriever class.

You can use visualization techniques to understand why the network classifies this image as a golden retriever.

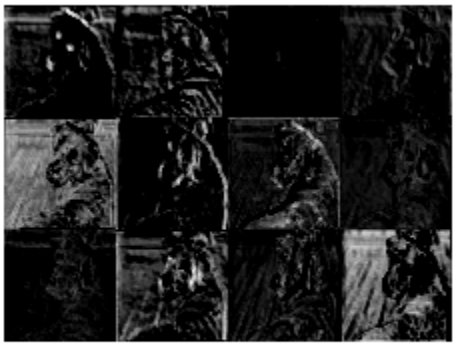
Activation Visualization

One of the simplest ways of understanding network behavior is to visualize the activations of each layer. Most convolutional neural networks learn to detect features such as color and edges in their first convolutional layer. In deeper convolutional layers, the network learns to detect more complicated features, such as eyes. Pass the image through the network and examine the output activations of the `conv2-relu_3x3_reduce` layer.

```
act = activations(net,img,"conv2-relu_3x3_reduce");
sz = size(act);
act = reshape(act,[sz(1) sz(2) 1 sz(3)]);
```

Display the activations for the first 12 channels of the layer.

```
I = imtile(mat2gray(act(:,:,:,1:12)));
figure
imshow(I)
```



White pixels represent strong positive activations and black pixels represent strong negative activations. You can see that the network is learning low-level features, such as edges and texture. The first channel highlights the eyes and nose of the dog, possibly due to their distinctive edge and color.

Investigate a deeper layer.

```
actDeep = activations(net,img,"inception_5b-output");
sz = size(actDeep)
```

```
sz = 1×3

           7           7        1024
```

```
actDeep = reshape(actDeep,[sz(1) sz(2) 1 sz(3)]);
```

This layer has 1024 channels. Each channel has an image. Investigating every image in detail is impractical. Instead, you can gain insight into the network behavior by considering the channel with the strongest activation.

```
[maxValue,maxValueIndex] = max(max(max(actDeep)));
actDeepMax = actDeep(:,:,:,maxValueIndex);

tiledlayout("flow")
nexttile
imshow(img)
nexttile
imshow(imresize(mat2gray(actDeepMax),inputSize))
```

The strongest activating channel focuses on the head of the dog, indicating that this layer is picking out more complex features.

To further explore network behavior, you can use more complex visualization methods.

Grad-CAM

Explore the network predictions using gradient-weighted class activation mapping (Grad-CAM). To understand which parts of the image are most important for classification, Grad-CAM uses the gradient of the classification score with respect to the convolutional features determined by the network. The places where this gradient is large are exactly the places where the final score depends most on the data. Compute the Grad-CAM map using the `gradCAM` function and the predicted class.

```
gradcamMap = gradCAM(net,img,YPred);
```

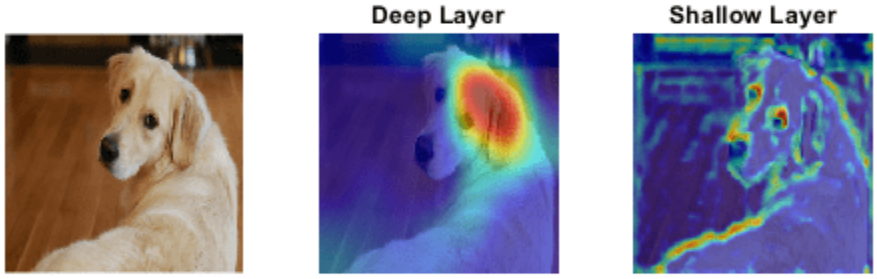
By default, the `gradCAM` function extracts the feature maps from the last ReLU layer with nonsingleton spatial dimensions or the last layer that gathers the outputs of ReLU layers (such as depth concatenation or addition layers). You can compute the Grad-CAM map for earlier layers in the

network by specifying the feature layer. Compute the Grad-CAM map for the early convolutional layer `conv2-relu_3x3`.

```
gradcamMapShallow = gradCAM(net,img,YPred,'FeatureLayer',"conv2-relu_3x3");
```

Use the `plotMaps` on page 5-0 supporting function, listed at the end of this example, to compare the Grad-CAM maps.

```
figure
alpha = 0.5;
cmap = "jet";
plotMaps(img,gradcamMap,gradcamMapShallow,"Deep Layer","Shallow Layer",alpha,cmap)
```



The Grad-CAM map for the layer at the end of the network highlights the head and ear of the dog, suggesting that the shape of the ear and the eye are important for classifying this dog as a golden retriever. The Grad-CAM map produced by the earlier layer highlights the edges of the dog. This is because earlier layers in a network learn simple features such as color and edges, while deep layers learn more complex features such as ears or eyes.

Occlusion Sensitivity

Compute the occlusion sensitivity of the image. Occlusion sensitivity is a simple technique for measuring network sensitivity to small perturbations in the input data. This method perturbs small areas of the input by replacing it with an occluding mask, typically a gray square. The mask moves across the image and the change in probability score for a given class is measured. You can use this method to highlight which parts of the image are most important to the classification. You can perform occlusion sensitivity using `occlusionSensitivity`.

Compute the occlusion sensitivity map for the golden retriever class.

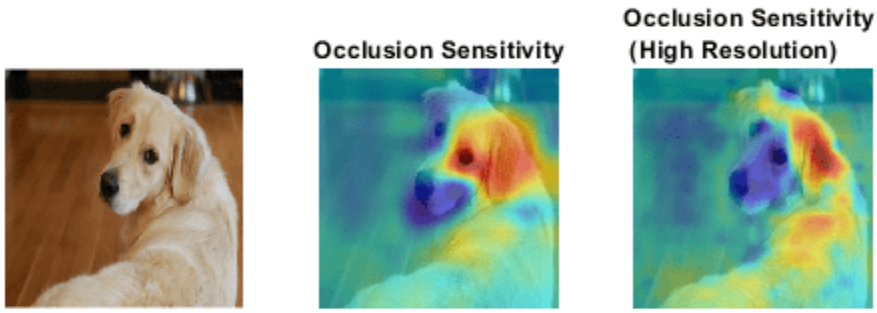
```
occlusionMap = occlusionSensitivity(net,img,YPred);
```

To examine the results of occlusion with higher resolution, reduce the mask size and stride using the `MaskSize` and `Stride` options. A smaller `Stride` value yields a higher resolution map but can take longer to compute and use more memory. A smaller `MaskSize` value yields more detail but can lead to noisier results. To get the best results from occlusion sensitivity, you must carefully choose the right values for the `MaskSize` and `Stride` options.

```
occlusionMapDetail = occlusionSensitivity(net,img,YPred,'Stride',10,'MaskSize',15);
```

Use the `plotMaps` function to compare the different occlusion sensitivity results.

```
plotMaps(img,occlusionMap,occlusionMapDetail, ...
    "Occlusion Sensitivity","Occlusion Sensitivity \newline (High Resolution)",alpha,cmap)
```



The lower resolution map shows similar results to Grad-CAM, highlighting the ear and eye of the dog. The higher resolution map shows that the ear is most important to the classification. The higher resolution map also indicates that the fur on the back of the dog is contributing to the classification decision.

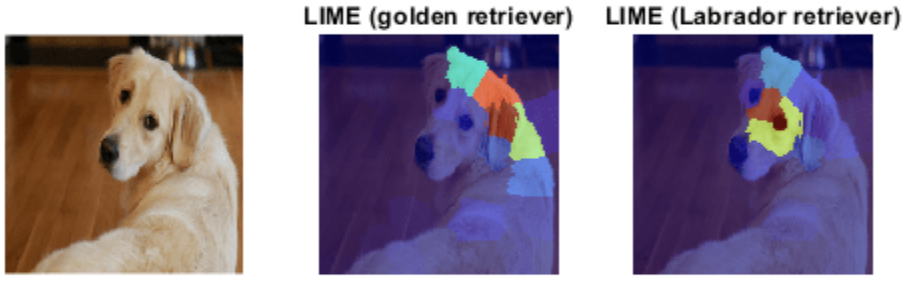
LIME

Next, consider the locally interpretable model-agnostic explanations (LIME) technique. LIME approximates the classification behavior of a deep neural network using a simpler, more interpretable model, such as a regression tree. Interpreting the decisions of this simpler model provides insight

into the decisions of the neural network. The simple model is used to determine the importance of features of the input data, as a proxy for the importance of the features to the deep neural network. The LIME technique uses a very different underlying mechanism to occlusion sensitivity or Grad-CAM.

Use the `imageLIME` function to view the most important features in the classification decision of a deep network. Compute the LIME map for the top two classes: golden retriever and Labrador retriever.

```
limeMapClass1 = imageLIME(net,img,topClasses(1));
limeMapClass2 = imageLIME(net,img,topClasses(2));

titleClass1 = "LIME (" + string(topClasses(1)) + ")";
titleClass2 = "LIME (" + string(topClasses(2)) + ")";
plotMaps(img,limeMapClass1,limeMapClass2,titleClass1,titleClass2,alpha,cmap)
```



The maps show which areas of the image are important to the classification. Red areas of the map have a higher importance—an image lacking these areas would have a lower score for the specified class. For the golden retriever class, the network focuses on the dog's head and ear to make its prediction. For the Labrador retriever class, the network is more focused on the dog's nose and eyes, rather than the ear. While both maps highlight the dog's forehead, for the network, the dog's ear and neck indicate the golden retriever class, while the dog's eyes indicate the Labrador retriever class.

The LIME maps are consistent with the occlusion sensitivity and Grad-CAM maps. Comparing the results of different interpretability techniques is important for verifying the conclusions you make.

Gradient Attribution

Gradient attribution methods produce pixel-resolution maps showing which pixels are most important to the network classification decision. These methods compute the gradient of the class score with respect to the input pixels. Intuitively, the map shows which pixels most affect the class score when changed. The gradient attribution methods produce maps with a higher resolution than those from Grad-CAM or occlusion sensitivity, but that tend to be much noisier, as a well-trained deep network is not strongly dependent on the exact value of specific pixels.

Use the `gradientAttribution` on page 5-0 supporting function, listed at the end of this example, to compute the gradient attribution map for the golden retriever class.

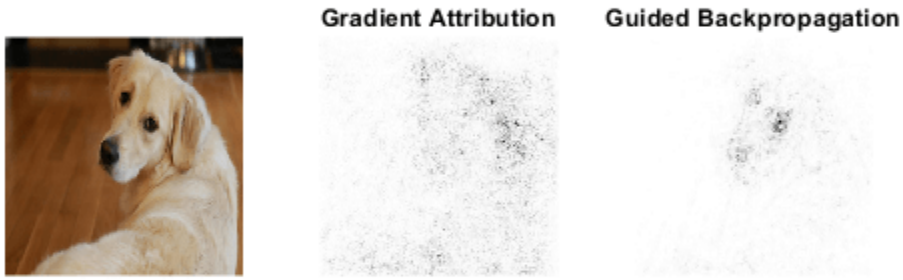
```
softmaxName = 'prob';
pixelMap = gradientAttribution(net,img,YPred,softmaxName,"autodiff");
```

You can obtain a sharper gradient attribution map by modifying the backwards pass through ReLU layers so that elements of the gradient that are less than zero and elements of the input to the ReLU layer that are less than zero are both set to zero. This method is known as guided backpropagation. Compute the gradient attribution map for the network using guided backpropagation.

```
pixelGuidedBackpropMap = gradientAttribution(net,img,YPred,softmaxName,"guided-backprop");
```

Display the gradient attribution maps using a custom colormap with 255 colors that maps values of `0` to white and `1` to black. The darker pixels are those most important for classification.

```
alpha = 1;
cmap = [linspace(1,0,255)' linspace(1,0,255)' linspace(1,0,255)'];
plotMaps(img,pixelMap,pixelGuidedBackpropMap, ...
    "Gradient Attribution","Guided Backpropagation",alpha,cmap)
```



The darkest parts of the map are those centered around the dog. The map is very noisy, but it does suggest that the network is using the expected information in the image to perform classification. The pixels in the dog have much more impact on the classification score than the pixels of the background. In the guided backpropagation map, the pixels are focused on the face of the dog, specifically the eyes and nose. Interestingly, this method highlights different regions than the lower resolution visualization techniques. The result suggests that, at a pixel level, the nose and eyes of the dog are important for classifying the image as a golden retriever.

Deep Dream Image

Deep dream is a feature visualization technique that creates images that strongly activate network layers. By visualizing these images, you can highlight the image features learned by a network. These images are useful for understanding and diagnosing network behavior. You can generate images by visualizing the features of the layers toward the end of the network. Unlike the previous methods, this technique is global and shows you the overall behavior of the network, not just for a specific input image.

To produce images that resemble a given class most closely, use the final fully connected layer `loss3-classifier`. Generate deep dream images for the top three classes the network predicts for the test image. Set `'Verbose'` to `false` to suppress detailed information on the optimization process.

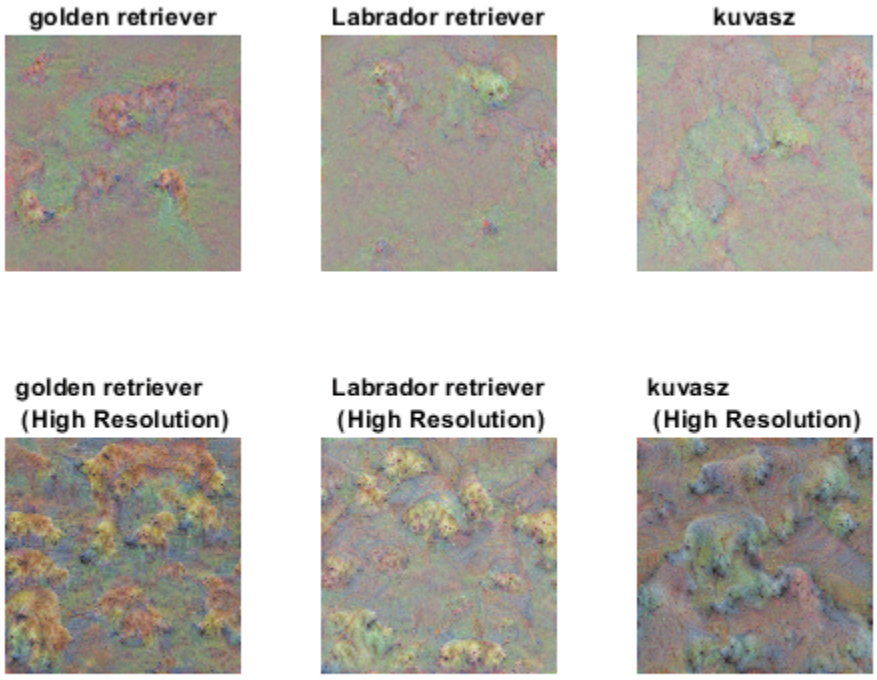
```
channels = topIdx;
learnableLayer =  "loss3-classifier";
dreamImage = deepDreamImage(net,learnableLayer,channels,'Verbose',false);
```

Increasing the number of pyramid levels and iterations per pyramid level can produce more detailed images at the expense of additional computation. Generate detailed deep dream images.

```
dreamImageDetailed = deepDreamImage(net,learnableLayer,channels, ...
    'Verbose',false,'NumIterations',100,'PyramidLevels',4);
```

Compare the deep dream images of the top three classes.

```
tiledlayout(2,3)

for i = 1:3
nexttile
imshow(dreamImage(:,:,:,i));
title(string(topClasses(i)));
end

for i = 1:3
nexttile
imshow(dreamImageDetailed(:,:,:,i));
title(string(topClasses(i)) + "\newline (High Resolution)");
end
```



The deep dream images show how the network envisions each of the three classes. Although these images are quite abstract, you can see key features for each of the top classes. It also shows how the network distinguishes the golden and Labrador retriever classes.

Supporting Functions

Replace Layers Function

The `replaceLayersOfType` function replaces all layers of the specified class with instances of a new layer. The new layers have the same names as the original layers.

```
function lgraph = replaceLayersOfType(lgraph,layerType,newLayer)

% Replace layers in the layerGraph lgraph of the type specified by
% layerType with copies of the layer newLayer.

for i=1:length(lgraph.Layers)
    if isa(lgraph.Layers(i),layerType)
        % Match names between the old and new layers.
        layerName = lgraph.Layers(i).Name;
        newLayer.Name = layerName;

        lgraph = replaceLayer(lgraph,layerName,newLayer);
    end
end
end
```

Plot Maps

Plot two maps, `map1` and `map2`, for the input image `img`. Use `alpha` to set the transparency of the map. Specify which colormap to use using `cmap`.

```
function plotMaps(img,map1,map2,title1,title2,alpha,cmap)

figure
subplot(1,3,1)
imshow(img)

subplot(1,3,2)
imshow(img)
hold on
imagesc(map1,'AlphaData',alpha)
colormap(cmap)
title(title1)
hold off

subplot(1,3,3)
imshow(img)
hold on
imagesc(map2,'AlphaData',alpha)
colormap(cmap)
title(title2)
hold off
end
```

Gradient Attribution Map

Compute the gradient attribution map. You must specify the softmax layer. You can compute the basic map, or a higher resolution map using guided backpropagation.

```
function map = gradientAttribution(net,img,YPred,softmaxName,method)
```

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,lgraph.Layers(end).Name);
dlnet = dlnetwork(lgraph);

% To use automatic differentiation, convert the image to a dlarray.
dlImg = dlarray(single(img),"SSC");

if method == "autodiff"
% Use dlfeval and the gradientMap function to compute the derivative. The gradientMap
% function passes the image forward through the network to obtain the class scores
% and contains a call to dlgradient to evaluate the gradients of the scores with respect
% to the image.
dydI = dlfeval(@gradientMap,dlnet,dlImg,softmaxName,YPred);
end

if method == "guided-backprop"

% Use the custom layer CustomBackpropReluLayer (attached as a supporting file)
% with a nonstandard backward pass, and use it with automatic differentiation.
customRelu = CustomBackpropReluLayer();

% Set the BackpropMode property of each CustomBackpropReluLayer to "guided-backprop".
customRelu.BackpropMode = "guided-backprop";

% Use the supporting function replaceLayersofType to replace all instances of reluLayer in the ne
% instances of CustomBackpropReluLayer.
lgraphGB = replaceLayersOfType(lgraph, ...
    'nnet.cnn.layer.ReLULayer',customRelu);

% Convert the layer graph containing the CustomBackpropReluLayers into a dlnetwork.
dlnetGB = dlnetwork(lgraphGB);
dydI = dlfeval(@gradientMap,dlnetGB,dlImg,softmaxName,YPred);
end

% Sum the absolute values of each pixel along the channel dimension, then rescale
% between 0 and 1.
map = sum(abs(extractdata(dydI)),3);
map = rescale(map);
end
```

Gradient Map

Compute the gradient of a class score with respect to one or more input images.

```
function dydI = gradientMap(dlnet,dlImgs,softmaxName,classIdx)

dydI = dlarray(zeros(size(dlImgs)));

for i=1:size(dlImgs,4)
    I = dlImgs(:,:,:,i);
    scores = predict(dlnet,I,'Outputs',{softmaxName});
    classScore = scores(classIdx);
    dydI(:,:,:,i) = dlgradient(classScore,I);
```

```
end
end
```

References

[1] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. "Learning Deep Features for Discriminative Localization." In *2016 Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition: 2921–29. Las Vegas, NV, USA: IEEE, 2016. https://doi.org/10.1109/CVPR.2016.319.*

[2] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization.” In *2017 Proceedings of the IEEE Conference on Computer Vision*: 618–626. Venice, Italy: IEEE, 2017. https://doi.org/10.1109/ICCV.2017.74.

[3] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. “‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier.” In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144. New York, NY: Association for Computing Machinery, 2016. https://doi.org/10.1145/2939672.2939778.

[4] Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.” Preprint, submitted April 19, 2014. https://arxiv.org/abs/1312.6034.

[5] TensorFlow. "DeepDreaming with TensorFlow." https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/deepdream.ipynb.

See Also

`gradCAM` | `imageLIME` | `occlusionSensitivity` | `deepDreamImage` | `tsne`

Related Examples

- “Deep Learning Visualization Methods” on page 5-186
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Understand Network Predictions Using LIME” on page 5-42
- “Understand Network Predictions Using Occlusion” on page 5-24
- “View Network Behavior Using tsne” on page 5-129

Deep Dream Images Using GoogLeNet

This example shows how to generate images using `deepDreamImage` with the pretrained convolutional neural network GoogLeNet.

Deep Dream is a feature visualization technique in deep learning that synthesizes images that strongly activate network layers. By visualizing these images, you can highlight the image features learned by a network. These images are useful for understanding and diagnosing network behavior.

You can generate interesting images by visualizing the features of the layers towards the end of the network.

The example uses Deep Learning Toolbox™ and Deep Learning Toolbox Model *for GoogLeNet Network* to generate the images.

Load Pretrained Network

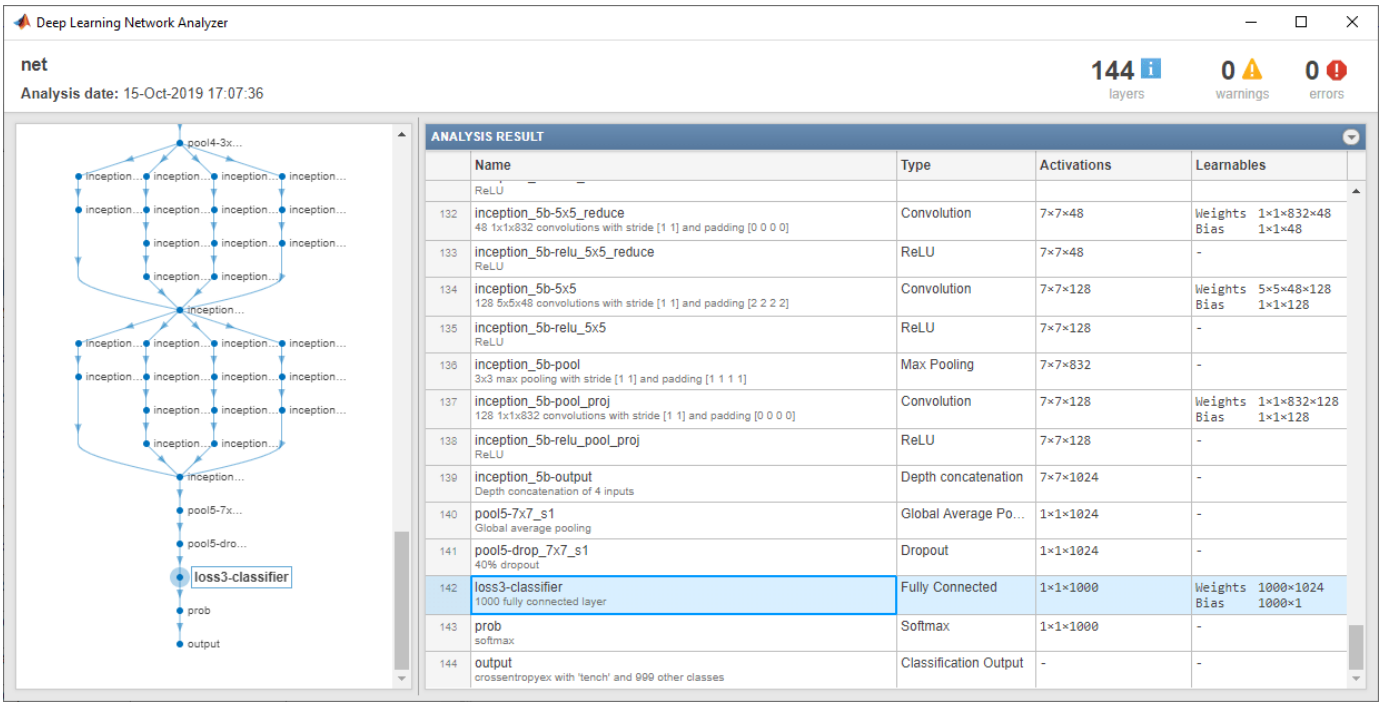
Load a pretrained GoogLeNet Network. If the Deep Learning Toolbox Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

```
net = googlenet;
```

Generate Image

To produce images that resemble a given class the most closely, select the fully connected layer. First, locate the layer index of this layer by viewing the network architecture using `analyzeNetwork`.

```
analyzeNetwork(net)
```



Then select the fully connected layer, in this example, 142.

```
layer = 142;
layerName = net.Layers(layer).Name
```

```
layerName =
'loss3-classifier'
```

You can generate multiple images at once by selecting multiple classes. Select the classes you want to visualize by setting `channels` to be the indices of those class names.

```
channels = [114 293 341 484 563 950];
```

The classes are stored in the `Classes` property of the output layer (the last layer). You can view the names of the selected classes by selecting the entries in `channels`.

```
net.Layers(end).Classes(channels)
```

```
ans = 6×1 categorical
     snail
     tiger
     zebra
     castle
     fountain
     strawberry
```

Generate the images using `deepDreamImage`. This command uses a compatible GPU, if available. Otherwise it uses the CPU. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

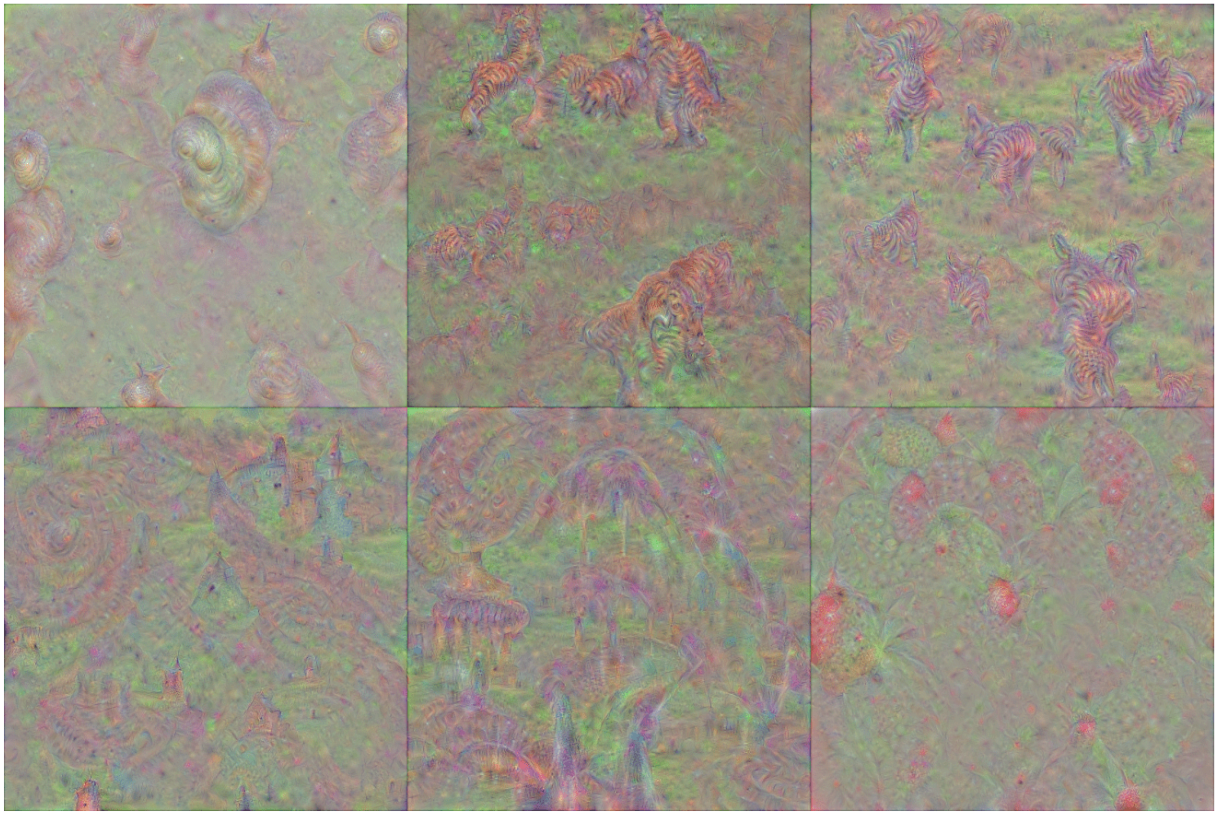
```
I = deepDreamImage(net,layerName,channels);
```

```
|=============================================|
|  Iteration  |  Activation  |  Pyramid Level  |
|             |   Strength   |                 |
|=============================================|
|           1 |         0.09 |               1 |
|           2 |         0.67 |               1 |
|           3 |         4.86 |               1 |
|           4 |         8.41 |               1 |
|           5 |        11.27 |               1 |
|           6 |        14.86 |               1 |
|           7 |        17.39 |               1 |
|           8 |        22.84 |               1 |
|           9 |        27.78 |               1 |
|          10 |        34.39 |               1 |
|           1 |         3.99 |               2 |
|           2 |        11.51 |               2 |
|           3 |        13.82 |               2 |
|           4 |        19.87 |               2 |
|           5 |        20.67 |               2 |
|           6 |        20.82 |               2 |
|           7 |        24.01 |               2 |
|           8 |        27.20 |               2 |
|           9 |        28.24 |               2 |
|          10 |        35.93 |               2 |
|           1 |        34.91 |               3 |
|           2 |        46.18 |               3 |
```

```
|           3 |        41.03 |                3 |
|           4 |        48.84 |                3 |
|           5 |        51.13 |                3 |
|           6 |        58.65 |                3 |
|           7 |        58.12 |                3 |
|           8 |        61.68 |                3 |
|           9 |        71.53 |                3 |
|          10 |        76.01 |                3 |
|=============================================|
```

Display all the images together using `imtile`.

```
figure
I = imtile(I);
imshow(I)
```



Generate More Detailed Images

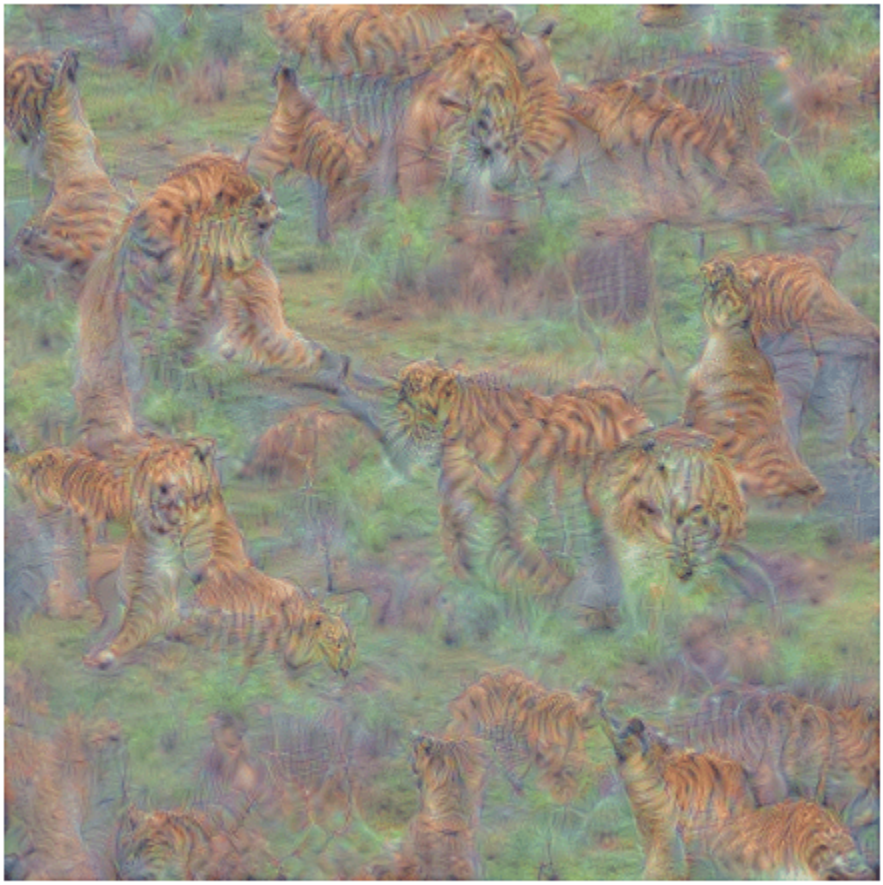
Increasing the number of pyramid levels and iterations per pyramid level can produce more detailed images at the expense of additional computation.

You can increase the number of iterations using the `'NumIterations'` option. Set the number of iterations to 100.

```
iterations = 100;
```

Generate a detailed image that strongly activates the 'tiger' class (channel 293). Set `'Verbose'` to false to suppress detailed information on the optimization process.

```
channels = 293;
I = deepDreamImage(net,layerName,channels, ...
    'Verbose',false, ...
    'NumIterations',iterations);

figure
imshow(I)
```



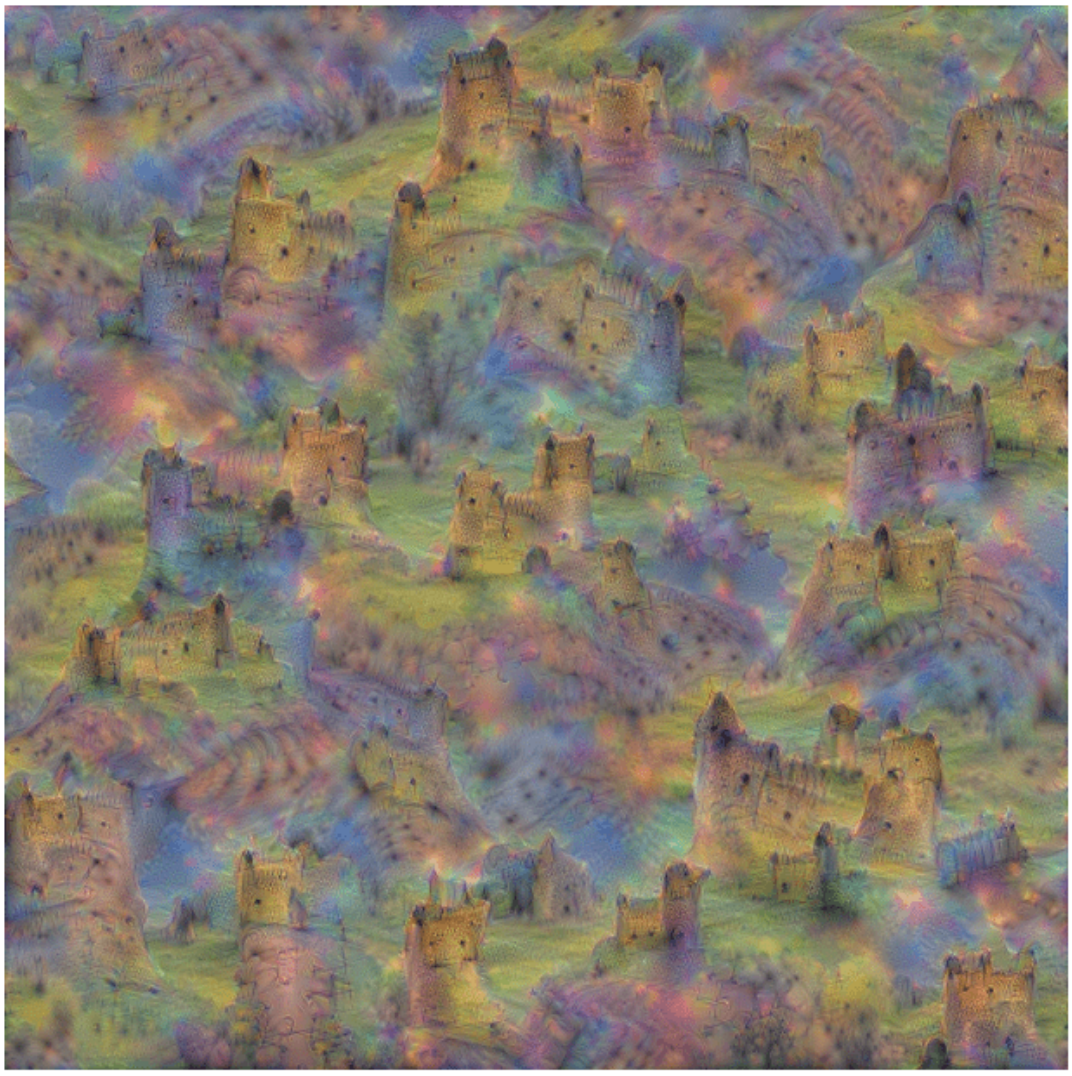
To produce larger and more detailed output images, you can increase both the number of pyramid levels and iterations per pyramid level.

Set the number of pyramid levels to 4.

```
levels = 4;
```

Generate a detailed image that strongly activates the 'castle' class (channel 484).

```
channels = 484;

I = deepDreamImage(net,layerName,channels, ...
    'Verbose',false, ...
    'NumIterations',iterations, ...
    'PyramidLevels',levels);

figure
imshow(I)
```



See Also

`googlenet` | `deepDreamImage` | `occlusionSensitivity` | `imageLIME` | `gradCAM`

Related Examples

- “Deep Learning Visualization Methods” on page 5-186
- “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Visualize Activations of a Convolutional Neural Network” on page 5-141
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21

Grad-CAM Reveals the Why Behind Deep Learning Decisions

This example shows how to use the gradient-weighted class activation mapping (Grad-CAM) technique to understand why a deep learning network makes its classification decisions. Grad-CAM, invented by Selvaraju and coauthors [1] on page 5-0 , uses the gradient of the classification score with respect to the convolutional features determined by the network in order to understand which parts of the image are most important for classification. This example uses the GoogLeNet pretrained network for images.

Grad-CAM is a generalization of the class activation mapping (CAM) technique. For activation mapping techniques on live webcam data, see "Investigate Network Predictions Using Class Activation Mapping" on page 5-123. Grad-CAM can also be applied to nonclassification examples such as regression or semantic segmentation. For an example showing how to use Grad-CAM to investigate the predictions of a semantic segmentation network, see "Explore Semantic Segmentation Network Using Grad-CAM" on page 5-66.

Load Pretrained Network

Load the GoogLeNet network.

```
net = googlenet;
```

Classify Image

Read the GoogLeNet image size.

```
inputSize = net.Layers(1).InputSize(1:2);
```

Load `sherlock.jpg`., an image of a golden retriever included with this example.

```
img = imread("sherlock.jpg");
```

Resize the image to the network input dimensions.

```
img = imresize(img,inputSize);
```

Classify the image and display it, along with its classification and classification score.

```
[classfn,score] = classify(net,img);
imshow(img);
title(sprintf("%s (%.2f)", classfn, score(classfn)));
```

GoogLeNet correctly classifies the image as a golden retriever. But why? What characteristics of the image cause the network to make this classification?

Grad-CAM Explains Why

The Grad-CAM technique utilizes the gradients of the classification score with respect to the final convolutional feature map, to identify the parts of an input image that most impact the classification score. The places where this gradient is large are exactly the places where the final score depends most on the data.

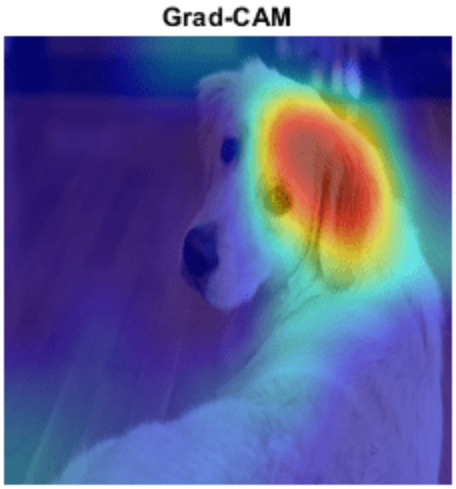
The `gradCAM` function computes the importance map by taking the derivative of the reduction layer output for a given class with respect to a convolutional feature map. For classification tasks, the `gradCAM` function automatically selects suitable layers to compute the importance map for. You can also specify the layers with the `'ReductionLayer'` and `'FeatureLayer'` name-value arguments.

Compute the Grad-CAM map.

```
map = gradCAM(net,img,classfn);
```

Show the Grad-CAM map on top of the image by using an `'AlphaData'` value of 0.5. The `'jet'` colormap has deep blue as the lowest value and deep red as the highest.

```
imshow(img);
hold on;
imagesc(map,'AlphaData',0.5);
colormap jet
hold off;
title("Grad-CAM");
```



Clearly, the upper face and ear of the dog have the greatest impact on the classification.

For a different approach to investigating the reasons for deep network classifications, see `occlusionSensitivity` and `imageLIME`.

References

[1] Selvaraju, R. R., M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. "*Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization*." In IEEE International Conference on Computer Vision (ICCV), 2017, pp. 618–626. Available at `Grad-CAM` on the Computer Vision Foundation Open Access website.

See Also

`gradCAM` | `imageLIME` | `occlusionSensitivity` | `deepDreamImage`

More About

- “Explore Semantic Segmentation Network Using Grad-CAM” on page 5-66
- “Investigate Network Predictions Using Class Activation Mapping” on page 5-123
- “Deep Learning Visualization Methods” on page 5-186
- “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2
- “Understand Network Predictions Using LIME” on page 5-42

Understand Network Predictions Using Occlusion

This example shows how to use occlusion sensitivity maps to understand why a deep neural network makes a classification decision. Occlusion sensitivity is a simple technique for understanding which parts of an image are most important for a deep network's classification. You can measure a network's sensitivity to occlusion in different regions of the data using small perturbations of the data. Use occlusion sensitivity to gain a high-level understanding of what image features a network uses to make a particular classification, and to provide insight into the reasons why a network can misclassify an image.

Deep Learning Toolbox provides the `occlusionSensitivity` function to compute occlusion sensitivity maps for deep neural networks that accept image inputs. The `occlusionSensitivity` function perturbs small areas of the input by replacing it with an occluding mask, typically a gray square. The mask moves across the image, and the change in probability score for a given class is measured as a function of mask position. You can use this method to highlight which parts of the image are most important to the classification: when that part of the image is occluded, the probability score for the predicted class will fall sharply.

Load Pretrained Network and Image

Load the pretrained network GoogLeNet, which will be used for image classification.

```
net = googlenet;
```

Extract the image input size and the output classes of the network.

```
inputSize = net.Layers(1).InputSize(1:2);
classes = net.Layers(end).Classes;
```

Load the image. The image is of a dog named Laika. Resize the image to the network input size.

```
imgLaikaGrass = imread("laika_grass.jpg");
imgLaikaGrass = imresize(imgLaikaGrass,inputSize);
```

Classify the image, and display the three classes with the highest classification score in the image title.

```
[YPred,scores] = classify(net,imgLaikaGrass);
[~,topIdx] = maxk(scores, 3);
topScores = scores(topIdx);
topClasses = classes(topIdx);

imshow(imgLaikaGrass)
titleString = compose("%s (%.2f)",topClasses,topScores');
title(sprintf(join(titleString, "; ")));
```

Laika is a poodle-cocker spaniel cross. This breed is not a class in GoogLeNet, so the network has some difficulty classifying the image. The network is not very confident in its predictions — the predicted class `miniature poodle` only has a score of 23%. The class with the next highest score is also a type of poodle, which is a reasonable classification. The network also assigns a moderate probability to the `Tibetan terrier` class. We can use occlusion to understand which parts of the image cause the network to suggest these three classes.

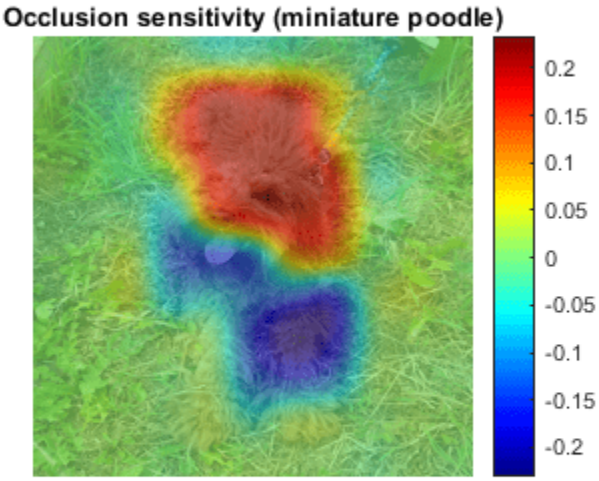
Identify Areas of an Image the Network Uses for Classification

You can use occlusion to find out which parts of the image are important for the classification. First, look at the predicted class of `miniature poodle`. What parts of the image suggest this class? Use the occlusion sensitivity function to map the change in the classification score when parts of the image are occluded.

```
map = occlusionSensitivity(net,imgLaikaGrass,YPred);
```

Display the image of Laika with the occlusion sensitivity map overlaid.

```
imshow(imgLaikaGrass,'InitialMagnification', 150)
hold on
imagesc(map,'AlphaData',0.5)
colormap jet
colorbar

title(sprintf("Occlusion sensitivity (%s)", ...
    YPred))
```



The occlusion map shows which parts of the image have a positive contribution to the score for the `miniature poodle` class, and which parts have a negative contribution. Red areas of the map have a higher value and are evidence for the `miniature poodle` class — when the red areas are obscured, the score for `miniature poodle` goes down. In this image, Laika's head, back, and ears provide the strongest evidence for the `miniature poodle` class.

Blue areas of the map with lower values are parts of the image that lead to an increase in the score for `miniature poodle` when occluded. Often, these areas are evidence of another class, and can confuse the network. In this case, Laika's mouth and legs have a negative contribution to the overall score for `miniature poodle`.

The occlusion map is strongly focused on the dog in the image, which shows that GoogLeNet is classifying the correct object in the image. If your network is not producing the results you expect, an occlusion map can help you understand why. For example, if the network is strongly focused on other parts of the image, this suggests that the network learned the wrong features.

You can get similar results using the gradient class activation mapping (Grad-CAM) technique. Grad-CAM uses the gradient of the classification score with respect to the last convolutional layer in a network in order to understand which parts of the image are most important for classification. For an example, see “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21.

Occlusion sensitivity and Grad-CAM usually return qualitatively similar results, although they work in different ways. Typically, you can compute the Grad-CAM map faster that the occlusion map, without tuning any parameters. However, the Grad-CAM map can usually has a lower spatial resolution than an occlusion map and can miss fine details. The underlying resolution of Grad-CAM is the spatial

resolution of the last convolutional feature map; in the case of GoogleNet this is 7-by-7 pixels. To get the best results from occlusion sensitivity, you must choose the right values for the `MaskSize` and `Stride` options. This tuning provides more flexibility to examine the input features at different length scales.

Compare Evidence for Different Classes

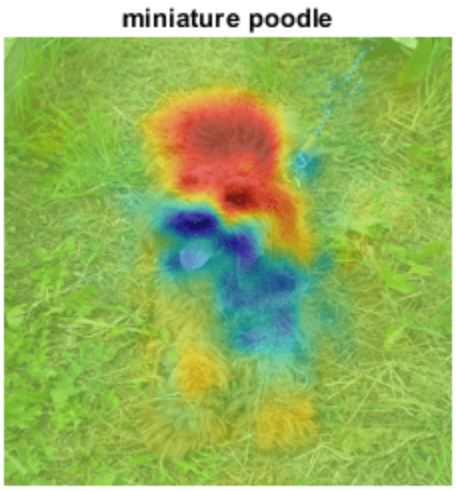
You can use occlusion to compare which parts of the image the network identifies as evidence for different classes. This can be useful in cases where the network is not confident in the classification and gives similar scores to several classes.

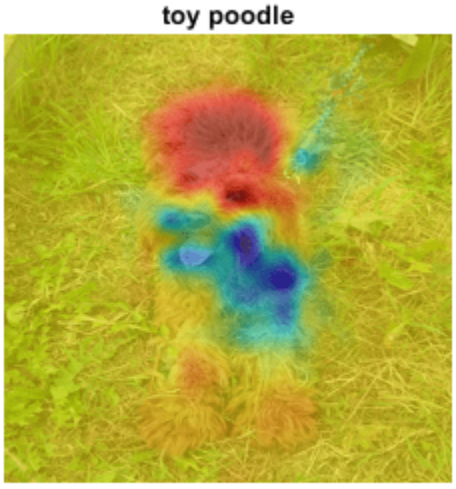
Compute an occlusion map for each of the top three classes. To examine the results of occlusion with higher resolution, reduce the mask size and stride using the `MaskSize` and `Stride` options. A smaller `Stride` leads to a higher-resolution map, but can take longer to compute and use more memory. A smaller `MaskSize` illustrates smaller details, but can lead to noisier results.

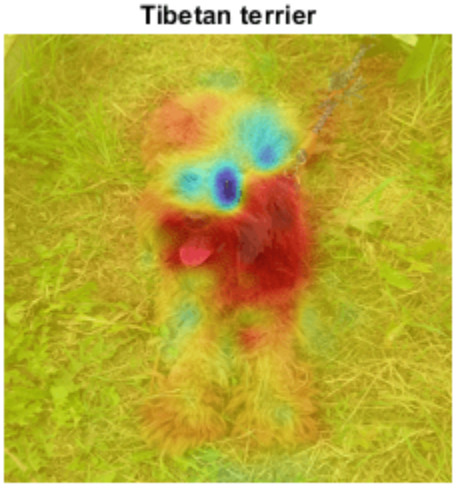
```
topClasses = classes(topIdx);
topClassesMap = occlusionSensitivity(net, imgLaikaGrass, topClasses, ...
    "Stride", 10, ...
    "MaskSize", 15);
```

Plot the results for each of the top three classes.

```
for i=1:length(topIdx)
    figure
    imshow(imgLaikaGrass);
    hold on
    imagesc(topClassesMap(:,:,i), 'AlphaData', 0.5);
    colormap jet;

    classLabel = string(classes(topIdx(i)));
    title(sprintf("%s", classLabel));
end
```







Different parts of the image have a very different impact on the class scores for different dog breeds. The dog's back has a strong influence in favor of the `miniature poodle` and `toy poodle` classes, while the mouth and ears contribute to the `Tibetan terrier` class.

Investigate Misclassification Issues

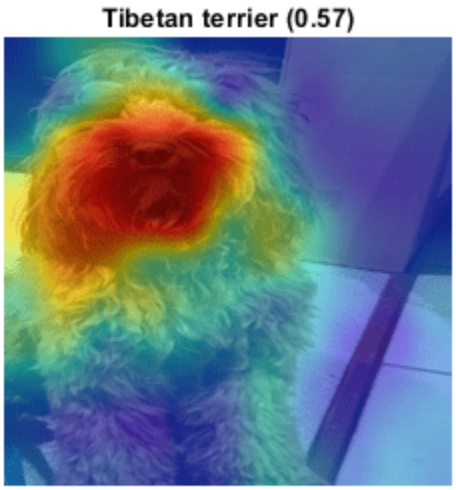
If your network is consistently misclassifying certain types of input data, you can use occlusion sensitivity to determine if particular features of your input data are confusing the network. From the occlusion map of Laika sitting on the grass, you could expect that images of Laika which are more focused on her face are likely to be misclassified as `Tibetan terrier`. You can verify that this is the case using another image of Laika.

```
imgLaikaSit = imresize(imread("laika_sitting.jpg"),inputSize);

[YPred,scores] = classify(net,imgLaikaSit);
[score,idx] = max(scores);
YPred, score
```

```
YPred = categorical
     Tibetan terrier


score = single
    0.5668
```

Compute the occlusion map of the new image.

```
map = occlusionSensitivity(net,imgLaikaSit,YPred);

imshow(imgLaikaSit);
hold on;
imagesc(map, 'AlphaData', 0.5);
colormap jet;

title(sprintf("%s (%.2f)",...
    string(classes(idx)),score));
```



Again, the network strongly associates the dog's nose and mouth with the `Tibetan terrier` class. This highlights a possible failure mode of the network, since it suggests that images of Laika's face will consistently be misclassified as `Tibetan terrier`.

You can use the insights gained from the `occlusionSensitivity` function to make sure your network is focusing on the correct features of the input data. The cause of the classification problem in this example is that the available classes of GoogleNet do not include cross-breed dogs like Laika. The occlusion map demonstrates why the network is confused by these images of Laika. It is important to be sure that the network you are using is suitable for the task at hand.

In this example, the network is mistakenly identifying different parts of the object in the image as different classes. One solution to this issue is to retrain the network with more labeled data that covers a wider range of observations of the misclassified class. For example, the network here could be retrained using a large number of images of Laika taken at different angles, so that it learns to associate both the back and the front of the dog with the correct class.

References

[1] Zeiler M.D., Fergus R. (2014) Visualizing and Understanding Convolutional Networks. In: Fleet D., Pajdla T., Schiele B., Tuytelaars T. (eds) Computer Vision – ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol 8689. Springer, Cham

See Also

`googlenet` | `occlusionSensitivity`

More About

- “Deep Learning Visualization Methods” on page 5-186
- “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Visualize Features of a Convolutional Neural Network” on page 5-156
- “Visualize Activations of a Convolutional Neural Network” on page 5-141

Investigate Classification Decisions Using Gradient Attribution Techniques

This example shows how to use gradient attribution maps to investigate which parts of an image are most important for classification decisions made by a deep neural network.

Deep neural networks can look like black box decision makers — they give excellent results on complex problems, but it can be hard to understand why a network gives a particular output. Explainability is increasingly important as deep networks are used in more applications. To consider a network explainable, it must be clear what parts of the input data the network is using to make a decision and how much this data contributes to the network output.

A range of visualization techniques are available to determine if a network is using sensible parts of the input data to make a classification decision. As well as the gradient attribution methods shown in this example, you can use techniques such as gradient-weighted class-activation mapping (Grad-CAM) and occlusion sensitivity. For examples, see

- “Understand Network Predictions Using Occlusion” on page 5-24
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21

The gradient attribution methods explored in this example provide pixel-resolution maps that show which pixels are most important to the network's classification. They compute the gradient of the class score with respect to the input pixels. Intuitively, the map shows which pixels most affect the class score when changed. The gradient attribution methods produce maps with higher resolution than those from Grad-CAM or occlusion sensitivity, but that tend to be much noisier, as a well-trained deep network is not strongly dependent on the exact value of specific pixels. Use the gradient attribution techniques to find the broad areas of an image that are important to the classification.

The simplest gradient attribution map is the gradient of the class score for the predicted class with respect to each pixel in the input image [1]. This shows which pixels have the largest impact on the class score, and therefore which pixels are most important to the classification. This example shows how to use gradient attribution and two extended methods: guided backpropagation [2] and integrated gradients [3]. The use of these techniques is under debate as it is not clear how much insight these extensions can provide into the model [4].

Load Pretrained Network and Image

Load the pretrained GoogLeNet network.

```
net = googlenet;
```

Extract the image input size and the output classes of the network.

```
inputSize = net.Layers(1).InputSize(1:2);
classes = net.Layers(end).Classes;
```

Load the image. The image is of a dog named Laika. Resize the image to the network input size.

```
img = imread("laika_grass.jpg");
img = imresize(img,inputSize);
```

Classify the image and display the predicted class and classification score.

```
[YPred, scores] = classify(net, img);
[score, classIdx] = max(scores);
```

```
predClass = classes(classIdx);

imshow(img);
title(sprintf("%s (%.2f)",string(predClass),score));
```

The network classifies Laika as a miniature poodle, which is a reasonable guess. She is a poodle/cocker spaniel cross.

Compute Gradient Attribution Map Using Automatic Differentiation

The gradient attribution techniques rely on finding the gradient of the prediction score with respect to the input image. The gradient attribution map is calculated using the following formula:

$$W_{\mathrm{xy}}^{c} = \frac{\partial S^{c}}{\partial I_{\mathrm{xy}}}$$

where W_{xy}^{c} represents the importance of the pixel at location (x, y) to the prediction of class c, S^{c} is the softmax score for that class, and I_{xy} is the image at pixel location (x, y) [1].

Convert the network to a `dlnetwork` so that you can use automatic differentiation to compute the gradients.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,lgraph.Layers(end).Name);

dlnet = dlnetwork(lgraph);
```

Specify the name of the softmax layer, `'prob'`.

```
softmaxName = 'prob';
```

To use automatic differentiation, convert the image of Laika to a `dlarray`.

```
dlImg = dlarray(single(img),'SSC');
```

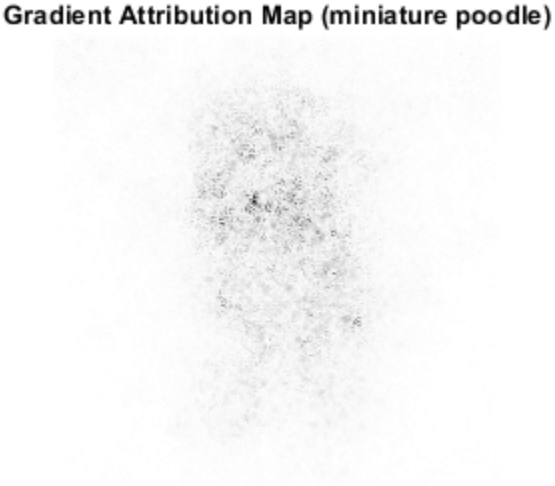
Use `dlfeval` and the `gradientMap` function (defined in the Supporting Functions on page 5-0 section of this example) to compute the derivative $\frac{\partial S^C}{\partial I_{xy}}$. The `gradientMap` function passes the image forward through the network to obtain the class scores and contains a call to `dlgradient` to evaluate the gradients of the scores with respect to the image.

```
dydI = dlfeval(@gradientMap,dlnet,dlImg,softmaxName,classIdx);
```

The attribution map `dydI` is a 227-by-227-by-3 array. Each element in each channel corresponds to the gradient of the class score with respect to the input image for that channel of the original RGB image.

There are a number of ways to visualize this map. Directly plotting the gradient attribution map as an RGB image can be unclear as the map is typically quite noisy. Instead, sum the absolute values of each pixel along the channel dimension, then rescale between `0` and `1`. Display the gradient attribution map using a custom colormap with 255 colors that maps values of `0` to white and `1` to black.

```
map = sum(abs(extractdata(dydI)),3);
map = rescale(map);

cmap = [linspace(1,0,255)' linspace(1,0,255)' linspace(1,0,255)'];

imshow(map, "Colormap", cmap);
title("Gradient Attribution Map (" + string(predClass) + ")");
```



The darkest parts of the map are those centered around the dog. The map is extremely noisy, but it does suggest that the network is using the expected information in the image to perform classification. The pixels in the dog have much more impact on the classification score than the pixels of the grassy background.

Sharpen the Gradient Attribution Map Using Guided Backpropagation

You can obtain a sharper gradient attribution map by modifying the network's backwards pass through ReLU layers so that elements of the gradient that are less than zero and elements of the input to the ReLU layer that are less than zero are both set to zero. This is known as guided backpropagation [2].

The guided backpropagation backward function is:

$$\frac{dL}{dZ} = (X > 0) * \left(\frac{dL}{dZ} > 0\right) * \frac{dL}{dZ}$$

where L is the loss, X is the input to the ReLU layer, and Z is the output.

You can write a custom layer with a non-standard backward pass, and use it with automatic differentiation. A custom layer class `CustomBackpropReluLayer` that implements this modification is included as a supporting file in this example. When automatic differentiation backpropagates through `CustomBackpropReluLayer` objects, it uses the modified guided backpropagation function defined in the custom layer.

Use the supporting function `replaceLayersofType` (defined in the Supporting Functions on page 5-0 section of this example) to replace all instances of `reluLayer` in the network with instances of `CustomBackpropReluLayer`. Set the `BackpropMode` property of each `CustomBackpropReluLayer` to `"guided-backprop"`.

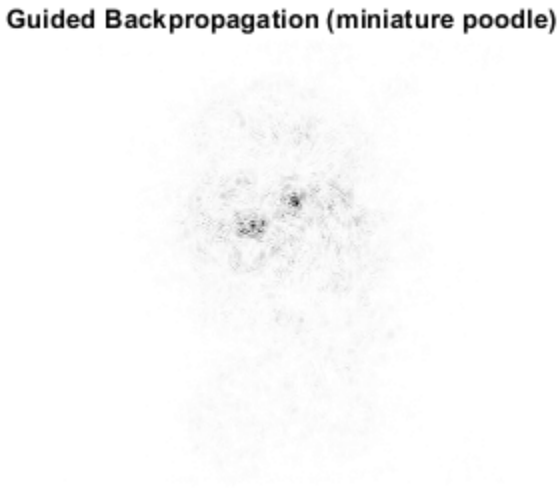
```
customRelu = CustomBackpropReluLayer();
customRelu.BackpropMode = "guided-backprop";

lgraphGB = replaceLayersOfType(lgraph, ...
    "nnet.cnn.layer.ReLULayer",customRelu);
```

Convert the layer graph containing the `CustomBackpropReluLayers` into a `dlnetwork`.

```
dlnetGB = dlnetwork(lgraphGB);
```

Compute and plot the gradient attribution map for the network using guided backpropagation.

```
dydIGB = dlfeval(@gradientMap,dlnetGB,dlImg,softmaxName,classIdx);

mapGB = sum(abs(extractdata(dydIGB)),3);
mapGB = rescale(mapGB);

imshow(mapGB, "Colormap", cmap);
title("Guided Backpropagation (" + string(predClass) + ")");
```



You can see that guided backpropagation technique more clearly highlights different parts of the dog, such as the eyes and nose.

You can also use the Zeiler-Fergus technique for backpropagation through ReLU layers [5]. For the Zeiler-Fergus technique, the backward function is given as:

$$\frac{dL}{dZ} = \left(\frac{dL}{dZ} > 0\right) * \frac{dL}{dZ}$$

Set the `BackpropMode` property of the `CustomBackpropReluLayer` instances to `"zeiler-fergus"`.

```
customReluZF = CustomBackpropReluLayer();
customReluZF.BackpropMode = "zeiler-fergus";

lgraphZF = replaceLayersOfType(lgraph, ...
    "nnet.cnn.layer.ReLULayer",customReluZF);

dlnetZF = dlnetwork(lgraphZF);

dydIZF = dlfeval(@gradientMap,dlnetZF,dlImg,softmaxName,classIdx);


mapZF = sum(abs(extractdata(dydIZF)),3);
mapZF = rescale(mapZF);

imshow(mapZF,"Colormap", cmap);
title("Zeiler-Fergus (" + string(predClass) + ")");
```

The gradient attribution maps computed using the Zeiler-Fergus backpropagation technique are much less clear than those computed using guided backpropagation.

Evaluate Sensitivity to Image Changes Using Integrated Gradients

The integrated gradients approach computes integrates the gradients of class score with respect to image pixels across a set of images that are linearly interpolated between a baseline image and the original image of interest [3]. The integrated gradients technique is designed to be sensitive to the changes in the pixel value over the integration, such that if a change in a pixel value affects the class score, that pixel has a non-zero value in the map. Non-linearities in the network, such as ReLU layers, can prevent this sensitivity in simpler gradient attribution techniques.

The integrated gradients attribution map is calculated as

$$W_{\mathrm{xy}}^{c} = \left(I_{\mathrm{xy}} - I_{\mathrm{xy}}^{0}\right)\int_{\alpha=0}^{1} d\alpha \frac{\partial S^{c}\left(I_{\mathrm{xy}}(\alpha)\right)}{\partial I_{\mathrm{xy}}(\alpha)},$$

where W_{xy}^{c} is the map's value for class c at pixel location (x, y), I_{xy}^{0} is a baseline image, and $I_{\mathrm{xy}}(\alpha)$ is the image at a distance α along the path between the baseline image and the input image:

$$I_{\mathrm{xy}}(\alpha) = I_{\mathrm{xy}}^{0} + \alpha\left(I_{\mathrm{xy}} - I_{\mathrm{xy}}^{0}\right).$$

In this example, the integrated gradients formula is simplified by summing over a discrete index,n, instead of integrating over α :

$$W_{\mathrm{xy}}^{c} = \left(I_{\mathrm{xy}} - I_{\mathrm{xy}}^{0}\right)\sum_{n=0}^{N} \frac{\partial S^{c}\left(I_{\mathrm{xy}}^{n}\right)}{\partial I_{\mathrm{xy}}^{n}},$$

with

$$I_{\mathrm{xy}}^{n} = I_{\mathrm{xy}}^{0} + \frac{n}{N}\left(I_{\mathrm{xy}} - I_{\mathrm{xy}}^{0}\right).$$

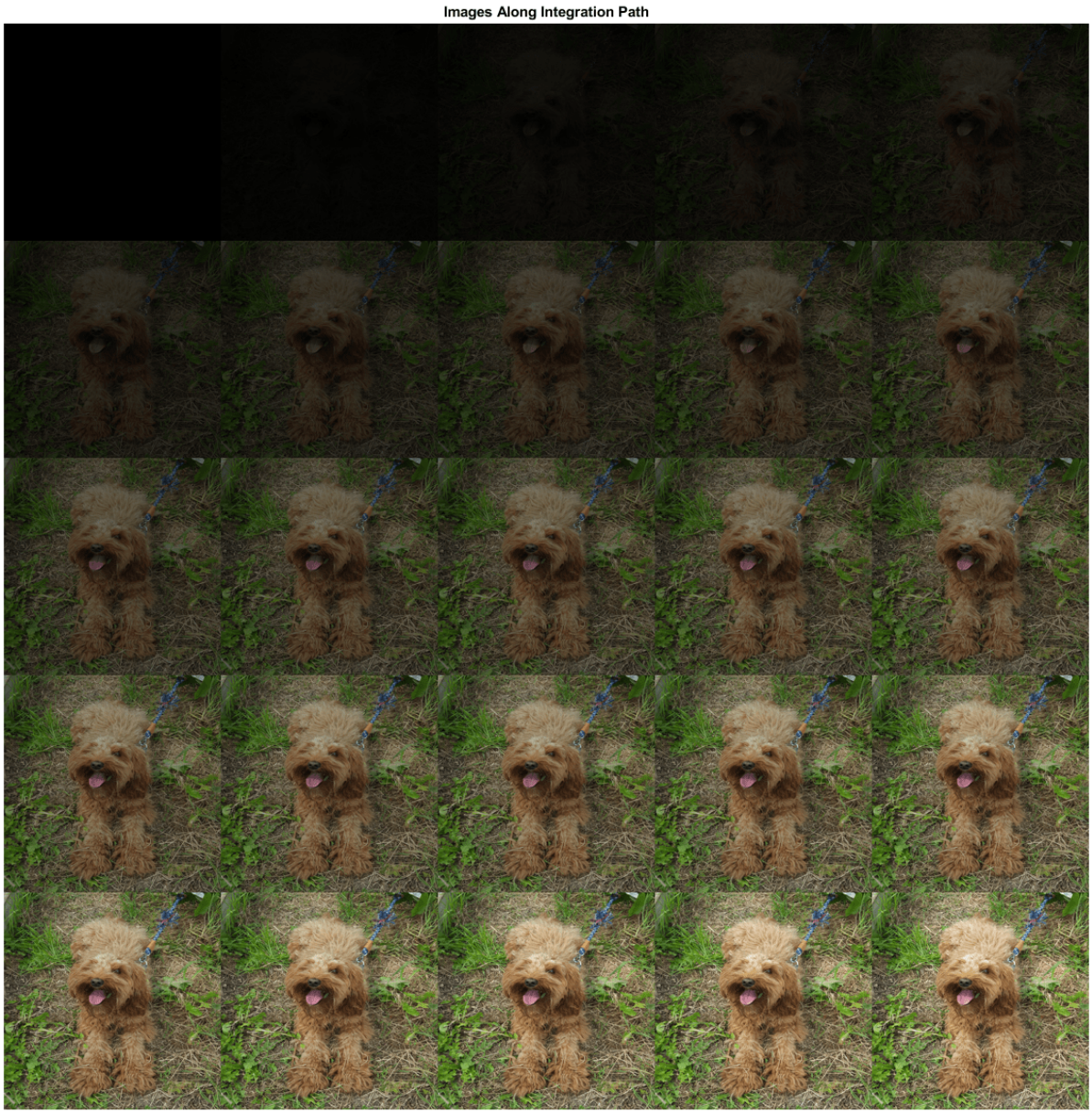
For image data, choose the baseline image to be a black image of zeros. Find the image that is the difference between the original image and the baseline image. In this case, `differenceImg` is the same as the original image as the baseline image is zero.

```
baselineImg = zeros([inputSize, 3]);
differenceImg = single(img) - baselineImg;
```

Create an array of images corresponding to discrete steps along the linear path from the baseline image to the original input image. A larger number of images will give smoother results but take longer to compute.

```
numPathImages = 25 ;
```

```
pathImgs = zeros([inputSize 3 numPathImages-1]);
for n=0:numPathImages-1
    pathImgs(:,:,:,n+1) = baselineImg + (n)/(numPathImages-1) * differenceImg;
end

figure;
imshow(imtile(rescale(pathImgs)));
title("Images Along Integration Path");
```



Convert the mini-batch of path images to a `dlarray`. Format the data with the format `'SSCB'` for the two spatial, one channel and one batch dimensions. Each path image is a single observation in the mini-batch. Compute the gradient map for the resulting batch of images along the path.

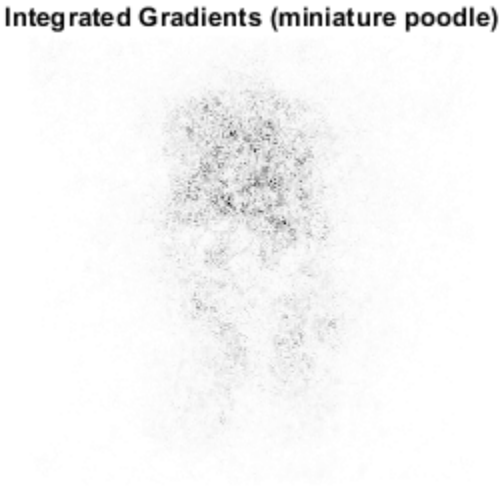
```
dlPathImgs = dlarray(pathImgs, 'SSCB');
dydIIG = dlfeval(@gradientMap, dlnet, dlPathImgs, softmaxName, classIdx);
```

For each channel, sum the gradients of all observations in the mini-batch.

```
dydIIGSum = sum(dydIIG,4);
```

Multiply each element of the summed gradient attribution maps with the corresponding element of `differenceImg`. To compute the integrated gradient attribution map, sum over each channel and rescale.

```
dydIIGSum = differenceImg .* dydIIGSum;

mapIG = sum(extractdata(abs(dydIIGSum)),3);
mapIG = rescale(mapIG);

imshow(mapIG, "Colormap", cmap);
title("Integrated Gradients (" + string(predClass) + ")");
```



The computed map shows the network is more strongly focused on the dog's face as a means of deciding on its class.

The gradient attribution techniques demonstrated here can be used to check whether your network is focusing on the expected parts of the image when making a classification. To get good insights into the way your model is working and explain classification decisions, you can perform these techniques on a range of images and find the specific features that strongly contribute to a particular class. The unmodified gradient attributions technique is likely to be the more reliable method for explaining network decisions. While the guided backpropagation and integrated gradient techniques can produce the clearest gradient maps, it is not clear how much insight these techniques can provide into how the model works [4].

Supporting Functions

Gradient Map Function

The function `gradientMap` computes the gradients of the score with respect to an image, for a specified class. The function accepts a single image or a mini-batch of images. Within this example, the function `gradientMap` is introduced in the section Compute Gradient Attribution Map Using Automatic Differentiation on page 5-0 .

```
function dydI = gradientMap(dlnet, dlImgs, softmaxName, classIdx)
% Compute the gradient of a class score with respect to one or more input
% images.

dydI = dlarray(zeros(size(dlImgs)));

for i=1:size(dlImgs,4)
    I = dlImgs(:,:,:,i);
    scores = predict(dlnet,I,'Outputs',{softmaxName});
    classScore = scores(classIdx);
    dydI(:,:,:,i) = dlgradient(classScore,I);
end
end
```

Replace Layers Function

The `replaceLayersOfType` function replaces all layers of the specified class with instances of a new layer. The new layers are named with the same names as the original layers. Within this example, the function `replaceLayersOfType` is introduced in the section Sharpen the Gradient Attribution Map using Guided Backpropagation on page 5-0 .

```
function lgraph = replaceLayersOfType(lgraph, layerType, newLayer)
% Replace layers in the layerGraph lgraph of the type specified by
% layerType with copies of the layer newLayer.

for i=1:length(lgraph.Layers)
    if isa(lgraph.Layers(i), layerType)
        % Match names between old and new layer.
        layerName = lgraph.Layers(i).Name;
        newLayer.Name = layerName;

        lgraph = replaceLayer(lgraph, layerName, newLayer);
    end
end
end
```

References

[1] Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.” *ArXiv:1312.6034 [Cs]*, April 19, 2014. http://arxiv.org/abs/1312.6034.

[2] Springenberg, Jost Tobias, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. “Striving for Simplicity: The All Convolutional Net.” *ArXiv:1412.6806 [Cs]*, April 13, 2015. http://arxiv.org/abs/1412.6806.

[3] Sundararajan, Mukund, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks." *Proceedings of the 34th International Conference on Machine Learning (PMLR)* 70 (2017): 3319-3328

[4] Adebayo, Julius, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. “Sanity Checks for Saliency Maps.” *ArXiv:1810.03292 [Cs, Stat]*, October 27, 2018. http://arxiv.org/abs/1810.03292.

[5] Zeiler, Matthew D. and Rob Fergus. "Visualizing and Understanding Convolutional Networks." In *Computer Vision – ECCV 2014. Lecture Notes in Computer Science 8689*, edited by D. Fleet, T. Pajdla, B. Schiele, T. Tuytelaars. Springer, Cham, 2014.

See Also

`googlenet` | `occlusionSensitivity` | `dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `gradCAM` | `imageLIME`

More About

- “Deep Learning Visualization Methods” on page 5-186
- “Understand Network Predictions Using Occlusion” on page 5-24
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Understand Network Predictions Using LIME” on page 5-42
- “Specify Custom Layer Backward Function” on page 18-107

Understand Network Predictions Using LIME

This example shows how to use locally interpretable model-agnostic explanations (LIME) to understand why a deep neural network makes a classification decision.

Deep neural networks are very complex and their decisions can be hard to interpret. The LIME technique approximates the classification behavior of a deep neural network using a simpler, more interpretable model, such as a regression tree. Interpreting the decisions of this simpler model provides insight into the decisions of the neural network [1]. The simple model is used to determine the importance of features of the input data, as a proxy for the importance of the features to the deep neural network.

When a particular feature is very important to a deep network's classification decision, removing that feature significantly affects the classification score. That feature is therefore important to the simple model too.

Deep Learning Toolbox provides the `imageLIME` function to compute maps of the feature importance determined by the LIME technique. The LIME algorithm for images works by:

- Segmenting an image into features.
- Generating many synthetic images by randomly including or excluding features. Excluded features have every pixel replaced with the value of the image average, so they no longer contain information useful for the network.
- Classifying the synthetic images with the deep network.
- Fitting a simpler regression model using the presence or absence of image features for each synthetic image as binary regression predictors for the scores of the target class.The model approximates the behavior of the complex deep neural network in the region of the observation.
- Computing the importance of features using the simple model, and converting this feature importance into a map that indicates the parts of the image that are most important to the model.

You can compare results from the LIME technique to other explainability techniques, such as occlusion sensitivity or Grad-CAM. For examples of how to use these related techniques, see the following examples.

- “Understand Network Predictions Using Occlusion” on page 5-24
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21

Load Pretrained Network and Image

Load the pretrained network GoogLeNet.

```
net = googlenet;
```

Extract the image input size and the output classes of the network.

```
inputSize = net.Layers(1).InputSize(1:2);
classes = net.Layers(end).Classes;
```

Load the image. The image is of a retriever called Sherlock. Resize the image to the network input size.

```
img = imread("sherlock.jpg");
img = imresize(img,inputSize);
```

Classify the image, and display the three classes with the highest classification score in the image title.

```
[YPred,scores] = classify(net,img);
[~,topIdx] = maxk(scores, 3);
topScores = scores(topIdx);
topClasses = classes(topIdx);

imshow(img)
titleString = compose("%s (%.2f)",topClasses,topScores');
title(sprintf(join(titleString, "; ")));
```

GoogLeNet classifies Sherlock as a `golden retriever`. Understandably, the network also assigns a high probability to the `Labrador retriever` class. You can use `imageLIME` to understand which parts of the image the network is using to make these classification decisions.

Identify Areas of an Image the Network Uses for Classification

You can use LIME to find out which parts of the image are important for a class. First, look at the predicted class of `golden retriever`. What parts of the image suggest this class?

By default, `imageLIME` identifies features in the input image by segmenting the image into superpixels. This method of segmentation requires Image Processing Toolbox; however, if you do not have Image Processing Toolbox, you can use the option `"Segmentation","grid"` to segment the image into square features.

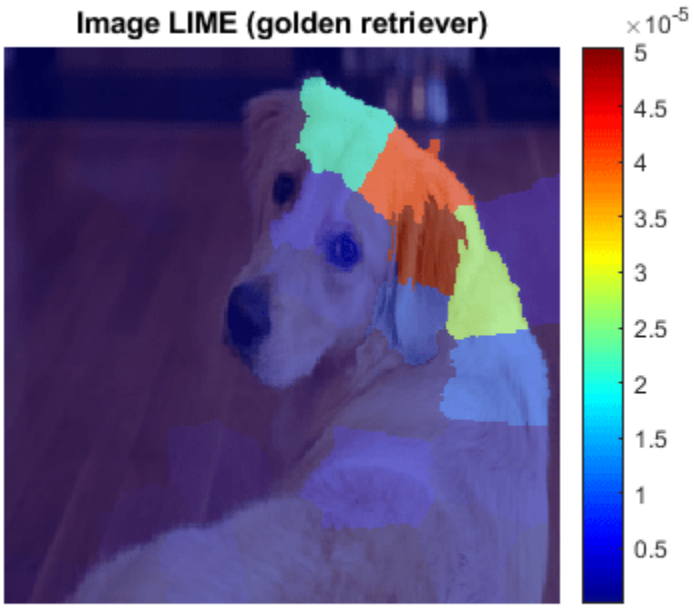
Use the `imageLIME` function to map the importance of different superpixel features. By default, the simple model is a regression tree.

```
map = imageLIME(net,img,YPred);
```

Display the image of Sherlock with the LIME map overlaid.

```
figure
imshow(img,'InitialMagnification',150)
```

```
hold on
imagesc(map,'AlphaData',0.5)
colormap jet
colorbar

title(sprintf("Image LIME (%s)", ...
    YPred))
hold off
```



The maps shows which areas of the image are important to the classification of `golden retriever`. Red areas of the map have a higher importance — when these areas are removed, the score for the `golden retriever` class goes down. The network focuses on the dog's face and ear to make its prediction of golden retriever. This is consistent with other explainability techniques like occlusion sensitivity or Grad-CAM.

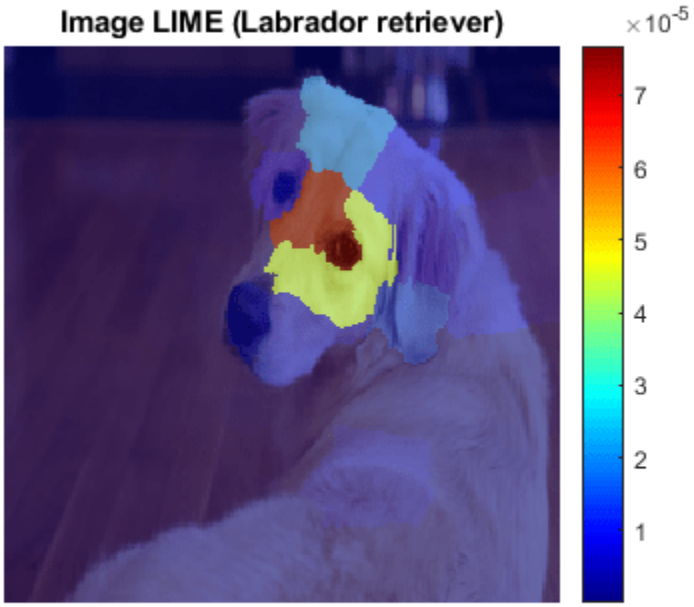
Compare to Results of a Different Class

GoogLeNet predicts a score of 55% for the `golden retriever` class, and 40% for the `Labrador retriever` class. These classes are very similar. You can determine which parts of the dog are more important for both classes by comparing the LIME maps computed for each class.

Using the same settings, compute the LIME map for the `Labrador retriever` class.

```
secondClass = topClasses(2);
map = imageLIME(net,img,secondClass);
figure;
```

```
imshow(img,'InitialMagnification',150)
hold on
imagesc(map,'AlphaData',0.5)
colormap jet
colorbar

title(sprintf("Image LIME (%s)",secondClass))
hold off
```



For the `Labrador retriever` class, the network is more focused on the dog's nose and eyes, rather than the ear. While both maps highlight the dog's forehead, the network has decided that the dog's ear and neck indicate the `golden retriever` class, while the dog's eye and nose indicate the `Labrador retriever` class.

Compare LIME with Grad-CAM

Other image interpretability techniques such as Grad-CAM upsample the resulting map to produce a smooth heatmap of the important areas of the image. You can produce similar-looking maps with `imageLIME`, by calculating the importance of square or rectangular features and upsampling the resulting map.

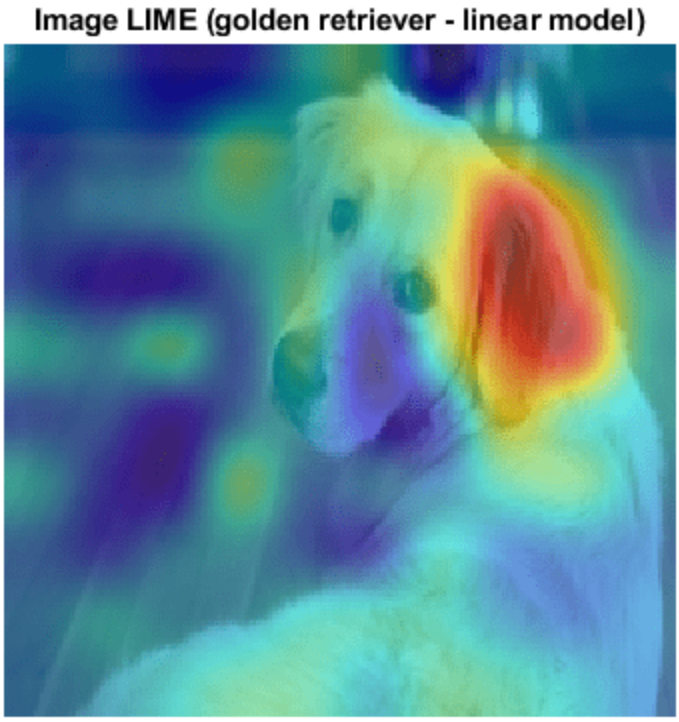
To segment the image into a grid of square features instead of irregular superpixels, use the `"Segmentation","grid"` name-value pair. Upsample the computed map to match the image resolution using bicubic interpolation, by setting `"OutputUpsampling","bicubic"`.

To increase the resolution of the initially computed map, increase the number of features to 100 by specifying the `"NumFeatures",100` name-value pair. As the image is square, this produces a 10-by-10 grid of features.

The LIME technique generates synthetic images based on the original observation by randomly choosing some features and replacing all the pixels in those features with the average image pixel, effectively removing that feature. Increase the number of random samples to 6000 by setting `"NumSamples",6000`. When you increase the number of features, increasing the number of samples usually gives better results.

By default the `imageLIME` function uses a regression tree as its simple model. Instead, fit a linear regression model with lasso regression by setting `"Model","linear"`.

```
map = imageLIME(net,img,"golden retriever", ...
    "Segmentation","grid",...
    "OutputUpsampling","bicubic",...
    "NumFeatures",100,...
    "NumSamples",6000,...
    "Model","linear");

imshow(img,'InitialMagnification', 150)
hold on
imagesc(map,'AlphaData',0.5)
colormap jet

title(sprintf("Image LIME (%s - linear model)", ...
    YPred))
hold off
```



Similar to the gradient map computed by Grad-CAM, the LIME technique also strongly identifies the dog's ear as significant to the prediction of `golden retriever`.

Display Only the Most Important Features

LIME results are often plotted by showing only the most important few features. When you use the `imageLIME` function, you can also obtain a map of the features used in the computation and the calculated importance of each feature. Use these results to determine the four most important superpixel features and display only the four most important features in an image.

Compute the LIME map and obtain the feature map and the calculated importance of each feature.

```
[map,featureMap,featureImportance] = imageLIME(net,img,YPred);
```

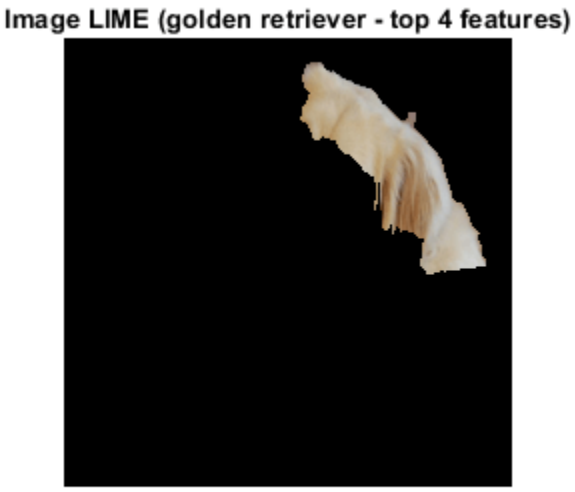
Find the indices of the top four features.

```
numTopFeatures = 4;
[~,idx] = maxk(featureImportance,numTopFeatures);
```

Next, mask out the image using the LIME map so only pixels in the most important four superpixels are visible. Display the masked image.

```
mask = ismember(featureMap,idx);
maskedImg = uint8(mask).*img;

figure
imshow(maskedImg);
```

```
title(sprintf("Image LIME (%s - top %i features)", ...
    YPred, numTopFeatures))
```



References

[1] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "'Why Should I Trust You?': Explaining the Predictions of Any Classifier." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–44. San Francisco California USA: ACM, 2016. https://doi.org/10.1145/2939672.2939778.

See Also

`DAGNetwork` | `googlenet` | `occlusionSensitivity` | `imageLIME` | `gradCAM`

More About

- "Investigate Spectrogram Classifications Using LIME" on page 5-49
- "Interpret Deep Network Predictions on Tabular Data Using LIME" on page 5-59
- "Deep Learning Visualization Methods" on page 5-186
- "Understand Network Predictions Using Occlusion" on page 5-24
- "Grad-CAM Reveals the Why Behind Deep Learning Decisions" on page 5-21

Investigate Spectrogram Classifications Using LIME

This example shows how to use locally interpretable model-agnostic explanations (LIME) to investigate the robustness of a deep convolutional neural network trained to classify spectrograms. LIME is a technique for visualizing which parts of an observation contribute to the classification decision of a network. This example uses the `imageLIME` function to understand which features in the spectrogram data are most important for classification.

In this example, you create and train a neural network to classify four kinds of simulated time series data:

- Sine waves of a single frequency
- Superposition of three sine waves
- Broad Gaussian peaks in the time series
- Gaussian pulses in the time series

To make this problem more realistic, the time series include added confounding signals: a constant low-frequency background sinusoid and a large amount of high-frequency noise. Noisy time series data is a challenging sequence classification problem. You can approach the problem by first converting the time series data into a time-frequency spectrogram to reveal the underlying features in the time series data. You can then input the spectrograms to an image classification network.

Generate Waveforms and Spectrograms

Generate time series data for the four classes. This example uses the helper function `generateSpectrogramData` to generate the time series and the corresponding spectrogram data. The helper functions used in this example are attached as supporting files.

```
numObsPerClass = 500;

classes = categorical(["SingleFrequency","ThreeFrequency","Gaussian","Pulse"]);
numClasses = length(classes);

[noisyTimeSeries,spectrograms,labels] = generateSpectrogramData(numObsPerClass,classes);
```

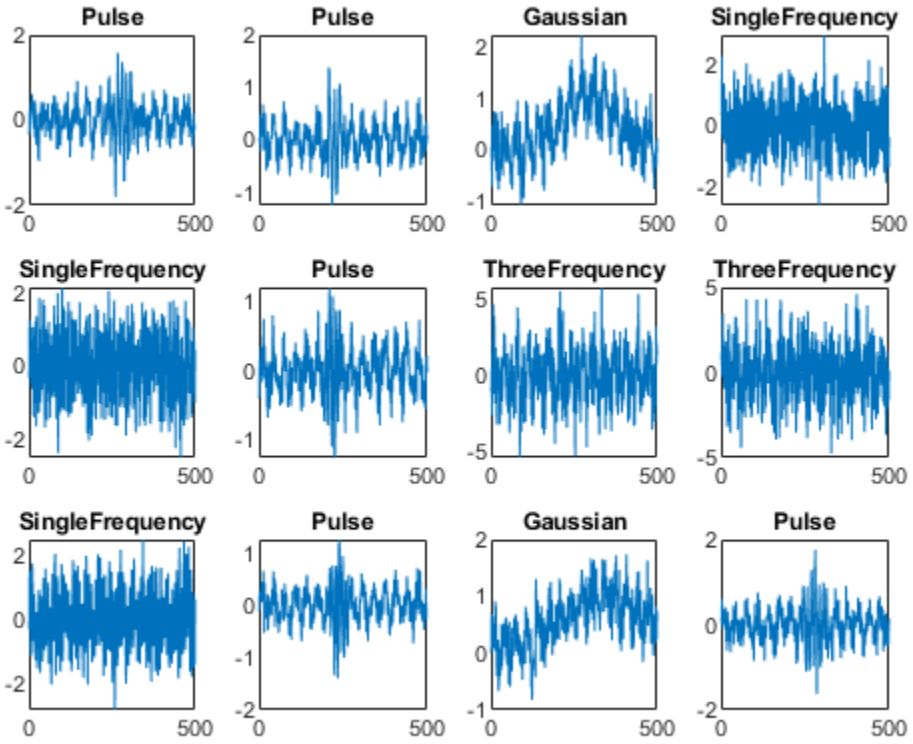
Compute the size of the spectrogram images and the number of observations.

```
inputSize = size(spectrograms, [1 2]);
numObs = size(spectrograms,4);
```

Plot Generated Data

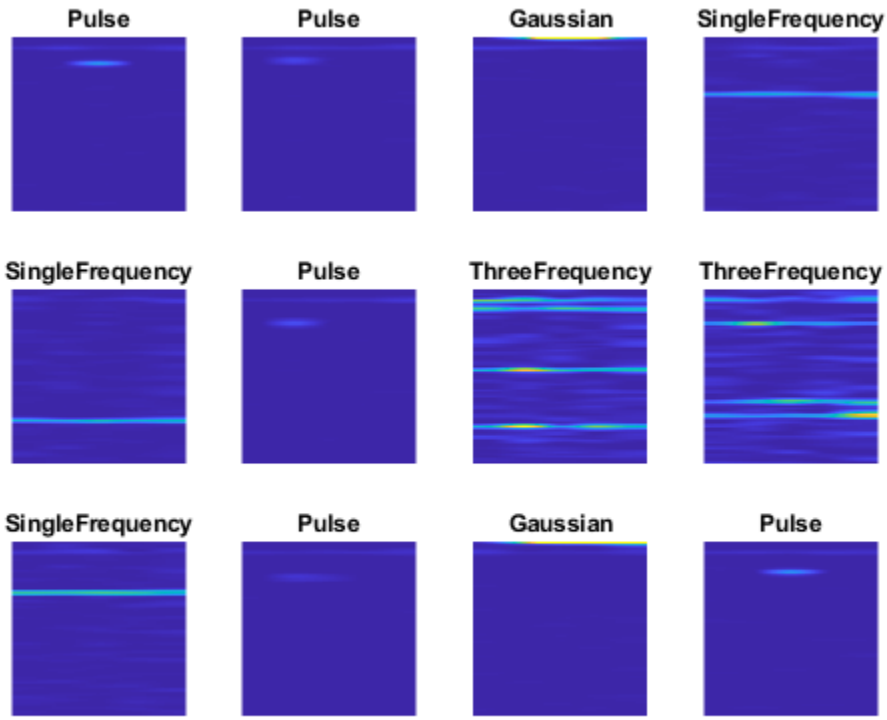
Plot a subset of the time series data with noise added. Because the noise has a comparable amplitude to the signal, the data appears noisy in the time domain. This feature makes classification a challenging problem.

```
figure
numPlots = 12;

for i=1:numPlots
    subplot(3,4,i)
    plot(noisyTimeSeries(i,:))
    title(labels(i))
end
```



Plot the time-frequency spectrograms of the noisy data, in the same order as the time series plots. The horizontal axis is time and the vertical axis is frequency.

```
figure
for i=1:12
    subplot(3,4,i)
    imshow(spectrograms(:,:,1,i))
    hold on
    colormap parula
    title(labels(i))
    hold off
end
```



Features from each class are clearly visible, demonstrating why converting from the time domain to spectrogram images can be beneficial for this type of problem. For example, the `SingleFrequency` class has a single peak at the fundamental frequency, visible as a horizontal bar in the spectrogram. For the `ThreeFrequency` class, the three frequencies are visible.

All classes display a faint band at low frequency (near the top of the image), corresponding to the background sinusoid.

Split Data

Use the `splitlabels` function to divide the data into training and validation data. Use 80% of the data for training and 20% for validation.

```
splitIndices = splitlabels(labels,0.8);

trainLabels = labels(splitIndices{1});
trainSpectrograms = spectrograms(:,:,:,splitIndices{1});

valLabels = labels(splitIndices{2});
valSpectrograms = spectrograms(:,:,:,splitIndices{2});
```

Define Neural Network Architecture

Create a convolutional neural network with blocks of convolution, batch normalization, and ReLU layers.

```
dropoutProb = 0.2;
numFilters = 8;
```

```
layers = [
    imageInputLayer(inputSize)

    convolution2dLayer(3,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numFilters,'Padding','same')
    batchNormalizationLayer

    convolution2dLayer(3,4*numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    globalMaxPooling2dLayer

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Define Training Options

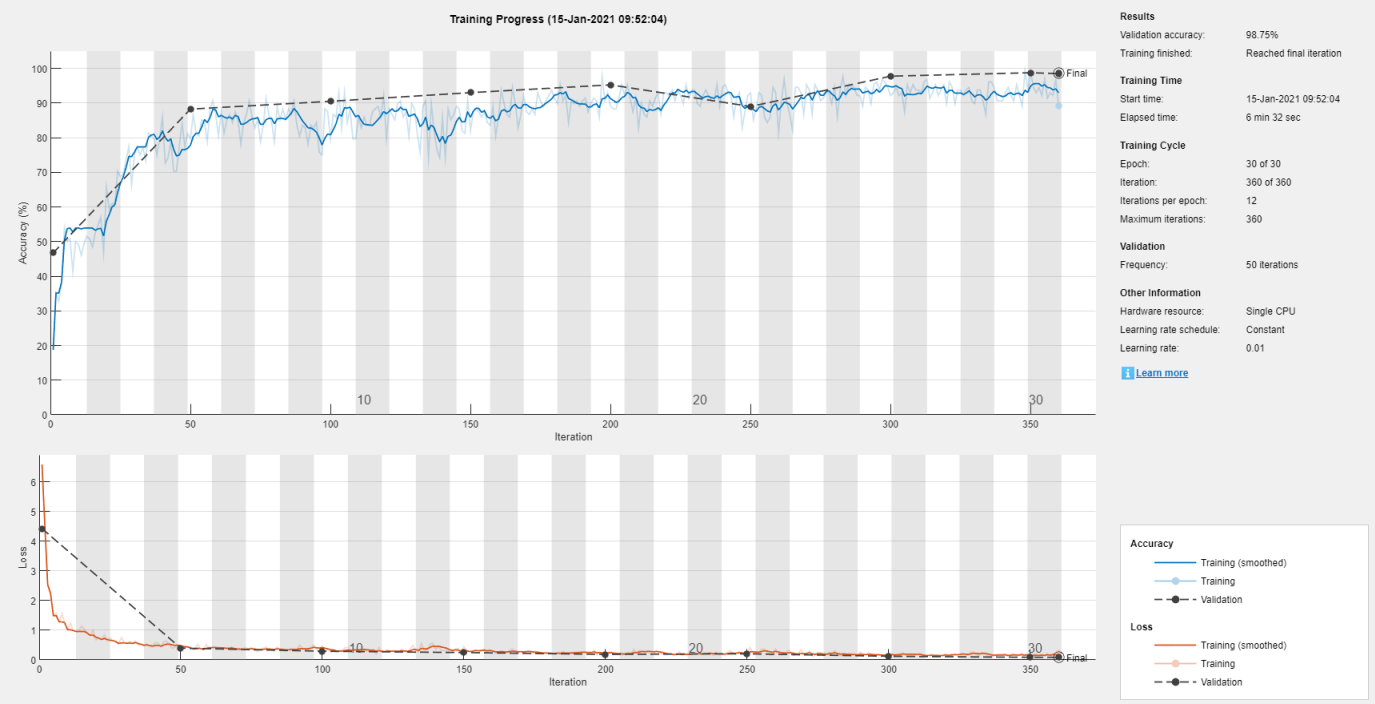
Define options for training using the SGDM optimizer. Shuffle the data every epoch by setting the `'Shuffle'` option to `'every-epoch'`. Monitor the training progress by setting the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

```
options = trainingOptions('sgdm', ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{valSpectrograms,valLabels});
```

Train Network

Train the network to classify the spectrogram images.

```
net = trainNetwork(trainSpectrograms,trainLabels,layers,options);
```



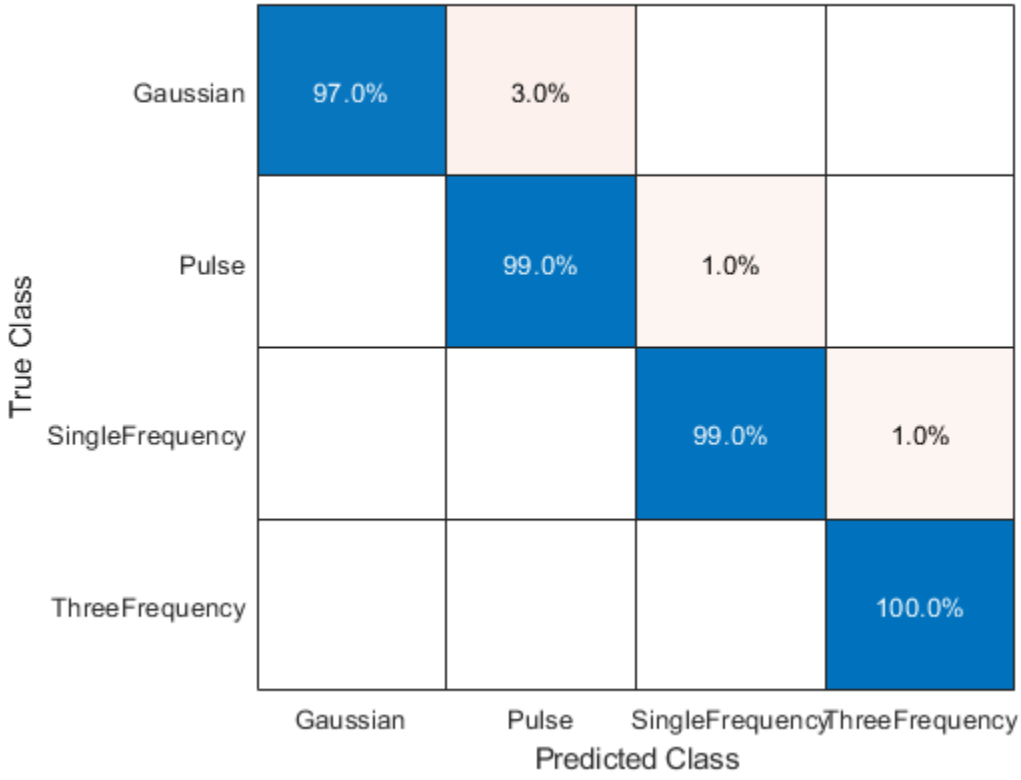
Accuracy

Classify the validation observations using the trained network.

```
predLabels = classify(net,valSpectrograms);
```

Investigate the network performance by plotting a confusion matrix with `confusionchart`.

```
figure
confusionchart(valLabels,predLabels,'Normalization','row-normalized')
```



The network accurately classifies the validation spectrograms, with close to 100% accuracy for most of the classes.

Investigate Network Predictions

Use the `imageLIME` function to understand which features in the image data are most important for classification.

The LIME technique segments an image into several features and generates synthetic observations by randomly including or excluding features. Each pixel in an excluded feature is replaced with the value of the average image pixel. The network classifies these synthetic observations, and uses the resulting scores for the predicted class, along with the presence or absence of a feature, as responses and predictors to train a regression problem with a simpler model—in this example, a regression tree. The regression tree tries to approximate the behavior of the network on a single observation. It learns which features are important and significantly impact the class score.

Define Custom Segmentation Map

By default, `imageLIME` uses superpixel segmentation to divide the image into features. This option works well for natural images, but is less effective for spectrogram data. You can specify a custom segmentation map by setting the `'Segmentation'` name-value argument to a numeric array the same size as the image, where each element is an integer corresponding to the index of the feature that pixel is in.

For the spectrogram data, the spectrogram images have much finer features in the y-dimension (frequency) than the x-dimension (time). Generate a segmentation map with 240 segments, in a 40-

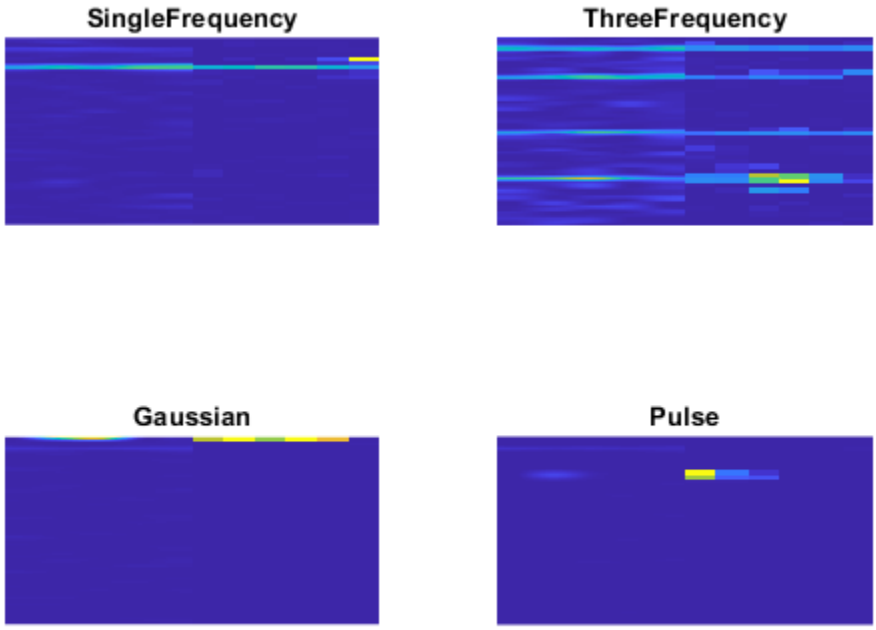
by-6 grid, to provide higher frequency resolution. Upsample the grid to the size of the image by using the `imresize` function, specifying the upsampling method as `'nearest'`.

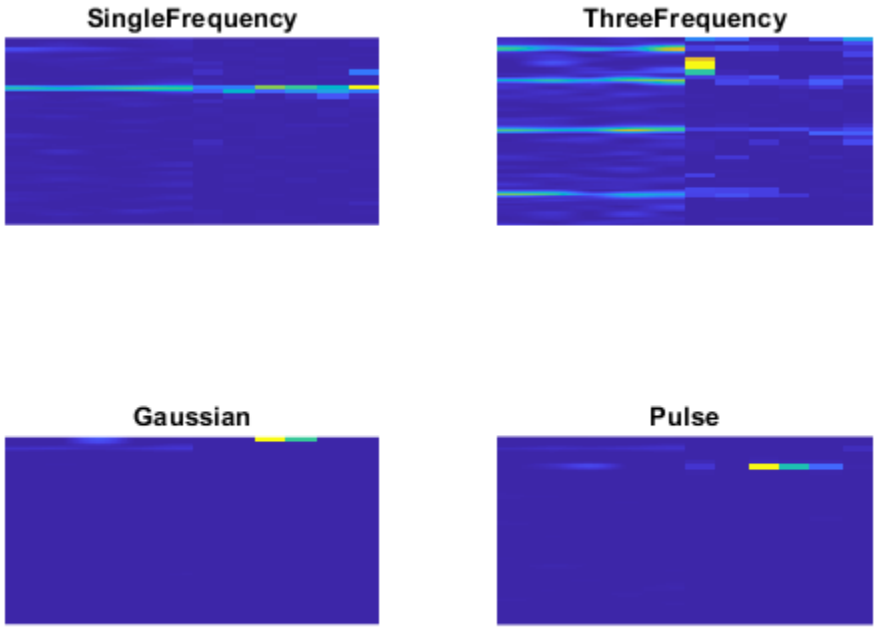
```
featureIdx = 1:240;
segmentationMap = reshape(featureIdx,6,40)';
segmentationMap = imresize(segmentationMap,inputSize,'nearest');
```

Compute LIME Map

Plot the spectrogram and compute the LIME map for two observations from each class.

```
obsToShowPerClass = 2;

for j=1:obsToShowPerClass
    figure

    for i=1:length(classes)

        idx = find(valLabels == classes(i),obsToShowPerClass);

        % Read the test image and label.
        testSpectrogram = valSpectrograms(:,:,:,idx(j));
        testLabel = valLabels(idx(j));

        % Compute the LIME importance map.
        map = imageLIME(net,testSpectrogram,testLabel, ...
            'NumSamples',4096, ...
            'Segmentation',segmentationMap);

        % Rescale the map to the size of the image.
        mapRescale = uint8(255*rescale(map));

        % Plot the spectrogram image next to the LIME map.
        subplot(2,2,i)
        imshow(imtile({testSpectrogram,mapRescale}))
        title(string(testLabel))
        colormap parula
    end
end
```


SingleFrequency
ThreeFrequency
Gaussian
Pulse



The LIME maps demonstrate that for most classes, the network is focused on the relevant features for classification. For example, for the `SingleFrequency` class, the network focuses on the frequency corresponding to the power spectrum of the sine wave and not on spurious background details or noise.

For the `SingleFrequency` class, the network uses the frequency to classify. For the `Pulse` and `Gaussian` classes, the network additionally focuses on the correct frequency part of the spectrogram. For these three classes, the network is not confused by the background frequency visible near the top of all of the spectrograms. This information is not helpful for distinguishing between these classes (as it is present in all classes), so the network ignores it. In contrast, for the `ThreeFrequency` class, the constant background frequency is relevant to the classification decision of the network. For this class, the network does not ignore this frequency, but treats it with similar importance to the three actual frequencies.

The `imageLIME` results demonstrate that the network is correctly using peaks in the time-frequency spectrograms and is not confused by the spurious background sinusoid for all classes except for the `ThreeFrequency` class, where the network does not distinguish between the three frequencies in the signal and the low-frequency background.

See Also

`imageLIME` | `pspectrum` | `trainNetwork`

More About

- “Understand Network Predictions Using LIME” on page 5-42
- “Interpret Deep Network Predictions on Tabular Data Using LIME” on page 5-59
- “Deep Learning Visualization Methods” on page 5-186
- “Understand Network Predictions Using Occlusion” on page 5-24
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21

Interpret Deep Network Predictions on Tabular Data Using LIME

This example shows how to use the locally interpretable model-agnostic explanations (LIME) technique to understand the predictions of a deep neural network classifying tabular data. You can use the LIME technique to understand which predictors are most important to the classification decision of a network.

In this example, you interpret a feature data classification network using LIME. For a specified query observation, LIME generates a synthetic data set whose statistics for each feature match the real data set. This synthetic data set is passed through the deep neural network to obtain a classification, and a simple, interpretable model is fitted. This simple model can be used to understand the importance of the top few features to the classification decision of the network. In training this interpretable model, synthetic observations are weighted by their distance from the query observation, so the explanation is "local" to that observation.

This example uses `lime` (Statistics and Machine Learning Toolbox) and `fit` (Statistics and Machine Learning Toolbox) to generate a synthetic data set and fit a simple interpretable model to the synthetic data set. To understand the predictions of a trained image classification neural network, use `imageLIME`. For more information, see “Understand Network Predictions Using LIME” on page 5-42.

Load Data

Load the Fisher iris data set. This data contains 150 observations with four input features representing the parameters of the plant and one categorical response representing the plant species. Each observation is classified as one of the three species: setosa, versicolor, or virginica. Each observation has four measurements: sepal width, sepal length, petal width, and petal length.

```
filename = fullfile(toolboxdir('stats'),'statsdemos','fisheriris.mat');
load(filename)
```

Convert the numeric data to a table.

```
features = ["Sepal length","Sepal width","Petal length","Petal width"];

predictors = array2table(meas,"VariableNames",features);
trueLabels = array2table(categorical(species),"VariableNames","Response");
```

Create a table of training data whose final column is the response.

```
data = [predictors trueLabels];
```

Calculate the number of observations, features, and classes.

```
numObservations = size(predictors,1);
numFeatures = size(predictors,2);
numClasses = length(categories(data{:,5}));
```

Split Data into Training, Validation, and Test Sets

Partition the data set into training, validation, and test sets. Set aside 15% of the data for validation and 15% for testing.

Determine the number of observations for each partition. Set the random seed to make the data splitting and CPU training reproducible.

```
rng('default');
numObservationsTrain = floor(0.7*numObservations);
numObservationsValidation = floor(0.15*numObservations);
```

Create an array of random indices corresponding to the observations and partition it using the partition sizes.

```
idx = randperm(numObservations);
idxTrain = idx(1:numObservationsTrain);
idxValidation = idx(numObservationsTrain + 1:numObservationsTrain + numObservationsValidation);
idxTest = idx(numObservationsTrain + numObservationsValidation + 1:end);
```

Partition the table of data into training, validation, and testing partitions using the indices.

```
dataTrain = data(idxTrain,:);
dataVal = data(idxValidation,:);
dataTest = data(idxTest,:);
```

Define Network Architecture

Create a simple multi-layer perceptron, with a single hidden layer with five neurons and ReLU activations. The feature input layer accepts data containing numeric scalars representing features, such as the Fisher iris data set.

```
numHiddenUnits = 5;
layers = [
    featureInputLayer(numFeatures)
    fullyConnectedLayer(numHiddenUnits)
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Define Training Options and Train Network

Train the network using stochastic gradient descent with momentum (SGDM). Set the maximum number of epochs to 30 and use a mini-batch size of 15, as the training data does not contain many observations.

```
opts = trainingOptions("sgdm", ...
    "MaxEpochs",30, ...
    "MiniBatchSize",15, ...
    "Shuffle","every-epoch", ...
    "ValidationData",dataVal, ...
    "ExecutionEnvironment","cpu");
```

Train the network.

```
net = trainNetwork(dataTrain,layers,opts);
```

```
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:00 |       40.00% |       31.82% |       1.3060 |       1.2
|       8 |          50 |       00:00:00 |       86.67% |       90.91% |       0.4223 |       0.3
|      15 |         100 |       00:00:00 |       93.33% |       86.36% |       0.2947 |       0.2
|      22 |         150 |       00:00:00 |       86.67% |       81.82% |       0.2804 |       0.3
```

```
|      29 |         200 |       00:00:01 |       86.67% |       90.91% |       0.2268 |       0.2
|      30 |         210 |       00:00:01 |       93.33% |       95.45% |       0.2782 |       0.1
|=========================================================================================================
```

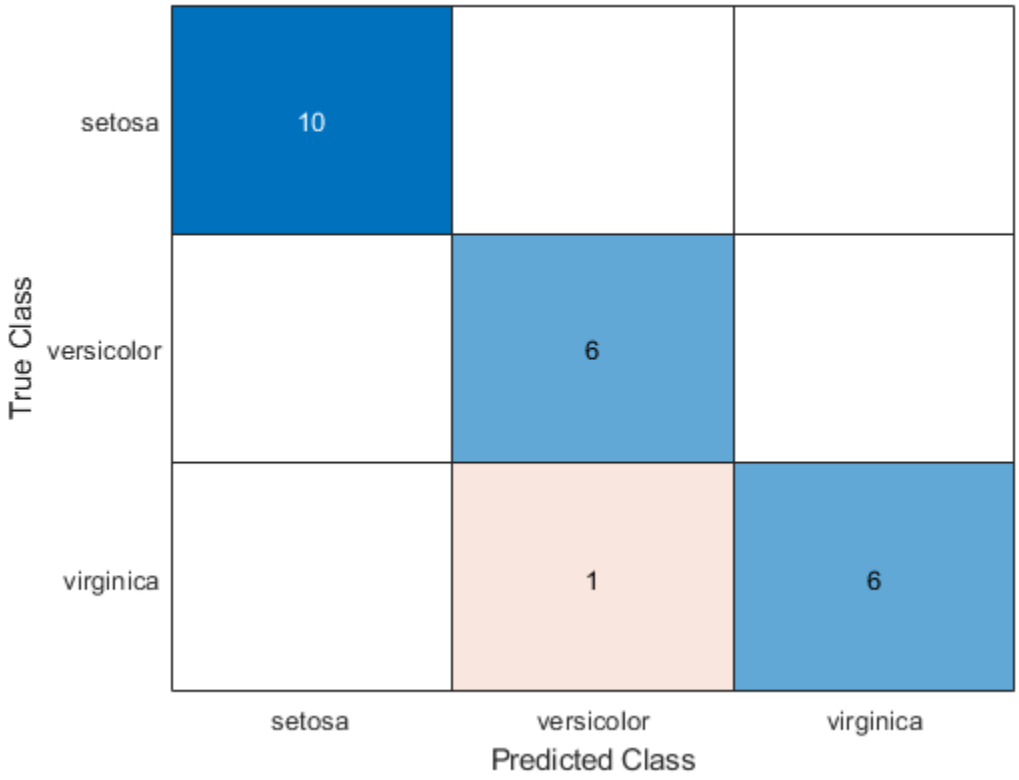
Assess Network Performance

Classify observations from the test set using the trained network.

```
predictedLabels = net.classify(dataTest);
trueLabels = dataTest{:,end};
```

Visualize the results using a confusion matrix.

```
figure
confusionchart(trueLabels,predictedLabels)
```



The network successfully uses the four plant features to predict the species of the test observations.

Understand How Different Predictors Are Important to Different Classes

Use LIME to understand the importance of each predictor to the classification decisions of the network.

Investigate the two most important predictors for each observation.

```
numImportantPredictors = 2;
```

Use `lime` to create a synthetic data set whose statistics for each feature match the real data set. Create a `lime` object using a deep learning model `blackbox` and the predictor data contained in

`predictors`. Use a low `'KernelWidth'` value so `lime` uses weights that are focused on the samples near the query point.

```
blackbox = @(x)classify(net,x);
explainer = lime(blackbox,predictors,'Type','classification','KernelWidth',0.1);
```

You can use the LIME explainer to understand the most important features to the deep neural network. The function estimates the importance of a feature by using a simple linear model that approximates the neural network in the vicinity of a query observation.

Find the indices of the first two observations in the test data corresponding to the setosa class.

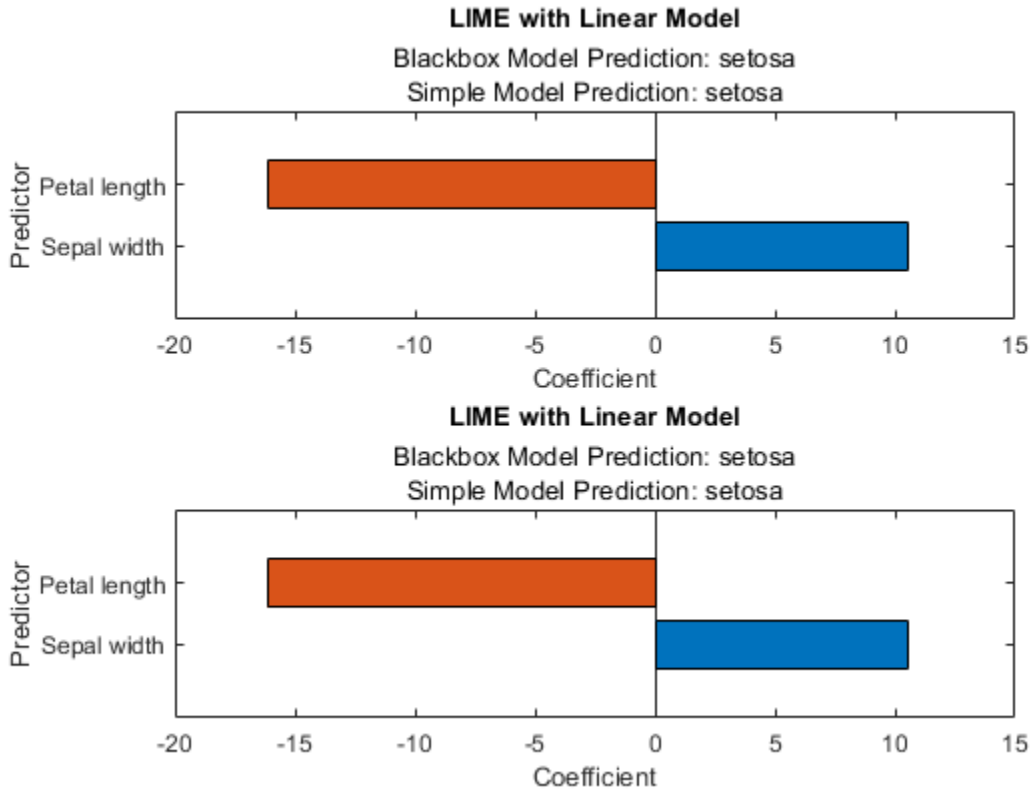
```
trueLabelsTest = dataTest{:,end};

label = "setosa";
idxSetosa = find(trueLabelsTest == label,2);
```

Use the `fit` function to fit a simple linear model to the first two observations from the specified class.

```
explainerObs1 = fit(explainer,dataTest(idxSetosa(1),1:4),numImportantPredictors);
explainerObs2 = fit(explainer,dataTest(idxSetosa(2),1:4),numImportantPredictors);
```

Plot the results.

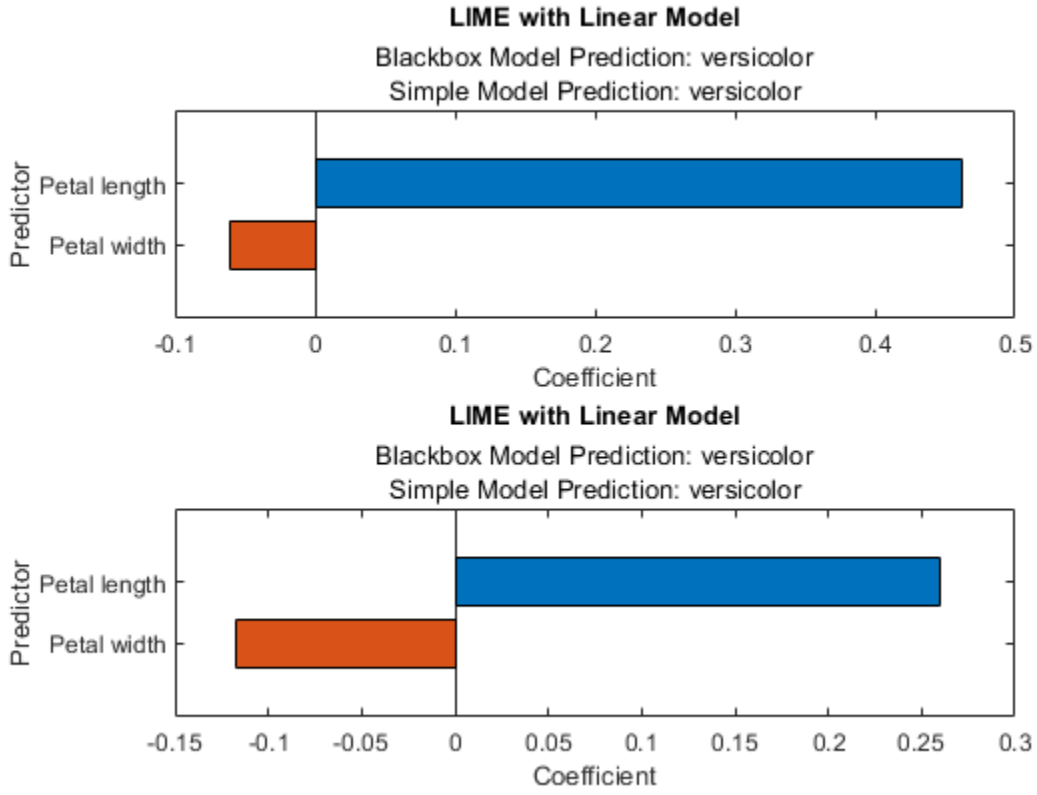
```
figure
subplot(2,1,1)
plot(explainerObs1);
subplot(2,1,2)
plot(explainerObs2);
```



For the setosa class, the most important predictors are a low petal length value and a high sepal width value.

Perform the same analysis for class versicolor.

```
label = "versicolor";
idxVersicolor = find(trueLabelsTest == label,2);

explainerObs1 = fit(explainer,dataTest(idxVersicolor(1),1:4),numImportantPredictors);
explainerObs2 = fit(explainer,dataTest(idxVersicolor(2),1:4),numImportantPredictors);

figure
subplot(2,1,1)
plot(explainerObs1);
subplot(2,1,2)
plot(explainerObs2);
```

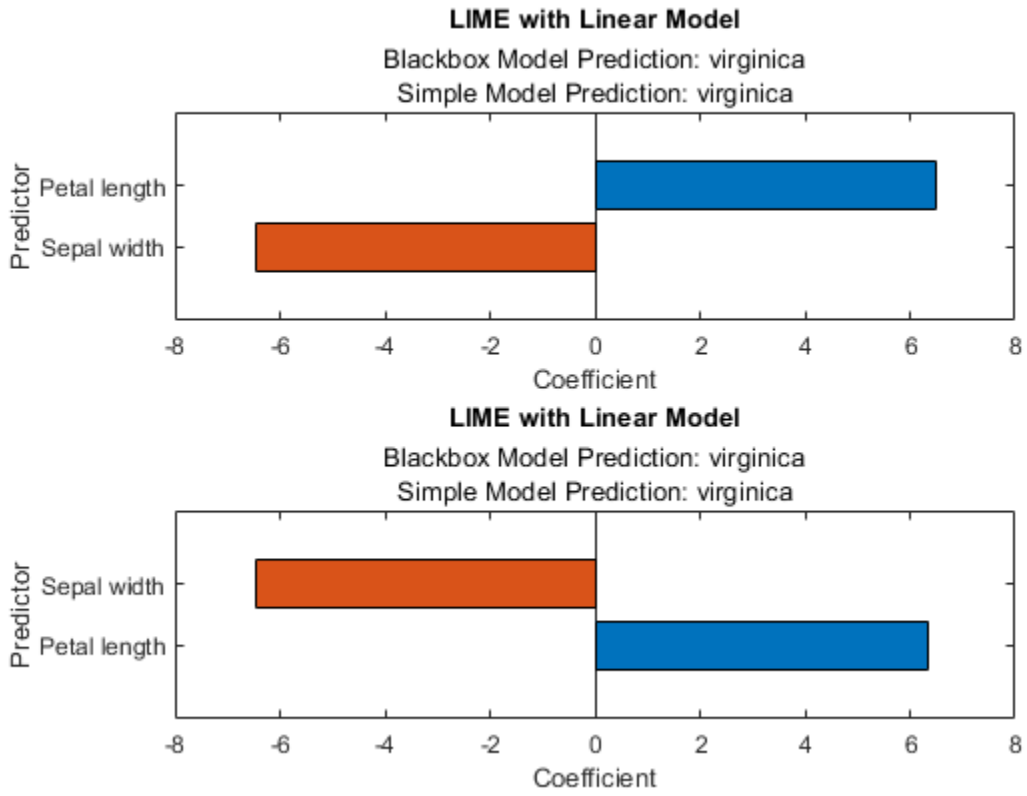


For the versicolor class, a high petal length value is important.

Finally, consider the virginica class.

```
label = "virginica";
idxVirginica = find(trueLabelsTest == label,2);

explainerObs1 = fit(explainer,dataTest(idxVirginica(1),1:4),numImportantPredictors);
explainerObs2 = fit(explainer,dataTest(idxVirginica(2),1:4),numImportantPredictors);

figure
```

```
subplot(2,1,1)
plot(explainerObs1);
subplot(2,1,2)
plot(explainerObs2);
```



For the virginica class, a high petal length value and a low sepal width value is important.

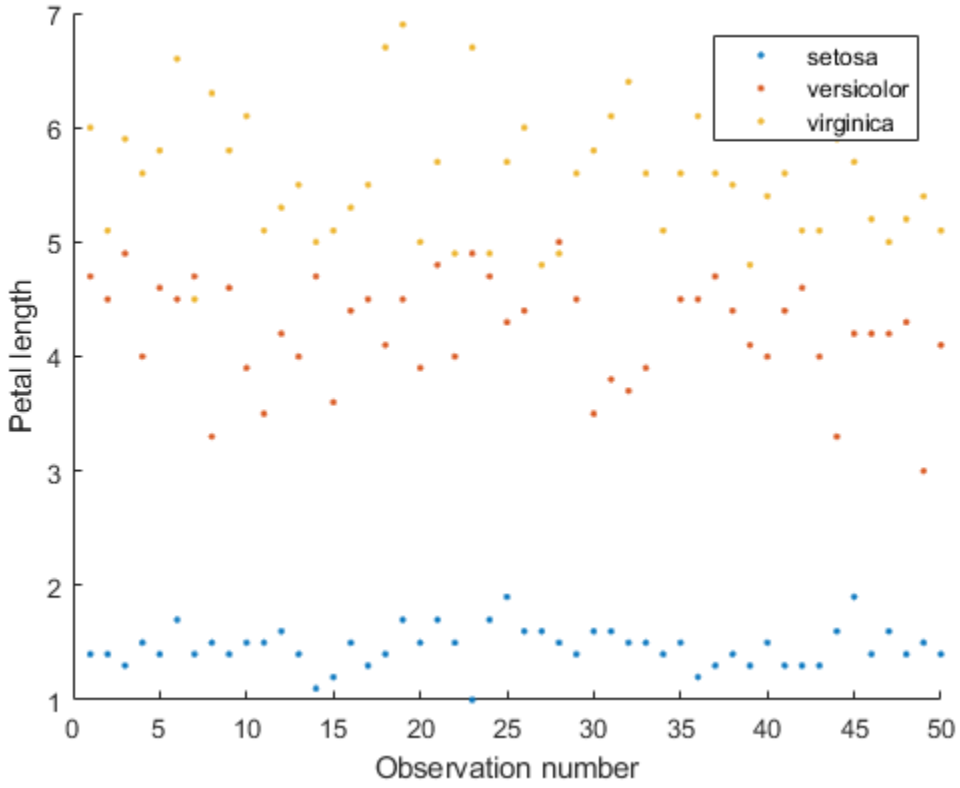
Validate LIME Hypothesis

The LIME plots suggest that a high petal length value is associated with the versicolor and virginica classes and a low petal length value is associated with the setosa class. You can investigate the results further by exploring the data.

Plot the petal length of each image in the data set.

```
setosaIdx = ismember(data{:,end},"setosa");
versicolorIdx = ismember(data{:,end},"versicolor");
virginicaIdx = ismember(data{:,end},"virginica");

figure
hold on
plot(data{setosaIdx,"Petal length"},'.')
plot(data{versicolorIdx,"Petal length"},'.')
plot(data{virginicaIdx,"Petal length"},'.')
hold off

xlabel("Observation number")
```

```
ylabel("Petal length")
legend(["setosa","versicolor","virginica"])
```



The setosa class has much lower petal length values than the other classes, matching the results produced from the `lime` model.

See Also

`fit` | `lime` | `trainNetwork` | `classify` | `featureInputLayer` | `imageLIME`

More About

- “Understand Network Predictions Using LIME” on page 5-42
- “Investigate Spectrogram Classifications Using LIME” on page 5-49
- “Understand Network Predictions Using Occlusion” on page 5-24
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Deep Learning Visualization Methods” on page 5-186

Explore Semantic Segmentation Network Using Grad-CAM

This example shows how to explore the predictions of a semantic segmentation network using Grad-CAM.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. You can use Grad-CAM, a deep learning visualization technique, to see which regions of the image are important for the pixel classification decision.

Load Data Set

This example uses the CamVid data set [1] from the University of Cambridge for training. This data set is a collection of images containing street-level views obtained while driving. The data set provides pixel-level labels for 32 semantic classes, including car, pedestrian, and road.

Download CamVid Data Set

Download the CamVid data set.

```
rng('default')

imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

outputFolder = fullfile(tempdir,'CamVid');
labelsZip = fullfile(outputFolder,'labels.zip');
imagesZip = fullfile(outputFolder,'images.zip');

if ~exist(labelsZip, 'file') || ~exist(imagesZip,'file')
    mkdir(outputFolder)

    disp('Downloading 16 MB CamVid data set labels...');
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder,'labels'));

    disp('Downloading 557 MB CamVid data set images...');
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder,'images'));
end
```

```
Downloading 16 MB CamVid data set labels...

Downloading 557 MB CamVid data set images...
```

Load CamVid Images

Use an `imageDatastore` to load the CamVid images. The `imageDatastore` enables you to efficiently load a large collection of images on disk.

```
imgDir = fullfile(outputFolder,'images','701_StillsRaw_full');
imds = imageDatastore(imgDir);
```

The data set contains 32 classes. To make training easier, reduce the number of classes to 11 by grouping multiple classes from the original data set together. For example, create a "`Car`" class that combines the "`Car`", "`SUVPickupTruck`", "`Truck_Bus`", "`Train`", and "`OtherMoving`" classes from the original data set. Return the grouped label IDs by using the supporting function `camvidPixelLabelIDs`, which is listed at the end of this example.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];

labelIDs = camvidPixelLabelIDs;
```

Use the classes and label IDs to create a `pixelLabelDatastore`.

```
labelDir = fullfile(outputFolder,'labels');
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Load Pretrained Semantic Segmentation Network

Load a pretrained semantic segmentation network. The pretrained model allows you to run the entire example without having to wait for training to complete. This example loads a trained Deeplab v3+ network with weights initialized from a pretrained ResNet-18 network. To get a pretrained ResNet-18, install `resnet18`. For more information on building and training a semantic segmentation network, see Semantic Segmentation Using Deep Learning.

```
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/deeplabv3plusResnet18CamVid.m
pretrainedFolder = fullfile(tempdir,'pretrainedNetwork');
pretrainedNetwork = fullfile(pretrainedFolder,'deeplabv3plusResnet18CamVid.mat');

if ~exist(pretrainedNetwork,'file')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetwork,pretrainedURL);
end
```

```
Downloading pretrained network (58 MB)...
```

```
pretrainedNet = load(pretrainedNetwork);
net = pretrainedNet.net;
```

Test Network

The trained semantic segmentation network predicts the label of each pixel within an image. You can test the network by predicting the pixel labels of an image.

Load a test image.

```
figure
img = readimage(imds,615);
imshow(img,'InitialMagnification',35)
```

Use the `semanticseg` function to predict the pixel labels of the image by using the trained semantic segmentation network.

```
predLabels = semanticseg(img,net);
```

Display the results.

```
cmap = camvidColorMap;
segImg = labeloverlay(img,predLabels,'Colormap',cmap,'Transparency',0.4);
figure
imshow(segImg,'InitialMagnification',40)

pixelLabelColorbar(cmap,classes)
```

You can see that the network labels the parts of the image fairly accurately. The network does misclassify some areas, for example, the road to the left of the intersection, which is partially misclassified as pavement.

Explore Network Predictions

Deep networks are complex, so understanding how a network determines a particular prediction is difficult. You can use Grad-CAM to see which areas of the test image the semantic segmentation network is using to make its pixel classifications.

Grad-CAM computes the gradient of a differentiable output, such as class score, with respect to the convolutional features in a chosen layer. Grad-CAM is typically used for image classification tasks [2]; however, it can also be extended to semantic segmentation problems [3].

In semantic segmentation tasks, the softmax layer of the network outputs a score for each class for every pixel in the original image. This contrasts with standard image classification problems, where the softmax layer outputs a score for each class for the entire image. The Grad-CAM map for class c is

$$M^c = ReLU(\sum_k \alpha_c^k A^k) \text{ where } \alpha_c^k = 1/N \sum_{i,j} \frac{dy^c}{dA_{i,j}^k}$$

N is the number of pixels, A^k is the feature map of interest, and y^c corresponds to a scalar class score. For a simple image classification problem, y^c is the softmax score for the class of interest. For semantic segmentation, you can obtainy^c by reducing the pixel-wise class scores for the class of interest to a scalar. For example, sum over the spatial dimensions of the softmax layer:

$y^C = \sum_{(i,j) \in P} y^C_{i,j}$, where P is the pixels in the output layer of a semantic segmentation network [3]. In this example, the output layer is the softmax layer before the pixel classification layer. The map M^C highlights areas that influence the decision for class c. Higher values indicate regions of the image that are important for the pixel classification decision.

To use Grad-CAM, you must select a feature layer to extract the feature map from and a reduction layer to extract the output activations from. Use `analyzeNetwork` to find the layers to use with Grad-CAM.

```
analyzeNetwork(net)
```

Specify a feature layer. Typically this is a ReLU layer which takes the output of a convolutional layer at the end of the network.

```
featureLayer = 'dec_relu4';
```

Specify a reduction layer. The `gradCAM` function sums the spatial dimensions of the reduction layer, for the specified classes, to produce a scalar value. This scalar value is then differentiated with respect to each feature in the feature layer. For semantic segmentation problems, the reduction layer is usually the softmax layer.

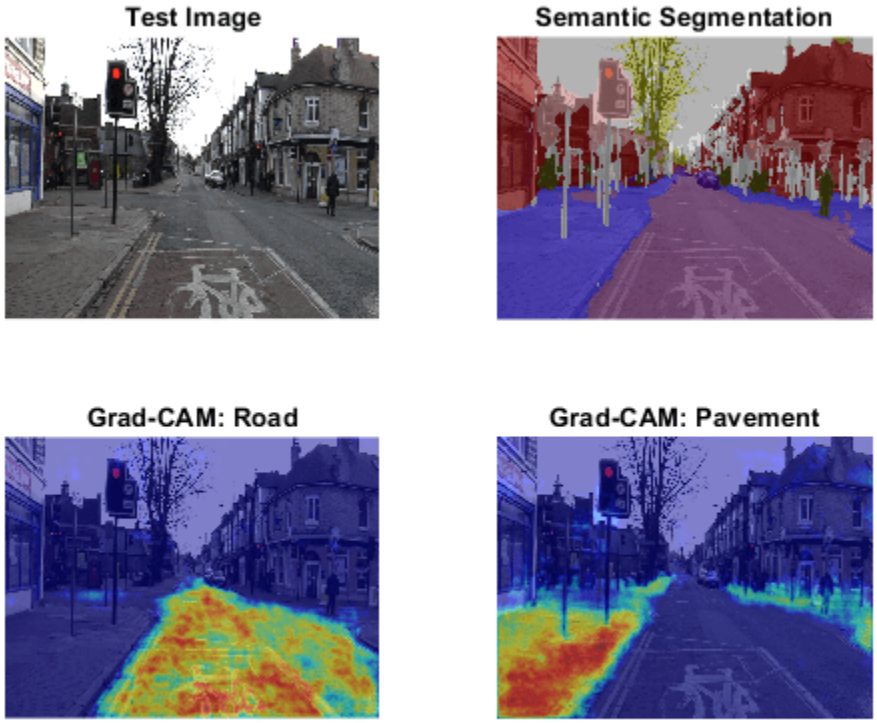
```
reductionLayer = 'softmax-out';
```

Compute the Grad-CAM map for the road and pavement classes.

```
classes = ["Road" "Pavement"];

gradCAMMap = gradCAM(net,img,classes, ...
    'ReductionLayer',reductionLayer, ...
    'FeatureLayer',featureLayer);
```

Compare the Grad-CAM map for the two classes to the semantic segmentation map.

```
predLabels = semanticseg(img,net);
segMap = labeloverlay(img,predLabels,'Colormap',cmap,'Transparency',0.4);

figure;
subplot(2,2,1)
imshow(img)
title('Test Image')
subplot(2,2,2)
imshow(segMap)
title('Semantic Segmentation')
subplot(2,2,3)
imshow(img)
hold on
imagesc(gradCAMMap(:,:,1),'AlphaData',0.5)
title('Grad-CAM: ' + classes(1))
colormap jet
subplot(2,2,4)
imshow(img)
hold on
imagesc(gradCAMMap(:,:,2),'AlphaData',0.5)
title('Grad-CAM: ' + classes(2))
colormap jet
```



The Grad-CAM maps and semantic segmentation map show similar highlighting. None of the maps distinguish the road to the left of the intersection, which the semantic segmentation map labels as pavement. The Grad-CAM map for the pavement class shows that the edge of the pavement is more important than the center for the classification decision of the network. The network possibly misclassifies the road to the left of the intersection due to the poor visibility of the pavement edge.

Explore Intermediate Layers

The Grad-CAM map resembles the semantic segmentation map when you use a layer near the end of the network for the computation. You can also use Grad-CAM to investigate intermediate layers in the trained network. Earlier layers have a small receptive field size and learn small, low-level features compared to the layers at the end of the network.

Compute the Grad-CAM map for layers that are successively deeper in the network.

```
layers = ["res5b_relu","catAspp","dec_relu1"];
numLayers = length(layers);
```

The `res5b_relu` layer is near the middle of the network, whereas `dec_relu1` is near the end of the network.

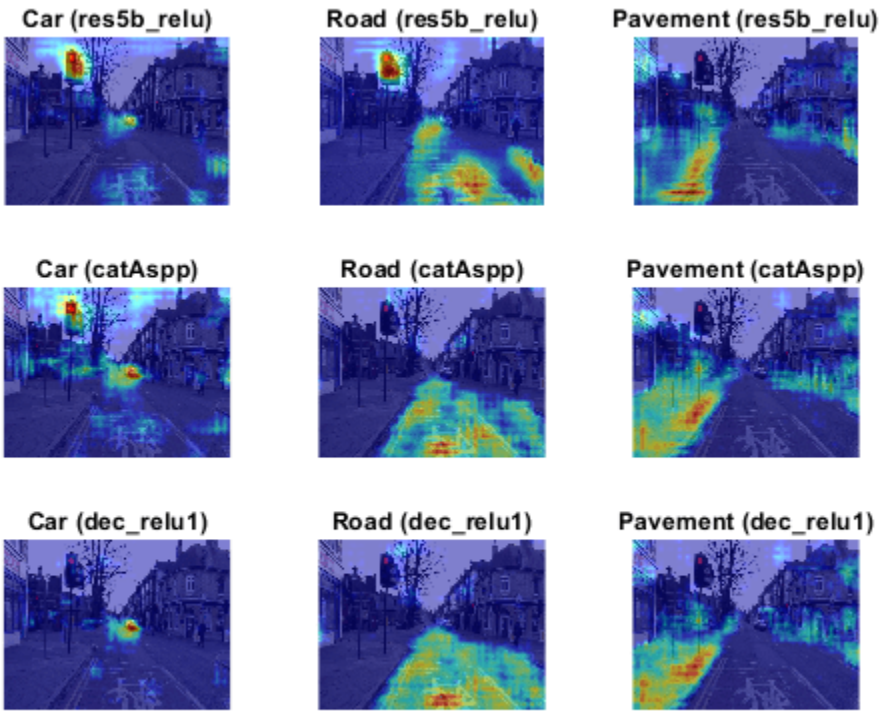
Investigate the network classification decisions for the car, road, and pavement classes. For each layer and class, compute the Grad-CAM map.

```
classes = ["Car" "Road" "Pavement"];
numClasses = length(classes);
```

```
gradCAMMaps = [];
for i = 1:numLayers
    gradCAMMaps(:,:,:,i) = gradCAM(net,img,classes, ...
        'ReductionLayer',reductionLayer, ...
        'FeatureLayer',layers(i));
end
```

Display the Grad-CAM maps for each layer and each class. The rows represent the map for each layer, with the layers ordered from those early in the network to those at the end of the network.

```
figure;
idx = 1;
for i=1:numLayers
    for j=1:numClasses
        subplot(numLayers,numClasses,idx)
        imshow(img)
        hold on
        imagesc(gradCAMMaps(:,:,j,i),'AlphaData',0.5)
        title(sprintf("%s (%s)",classes(j),layers(i)), ...
            "Interpreter","none")
        colormap jet
        idx = idx + 1;
    end
end
```



The later layers produce maps very similar to the segmentation map. However, the layers earlier in the network produce more abstract results and are typically more concerned with lower level

features like edges, with less awareness of semantic classes. For example, in the maps for earlier layers, you can see that for both car and road classes, the traffic light is highlighted. This suggests that the earlier layers focus on areas of the image that are related to the class but do not necessarily belong to it. For example, a traffic light is likely to appear near to a road, so the network might be using this information to predict which pixels are roads. You can also see that for the pavement class, the earlier layers are highly focused on the edge, suggesting this feature is important to the network when detecting which pixels are in the pavement class.

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. “Semantic Object Classes in Video: A High-Definition Ground Truth Database.” *Pattern Recognition Letters* 30, no. 2 (January 2009): 88–97. https://doi.org/10.1016/j.patrec.2008.04.005.

[2] Selvaraju, R. R., M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization." In IEEE International Conference on Computer Vision (ICCV), 2017, pp. 618–626. Available at `Grad-CAM` on the Computer Vision Foundation Open Access website.

[3] Vinogradova, Kira, Alexandr Dibrov, and Gene Myers. “Towards Interpretable Semantic Segmentation via Gradient-Weighted Class Activation Mapping (Student Abstract).” *Proceedings of the AAAI Conference on Artificial Intelligence* 34, no. 10 (April 3, 2020): 13943–44. https://doi.org/10.1609/aaai.v34i10.7244.

Supporting Functions

```
function labelIDs = camvidPixelLabelIDs()
% Return the label IDs corresponding to each class.
%
% The CamVid data set has 32 classes. Group them into 11 classes following
% the original SegNet training methodology [1].
%
% The 11 classes are:
%   "Sky", "Building", "Pole", "Road", "Pavement", "Tree", "SignSymbol",
%   "Fence", "Car", "Pedestrian",  and "Bicyclist".
%
% CamVid pixel label IDs are provided as RGB color values. Group them into
% 11 classes and return them as a cell array of M-by-3 matrices. The
% original CamVid class names are listed alongside each RGB value. Note
% that the Other/Void class are excluded below.
labelIDs = { ...

    % "Sky"
    [
    128 128 128; ... % "Sky"
    ]

    % "Building"
    [
    000 128 064; ... % "Bridge"
    128 000 000; ... % "Building"
    064 192 000; ... % "Wall"
    064 000 064; ... % "Tunnel"
    192 000 128; ... % "Archway"
    ]

    % "Pole"
```

```
[
192 192 128; ... % "Column_Pole"
000 000 064; ... % "TrafficCone"
]

% Road
[
128 064 128; ... % "Road"
128 000 192; ... % "LaneMkgsDriv"
192 000 064; ... % "LaneMkgsNonDriv"
]

% "Pavement"
[
000 000 192; ... % "Sidewalk"
064 192 128; ... % "ParkingBlock"
128 128 192; ... % "RoadShoulder"
]

% "Tree"
[
128 128 000; ... % "Tree"
192 192 000; ... % "VegetationMisc"
]

% "SignSymbol"
[
192 128 128; ... % "SignSymbol"
128 128 064; ... % "Misc_Text"
000 064 064; ... % "TrafficLight"
]

% "Fence"
[
064 064 128; ... % "Fence"
]

% "Car"
[
064 000 128; ... % "Car"
064 128 192; ... % "SUVPickupTruck"
192 128 192; ... % "Truck_Bus"
192 064 128; ... % "Train"
128 064 064; ... % "OtherMoving"
]

% "Pedestrian"
[
064 064 000; ... % "Pedestrian"
192 128 064; ... % "Child"
064 000 192; ... % "CartLuggagePram"
064 128 064; ... % "Animal"
]

% "Bicyclist"
[
000 128 192; ... % "Bicyclist"
192 000 192; ... % "MotorcycleScooter"
```

```
    ]

    };
end

function pixelLabelColorbar(cmap, classNames)
% Add a colorbar to the current axis. The colorbar is formatted
% to display the class names with the color.

colormap(gca,cmap)

% Add a colorbar to the current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(cmap,1);

% Center tick labels.
c.Ticks = 1/(numClasses*2):1/numClasses:1;

% Remove tick marks.
c.TickLength = 0;
end

function cmap = camvidColorMap
% Define the colormap used by the CamVid data set.

cmap = [
    128 128 128   % Sky
    128 0 0       % Building
    192 192 192   % Pole
    128 64 128    % Road
    60 40 222     % Pavement
    128 128 0     % Tree
    192 128 128   % SignSymbol
    64 64 128     % Fence
    64 0 128      % Car
    64 64 0       % Pedestrian
    0 128 192     % Bicyclist
    ];

% Normalize between [0 1].
cmap = cmap ./ 255;
end
```

See Also

`gradCAM` | `semanticseg` | `pixelLabelDatastore`

More About

- “Semantic Segmentation With Deep Learning” (Computer Vision Toolbox)
- “Investigate Spectrogram Classifications Using LIME” on page 5-49
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Deep Learning Visualization Methods” on page 5-186

Generate Untargeted and Targeted Adversarial Examples for Image Classification

This example shows how to use the fast gradient sign method (FGSM) and the basic iterative method (BIM) to generate adversarial examples for a pretrained neural network.

Neural networks can be susceptible to a phenomenon known as *adversarial examples* [1], where very small changes to an input can cause the input to be misclassified. These changes are often imperceptible to humans.

In this example, you create two types of adversarial examples:

- Untargeted — Modify an image so that it is misclassified as any incorrect class.
- Targeted — Modify an image so that it is misclassified as a specific class.

Load Network and Image

Load a network that has been trained on the ImageNet [2] data set and convert it to a `dlnetwork`.

```
net = squeezenet;
```

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph,lgraph.Layers(end).Name);
dlnet = dlnetwork(lgraph);
```

Extract the class labels.

```
classes = categories(net.Layers(end).Classes);
```

Load an image to use to generate an adversarial example. The image is a picture of a golden retriever.

```
img = imread('sherlock.jpg');
T = "golden retriever";
```

Resize the image to match the input size of the network.

```
inputSize = dlnet.Layers(1).InputSize;
img = imresize(img,inputSize(1:2));

figure
imshow(img)
title("Ground Truth: " + T)
```

Prepare the image by converting it to a `dlarray`.

```
X = dlarray(single(img),"SSCB");
```

Prepare the label by one-hot encoding it.

```
T = onehotencode(T,1,'ClassNames',classes);
T = dlarray(single(T),"CB");
```

Untargeted Fast Gradient Sign Method

Create an adversarial example using the untargeted FGSM [3]. This method calculates the gradient $\nabla_X L(X, T)$ of the loss function L, with respect to the image X you want to find an adversarial example for, and the class label T. This gradient describes the direction to "push" the image in to increase the chance it is misclassified. You can then add or subtract a small error from each pixel to increase the likelihood the image is misclassified.

The adversarial example is calculated as follows:

$X_{\text{adv}} = X + \epsilon \cdot \text{sign}(\nabla_X L(X, T))$.

Parameter ϵ controls the size of the push. A larger ϵ value increases the chance of generating a misclassified image, but makes the change in the image more visible. This method is untargeted, as the aim is to get the image misclassified, regardless of which class.

Calculate the gradient of the image with respect to the golden retriever class.

```
gradient = dlfeval(@untargetedGradients,dlnet,X,T);
```

Set `epsilon` to 1 and generate the adversarial example.

```
epsilon = 1;
XAdv = X + epsilon*sign(gradient);
```

Predict the class of the original image and the adversarial image.

```
YPred = predict(dlnet,X);
YPred = onehotdecode(squeeze(YPred),classes,1)
```

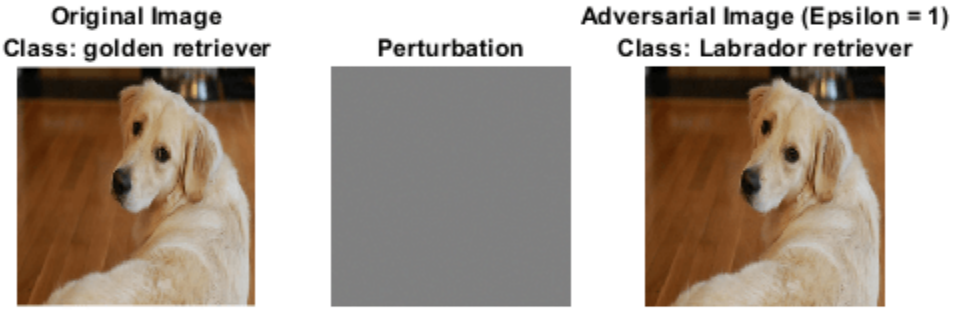
```
YPred = categorical
     golden retriever
```

```
YPredAdv = predict(dlnet,XAdv);
YPredAdv = onehotdecode(squeeze(YPredAdv),classes,1)
```

```
YPredAdv = categorical
     Labrador retriever
```

Display the original image, the perturbation added to the image, and the adversarial image. If the `epsilon` value is large enough, the adversarial image has a different class label from the original image.

```
showAdversarialImage(X,YPred,XAdv,YPredAdv,epsilon);
```



The network correctly classifies the unaltered image as a golden retriever. However, because of perturbation, the network misclassifies the adversarial image as a labrador retriever. Once added to the image, the perturbation is imperceptible, demonstrating how adversarial examples can exploit robustness issues within a network.

Targeted Adversarial Examples

A simple improvement to FGSM is to perform multiple iterations. This approach is known as the basic iterative method (BIM) [4] or projected gradient descent [5]. For the BIM, the size of the perturbation is controlled by parameter α representing the step size in each iteration. This is as the BIM usually takes many, smaller, FGSM steps in the direction of the gradient. After each iteration, clip the perturbation to ensure the magnitude does not exceed ϵ. This method can yield adversarial examples with less distortion than FGSM.

When you use untargeted FGSM, the predicted label of the adversarial example can be very similar to the label of the original image. For example, a dog might be misclassified as a different kind of dog. However, you can easily modify these methods to misclassify an image as a specific class. Instead of maximizing the cross-entropy loss, you can minimize the mean squared error between the output of the network and the desired target output.

Generate a targeted adversarial example using the BIM and the great white shark target class.

```
targetClass = "great white shark";
targetClass = onehotencode(targetClass,1,'ClassNames',classes);
```

Increase the `epsilon` value to 5, set the step size `alpha` to 0.2, and perform 25 iterations. Note that you may have to adjust these settings for other networks.

```
epsilon = 5;
alpha = 0.2;
numIterations = 25;
```

Keep track of the perturbation and clip any values that exceed `epsilon`.

```
delta = zeros(size(X),'like',X);
for i = 1:numIterations
    gradient = dlfeval(@targetedGradients,dlnet,X+delta,targetClass);

    delta = delta - alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdvTarget = X + delta;
```

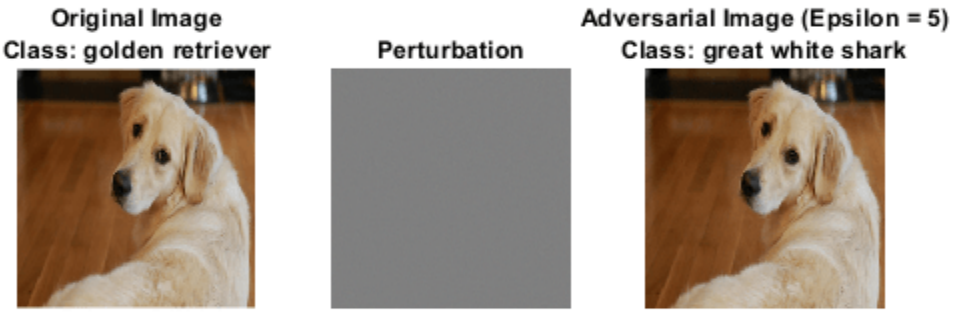
Predict the class of the targeted adversarial example.

```
YPredAdvTarget = predict(dlnet,XAdvTarget);
YPredAdvTarget = onehotdecode(squeeze(YPredAdvTarget),classes,1)
```

```
YPredAdvTarget = categorical
     great white shark
```

Display the original image, the perturbation added to the image, and the targeted adversarial image.

```
showAdversarialImage(X,YPred,XAdvTarget,YPredAdvTarget,epsilon);
```



Because of imperceptible perturbation, the network classifies the adversarial image as a great white shark.

To make the network more robust against adversarial examples, you can use adversarial training. For an example showing how to train a network robust to adversarial examples, see “Train Image Classification Network Robust to Adversarial Examples” on page 5-83.

Supporting Functions

Untargeted Input Gradient Function

Calculate the gradient used to create an untargeted adversarial example. This gradient is the gradient of the cross-entropy loss.

```
function gradient = untargetedGradients(dlnet,X,target)

Y = predict(dlnet,X);
Y = stripdims(squeeze(Y));
loss = crossentropy(Y,target,'DataFormat','CB');
gradient = dlgradient(loss,X);

end
```

Targeted Input Gradient Function

Calculate the gradient used to create a targeted adversarial example. This gradient is the gradient of the mean squared error.

```
function gradient = targetedGradients(dlnet,X,target)

Y = predict(dlnet,X);
Y = stripdims(squeeze(Y));
loss = mse(Y,target,'DataFormat','CB');
gradient = dlgradient(loss,X);

end
```

Show Adversarial Image

Show an image, the corresponding adversarial image, and the difference between the two (perturbation).

```
function showAdversarialImage(image,label,imageAdv,labelAdv,epsilon)

figure
subplot(1,3,1)
imgTrue = uint8(extractdata(image));
imshow(imgTrue)
title("Original Image" + newline + "Class: " + string(label))

subplot(1,3,2)
perturbation = uint8(extractdata(imageAdv-image+127.5));
imshow(perturbation)
title("Perturbation")

subplot(1,3,3)
advImg = uint8(extractdata(imageAdv));
imshow(advImg)
title("Adversarial Image (Epsilon = " + string(epsilon) + ")" + newline + ...
    "Class: " + string(labelAdv))
end
```

References

[1] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples.” Preprint, submitted March 20, 2015. https://arxiv.org/abs/1412.6572.

[2] *ImageNet*. http://www.image-net.org.

[3] Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. “Intriguing Properties of Neural Networks.” Preprint, submitted February 19, 2014. https://arxiv.org/abs/1312.6199.

[4] Kurakin, Alexey, Ian Goodfellow, and Samy Bengio. “Adversarial Examples in the Physical World.” Preprint, submitted February 10, 2017. https://arxiv.org/abs/1607.02533.

[5] Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. “Towards Deep Learning Models Resistant to Adversarial Attacks.” Preprint, submitted September 4, 2019. https://arxiv.org/abs/1706.06083.

See Also

`dlnetwork` | `onehotdecode` | `onehotencode` | `predict` | `dlfeval` | `dlgradient`

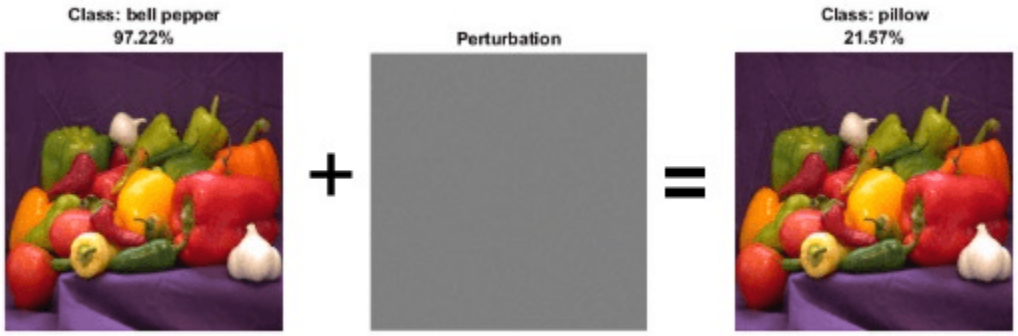
More About

- “Train Image Classification Network Robust to Adversarial Examples” on page 5-83
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Understand Network Predictions Using LIME” on page 5-42

Train Image Classification Network Robust to Adversarial Examples

This example shows how to train a neural network that is robust to adversarial examples using fast gradient sign method (FGSM) adversarial training.

Neural networks can be susceptible to a phenomenon known as *adversarial examples* [1], where very small changes to an input can cause it to be misclassified. These changes are often imperceptible to humans.



Techniques for creating adversarial examples include the FGSM [2] and the basic iterative method (BIM) [3], also known as projected gradient descent [4]. These techniques can significantly degrade the accuracy of a network.

You can use *adversarial training* [5] to train networks that are robust to adversarial examples. This example shows how to:

1. Train an image classification network.
2. Investigate network robustness by generating adversarial examples.
3. Train an image classification network that is robust to adversarial examples.

Load Training Data

The `digitTrain4DArrayData` function loads images of handwritten digits and their digit labels. Create an `arrayDatastore` object for the images and the labels, and then use the `combine` function to make a single datastore containing all the training data.

```
rng default
[XTrain,TTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsTTrain = arrayDatastore(TTrain);

dsTrain = combine(dsXTrain,dsTTrain);
```

Extract the class names.

```
classes = categories(TTrain);
```

Construct Network Architecture

Define an image classification network.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none','Name','input')
    convolution2dLayer(3,32,'Padding',1,'Name','conv1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,64,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    maxPooling2dLayer(2,'Stride',2,'Name','pool')
    fullyConnectedLayer(10,'Name','fc2')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph);
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes as input a `dlnetwork` object and a mini-batch of input data with corresponding labels and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Train Network

Train the network using a custom training loop.

Specify the training options. Train for 30 epochs with a mini-batch size of 100 and a learning rate of 0.01.

```
numEpochs = 30;
miniBatchSize = 100;
learnRate = 0.01;
executionEnvironment = "auto";
```

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to convert the labels to one-hot encoded variables.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain, ...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn',@preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

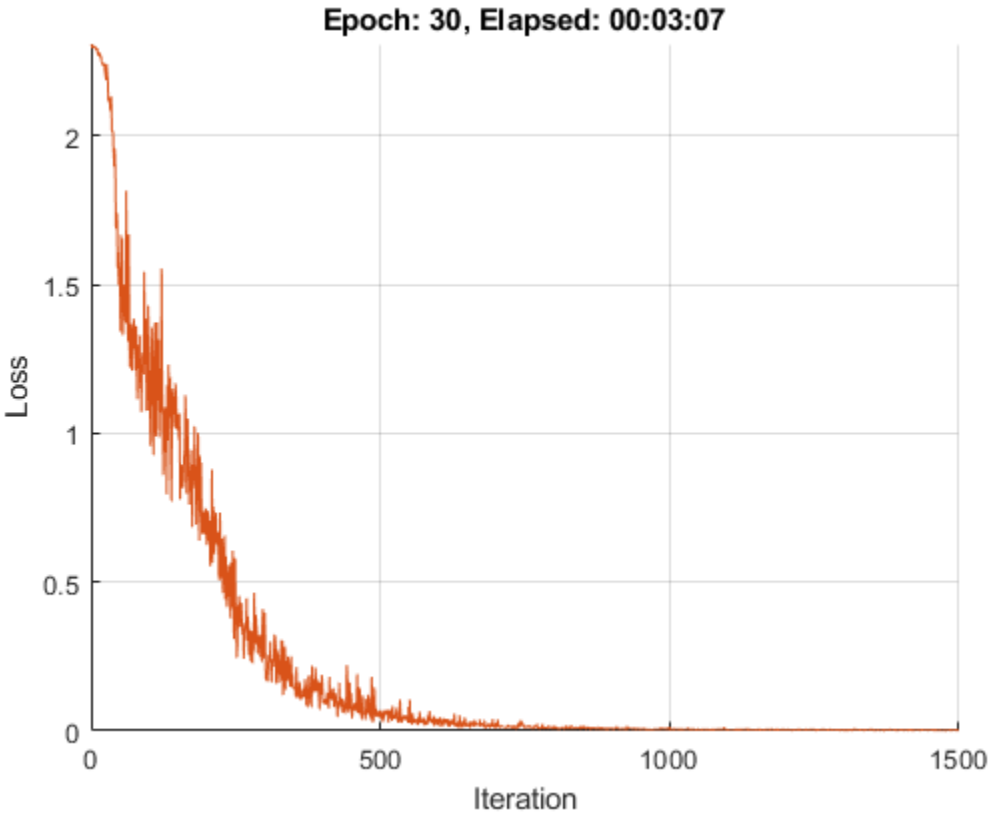
Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Train the network using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using the `dlfeval` and `modelGradients` functions and update the network state.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration +1;

        % Read mini-batch of data.
        [dlX,dlT] = next(mbq);

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
            dlT = gpuArray(dlT);
        end

        % Evaluate the model gradients, state, and loss.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlT);
        dlnet.State = state;

        % Update the network parameters using the SGDM optimizer.
        [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
    end
end
```



Test Network

Test the classification accuracy of the network by evaluating network predictions on a test data set.

Create a `minibatchqueue` object containing the test data.

```
[XTest,TTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsTTest = arrayDatastore(TTest);

dsTest = combine(dsXTest,dsTTest);

mbqTest = minibatchqueue(dsTest, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatch, ...
    'MiniBatchFormat','SSCB');
```

Predict the classes of the test data using the trained network and the `modelPredictions` function defined at the end of this example.

```
YPred = modelPredictions(dlnet,mbqTest,classes);
acc = mean(YPred == TTest)
```

```
acc = 0.9866
```

The network accuracy is very high.

Test Network with Adversarial Inputs

Apply adversarial perturbations to the input images and see how doing so affects the network accuracy.

You can generate adversarial examples using techniques such as FGSM and BIM. FGSM is a simple technique that takes a single step in the direction of the gradient $\nabla_X L(X, T)$ of the loss function L, with respect to the image X you want to find an adversarial example for, and the class label T. The adversarial example is calculated as

$X_{\text{adv}} = X + \epsilon \,.\, \text{sign}(\nabla_X L(X, T))$.

Parameter ϵ controls how different the adversarial examples look from the original images. In this example, the values of the pixels are between 0 and 1, so an ϵ value of 0.1 alters each individual pixel value by up to 10% of the range. The value of ϵ depends on the image scale. For example, if your image is instead between 0 and 255, you need to multiply this value by 255.

BIM is a simple improvement to FGSM which applies FGSM over multiple iterations and applies a threshold. After each iteration, the BIM clips the perturbation to ensure the magnitude does not exceed ϵ. This method can yield adversarial examples with less distortion than FGSM. For more information about generating adversarial examples, see “Generate Untargeted and Targeted Adversarial Examples for Image Classification” on page 5-76.

Create adversarial examples using the BIM. Set `epsilon` to 0.1.

```
epsilon = 0.1;
```

For the BIM, the size of the perturbation is controlled by parameter α representing the step size in each iteration. This is as the BIM usually takes many, smaller, FGSM steps in the direction of the gradient.

Define the step size `alpha` and the number of iterations.

```
alpha = 0.01;
numAdvIter = 20;
```

Use the `adversarialExamples` function (defined at the end of this example) to compute adversarial examples using the BIM on the test data set. This function also returns the new predictions for the adversarial images.

```
reset(mbqTest)
[XAdv,YPredAdv] = adversarialExamples(dlnet,mbqTest,epsilon,alpha,numAdvIter,classes);
```

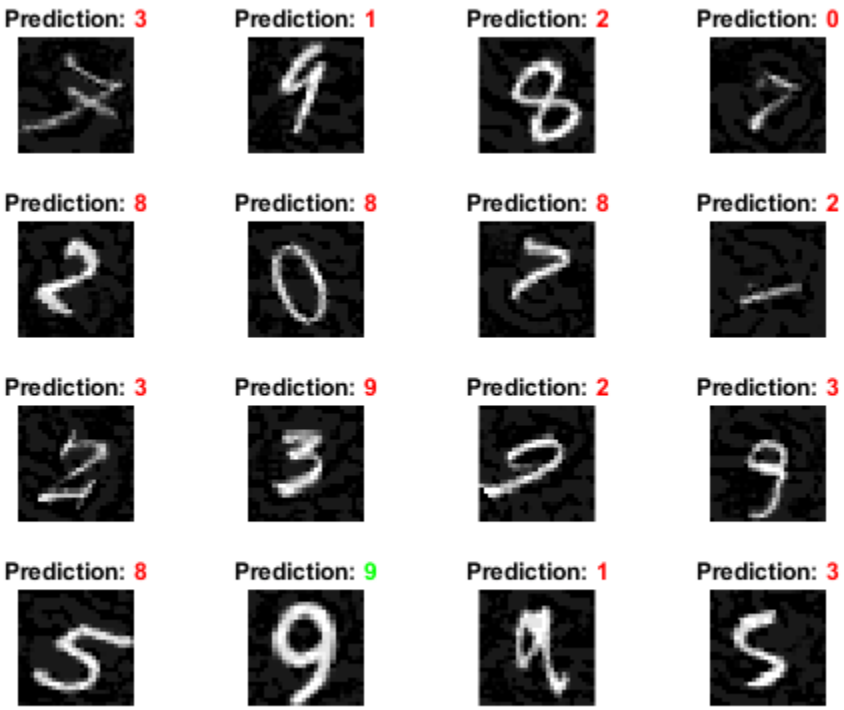
Compute the accuracy of the network on the adversarial example data.

```
accAdversarial = mean(YPredAdv == TTest)
```

```
accAdversarial = 0.0114
```

Plot the results.

```
visualizePredictions(XAdv,YPredAdv,TTest);
```



You can see that the accuracy is severely degraded by the BIM, even though the image perturbation is hardly visible.

Train Robust Network

You can train a network to be robust against adversarial examples. One popular method is adversarial training. Adversarial training involves applying adversarial perturbations to the training data during the training process [4] [5].

FGSM adversarial training is a fast and effective technique for training a network to be robust to adversarial examples. The FGSM is similar to the BIM, but it takes a single larger step in the direction of the gradient to generate an adversarial image.

Adversarial training involves applying the FGSM technique to each mini-batch of training data. However, for the training to be effective, these criteria must apply:

- The FGSM training method must use a randomly initialized perturbation instead of a perturbation that is initialized to zero.
- For the network to be robust to perturbations of size ϵ, perform FGSM training with a value slightly larger than ϵ. For this example, during adversarial training, you perturb the images using step size $\alpha = 1.25\epsilon$.

Train a new network with FGSM adversarial training. Start by using the same untrained network architecture as in the original network.

```
dlnetRobust = dlnetwork(lgraph);
```

Define the adversarial training parameters. Set the number of iterations to 1, as the FGSM is equivalent to the BIM with a single iteration. Randomly initialize the perturbation and perturb the images using `alpha`.

```
numIter = 1;
initialization = "random";
alpha = 1.25*epsilon;
```

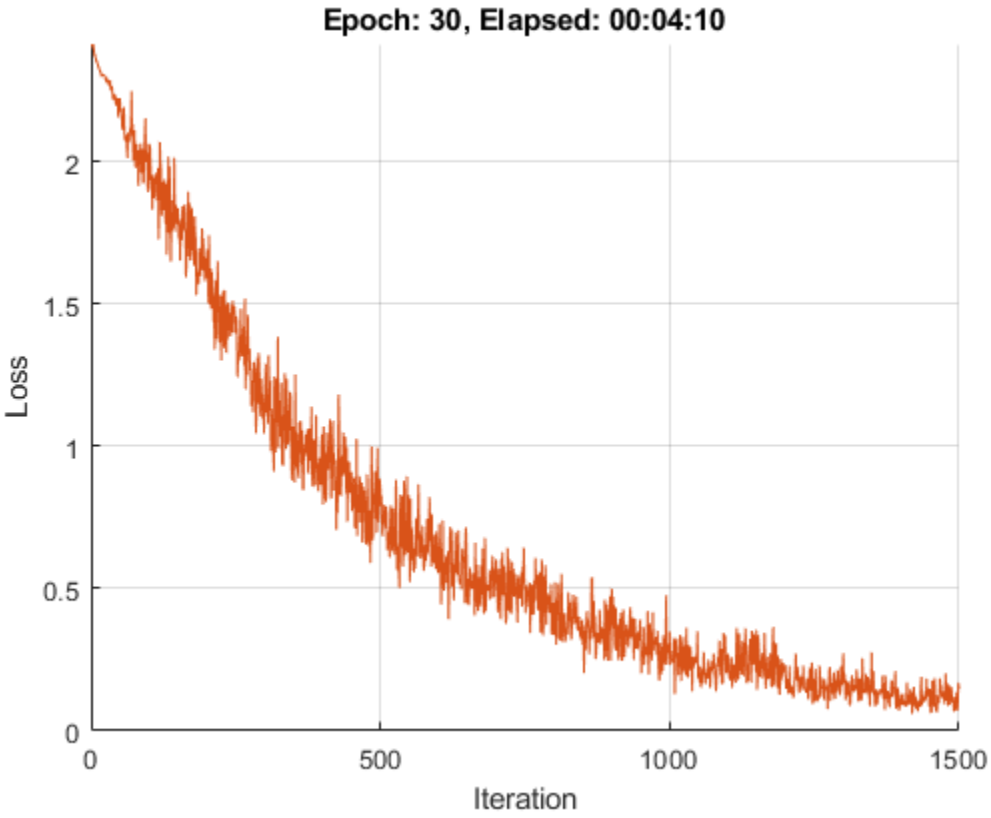
Initialize the training progress plot.

```
figure
lineLossRobustTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Train the robust network using a custom training loop and the same training options as previously defined. This loop is the same as in the previous custom training, but with added adversarial perturbation.

```
velocity = [];
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,dlT] = next(mbq);

        %  If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
            dlT = gpuArray(dlT);
        end

        % Apply adversarial perturbations to the data.
        dlX = basicIterativeMethod(dlnetRobust,dlX,dlT,alpha,epsilon, ...
            numIter,initialization);

        % Evaluate the model gradients, state, and loss.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnetRobust,dlX,dlT);
        dlnet.State = state;

        % Update the network parameters using the SGDM optimizer.
        [dlnetRobust,velocity] = sgdmupdate(dlnetRobust,gradients,velocity,learnRate);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossRobustTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
```

```
        drawnow
    end
end
```



Test Robust Network

Calculate the accuracy of the robust network on the digits test data. The accuracy of the robust network can be slightly lower than the nonrobust network on the standard data.

```
reset(mbqTest)
YPred = modelPredictions(dlnetRobust,mbqTest,classes);
accRobust = mean(YPred == TTest)
```

```
accRobust = 0.9972
```

Compute the adversarial accuracy.

```
reset(mbqTest)
[XAdv,YPredAdv] = adversarialExamples(dlnetRobust,mbqTest,epsilon,alpha,numAdvIter,classes);
accRobustAdv = mean(YPredAdv == TTest)
```

```
accRobustAdv = 0.7558
```

The adversarial accuracy of the robust network is much better than that of the original network.

Supporting Functions

Model Gradients Function

The `modelGradients` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels T and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,T)

[dlYPred,state] = forward(dlnet,dlX);

loss = crossentropy(dlYPred,T);
gradients = dlgradient(loss,dlnet.Learnables);

loss = double(gather(extractdata(loss)));

end
```

Input Gradients Function

The `modelGradientsInput` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels T and returns the gradients of the loss with respect to the input data `dlX`.

```
function gradient = modelGradientsInput(dlnet,dlX,T)

T = squeeze(T);
T = dlarray(T,'CB');

[dlYPred] = forward(dlnet,dlX);

loss = crossentropy(dlYPred,T);
gradient = dlgradient(loss,dlX);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a four-dimensional array.
2. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);
```

```
% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)

predictions = [];

while hasdata(mbq)

    dlXTest = next(mbq);
    dlYPred = predict(dlnet,dlXTest);

    YPred = onehotdecode(dlYPred,classes,1)';

    predictions = [predictions; YPred];
end

end
```

Adversarial Examples Function

Generate adversarial examples for a `minibatchqueue` object using the basic iterative method (BIM) and predict the class of the adversarial examples using the trained network `dlnet`.

```
function [XAdv,predictions] = adversarialExamples(dlnet,mbq,epsilon,alpha,numIter,classes)

XAdv = {};
predictions = [];
iteration = 0;

% Generate adversarial images for each mini-batch.
while hasdata(mbq)

    iteration = iteration +1;
    [dlX,dlT] = next(mbq);

    initialization = "zero";

    % Generate adversarial images.
    XAdvMBQ = basicIterativeMethod(dlnet,dlX,dlT,alpha,epsilon, ...
        numIter,initialization);

    % Predict the class of the adversarial images.
    dlYPred = predict(dlnet,XAdvMBQ);
    YPred = onehotdecode(dlYPred,classes,1)';
```

```
    XAdv{iteration} = XAdvMBQ;
    predictions = [predictions; YPred];
end

% Concatenate.
XAdv = cat(4,XAdv{:});

end
```

Basic Iterative Method Function

Generate adversarial examples using the basic iterative method (BIM). This method runs for multiple iterations with a threshold at the end of each iteration to ensure that the entries do not exceed `epsilon`. When `numIter` is set to 1, this is equivalent to using the fast gradient sign method (FGSM).

```
function XAdv = basicIterativeMethod(dlnet,dlX,dlT,alpha,epsilon,numIter,initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(dlX),'like',dlX);
else
    delta = epsilon*(2*rand(size(dlX),'like',dlX) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,dlnet,dlX+delta,dlT);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = dlX + delta;

end
```

Visualize Prediction Results Function

Visualize images along with their predicted classes. Correct predictions use green text. Incorrect predictions use red text.

```
function visualizePredictions(XTest,YPred,TTest)

figure
height = 4;
width = 4;
numImages = height*width;

% Select random images from the data.
indices = randperm(size(XTest,4),numImages);

XTest = extractdata(XTest);
XTest = XTest(:,:,:,indices);
YPred = YPred(indices);
TTest = TTest(indices);
```

```
% Plot images with the predicted label.
for i = 1:(numImages)
    subplot(height,width,i)
    imshow(XTest(:,:,:,i))

    % If the prediction is correct, use green. If the prediction is false,
    % use red.
    if YPred(i) == TTest(i)
        color = "\color{green}";
    else
        color = "\color{red}";
    end
    title("Prediction: " + color + string(YPred(i)))
end

end
```

References

[1] Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. “Intriguing Properties of Neural Networks.” Preprint, submitted February 19, 2014. https://arxiv.org/abs/1312.6199.

[2] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples.” Preprint, submitted March 20, 2015. https://arxiv.org/abs/1412.6572.

[3] Kurakin, Alexey, Ian Goodfellow, and Samy Bengio. “Adversarial Examples in the Physical World.” Preprint, submitted February 10, 2017. https://arxiv.org/abs/1607.02533.

[4] Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. “Towards Deep Learning Models Resistant to Adversarial Attacks.” Preprint, submitted September 4, 2019. https://arxiv.org/abs/1706.06083.

[5] Wong, Eric, Leslie Rice, and J. Zico Kolter. “Fast Is Better than Free: Revisiting Adversarial Training.” Preprint, submitted January 12, 2020. https://arxiv.org/abs/2001.03994.

See Also

`dlfeval` | `dlnetwork` | `dlgradient` | `arrayDatastore` | `minibatchqueue`

More About

- “Generate Untargeted and Targeted Adversarial Examples for Image Classification” on page 5-76
- “Define Deep Learning Network for Custom Training Loops” on page 18-209
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21

Resume Training from Checkpoint Network

This example shows how to save checkpoint networks while training a deep learning network and resume training from a previously saved network.

Load Sample Data

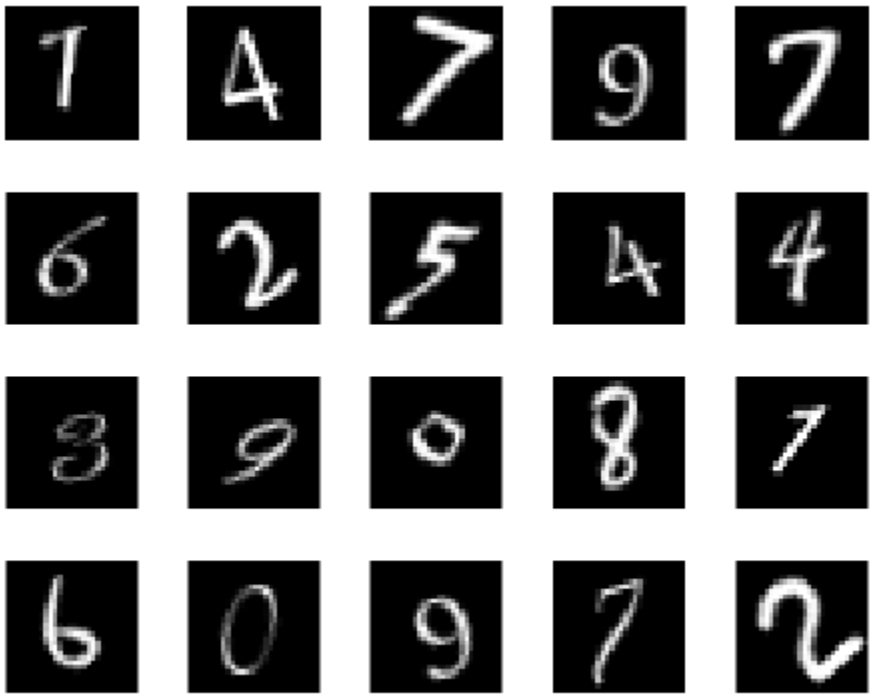
Load the sample data as a 4-D array. `digitTrain4DArrayData` loads the digit training set as 4-D array data. `XTrain` is a 28-by-28-by-1-by-5000 array, where 28 is the height and 28 is the width of the images. 1 is the number of channels and 5000 is the number of synthetic images of handwritten digits. `YTrain` is a categorical vector containing the labels for each observation.

```
[XTrain,YTrain] = digitTrain4DArrayData;
size(XTrain)
```

```
ans = 1×4

          28          28           1        5000
```

Display some of the images in `XTrain`.

```
figure;
perm = randperm(size(XTrain,4),20);
for i = 1:20
    subplot(4,5,i);
    imshow(XTrain(:,:,:,perm(i)));
end
```



Define Network Architecture

Define the neural network architecture.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(7)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

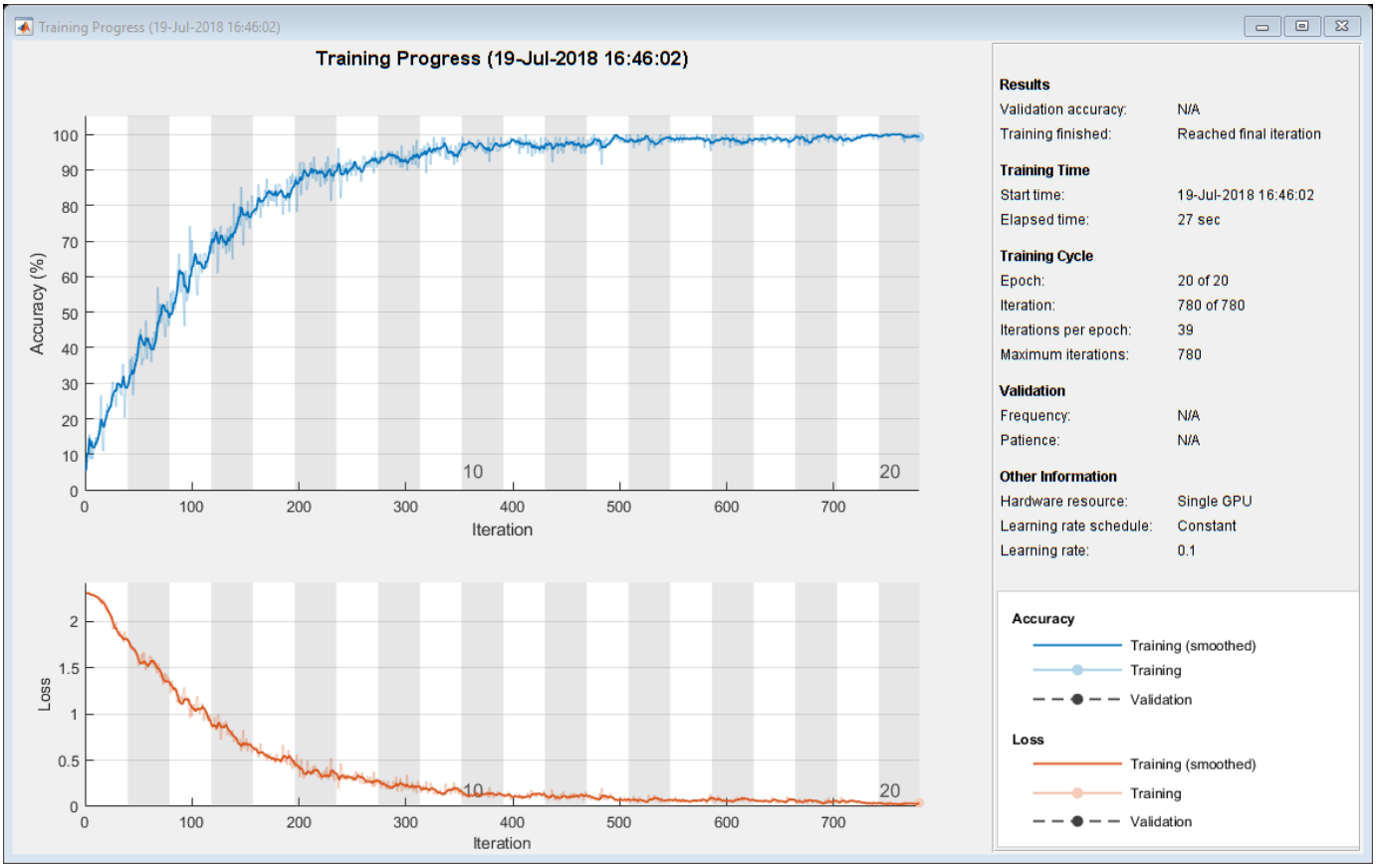
Specify Training Options and Train Network

Specify training options for stochastic gradient descent with momentum (SGDM) and specify the path for saving the checkpoint networks.

```
checkpointPath = pwd;
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.1, ...
    'MaxEpochs',20, ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch', ...
    'CheckpointPath',checkpointPath);
```

Train the network. `trainNetwork` uses a GPU if there is one available. If there is no available GPU, then it uses CPU. `trainNetwork` saves one checkpoint network each epoch and automatically assigns unique names to the checkpoint files.

```
net1 = trainNetwork(XTrain,YTrain,layers,options);
```



Load Checkpoint Network and Resume Training

Suppose that training was interrupted and did not complete. Rather than restarting the training from the beginning, you can load the last checkpoint network and resume training from that point. `trainNetwork` saves the checkpoint files with file names on the form `net_checkpoint__195__2018_07_13__11_59_10.mat`, where 195 is the iteration number, `2018_07_13` is the date, and `11_59_10` is the time `trainNetwork` saved the network. The checkpoint network has the variable name `net`.

Load the checkpoint network into the workspace.

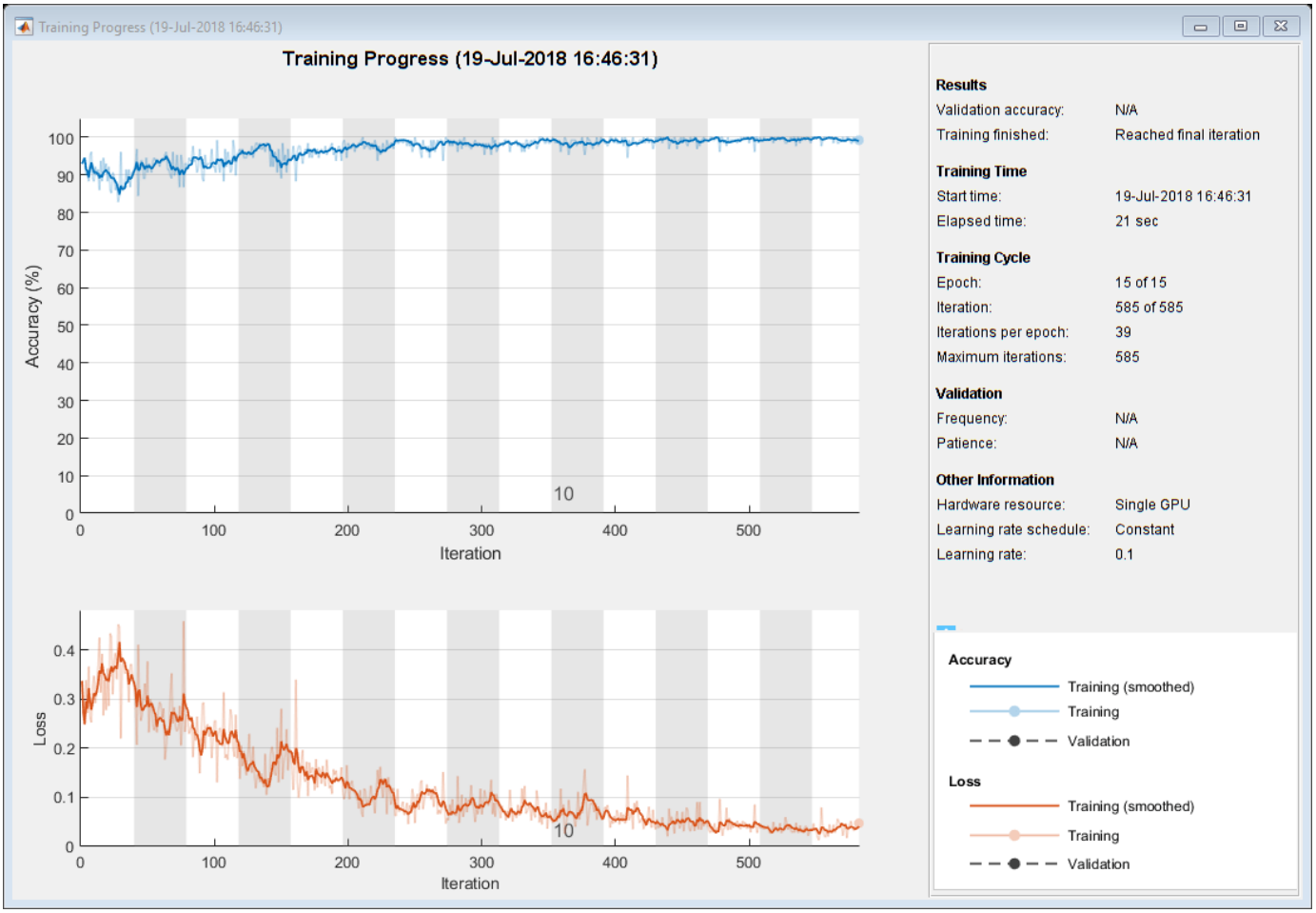
```
load('net_checkpoint__195__2018_07_13__11_59_10.mat','net')
```

Specify the training options and reduce the maximum number of epochs. You can also adjust other training options, such as the initial learning rate.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.1, ...
    'MaxEpochs',15, ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch', ...
    'CheckpointPath',checkpointPath);
```

Resume training using the layers of the checkpoint network you loaded with the new training options. If the checkpoint network is a DAG network, then use `layerGraph(net)` as the argument instead of `net.Layers`.

```
net2 = trainNetwork(XTrain,YTrain,net.Layers,options);
```



See Also

`trainingOptions` | `trainNetwork`

Related Examples

- “Create Simple Deep Learning Network for Classification” on page 3-47

More About

- “Learn About Convolutional Neural Networks” on page 1-17
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42

Deep Learning Using Bayesian Optimization

This example shows how to apply Bayesian optimization to deep learning and find optimal network hyperparameters and training options for convolutional neural networks.

To train a deep neural network, you must specify the neural network architecture, as well as options of the training algorithm. Selecting and tuning these hyperparameters can be difficult and take time. Bayesian optimization is an algorithm well suited to optimizing hyperparameters of classification and regression models. You can use Bayesian optimization to optimize functions that are nondifferentiable, discontinuous, and time-consuming to evaluate. The algorithm internally maintains a Gaussian process model of the objective function, and uses objective function evaluations to train this model.

This example shows how to:

- Download and prepare the CIFAR-10 data set for network training. This data set is one of the most widely used data sets for testing image classification models.
- Specify variables to optimize using Bayesian optimization. These variables are options of the training algorithm, as well as parameters of the network architecture itself.
- Define the objective function, which takes the values of the optimization variables as inputs, specifies the network architecture and training options, trains and validates the network, and saves the trained network to disk. The objective function is defined at the end of this script.
- Perform Bayesian optimization by minimizing the classification error on the validation set.
- Load the best network from disk and evaluate it on the test set.

As an alternative, you can use Bayesian optimization to find optimal training options in Experiment Manager. For more information, see “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26.

Prepare Data

Download the CIFAR-10 data set [1]. This data set contains 60,000 images, and each image has the size 32-by-32 and three color channels (RGB). The size of the whole data set is 175 MB. Depending on your internet connection, the download process can take some time.

```
datadir = tempdir;
downloadCIFARData(datadir);
```

Load the CIFAR-10 data set as training images and labels, and test images and labels. To enable network validation, use 5000 of the test images for validation.

```
[XTrain,YTrain,XTest,YTest] = loadCIFARData(datadir);

idx = randperm(numel(YTest),5000);
XValidation = XTest(:,:,:,idx);
XTest(:,:,:,idx) = [];
YValidation = YTest(idx);
YTest(idx) = [];
```

You can display a sample of the training images using the following code.

```
figure;
idx = randperm(numel(YTrain),20);
for i = 1:numel(idx)
```

```
    subplot(4,5,i);
    imshow(XTrain(:,:,:,idx(i)));
end
```

Choose Variables to Optimize

Choose which variables to optimize using Bayesian optimization, and specify the ranges to search in. Also, specify whether the variables are integers and whether to search the interval in logarithmic space. Optimize the following variables:

- Network section depth. This parameter controls the depth of the network. The network has three sections, each with `SectionDepth` identical convolutional layers. So the total number of convolutional layers is `3*SectionDepth`. The objective function later in the script takes the number of convolutional filters in each layer proportional to `1/sqrt(SectionDepth)`. As a result, the number of parameters and the required amount of computation for each iteration are roughly the same for different section depths.
- Initial learning rate. The best learning rate can depend on your data as well as the network you are training.
- Stochastic gradient descent momentum. Momentum adds inertia to the parameter updates by having the current update contain a contribution proportional to the update in the previous iteration. This results in more smooth parameter updates and a reduction of the noise inherent to stochastic gradient descent.
- L2 regularization strength. Use regularization to prevent overfitting. Search the space of regularization strength to find a good value. Data augmentation and batch normalization also help regularize the network.

```
optimVars = [
    optimizableVariable('SectionDepth',[1 3],'Type','integer')
    optimizableVariable('InitialLearnRate',[1e-2 1],'Transform','log')
    optimizableVariable('Momentum',[0.8 0.98])
    optimizableVariable('L2Regularization',[1e-10 1e-2],'Transform','log')];
```

Perform Bayesian Optimization

Create the objective function for the Bayesian optimizer, using the training and validation data as inputs. The objective function trains a convolutional neural network and returns the classification error on the validation set. This function is defined at the end of this script. Because `bayesopt` uses the error rate on the validation set to choose the best model, it is possible that the final network overfits on the validation set. The final chosen model is then tested on the independent test set to estimate the generalization error.

```
ObjFcn = makeObjFcn(XTrain,YTrain,XValidation,YValidation);
```

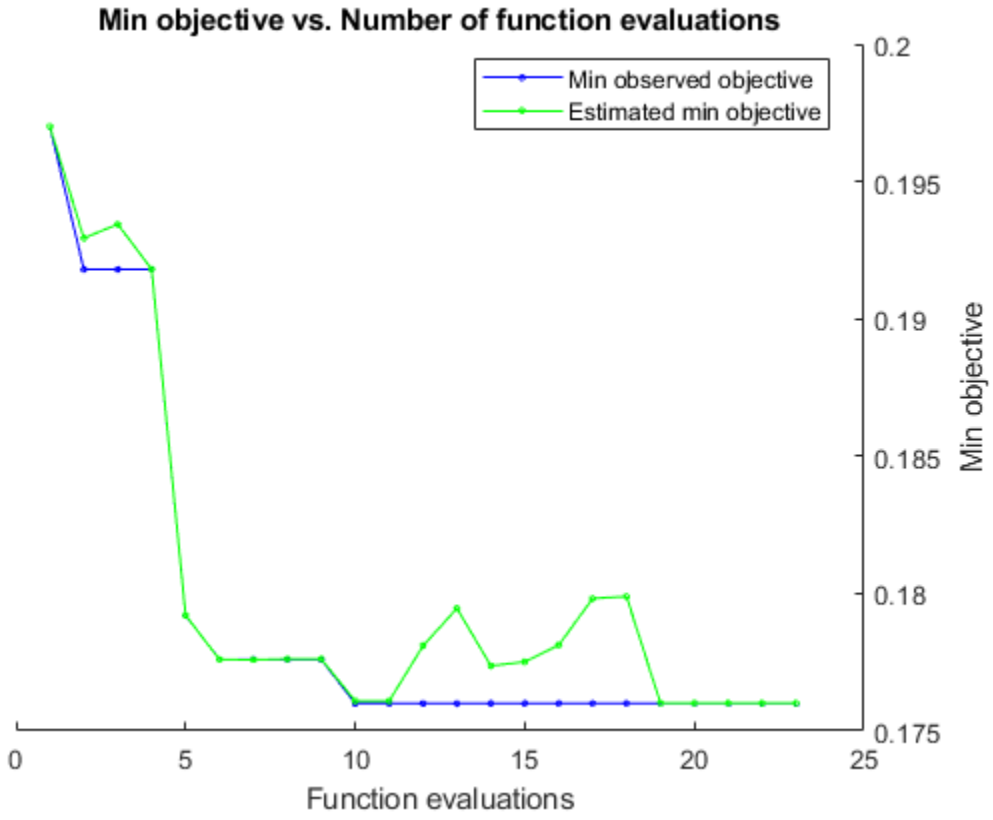
Perform Bayesian optimization by minimizing the classification error on the validation set. Specify the total optimization time in seconds. To best utilize the power of Bayesian optimization, you should perform at least 30 objective function evaluations. To train networks in parallel on multiple GPUs, set the `'UseParallel'` value to `true`. If you have a single GPU and set the `'UseParallel'` value to `true`, then all workers share that GPU, and you obtain no training speed-up and increase the chances of the GPU running out of memory.

After each network finishes training, `bayesopt` prints the results to the command window. The `bayesopt` function then returns the file names in `BayesObject.UserDataTrace`. The objective function saves the trained networks to disk and returns the file names to `bayesopt`.

```
BayesObject = bayesopt(ObjFcn,optimVars, ...
    'MaxTime',14*60*60, ...
    'IsObjectiveDeterministic',false, ...
    'UseParallel',false);
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	SectionDepth	InitialRate
1	Best	0.197	955.69	0.197	0.197	3	0.0
2	Best	0.1918	790.38	0.1918	0.19293	2	0.0
3	Accept	0.2438	660.29	0.1918	0.19344	1	0.05
4	Accept	0.208	672.81	0.1918	0.1918	1	0.1
5	Best	0.1792	844.07	0.1792	0.17921	2	0.0
6	Best	0.1776	851.49	0.1776	0.17759	2	0.2
7	Accept	0.2232	883.5	0.1776	0.17759	2	0.0
8	Accept	0.2508	822.65	0.1776	0.17762	1	0.02
9	Accept	0.1974	1947.6	0.1776	0.17761	3	0.0
10	Best	0.176	1938.4	0.176	0.17608	2	0
11	Accept	0.1914	2874.4	0.176	0.17608	3	0.0
12	Accept	0.181	2578	0.176	0.17809	2	0.3
13	Accept	0.1838	2410.8	0.176	0.17946	2	0.3
14	Accept	0.1786	2490.6	0.176	0.17737	2	0.4
15	Accept	0.1776	2668	0.176	0.17751	2	0.9
16	Accept	0.1824	3059.8	0.176	0.17812	2	0.4
17	Accept	0.1894	3091.5	0.176	0.17982	2	0.9
18	Accept	0.217	2794.5	0.176	0.17989	1	0
19	Accept	0.2358	4054.2	0.176	0.17601	3	0.2
20	Accept	0.2216	4411.7	0.176	0.17601	3	0.0

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	SectionDepth	InitialRate
21	Accept	0.2038	3906.4	0.176	0.17601	2	0.0
22	Accept	0.2492	4103.4	0.176	0.17601	2	0.5
23	Accept	0.1814	4240.5	0.176	0.17601	2	0.2




```
_____________________________________________________________
Optimization completed.
MaxTime of 50400 seconds reached.
Total function evaluations: 23
Total elapsed time: 53088.5123 seconds
Total objective function evaluation time: 53050.7026

Best observed feasible point:
    SectionDepth    InitialLearnRate    Momentum    L2Regularization
    ____________    ________________    ________    ________________

         2              0.3526         0.82381        1.4244e-07

Observed objective function value = 0.176
Estimated objective function value = 0.17601
Function evaluation time = 1938.4483

Best estimated feasible point (according to models):
    SectionDepth    InitialLearnRate    Momentum    L2Regularization
    ____________    ________________    ________    ________________

         2              0.3526         0.82381        1.4244e-07

Estimated objective function value = 0.17601
Estimated function evaluation time = 1898.2641
```

Evaluate Final Network

Load the best network found in the optimization and its validation accuracy.

```
bestIdx = BayesObject.IndexOfMinimumTrace(end);
fileName = BayesObject.UserDataTrace{bestIdx};
savedStruct = load(fileName);
valError = savedStruct.valError
```

```
valError = 0.1760
```

Predict the labels of the test set and calculate the test error. Treat the classification of each image in the test set as independent events with a certain probability of success, which means that the number of incorrectly classified images follows a binomial distribution. Use this to calculate the standard error (`testErrorSE`) and an approximate 95% confidence interval (`testError95CI`) of the generalization error rate. This method is often called the *Wald method*. `bayesopt` determines the best network using the validation set without exposing the network to the test set. It is then possible that the test error is higher than the validation error.

```
[YPredicted,probs] = classify(savedStruct.trainedNet,XTest);
testError = 1 - mean(YPredicted == YTest)
```

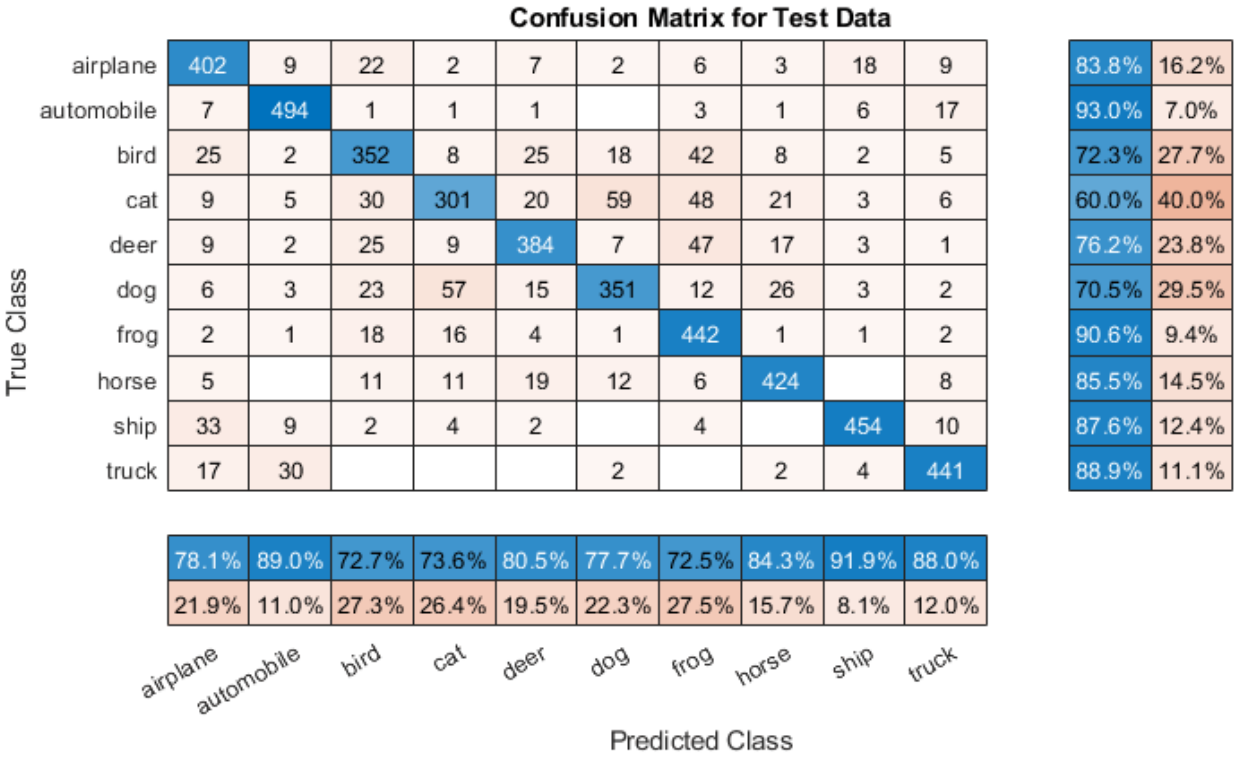
```
testError = 0.1910
```

```
NTest = numel(YTest);
testErrorSE = sqrt(testError*(1-testError)/NTest);
testError95CI = [testError - 1.96*testErrorSE, testError + 1.96*testErrorSE]
```

```
testError95CI = 1×2

    0.1801    0.2019
```

Plot the confusion matrix for the test data. Display the precision and recall for each class by using column and row summaries.

```
figure('Units','normalized','Position',[0.2 0.2 0.4 0.4]);
cm = confusionchart(YTest,YPredicted);
cm.Title = 'Confusion Matrix for Test Data';
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



You can display some test images together with their predicted classes and the probabilities of those classes using the following code.

```
figure
idx = randperm(numel(YTest),9);
for i = 1:numel(idx)
    subplot(3,3,i)
    imshow(XTest(:,:,:,idx(i)));
    prob = num2str(100*max(probs(idx(i),:)),3);
    predClass = char(YPredicted(idx(i)));
    label = [predClass,', ',prob,'%'];
    title(label)
end
```

Objective Function for Optimization

Define the objective function for optimization. This function performs the following steps:

1. Takes the values of the optimization variables as inputs. `bayesopt` calls the objective function with the current values of the optimization variables in a table with each column name equal to the variable name. For example, the current value of the network section depth is `optVars.SectionDepth`.
2. Defines the network architecture and training options.
3. Trains and validates the network.
4. Saves the trained network, the validation error, and the training options to disk.
5. Returns the validation error and the file name of the saved network.

```
function ObjFcn = makeObjFcn(XTrain,YTrain,XValidation,YValidation)
ObjFcn = @valErrorFun;
    function [valError,cons,fileName] = valErrorFun(optVars)
```

Define the convolutional neural network architecture.

- Add padding to the convolutional layers so that the spatial output size is always the same as the input size.
- Each time you down-sample the spatial dimensions by a factor of two using max pooling layers, increase the number of filters by a factor of two. Doing so ensures that the amount of computation required in each convolutional layer is roughly the same.
- Choose the number of filters proportional to `1/sqrt(SectionDepth)`, so that networks of different depths have roughly the same number of parameters and require about the same amount of computation per iteration. To increase the number of network parameters and the overall network flexibility, increase `numF`. To train even deeper networks, change the range of the `SectionDepth` variable.
- Use `convBlock(filterSize,numFilters,numConvLayers)` to create a block of `numConvLayers` convolutional layers, each with a specified `filterSize` and `numFilters` filters, and each followed by a batch normalization layer and a ReLU layer. The `convBlock` function is defined at the end of this example.

```
    imageSize = [32 32 3];
    numClasses = numel(unique(YTrain));
    numF = round(16/sqrt(optVars.SectionDepth));
    layers = [
        imageInputLayer(imageSize)

        % The spatial input and output sizes of these convolutional
        % layers are 32-by-32, and the following max pooling layer
        % reduces this to 16-by-16.
        convBlock(3,numF,optVars.SectionDepth)
        maxPooling2dLayer(3,'Stride',2,'Padding','same')

        % The spatial input and output sizes of these convolutional
        % layers are 16-by-16, and the following max pooling layer
        % reduces this to 8-by-8.
        convBlock(3,2*numF,optVars.SectionDepth)
        maxPooling2dLayer(3,'Stride',2,'Padding','same')

        % The spatial input and output sizes of these convolutional
        % layers are 8-by-8. The global average pooling layer averages
        % over the 8-by-8 inputs, giving an output of size
        % 1-by-1-by-4*initialNumFilters. With a global average
        % pooling layer, the final classification output is only
        % sensitive to the total amount of each feature present in the
        % input image, but insensitive to the spatial positions of the
        % features.
        convBlock(3,4*numF,optVars.SectionDepth)
        averagePooling2dLayer(8)

        % Add the fully connected layer and the final softmax and
        % classification layers.
        fullyConnectedLayer(numClasses)
        softmaxLayer
        classificationLayer];
```

Specify options for network training. Optimize the initial learning rate, SGD momentum, and L2 regularization strength.

Specify validation data and choose the `'ValidationFrequency'` value such that `trainNetwork` validates the network once per epoch. Train for a fixed number of epochs and lower the learning rate by a factor of 10 during the last epochs. This reduces the noise of the parameter updates and lets the network parameters settle down closer to a minimum of the loss function.

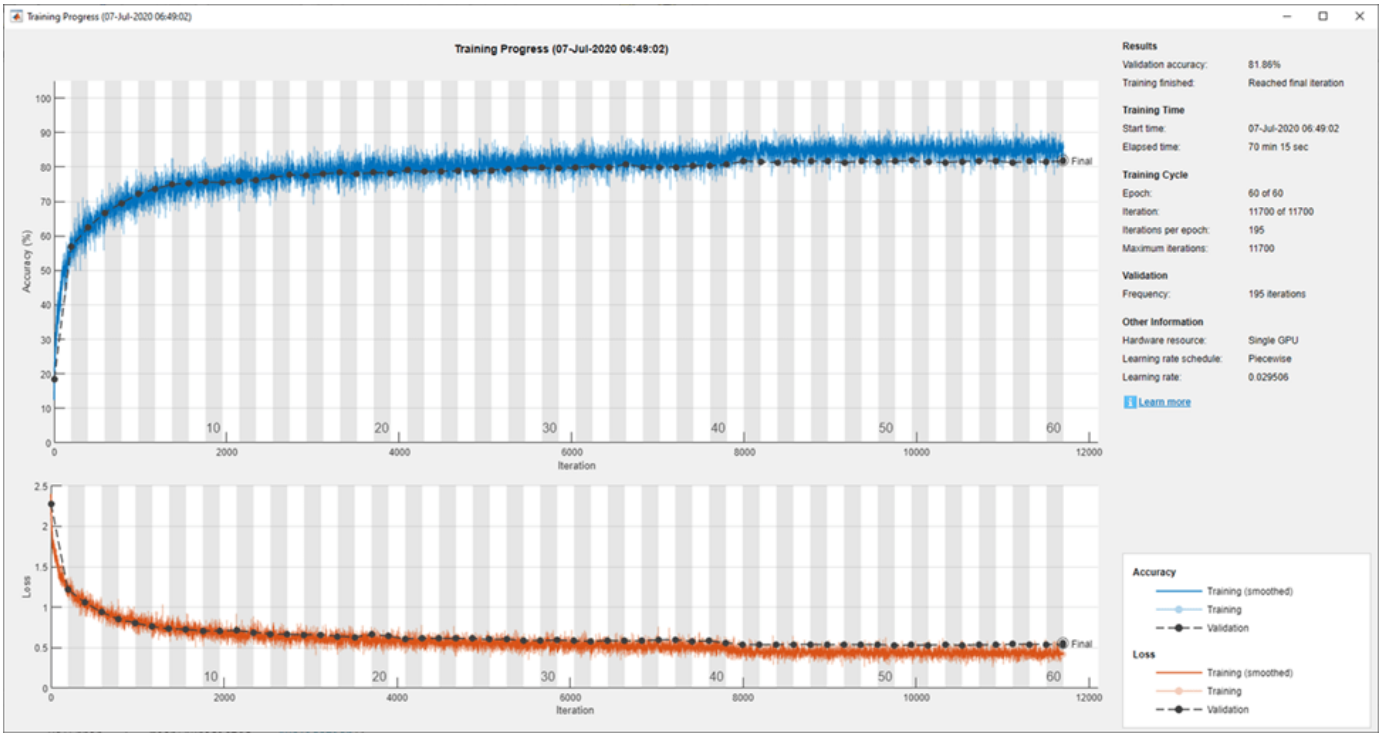
```
miniBatchSize = 256;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'InitialLearnRate',optVars.InitialLearnRate, ...
    'Momentum',optVars.Momentum, ...
    'MaxEpochs',60, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',40, ...
    'LearnRateDropFactor',0.1, ...
    'MiniBatchSize',miniBatchSize, ...
    'L2Regularization',optVars.L2Regularization, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',validationFrequency);
```

Use data augmentation to randomly flip the training images along the vertical axis, and randomly translate them up to four pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
datasource = augmentedImageDatastore(imageSize,XTrain,YTrain,'DataAugmentation',imageAugr
```

Train the network and plot the training progress during training. Close all training plots after training finishes.

```
trainedNet = trainNetwork(datasource,layers,options);
close(findall(groot,'Tag','NNET_CNN_TRAININGPLOT_UIFIGURE'))
```



Evaluate the trained network on the validation set, calculate the predicted image labels, and calculate the error rate on the validation data.

```
        YPredicted = classify(trainedNet,XValidation);
        valError = 1 - mean(YPredicted == YValidation);
```

Create a file name containing the validation error, and save the network, validation error, and training options to disk. The objective function returns `fileName` as an output argument, and `bayesopt` returns all the file names in `BayesObject.UserDataTrace`. The additional required output argument `cons` specifies constraints among the variables. There are no variable constraints.

```
        fileName = num2str(valError) + ".mat";
        save(fileName,'trainedNet','valError','options')
        cons = [];

    end
end
```

The `convBlock` function creates a block of `numConvLayers` convolutional layers, each with a specified `filterSize` and `numFilters` filters, and each followed by a batch normalization layer and a ReLU layer.

```
function layers = convBlock(filterSize,numFilters,numConvLayers)
layers = [
    convolution2dLayer(filterSize,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer];
```

```
layers = repmat(layers,numConvLayers,1);
end
```

References

[1] Krizhevsky, Alex. "Learning multiple layers of features from tiny images." (2009). https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

See Also

Experiment Manager | `trainNetwork` | `trainingOptions` | `bayesopt`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-17
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2
- “Compare Layer Weight Initializers” on page 18-181
- “Specify Custom Weight Initialization Function” on page 18-175
- “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26

Train Deep Learning Networks in Parallel

This example shows how to run multiple deep learning experiments on your local machine. Using this example as a template, you can modify the network layers and training options to suit your specific application needs. You can use this approach with a single or multiple GPUs. If you have a single GPU, the networks train one after the other in the background. The approach in this example enables you to continue using MATLAB® while deep learning experiments are in progress.

As an alternative, you can use Experiment Manager to interactively train multiple deep networks in parallel. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16.

Prepare Data Set

Before you can run the example, you must have access to a local copy of a deep learning data set. This example uses a data set with synthetic images of digits from 0 to 9. In the following code, change the location to point to your data set.

```
datasetLocation = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
```

If you want to run the experiments with more resources, you can run this example in a cluster in the cloud.

- Upload the data set to an Amazon S3 bucket. For an example, see “Upload Deep Learning Data to the Cloud” on page 7-53.
- Create a cloud cluster. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. For more information, see “Create Cloud Cluster” (Parallel Computing Toolbox).
- Select your cloud cluster as the default, on the **Home** tab, in the **Environment** section, select **Parallel > Select a Default Cluster**.

Load Data Set

Load the data set by using an `imageDatastore` object. Split the data set into training, validation, and test sets.

```
imds = imageDatastore(datasetLocation, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

[imdsTrain,imdsValidation,imdsTest] = splitEachLabel(imds,0.8,0.1);
```

To train the network with augmented image data, create an `augmentedImageDatastore`. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [28 28 1];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

Train Networks in Parallel

Start a parallel pool with as many workers as GPUs. You can check the number of available GPUs by using the `gpuDeviceCount` (Parallel Computing Toolbox) function. MATLAB assigns a different GPU to each worker. By default, `parpool` uses your default cluster profile. If you have not changed the default, it is `local`. This example was run using a machine with 2 GPUs.

```
numGPUs = gpuDeviceCount("available");
parpool(numGPUs);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

To send training progress information from the workers during training, use a `parallel.pool.DataQueue` (Parallel Computing Toolbox) object. To learn more about how to use data queues to obtain feedback during training, see the example “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32.

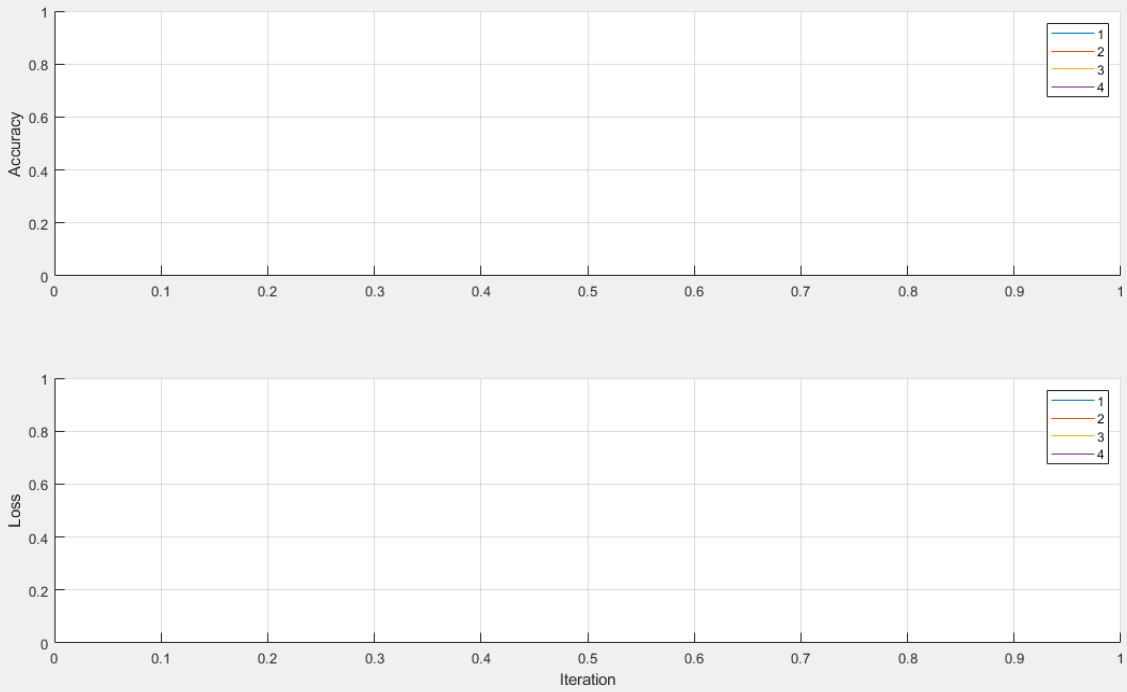
```
dataqueue = parallel.pool.DataQueue;
```

Define the network layers and training options. For code readability, you can define them in a separate function that returns several network architectures and training options. In this case, `networkLayersAndOptions` returns a cell array of network layers and an array of training options of the same length. Open this example in MATLAB and then click `networkLayersAndOptions` to open the supporting function `networkLayersAndOptions`. Paste in your own network layers and options. The file contains sample training options that show how to send information to the data queue using an output function.

```
[layersCell,options] = networkLayersAndOptions(augmentedImdsTrain,imdsValidation,dataqueue);
```

Prepare the training progress plots, and set a callback function to update these plots after each worker sends data to the queue. `preparePlots` and `updatePlots` are supporting functions for this example.

```
handles = preparePlots(numel(layersCell));
```




```
afterEach(dataqueue,@(data) updatePlots(handles,data));
```

To hold the computation results in parallel workers, use future objects. Preallocate an array of future objects for the result of each training.

```
trainingFuture(1:numel(layersCell)) = parallel.FevalFuture;
```

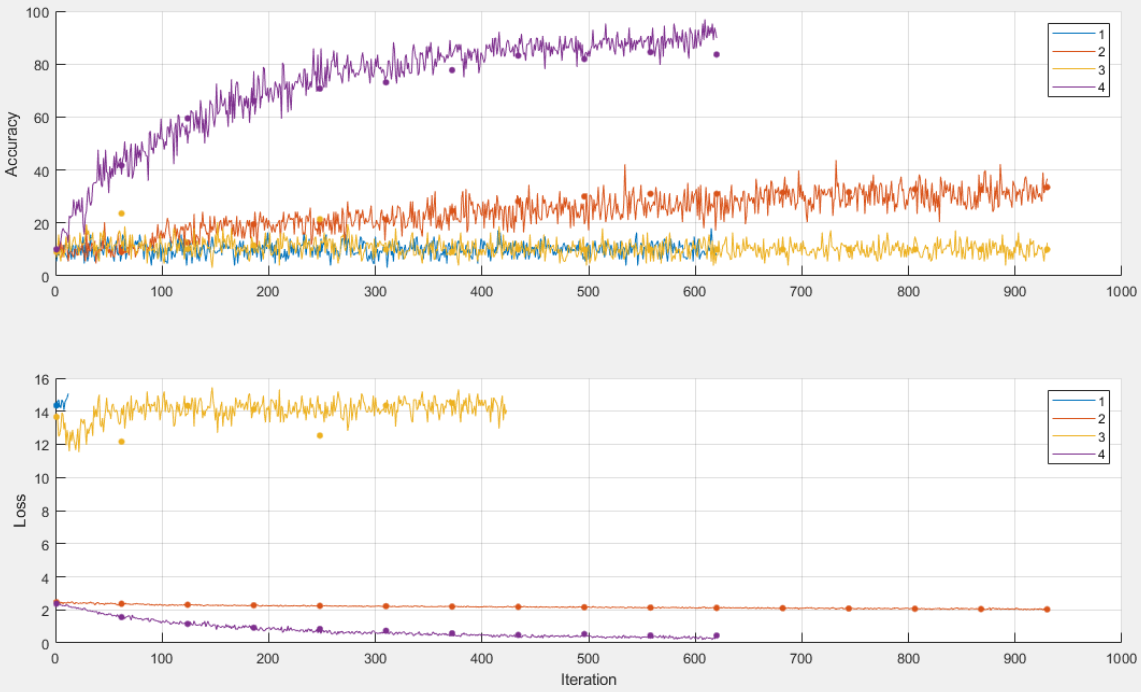
Loop through the network layers and options by using a `for` loop, and use `parfeval` (Parallel Computing Toolbox) to train the networks on a parallel worker. To request two output arguments from `trainNetwork`, specify 2 as the second input argument to `parfeval`.

```
for i=1:numel(layersCell)
    trainingFuture(i) = parfeval(@trainNetwork,2,augmentedImdsTrain,layersCell{i},options(i));
end
```

`parfeval` does not block MATLAB, so you can continue working while the computations take place.

To fetch results from future objects, use the `fetchOutputs` function. For this example, fetch the trained networks and their training information. `fetchOutputs` blocks MATLAB until the results are available. This step can take a few minutes.

```
[network,trainingInfo] = fetchOutputs(trainingFuture);
```



Save the results to disk using the `save` function. To load the results again later, use the `load` function. Use `sprintf` and `datetime` to name the file using the current date and time.

```
filename = sprintf('experiment-%s',datetime('now','Format','yyyyMMdd''T''HHmmss'));
save(filename,'network','trainingInfo');
```

Plot Results

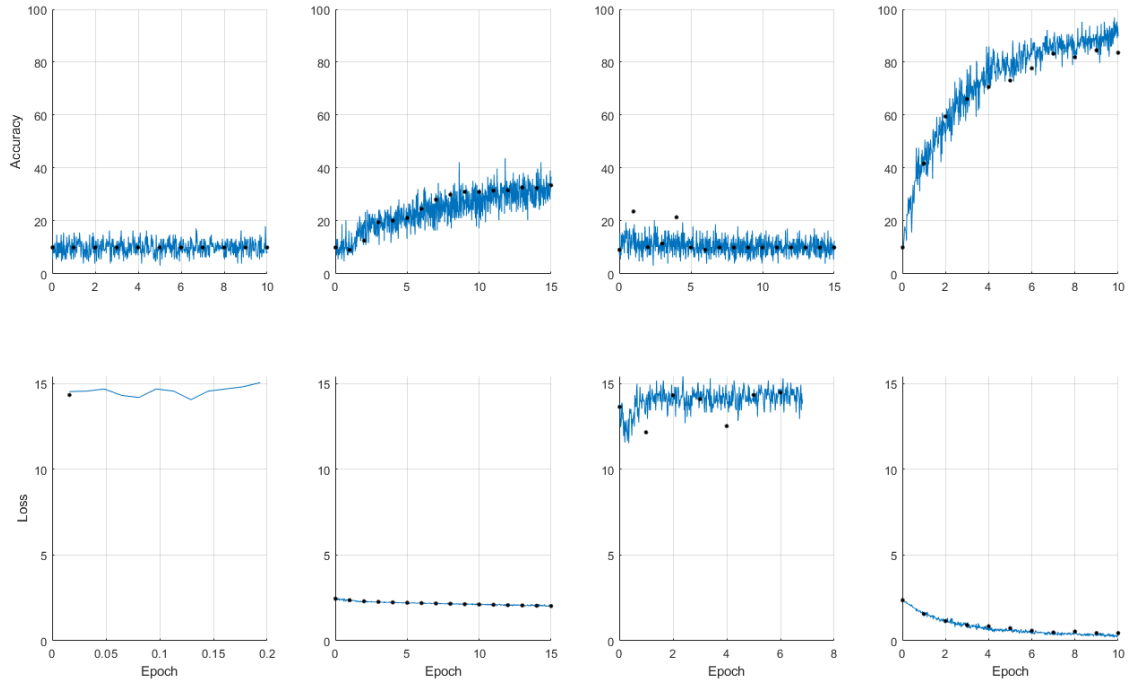
After the networks complete training, plot their training progress by using the information in `trainingInfo`.

Use subplots to distribute the different plots for each network. For this example, use the first row of subplots to plot the training accuracy against the number of epoch along with the validation accuracy.

```
figure('Units','normalized','Position',[0.1 0.1 0.6 0.6]);
title('Training Progress Plots');

for i=1:numel(layersCell)
    subplot(2,numel(layersCell),i);
    hold on; grid on;
    ylim([0 100]);
    iterationsPerEpoch = floor(augmentedImdsTrain.NumObservations/options(i).MiniBatchSize);
    epoch = (1:numel(trainingInfo(i).TrainingAccuracy))/iterationsPerEpoch;
    plot(epoch,trainingInfo(i).TrainingAccuracy);
    plot(epoch,trainingInfo(i).ValidationAccuracy,'.k','MarkerSize',10);
end
subplot(2,numel(layersCell),1), ylabel('Accuracy');
```

Then, use the second row of subplots to plot the training loss against the number of epoch along with the validation loss.

```
for i=1:numel(layersCell)
    subplot(2,numel(layersCell),numel(layersCell) + i);
    hold on; grid on;
    ylim([0 max([trainingInfo.TrainingLoss])]);
    iterationsPerEpoch = floor(augmentedImdsTrain.NumObservations/options(i).MiniBatchSize);
    epoch = (1:numel(trainingInfo(i).TrainingAccuracy))/iterationsPerEpoch;
    plot(epoch,trainingInfo(i).TrainingLoss);
    plot(epoch,trainingInfo(i).ValidationLoss,'.k','MarkerSize',10);
    xlabel('Epoch');
end
subplot(2,numel(layersCell),numel(layersCell)+1), ylabel('Loss');
```



After you choose a network, you can use `classify` and obtain its accuracy on the test data `imdsTest`.

See Also

Experiment Manager | `augmentedImageDatastore` | `imageDatastore` | `parfeval` | `fetchOutputs` | `trainNetwork` | `trainingOptions`

Related Examples

- “Train Network Using Automatic Multi-GPU Support” on page 7-42
- “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16

More About

- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2

Monitor Deep Learning Training Progress

When you train networks for deep learning, it is often useful to monitor the training progress. By plotting various metrics during training, you can learn how the training is progressing. For example, you can determine if and how quickly the network accuracy is improving, and whether the network is starting to overfit the training data.

When you specify `'training-progress'` as the `'Plots'` value in `trainingOptions` and start network training, `trainNetwork` creates a figure and displays training metrics at every iteration. Each iteration is an estimation of the gradient and an update of the network parameters. If you specify validation data in `trainingOptions`, then the figure shows validation metrics each time `trainNetwork` validates the network. The figure plots the following:

- **Training accuracy** — Classification accuracy on each individual mini-batch.
- **Smoothed training accuracy** — Smoothed training accuracy, obtained by applying a smoothing algorithm to the training accuracy. It is less noisy than the unsmoothed accuracy, making it easier to spot trends.
- **Validation accuracy** — Classification accuracy on the entire validation set (specified using `trainingOptions`).
- **Training loss**, **smoothed training loss**, and **validation loss** — The loss on each mini-batch, its smoothed version, and the loss on the validation set, respectively. If the final layer of your network is a `classificationLayer`, then the loss function is the cross entropy loss. For more information about loss functions for classification and regression problems, see “Output Layers” on page 1-39.

For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

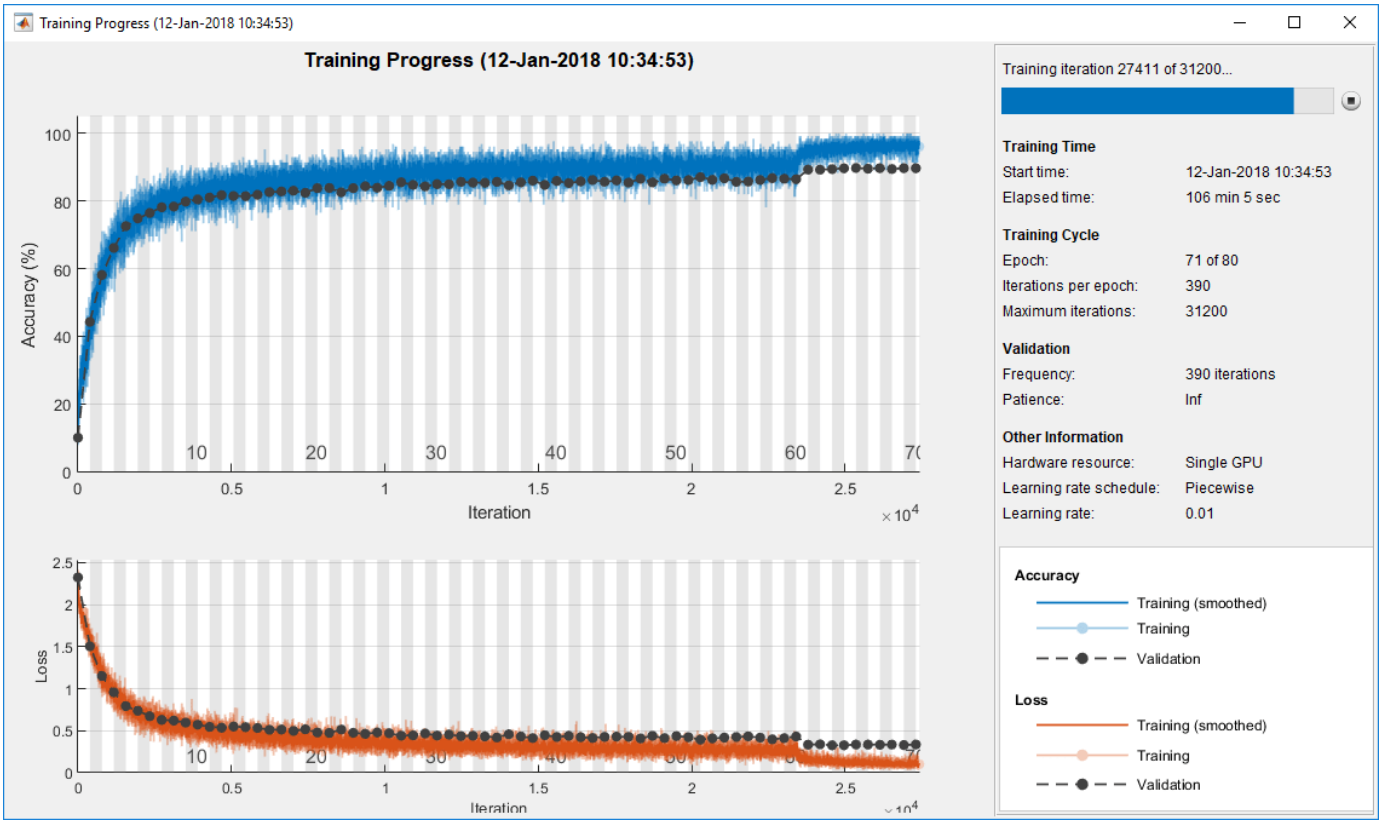
The figure marks each training **Epoch** using a shaded background. An epoch is a full pass through the entire data set.

During training, you can stop training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving. After you click the stop button, it can take a while for the training to complete. Once training is complete, `trainNetwork` returns the trained network.

When training finishes, view the **Results** showing the finalized validation accuracy and the reason that training finished. If the `'OutputNetwork'` training option is set to `'last-iteration'` (default), the finalized metrics correspond to the last training iteration. If the `'OutputNetwork'` training option is set to `'best-validation-loss'`, the finalized metrics correspond to the iteration with the lowest validation loss. The iteration from which the final validation metrics are calculated is labeled **Final** in the plots.

If your network contains batch normalization layers, then the final validation metrics can be different to the validation metrics evaluated during training. This is because the mean and variance statistics used for batch normalization can be different after training completes. For example, if the `'BatchNormalizationStatisics'` training option is `'population'`, then after training, the software finalizes the batch normalization statistics by passing through the training data once more and uses the resulting mean and variance. If the `'BatchNormalizationStatisics'` training option is `'moving'`, then the software approximates the statistics during training using a running estimate and uses the latest values of the statistics.

On the right, view information about the training time and settings. To learn more about training options, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.



Plot Training Progress During Training

Train a network and plot the training progress during training.

Load the training data, which contains 5000 images of digits. Set aside 1000 of the images for network validation.

```
[XTrain,YTrain] = digitTrain4DArrayData;

idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,:,idx);
XTrain(:,:,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Construct a network to classify the digit image data.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)
```

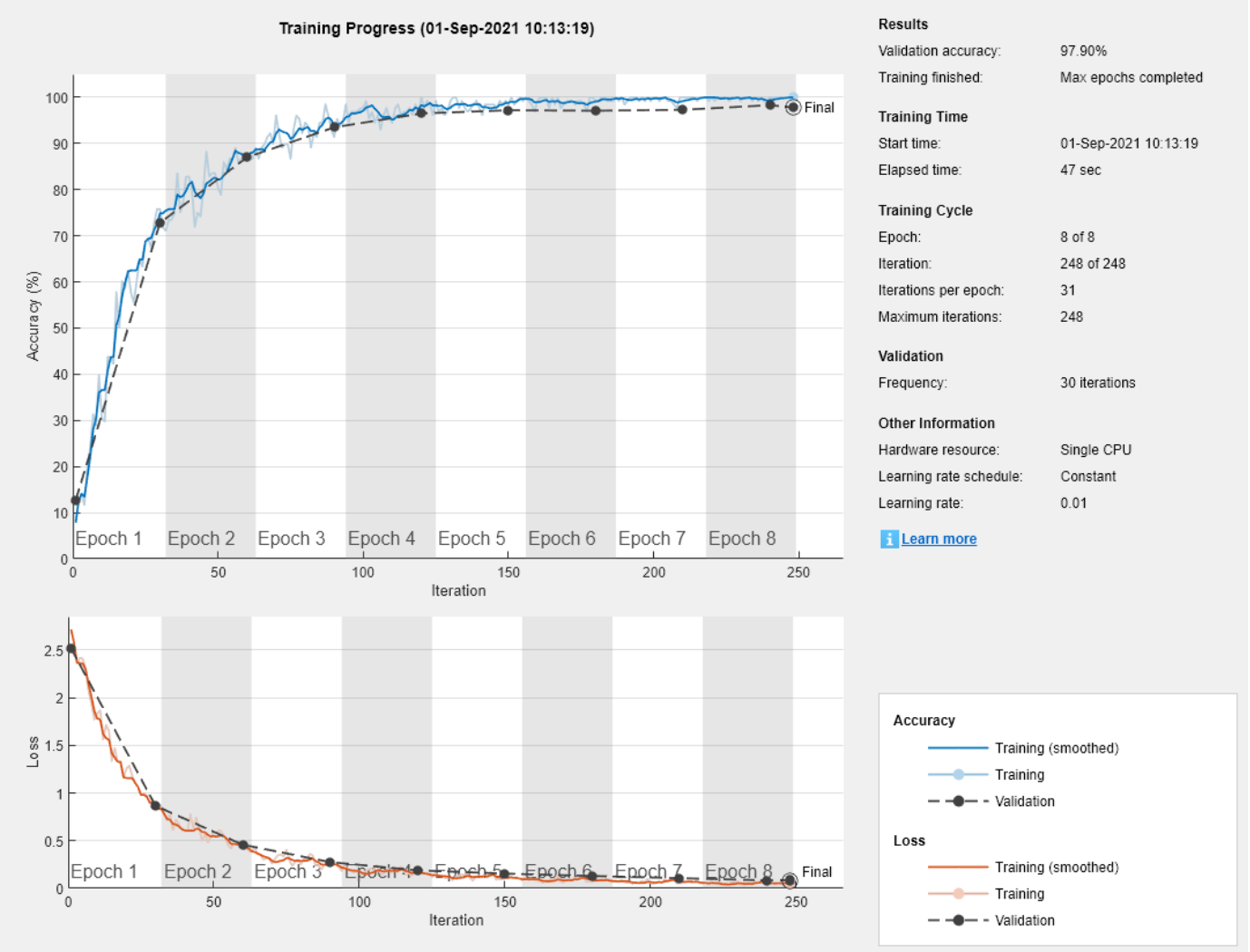
```
    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify options for network training. To validate the network at regular intervals during training, specify validation data. Choose the `'ValidationFrequency'` value so that the network is validated about once per epoch. To plot training progress during training, specify `'training-progress'` as the `'Plots'` value.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',8, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



See Also

`trainNetwork` | `trainingOptions`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-17
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Customize Output During Deep Learning Network Training

This example shows how to define an output function that runs at each iteration during training of deep learning neural networks. If you specify output functions by using the `'OutputFcn'` name-value pair argument of `trainingOptions`, then `trainNetwork` calls these functions once before the start of training, after each training iteration, and once after training has finished. Each time the output functions are called, `trainNetwork` passes a structure containing information such as the current iteration number, loss, and accuracy. You can use output functions to display or plot progress information, or to stop training. To stop training early, make your output function return `true`. If any output function returns `true`, then training finishes and `trainNetwork` returns the latest network.

To stop training when the loss on the validation set stops decreasing, simply specify validation data and a validation patience using the `'ValidationData'` and the `'ValidationPatience'` name-value pair arguments of `trainingOptions`, respectively. The validation patience is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops. You can add additional stopping criteria using output functions. This example shows how to create an output function that stops training when the classification accuracy on the validation data stops improving. The output function is defined at the end of the script.

Load the training data, which contains 5000 images of digits. Set aside 1000 of the images for network validation.

```
[XTrain,YTrain] = digitTrain4DArrayData;

idx = randperm(size(XTrain,4),1000);
XValidation = XTrain(:,:,:,idx);
XTrain(:,:,:,idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

Construct a network to classify the digit image data.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Specify options for network training. To validate the network at regular intervals during training, specify validation data. Choose the `'ValidationFrequency'` value so that the network is validated once per epoch.

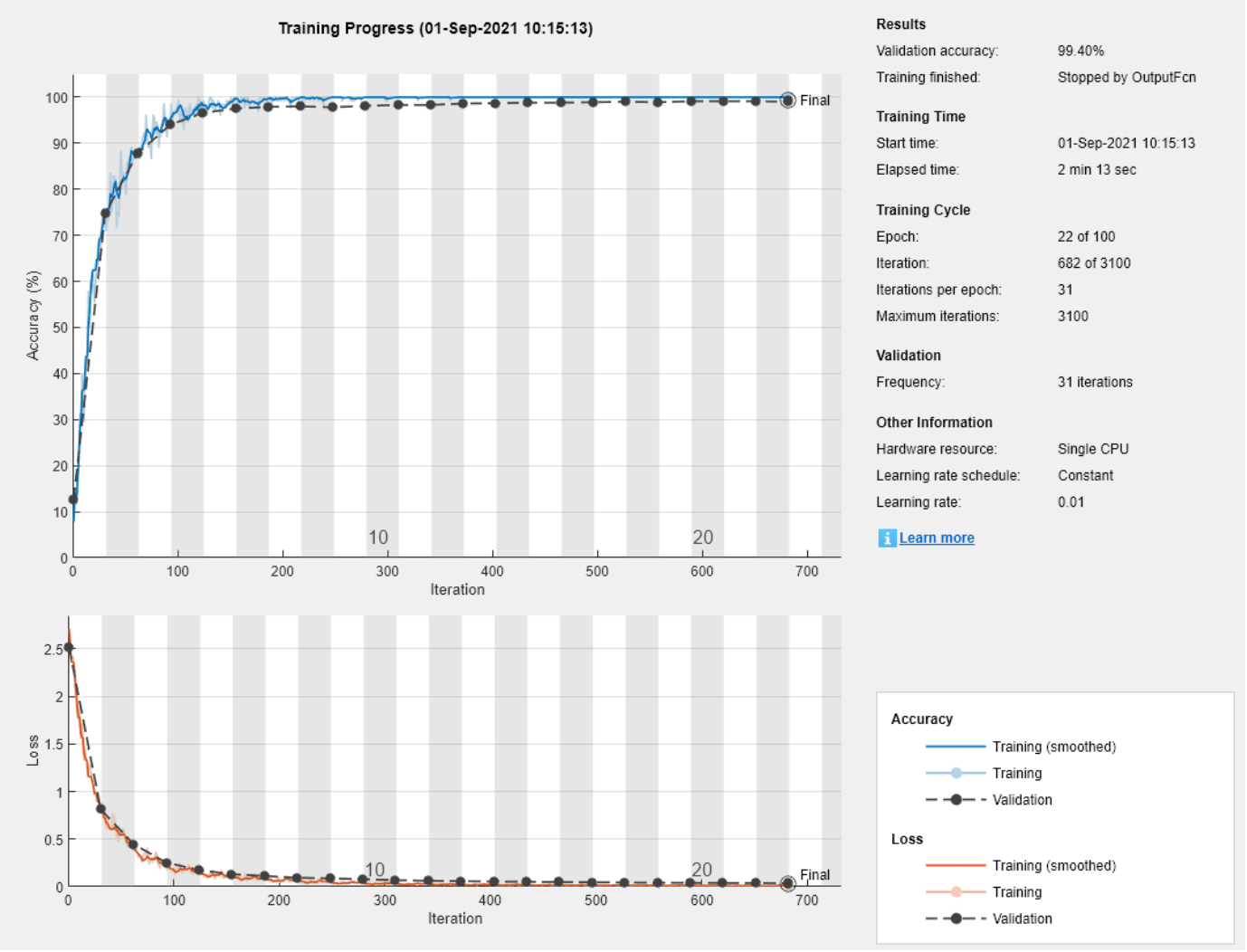
To stop training when the classification accuracy on the validation set stops improving, specify `stopIfAccuracyNotImproving` as an output function. The second input argument of `stopIfAccuracyNotImproving` is the number of times that the accuracy on the validation set can be smaller than or equal to the previously highest accuracy before network training stops. Choose any large value for the maximum number of epochs to train. Training should not reach the final epoch because training stops automatically.

```
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',100, ...
    'MiniBatchSize',miniBatchSize, ...
    'VerboseFrequency',validationFrequency, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',validationFrequency, ...
    'Plots','training-progress', ...
    'OutputFcn',@(info)stopIfAccuracyNotImproving(info,3));
```

Train the network. Training stops when the validation accuracy stops increasing.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:05 |        7.81% |       12.70% |       2.7155 |       2.5
|       1 |          31 |       00:00:12 |       71.88% |       74.70% |       0.8804 |       0.8
|       2 |          62 |       00:00:18 |       86.72% |       87.80% |       0.3929 |       0.4
|       3 |          93 |       00:00:23 |       94.53% |       94.00% |       0.2230 |       0.2
|       4 |         124 |       00:00:29 |       96.09% |       96.60% |       0.1482 |       0.1
|       5 |         155 |       00:00:35 |       99.22% |       97.50% |       0.1017 |       0.1
|       6 |         186 |       00:00:41 |       99.22% |       97.90% |       0.0783 |       0.1
|       7 |         217 |       00:00:46 |      100.00% |       98.00% |       0.0558 |       0.0
|       8 |         248 |       00:00:53 |      100.00% |       97.80% |       0.0441 |       0.0
|       9 |         279 |       00:00:57 |      100.00% |       98.10% |       0.0349 |       0.0
|      10 |         310 |       00:01:03 |      100.00% |       98.30% |       0.0275 |       0.0
|      11 |         341 |       00:01:09 |      100.00% |       98.30% |       0.0242 |       0.0
|      12 |         372 |       00:01:14 |      100.00% |       98.60% |       0.0217 |       0.0
|      13 |         403 |       00:01:19 |      100.00% |       98.70% |       0.0191 |       0.0
|      14 |         434 |       00:01:24 |      100.00% |       98.80% |       0.0167 |       0.0
|      15 |         465 |       00:01:30 |      100.00% |       98.80% |       0.0145 |       0.0
|      16 |         496 |       00:01:35 |      100.00% |       98.90% |       0.0127 |       0.0
|      17 |         527 |       00:01:41 |      100.00% |       99.00% |       0.0112 |       0.0
|      18 |         558 |       00:01:47 |      100.00% |       98.90% |       0.0101 |       0.0
|      19 |         589 |       00:01:52 |      100.00% |       99.10% |       0.0092 |       0.0
|      20 |         620 |       00:01:59 |      100.00% |       99.10% |       0.0086 |       0.0
|      21 |         651 |       00:02:06 |      100.00% |       99.10% |       0.0080 |       0.0
|      22 |         682 |       00:02:12 |      100.00% |       99.00% |       0.0076 |       0.0
|=========================================================================================================
Training finished: Stopped by OutputFcn.
```



Define Output Function

Define the output function `stopIfAccuracyNotImproving(info,N)`, which stops network training if the best classification accuracy on the validation data does not improve for `N` network validations in a row. This criterion is similar to the built-in stopping criterion using the validation loss, except that it applies to the classification accuracy instead of the loss.

```
function stop = stopIfAccuracyNotImproving(info,N)

stop = false;

% Keep track of the best validation accuracy and the number of validations for which
% there has not been an improvement of the accuracy.
persistent bestValAccuracy
persistent valLag

% Clear the variables when training starts.
if info.State == "start"
    bestValAccuracy = 0;
    valLag = 0;
```

```
elseif ~isempty(info.ValidationLoss)

    % Compare the current validation accuracy to the best accuracy so far,
    % and either set the best accuracy to the current accuracy, or increase
    % the number of validations for which there has not been an improvement.
    if info.ValidationAccuracy > bestValAccuracy
        valLag = 0;
        bestValAccuracy = info.ValidationAccuracy;
    else
        valLag = valLag + 1;
    end

    % If the validation lag is at least N, that is, the validation accuracy
    % has not improved for at least N validations, then return true and
    % stop training.
    if valLag >= N
        stop = true;
    end

end

end
```

See Also

`trainNetwork` | `trainingOptions`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-17
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-42
- “Deep Learning in MATLAB” on page 1-2
- “Compare Layer Weight Initializers” on page 18-181
- “Specify Custom Weight Initialization Function” on page 18-175

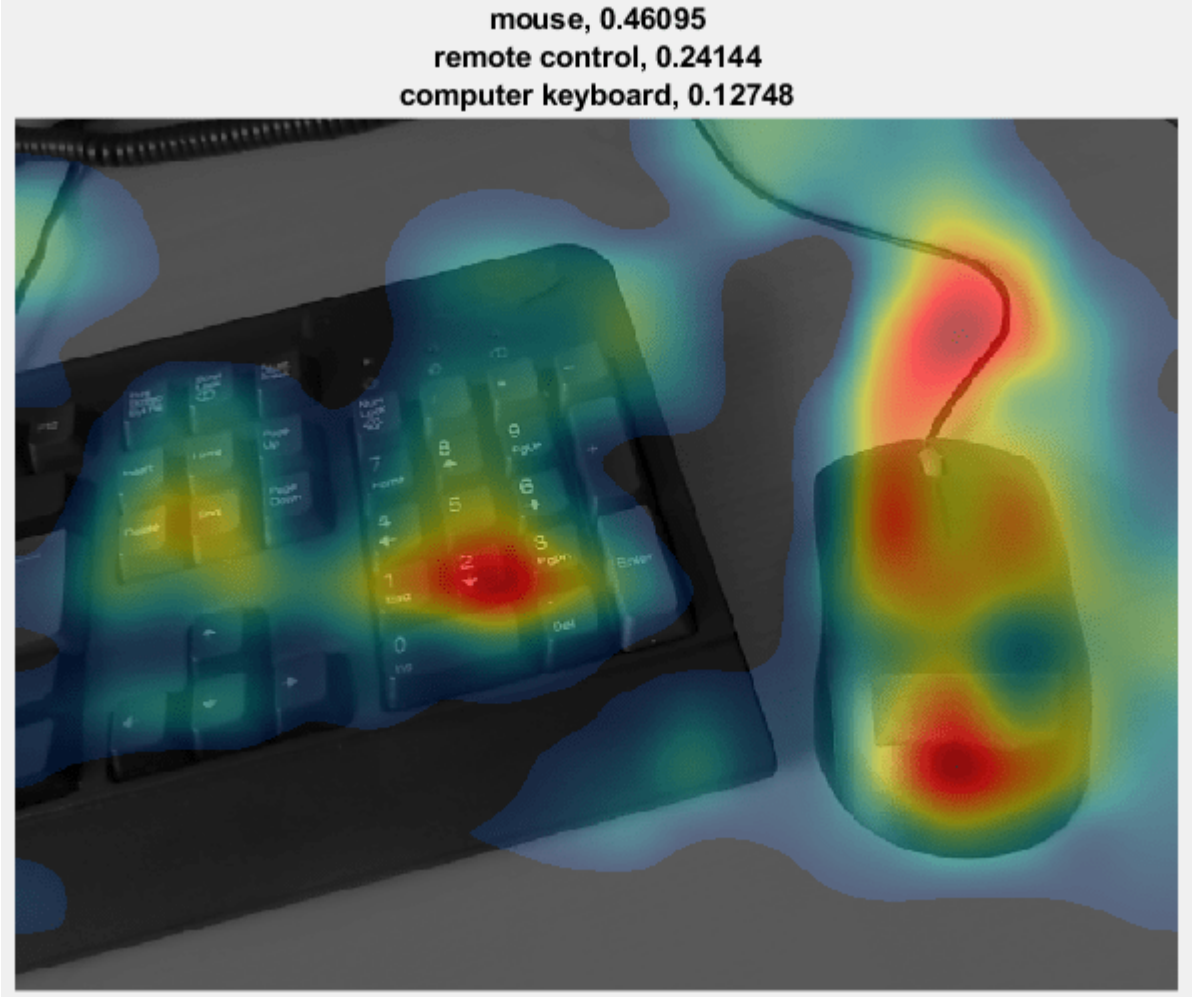
Investigate Network Predictions Using Class Activation Mapping

This example shows how to use class activation mapping (CAM) to investigate and explain the predictions of a deep convolutional neural network for image classification.

Deep learning networks are often considered to be "black boxes" that offer no way of figuring out what a network has learned or which part of an input to the network was responsible for the prediction of the network. When these models fail and give incorrect predictions, they often fail spectacularly without any warning or explanation. Class activation mapping [1] is one technique that you can use to get visual explanations of the predictions of convolutional neural networks. Incorrect, seemingly unreasonable predictions can often have reasonable explanations. Using class activation mapping, you can check if a specific part of an input image "confused" the network and led it to make an incorrect prediction.

You can use class activation mapping to identify bias in the training set and increase model accuracy. If you discover that the network bases predictions on the wrong features, then you can make the network more robust by collecting better data. For example, suppose that you train a network to distinguish images of cats and dogs. The network has high accuracy on the training set, but performs poorly on real-world examples. By using class activation mapping on the training examples, you discover that the network is basing predictions not on the cats and dogs in the images, but on the backgrounds. You then realize that all your cat pictures have red backgrounds, all your dog pictures have green backgrounds, and that it is the color of the background that the network learned during training. You can then collect new data that does not have this bias.

This example class activation map shows which regions of the input image contribute the most to the predicted class `mouse`. Red regions contribute the most.



Load Pretrained Network and Webcam

Load a pretrained convolutional neural network for image classification. SqueezeNet, GoogLeNet, ResNet-18, and MobileNet-v2 are relatively fast networks. SqueezeNet is the fastest network and its class activation map has four times higher resolution than the maps of the other networks. You cannot use class activation mapping with networks that have multiple fully connected layers at the end of the network, such as AlexNet, VGG-16, and VGG-19.

```
netName = "squeezenet";
net = eval(netName);
```

Create a `webcam` object and connect to your webcam.

```
camera = webcam;
```

Extract the image input size and the output classes of the network. The `activationLayerName` helper function, defined at the end of this example, returns the name of the layer to extract the activations from. This layer is the ReLU layer that follows the last convolutional layer of the network.

```
inputSize = net.Layers(1).InputSize(1:2);
classes = net.Layers(end).Classes;
layerName = activationLayerName(netName);
```

Display Class Activation Maps

Create a figure and perform class activation mapping in a loop. To terminate execution of the loop, close the figure.

```
h = figure('Units','normalized','Position',[0.05 0.05 0.9 0.8],'Visible','on');

while ishandle(h)
```

Take a snapshot using the webcam. Resize the image so that the length of its shortest side (in this case, the image height) equals the image input size of the network. As you resize, preserve the aspect ratio of the image. You can also resize the image to a larger or smaller size. Making the image larger increases the resolution of the final class activation map, but can lead to less accurate overall predictions.

Compute the activations of the resized image in the ReLU layer that follows the last convolutional layer of the network.

```
    im = snapshot(camera);
    imResized = imresize(im,[inputSize(1), NaN]);
    imageActivations = activations(net,imResized,layerName);
```

The class activation map for a specific class is the activation map of the ReLU layer that follows the final convolutional layer, weighted by how much each activation contributes to the final score of that class. Those weights equal the weights of the final fully connected layer of the network for that class. SqueezeNet does not have a final fully connected layer. Instead, the output of the ReLU layer that follows the last convolutional layer is already the class activation map.

You can generate a class activation map for any output class. For example, if the network makes an incorrect classification, you can compare the class activation maps for the true and predicted classes. For this example, generate the class activation map for the predicted class with the highest score.

```
    scores = squeeze(mean(imageActivations,[1 2]));

    if netName ~= "squeezenet"
        fcWeights = net.Layers(end-2).Weights;
        fcBias = net.Layers(end-2).Bias;
        scores =  fcWeights*scores + fcBias;

        [~,classIds] = maxk(scores,3);

        weightVector = shiftdim(fcWeights(classIds(1),:),-1);
        classActivationMap = sum(imageActivations.*weightVector,3);
    else
        [~,classIds] = maxk(scores,3);
        classActivationMap = imageActivations(:,:,classIds(1));
    end
```

Calculate the top class labels and the final normalized class scores.

```
    scores = exp(scores)/sum(exp(scores));
    maxScores = scores(classIds);
    labels = classes(classIds);
```

Plot the class activation map. Display the original image in the first subplot. In the second subplot, use the `CAMshow` helper function, defined at the end of this example, to display the class activation map on top of a darkened grayscale version of the original image. Display the top three predicted labels with their predicted scores.

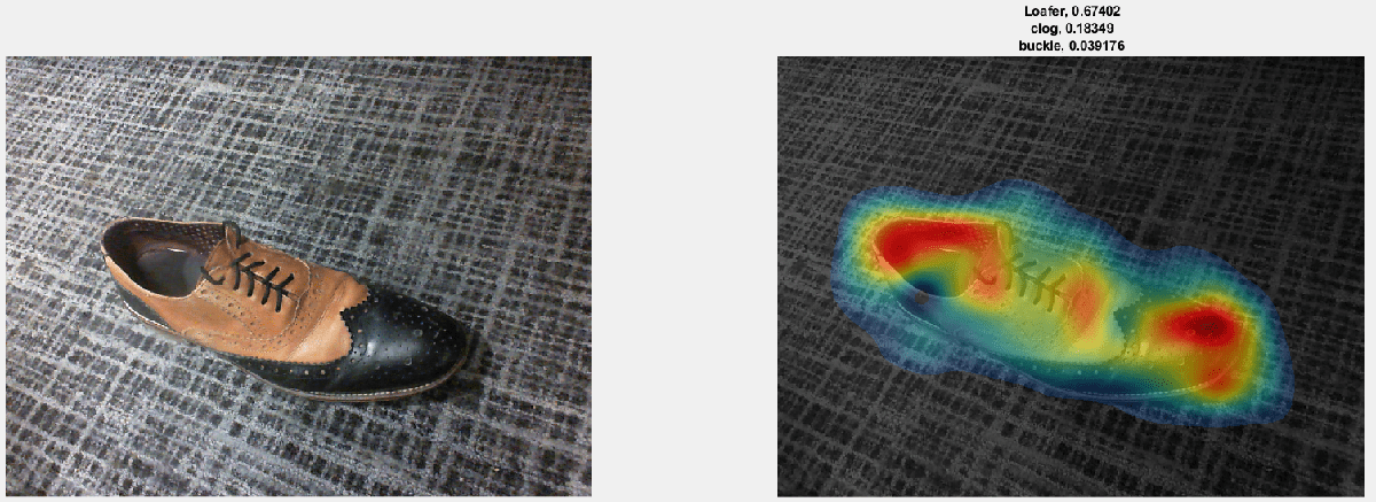
```
    subplot(1,2,1)
    imshow(im)

    subplot(1,2,2)
    CAMshow(im,classActivationMap)
    title(string(labels) + ", " + string(maxScores));

    drawnow

end
```

Clear the webcam object.

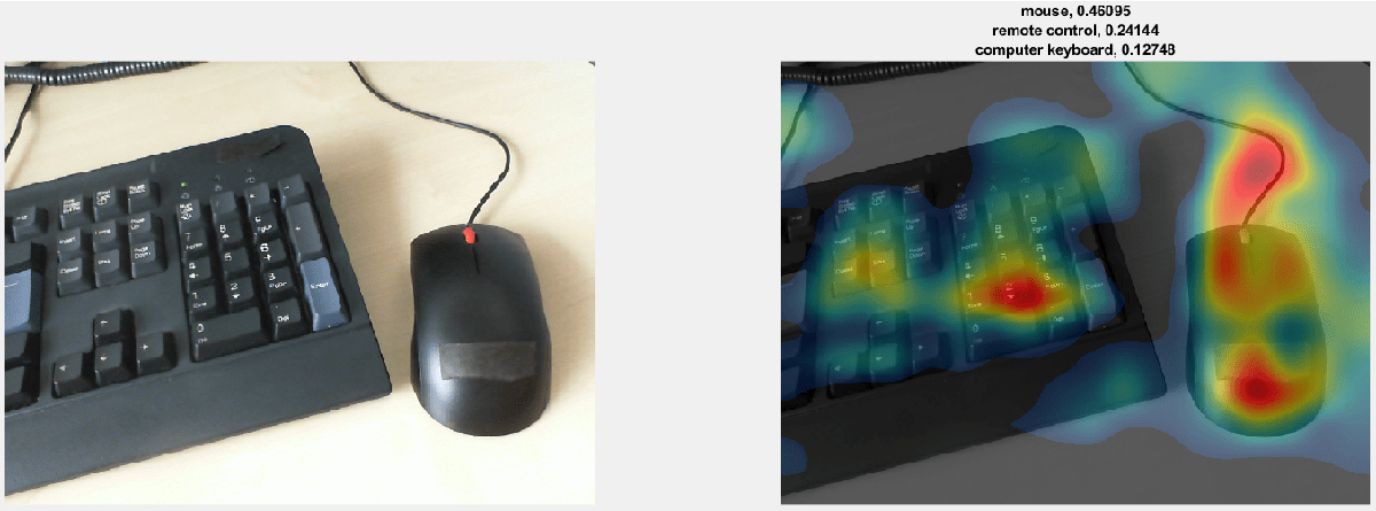
```
clear camera
```

Example Maps

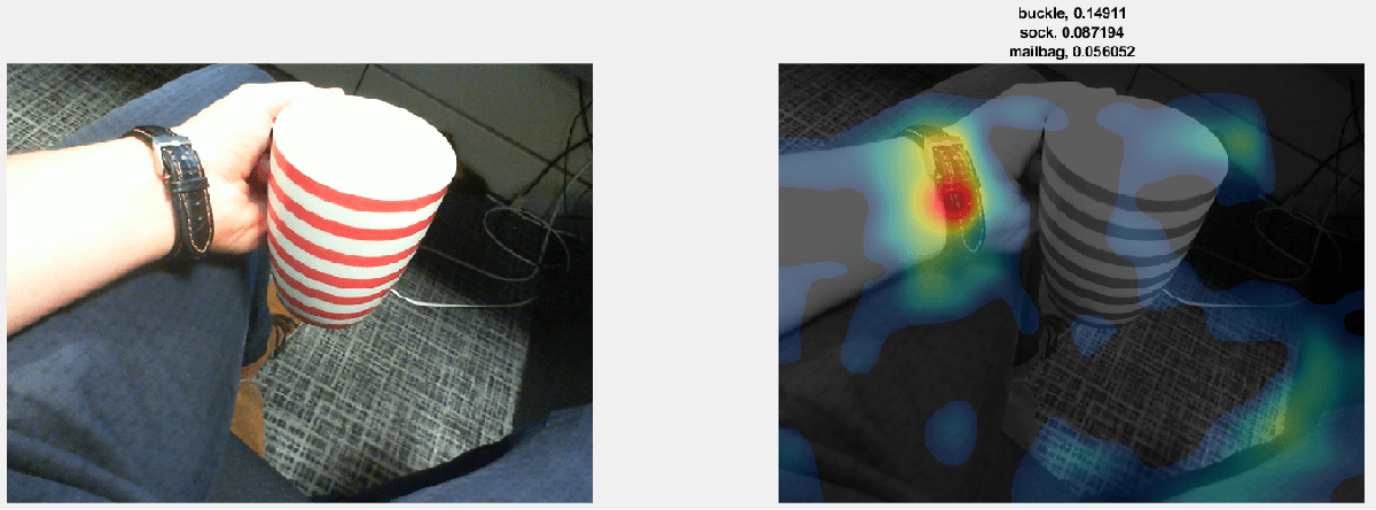
The network correctly identifies the object in this image as a loafer (a type of shoe). The class activation map in the image to the right shows the contribution of each region of the input image to the predicted class `Loafer`. Red regions contribute the most. The network bases its classification on the entire shoe, but the strongest input comes from the red areas – that is, the tip and the opening of the shoe.



The network classifies this image as a mouse. As the class activation map shows, the prediction is based not only on the mouse in the image, but also the keyboard. Because the training set likely has many images of mice next to keyboards, the network predicts that images containing keyboards are more likely to contain mice.



The network classifies this image of a coffee cup as a buckle. As the class activation map shows, the network misclassifies the image because the image contains too many confounding objects. The network detects and focuses on the watch wristband, not the coffee cup.



Helper Functions

`CAMshow(im,CAM)` overlays the class activation map `CAM` on a darkened, grayscale version of the image `im`. The function resizes the class activation map to the size of `im`, normalizes it, thresholds it from below, and visualizes it using a `jet` colormap.

```
function CAMshow(im,CAM)
imSize = size(im);
CAM = imresize(CAM,imSize(1:2));
CAM = normalizeImage(CAM);
CAM(CAM<0.2) = 0;
cmap = jet(255).*linspace(0,1,255)';
CAM = ind2rgb(uint8(CAM*255),cmap)*255;

combinedImage = double(rgb2gray(im))/2 + CAM;
combinedImage = normalizeImage(combinedImage)*255;
```

```
imshow(uint8(combinedImage));
end

function N = normalizeImage(I)
minimum = min(I(:));
maximum = max(I(:));
N = (I-minimum)/(maximum-minimum);
end

function layerName = activationLayerName(netName)

if netName == "squeezenet"
    layerName = 'relu_conv10';
elseif netName == "googlenet"
    layerName = 'inception_5b-output';
elseif netName == "resnet18"
    layerName = 'res5b_relu';
elseif netName == "mobilenetv2"
    layerName = 'out_relu';
end

end
```

References

[1] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. "Learning deep features for discriminative localization." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2921-2929. 2016.

See Also

`activations` | `squeezenet` | `occlusionSensitivity` | `gradCAM` | `imageLIME`

Related Examples

- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Deep Learning Visualization Methods” on page 5-186
- “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2
- “Understand Network Predictions Using LIME” on page 5-42
- “Understand Network Predictions Using Occlusion” on page 5-24

View Network Behavior Using tsne

This example shows how to use the `tsne` function to view activations in a trained network. This view can help you understand how a network works.

The `tsne` (Statistics and Machine Learning Toolbox) function in Statistics and Machine Learning Toolbox™ implements t-distributed stochastic neighbor embedding (t-SNE) [1]. This technique maps high-dimensional data (such as network activations in a layer) to two dimensions. The technique uses a nonlinear map that attempts to preserve distances. By using t-SNE to visualize the network activations, you can gain an understanding of how the network responds.

You can use t-SNE to visualize how deep learning networks change the representation of input data as it passes through the network layers. You can also use t-SNE to find issues with the input data and to understand which observations the network classifies incorrectly.

For example, t-SNE can reduce the multidimensional activations of a softmax layer to a 2-D representation with a similar structure. Tight clusters in the resulting t-SNE plot correspond to classes that the network usually classifies correctly. The visualization allows you to find points that appear in the wrong cluster, indicating an observation that the network classifies incorrectly. The observation might be labeled incorrectly, or the network might predict that an observation is an instance of a different class because it appears similar to other observations from that class. Note that the t-SNE reduction of the softmax activations uses only those activations, not the underlying observations.

Download Data Set

This example uses the Example Food Images data set, which contains 978 photographs of food in nine classes and is approximately 77 MB in size. Download the data set into your temporary directory by calling the `downloadExampleFoodImagesData` helper function; the code for this helper function appears at the end of this example on page 5-0 .

```
dataDir = fullfile(tempdir, "ExampleFoodImageDataset");
url = "https://www.mathworks.com/supportfiles/nnet/data/ExampleFoodImageDataset.zip";

if ~exist(dataDir, "dir")
    mkdir(dataDir);
end

downloadExampleFoodImagesData(url,dataDir);
```

```
Downloading MathWorks Example Food Image dataset...
This can take several minutes to download...
Download finished...
Unzipping file...
Unzipping finished...
Done.
```

Train Network to Classify Food Images

Modify the SqueezeNet pretrained network to classify images of food from the data set. Replace the final convolutional layer, which has 1000 filters for the 1000 classes of ImageNet, with a new convolutional layer that has only nine filters. Each filter corresponds to a single type of food.

```
lgraph = layerGraph(squeezenet());
lgraph = lgraph.replaceLayer("ClassificationLayer_predictions",...
    classificationLayer("Name", "ClassificationLayer_predictions"));
```

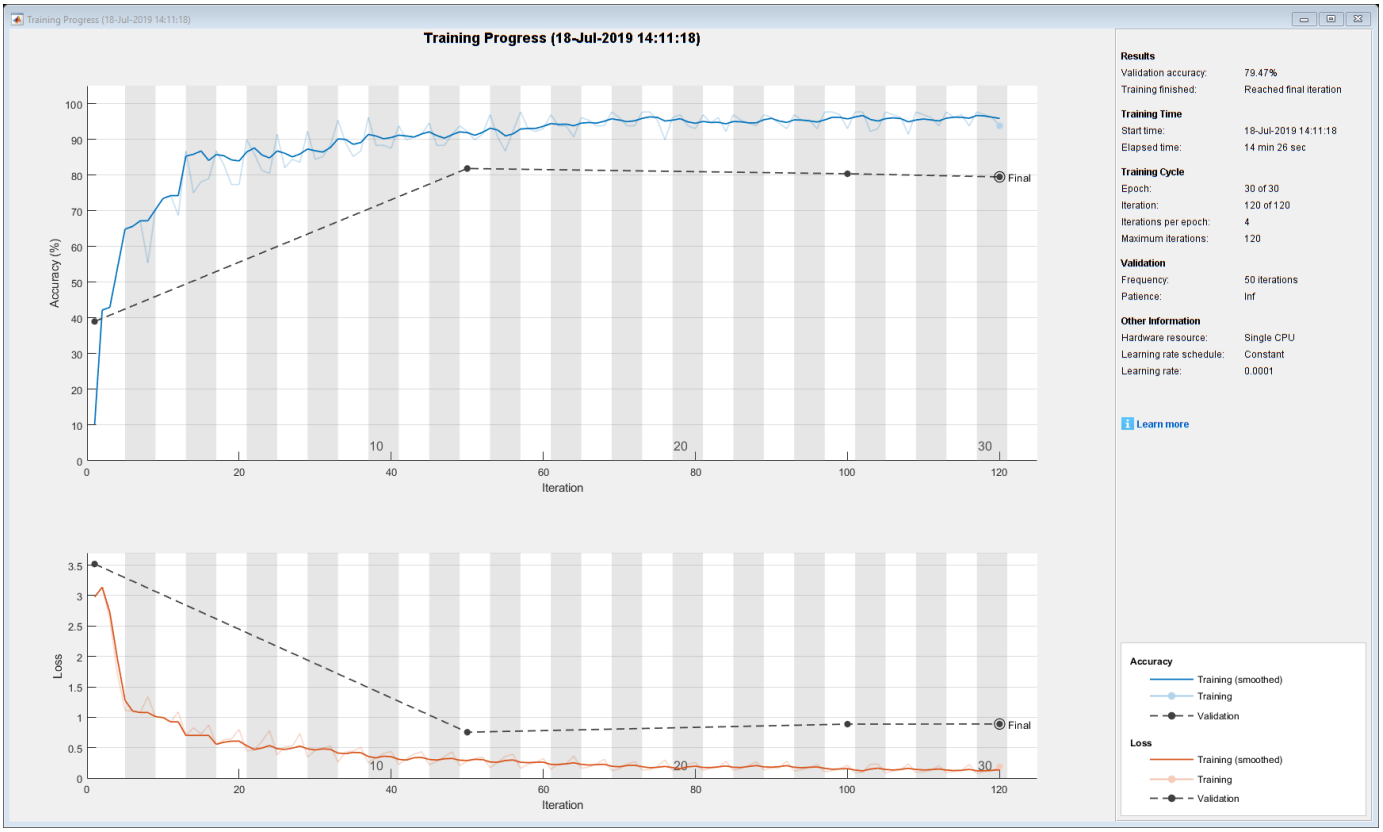
```
newConv =  convolution2dLayer([14 14], 9, "Name", "conv", "Padding", "same");
lgraph = lgraph.replaceLayer("conv10", newConv);
```

Create an `imageDatastore` containing paths to the image data. Split the datastore into training and validation sets, using 65% of the data for training and the rest for validation. Because the data set is fairly small, overfitting is a significant issue. To minimize overfitting, augment the training set with random flips and scaling.

```
imds = imageDatastore(dataDir, ...
    "IncludeSubfolders", true, "LabelSource", "foldernames");

aug = imageDataAugmenter("RandXReflection", true, ...
    "RandYReflection", true, ...
    "RandXScale", [0.8 1.2], ...
    "RandYScale", [0.8 1.2]);

trainingFraction = 0.65;
[trainImds,valImds] = splitEachLabel(imds, trainingFraction);

augImdsTrain = augmentedImageDatastore([227 227], trainImds, ...
    'DataAugmentation', aug);
augImdsVal = augmentedImageDatastore([227 227], valImds);
```

Create training options and train the network. SqueezeNet is a small network that is quick to train. You can train on a GPU or a CPU; this example trains on a CPU.

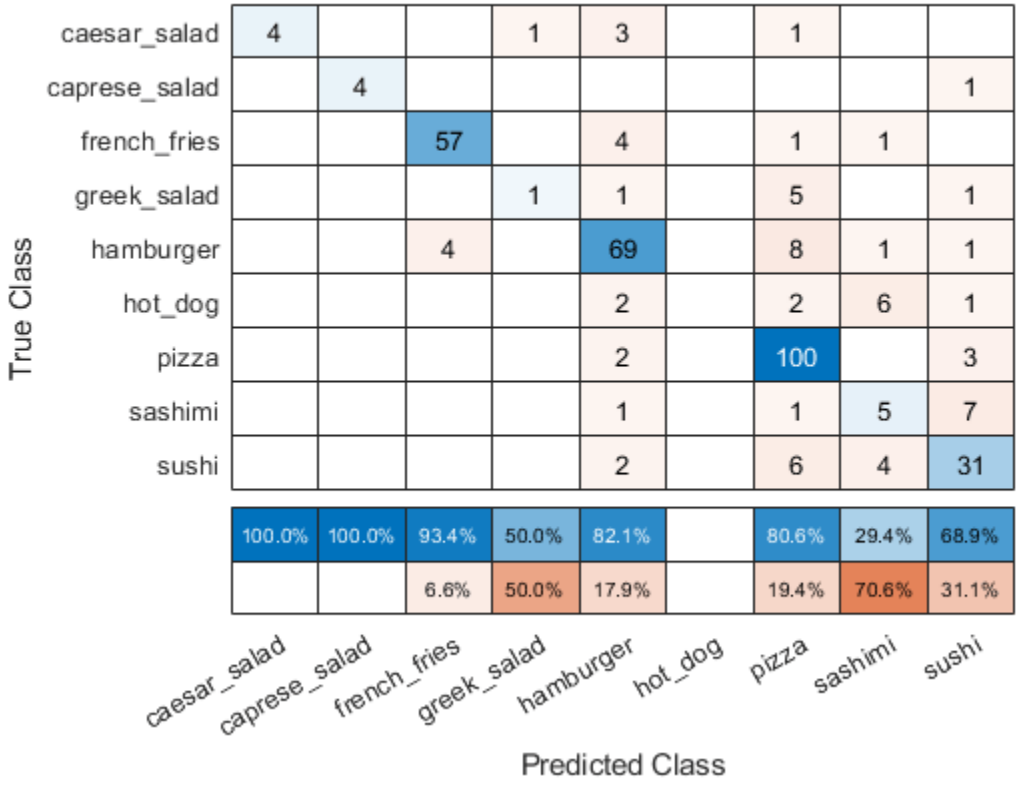
```
opts = trainingOptions("adam", ...
    "InitialLearnRate", 1e-4, ...
    "MaxEpochs", 30, ...
    "ValidationData", augImdsVal, ...
    "Verbose", false,...
    "Plots", "training-progress", ...
    "ExecutionEnvironment","cpu",...
    "MiniBatchSize",128);
rng default
net = trainNetwork(augImdsTrain, lgraph, opts);
```



Classify Validation Data

Use the network to classify images in the validation set. To verify that the network is reasonably accurate at classifying new data, plot a confusion matrix of the true and predicted labels.

```
figure();
YPred = classify(net,augImdsVal);
confusionchart(valImds.Labels,YPred,'ColumnSummary',"column-normalized")
```



The network classifies several images well. The network appears to have trouble with sushi images, classifying many as sushi but some as pizza or hamburger. The network does not classify any images into the hot dog class.

Compute Activations for Several Layers

To continue to analyze the network performance, compute activations for every observation in the data set at an early max pooling layer, the final convolutional layer, and the final softmax layer. Output the activations as an NxM matrix, where N is the number of observations and M is the number of dimensions of the activation. M is the product of spatial and channel dimensions. Each row is an observation, and each column is a dimension. At the softmax layer M = 9, because the food data set has nine classes. Each row in the matrix contains nine elements, corresponding to the probabilities that an observation belongs to each of the nine classes of food.

```
earlyLayerName = "pool1";
finalConvLayerName = "conv";
softmaxLayerName = "prob";
pool1Activations = activations(net,...
    augImdsVal,earlyLayerName,"OutputAs","rows");
finalConvActivations = activations(net,...
    augImdsVal,finalConvLayerName,"OutputAs","rows");
softmaxActivations = activations(net,...
    augImdsVal,softmaxLayerName,"OutputAs","rows");
```

Ambiguity of Classifications

You can use the softmax activations to calculate the image classifications that are most likely to be incorrect. Define the *ambiguity* of a classification as the ratio of the second-largest probability to the largest probability. The ambiguity of a classification is between zero (nearly certain classification) and 1 (nearly as likely to be classified to the most likely class as the second class). An ambiguity of near 1 means the network is unsure of the class in which a particular image belongs. This uncertainty might be caused by two classes whose observations appear so similar to the network that it cannot learn the differences between them. Or, a high ambiguity can occur because a particular observation contains elements of more than one class, so the network cannot decide which classification is correct. Note that low ambiguity does not necessarily imply correct classification; even if the network has a high probability for a class, the classification can still be incorrect.

```
[R,RI] = maxk(softmaxActivations,2,2);
ambiguity = R(:,2)./R(:,1);
```

Find the most ambiguous images.

```
[ambiguity,ambiguityIdx] = sort(ambiguity,"descend");
```

View the most probable classes of the ambiguous images and the true classes.

```
classList = unique(valImds.Labels);
top10Idx = ambiguityIdx(1:10);
top10Ambiguity = ambiguity(1:10);
mostLikely = classList(RI(ambiguityIdx,1));
secondLikely = classList(RI(ambiguityIdx,2));
table(top10Idx,top10Ambiguity,mostLikely(1:10),secondLikely(1:10),valImds.Labels(ambiguityIdx(1:1
    'VariableNames',["Image #","Ambiguity","Likeliest","Second","True Class"])
```

```
ans=10×5 table
    Image #    Ambiguity    Likeliest       Second        True Class
    _______    _________    _________    ____________    ____________

       94        0.9879     hamburger    pizza           hamburger
      175       0.96311     hamburger    french_fries    hot_dog
      179       0.94939     pizza        hamburger       hot_dog
      337       0.93426     sushi        sashimi         sushi
      256       0.92972     sushi        pizza           pizza
      297       0.91776     sushi        sashimi         sashimi
      283       0.80407     pizza        sushi           pizza
       27       0.80278     hamburger    pizza           french_fries
      302       0.79283     sashimi      sushi           sushi
      201       0.76034     pizza        greek_salad     pizza
```

The network predicts that image 27 is most likely hamburger or pizza. However, this image is actually French fries. View the image to see why this misclassification might occur.

```
v = 27;
figure();
imshow(valImds.Files{v});
title(sprintf("Observation: %i\n" + ...
    "Actual: %s. Predicted: %s", v, ...
    string(valImds.Labels(v)), string(YPred(v))), ...
    'Interpreter', 'none');
```

The image contains several distinct regions, some of which might confuse the network.

Compute 2-D Representations of Data Using t-SNE

Calculate a low-dimensional representation of the network data for an early max pooling layer, the final convolutional layer, and the final softmax layer. Use the `tsne` function to reduce the dimensionality of the activation data from M to 2. The larger the dimensionality of the activations, the longer the t-SNE computation takes. Therefore, computation for the early max pooling layer, where activations have 200,704 dimensions, takes longer than for the final softmax layer. Set the random seed for reproducibility of the t-SNE result.

```
rng default
pool1tsne = tsne(pool1Activations);
finalConvtsne = tsne(finalConvActivations);
softmaxtsne = tsne(softmaxActivations);
```

Compare Network Behavior for Early and Later Layers

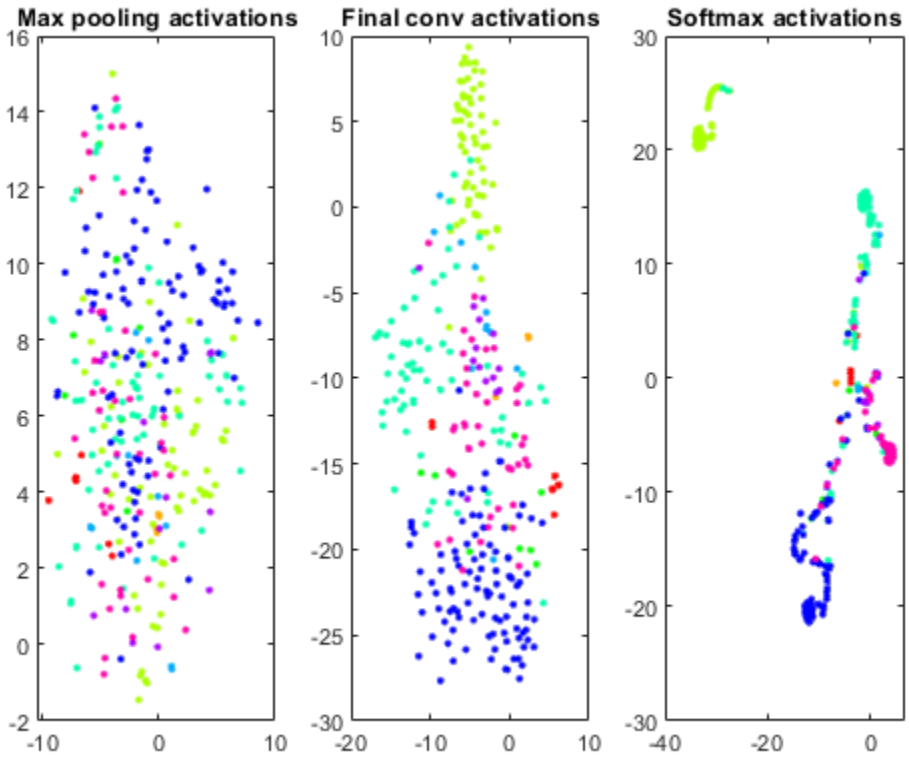
The t-SNE technique tries to preserve distances so that points near each other in the high-dimensional representation are also near each other in the low-dimensional representation. As shown in the confusion matrix, the network is effective at classifying into different classes. Therefore, images that are semantically similar (or of the same type), such as caesar salad and caprese salad,

are near each other in the softmax activations space. t-SNE captures this proximity in a 2-D representation that is easier to understand and plot than the nine-dimensional softmax scores.

Early layers tend to operate on low-level features such as edges and colors. Deeper layers have learned high-level features with more semantic meaning, such as the difference between a pizza and a hot dog. Therefore, activations from early layers do not show any clustering by class. Two images that are similar pixelwise (for example, they both contain a lot of green pixels) are near each other in the high-dimensional space of the activations, regardless of their semantic contents. Activations from later layers tend to cluster points from the same class together. This behavior is most pronounced at the softmax layer and is preserved in the two-dimensional t-SNE representation.

Plot the t-SNE data for the early max pooling layer, the final convolutional layer, and the final softmax layer using the `gscatter` function. Observe that the early max pooling activations do not exhibit any clustering between images of the same class. Activations of the final convolutional layer are clustered by class to some extent, but less so than the softmax activations. Different colors correspond to observations of different classes.

```
doLegend = 'off';
markerSize = 7;
figure;

subplot(1,3,1);
gscatter(pool1tsne(:,1),pool1tsne(:,2),valImds.Labels, ...
    [],'.',markerSize,doLegend);
title("Max pooling activations");

subplot(1,3,2);
gscatter(finalConvtsne(:,1),finalConvtsne(:,2),valImds.Labels, ...
    [],'.',markerSize,doLegend);
title("Final conv activations");

subplot(1,3,3);
gscatter(softmaxtsne(:,1),softmaxtsne(:,2),valImds.Labels, ...
    [],'.',markerSize,doLegend);
title("Softmax activations");
```

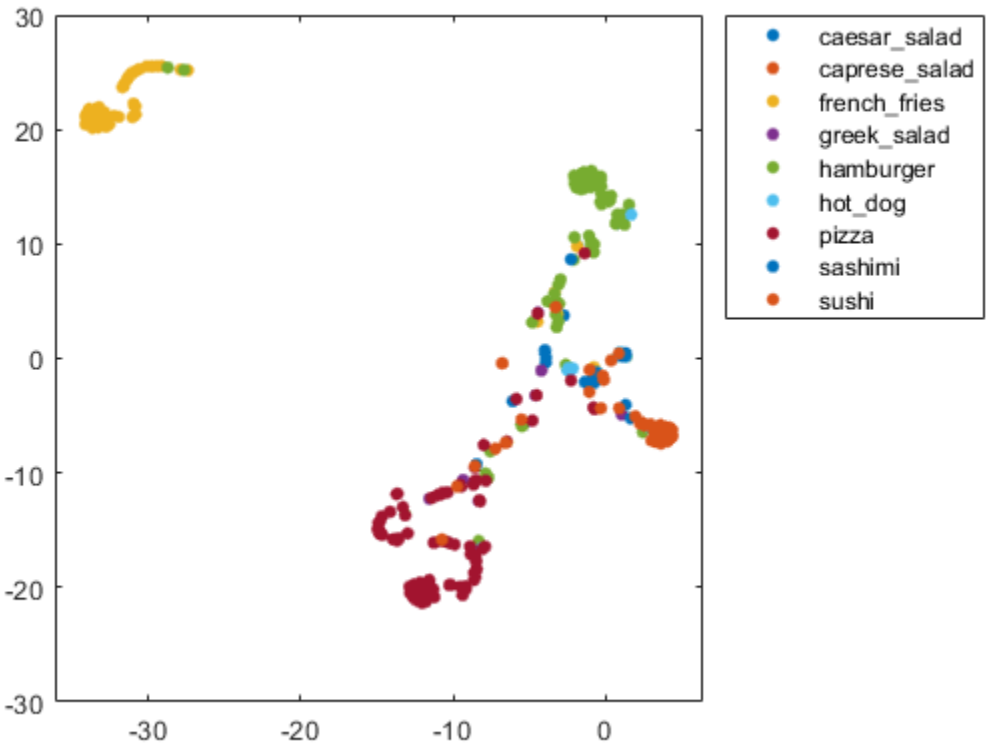


Explore Observations in t-SNE Plot

Create a larger plot of the softmax activations, including a legend labeling each class. From the t-SNE plot, you can understand more about the structure of the posterior probability distribution.

For example, the plot shows a distinct, separate cluster of French fries observations, whereas the sashimi and sushi clusters are not resolved very well. Similar to the confusion matrix, the plot suggests that the network is more accurate at predicting into the French fries class.

```
numClasses = length(classList);
colors = lines(numClasses);
h = figure;
gscatter(softmaxtsne(:,1),softmaxtsne(:,2),valImds.Labels,colors);

l = legend;
l.Interpreter = "none";
l.Location = "bestoutside";
```



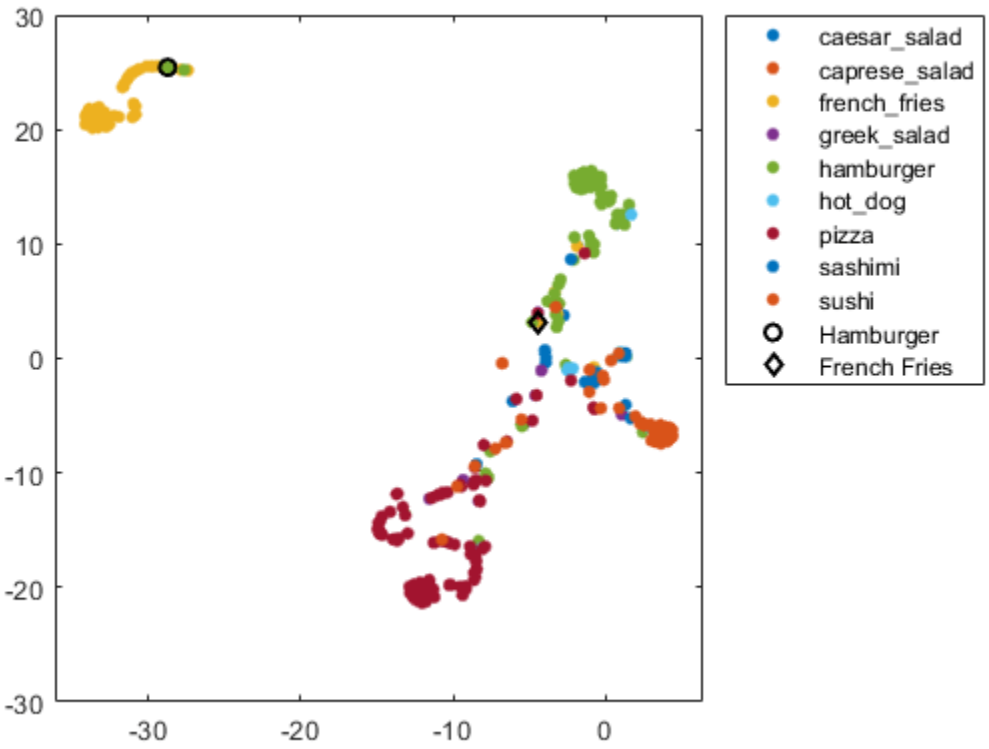
You can also use t-SNE to determine which images are misclassified by the network and why. Incorrect observations are often isolated points of the wrong color for their surrounding cluster. For example, a misclassified image of hamburger is very near the French fries region (the green dot nearest the center of the orange cluster). This dot is observation 99. Circle this observation on the t-SNE plot, and display the image with `imshow`.

```
obs =  99                ;
figure(h)
hold on;
hs = scatter(softmaxtsne(obs, 1), softmaxtsne(obs, 2), ...
    'black','LineWidth',1.5);
l.String{end} = 'Hamburger';
hold off;
figure();
imshow(valImds.Files{obs});
title(sprintf("Observation: %i\n" + ...
    "Actual: %s. Predicted: %s", obs, ...
    string(valImds.Labels(obs)), string(YPred(obs))), ...
    'Interpreter', 'none');
```

If an image contains multiple types of food, the network can get confused. In this case, the network classifies the image as French fries even though the food in the foreground is hamburger. The French fries visible at the edge of the image cause the confusion.

Similarly, the ambiguous image 27 (shown earlier in the example) has multiple regions. Examine the t-SNE plot highlighting the ambiguous aspect of this French fries image.

```
obs = 27                                ;
figure(h)
hold on;
h = scatter(softmaxtsne(obs, 1), softmaxtsne(obs, 2), ...
    'k','d','LineWidth',1.5);
l.String{end} = 'French Fries';
hold off;
```



The image is not in a well-defined cluster in the plot, which indicates that the classification is likely incorrect. The image is far from the French fries cluster, and close to the hamburger cluster.

The *why* of a misclassification must be provided by other information, typically a hypothesis based on the contents of the image. You can then test the hypothesis using other data, or using tools that indicate which spatial regions of an image are important to network classification. For examples, see `occlusionSensitivity` and “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21.

References

[1] van der Maaten, Laurens, and Geoffrey Hinton. "Visualizing Data using t-SNE." *Journal of Machine Learning Research 9,* 2008, pp. 2579–2605.

Helper Function

```
function downloadExampleFoodImagesData(url, dataDir)
% Download the Example Food Image data set, containing 978 images of
% different types of food split into 9 classes.

% Copyright 2019 The MathWorks, Inc.

fileName = "ExampleFoodImageDataset.zip";
fileFullPath = fullfile(dataDir, fileName);

% Download the .zip file into a temporary directory.
if ~exist(fileFullPath, "file")
```

```
    fprintf("Downloading MathWorks Example Food Image dataset...\n");
    fprintf("This can take several minutes to download...\n");
    websave(fileFullPath, url);
    fprintf("Download finished...\n");
else
    fprintf("Skipping download, file already exists...\n");
end

% Unzip the file.
%
% Check if the file has already been unzipped by checking for the presence
% of one of the class directories.
exampleFolderFullPath = fullfile(dataDir, "pizza");
if ~exist(exampleFolderFullPath, "dir")
    fprintf("Unzipping file...\n");
    unzip(fileFullPath, dataDir);
    fprintf("Unzipping finished...\n");
else
    fprintf("Skipping unzipping, file already unzipped...\n");
end
fprintf("Done.\n");

end
```

See Also

`squeezenet` | `layerGraph` | `trainingOptions` | `trainNetwork` | `occlusionSensitivity` | `classify` | `activations` | `tsne`

More About

- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Investigate Network Predictions Using Class Activation Mapping” on page 5-123
- “Visualize Features of a Convolutional Neural Network” on page 5-156
- “Visualize Activations of a Convolutional Neural Network” on page 5-141
- “Deep Learning Visualization Methods” on page 5-186

Visualize Activations of a Convolutional Neural Network

This example shows how to feed an image to a convolutional neural network and display the activations of different layers of the network. Examine the activations and discover which features the network learns by comparing areas of activation with the original image. Find out that channels in earlier layers learn simple features like color and edges, while channels in the deeper layers learn complex features like eyes. Identifying features in this way can help you understand what the network has learned.

The example requires Deep Learning Toolbox™ and the Image Processing Toolbox™.

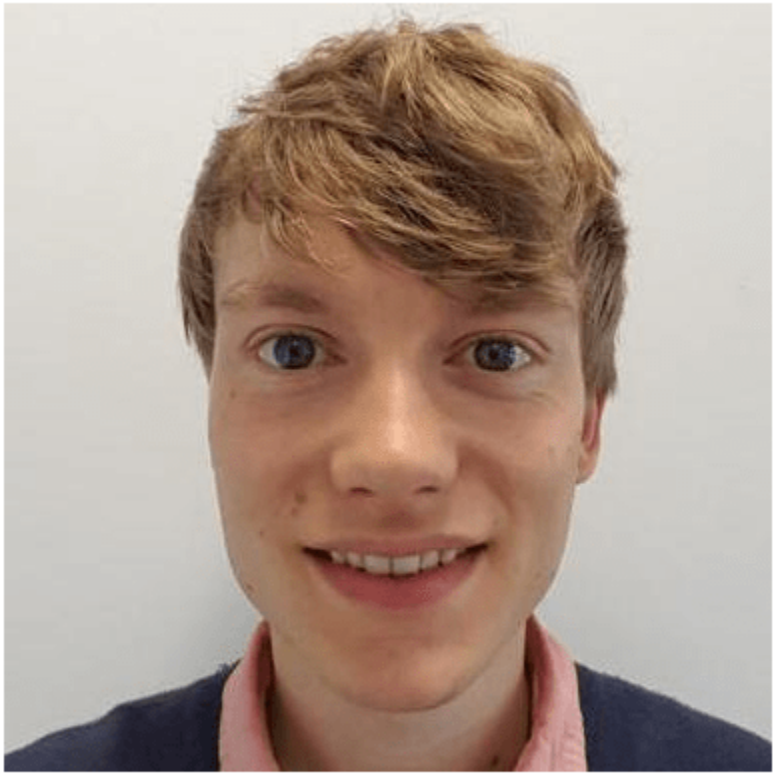
Load Pretrained Network and Data

Load a pretrained SqueezeNet network.

```
net = squeezenet;
```

Read and show an image. Save its size for future use.

```
im = imread('face.jpg');
imshow(im)
```

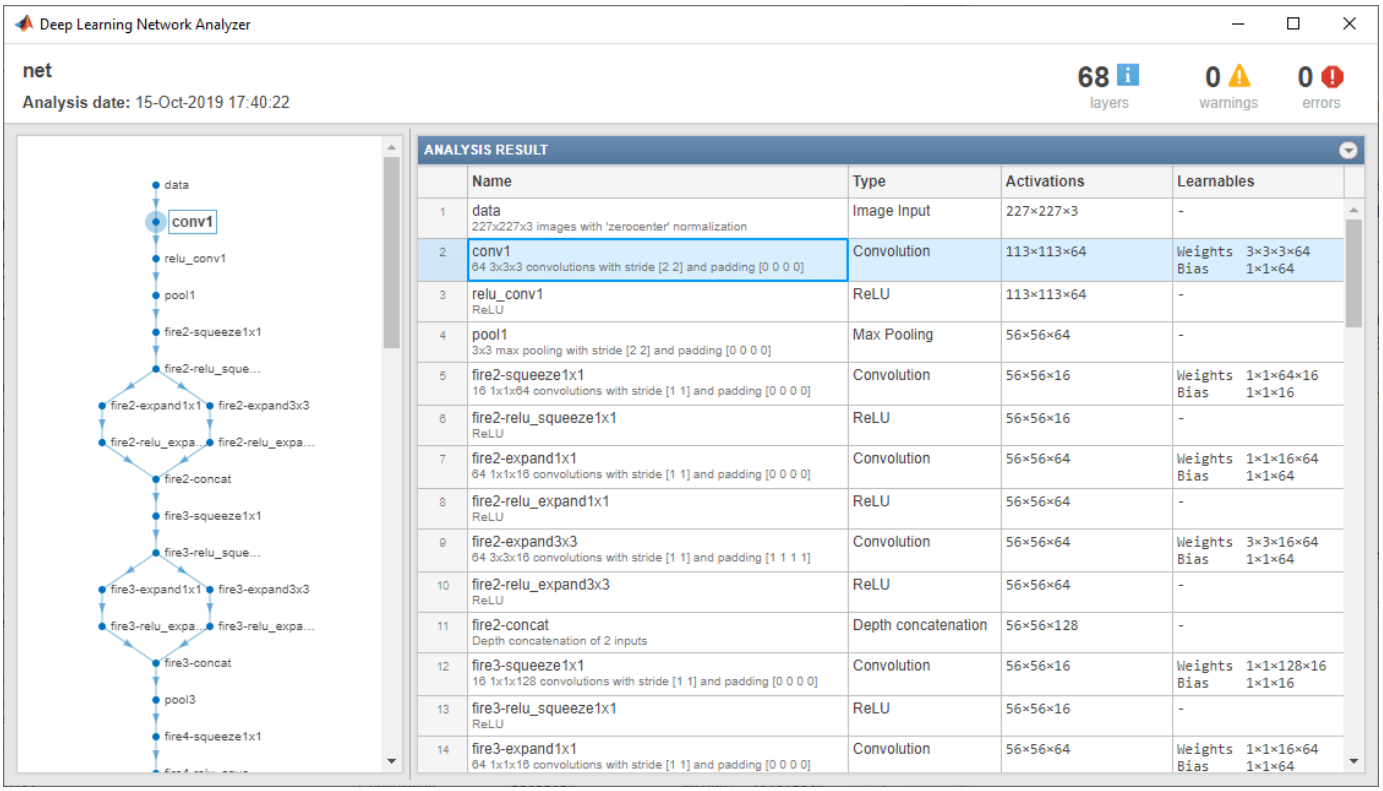


```
imgSize = size(im);
imgSize = imgSize(1:2);
```

View Network Architecture

Analyze the network to see which layers you can look at. The convolutional layers perform convolutions with learnable parameters. The network learns to identify useful features, often with one feature per channel. Observe that the first convolutional layer has 64 channels.

```
analyzeNetwork(net)
```



The Image Input layer specifies the input size. You can resize the image before passing it through the network, but the network also can process larger images. If you feed the network larger images, the activations also become larger. However, since the network is trained on images of size 227-by-227, it is not trained to recognize objects or features larger than that size.

Show Activations of First Convolutional Layer

Investigate features by observing which areas in the convolutional layers activate on an image and comparing with the corresponding areas in the original images. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the `conv1` layer.

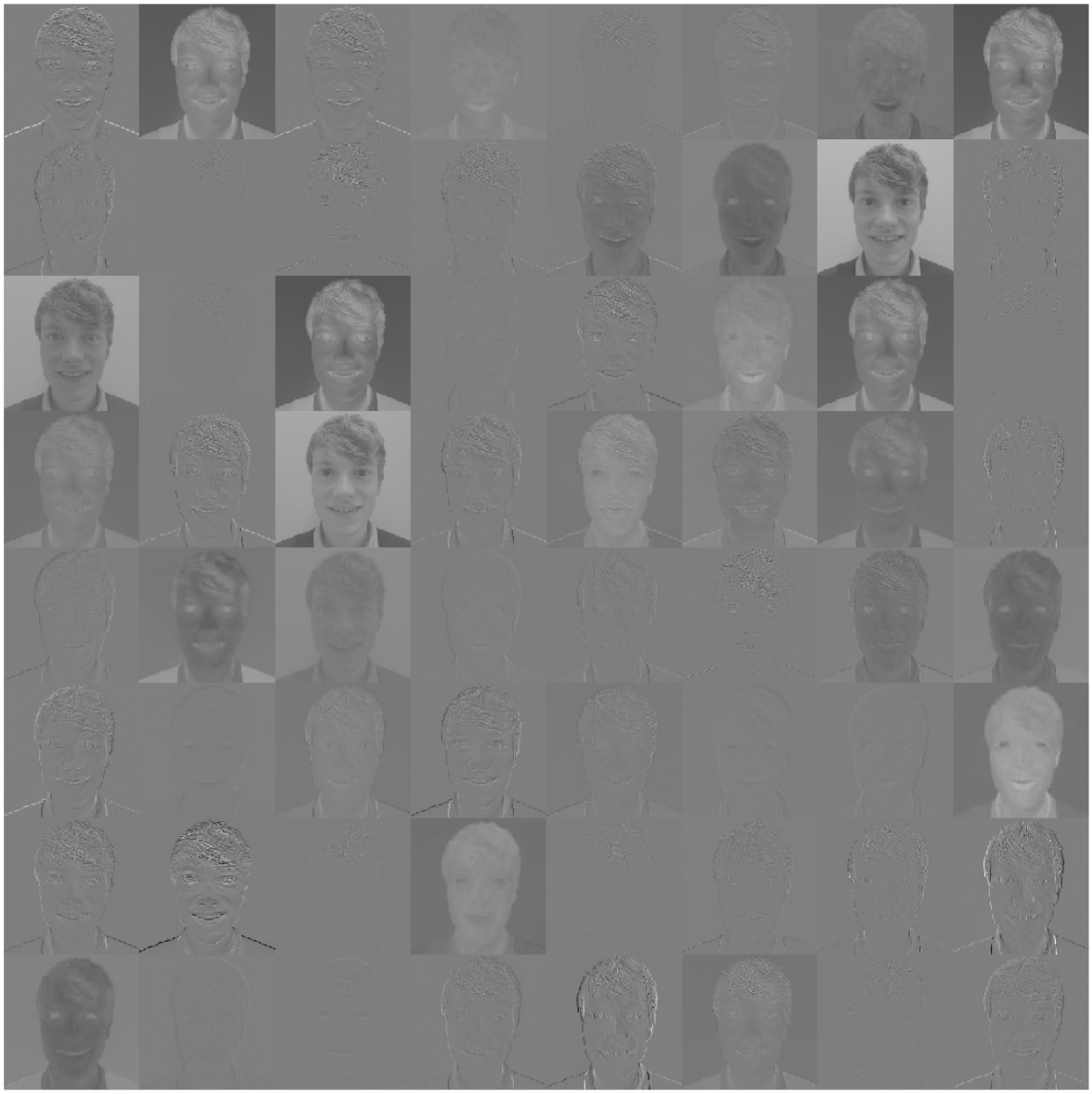
```
act1 = activations(net,im,'conv1');
```

The activations are returned as a 3-D array, with the third dimension indexing the channel on the `conv1` layer. To show these activations using the `imtile` function, reshape the array to 4-D. The third dimension in the input to `imtile` represents the image color. Set the third dimension to have size 1 because the activations do not have color. The fourth dimension indexes the channel.

```
sz = size(act1);
act1 = reshape(act1,[sz(1) sz(2) 1 sz(3)]);
```

Now you can show the activations. Each activation can take any value, so normalize the output using `mat2gray`. All activations are scaled so that the minimum activation is 0 and the maximum is 1. Display the 64 images on an 8-by-8 grid, one for each channel in the layer.

```
I = imtile(mat2gray(act1),'GridSize',[8 8]);
imshow(I)
```



Investigate the Activations in Specific Channels

Each tile in the grid of activations is the output of a channel in the `conv1` layer. White pixels represent strong positive activations and black pixels represent strong negative activations. A channel that is mostly gray does not activate as strongly on the input image. The position of a pixel in the activation of a channel corresponds to the same position in the original image. A white pixel at some location in a channel indicates that the channel is strongly activated at that position.

Resize the activations in channel 22 to have the same size as the original image and display the activations.

```
act1ch22 = act1(:,:,:,22);
act1ch22 = mat2gray(act1ch22);
act1ch22 = imresize(act1ch22,imgSize);

I = imtile({im,act1ch22});
imshow(I)
```

You can see that this channel activates on red pixels, because the whiter pixels in the channel correspond to red areas in the original image.

Find the Strongest Activation Channel

You also can try to find interesting channels by programmatically investigating channels with large activations. Find the channel with the largest activation using the `max` function, resize, and show the activations.

```
[maxValue,maxValueIndex] = max(max(max(act1)));
act1chMax = act1(:,:,:,maxValueIndex);
act1chMax = mat2gray(act1chMax);
act1chMax = imresize(act1chMax,imgSize);
```

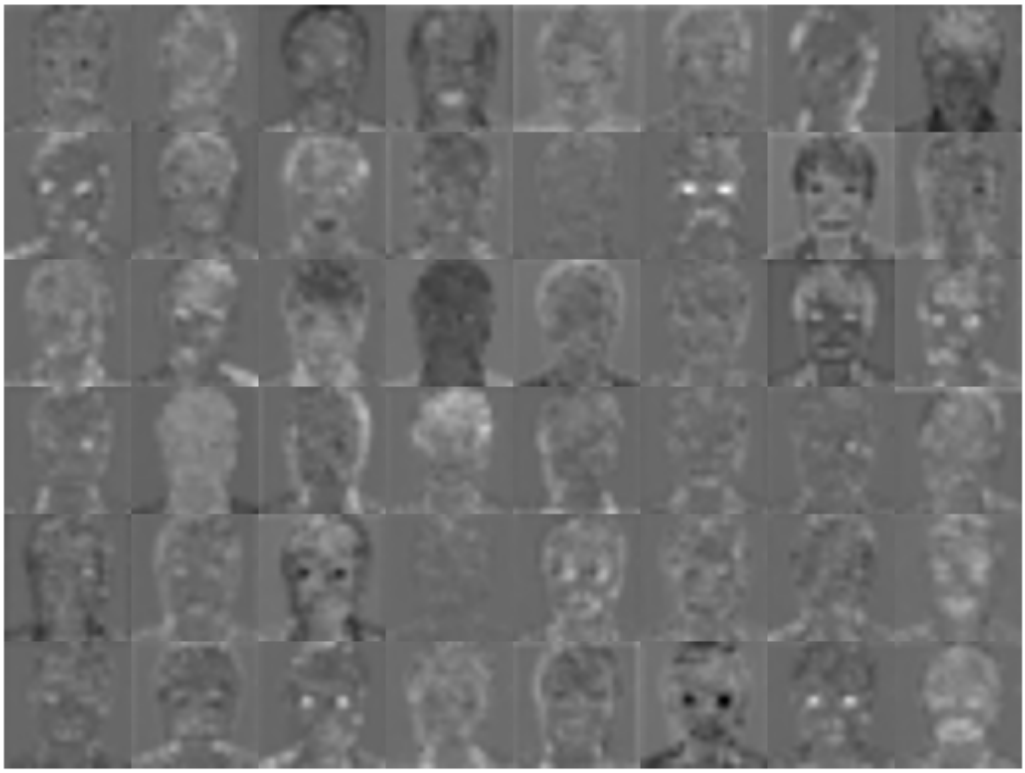
```
I = imtile({im,act1chMax});
imshow(I)
```

Compare to the original image and notice that this channel activates on edges. It activates positively on light left/dark right edges, and negatively on dark left/light right edges.

Investigate a Deeper Layer

Most convolutional neural networks learn to detect features like color and edges in their first convolutional layer. In deeper convolutional layers, the network learns to detect more complicated features. Later layers build up their features by combining features of earlier layers. Investigate the `fire6-squeeze1x1` layer in the same way as the `conv1` layer. Calculate, reshape, and show the activations in a grid.

```
act6 = activations(net,im,'fire6-squeeze1x1');
sz = size(act6);
act6 = reshape(act6,[sz(1) sz(2) 1 sz(3)]);

I = imtile(imresize(mat2gray(act6),[64 64]),'GridSize',[6 8]);
imshow(I)
```



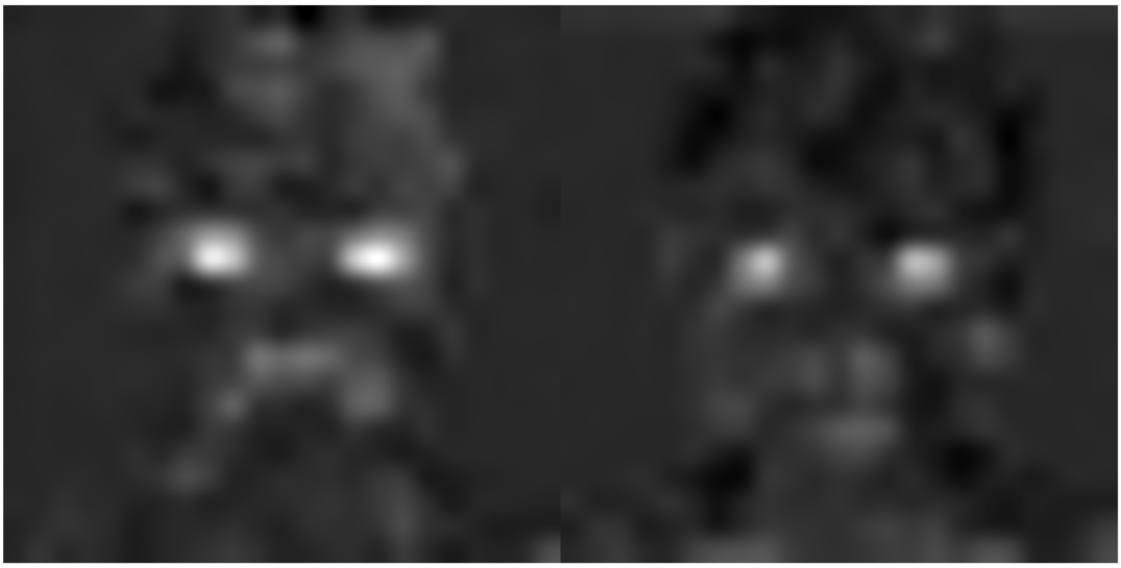
There are too many images to investigate in detail, so focus on some of the more interesting ones. Display the strongest activation in the `fire6-squeeze1x1` layer.

```
[maxValue6,maxValueIndex6] = max(max(max(act6)));
act6chMax = act6(:,:,:,maxValueIndex6);
imshow(imresize(mat2gray(act6chMax),imgSize))
```

In this case, the maximum activation channel is not as interesting for detailed features as some others, and shows strong negative (dark) as well as positive (light) activation. This channel is possibly focusing on faces.

In the grid of all channels, there are channels that might be activating on eyes. Investigate channels 14 and 47 further.

```
I = imtile(imresize(mat2gray(act6(:,:,:,[14 47])),imgSize));
imshow(I)
```



Many of the channels contain areas of activation that are both light and dark. These are positive and negative activations, respectively. However, only the positive activations are used because of the rectified linear unit (ReLU) that follows the `fire6-squeeze1x1` layer. To investigate only positive activations, repeat the analysis to visualize the activations of the `fire6-relu_squeeze1x1` layer.

```
act6relu = activations(net,im,'fire6-relu_squeeze1x1');
sz = size(act6relu);
act6relu = reshape(act6relu,[sz(1) sz(2) 1 sz(3)]);

I = imtile(imresize(mat2gray(act6relu(:,:,:,[14 47])),imgSize));
imshow(I)
```

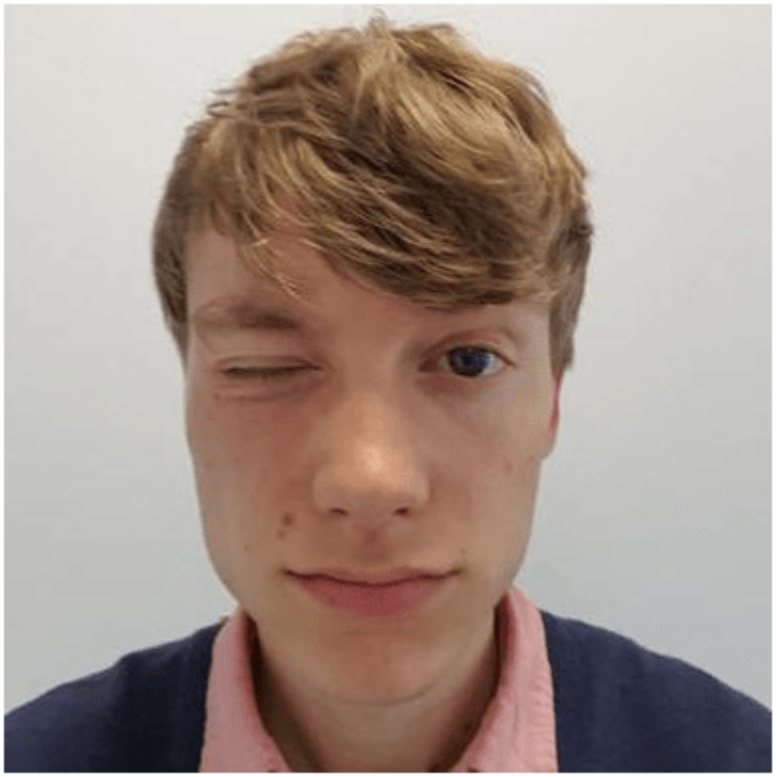
Compared to the activations of the `fire6-squeeze1x1` layer, the activations of the `fire6-relu_squeeze1x1` layer clearly pinpoint areas of the image that have strong facial features.

Test Whether a Channel Recognizes Eyes

Check whether channels 14 and 47 of the `fire6-relu_squeeze1x1` layer activate on eyes. Input a new image with one closed eye to the network and compare the resulting activations with the activations of the original image.

Read and show the image with one closed eye and compute the activations of the `fire6-relu_squeeze1x1` layer.

```
imClosed = imread('face-eye-closed.jpg');
imshow(imClosed)
```



```
act6Closed = activations(net,imClosed,'fire6-relu_squeeze1x1');
sz = size(act6Closed);
act6Closed = reshape(act6Closed,[sz(1),sz(2),1,sz(3)]);
```

Plot the images and activations in one figure.

```
channelsClosed = repmat(imresize(mat2gray(act6Closed(:,:,:,[14 47])),imgSize),[1 1 3]);
channelsOpen = repmat(imresize(mat2gray(act6relu(:,:,:,[14 47])),imgSize),[1 1 3]);
I = imtile(cat(4,im,channelsOpen*255,imClosed,channelsClosed*255));
imshow(I)
title('Input Image, Channel 14, Channel 47');
```

You can see from the activations that both channels 14 and 47 activate on individual eyes, and to some degree also on the area around the mouth.

The network has never been told to learn about eyes, but it has learned that eyes are a useful feature to distinguish between classes of images. Previous machine learning approaches often manually designed features specific to the problem, but these deep convolutional networks can learn useful features for themselves. For example, learning to identify eyes could help the network distinguish between a leopard and a leopard print rug.

See Also

`squeezenet` | `activations` | `deepDreamImage`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Dream Images Using GoogLeNet” on page 5-15
- “Visualize Features of a Convolutional Neural Network” on page 5-156
- “Deep Learning Visualization Methods” on page 5-186

Visualize Activations of LSTM Network

This example shows how to investigate and visualize the features learned by LSTM networks by extracting the activations.

Load pretrained network. `JapaneseVowelsNet` is a pretrained LSTM network trained on the Japanese Vowels dataset as described in [1] and [2]. It was trained on the sequences sorted by sequence length with a mini-batch size of 27.

```
load JapaneseVowelsNet
```

View the network architecture.

```
net.Layers
```

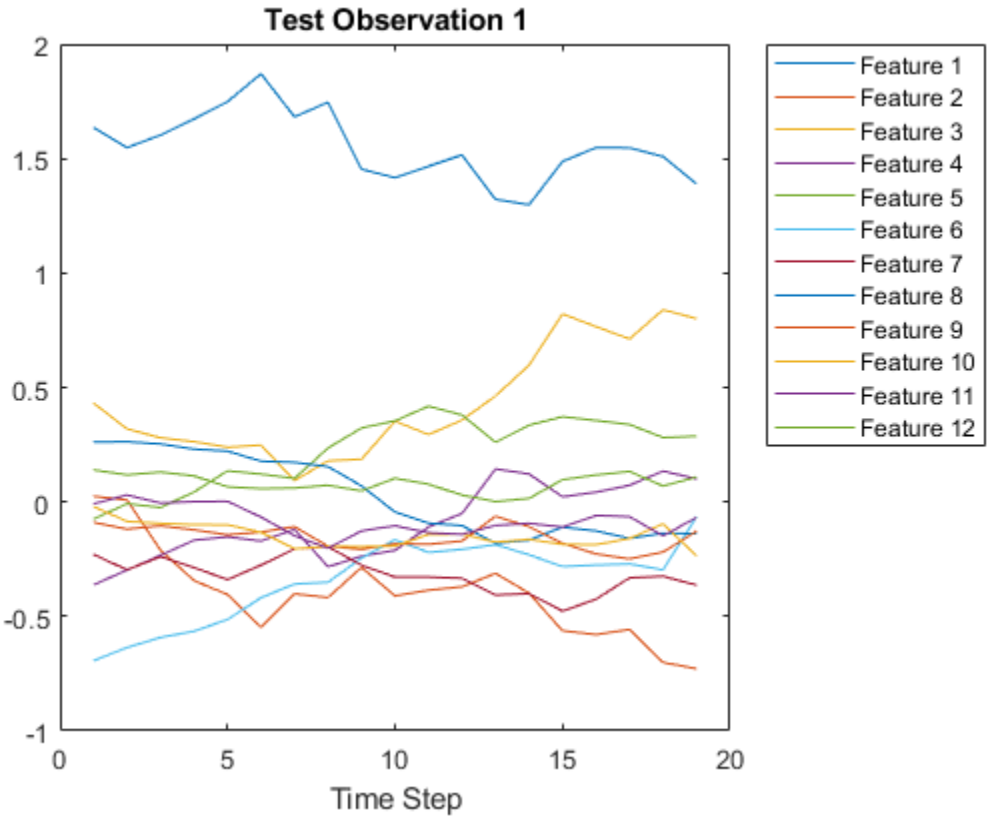
```
ans =
  5x1 Layer array with layers:

     1   'sequenceinput'   Sequence Input          Sequence input with 12 dimensions
     2   'lstm'            LSTM                    LSTM with 100 hidden units
     3   'fc'              Fully Connected         9 fully connected layer
     4   'softmax'         Softmax                 softmax
     5   'classoutput'     Classification Output   crossentropyex with '1' and 8 other classes
```

Load the test data.

```
[XTest,YTest] = japaneseVowelsTestData;
```

Visualize the first time series in a plot. Each line corresponds to a feature.

```
X = XTest{1};

figure
plot(XTest{1}')
xlabel("Time Step")
title("Test Observation 1")
numFeatures = size(XTest{1},1);
legend("Feature " + string(1:numFeatures),'Location','northeastoutside')
```

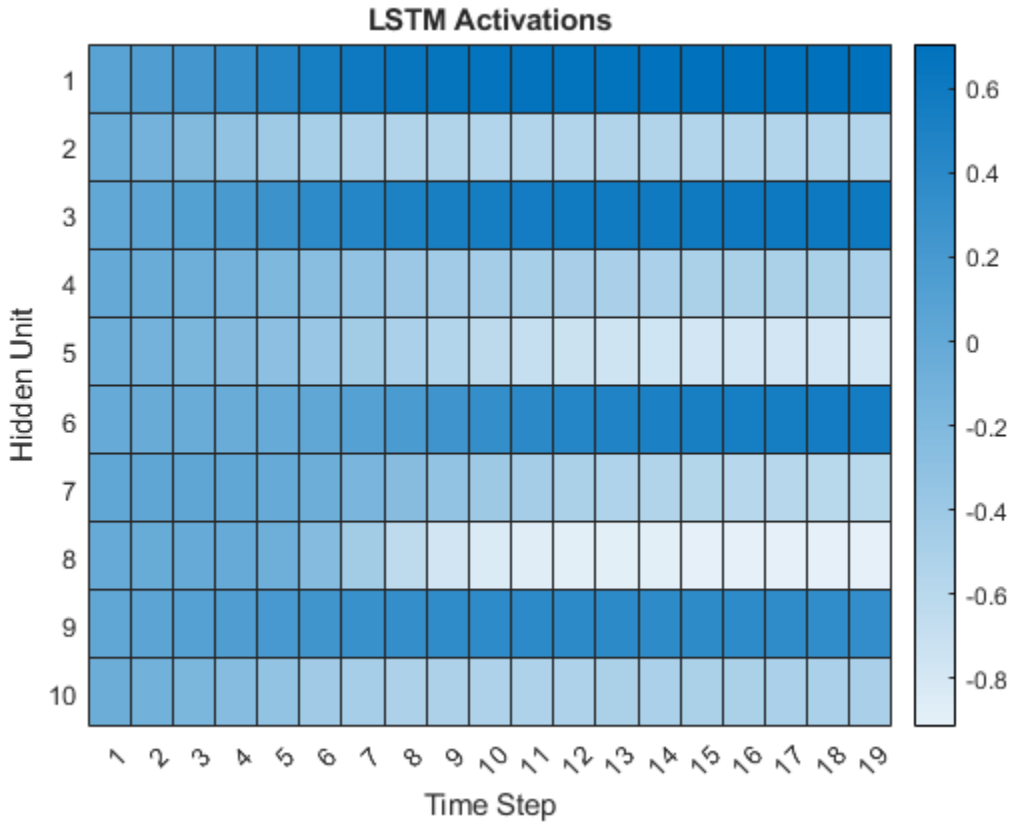


For each time step of the sequences, get the activations output by the LSTM layer (layer 2) for that time step and update the network state.

```
sequenceLength = size(X,2);
idxLayer = 2;
outputSize = net.Layers(idxLayer).NumHiddenUnits;

for i = 1:sequenceLength
    features(:,i) = activations(net,X(:,i),idxLayer);
    [net, YPred(i)] = classifyAndUpdateState(net,X(:,i));
end
```

Visualize the first 10 hidden units using a heatmap.

```
figure
heatmap(features(1:10,:));
xlabel("Time Step")
ylabel("Hidden Unit")
title("LSTM Activations")
```



The heatmap shows how strongly each hidden unit activates and highlights how the activations change over time.

References

[1] M. Kudo, J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pages 1103–1111.

[2] *UCI Machine Learning Repository: Japanese Vowels Dataset*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `bilstmLayer` | `sequenceInputLayer` | `activations`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75

- “Deep Learning in MATLAB” on page 1-2

Visualize Features of a Convolutional Neural Network

This example shows how to visualize the features learned by convolutional neural networks.

Convolutional neural networks use *features* to classify images. The network learns these features itself during the training process. What the network learns during training is sometimes unclear. However, you can use the `deepDreamImage` function to visualize the features learned.

The *convolutional* layers output a 3D activation volume, where slices along the third dimension correspond to a single filter applied to the layer input. The channels output by *fully connected* layers at the end of the network correspond to high-level combinations of the features learned by earlier layers.

You can visualize what the learned features look like by using `deepDreamImage` to generate images that strongly activate a particular channel of the network layers.

The example requires Deep Learning Toolbox™ and Deep Learning Toolbox Model *for GoogLeNet Network* support package.

Load Pretrained Network

Load a pretrained GoogLeNet network.

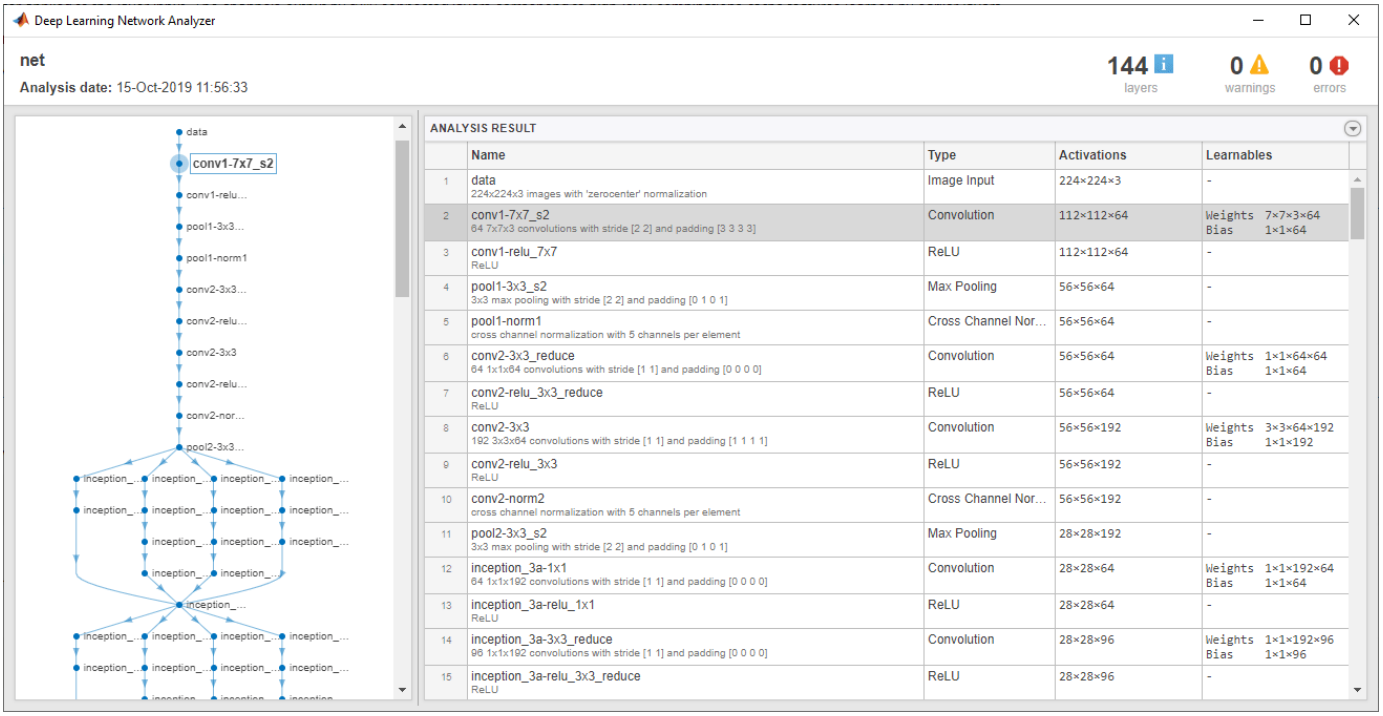
```
net = googlenet;
```

Visualize Early Convolutional Layers

There are multiple convolutional layers in the GoogLeNet network. The convolutional layers towards the beginning of the network have a small receptive field size and learn small, low-level features. The layers towards the end of the network have larger receptive field sizes and learn larger features.

Using `analyzeNetwork`, view the network architecture and locate the convolutional layers.

```
analyzeNetwork(net)
```



Features on Convolutional Layer 1

Set `layer` to be the first convolutional layer. This layer is the second layer in the network and is named `'conv1-7x7_s2'`.

```
layer = 2;
name = net.Layers(layer).Name
```

```
name =
'conv1-7x7_s2'
```

Visualize the first 36 features learned by this layer using `deepDreamImage` by setting `channels` to be the vector of indices `1:36`. Set `'PyramidLevels'` to 1 so that the images are not scaled. To display the images together, you can use `imtile`.

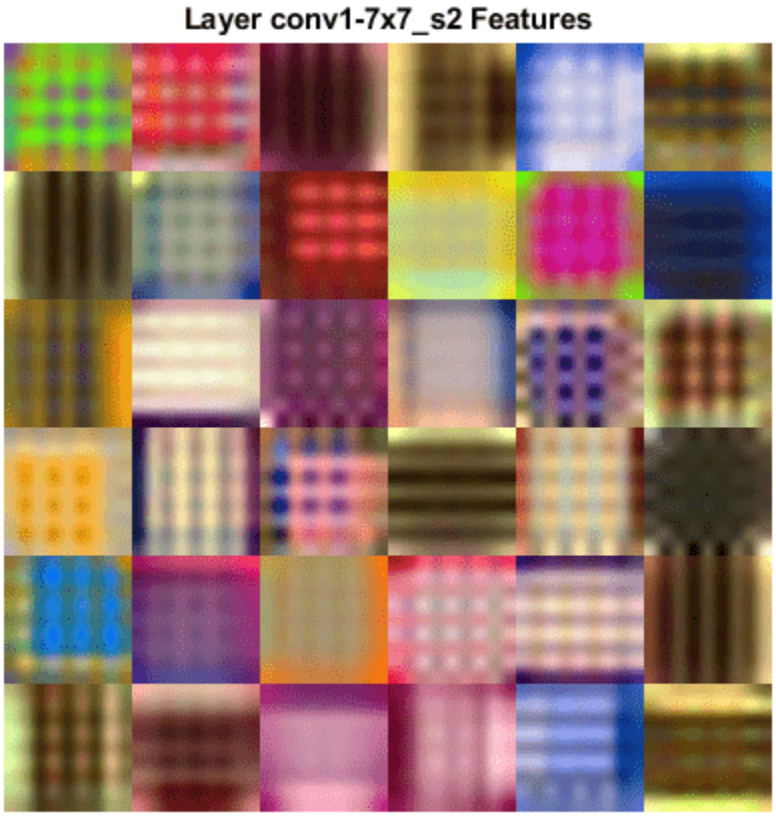
`deepDreamImage` uses a compatible GPU, by default, if available. Otherwise it uses the CPU. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
channels = 1:36;
I = deepDreamImage(net,name,channels, ...
    'PyramidLevels',1);
```

```
|=============================================|
|  Iteration  |  Activation  |  Pyramid Level  |
|             |   Strength   |                 |
|=============================================|
|           1 |         0.26 |               1 |
|           2 |         6.99 |               1 |
|           3 |        14.24 |               1 |
|           4 |        21.49 |               1 |
|           5 |        28.74 |               1 |
```

```
|           6 |        35.99 |               1 |
|           7 |        43.24 |               1 |
|           8 |        50.50 |               1 |
|           9 |        57.75 |               1 |
|          10 |        65.00 |               1 |
|=============================================|
```

```
figure
I = imtile(I,'ThumbnailSize',[64 64]);
imshow(I)
title(['Layer ',name,' Features'],'Interpreter','none')
```



These images mostly contain edges and colors, which indicates that the filters at layer `'conv1-7x7_s2'` are edge detectors and color filters.

Features on Convolutional Layer 2

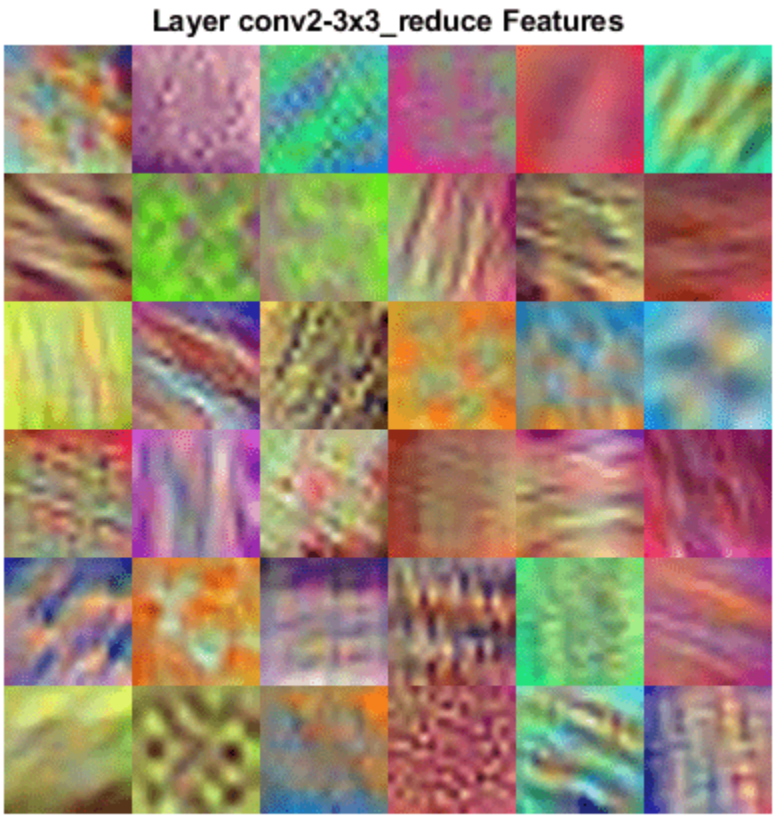
The second convolutional layer is named `'conv2-3x3_reduce'`, which corresponds to layer 6. Visualize the first 36 features learned by this layer by setting `channels` to be the vector of indices `1:36`.

To suppress detailed output on the optimization process, set `'Verbose'` to `'false'` in the call to `deepDreamImage`.

```
layer = 6;
name = net.Layers(layer).Name
```

```
name =
'conv2-3x3_reduce'
```

```
channels = 1:36;
I = deepDreamImage(net,name,channels, ...
    'Verbose',false, ...
    'PyramidLevels',1);
figure
I = imtile(I,'ThumbnailSize',[64 64]);
imshow(I)
name = net.Layers(layer).Name;
title(['Layer ',name,' Features'],'Interpreter','none')
```



Filters for this layer detect more complex patterns than the first convolutional layer.

Visualize Deeper Convolutional Layers

The deeper layers learn high-level combinations of features learned by the earlier layers.

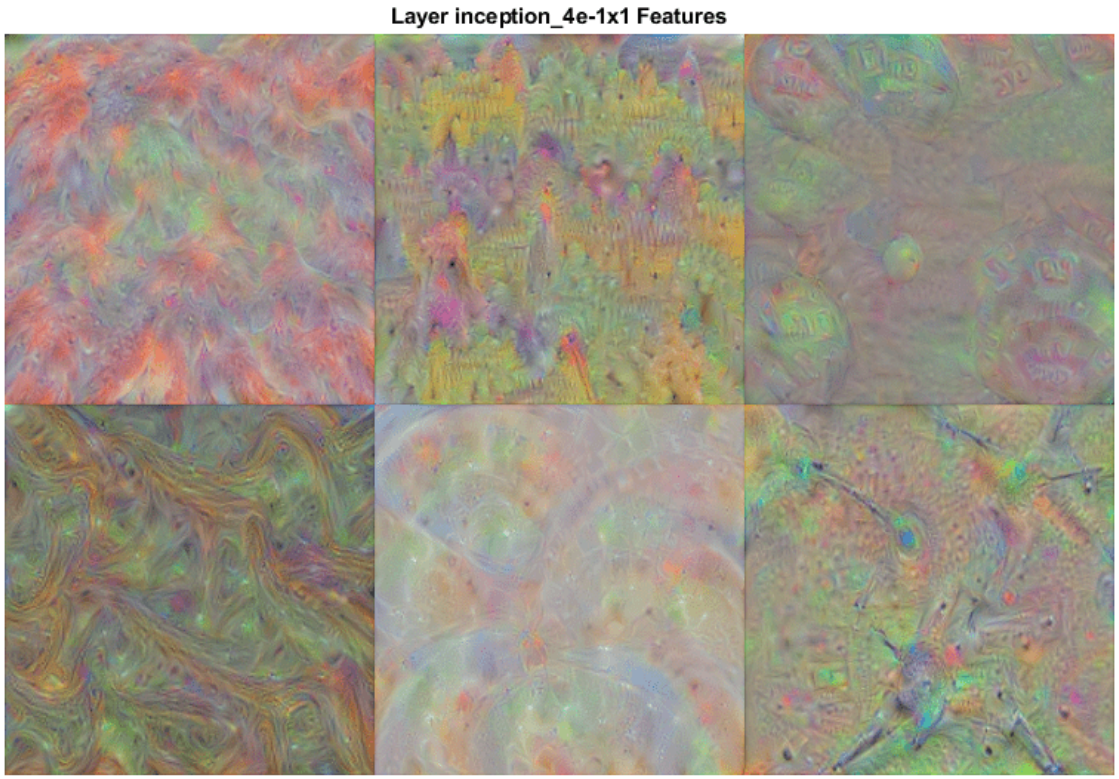
Increasing the number of pyramid levels and iterations per pyramid level can produce more detailed images at the expense of additional computation. You can increase the number of iterations using the

`'NumIterations'` option and increase the number of pyramid levels using the `'PyramidLevels'` option.

```
layer = 97;
name = net.Layers(layer).Name
```

```
name =
'inception_4e-1x1'
```

```
channels = 1:6;
I = deepDreamImage(net,name,channels, ...
    'Verbose',false, ...
    "NumIterations",20, ...
    'PyramidLevels',2);
figure
I = imtile(I,'ThumbnailSize',[250 250]);
imshow(I)
name = net.Layers(layer).Name;
title(['Layer ',name,' Features'],'Interpreter','none')
```



Notice that the layers which are deeper into the network yield more detailed filters which have learned complex patterns and textures.

Visualize Fully Connected Layer

To produce images that resemble each class the most closely, select the fully connected layer, and set `channels` to be the indices of the classes.

Select the fully connected layer (layer 142).

```
layer = 142;
name = net.Layers(layer).Name
```

```
name =
'loss3-classifier'
```

Select the classes you want to visualize by setting `channels` to be the indices of those class names.

```
channels = [114 293 341 484 563 950];
```

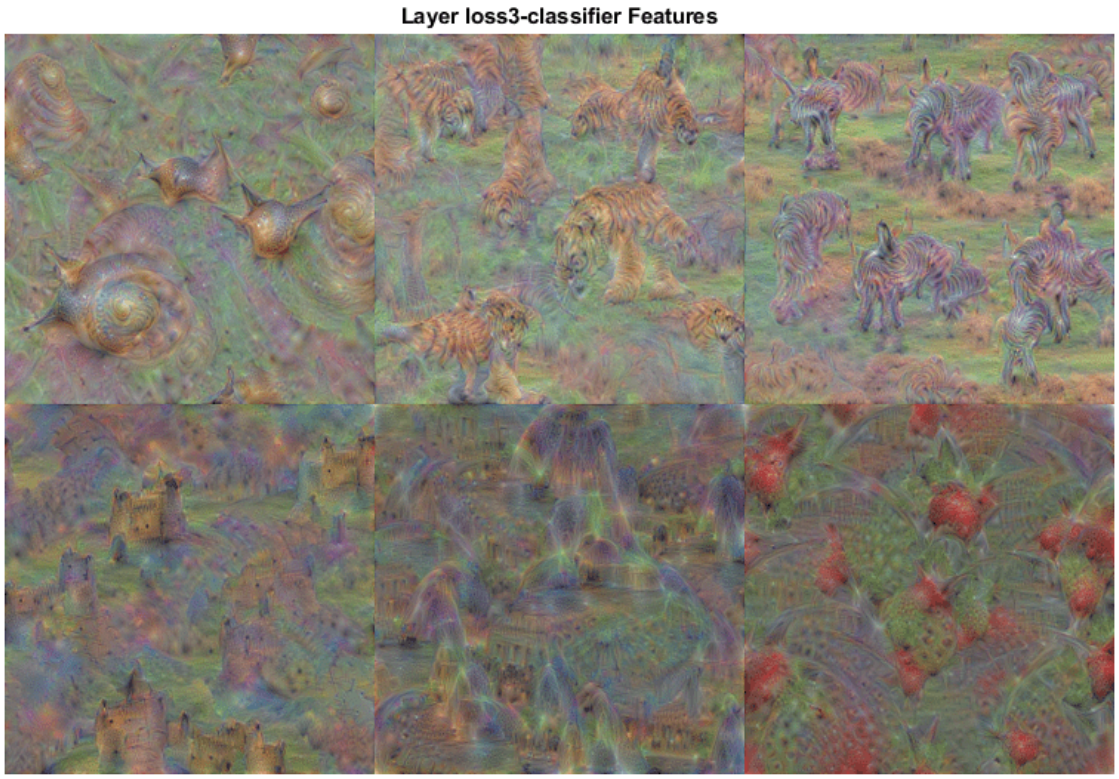
The classes are stored in the `Classes` property of the output layer (the last layer). You can view the names of the selected classes by selecting the entries in `channels`.

```
net.Layers(end).Classes(channels)
```

```
ans = 6×1 categorical
     snail
     tiger
     zebra
     castle
     fountain
     strawberry
```

Generate detailed images that strongly activate these classes. Set `'NumIterations'` to 100 in the call to `deepDreamImage` to produce more detailed images. The images generated from the fully connected layer correspond to the image classes.

```
I = deepDreamImage(net,name,channels, ...
    'Verbose',false, ...
    'NumIterations',100, ...
    'PyramidLevels',2);
figure
I = imtile(I,'ThumbnailSize',[250 250]);
imshow(I)
name = net.Layers(layer).Name;
title(['Layer ',name,' Features'])
```



The images generated strongly activate the selected classes. The image generated for the ‘zebra’ class contain distinct zebra stripes, whilst the image generated for the ‘castle’ class contains turrets and windows.

See Also

`googlenet` | `deepDreamImage` | `occlusionSensitivity` | `gradCAM` | `imageLIME`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Deep Dream Images Using GoogLeNet” on page 5-15
- “Deep Learning Visualization Methods” on page 5-186
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Visualize Activations of a Convolutional Neural Network” on page 5-141

Visualize Image Classifications Using Maximal and Minimal Activating Images

This example shows how to use a data set to find out what activates the channels of a deep neural network. This allows you to understand how a neural network works and diagnose potential issues with a training data set.

This example covers a number of simple visualization techniques, using a GoogLeNet transfer-learned on a food data set. By looking at images that maximally or minimally activate the classifier, you can discover why a neural network gets classifications wrong.

Load and Preprocess the Data

Load the images as an image datastore. This small data set contains a total of 978 observations with 9 classes of food.

Split this data into a training, validation, and test sets to prepare for transfer learning using GoogLeNet. Display a selection of images from the data set.

```
rng default
dataDir = fullfile(tempdir,"Food Dataset");
url = "https://www.mathworks.com/supportfiles/nnet/data/ExampleFoodImageDataset.zip";

if ~exist(dataDir,"dir")
    mkdir(dataDir);
end

downloadExampleFoodImagesData(url,dataDir);
```

```
Downloading MathWorks Example Food Image dataset...
This can take several minutes to download...
Download finished...
Unzipping file...
Unzipping finished...
Done.
```

```
imds = imageDatastore(dataDir, ...
    "IncludeSubfolders",true,"LabelSource","foldernames");
[imdsTrain,imdsValidation,imdsTest] = splitEachLabel(imds,0.6,0.2);

rnd = randperm(numel(imds.Files),9);
for i = 1:numel(rnd)
subplot(3,3,i)
imshow(imread(imds.Files{rnd(i)}))
label = imds.Labels(rnd(i));
title(label,"Interpreter","none")
end
```

Train Network to Classify Food Images

Use the pretrained GoogLeNet network and train it again to classify the 9 types of food. If you don't have the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package installed, then the software provides a download link.

To try a different pretrained network, open this example in MATLAB® and select a different network, such as `squeezenet`, a network that is even faster than `googlenet`. For a list of all available networks, see “Pretrained Deep Neural Networks” on page 1-8.

```
net = googlenet;
```

The first element of the `Layers` property of the network is the image input layer. This layer requires input images of size 224-by-224-by-3, where 3 is the number of color channels.

```
inputSize = net.Layers(1).InputSize;
```

Network Architecture

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, `'loss3-classifier'` and `'output'` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To train a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net);
```

In most networks, the last layer with learnable weights is a fully connected layer. Replace this fully connected layer with a new fully connected layer with the number of outputs equal to the number of classes in the new data set (9, in this example).

```
numClasses = numel(categories(imdsTrain.Labels));

newfclayer = fullyConnectedLayer(numClasses,...
    'Name','new_fc',...
    'WeightLearnRateFactor',10,...
    'BiasLearnRateFactor',10);
lgraph = replaceLayer(lgraph,net.Layers(end-2).Name,newfclayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newclasslayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,net.Layers(end).Name,newclasslayer);
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastore have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, randomly translate them up to 30 pixels, and scale them up to 10% horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange, ...
    'RandXScale',scaleRange, ...
    'RandYScale',scaleRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. Set `InitialLearnRate` to a small value to slow down learning in the transferred layers that are not already frozen. In the previous step, you increased the learning rate factors for the last learnable layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning in the new layers, slower learning in the middle layers, and no learning in the earlier, frozen layers.

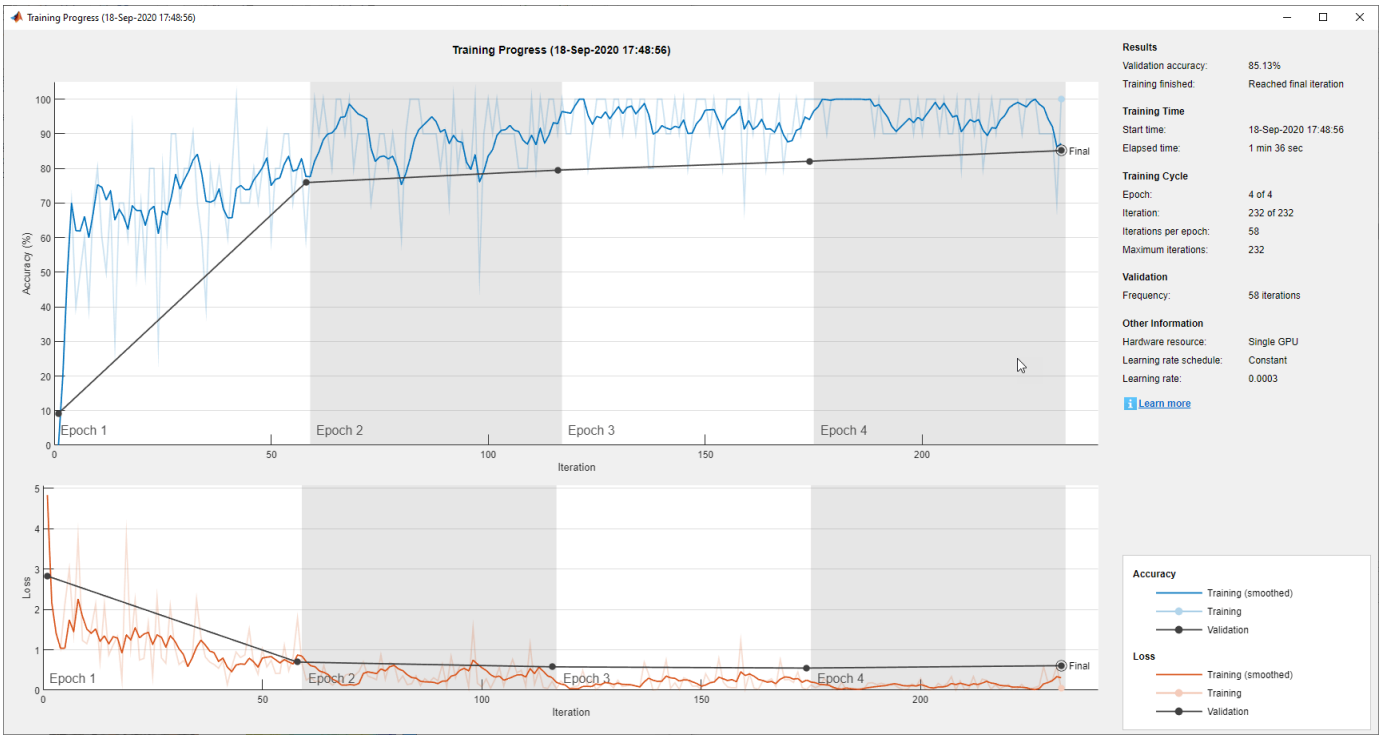
Specify the number of epochs to train for. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. Compute the validation accuracy once per epoch.

```
miniBatchSize = 10;
valFrequency = floor(numel(augimdsTrain.Files)/miniBatchSize);
```

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',4, ...
    'InitialLearnRate',3e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',valFrequency, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network using the training data. By default, `trainNetwork` uses a GPU if one is available. This requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Because this data set is small, the training is fast. If you run this example and train the network yourself, you will get different results and misclassifications caused by the randomness involved in the training process.

```
net = trainNetwork(augimdsTrain,lgraph,options);
```



Classify Test Images

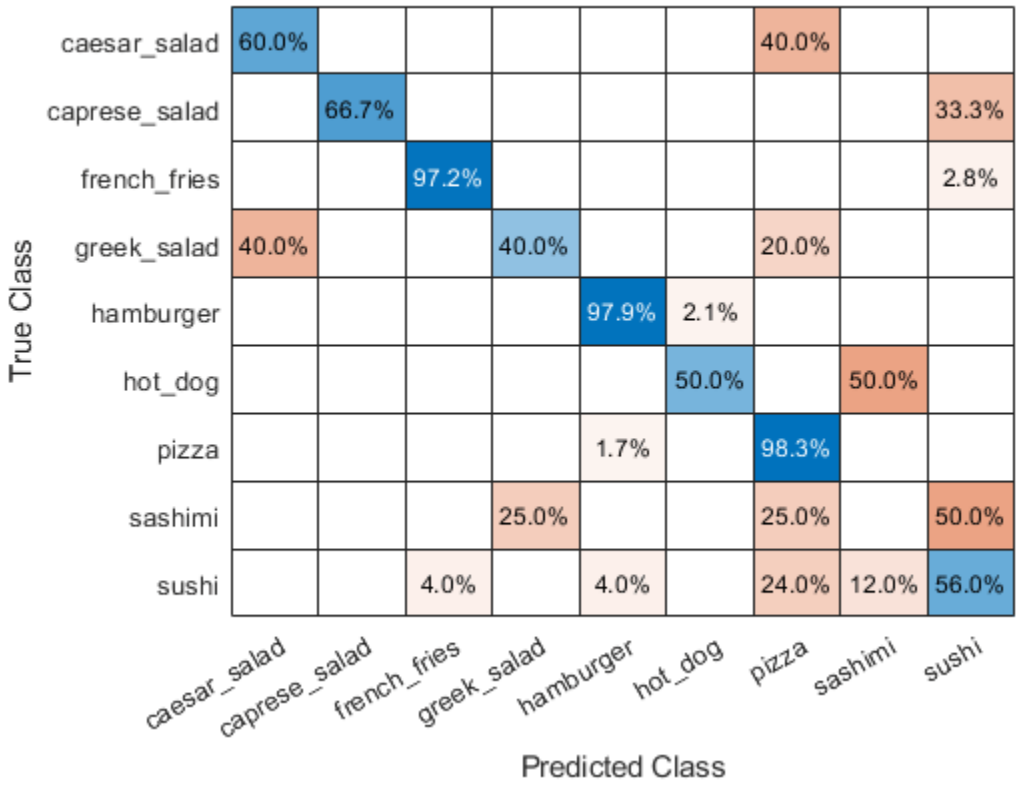
Classify the test images using the fine-tuned network and calculate the classification accuracy.

```
augimdsTest = augmentedImageDatastore(inputSize(1:2),imdsTest);
[predictedClasses,predictedScores] = classify(net,augimdsTest);

accuracy = mean(predictedClasses == imdsTest.Labels)

accuracy = 0.8418
```

Confusion Matrix for the Test Set

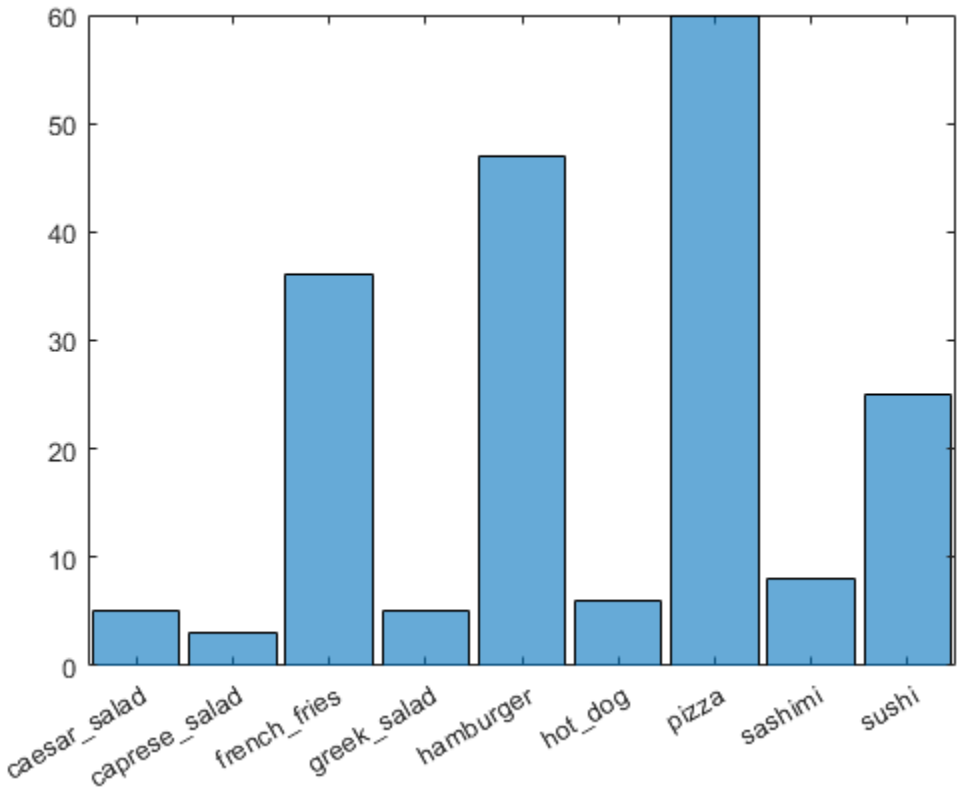
Plot a confusion matrix of the test set predictions. This highlights which particular classes cause most problems for the network.

```
figure;
confusionchart(imdsTest.Labels,predictedClasses,'Normalization',"row-normalized");
```



The confusion matrix shows that the network has poor performance for some classes, such as Greek salad, sashimi, hot dog, and sushi. These classes are underrepresented in the data set which may be impacting network performance. Investigate one of these classes to better understand why the network is struggling.

```
figure();
histogram(imdsValidation.Labels);
ax = gca();
ax.XAxis.TickLabelInterpreter = "none";
```



Investigate Classifications

Investigate network classification for the sushi class.

Sushi Most Like Sushi

First, find which images of sushi most strongly activate the network for the sushi class. This answers the question "Which images does the network think are most sushi-like?".

Plot the maximally-activating images, these are the input images that strongly activate the fully-connected layer's "sushi" neuron. This figure shows the top 4 images, in a descending class score.

```
chosenClass = "sushi";
classIdx = find(net.Layers(end).Classes == chosenClass);

numImgsToShow = 4;

[sortedScores,imgIdx] = findMaxActivatingImages(imdsTest,chosenClass,predictedScores,numImgsToSho

figure
plotImages(imdsTest,imgIdx,sortedScores,predictedClasses,numImgsToShow)
```

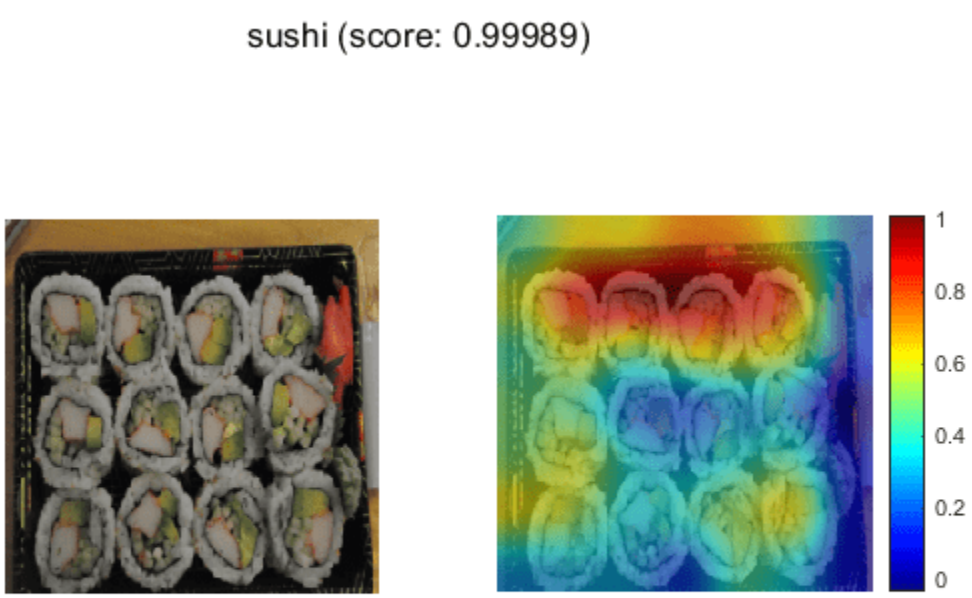
Visualize Cues for the Sushi Class

Is the network looking at the right thing for sushi? The maximally-activating images of the sushi class for the network all look similar to each other - a lot of round shapes clustered closely together.

The network is doing well at classifying those kinds of sushi. However, to verify that this is true and to better understand why the network makes its decisions, use a visualization technique like Grad-CAM. For more information on using Grad-CAM, see “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21.

Read the first resized image from the augmented image datastore, then plot the Grad-CAM visualization using `gradCAM`.

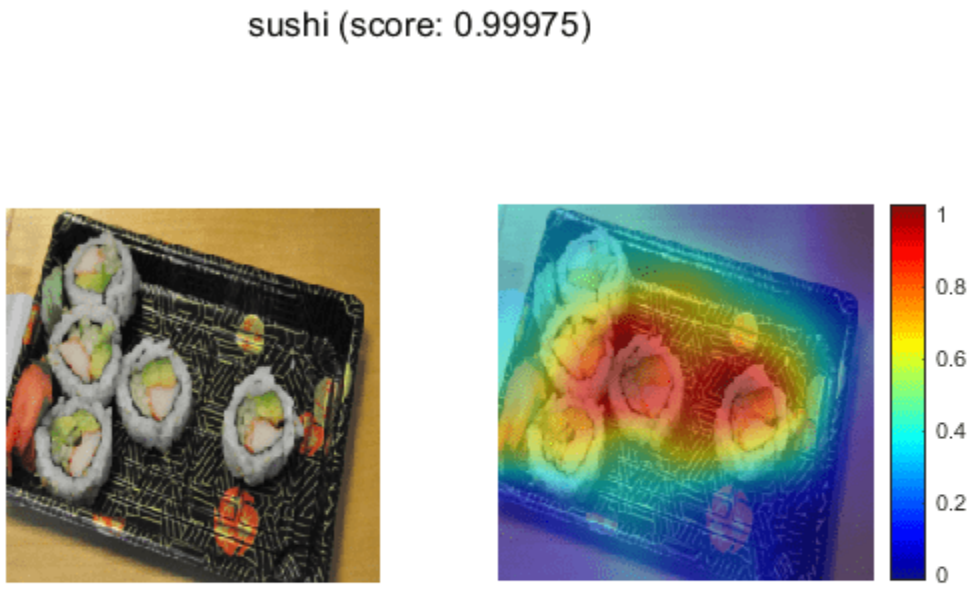
```
imageNumber = 1;

observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```



The Grad-CAM map confirms that the network is focussing on the sushi in the image. However you can also see that the network is looking at parts of the plate and the table.

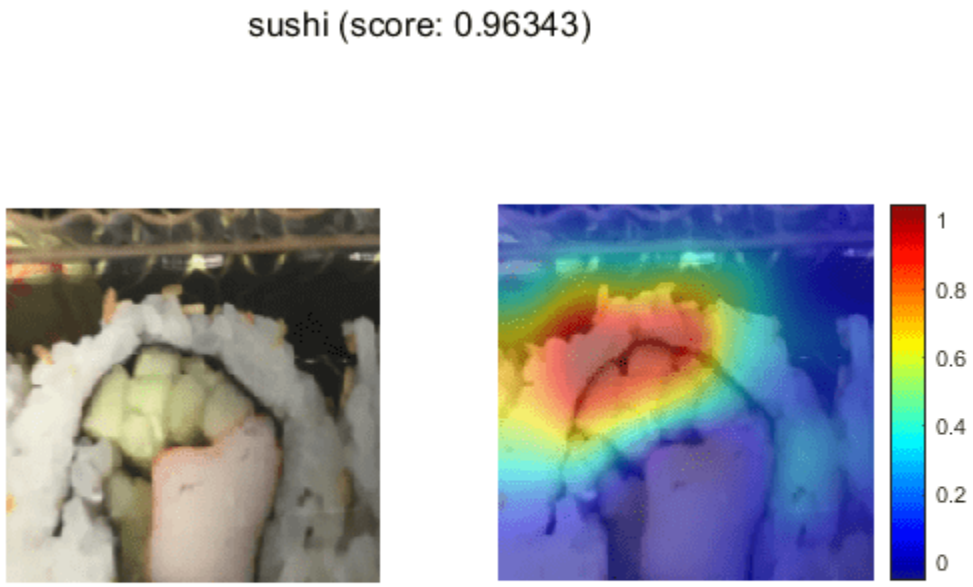
The second image has a cluster of sushi on the left and a lone sushi on the right. To see what the network focuses on, read the second image and plot the Grad-CAM.

```
imageNumber = 2;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
plotGradCAM(img,gradcamMap,alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```



The network classifies this image as sushi because it sees a group of sushi. However, is it able to classify one sushi on its own? Test this by looking at a picture of just one sushi.

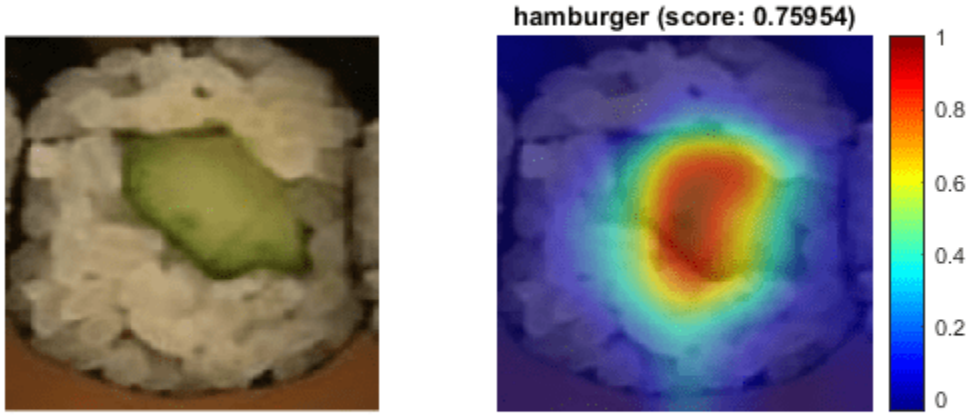
```
img = imread(strcat(tempdir,"Food Dataset/sushi/sushi_18.jpg"));
img = imresize(img,net.Layers(1).InputSize(1:2),"Method","bilinear","AntiAliasing",true);

[label,score] = classify(net,img);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```



The network is able to classify this lone sushi correctly. However, the GradCAM shows that the network is focussing on the top of the sushi and the cluster of cucumber, rather than the whole piece together.

Run the Grad-CAM visualization technique on a lone sushi that does not contain any stacked small pieces of ingredients.

```
img = imread("crop__sushi34-copy.jpg");
img = imresize(img,net.Layers(1).InputSize(1:2),"Method","bilinear","AntiAliasing",true);

[label,score] = classify(net,img);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
title(string(label)+" (score: "+ max(score)+")")
```



In this case, the visualization technique highlights why the network performs poorly. It incorrectly classifies the image of the sushi as a hamburger.

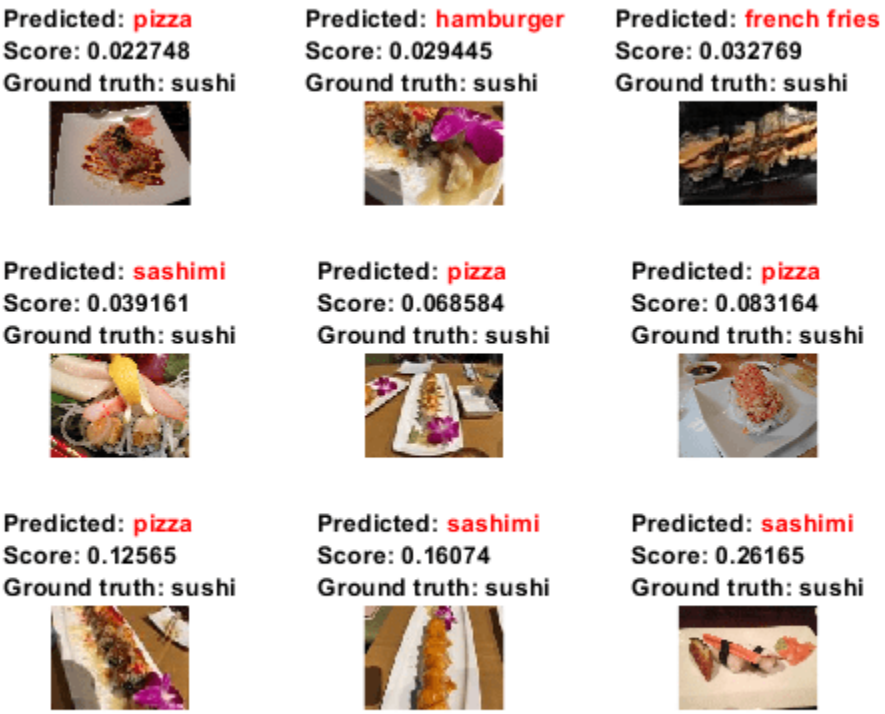
To solve this issue, you must feed the network with more images of lone sushi during the training process.

Sushi Least Like Sushi

Now find which images of sushi activate the network for the sushi class the least. This answers the question "Which images does the network think are less sushi-like?".

This is useful because it finds the images on which the network performs badly, and it provides some insight into its decision.

```
chosenClass = "sushi";
numImgsToShow = 9;

[sortedScores,imgIdx] = findMinActivatingImages(imdsTest,chosenClass,predictedScores,numImgsToSho

figure
plotImages(imdsTest,imgIdx,sortedScores,predictedClasses,numImgsToShow)
```

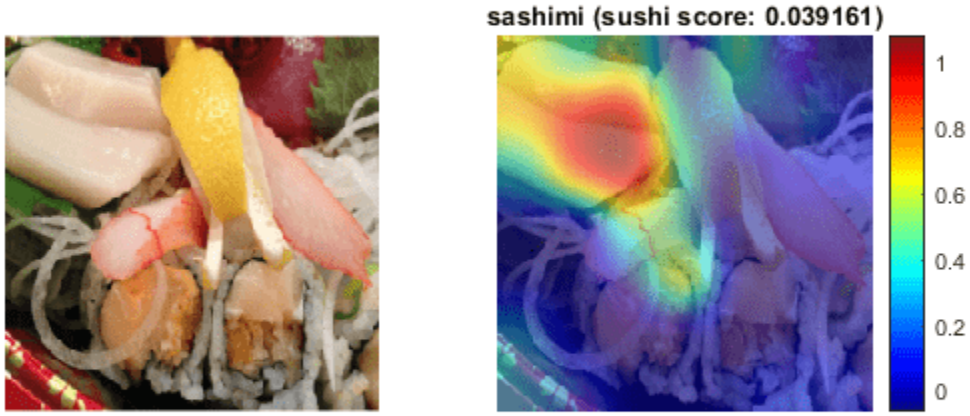


Investigate Sushi Misclassified as Sashimi

Why is the network classifying sushi as sashimi? The network classifies 3 out of the 9 images as sashimi. Some of these images, for example images 4 and 9, actually contain sashimi, which means the network isn't actually misclassifying them. These images are mislabeled.

To see what the network is focusing on, run the Grad-CAM technique on one of these images.

```
imageNumber = 4;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
title(string(label)+" (sushi score: "+ max(score)+")")
```



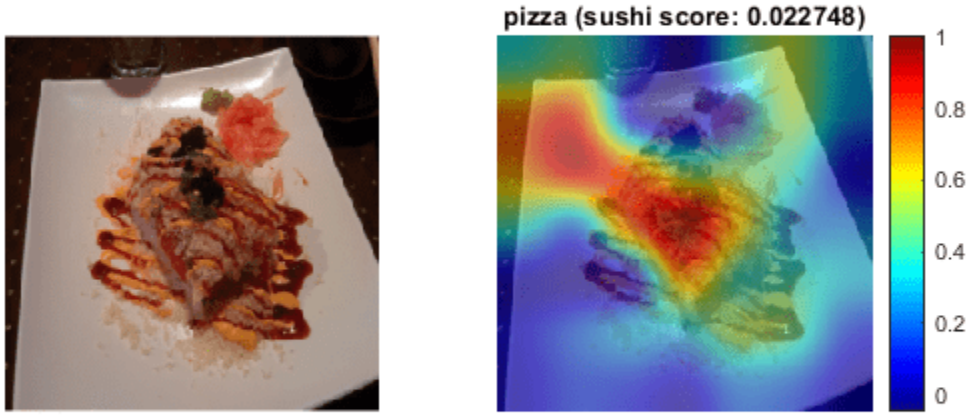
As expected, the network focuses on the sashimi instead of the sushi.

Investigate Sushi Misclassified as Pizza

Why is the network classifying sushi as pizza? The network classifies four of the images as pizza instead of sushi. Consider image 1, this image has a colourful topping which may be confusing the network.

To see which part of the image the network is looking at, run the Grad-CAM technique on one of these images.

```
imageNumber = 1;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
title(string(label)+" (sushi score: "+ max(score)+")")
```



The network strongly focuses on the toppings. To help the network distinguish pizza from sushi with toppings, add more training images of sushi with toppings. The network also focuses on the plate. This may be as the network has learned to associate certain foods with certain types of plates, as also highlighted when looking at the sushi images. To improve the network's performance, train using more examples of food on different types of plates.

Investigate Sushi Misclassified as a Hamburger

Why is the network classifying sushi as a hamburger? To see what the network is focusing on, run the Grad-CAM technique on the misclassified image.

```
imageNumber = 2;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
title(string(label)+" (sushi score: "+ max(score)+")")
```

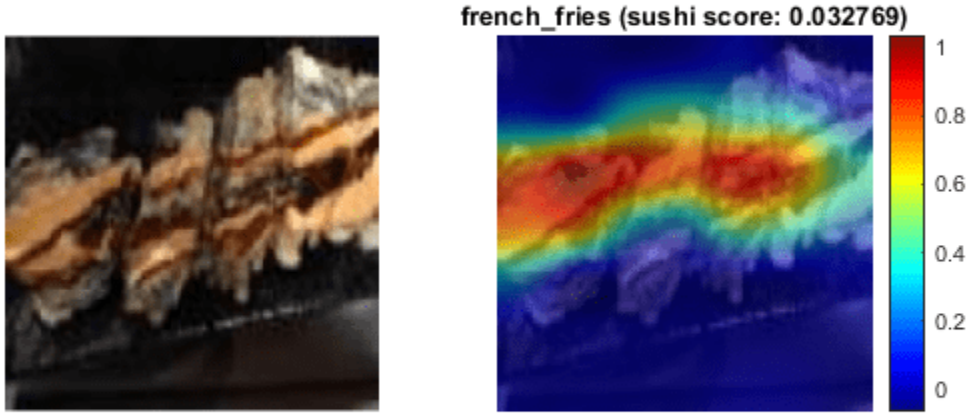
The network is focussing on the flower in the image. The colourful purple flower and brown stalk has confused the network into identifying this image as a hamburger.

Investigate Sushi Misclassified as French Fries

Why is the network classifying sushi as French fries? The network classifies the 3rd image as French fries instead of sushi. This specific sushi has a yellow topping and the network might associate this color with French fries.

Run Grad-CAM on this image.

```
imageNumber = 3;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = gradCAM(net,img,label);

figure
alpha = 0.5;
plotGradCAM(img,gradcamMap,alpha);
title(string(label)+" (sushi score: "+ max(score)+")","Interpreter","none")
```



The networks focuses on the yellow sushi classifying it as French fries. As with the hamburger, the unusual colour has caused the network to misclassify the sushi.

To help the network in this specific case, train it with more images of yellow foods that are not French fries.

Conclusions

Investigating the datapoints that give rise to large or small class scores, and datapoints that the network classifies confidently but incorrectly, is a simple technique which can provide useful insight into how a trained network is functioning. In the case of the food data set, this example highlighted that:

- The test data contains several images with incorrect true labels, such as the "sashimi" which is actually "sushi". The data also contains incomplete labels, such as images which contain both sushi and sashimi.
- The network considers a "sushi" to be "multiple, clustered, round-shaped things". However, it must be able to distinguish a lone sushi as well.
- Any sushi or sashimi with toppings or unusual colors confuses the network. To resolve this problem, the data must have a wider variety of sushi and sashimi.
- To improve performance the network needs to see more images from the underrepresented classes.

Helper Functions

```
function downloadExampleFoodImagesData(url,dataDir)
% Download the Example Food Image data set, containing 978 images of
% different types of food split into 9 classes.

% Copyright 2019 The MathWorks, Inc.

fileName = "ExampleFoodImageDataset.zip";
fileFullPath = fullfile(dataDir,fileName);

% Download the .zip file into a temporary directory.
if ~exist(fileFullPath,"file")
    fprintf("Downloading MathWorks Example Food Image dataset...\n");
    fprintf("This can take several minutes to download...\n");
    websave(fileFullPath,url);
    fprintf("Download finished...\n");
else
    fprintf("Skipping download, file already exists...\n");
end

% Unzip the file.
%
% Check if the file has already been unzipped by checking for the presence
% of one of the class directories.
exampleFolderFullPath = fullfile(dataDir,"pizza");
if ~exist(exampleFolderFullPath,"dir")
    fprintf("Unzipping file...\n");
    unzip(fileFullPath,dataDir);
    fprintf("Unzipping finished...\n");
else
    fprintf("Skipping unzipping, file already unzipped...\n");
end
fprintf("Done.\n");

end

function [sortedScores,imgIdx] = findMaxActivatingImages(imds,className,predictedScores,numImgsTo
% Find the predicted scores of the chosen class on all the images of the chosen class
% (e.g. predicted scores for sushi on all the images of sushi)
[scoresForChosenClass,imgsOfClassIdxs] = findScoresForChosenClass(imds,className,predictedScores)

% Sort the scores in descending order
[sortedScores,idx] = sort(scoresForChosenClass,'descend');

% Return the indices of only the first few
imgIdx = imgsOfClassIdxs(idx(1:numImgsToShow));

end

function [sortedScores,imgIdx] = findMinActivatingImages(imds,className,predictedScores,numImgsTo
% Find the predicted scores of the chosen class on all the images of the chosen class
% (e.g. predicted scores for sushi on all the images of sushi)
[scoresForChosenClass,imgsOfClassIdxs] = findScoresForChosenClass(imds,className,predictedScores)

% Sort the scores in ascending order
[sortedScores,idx] = sort(scoresForChosenClass,'ascend');
```

```
% Return the indices of only the first few
imgIdx = imgsOfClassIdxs(idx(1:numImgsToShow));

end

function [scoresForChosenClass,imgsOfClassIdxs] = findScoresForChosenClass(imds,className,predict
% Find the index of className (e.g. "sushi" is the 9th class)
uniqueClasses = unique(imds.Labels);
chosenClassIdx = string(uniqueClasses) == className;

% Find the indices in imageDatastore that are images of label "className"
% (e.g. find all images of class sushi)
imgsOfClassIdxs = find(imds.Labels == className);

% Find the predicted scores of the chosen class on all the images of the
% chosen class
% (e.g. predicted scores for sushi on all the images of sushi)
scoresForChosenClass = predictedScores(imgsOfClassIdxs,chosenClassIdx);
end

function plotImages(imds,imgIdx,sortedScores,predictedClasses,numImgsToShow)

for i=1:numImgsToShow
    score = sortedScores(i);
    sortedImgIdx = imgIdx(i);
    predClass = predictedClasses(sortedImgIdx);
    correctClass = imds.Labels(sortedImgIdx);

    imgPath = imds.Files{sortedImgIdx};

    if predClass == correctClass
        color = "\color{green}";
    else
        color = "\color{red}";
    end

    predClassTitle = strrep(string(predClass),'_',' ');
    correctClassTitle = strrep(string(correctClass),'_',' ');

    subplot(3,ceil(numImgsToShow./3),i)
    imshow(imread(imgPath));
    title("Predicted: " + color + predClassTitle + "\newline\color{black}Score: " + num2str(score
end

end

function plotGradCAM(img,gradcamMap,alpha)

subplot(1,2,1)
imshow(img);

h = subplot(1,2,2);
imshow(img)
hold on;
imagesc(gradcamMap,'AlphaData',alpha);

originalSize2 = get(h,'Position');
```

```
colormap jet
colorbar

set(h,'Position',originalSize2);
hold off;
end
```

See Also

`googlenet` | `imageDatastore` | `augmentedImageDatastore` | `confusionchart` | `dlnetwork` | `classify` | `occlusionSensitivity` | `gradCAM` | `imageLIME`

Related Examples

- “Deep Learning Visualization Methods” on page 5-186
- “Visualize Activations of a Convolutional Neural Network” on page 5-141
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Understand Network Predictions Using LIME” on page 5-42
- “Deep Learning Visualization Methods” on page 5-186

More About

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8

Monitor GAN Training Progress and Identify Common Failure Modes

Training GANs can be a challenging task. This is because the generator and the discriminator networks compete against each other during the training. In fact, if one network learns too quickly, then the other network may fail to learn. This can often result in the network not being able to converge. To diagnose issues and monitor on a scale from 0 to 1 how well the generator and discriminator achieve their respective goals you can plot their scores. For an example showing how to train a GAN and plot the generator and discriminator scores, see “Train Generative Adversarial Network (GAN)” on page 3-76.

The discriminator learns to classify input images as "real" or "generated". The output of the discriminator corresponds to a probability $\widehat{Y}$ that the input images belong to the class "real".

The generator score is the mean of the probabilities corresponding to the discriminator output for the generated images:

$$\text{scoreGenerator} = \text{mean}(\widehat{Y}_{\text{Generated}}),$$

where $\widehat{Y}_{\text{Generated}}$ contains the probabilities for the generated images.

Given that $1 - \widehat{Y}$ is the probability of an image belonging to the class "generated", the discriminator score is the mean of the probabilities of the input images belonging to the correct class:

$$\text{scoreDiscriminator} = \frac{1}{2}\text{mean}(\widehat{Y}_{\text{Real}}) + \frac{1}{2}\text{mean}(1 - \widehat{Y}_{\text{Generated}}),$$

where $\widehat{Y}_{\text{Real}}$ contains the discriminator output probabilities for the real images and the numbers of real and generated images passed to the discriminator are equal.

In the ideal case, both scores would be 0.5. This is because the discriminator cannot tell the difference between real and fake images. However, in practice this scenario is not the only case in which you can achieve a successful GAN.

To monitor the training progress you can visually inspect the images over time and check if they are improving. If the images are not improving, then you can use the score plot to help you diagnose some problems. In some cases, the score plot can tell you there is no point continuing training, and you should stop, because a failure mode has occurred that training cannot recover from. The following sections tell you what to look for in the score plot and in the generated images to diagnose some common failure modes (convergence failure and mode collapse), and suggests possible actions you can take to improve the training.

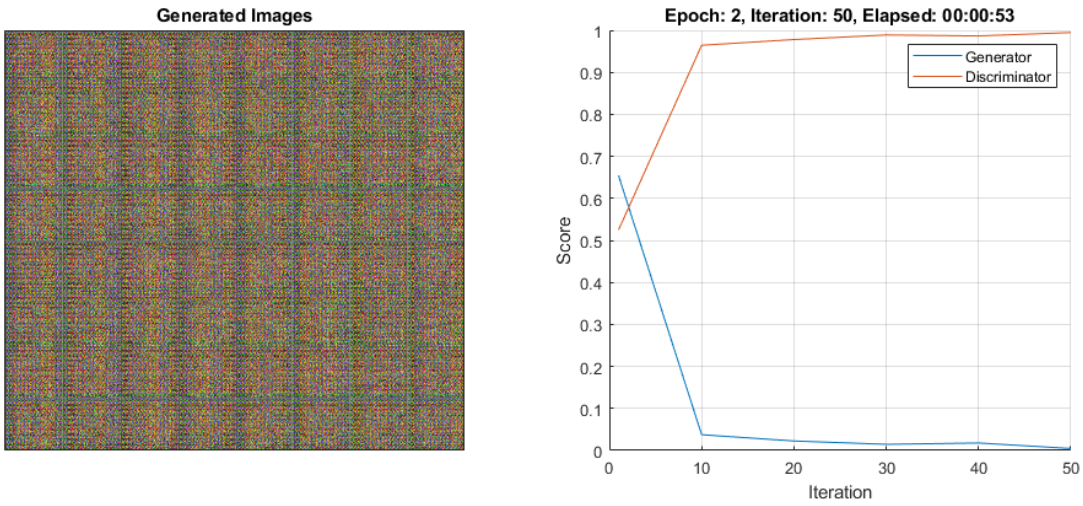
Convergence Failure

Convergence failure happens when the generator and discriminator do not reach a balance during training.

Discriminator Dominates

This scenario happens when the generator score reaches zero or near zero and the discriminator score reaches one or near one.

This plot shows an example of the discriminator overpowering the generator. Notice that the generator score approaches zero and does not recover. In this case, the discriminator classifies most of the images correctly. In turn, the generator cannot produce any images that fool the discriminator and thus fails to learn.



If the score does not recover from these values for many iterations, then it is better to stop the training. If this happens, then try balancing the performance of generator and the discriminator by:

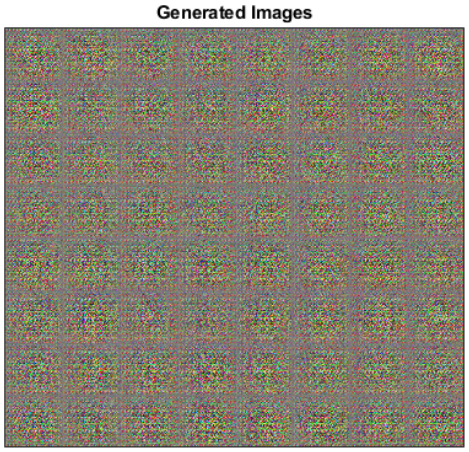
- Impairing the discriminator by randomly giving false labels to real images (one-sided label flipping)
- Impairing the discriminator by adding dropout layers
- Improving the generator's ability to create more features by increasing the number of filters in its convolution layers
- Impairing the discriminator by reducing its number of filters

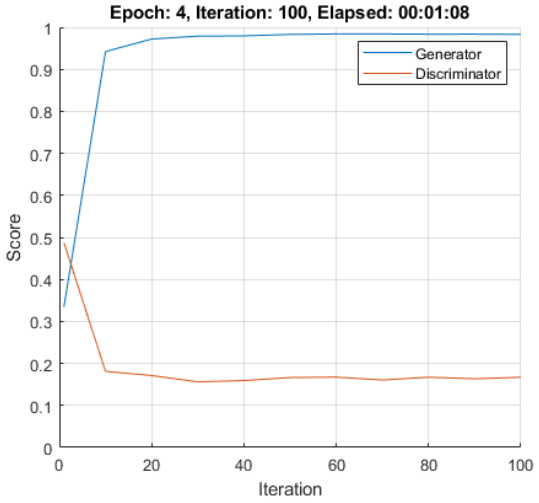
For an example showing how to flip the labels of the real images, see “Train Generative Adversarial Network (GAN)” on page 3-76.

Generator Dominates

This scenario happens when the generator score reaches one or near one.

This plot shows an example of the generator overpowering the discriminator. Notice that the generator score goes to one for a many iterations. In this case, the generator learns how to fool the discriminator almost always. When this happens very early in the training process, the generator is likely to learn a very simple feature representation which fools the discriminator easily. This means that the generated images can be very poor, despite having high scores. Note that in this example, the score of the discriminator does not go very close to zero because it is still able to classify correctly some real images.





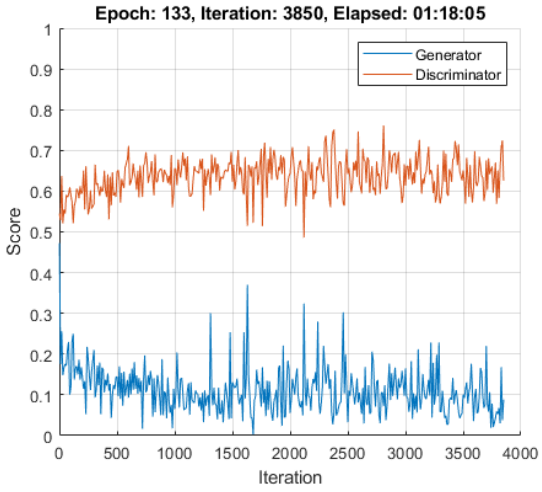
If the score does not recover from these values for many iterations, then it is better to stop the training. If this happens, then try balancing the performance of generator and the discriminator by:

- Improving the discriminator's ability to learn more features by increasing the number of filters
- Impairing the generator by adding dropout layers
- Impairing the generator by reducing its number of filters

Mode Collapse

Mode collapse is when the GAN produces a small variety of images with many duplicates (modes). This happens when the generator is unable to learn a rich feature representation because it learns to associate similar outputs to multiple different inputs. To check for mode collapse, inspect the generated images. If there is little diversity in the output and some of them are almost identical, then there is likely mode collapse.

This plot shows an example of mode collapse. Notice that the generated images plot contains a lot of almost identical images, even though the inputs to the generator were different and random.



If you observe this happening, then try to increase the ability of the generator to create more diverse outputs by:

- Increasing the dimensions of the input data to the generator
- Increasing the number of filters of the generator to allow it to generate a wider variety of features
- Impairing the discriminator by randomly giving false labels to real images (one-sided label flipping)

For an example showing how to flip the labels of the real images, see “Train Generative Adversarial Network (GAN)” on page 3-76.

See Also

`dlnetwork` | `forward` | `predict` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Train Conditional Generative Adversarial Network (CGAN)” on page 3-88
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Automatic Differentiation Background” on page 18-200

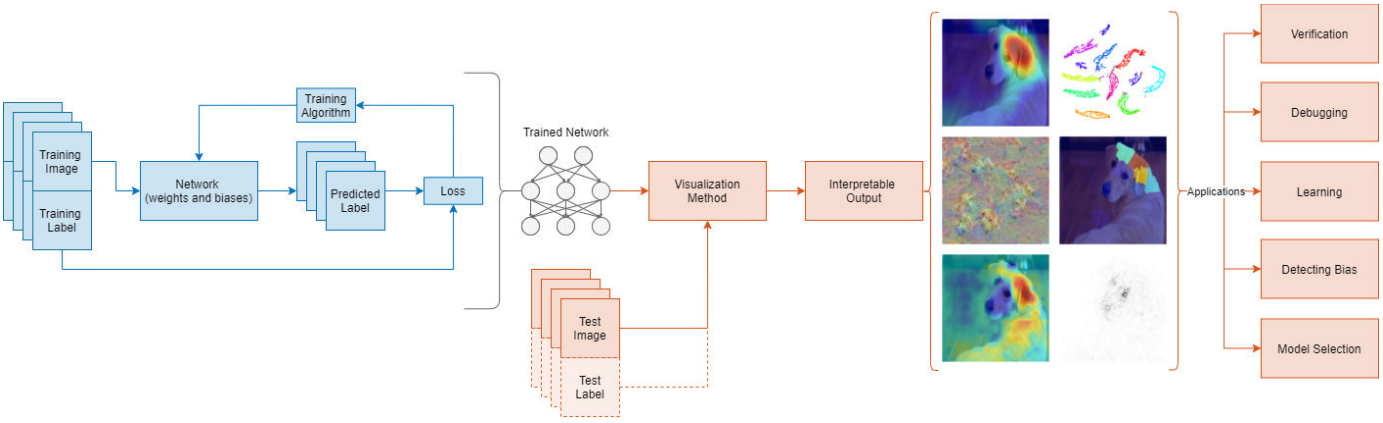
Deep Learning Visualization Methods

Deep learning networks are often described as "black boxes" because the reason that a network makes a certain decision is not always obvious. Increasingly, deep learning networks are being used in domains from medical treatment to loan applications, so understanding why a network makes a particular decision is crucial.

You can use interpretability techniques to translate network behavior into output that a person can interpret. This interpretable output can then answer questions about the predictions of a network. Interpretability techniques have many applications, for example, verification, debugging, learning, assessing bias, and model selection.

You can apply interpretability techniques after network training, or build them into the network. The advantage of post-training methods is that you do not have to spend time constructing an interpretable deep learning network. This topic focuses on post-training methods that use test images to explain the predictions of a network trained on image data.

Visualization methods are a type of interpretability technique that explain network predictions using visual representations of what a network is looking at. There are many techniques for visualizing network behavior, such as heat maps, saliency maps, feature importance maps, and low-dimensional projections.



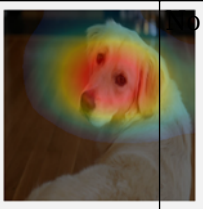
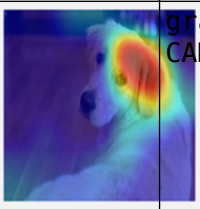
Visualization Methods

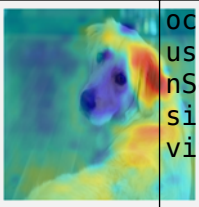
Interpretability techniques have varying characteristics; which method you use will depend on the interpretation you want and the network you have trained. Methods can be *local* and only investigate network behaviour for a specific input or *global* and investigate network behaviour across an entire data set.

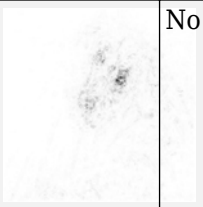
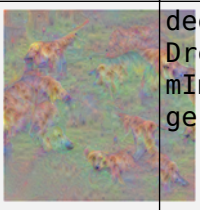
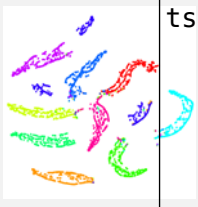
Each visualization method has a specific approach that determines the output it produces. A common distinction between methods is if they are gradient or perturbation based. *Gradient-based* methods backpropagate the signal from the output back towards the input. *Perturbation-based* methods perturb the input to the network and consider the effect of the perturbation on prediction. Another approach to interpretability technique involves mapping or approximating the complex network model to a more interpretable space. For example, some methods approximate the network predictions using a simpler, more interpretable model. Other methods use dimension reduction techniques to reduce high-dimensional activations down to interpretable 2-D or 3-D space.

The following table compares visualization interpretability techniques for deep learning models for image classification. For an example showing how to use visualization methods to investigate the predictions of an image classification network, see “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2.

Deep Learning Visualization Methods for Image Classification

Method	Example Visualization	Function	Locality	Approach	Resolution	Requires Tuning	Description
Activations		`activations`	Local	Activation visualization	Low	No	Visualizing activations is a simple way of understanding network behavior. Most convolutional neural networks learn to detect features like color and edges in their first convolutional layers. In deeper convolutional layers, the network learns to detect more complicated features. For more information, see “Visualize Activations of a Convolutional Neural Network” on page 5-141.
CAM		`cam`	Local	Gradient-based class activation heat map	Low	No	Class activation mapping (CAM) is a simple technique for generating visual explanations of the predictions of convolutional neural networks [1]. CAM uses the global average pooling layer in a convolutional neural network to generate a map that highlights which parts of an image the network is using with respect to a particular class label. For more information, see “Investigate Network Predictions Using Class Activation Mapping” on page 5-123.
Grad-CAM		`gradCAM`	Local	Gradient-based class activation heat map	Low	No	Gradient-weighted class activation mapping (Grad-CAM) is a generalization of the CAM method that uses the gradient of the classification score with respect to the convolutional features determined by the network to understand which parts of the image are most important for classification [2]. The places where the gradient is large are the places where the final score depends most on the data. Grad-CAM gives similar results to CAM without the architecture restrictions of CAM. For more information, see “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21 and “Explore Semantic Segmentation Network Using Grad-CAM” on page 5-66.

Method	Example Visualization	Function	Locality	Approach	Resolution	Requires Tuning	Description
Occlusion sensitivity		`occlusionSensitivity`	Local	Perturbation-based heat map	Low to medium	Yes	Occlusion sensitivity measures network sensitivity to small perturbations in input data. The method perturbs small areas of the input by replacing it with an occluding mask, typically a gray square. As the mask moves across the image, the technique measures the change in probability score for a given class. You can use occlusion sensitivity to highlight which parts of the image are most important to the classification. To get the best results from occlusion sensitivity, you must choose the right values for the `MaskSize` and `Stride` options. This tuning provides more flexibility to examine the input features at different length scales. For more information, see “Understand Network Predictions Using Occlusion” on page 5-24.
LIME		`imageLIME`	Local	Perturbation-based proxy model, feature importance	Low to high	Yes	The LIME technique approximates the classification behavior of a deep learning network using a simpler, more interpretable model, such as a linear model or a regression tree [3]. The simple model determines the importance of features of the input data, as a proxy for the importance of the features to the deep learning network. For more information, see “Understand Network Predictions Using LIME” on page 5-42 and “Investigate Spectrogram Classifications Using LIME” on page 5-49.

Method	Example Visualization	Function	Locality	Approach	Resolution	Requires Tuning	Description
Gradient attribution		No	Local	Gradient-based saliency map	High	No	Gradient attribution methods provide pixel-resolution maps showing which pixels are most important to the network classification decisions [4][5]. These methods compute the gradient of the class score with respect to the input pixels. Intuitively, the maps show which pixels most affect the class score when changed. The gradient attribution methods produce maps the same size as the input image. Therefore, gradient attribution maps have a high resolution, but they tend to be much noisier, as a well-trained deep network is not strongly dependent on the exact value of specific pixels. For more information, see “Investigate Classification Decisions Using Gradient Attribution Techniques” on page 5-31.
Deep dream		`deepDreamImage`	Global	Gradient-based activation maximization	Low to high	Yes	Deep Dream is a feature visualization technique that synthesizes images that strongly activate network layers [6]. By visualizing these images, you can highlight the image features learned by a network. These images are useful for understanding and diagnosing network behavior. For more information, see “Deep Dream Images Using GoogLeNet” on page 5-15.
t-SNE		`tsne`	Global	Dimension reduction	N/A	No	t-SNE is a dimension reduction technique that preserves distances so that points near each other in the high-dimension representation are also near each other in the low-dimensional representation [7]. You can use t-SNE to visualize how deep learning networks change the representation of input data as it passes through the network layers. For more information, see “View Network Behavior Using tsne” on page 5-129.

Method	Example Visualization	Function	Locality	Approach	Resolution	Requires Tuning	Description
Maximal and minimal activating images		No	Global	Gradient-based activation maximization	N/A	No	Visualizing images that strongly or weakly activate the network for each class is a simple way of understating your network. Images that strongly activate highlight what the network thinks a "typical" image from that class looks like. Images that weakly activate can help you to discover why your network makes incorrect classification predictions. For more information, see “Visualize Image Classifications Using Maximal and Minimal Activating Images” on page 5-163.

Interpretability Methods for Nonimage Data

Many interpretability focus on interpreting image classification or regression networks. Interpreting nonimage data is often more challenging due to the nonvisual nature of the data. To explore the activations of an LSTM network, use the `activations` and `tsne` functions. For an example showing how to explore the predictions of an LSTM network, see “Visualize Activations of LSTM Network” on page 5-152. To explore the behavior of a network trained on tabular features, use the `lime` and `shapley` functions. For an example showing how to interpret a feature input network, see “Interpret Deep Network Predictions on Tabular Data Using LIME” on page 5-59. For more information about interpreting machine learning models, see “Interpret Machine Learning Models” (Statistics and Machine Learning Toolbox).

References

[1] Zhou, Bolei, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. "Learning Deep Features for Discriminative Localization." In *2016 Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* : 2921–2929. Las Vegas: IEEE, 2016.

[2] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization.” In *2017 Proceedings of the IEEE Conference on Computer Vision*: 618–626. Venice, Italy: IEEE, 2017. https://doi.org/10.1109/ICCV.2017.74.

[3] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. “‘Why Should I Trust You?’: Explaining the Predictions of Any Classifier.” In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144. New York, NY: Association for Computing Machinery, 2016. https://doi.org/10.1145/2939672.2939778.

[4] Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps.” Preprint, submitted April 19, 2014. https://arxiv.org/abs/1312.6034.

[5] Tomsett, Richard, Dan Harborne, Supriyo Chakraborty, Prudhvi Gurram, and Alun Preece. “Sanity Checks for Saliency Metrics.” *Proceedings of the AAAI Conference on Artificial Intelligence*, 34, no. 04, (April 2020): 6021–29, https://doi.org/10.1609/aaai.v34i04.6064.

[6] TensorFlow. "DeepDreaming with TensorFlow." https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/deepdream.ipynb.

[7] van der Maaten, Laurens, and Geoffrey Hinton. "Visualizing Data Using t-SNE." *Journal of Machine Learning Research,* 9 (2008): 2579–2605.

See Also

`gradCAM` | `imageLIME` | `occlusionSensitivity` | `deepDreamImage` | `tsne` | `activations`

Related Examples

- “Explore Network Predictions Using Deep Learning Visualization Techniques” on page 5-2
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-21
- “Understand Network Predictions Using LIME” on page 5-42
- “Understand Network Predictions Using Occlusion” on page 5-24
- “View Network Behavior Using tsne” on page 5-129
- “Interpret Machine Learning Models” (Statistics and Machine Learning Toolbox)

Manage Deep Learning Experiments

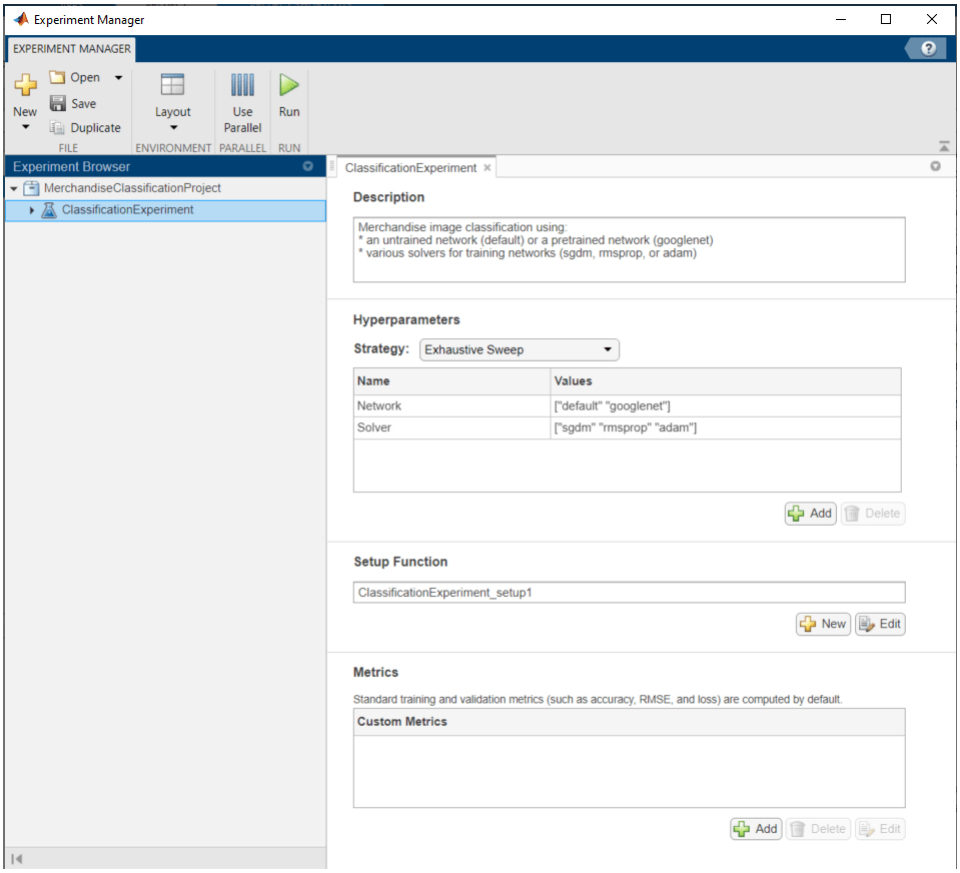
Create a Deep Learning Experiment for Classification

This example shows how to train a deep learning network for classification by using **Experiment Manager**. In this example, you train two networks to classify images of MathWorks merchandise into five classes. Each network is trained using three algorithms. In each case, a confusion matrix compares the true classes for a set of validation images with the classes predicted by the trained network. For more information on training a network for image classification, see “Train Deep Learning Network to Classify New Images” on page 3-6.

This experiment requires the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package. Before you run the experiment, install this support package by calling the `googlenet` function and clicking the download link.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`ClassificationExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Merchandise image classification using:
* an untrained network (default) or a pretrained network (googlenet)
* various solvers for training networks (sgdm, rmsprop, or adam)
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses two hyperparameters:

- `Network` specifies the network to train. The options include `"default"` (the default network provided by the experiment template for image classification) and `"googlenet"` (a pretrained GoogLeNet network with modified layers for transfer learning).
- `Solver` indicates the algorithm used to train the network. The options include `"sgdm"` (stochastic gradient descent with momentum), `"rmsprop"` (root mean square propagation), and `"adam"` (adaptive moment estimation). For more information about these algorithms, see “Stochastic Gradient Descent”.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. In this example, the setup function has three sections.

- **Load Training Data** defines image datastores containing the training and validation data. This example loads images from the file `MerchData.zip`. This small data set contains 75 images of MathWorks merchandise, belonging to five different classes. The images are of size 227-by-227-by-3. For more information on this data set, see “Image Data Sets” on page 19-118.

```
filename = "MerchData.zip";
dataFolder = fullfile(tempdir,"MerchData");
if ~exist(dataFolder,"dir")
    unzip(filename,tempdir);
end

imdsTrain = imageDatastore(dataFolder, ...
    IncludeSubfolders=true, ....
    LabelSource="foldernames");

numTrainingFiles = 0.7;
[imdsTrain,imdsValidation] = splitEachLabel(imdsTrain,numTrainingFiles);
```

- **Define Network Architecture** defines the architecture for a convolutional neural network for deep learning classification. In this example, the choice of network to train depends on the value of the hyperparameter `Network`.

```
switch params.Network
    case "default"
        inputSize = [227 227 3];
        numClasses = 5;
        layers = [
            imageInputLayer(inputSize)
            convolution2dLayer(5,20)
            batchNormalizationLayer
            reluLayer
            fullyConnectedLayer(numClasses)
            softmaxLayer
            classificationLayer];
    case "googlenet"
```

```
        inputSize = [224 224 3];
        numClasses = 5;
        imdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain);
        imdsValidation = augmentedImageDatastore(inputSize(1:2), ...
            imdsValidation);
        net = googlenet;
        layers = layerGraph(net);
        newLearnableLayer = fullyConnectedLayer(numClasses, ...
            Name="new_fc", ...
            WeightLearnRateFactor=10, ...
            BiasLearnRateFactor=10);
        layers = replaceLayer(layers,"loss3-classifier",newLearnableLayer);
        newClassLayer = classificationLayer(Name="new_classoutput");
        layers = replaceLayer(layers,"output",newClassLayer);
    otherwise
        error("Undefined network selection.");
end
```

- **Specify Training Options** defines a `trainingOptions` object for the experiment. The example trains the network for 8 epochs using the algorithm specified by the `Solver` entry in the hyperparameter table.

```
options = trainingOptions(params.Solver, ...
    MiniBatchSize=10, ...
    MaxEpochs=8, ...
    InitialLearnRate=1e-4, ...
    Shuffle="every-epoch", ...
    ValidationData=imdsValidation, ...
    ValidationFrequency=5, ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in the appendix at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any custom metric functions.

Run Experiment

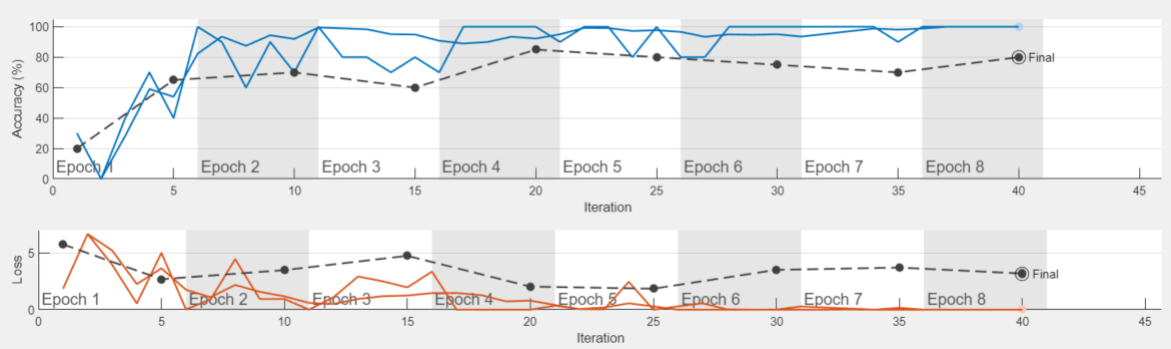
When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the accuracy and loss for each trial.

Trial	Status	Progress	Elapsed Time	Network	Solver	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete	100.0%	0 hr 0 min 7 sec	default	sgdm	100.0000	1.1921e-8	80.0000	3.1888
2	Complete	100.0%	0 hr 0 min 22 sec	googlenet	sgdm	100.0000	0.0495	95.0000	0.2479
3	Complete	100.0%	0 hr 0 min 8 sec	default	rmsprop	100.0000	0.0000	80.0000	2.6453
4	Complete	100.0%	0 hr 0 min 23 sec	googlenet	rmsprop	100.0000	0.0003	95.0000	0.2989
5	Complete	100.0%	0 hr 0 min 7 sec	default	adam	100.0000	0.0000	80.0000	2.5187
6	Complete	100.0%	0 hr 0 min 26 sec	googlenet	adam	100.0000	0.0003	90.0000	0.4894

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



Evaluate Results

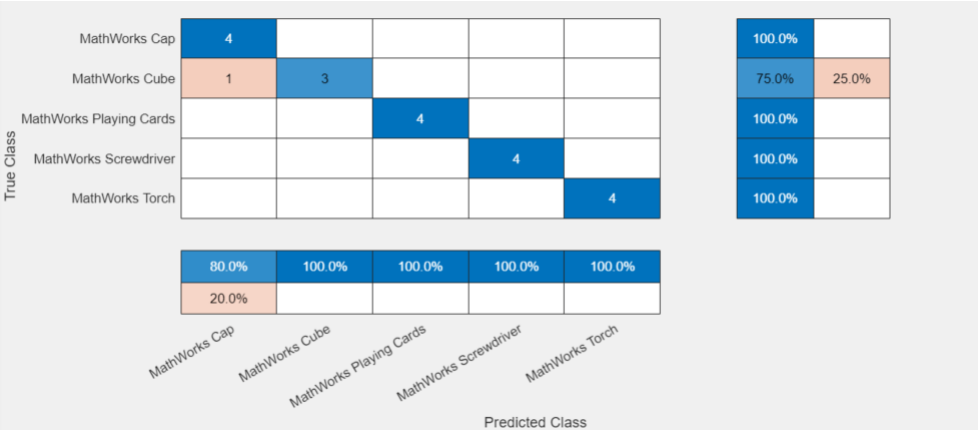
To find the best result for your experiment, sort the table of results by validation accuracy.

1. Point to the **Validation Accuracy** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

The trial with the highest validation accuracy appears at the top of the results table.

Trial	Status	Progress	Elapsed Time	Network	Solver	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
2	Complete	100.0%	0 hr 0 min 22 sec	googlenet	sgdm	100.0000	0.0495	95.0000	0.2479
4	Complete	100.0%	0 hr 0 min 23 sec	googlenet	rmsprop	100.0000	0.0003	95.0000	0.2989
6	Complete	100.0%	0 hr 0 min 26 sec	googlenet	adam	100.0000	0.0003	90.0000	0.4894
1	Complete	100.0%	0 hr 0 min 7 sec	default	sgdm	100.0000	1.1921e-8	80.0000	3.1888
3	Complete	100.0%	0 hr 0 min 8 sec	default	rmsprop	100.0000	0.0000	80.0000	2.6453
5	Complete	100.0%	0 hr 0 min 7 sec	default	adam	100.0000	0.0000	80.0000	2.5187

To display the confusion matrix for this trial, select the top row in the results table and click **Confusion Matrix**.



To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **Validation Accuracy** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.

For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `imdsTrain` is an image datastore for the training data.
- `layers` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [imdsTrain,layers,options] = ClassificationExperiment_setup1(params)

filename = "MerchData.zip";
dataFolder = fullfile(tempdir,"MerchData");
if ~exist(dataFolder,"dir")
```

```
    unzip(filename,tempdir);
end

imdsTrain = imageDatastore(dataFolder, ...
    IncludeSubfolders=true, ....
    LabelSource="foldernames");

numTrainingFiles = 0.7;
[imdsTrain,imdsValidation] = splitEachLabel(imdsTrain,numTrainingFiles);

switch params.Network
    case "default"
        inputSize = [227 227 3];
        numClasses = 5;

        layers = [
            imageInputLayer(inputSize)
            convolution2dLayer(5,20)
            batchNormalizationLayer
            reluLayer
            fullyConnectedLayer(numClasses)
            softmaxLayer
            classificationLayer];

    case "googlenet"
        inputSize = [224 224 3];
        numClasses = 5;

        imdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain);
        imdsValidation = augmentedImageDatastore(inputSize(1:2), ...
            imdsValidation);

        net = googlenet;
        layers = layerGraph(net);

        newLearnableLayer = fullyConnectedLayer(numClasses, ...
            Name="new_fc", ...
            WeightLearnRateFactor=10, ...
            BiasLearnRateFactor=10);
        layers = replaceLayer(layers,"loss3-classifier",newLearnableLayer);

        newClassLayer = classificationLayer(Name="new_classoutput");
        layers = replaceLayer(layers,"output",newClassLayer);

    otherwise
        error("Undefined network selection.");
end

options = trainingOptions(params.Solver, ...
    MiniBatchSize=10, ...
    MaxEpochs=8, ...
    InitialLearnRate=1e-4, ...
    Shuffle="every-epoch", ...
    ValidationData=imdsValidation, ...
    ValidationFrequency=5, ...
    Verbose=false);
```

```
end
```

See Also

Apps
Experiment Manager

Functions
`googlenet` | `trainNetwork` | `trainingOptions`

More About

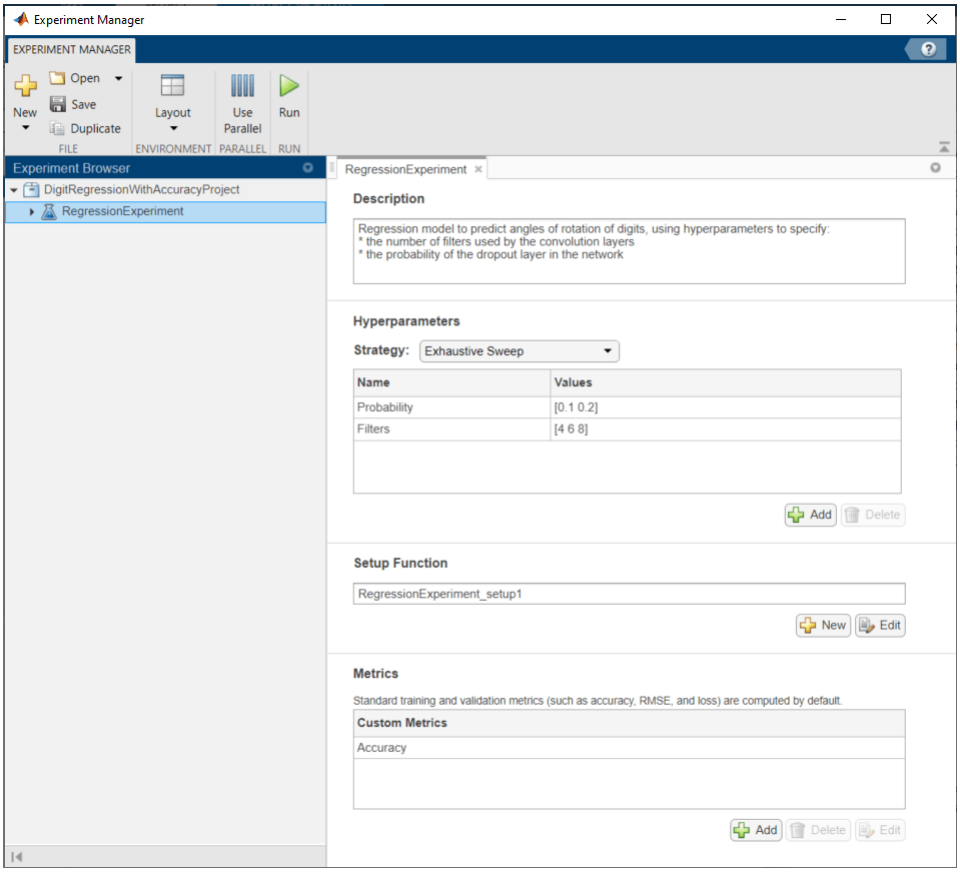
- “Get Started with Transfer Learning”
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19
- “Create a Deep Learning Experiment for Regression” on page 6-9

Create a Deep Learning Experiment for Regression

This example shows how to train a deep learning network for regression by using **Experiment Manager**. In this example, you use a regression model to predict the angles of rotation of handwritten digits. A custom metric function determines the fraction of angle predictions within an acceptable error margin from the true angles. For more information on using a regression model, see “Train Convolutional Neural Network for Regression” on page 3-53.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`RegressionExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Regression model to predict angles of rotation of digits, using hyperparameters to specify:
* the number of filters used by the convolution layers
* the probability of the dropout layer in the network
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses two hyperparameters:

- `Probability` sets the probability of the dropout layer in the neural network. By default, the values for this hyperparameter are specified as `[0.1 0.2]`.
- `Filters` indicates the number of filters used by the first convolution layer in the neural network. In the subsequent convolution layers, the number of filters is a multiple of this value. By default, the values of this hyperparameter are specified as `[4 6 8]`.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns four outputs that you use to train a network for image regression problems. The setup function has three sections.

- **Load Training Data** defines the training and validation data for the experiment as 4-D arrays. The training and validation data sets each contain 5000 images of digits from 0 to 9. The regression values correspond to the angles of rotation of the digits.

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
[XValidation,~,YValidation] = digitTest4DArrayData;
```

- **Define Network Architecture** defines the architecture for a convolutional neural network for regression.

```
inputSize = [28 28 1];
numFilters = params.Filters;
layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(3,numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,Stride=2)
    convolution2dLayer(3,2*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,Stride=2)
    convolution2dLayer(3,4*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer
    dropoutLayer(params.Probability)
    fullyConnectedLayer(1)
    regressionLayer];
```

- **Specify Training Options** defines a `trainingOptions` object for the experiment. The example trains the network for 30 epochs. The learning rate is initially 0.001 and drops by a factor of 0.1 after 20 epochs. The software trains the network on the training data and calculates the root mean squared error (RMSE) and loss on the validation data at regular intervals during training. The validation data is not used to update the network weights.

```
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
```

```
options = trainingOptions("sgdm", ...
    MiniBatchSize=miniBatchSize, ...
    MaxEpochs=30, ...
    InitialLearnRate=1e-3, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=20, ...
    Shuffle="every-epoch", ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. Experiment Manager evaluates these functions each time it finishes training the network. To inspect a metric function, select the name of the metric function and click **Edit**. The metric function opens in MATLAB Editor.

This example includes a metric function `Accuracy` that determines the percentage of angle predictions within an acceptable error margin from the true angles. By default, the function uses a threshold of 10 degrees. The code for the metric function appears in Appendix 2 at the end of this example.

Run Experiment

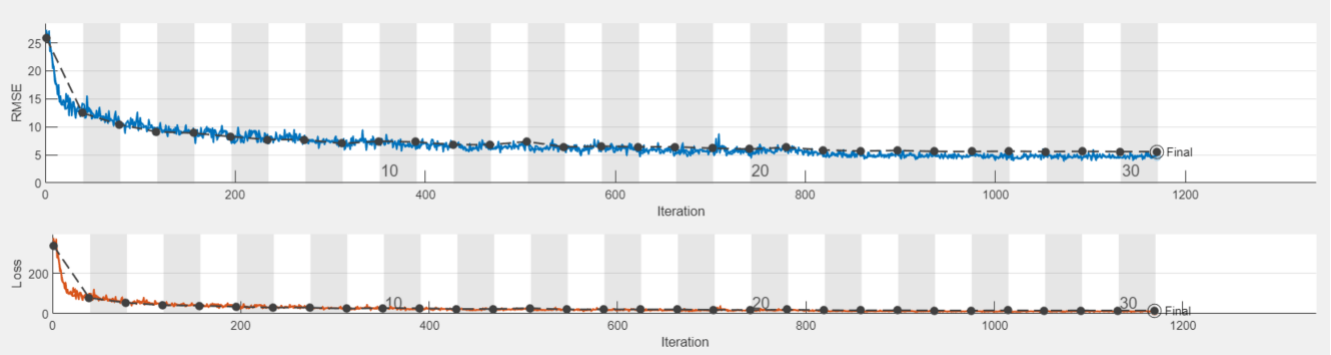
When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the RMSE and loss for each trial. The table also displays the accuracy of the trial, as determined by the custom metric function `Accuracy`.

Trial	Status	Progress	Elapsed Time	Probability	Filters	Training RMSE	Training Loss	Validation RMSE	Validation Loss	Accuracy
1	Complete	100.0%	0 hr 1 min 13 sec	0.1000	4.0000	4.6942	11.0176	5.5271	15.2746	93.6800
2	Complete	100.0%	0 hr 1 min 19 sec	0.2000	4.0000	4.8882	11.9472	5.7182	16.3490	92.4400
3	Complete	100.0%	0 hr 1 min 24 sec	0.1000	6.0000	3.7716	7.1125	4.7295	11.1842	96.5000
4	Complete	100.0%	0 hr 1 min 17 sec	0.2000	6.0000	4.5593	10.3937	4.7918	11.4805	96.3000
5	Complete	100.0%	0 hr 1 min 18 sec	0.1000	8.0000	3.6132	6.5276	4.3334	9.3892	97.4800
6	Complete	100.0%	0 hr 1 min 20 sec	0.2000	8.0000	4.2288	8.9413	4.4778	10.0253	97.0200

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



Evaluate Results

To find the best result for your experiment, sort the table of results by accuracy.

1. Point to the **Accuracy** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

The trial with the highest accuracy appears at the top of the results table.

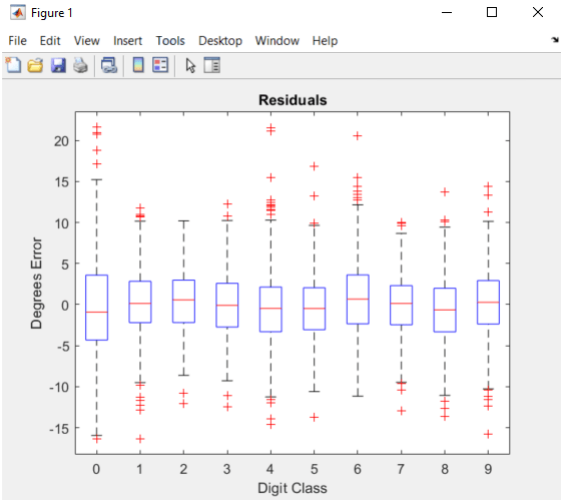
Trial	Status	Progress	Elapsed Time	Probability	Filters	Training RMSE	Training Loss	Validation RMSE	Validation Loss	Accuracy
5	Complete	100.0%	0 hr 1 min 18 sec	0.1000	8.0000	3.6132	6.5276	4.3334	9.3892	97.4800
6	Complete	100.0%	0 hr 1 min 20 sec	0.2000	8.0000	4.2288	8.9413	4.4778	10.0253	97.0200
3	Complete	100.0%	0 hr 1 min 24 sec	0.1000	6.0000	3.7716	7.1125	4.7295	11.1842	96.5000
4	Complete	100.0%	0 hr 1 min 17 sec	0.2000	6.0000	4.5593	10.3937	4.7918	11.4805	96.3000
1	Complete	100.0%	0 hr 1 min 13 sec	0.1000	4.0000	4.6942	11.0176	5.5271	15.2746	93.6800
2	Complete	100.0%	0 hr 1 min 19 sec	0.2000	4.0000	4.8882	11.9472	5.7182	16.3490	92.4400

To test the performance of an individual trial, export the trained network and display a box plot of the residuals for each digit class.

1. Select the trial with the highest accuracy.
2. On the **Experiment Manager** toolstrip, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported network. The default name is `trainedNetwork`.
4. Use the exported network as the input to the function `plotResiduals`, which is listed in Appendix 3 at the end of this example. For instance, in the MATLAB Command Window, enter:

```
plotResiduals(trainedNetwork)
```

The function creates a residual box plot for each digit. The digit classes with highest accuracy have a mean close to zero and little variance.



To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **Accuracy** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.

For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `XTrain` is a 4-D array containing the training data.
- `YTrain` is a 1-D array containing the regression values for training,
- `layers` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [XTrain,YTrain,layers,options] = RegressionExperiment_setup1(params)
```

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
[XValidation,~,YValidation] = digitTest4DArrayData;

inputSize = [28 28 1];
numFilters = params.Filters;

layers = [
    imageInputLayer(inputSize)

    convolution2dLayer(3,numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,Stride=2)

    convolution2dLayer(3,2*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer

    averagePooling2dLayer(2,Stride=2)

    convolution2dLayer(3,4*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(3,4*numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer

    dropoutLayer(params.Probability)
    fullyConnectedLayer(1)
    regressionLayer];

miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions("sgdm", ...
    MiniBatchSize=miniBatchSize, ...
    MaxEpochs=30, ...
    InitialLearnRate=1e-3, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.1, ...
    LearnRateDropPeriod=20, ...
    Shuffle="every-epoch", ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);

end
```

Appendix 2: Compute Accuracy of Regression Model

This function calculates the number of predictions within an acceptable error margin from the true angles.

```
function metricOutput = Accuracy(trialInfo)
```

```
[XValidation,~,YValidation] = digitTest4DArrayData;
YPredicted = predict(trialInfo.trainedNetwork,XValidation);
predictionError = YValidation - YPredicted;

thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numValidationImages = numel(YValidation);

metricOutput = 100*numCorrect/numValidationImages;

end
```

Appendix 3: Display Box Plot of Residuals for Each Digit

This function creates a residual box plot for each digit.

```
function plotResiduals(net)

[XValidation,~,YValidation] = digitTest4DArrayData;
YPredicted = predict(net,XValidation);
predictionError = YValidation - YPredicted;
residualMatrix = reshape(predictionError,500,10);

figure
boxplot(residualMatrix,...
    'Labels',{'0','1','2','3','4','5','6','7','8','9'})
xlabel('Digit Class')
ylabel('Degrees Error')
title('Residuals')

end
```

See Also

Apps
Experiment Manager

Functions
`trainNetwork` | `trainingOptions`

More About

- “Train Convolutional Neural Network for Regression” on page 3-53
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19
- “Create a Deep Learning Experiment for Classification” on page 6-2

Use Experiment Manager to Train Networks in Parallel

By default, **Experiment Manager** runs one trial of your experiment at a time on a single CPU. If you have Parallel Computing Toolbox, you can configure your experiment to run multiple trials at the same time or to run a single trial at a time on multiple GPUs, on a cluster, or in the cloud.

Training Scenario	Recommendation
Run multiple trials at the same time using one parallel worker for each trial.	Set up your parallel environment and enable the **Use Parallel** option before running your experiment. Experiment Manager runs as many simultaneous trials as there are workers in your parallel pool. If you have multiple GPUs, parallel execution typically increases the speed of your experiment. However, if you have a single GPU, all workers share that GPU, so you do not get the training speed-up and you increase the chances of the GPU running out of memory.
Run a single trial at a time on multiple parallel workers.	**Built-In Training Experiments:** • In the experiment setup function, set the training option `ExecutionEnvironment` to `"multi-gpu"` or `"parallel"`. For more information, see "Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud" on page 7-2. • If you are using a partitionable datastore, enable background dispatching by setting the training option `DispatchInBackground` to `true`. For more information, see "Use Datastore for Parallel Training and Background Dispatching" on page 19-8. • Set up your parallel environment and disable the **Use Parallel** option before running your experiment.
	Custom Training Experiments: • In the experiment training function, set up your parallel environment and use an `spmd` block to define a custom parallel training loop. For more information, see "Train Network in Parallel with Custom Training Loop" on page 7-55. • Disable the **Use Parallel** option before running your experiment.

Tip To run an experiment in parallel using MATLAB Online, you must have access to a Cloud Center cluster. For more information, see "Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online" (Parallel Computing Toolbox).

Set Up Parallel Environment

Train on Multiple GPUs

If you have multiple GPUs, parallel execution typically increases the speed of your experiment. Using a GPU for deep learning requires Parallel Computing Toolbox and a supported GPU device. For more information, see "GPU Support by Release" (Parallel Computing Toolbox).

- For built-in training experiments, GPU support is automatic. By default, these experiments use a GPU if one is available.
- For custom training experiments, computations occur on a CPU by default. To train on a GPU, convert your data to `gpuArray` objects. To determine whether a usable GPU is available, call the `canUseGPU` function.

For best results, before you run your experiment, create a parallel pool with as many workers as GPUs. You can check the number of available GPUs by using the `gpuDeviceCount` function.

```
numGPUs = gpuDeviceCount("available");
parpool(numGPUs)
```

Train on Cluster or in Cloud

If your experiments take a long time to run on your local machine, you can accelerate training by using a computer cluster on your onsite network or by renting high-performance GPUs in the cloud. After you complete the initial setup, you can run your experiments with minimal changes to your code. Working on a cluster or in the cloud requires MATLAB Parallel Server™. For more information, see “Deep Learning in the Cloud” on page 7-10.

Run Multiple Trials in Parallel

To run multiple trials of your experiment in parallel, on the Experiment Manager toolstrip, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile.

Experiment Manager runs as many simultaneous trials as there are workers in your parallel pool. All other trials in your experiment are queued for later evaluation. A table of results displays the status and progress of each trial.

Trial	Status	Progress	Elapsed Time	myInitialLearnRate	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete	100.0%	0 hr 3 min 33 sec	0.0025	100.0000	0.0699	58.6400	1.2963
2	Complete	100.0%	0 hr 3 min 30 sec	0.0050	100.0000	0.0318	62.4800	1.3210
3	Running	30.7%	0 hr 0 min 56 sec	0.0075				
4	Running	30.7%	0 hr 0 min 56 sec	0.0100				
5	Queued	0.0%		0.0125				
6	Queued	0.0%		0.0150				

While the experiment is running, you can track its progress by displaying the training plot for each trial. You can also stop trials that appear to be underperforming. For more information, see “Stop and Restart Training”.

Note Experiment Manager does not support the execution of multiple trials in parallel when you set the training option `ExecutionEnvironment` to `"multi-gpu"` or `"parallel"` or when you enable the training option `DispatchInBackground`. Use these options to speed up your training only if you intend to run one trial of your experiment at a time. For more information, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2 and “Use Datastore for Parallel Training and Background Dispatching” on page 19-8.

See Also

Apps
Experiment Manager

Functions
`trainingOptions` | `canUseGPU` | `gpuDeviceCount` | `parpool` | `spmd`

Objects
`gpuArray`

Related Examples

- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46
- “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32
- “Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online” (Parallel Computing Toolbox)

Evaluate Deep Learning Experiments by Using Metric Functions

This example shows how to use metric functions to evaluate the results of an experiment. By default, when you run a built-in training experiment, **Experiment Manager** computes the loss, accuracy (for classification experiments), and root mean squared error (for regression experiments) for each trial in your experiment. To compute other measures, create your own metric function. For example, you can define metric functions to:

- Test the prediction performance of a trained network.
- Evaluate the training progress by computing the slope of the validation loss over the final epoch.
- Display the size of the network used in an experiment that uses different network architectures for each trial.

When each trial finishes training, Experiment Manager evaluates the metric functions and displays their values in the results table.

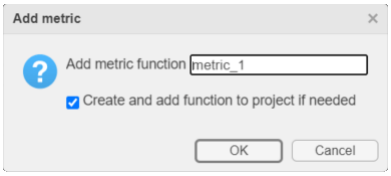
In this example, you train a network to classify images of handwritten digits. Two metric functions determine how well the trained network identifies the images of the numerals one and seven. For more information on using Experiment Manager to train a network for image classification, see “Image Classification by Sweeping Hyperparameters”.

Define Metric Functions

Add a metric function to a built-in training experiment.

1. In the **Experiment** pane, under **Metrics**, click **Add**.

2. In the **Add metric** dialog box, enter a name for the metric function and click **OK**. If you enter the name of a function that already exists in the project, Experiment Manager adds it to the experiment. Otherwise, Experiment Manager creates a function defined by a default template.



3. Select the name of the metric function and click **Edit**. The metric function opens in MATLAB® Editor.

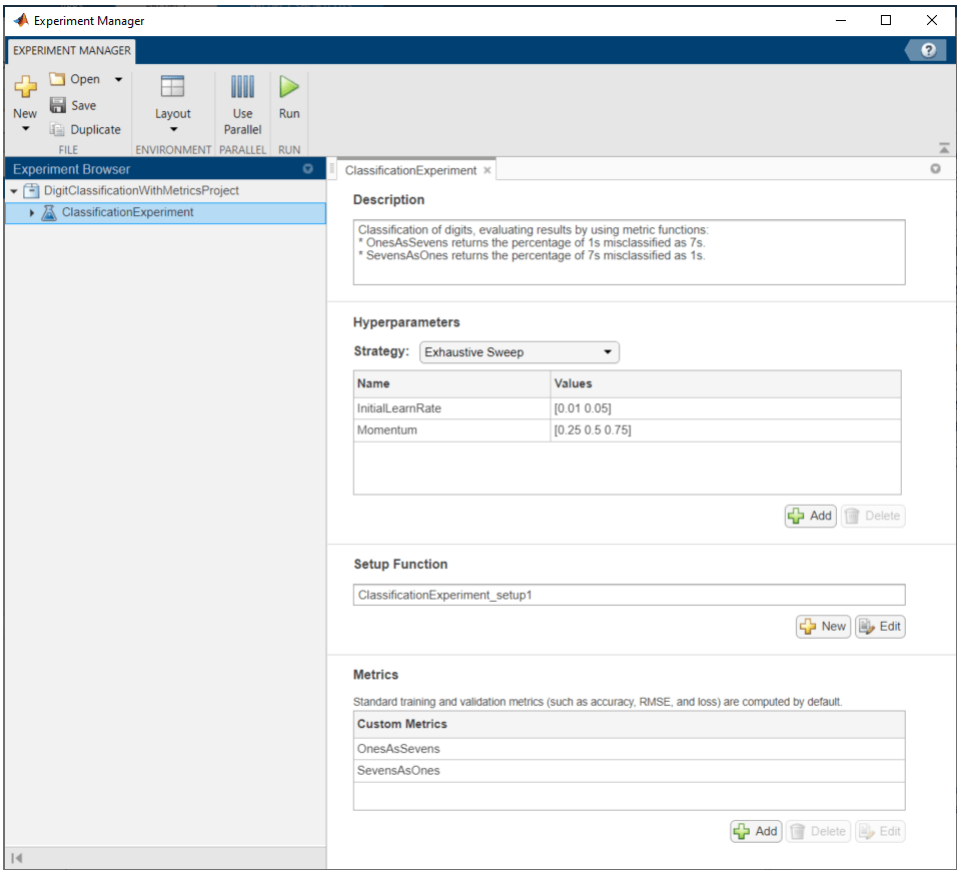
The input to a metric function is a structure with three fields:

- `trainedNetwork` is the `SeriesNetwork` object or `DAGNetwork` object returned by the `trainNetwork` function. For more information, see “net”.
- `trainingInfo` is a structure containing the training information returned by the `trainNetwork` function. For more information, see “info”.
- `parameters` is a structure with fields from the hyperparameter table.

The output of a custom metric function must be a scalar number, a logical value, or a string.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`ClassificationExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Classification of digits, evaluating results by using metric functions:
* OnesAsSevens returns the percentage of 1s misclassified as 7s.
* SevensAsOnes returns the percentage of 7s misclassified as 1s.
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses the hyperparameters `InitialLearnRate` and `Momentum`.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. In this example, the setup function has three sections.

- **Load Training Data** defines image datastores containing the training and validation data. This example loads images from the Digits data set. For more information on this data set, see “Image Data Sets” on page 19-118.

```
digitDatasetPath = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imdsTrain = imageDatastore(digitDatasetPath, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

numTrainingFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imdsTrain,numTrainingFiles);
```

- **Define Network Architecture** defines the architecture for a convolutional neural network for deep learning classification. This example uses the default classification network provided by the setup function template.

```
inputSize = [28 28 1];
numClasses = 10;
layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

- **Specify Training Options** defines a `trainingOptions` object for the experiment. The example loads the values for the training options `'InitialLearnRate'` and `'Momentum'` from the hyperparameter table.

```
options = trainingOptions("sgdm", ...
    MaxEpochs=5, ...
    ValidationData=imdsValidation, ...
    ValidationFrequency=30, ...
    InitialLearnRate=params.InitialLearnRate, ...
    Momentum=params.Momentum, ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. Experiment Manager evaluates these functions each time it finishes training the network. To inspect a metric function, select the name of the metric function and click **Edit**. The metric function opens in MATLAB Editor.

This example includes two metric functions.

- `OnesAsSevens` returns the percentage of images of the numeral one that the trained network misclassifies as sevens.
- `SevensAsOnes` returns the percentage of images of the numeral seven that the trained network misclassifies as ones.

Each of these functions uses the trained network to classify the entire Digits data set. Then, the functions determine the number of images for which the actual label and the predicted label

disagree. For example, the function `OnesAsSevens` computes the number of images with an actual label of `'1'` and a predicted label of `'7'`. Similarly, the function `SevensAsOnes` computes the number of images with an actual label of `'7'` and a predicted label of `'1'`. The code for these metric functions appears in Appendix 2 and Appendix 3 at the end of this example.

Run Experiment

When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the metric function values for each trial.

Trial	Status	Progress	Elapsed Time	InitialLearnRate	Momentum	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss	OnesAsSevens	SevensAsOnes
1	Complete	100.0%	0 hr 0 min 59 sec	0.0100	0.2500	96.0938	0.2141	55.2000	1.3407	1.5000	2.6000
2	Complete	100.0%	0 hr 1 min 6 sec	0.0500	0.2500	96.8750	0.1272	56.6000	1.6953	2.2000	0.8000
3	Complete	100.0%	0 hr 1 min 3 sec	0.0100	0.5000	96.8750	0.1357	56.7600	1.3221	1.5000	2.1000
4	Complete	100.0%	0 hr 1 min 3 sec	0.0500	0.5000	94.5313	0.1784	53.5600	2.3754	2.3000	0.9000
5	Complete	100.0%	0 hr 1 min 2 sec	0.0100	0.7500	100.0000	0.0616	60.6000	1.2998	1.4000	1.5000
6	Complete	100.0%	0 hr 0 min 57 sec	0.0500	0.7500	82.8125	0.5932	47.9600	2.4599	2.5000	1.8000

Evaluate Results

To find the best result for your experiment, sort the table of results. For example, find the trial with the smallest number of misclassified ones.

1. Point to the **OnesAsSevens** column.
2. Click the triangle icon.
3. Select **Sort in Ascending Order**.

Trial	Status	Progress	Elapsed Time	InitialLearnRate	Momentum	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss	OnesAsSevens	SevensAsOnes
5	Complete	100.0%	0 hr 1 min 2 sec	0.0100	0.7500	100.0000	0.0616	60.6000	1.2998	1.4000	1.5000
1	Complete	100.0%	0 hr 0 min 59 sec	0.0100	0.2500	96.0938	0.2141	55.2000	1.3407	1.5000	2.6000
3	Complete	100.0%	0 hr 1 min 3 sec	0.0100	0.5000	96.8750	0.1357	56.7600	1.3221	1.5000	2.1000
2	Complete	100.0%	0 hr 1 min 6 sec	0.0500	0.2500	96.8750	0.1272	56.6000	1.6953	2.2000	0.8000
4	Complete	100.0%	0 hr 1 min 3 sec	0.0500	0.5000	94.5313	0.1784	53.5600	2.3754	2.3000	0.9000
6	Complete	100.0%	0 hr 0 min 57 sec	0.0500	0.7500	82.8125	0.5932	47.9600	2.4599	2.5000	1.8000

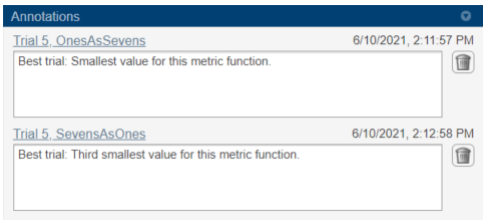
Similarly, find the trial with the smallest number of misclassified sevens by opening the drop-down menu for the **SevensAsOnes** column and selecting **Sort in Ascending Order**.

Trial	Status	Progress	Elapsed Time	InitialLearnRate	Momentum	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss	OnesAsSevens	SevensAsOnes
2	Complete	100.0%	0 hr 1 min 6 sec	0.0500	0.2500	96.8750	0.1272	56.6000	1.6953	2.2000	0.8000
4	Complete	100.0%	0 hr 1 min 3 sec	0.0500	0.5000	94.5313	0.1784	53.5600	2.3754	2.3000	0.9000
5	Complete	100.0%	0 hr 1 min 2 sec	0.0100	0.7500	100.0000	0.0616	60.6000	1.2998	1.4000	1.5000
6	Complete	100.0%	0 hr 0 min 57 sec	0.0500	0.7500	82.8125	0.5932	47.9600	2.4599	2.5000	1.8000
3	Complete	100.0%	0 hr 1 min 3 sec	0.0100	0.5000	96.8750	0.1357	56.7600	1.3221	1.5000	2.1000
1	Complete	100.0%	0 hr 0 min 59 sec	0.0100	0.2500	96.0938	0.2141	55.2000	1.3407	1.5000	2.6000

If no single trial minimizes both metric functions simultaneously, consider giving preference to a trial that ranks well for each metric. For instance, in these results, trial 5 ranks as one of the top three trials for each metric function.

To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **OnesAsSevens** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.
4. Repeat the previous steps for the **SevensAsOnes** cell.



For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `imdsTrain` is an image datastore for the training data.
- `layers` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [imdsTrain,layers,options] = ClassificationExperiment_setup1(params)

digitDatasetPath = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imdsTrain = imageDatastore(digitDatasetPath, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

numTrainingFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imdsTrain,numTrainingFiles);

inputSize = [28 28 1];
```

```
numClasses = 10;
layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

options = trainingOptions("sgdm", ...
    MaxEpochs=5, ...
    ValidationData=imdsValidation, ...
    ValidationFrequency=30, ...
    InitialLearnRate=params.InitialLearnRate, ...
    Momentum=params.Momentum, ...
    Verbose=false);

end
```

Appendix 2: Find Ones Misclassified as Sevens

This function determines the number of ones that are misclassified as sevens.

```
function metricOutput = OnesAsSevens(trialInfo)

actualValue = '1';
predValue = '7';

net = trialInfo.trainedNetwork;

digitDatasetPath = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imds = imageDatastore(digitDatasetPath, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

YActual = imds.Labels;
YPred = classify(net,imds);

K = sum(YActual == actualValue & YPred == predValue);
N = sum(YActual == actualValue);

metricOutput = 100*K/N;

end
```

Appendix 3: Find Sevens Misclassified as Ones

This function determines the number of sevens that are misclassified as ones.

```
function metricOutput = SevensAsOnes(trialInfo)

actualValue = '7';
```

```
predValue = '1';

net = trialInfo.trainedNetwork;

digitDatasetPath = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imds = imageDatastore(digitDatasetPath, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

YActual = imds.Labels;
YPred = classify(net,imds);

K = sum(YActual == actualValue & YPred == predValue);
N = sum(YActual == actualValue);

metricOutput = 100*K/N;

end
```

See Also

Apps
Experiment Manager

Functions
`trainNetwork` | `trainingOptions`

More About

- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Create a Deep Learning Experiment for Regression” on page 6-9
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26

Tune Experiment Hyperparameters by Using Bayesian Optimization

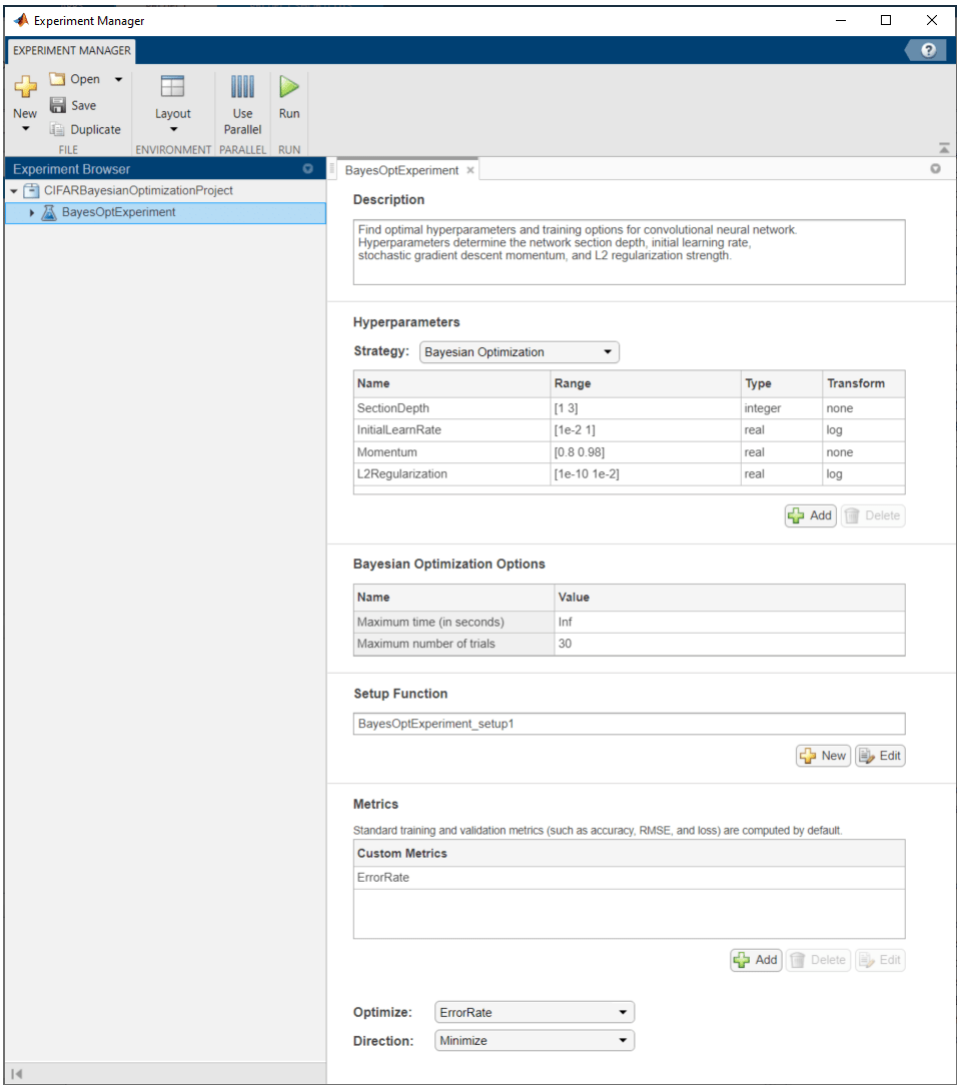
This example shows how to use Bayesian optimization in **Experiment Manager** to find optimal network hyperparameters and training options for convolutional neural networks. Bayesian optimization provides an alternative strategy to sweeping hyperparameters in an experiment. You specify a range of values for each hyperparameter and select a metric to optimize, and Experiment Manager searches for a combination of hyperparameters that optimizes your selected metric. Bayesian optimization requires Statistics and Machine Learning Toolbox™.

In this example, you train a network to classify images from the CIFAR-10 data set. The experiment uses Bayesian optimization to find the combination of hyperparameters that minimizes a custom metric function. The hyperparameters include options of the training algorithm, as well as parameters of the network architecture itself. The custom metric function determines the classification error on a randomly chosen test set. For more information on defining custom metrics in Experiment Manager, see “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19.

Alternatively, you can find optimal hyperparameter values programmatically by calling the `bayesopt` function. For more information, see “Deep Learning Using Bayesian Optimization” on page 5-99.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`BayesOptExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. Experiments that use Bayesian optimization include additional options to limit the duration of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Find optimal hyperparameters and training options for convolutional neural network.
Hyperparamters determine the network section depth, initial learning rate,
stochastic gradient descent momentum, and L2 regularization strength.
```

The **Hyperparameters** section specifies the strategy (`Bayesian Optimization`) and hyperparameter options to use for the experiment. For each hyperparameter, specify these options:

- **Range** — Enter a two-element vector that gives the lower bound and upper bound of a real- or integer-valued hyperparameter, or a string array or cell array that lists the possible values of a categorical hyperparameter.

- **Type** — Select `real` (real-valued hyperparameter), `integer` (integer-valued hyperparameter), or `categorical` (categorical hyperparameter).
- **Transform** — Select `none` (no transform) or `log` (logarithmic transform). For `log`, the hyperparameter must be `real` or `integer` and positive. With this option, the hyperparameter is searched and modeled on a logarithmic scale.

When you run the experiment, Experiment Manager searches for the best combination of hyperparameters. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials. This example uses these hyperparameters:

- `SectionDepth` — This parameter controls the depth of the network. The total number of layers in the network is `9*SectionDepth+7`. In the experiment setup function, the number of convolutional filters in each layer is proportional to `1/sqrt(SectionDepth)`, so the number of parameters and the required amount of computation for each iteration are roughly the same for different section depths.
- `InitialLearnRate` — If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result or diverge. The best learning rate can depend on your data as well as the network you are training.
- `Momentum` — Stochastic gradient descent momentum adds inertia to the parameter updates by having the current update contain a contribution proportional to the update in the previous iteration. The inertial effect results in smoother parameter updates and a reduction of the noise inherent to stochastic gradient descent.
- `L2Regularization` — Use L2 regularization to prevent overfitting. Search the space of regularization strength to find a good value. Data augmentation and batch normalization also help regularize the network.

Under **Bayesian Optimization Options**, you can specify the duration of the experiment by entering the maximum time (in seconds) and the maximum number of trials to run. To best use the power of Bayesian optimization, perform at least 30 objective function evaluations.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. In this example, the setup function has three sections.

- **Load Training Data** downloads and extracts images and labels from the CIFAR-10 data set. The data set is about 175 MB. Depending on your internet connection, the download process can take some time. For the training data, this example creates an `augmentedImageDatastore` by applying random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images. To enable network validation, the example uses 5000 images with no augmentation. For more information on this data set, see “Image Data Sets” on page 19-118.

```
datadir = tempdir;
downloadCIFARData(datadir);

[XTrain,YTrain,XTest,YTest] = loadCIFARData(datadir);
idx = randperm(numel(YTest),5000);
XValidation = XTest(:,:,:,idx);
YValidation = YTest(idx);

imageSize = [32 32 3];
pixelRange = [-4 4];
```

```
imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);
augimdsTrain = augmentedImageDatastore(imageSize,XTrain,YTrain, ...
    DataAugmentation=imageAugmenter);
```

- **Define Network Architecture** defines the architecture for a convolutional neural network for deep learning classification. In this example, the network to train has three blocks produced by the helper function `convBlock`, which is listed in Appendix 2 at the end of this example. Each block contains `SectionDepth` identical convolutional layers. Each convolutional layer is followed by a batch normalization layer and a ReLU layer. The convolutional layers have added padding so that their spatial output size is always the same as the input size. Between the blocks, max pooling layers downsample the spatial dimensions by a factor of two. To ensure that the amount of computation required in each convolutional layer is roughly the same, the number of filters increases by a factor of two from one section to the next. The number of filters in each convolutional layer is proportional to `1/sqrt(SectionDepth)`, so that networks of different depths have roughly the same number of parameters and require about the same amount of computation per iteration.

```
numClasses = numel(unique(YTrain));
numF = round(16/sqrt(params.SectionDepth));
layers = [
    imageInputLayer(imageSize)
    convBlock(3,numF,params.SectionDepth)
    maxPooling2dLayer(3,Stride=2,Padding="same")
    convBlock(3,2*numF,params.SectionDepth)
    maxPooling2dLayer(3,Stride=2,Padding="same")
    convBlock(3,4*numF,params.SectionDepth)
    averagePooling2dLayer(8)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

- **Specify Training Options** defines a `trainingOptions` object for the experiment using the values for the training options `'InitialLearnRate'`, `'Momentum'`, and `'L2Regularization'` generated by the Bayesian optimization algorithm. The example trains the network for a fixed number of epochs, validating once per epoch and lowering the learning rate by a factor of 10 during the last epochs to reduce the noise of the parameter updates and allow the network parameters to settle down closer to a minimum of the loss function.

```
miniBatchSize = 256;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions("sgdm", ...
    InitialLearnRate=params.InitialLearnRate, ...
    Momentum=params.Momentum, ...
    MaxEpochs=60, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=40, ...
    LearnRateDropFactor=0.1, ...
    MiniBatchSize=miniBatchSize, ...
    L2Regularization=params.L2Regularization, ...
    Shuffle="every-epoch", ...
    Verbose=false, ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=validationFrequency);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. Experiment Manager evaluates these functions each time it finishes training the network. To inspect a metric function, select the name of the metric function and click **Edit**. The metric function opens in MATLAB Editor.

This example includes the custom metric function `ErrorRate`. This function selects 5000 test images and labels at random, evaluates the trained network on these images, and calculates the proportion of images that the network misclassifies. The code for this function appears in Appendix 3 at the end of this example.

The **Optimize** and **Direction** fields indicate the metric that the Bayesian optimization algorithm uses as an objective function. For this experiment, Experiment Manager seeks to minimize the value of the `ErrorRate` metric.

Run Experiment

When you run the experiment, Experiment Manager searches for the best combination of hyperparameters with respect to the chosen metric. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the metric function values for each trial. Experiment Manager indicates the trial with the optimal value for the selected metric. For example, in this experiment, the third trial produces the smallest error rate.

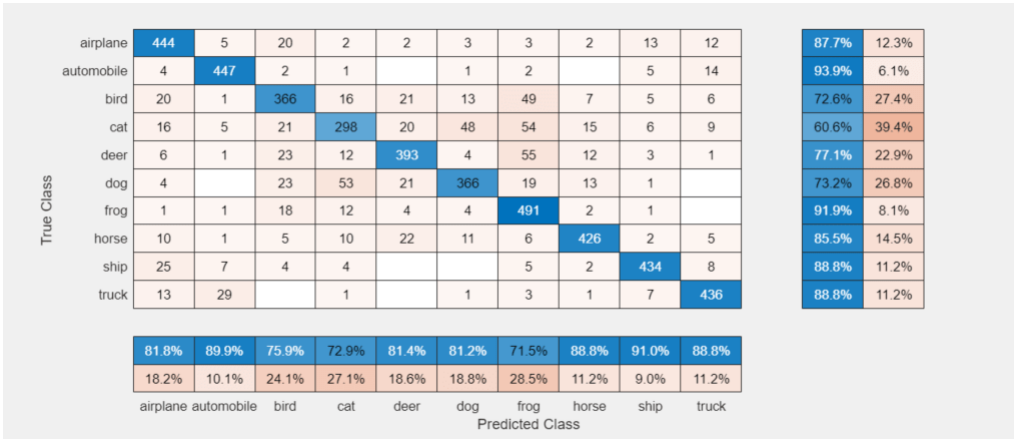
Trial	Status	Progress	Elapsed Time	SectionDepth	InitialLearnRate	Momentum	L2Regularization	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss	ErrorRate
1	Complete	100.0%	0 hr 15 min 0 sec	3.0000	0.5248	0.9176	1.0137e-10	83.9844	0.4018	81.8600	0.5678	0.1886
2	Complete	100.0%	0 hr 11 min 57 sec	2.0000	0.0108	0.9379	3.0356e-5	83.9844	0.4882	78.5600	0.6560	0.2220
3	Complete	100.0%	0 hr 9 min 29 sec	1.0000	0.1243	0.9517	0.0083	46.0938	1.3704	51.2000	1.3593	0.4950
4	Complete	100.0%	0 hr 12 min 21 sec	2.0000	0.1671	0.8082	1.1258e-9	85.1563	0.4302	80.5200	0.5992	0.1966
5	Complete	100.0%	0 hr 15 min 8 sec	3.0000	0.0574	0.9512	6.8631e-7	83.9844	0.4764	81.1600	0.5764	0.1936
6	Complete	100.0%	0 hr 15 min 13 sec	3.0000	0.1855	0.8001	1.8081e-5	85.1563	0.4529	81.3000	0.5839	0.1958
7	Complete	100.0%	0 hr 14 min 48 sec	3.0000	0.1002	0.8030	1.0046e-10	82.8125	0.4798	80.9200	0.5814	0.1916
8	Complete	100.0%	0 hr 12 min 12 sec	2.0000	0.9973	0.9658	0.0071	42.5781	1.5094	41.7600	1.5550	0.5792
9	Complete	100.0%	0 hr 9 min 34 sec	1.0000	0.9925	0.9065	1.5052e-6	82.0313	0.5091	78.8800	0.6726	0.2118
10	Complete	100.0%	0 hr 12 min 14 sec	2.0000	0.5958	0.9632	2.5953e-8	83.5938	0.4502	79.8600	0.6472	0.2100
11	Complete	100.0%	0 hr 12 min 16 sec	2.0000	0.0101	0.8005	1.1831e-6	77.7344	0.6240	74.9600	0.7325	0.2526
12	Complete	100.0%	0 hr 12 min 34 sec	2.0000	0.1305	0.9664	3.0691e-6	83.9844	0.4403	81.6000	0.5652	0.1992
13	Complete	100.0%	0 hr 15 min 18 sec	3.0000	0.9925	0.9677	9.0817e-7	80.0781	0.5765	76.5600	0.7324	0.2332
14	Complete	100.0%	0 hr 12 min 11 sec	2.0000	0.0101	0.8326	1.0679e-10	78.9063	0.5523	76.5200	0.7010	0.2440
15	Complete	100.0%	0 hr 15 min 4 sec	3.0000	0.9884	0.8493	4.3198e-10	85.9375	0.3886	81.0200	0.5862	0.1930
16	Complete	100.0%	0 hr 9 min 49 sec	1.0000	0.4740	0.8330	1.0079e-10	83.5938	0.5134	78.5600	0.6723	0.2096
17	Complete	100.0%	0 hr 15 min 6 sec	3.0000	0.3775	0.8045	6.0781e-7	83.2031	0.4293	80.6400	0.6031	0.1900
18	Complete	100.0%	0 hr 15 min 24 sec	3.0000	0.1079	0.8706	4.2113e-6	85.1563	0.4093	81.8800	0.5519	0.1858
19	Complete	100.0%	0 hr 9 min 54 sec	1.0000	0.1112	0.8034	3.1516e-6	81.6406	0.5374	77.1400	0.6840	0.2306
20	Complete	100.0%	0 hr 15 min 3 sec	3.0000	0.1011	0.9703	6.3725e-9	85.1563	0.4300	80.3800	0.6000	0.1978
21	Complete	100.0%	0 hr 14 min 57 sec	3.0000	0.0398	0.8912	1.4363e-5	83.5938	0.4800	80.8000	0.5727	0.1992
22	Complete	100.0%	0 hr 15 min 4 sec	3.0000	0.2909	0.9339	5.2420e-10	84.7656	0.4154	82.0200	0.5421	0.1756
23	Complete	100.0%	0 hr 15 min 7 sec	3.0000	0.0657	0.9765	1.2460e-10	83.2031	0.4669	81.3000	0.5742	0.1926
24	Complete	100.0%	0 hr 15 min 0 sec	3.0000	0.0102	0.8469	0.0002	79.6875	0.5038	78.5000	0.6499	0.2256
25	Complete	100.0%	0 hr 14 min 55 sec	3.0000	0.1920	0.9663	4.7280e-10	84.3750	0.3878	81.7800	0.5617	0.1862
26	Complete	100.0%	0 hr 15 min 1 sec	3.0000	0.1746	0.8335	1.3693e-6	82.8125	0.4208	81.5000	0.5694	0.1870
27	Complete	100.0%	0 hr 15 min 7 sec	3.0000	0.3198	0.8609	4.8106e-10	86.7188	0.4195	81.4200	0.5763	0.1876
28	Complete	100.0%	0 hr 14 min 52 sec	3.0000	0.2683	0.8485	3.4623e-10	85.5469	0.4083	81.9000	0.5570	0.1862
29	Complete	100.0%	0 hr 14 min 45 sec	3.0000	0.1514	0.9491	2.4234e-6	82.0313	0.4484	82.0000	0.5605	0.1890
30	Complete	100.0%	0 hr 14 min 42 sec	3.0000	0.3042	0.9558	9.2537e-10	85.1563	0.4130	80.4200	0.6042	0.1956

To determine the trial that optimizes the selected metric, Experiment Manager uses the best point criterion `'min-observed'`. For more information, see “Bayesian Optimization Algorithm” (Statistics and Machine Learning Toolbox) and `bestPoint` (Statistics and Machine Learning Toolbox).

Evaluate Results

To test the best trial in your experiment, first select the row in the results table with the lowest error rate.

To display the confusion matrix for the selected trial, click **Confusion Matrix**.



To perform additional computations, export the trained network to the workspace.

1. On the **Experiment Manager** toolstrip, click **Export**.
2. In the dialog window, enter the name of a workspace variable for the exported network. The default name is `trainedNetwork`.
3. Use the exported network as the input to the helper function `testSummary`, which is listed in Appendix 4 at the end of this example. For instance, in the MATLAB Command Window, enter:

```
testSummary(trainedNetwork)
```

This function evaluates the network in several ways:

- It predicts the labels of the entire test set and calculates the test error. Because Experiment Manager determines the best network without exposing the network to the entire test set, the test error can be higher than the value of the custom metric `ErrorRate`.
- It calculates the standard error (`testErrorSE`) and an approximate 95% confidence interval (`testError95CI`) of the generalization error rate by treating the classification of each image in the test set as an independent event with a certain probability of success. Using this assumption, the number of incorrectly classified images follows a binomial distribution. This method is often called the *Wald method*.
- It displays some test images together with their predicted classes and the probabilities of those classes.

The function displays a summary of these statistics in the MATLAB Command Window.

```
****************************************

Test error rate: 0.1776
Standard error: 0.0038
95% confidence interval: [0.1701, 0.1851]

****************************************
```

To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **ErrorRate** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.

For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `augimdsTrain` is an augmented image datastore for the training data.

- `layers` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [augimdsTrain,layers,options] = BayesOptExperiment_setup1(params)

datadir = tempdir;
downloadCIFARData(datadir);

[XTrain,YTrain,XTest,YTest] = loadCIFARData(datadir);
idx = randperm(numel(YTest),5000);
XValidation = XTest(:,:,:,idx);
YValidation = YTest(idx);

imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);
augimdsTrain = augmentedImageDatastore(imageSize,XTrain,YTrain, ...
    DataAugmentation=imageAugmenter);

numClasses = numel(unique(YTrain));
numF = round(16/sqrt(params.SectionDepth));
layers = [
    imageInputLayer(imageSize)

    convBlock(3,numF,params.SectionDepth)
    maxPooling2dLayer(3,Stride=2,Padding="same")

    convBlock(3,2*numF,params.SectionDepth)
    maxPooling2dLayer(3,Stride=2,Padding="same")

    convBlock(3,4*numF,params.SectionDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

miniBatchSize = 256;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions("sgdm", ...
    InitialLearnRate=params.InitialLearnRate, ...
    Momentum=params.Momentum, ...
    MaxEpochs=60, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropPeriod=40, ...
    LearnRateDropFactor=0.1, ...
    MiniBatchSize=miniBatchSize, ...
    L2Regularization=params.L2Regularization, ...
    Shuffle="every-epoch", ...
    Verbose=false, ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=validationFrequency);
```

```
end
```

Appendix 2: Create Block of Convolutional Layers

This function creates a block of `numConvLayers` convolutional layers, each with a specified `filterSize` and `numFilters` filters, and each followed by a batch normalization layer and a ReLU layer.

```
function layers = convBlock(filterSize,numFilters,numConvLayers)
layers = [
    convolution2dLayer(filterSize,numFilters,Padding="same")
    batchNormalizationLayer
    reluLayer];
layers = repmat(layers,numConvLayers,1);
end
```

Appendix 3: Compute Error Rate

This metric function takes as input a structure that contains the fields `trainedNetwork`, `trainingInfo`, and `parameters`.

- `trainedNetwork` is the `SeriesNetwork` object or `DAGNetwork` object returned by the `trainNetwork` function.
- `trainingInfo` is a structure containing the training information returned by the `trainNetwork` function.
- `parameters` is a structure with fields from the hyperparameter table.

The function selects 5000 test images and labels, evaluates the trained network on the test set, calculates the predicted image labels, and calculates the error rate on the test data.

```
function metricOutput = ErrorRate(trialInfo)

datadir = tempdir;
[~,~,XTest,YTest] = loadCIFARData(datadir);

idx = randperm(numel(YTest),5000);
XTest = XTest(:,:,:,idx);
YTest = YTest(idx);

YPredicted = classify(trialInfo.trainedNetwork,XTest);
metricOutput = 1 - mean(YPredicted == YTest);

end
```

Appendix 4: Summarize Test Statistics

This function computes the test error, standard error, and an approximate 95% confidence interval and displays a summary of these statistics in the MATLAB Command Window. The function also some test images together with their predicted classes and the probabilities of those classes.

```
function testSummary(net)

datadir = tempdir;
```

```
[~,~,XTest,YTest] = loadCIFARData(datadir);

[YPredicted,probs] = classify(net,XTest);
testError = 1 - mean(YPredicted == YTest);

NTest = numel(YTest);
testErrorSE = sqrt(testError*(1-testError)/NTest);
testError95CI = [testError - 1.96*testErrorSE, testError + 1.96*testErrorSE];

fprintf('\n*****************************************\n\n');
fprintf('Test error rate: %.4f\n',testError);
fprintf('Standard error: %.4f\n',testErrorSE);
fprintf('95%% confidence interval: [%.4f, %.4f]\n',testError95CI(1),testError95CI(2));
fprintf('\n*****************************************\n\n');

figure
idx = randperm(numel(YTest),9);
for i = 1:numel(idx)
    subplot(3,3,i)
    imshow(XTest(:,:,:,idx(i)));
    prob = num2str(100*max(probs(idx(i),:)),3);
    predClass = char(YPredicted(idx(i)));
    label = [predClass,', ',prob,'%'];
    title(label)
end
end
```

See Also

Apps
Experiment Manager

Functions
`trainNetwork` | `trainingOptions` | `bayesopt` | `bestPoint` | `optimizableVariable`

More About

- “Deep Learning Using Bayesian Optimization” on page 5-99
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Bayesian Optimization Algorithm” (Statistics and Machine Learning Toolbox)

Try Multiple Pretrained Networks for Transfer Learning

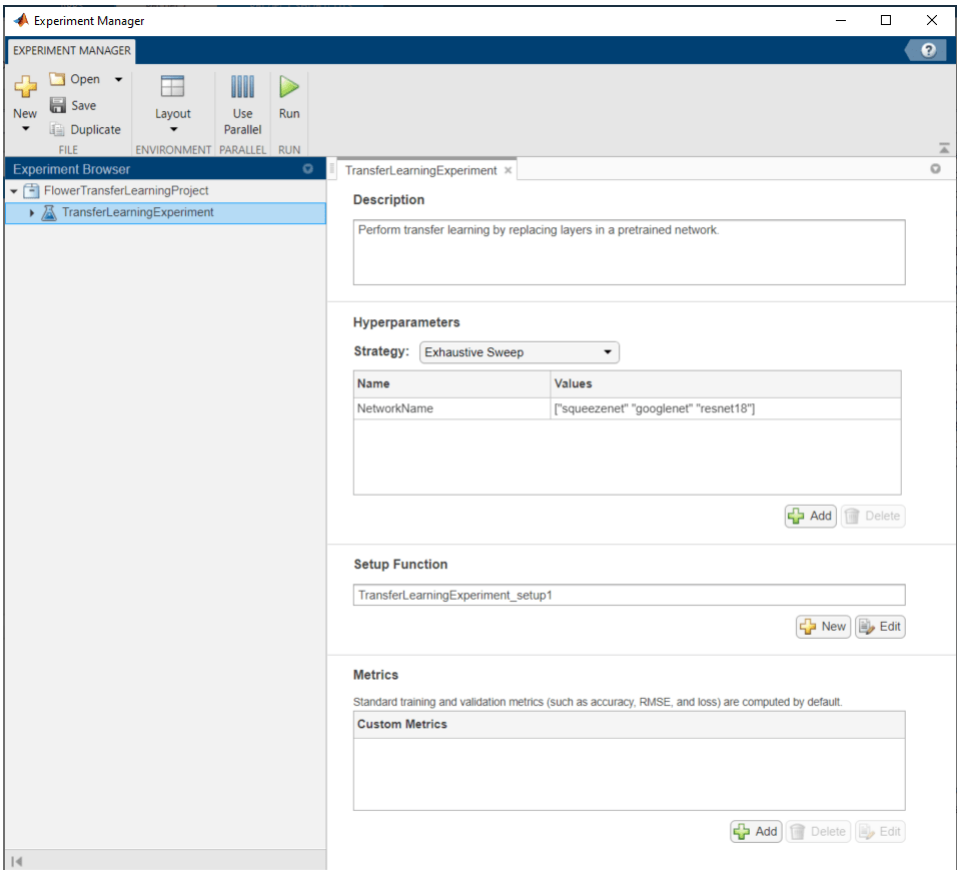
This example shows how to configure an experiment that replaces layers of different pretrained networks for transfer learning. Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

There are many pretrained networks available in Deep Learning Toolbox™. These pretrained networks have different characteristics that matter when choosing a network to apply to your problem. The most important characteristics are network accuracy, speed, and size. Choosing a network is generally a tradeoff between these characteristics. To compare the performance of different pretrained networks for your task, edit this experiment and specify which pretrained networks to use.

This experiment requires the Deep Learning Toolbox Model *for GoogLeNet Network* support package and the Deep Learning Toolbox Model *for ResNet-18 Network* support package. Before you run the experiment, install these support packages by calling the `googlenet` and `resnet18` functions and clicking the download links. For more information on other pretrained networks that you can download from the Add-On Explorer, see “Pretrained Deep Neural Networks” on page 1-8.

Open Experiment

First, open the example. **Experiment Manager** loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`TransferLearningExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Perform transfer learning by replacing layers in a pretrained network.
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. In this example, the hyperparameter `NetworkName` specifies the network to train and the value of the training option `'miniBatchSize'`.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. In this example, the setup function:

- Loads a pretrained network corresponding to the hyperparameter `NetworkName`.

```
networkName = params.NetworkName;

switch networkName
    case "squeezenet"
```

```
        net = squeezenet;
        miniBatchSize = 128;
    case "googlenet"
        net = googlenet;
        miniBatchSize = 128;
    case "resnet18"
        net = resnet18;
        miniBatchSize = 128;
    case "mobilenetv2"
        net = mobilenetv2;
        miniBatchSize = 128;
    case "resnet50"
        net = resnet50;
        miniBatchSize = 128;
    case "resnet101"
        net = resnet101;
        miniBatchSize = 64;
    case "inceptionv3"
        net = inceptionv3;
        miniBatchSize = 64;
    case "inceptionresnetv2"
        net = inceptionresnetv2;
        miniBatchSize = 64;
    otherwise
        error("Undefined network selection.");
end
```

- Downloads and extracts the Flowers data set, which is about 218 MB. For more information on this data set, see “Image Data Sets” on page 19-118.

```
url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flower Dataset (218 MB)...")
    websave(filename,url);
    untar(filename,downloadFolder)
end

imds = imageDatastore(imageFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
inputSize = net.Layers(1).InputSize;
augimdsTrain = augmentedImageDatastore(inputSize,imdsTrain);
augimdsValidation = augmentedImageDatastore(inputSize,imdsValidation);
```

- Replaces the learnable layers of the pretrained network to perform transfer learning. The helper function `findLayersToReplace`, which is listed in Appendix 2 at the end of this example, determines the layers in the network architecture to replace for transfer learning. For more information on the available pretrained networks, see “Pretrained Deep Neural Networks” on page 1-8.

```
lgraph = layerGraph(net);
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
numClasses = numel(categories(imdsTrain.Labels));
```

```
if isa(learnableLayer,"nnet.cnn.layer.FullyConnectedLayer")
    newLearnableLayer = fullyConnectedLayer(numClasses, ...
        Name="new_fc", ...
        WeightLearnRateFactor=10, ...
        BiasLearnRateFactor=10);
elseif isa(learnableLayer,"nnet.cnn.layer.Convolution2DLayer")
    newLearnableLayer = convolution2dLayer(1,numClasses, ...
        Name="new_conv", ...
        WeightLearnRateFactor=10, ...
        BiasLearnRateFactor=10);
end

lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);

newClassLayer = classificationLayer(Name="new_classoutput");
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);
```

- Defines a `trainingOptions` object for the experiment. The example trains the network for 10 epochs, using an initial learning rate of 0.0003 and validating the network every 5 epochs.

```
validationFrequencyEpochs = 5;

numObservations = augimdsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);
validationFrequency = validationFrequencyEpochs * numIterationsPerEpoch;

options = trainingOptions("sgdm", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
    InitialLearnRate=3e-4, ...
    Shuffle="every-epoch", ...
    ValidationData=augimdsValidation, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

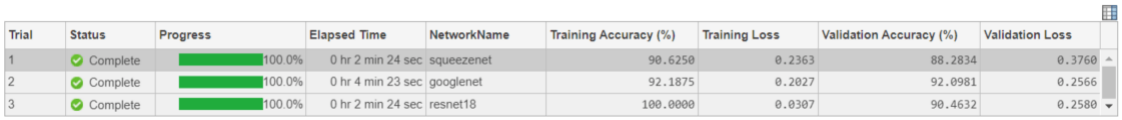
The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any custom metric functions.

Run Experiment

When you run the experiment, Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the accuracy and loss for each trial. While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. Click **Confusion Matrix** to display the confusion matrix for the validation data in each completed trial.



Trial	Status	Progress	Elapsed Time	NetworkName	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete	100.0%	0 hr 2 min 24 sec	squeezenet	90.6250	0.2363	88.2834	0.3760
2	Complete	100.0%	0 hr 4 min 23 sec	googlenet	92.1875	0.2027	92.0981	0.2566
3	Complete	100.0%	0 hr 2 min 24 sec	resnet18	100.0000	0.0307	90.4632	0.2580

When the experiment finishes, you can sort the results table by column, filter trials by using the **Filters** pane, or record observations by adding annotations. For more information, see “Sort, Filter, and Annotate Experiment Results”.

To test the performance of an individual trial, export the trained network or the training information for the trial. On the **Experiment Manager** toolstrip, select **Export** > **Trained Network** or **Export** > **Training Information**, respectively. For more information, see “net” and “info”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `augimdsTrain` is an augmented image datastore for the training data.
- `lgraph` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [augimdsTrain,lgraph,options] = TransferLearningExperiment_setup1(params)

networkName = params.NetworkName;

switch networkName
    case "squeezenet"
        net = squeezenet;
        miniBatchSize = 128;
    case "googlenet"
        net = googlenet;
        miniBatchSize = 128;
    case "resnet18"
        net = resnet18;
        miniBatchSize = 128;
    case "mobilenetv2"
        net = mobilenetv2;
        miniBatchSize = 128;
    case "resnet50"
```

```
        net = resnet50;
        miniBatchSize = 128;
    case "resnet101"
        net = resnet101;
        miniBatchSize = 64;
    case "inceptionv3"
        net = inceptionv3;
        miniBatchSize = 64;
    case "inceptionresnetv2"
        net = inceptionresnetv2;
        miniBatchSize = 64;
    otherwise
        error("Undefined network selection.");
end

url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flower Dataset (218 MB)...")
    websave(filename,url);
    untar(filename,downloadFolder)
end

imds = imageDatastore(imageFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
inputSize = net.Layers(1).InputSize;
augimdsTrain = augmentedImageDatastore(inputSize,imdsTrain);
augimdsValidation = augmentedImageDatastore(inputSize,imdsValidation);

lgraph = layerGraph(net);
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
numClasses = numel(categories(imdsTrain.Labels));

if isa(learnableLayer,"nnet.cnn.layer.FullyConnectedLayer")
    newLearnableLayer = fullyConnectedLayer(numClasses, ...
        Name="new_fc", ...
        WeightLearnRateFactor=10, ...
        BiasLearnRateFactor=10);
elseif isa(learnableLayer,"nnet.cnn.layer.Convolution2DLayer")
    newLearnableLayer = convolution2dLayer(1,numClasses, ...
        Name="new_conv", ...
        WeightLearnRateFactor=10, ...
        BiasLearnRateFactor=10);
end

lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);

newClassLayer = classificationLayer(Name="new_classoutput");
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);

validationFrequencyEpochs = 5;
```

```
numObservations = augimdsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);
validationFrequency = validationFrequencyEpochs * numIterationsPerEpoch;

options = trainingOptions("sgdm", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
    InitialLearnRate=3e-4, ...
    Shuffle="every-epoch", ...
    ValidationData=augimdsValidation, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);

end
```

Appendix 2: Find Layers to Replace

This function finds the single classification layer and the preceding learnable (fully connected or convolutional) layer of the layer graph `lgraph`.

```
function [learnableLayer,classLayer] = findLayersToReplace(lgraph)

if ~isa(lgraph,"nnet.cnn.LayerGraph")
    error("Argument must be a LayerGraph object.")
end

src = string(lgraph.Connections.Source);
dst = string(lgraph.Connections.Destination);
layerNames = string({lgraph.Layers.Name}');

isClassificationLayer = arrayfun(@(l) ...
    (isa(l,"nnet.cnn.layer.ClassificationOutputLayer")|isa(l,"nnet.layer.ClassificationLayer")),
    lgraph.Layers);

if sum(isClassificationLayer) ~= 1
    error("Layer graph must have a single classification layer.")
end
classLayer = lgraph.Layers(isClassificationLayer);

currentLayerIdx = find(isClassificationLayer);
while true

    if numel(currentLayerIdx) ~= 1
        error("Layer graph must have a single learnable layer preceding the classification layer
    end

    currentLayerType = class(lgraph.Layers(currentLayerIdx));
    isLearnableLayer = ismember(currentLayerType, ...
        ["nnet.cnn.layer.FullyConnectedLayer","nnet.cnn.layer.Convolution2DLayer"]);

    if isLearnableLayer
        learnableLayer =  lgraph.Layers(currentLayerIdx);
        return
    end

    currentDstIdx = find(layerNames(currentLayerIdx) == dst);
```

```
    currentLayerIdx = find(src(currentDstIdx) == layerNames);
end
end
```

See Also

Apps
Experiment Manager

Functions
`googlenet` | `resnet18` | `squeezenet` | `trainNetwork` | `trainingOptions`

More About

- “Pretrained Deep Neural Networks” on page 1-8
- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Experiment with Weight Initializers for Transfer Learning” on page 6-44

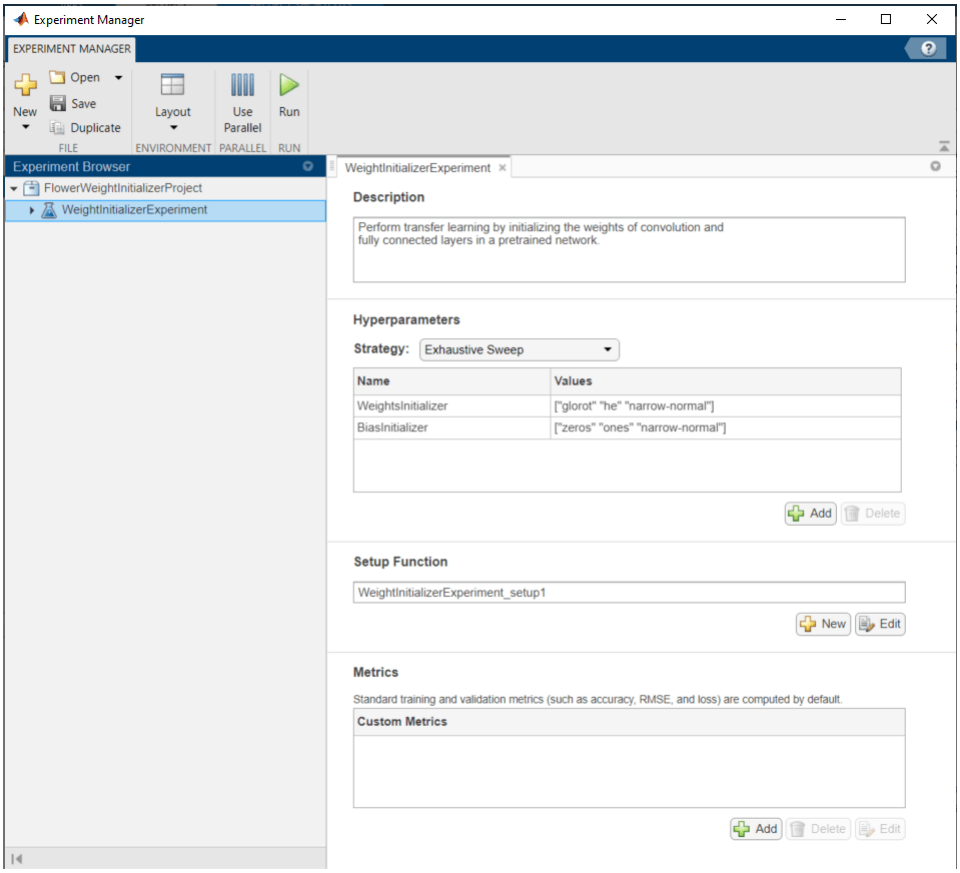
Experiment with Weight Initializers for Transfer Learning

This example shows how to configure an experiment that initializes the weights of convolution and fully connected layers using different weight initializers for training. To compare the performance of different weight initializers for your task, create an experiment using this example as a guide.

When training a deep learning network, the initialization of layer weights and biases can have a big impact on how well the network trains. The choice of initializer has a bigger impact on networks without batch normalization layers. For more information on weight initializers, see “Compare Layer Weight Initializers” on page 18-181.

Open Experiment

First, open the example. **Experiment Manager** loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`WeightInitializerExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Perform transfer learning by initializing the weights of convolution and
fully connected layers in a pretrained network.
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses the hyperparameters `WeightsInitializer` and `BiasInitializer` to specify the weight and bias initializers for the convolution and fully connected layers in a pretrained network. For more information about these initializers, see “`WeightsInitializer`” and “`BiasInitializer`”.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. In this example, the setup function:

- Loads a pretrained GoogLeNet network.

```
lgraph = googlenet(Weights="none");
```

- Downloads and extracts the Flowers data set, which is about 218 MB. For more information on this data set, see “Image Data Sets” on page 19-118.

```
url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flower Dataset (218 MB)...")
    websave(filename,url);
    untar(filename,downloadFolder)
end

imds = imageDatastore(imageFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
inputSize = net.Layers(1).InputSize;
augimdsTrain = augmentedImageDatastore(inputSize,imdsTrain);
augimdsValidation = augmentedImageDatastore(inputSize,imdsValidation);
```

- Initializes the input weight in the convolution and fully connected layers by using the initializers specified in the hyperparameter table. The auxiliary function `findLayersToReplace`, which is listed in Appendix 2 at the end of this example, determines the layers in the network architecture that can be modified for transfer learning.

```
numClasses = numel(categories(imdsTrain.Labels));
weightsInitializer = params.WeightsInitializer;
biasInitializer = params.BiasInitializer;

learnableLayer = findLayersToReplace(lgraph);
newLearnableLayer = fullyConnectedLayer(numClasses,Name="new_fc");
lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);

for i = 1:numel(lgraph.Layers)
    layer = lgraph.Layers(i);
```

```
    if class(layer) == "nnet.cnn.layer.Convolution2DLayer" || ...
            class(layer) == "nnet.cnn.layer.FullyConnectedLayer"
        layerName = layer.Name;
        newLayer = layer;

        newLayer.WeightsInitializer = weightsInitializer;
        newLayer.BiasInitializer = biasInitializer;

        lgraph = replaceLayer(lgraph,layerName,newLayer);
    end
end
```

- Defines a `trainingOptions` object for the experiment. The example trains the network for 10 epochs, using a mini-batch size of 128 and validating the network every 5 epochs.

```
miniBatchSize = 128;
validationFrequencyEpochs = 5;

numObservations = augimdsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);
validationFrequency = validationFrequencyEpochs * numIterationsPerEpoch;

options = trainingOptions("sgdm", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    ValidationData=augimdsValidation, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB® Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any custom metric functions.

Run Experiment

When you run the experiment, Experiment Manager trains the network defined by the setup function multiple times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the accuracy and loss for each trial. While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. Click **Confusion Matrix** to display the confusion matrix for the validation data in each completed trial.

Trial	Status	Progress	Elapsed Time	WeightsInitiali...	BiasInitializer	Training Accuracy (%)	Training Loss	Validation Accuracy (%)	Validation Loss
1	Complete	100.0%	0 hr 5 min 37 sec	glorot	zeros	69.5313	0.8787	67.8474	0.8679
2	Complete	100.0%	0 hr 0 min 37 sec	he	zeros	16.4063	NaN	17.1662	NaN
3	Complete	100.0%	0 hr 5 min 44 sec	narrow-normal	zeros	17.1875	1.6165	24.5232	1.5998
4	Complete	100.0%	0 hr 5 min 30 sec	glorot	ones	26.5625	1.5956	24.5232	1.5998
5	Complete	100.0%	0 hr 0 min 38 sec	he	ones	16.4063	NaN	17.1662	NaN
6	Complete	100.0%	0 hr 5 min 13 sec	narrow-normal	ones	17.1875	1.6168	24.5232	1.5999
7	Complete	100.0%	0 hr 5 min 50 sec	glorot	narrow-normal	59.3750	1.0149	69.4823	0.7866
8	Complete	100.0%	0 hr 0 min 38 sec	he	narrow-normal	21.8750	NaN	17.1662	NaN
9	Complete	100.0%	0 hr 5 min 28 sec	narrow-normal	narrow-normal	26.5625	1.6006	24.5232	1.5998

Note that, for the trials that use the He weight initializer, Experiment Manager interrupts the training because the training and validation loss become undefined after a few iterations. Continuing the training for those trials does not produce any useful results.

When the experiment finishes, you can sort the results table by column, filter trials by using the **Filters** pane, or record observations by adding annotations. For more information, see “Sort, Filter, and Annotate Experiment Results”.

To test the performance of an individual trial, export the trained network or the training information for the trial. On the **Experiment Manager** toolstrip, select **Export** > **Trained Network** or **Export** > **Training Information**, respectively. For more information, see “net” and “info”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `augimdsTrain` is an augmented image datastore object for the training data.
- `lgraph` is a layer graph that defines the neural network architecture.
- `options` is a `trainingOptions` object.

```
function [augimdsTrain,lgraph,options] = WeightInitializerExperiment_setup1(params)

lgraph = googlenet(Weights="none");

url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");
imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    disp("Downloading Flower Dataset (218 MB)...")
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

```
imds = imageDatastore(imageFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
inputSize = lgraph.Layers(1).InputSize;
augimdsTrain = augmentedImageDatastore(inputSize,imdsTrain);
augimdsValidation = augmentedImageDatastore(inputSize,imdsValidation);

numClasses = numel(categories(imdsTrain.Labels));
weightsInitializer = params.WeightsInitializer;
biasInitializer = params.BiasInitializer;
learnableLayer = findLayersToReplace(lgraph);
newLearnableLayer = fullyConnectedLayer(numClasses,Name="new_fc");
lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
for i = 1:numel(lgraph.Layers)
    layer = lgraph.Layers(i);

    if class(layer) == "nnet.cnn.layer.Convolution2DLayer" || ...
            class(layer) == "nnet.cnn.layer.FullyConnectedLayer"
        layerName = layer.Name;
        newLayer = layer;

        newLayer.WeightsInitializer = weightsInitializer;
        newLayer.BiasInitializer = biasInitializer;

        lgraph = replaceLayer(lgraph,layerName,newLayer);
    end
end

miniBatchSize = 128;
validationFrequencyEpochs = 5;
numObservations = augimdsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations/miniBatchSize);
validationFrequency = validationFrequencyEpochs * numIterationsPerEpoch;
options = trainingOptions("sgdm", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    ValidationData=augimdsValidation, ...
    ValidationFrequency=validationFrequency, ...
    Verbose=false);

end
```

Appendix 2: Find Layers to Replace

This function finds the single classification layer and the preceding learnable (fully connected or convolutional) layer of the layer graph `lgraph`.

```
function [learnableLayer,classLayer] = findLayersToReplace(lgraph)

if ~isa(lgraph,"nnet.cnn.LayerGraph")
    error("Argument must be a LayerGraph object.")
end

src = string(lgraph.Connections.Source);
```

```
dst = string(lgraph.Connections.Destination);
layerNames = string({lgraph.Layers.Name}');

isClassificationLayer = arrayfun(@(l) ...
    (isa(l,"nnet.cnn.layer.ClassificationOutputLayer")|isa(l,"nnet.layer.ClassificationLayer")),
    lgraph.Layers);

if sum(isClassificationLayer) ~= 1
    error("Layer graph must have a single classification layer.")
end
classLayer = lgraph.Layers(isClassificationLayer);

currentLayerIdx = find(isClassificationLayer);
while true

    if numel(currentLayerIdx) ~= 1
        error("Layer graph must have a single learnable layer preceding the classification layer
    end

    currentLayerType = class(lgraph.Layers(currentLayerIdx));
    isLearnableLayer = ismember(currentLayerType, ...
        ["nnet.cnn.layer.FullyConnectedLayer","nnet.cnn.layer.Convolution2DLayer"]);

    if isLearnableLayer
        learnableLayer =  lgraph.Layers(currentLayerIdx);
        return
    end

    currentDstIdx = find(layerNames(currentLayerIdx) == dst);
    currentLayerIdx = find(src(currentDstIdx) == layerNames);
end
end
```

See Also

Apps
Experiment Manager

Functions
`convolution2dLayer` | `fullyConnectedLayer` | `trainNetwork` | `trainingOptions`

More About

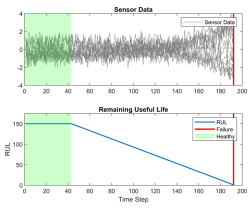
- “Compare Layer Weight Initializers” on page 18-181
- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Try Multiple Pretrained Networks for Transfer Learning” on page 6-36

Choose Training Configurations for LSTM Using Bayesian Optimization

This example shows how to create a deep learning experiment to find optimal network hyperparameters and training options for long short-term memory (LSTM) networks using Bayesian optimization. In this example, you use **Experiment Manager** to train LSTM networks that predict the remaining useful life (RUL) of engines. The experiment uses the Turbofan Engine Degradation Simulation Data Set described in [1]. For more information on processing this data set for sequence-to-sequence regression, see “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47.

Bayesian optimization provides an alternative strategy to sweeping hyperparameters in an experiment. You specify a range of values for each hyperparameter and select a metric to optimize, and Experiment Manager searches for a combination of hyperparameters that optimizes your selected metric. Bayesian optimization requires Statistics and Machine Learning Toolbox™. For more information, see “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26.

RUL captures how many operational cycles an engine can make before failure. To focus on the sequence data from when the engines are close to failing, preprocess the data by clipping the responses at a specified threshold. This preprocessing operation allows the network to focus on predictor data behaviors close to failing by treating instances with higher RUL values as equal. For example, this figure shows the first response observation and the corresponding clipped response with a threshold of 150.

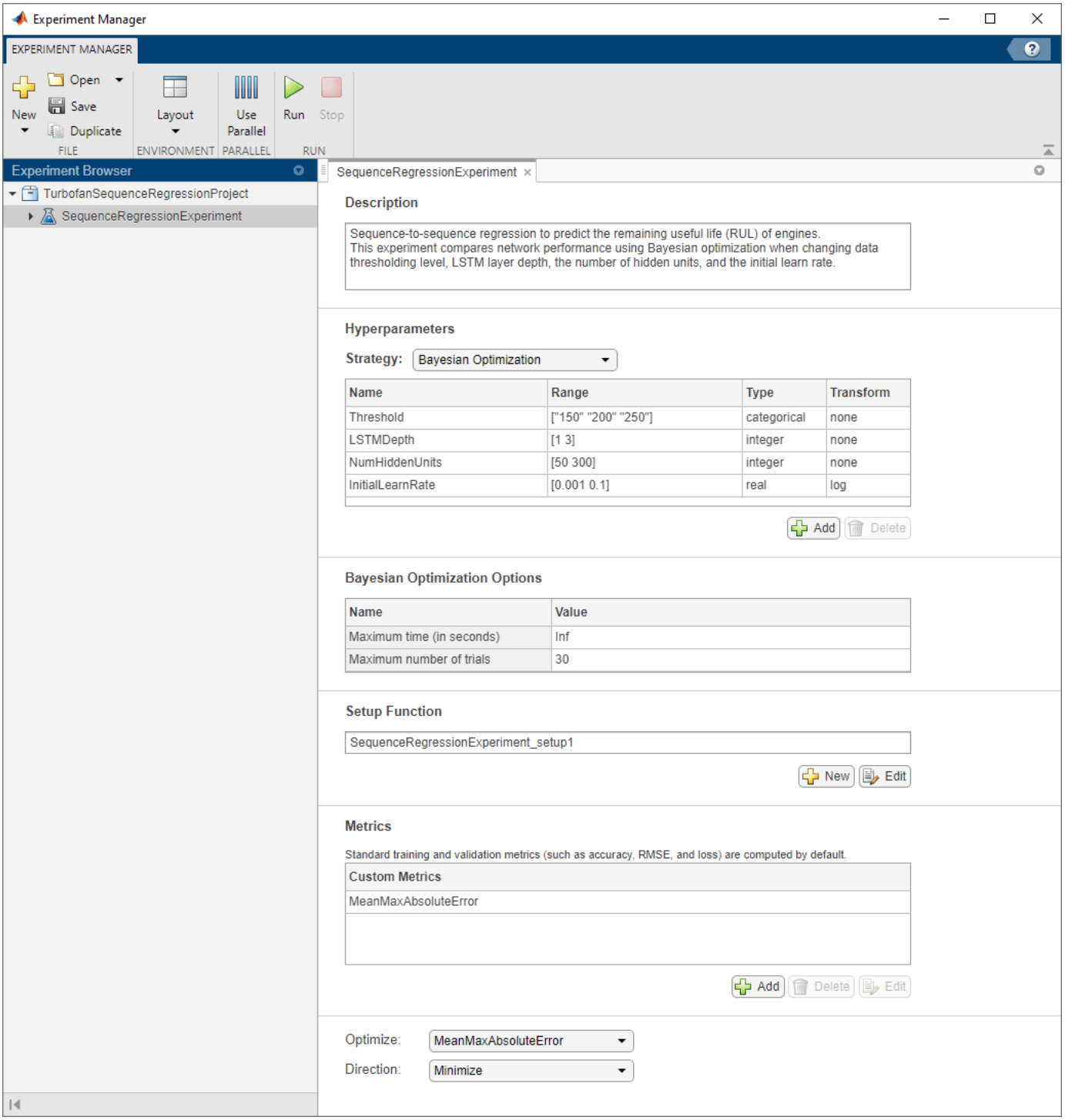


When you train a deep learning network, how you preprocess data, the number of layers and hidden units, and the initial learning rate in the network can affect the training behavior and performance of the network. Choosing the depth of an LSTM network involves balancing speed and accuracy. For example, deeper networks can be more accurate but take longer to train and converge [2].

By default, when you run a built-in training experiment for regression, Experiment Manager computes the loss and root mean squared error (RMSE) for each trial in your experiment. This example compares the performance of the network in each trial by using a custom metric that is specific to the problem data set. For more information on using custom metric functions, see “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment. To open the experiment, in the **Experiment Browser**, double-click the name of the experiment (`SequenceRegressionExperiment`).



Built-in training experiments consist of a description, a table of hyperparameters, a setup function, and a collection of metric functions to evaluate the results of the experiment. Experiments that use Bayesian optimization include additional options to limit the duration of the experiment. For more information, see “Configure Built-In Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Sequence-to-sequence regression to predict the remaining useful life (RUL) of engines.
This experiment compares network performance using Bayesian optimization when changing data
thresholding level, LSTM layer depth, the number of hidden units, and the initial learn rate.
```

The **Hyperparameter Table** specifies the strategy (`Bayesian Optimization`) and hyperparameter values to use for the experiment. For each hyperparameter, specify these options:

- **Range** — Enter a two-element vector that gives the lower bound and upper bound of a real- or integer-valued hyperparameter, or a string array or cell array that lists the possible values of a categorical hyperparameter.
- **Type** — Select `real` (real-valued hyperparameter), `integer` (integer-valued hyperparameter), or `categorical` (categorical hyperparameter).
- **Transform** — Select `none` (no transform) or `log` (logarithmic transform). For `log`, the hyperparameter must be `real` or `integer` and positive. With this option, the hyperparameter is searched and modeled on a logarithmic scale.

When you run the experiment, Experiment Manager searches for the best combination of hyperparameters. Each trial uses a new combination of the hyperparameter values based on the results of the previous trials. This example uses these hyperparameters:

- `Threshold` sets all response data above the threshold value to be equal to the threshold value. To prevent uniform response data, use threshold values greater or equal to 150. To limit the set of allowable values to 150, 200 and 250, the experiment models `Threshold` as a categorical hyperparameter.
- `LSTMDepth` indicates the number of LSTM layers used in the network. Specify this hyperparameter as an integer between 1 and 3.
- `NumHiddenUnits` determines the number of hidden units, or the amount of information stored at each time step, used in the network. Increasing the number of hidden units can result in overfitting the data and in a longer training time. Decreasing the number of hidden units can result in underfitting the data. Specify this hyperparameter as an integer between 50 and 300.
- `InitialLearnRate` specifies the initial learning rate used for training. If the learning rate is too low, then training takes a long time. If the learning rate is too high, then training can reach a suboptimal result or diverge. The best learning rate depends on your data as well as the network you are training. The experiment models this hyperparameter on a logarithmic scale because the range of values (0.001 to 0.1) spans several orders of magnitude.

Under **Bayesian Optimization Options**, you can specify the duration of the experiment by entering the maximum time (in seconds) and the maximum number of trials to run. To best use the power of Bayesian optimization, perform at least 30 objective function evaluations.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. The input to the setup function is a structure with fields from the hyperparameter table. The setup function returns four outputs that you use to train a network for image regression problems. In this example, the setup function has three sections.

- **Load and Preprocess Data** downloads and extracts the Turbofan Engine Degradation Simulation Data Set from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/ [3]. This section of the setup function also filters out constant valued features, normalizes the predictor data to have zero mean and unit variance, clips the response data by using the numerical value of the hyperparameter `Threshold`, and randomly selects training examples to use for validation.

```
dataFolder = fullfile(tempdir,"turbofan");

if ~exist(dataFolder,"dir")
    mkdir(dataFolder);
    oldDir = cd(dataFolder);
    filename = "CMAPSSData.zip";
    websave(filename,"https://ti.arc.nasa.gov/c/6/", ...
        weboptions("Timeout",Inf));
    unzip(filename,dataFolder);
    cd(oldDir);
end

filenameTrainPredictors = fullfile(dataFolder,"train_FD001.txt");
[XTrain,YTrain] = processTurboFanDataTrain(filenameTrainPredictors);

XTrain = helperFilter(XTrain);
XTrain = helperNormalize(XTrain);

thr = str2double(params.Threshold);
for i = 1:numel(YTrain)
    YTrain{i}(YTrain{i} > thr) = thr;
end

for i=1:numel(XTrain)
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end

[~,idx] = sort(sequenceLengths,"descend");
XTrain = XTrain(idx);
YTrain = YTrain(idx);

idx = randperm(numel(XTrain),10);
XValidation = XTrain(idx);
XTrain(idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];
```

- **Define Network Architecture** defines the architecture for an LSTM network for sequence-to-sequence regression. The network consists of LSTM layers followed by a fully connected layer of size 100 and a dropout layer with a dropout probability of 0.5. The hyperparameters `LSTMDepth` and `NumHiddenUnits` specify the number of LSTM layers and the number of hidden units for each layer.

```
numResponses = size(YTrain{1},1);
featureDimension = size(XTrain{1},1);
LSTMDepth = params.LSTMDepth;
numHiddenUnits = params.NumHiddenUnits;

layers = sequenceInputLayer(featureDimension);

for i = 1:LSTMDepth
    layers = [layers;lstmLayer(numHiddenUnits,OutputMode="sequence")];
end

layers = [layers
    fullyConnectedLayer(100)
    reluLayer()
    dropoutLayer(0.5)
```

```
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

- **Specify Training Options** defines the training options for the experiment. Because deeper networks take longer to converge, the number of epochs is set to 300 to ensure all network depths converge. This example validates the network every 30 iterations. The initial learning rate equals the `InitialLearnRate` value from the hyperparameter table and drops by a factor of 0.2 every 15 epochs. With the training option `ExecutionEnvironment` set to `"auto"`, the experiment runs on a GPU if one is available. Otherwise, Experiment Manager uses the CPU. Because this example compares network depths and trains for many epochs, using a GPU speeds up training time considerably. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
maxEpochs = 300;
miniBatchSize = 20;

options = trainingOptions("adam", ...
    ExecutionEnvironment="auto", ...
    MaxEpochs=maxEpochs, ...
    MiniBatchSize=miniBatchSize, ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=30, ...
    InitialLearnRate=params.InitialLearnRate, ...
    LearnRateDropFactor=0.2, ...
    LearnRateDropPeriod=15, ...
    GradientThreshold=1, ...
    Shuffle="never", ...
    Verbose=false);
```

To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. In addition, the code for the setup function appears in Appendix 1 at the end of this example.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. Experiment Manager evaluates these functions each time it finishes training the network. To inspect a metric function, select the name of the metric function and click **Edit**. The metric function opens in MATLAB Editor.

The prediction of the RUL of an engine requires careful consideration. If the prediction underestimates the RUL, engine maintenance might be scheduled before it is necessary. If the prediction overestimates the RUL, the engine might fail while in operation, resulting in high costs or safety concerns. To help mitigate these scenarios, this example includes a metric function `MeanMaxAbsoluteError` that identifies networks that underpredict or overpredict the RUL.

The `MeanMaxAbsoluteError` metric calculates the maximum absolute error, averaged across the entire training set. This metric calls the `predict` function to make a sequence of RUL predictions from the training set. Then, after calculating the maximum absolute error between each training response and predicted response sequence, the function computes the mean of all maximum absolute errors. This metric identifies the maximum deviations between the actual and predicted responses. The code for the metric function appears in Appendix 3 at the end of this example.

Run Experiment

When you run the experiment, Experiment Manager searches for the best combination of hyperparameters with respect to the chosen metric. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials. By default,

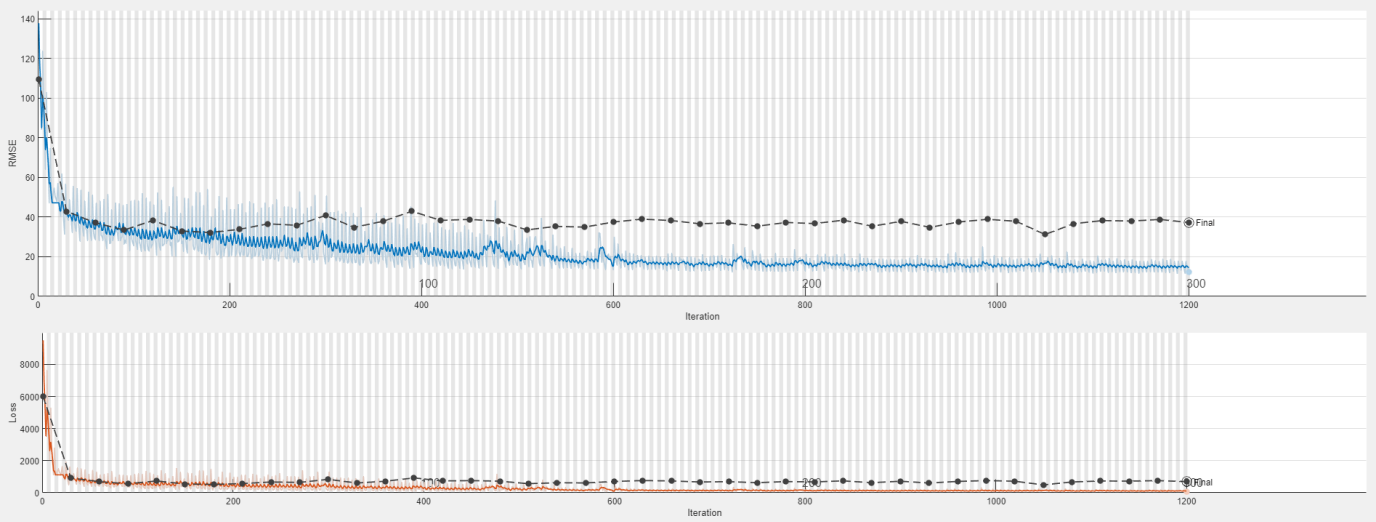
Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox™, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16 and “GPU Support by Release” (Parallel Computing Toolbox).

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the metric function values for each trial. Experiment Manager indicates the trial with the optimal value for the selected metric. For example, in this experiment, the 17th trial produces the smallest maximum absolute error.

Trial	Status	Progress	Elapsed Time	Threshold	LSTMDepth	NumHiddenUnits	InitialLearnRate	Training RMSE	Training Loss	Validation RMSE	Validation Loss	MeanMaxAbsoluteError
1	Complete	100.0%	0 hr 1 min 47 sec	250	3.0000	125.0000	0.0151	13.9260	96.9669	40.1059	804.2435	32.5122
2	Complete	100.0%	0 hr 0 min 55 sec	150	1.0000	211.0000	0.0035	18.0602	163.0860	25.8858	335.0379	81.8949
3	Complete	100.0%	0 hr 0 min 47 sec	200	1.0000	125.0000	0.0141	12.5286	78.4834	35.0976	615.9197	42.1444
4	Complete	100.0%	0 hr 1 min 40 sec	150	3.0000	67.0000	0.0690	20.3099	206.2463	26.4691	350.3077	75.5628
5	Complete	100.0%	0 hr 1 min 37 sec	250	2.0000	300.0000	0.0010	20.7612	215.5135	53.2138	1415.8561	82.6525
6	Complete	100.0%	0 hr 1 min 44 sec	250	3.0000	123.0000	0.0226	14.0104	98.1461	43.1816	932.3272	46.1874
7	Complete	100.0%	0 hr 1 min 15 sec	250	2.0000	127.0000	0.0103	13.0978	85.7760	42.1500	888.3116	32.7435
8	Complete	100.0%	0 hr 1 min 18 sec	250	2.0000	110.0000	0.0116	11.6478	67.8361	44.6151	995.2548	33.2170
9	Complete	100.0%	0 hr 2 min 13 sec	250	3.0000	290.0000	0.0103	25.1023	315.0637	53.4892	1430.5476	70.0437
10	Complete	100.0%	0 hr 1 min 17 sec	250	2.0000	151.0000	0.0129	15.0748	113.6254	43.4740	944.9951	37.7478
11	Complete	100.0%	0 hr 1 min 13 sec	200	2.0000	72.0000	0.0088	12.2400	74.9093	33.2967	554.3367	37.8605
12	Complete	100.0%	0 hr 1 min 39 sec	250	3.0000	61.0000	0.0090	13.3932	89.6892	53.7582	1444.9746	48.1162
13	Complete	100.0%	0 hr 0 min 46 sec	200	1.0000	50.0000	0.0180	12.0520	72.6257	33.5021	561.1965	47.9712
14	Complete	100.0%	0 hr 0 min 47 sec	200	1.0000	123.0000	0.0052	13.9904	97.8656	34.8887	608.6109	54.8624
15	Complete	100.0%	0 hr 1 min 16 sec	250	2.0000	123.0000	0.0126	14.2698	101.8134	41.5512	863.2503	33.0139
16	Complete	100.0%	0 hr 1 min 44 sec	250	3.0000	123.0000	0.0112	14.1878	100.6467	41.3411	854.5445	36.5277
17	Complete	100.0%	0 hr 1 min 14 sec	250	2.0000	115.0000	0.0132	12.1981	74.3966	37.2713	694.5749	27.9796
18	Complete	100.0%	0 hr 0 min 47 sec	250	1.0000	108.0000	0.0139	12.2210	74.6763	37.6666	709.3852	36.5185
19	Complete	100.0%	0 hr 1 min 16 sec	250	2.0000	110.0000	0.0021	23.3426	272.4373	47.1659	1112.3112	88.6509
20	Complete	100.0%	0 hr 1 min 15 sec	250	2.0000	105.0000	0.0135	13.8325	95.6690	38.8315	753.9423	32.4458
21	Complete	100.0%	0 hr 2 min 12 sec	200	3.0000	300.0000	0.0957	55.5013	1540.1990	78.2703	3063.1238	120.2193
22	Complete	100.0%	0 hr 1 min 43 sec	200	3.0000	103.0000	0.0116	14.9455	111.6840	33.0162	545.0359	41.4121
23	Complete	100.0%	0 hr 1 min 36 sec	250	2.0000	300.0000	0.0986	43.6283	951.7157	60.7718	1846.6049	100.2237
24	Complete	100.0%	0 hr 0 min 54 sec	200	1.0000	299.0000	0.0010	20.2781	205.6017	40.7736	831.2427	91.0674
25	Complete	100.0%	0 hr 1 min 33 sec	150	2.0000	298.0000	0.0978	30.6614	470.0586	32.2073	518.6553	105.5086
26	Complete	100.0%	0 hr 1 min 36 sec	150	3.0000	51.0000	0.0010	18.5255	171.5966	14.8591	110.3971	98.7727
27	Complete	100.0%	0 hr 1 min 10 sec	200	2.0000	51.0000	0.0010	20.5920	212.0153	44.2580	979.3835	99.5893
28	Complete	100.0%	0 hr 1 min 37 sec	150	3.0000	50.0000	0.0116	12.3040	75.6946	25.2074	317.7057	64.2568
29	Complete	100.0%	0 hr 1 min 37 sec	200	3.0000	51.0000	0.0994	39.3337	773.5708	39.2550	770.4790	76.4324
30	Complete	100.0%	0 hr 1 min 35 sec	150	2.0000	300.0000	0.0163	14.0525	98.7364	25.3989	322.5515	66.7313

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. The elapsed time for a trial to complete training increases with network depth.



Evaluate Results

In the table of results, the **MeanMaxAbsoluteError** value quantifies how much the network underpredicts or overpredicts the RUL. The **Validation RMSE** value quantifies how well the network generalizes to unseen data. To find the best result for your experiment, sort the table of results and select the trial that has the lowest **MeanMaxAbsoluteError** and **Validation RMSE** values.

1. Point to the **MeanMaxAbsoluteError** column.
2. Click the triangle icon.
3. Select **Sort in Ascending Order**.

Trial	Status	Progress	Elapsed Time	Threshold	LSTMDepth	NumHiddenUnits	InitialLearnRate	Training RMSE	Training Loss	Validation RMSE	Validation Loss	MeanMaxAbsoluteError
17	Complete	100.0%	0 hr 1 min 14 sec	250	2.0000	115.0000	0.0132	12.1981	74.3966	37.2713	694.5749	27.9796
20	Complete	100.0%	0 hr 1 min 15 sec	250	2.0000	105.0000	0.0135	13.8325	95.6690	38.8315	753.9423	32.4458
1	Complete	100.0%	0 hr 1 min 47 sec	250	3.0000	125.0000	0.0151	13.9260	96.9669	40.1059	804.2435	32.5122
7	Complete	100.0%	0 hr 1 min 15 sec	250	2.0000	127.0000	0.0103	13.0978	85.7760	42.1500	888.3116	32.7435
15	Complete	100.0%	0 hr 1 min 16 sec	250	2.0000	123.0000	0.0126	14.2698	101.8134	41.5512	863.2503	33.0139
8	Complete	100.0%	0 hr 1 min 18 sec	250	2.0000	110.0000	0.0116	11.6478	67.8361	44.6151	995.2548	33.2170
18	Complete	100.0%	0 hr 0 min 47 sec	250	1.0000	108.0000	0.0139	12.2210	74.6763	37.6666	709.3852	36.5185
16	Complete	100.0%	0 hr 1 min 44 sec	250	3.0000	123.0000	0.0112	14.1878	100.6467	41.3411	854.5445	36.5277
10	Complete	100.0%	0 hr 1 min 17 sec	250	2.0000	151.0000	0.0129	15.0748	113.6254	43.4740	944.9951	37.7478
11	Complete	100.0%	0 hr 1 min 13 sec	200	2.0000	72.0000	0.0088	12.2400	74.9093	33.2967	554.3367	37.8605
22	Complete	100.0%	0 hr 1 min 43 sec	200	3.0000	103.0000	0.0116	14.9455	111.6840	33.0162	545.0359	41.4121
3	Complete	100.0%	0 hr 0 min 47 sec	200	1.0000	125.0000	0.0141	12.5286	78.4834	35.0976	615.9197	42.1444

Similarly, find the trial with the smallest validation RMSE by opening the drop-down menu for the **Validation RMSE** column and selecting **Sort in Ascending Order**.

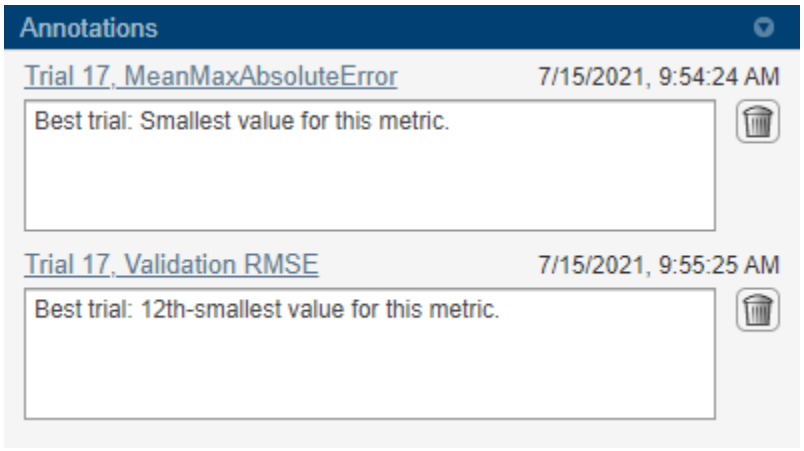
Trial	Status	Progress	Elapsed Time	Threshold	LSTMDepth	NumHiddenUnits	InitialLearnRate	Training RMSE	Training Loss	Validation RMSE	Validation Loss	MeanMaxAbsoluteError
26	Complete	100.0%	0 hr 1 min 36 sec	150	3.0000	51.0000	0.0010	18.5255	171.5966	14.8591	110.3971	98.7727
28	Complete	100.0%	0 hr 1 min 37 sec	150	3.0000	50.0000	0.0116	12.3040	75.6946	25.2074	317.7057	64.2568
30	Complete	100.0%	0 hr 1 min 35 sec	150	2.0000	300.0000	0.0163	14.0525	98.7364	25.3989	322.5515	66.7313
2	Complete	100.0%	0 hr 0 min 55 sec	150	1.0000	211.0000	0.0035	18.0602	163.0860	25.8858	335.0379	81.8949
4	Complete	100.0%	0 hr 1 min 40 sec	150	3.0000	67.0000	0.0690	20.3099	206.2463	26.4691	350.3077	75.5628
25	Complete	100.0%	0 hr 1 min 33 sec	150	2.0000	298.0000	0.0978	30.6614	470.0586	32.2073	518.6553	105.5086
22	Complete	100.0%	0 hr 1 min 43 sec	200	3.0000	103.0000	0.0116	14.9455	111.6840	33.0162	545.0359	41.4121
11	Complete	100.0%	0 hr 1 min 13 sec	200	2.0000	72.0000	0.0088	12.2400	74.9093	33.2967	554.3367	37.8605
13	Complete	100.0%	0 hr 0 min 46 sec	200	1.0000	50.0000	0.0180	12.0520	72.6257	33.5021	561.1965	47.9712
14	Complete	100.0%	0 hr 0 min 47 sec	200	1.0000	123.0000	0.0052	13.9904	97.8656	34.8887	608.6109	54.8624
3	Complete	100.0%	0 hr 0 min 47 sec	200	1.0000	125.0000	0.0141	12.5286	78.4834	35.0976	615.9197	42.1444
17	Complete	100.0%	0 hr 1 min 14 sec	250	2.0000	115.0000	0.0132	12.1981	74.3966	37.2713	694.5749	27.9796

If no single trial minimizes both values, consider giving preference to a trial that ranks well for each value. For instance, in these results, trial 17 has the smallest **MeanMaxAbsoluteError** value and the

12th-smallest **Validation RMSE** value. Trials 3, 11, and 22 have smaller **Validation RMSE** values but larger **MeanMaxAbsoluteError** values.

To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **MeanMaxAbsoluteError** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.
4. Repeat the previous steps for the **Validation RMSE** cell.

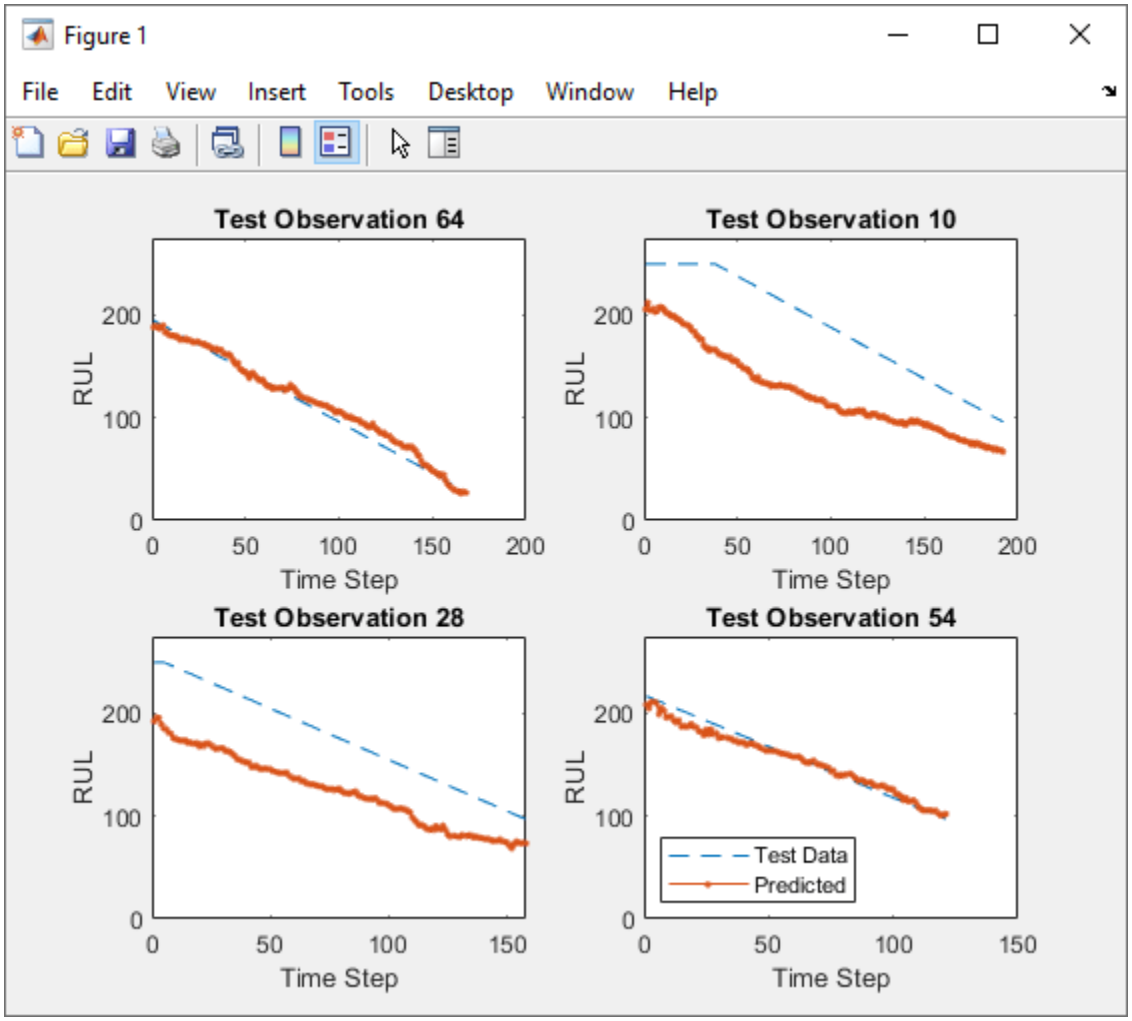


To test the performance of your best trial, export the trained network and display the predicted response sequence for several randomly chosen test sequences.

1. Select the best trial in your experiment.
2. On the **Experiment Manager** toolstrip, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported network. The default name is `trainedNetwork`.
4. Use the exported network and the `Threshold` value of the network as inputs to the helper function `plotSequences`, which is listed in Appendix 4 at the end of this example. For instance, in the MATLAB Command Window, enter:

```
plotSequences(trainedNetwork,250)
```

The function plots the true and predicted response sequences of unseen test data.



Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Setup Function

This function configures the training data, network architecture, and training options for the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.

Output

- `XTrain` is a cell array containing the training data.
- `YTrain` is a cell array containing the regression values for training,
- `layers` is a layer graph that defines the neural network architecture.

- `options` is a `trainingOptions` object.

```
function [XTrain,YTrain,layers,options] = SequenceRegressionExperiment_setup1(params)

dataFolder = fullfile(tempdir,"turbofan");

if ~exist(dataFolder,"dir")
    mkdir(dataFolder);
    oldDir = cd(dataFolder);
    filename = "CMAPSSData.zip";
    websave(filename,"https://ti.arc.nasa.gov/c/6/", ...
        weboptions("Timeout",Inf));
    unzip(filename,dataFolder);
    cd(oldDir);
end

filenameTrainPredictors = fullfile(dataFolder,"train_FD001.txt");
[XTrain,YTrain] = processTurboFanDataTrain(filenameTrainPredictors);

XTrain = helperFilter(XTrain);
XTrain = helperNormalize(XTrain);

thr = str2double(params.Threshold);
for i = 1:numel(YTrain)
    YTrain{i}(YTrain{i} > thr) = thr;
end

for i=1:numel(XTrain)
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end

[~,idx] = sort(sequenceLengths,"descend");
XTrain = XTrain(idx);
YTrain = YTrain(idx);

idx = randperm(numel(XTrain),10);
XValidation = XTrain(idx);
XTrain(idx) = [];
YValidation = YTrain(idx);
YTrain(idx) = [];

numResponses = size(YTrain{1},1);
featureDimension = size(XTrain{1},1);
LSTMDepth = params.LSTMDepth;
numHiddenUnits = params.NumHiddenUnits;

layers = sequenceInputLayer(featureDimension);

for i = 1:LSTMDepth
    layers = [layers;lstmLayer(numHiddenUnits,OutputMode="sequence")];
end

layers = [layers
    fullyConnectedLayer(100)
    reluLayer()
    dropoutLayer(0.5)
```

```
    fullyConnectedLayer(numResponses)
    regressionLayer];

maxEpochs = 300;
miniBatchSize = 20;

options = trainingOptions("adam", ...
    ExecutionEnvironment="auto", ...
    MaxEpochs=maxEpochs, ...
    MiniBatchSize=miniBatchSize, ...
    ValidationData={XValidation,YValidation}, ...
    ValidationFrequency=30, ...
    InitialLearnRate=params.InitialLearnRate, ...
    LearnRateDropFactor=0.2, ...
    LearnRateDropPeriod=15, ...
    GradientThreshold=1, ...
    Shuffle="never", ...
    Verbose=false);

end
```

Appendix 2: Filter and Normalize Predictive Maintenance Data

The helper function `helperFilter` filters the data by removing features with constant values. Features that remain constant for all time steps can negatively impact the training.

```
function [XTrain,XTest] = helperFilter(XTrain,XTest)
m = min([XTrain{:}],[],2);
M = max([XTrain{:}],[],2);
idxConstant = M == m;

for i = 1:numel(XTrain)
    XTrain{i}(idxConstant,:) = [];
    if nargin>1
        XTest{i}(idxConstant,:) = [];
    end
end
end
```

The helper function `helperNormalize` normalizes the training and test predictors to have zero mean and unit variance.

```
function [XTrain,XTest] = helperNormalize(XTrain,XTest)
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);

for i = 1:numel(XTrain)
    XTrain{i} = (XTrain{i} - mu) ./ sig;
    if nargin>1
        XTest{i} = (XTest{i} - mu) ./ sig;
    end
end
end
```

Appendix 3: Compute Mean of Maximum Absolute Errors

This metric function calculates the maximum absolute error of the trained network, averaged over the training set.

```
function metricOutput = MeanMaxAbsoluteError(trialInfo)

net = trialInfo.trainedNetwork;
thr = str2double(trialInfo.parameters.Threshold);

filenamePredictors = fullfile(tempdir,"turbofan","train_FD001.txt");
[XTrain,YTrain] = processTurboFanDataTrain(filenamePredictors);
XTrain = helperFilter(XTrain);
XTrain = helperNormalize(XTrain);

for i = 1:numel(YTrain)
    YTrain{i}(YTrain{i} > thr) = thr;
end

YPred = predict(net,XTrain,MiniBatchSize=1);

maxAbsErrors = zeros(1,numel(YTrain));
for i=1:numel(YTrain)
    absError = abs(YTrain{i}-YPred{i});
    maxAbsErrors(i) = max(absError);
end

metricOutput = mean(maxAbsErrors);

end
```

Appendix 4: Plot Predictive Maintenance Sequences

This function plots the true and predicted response sequences to allow you to evaluate the performance of your trained network. This function uses the helper functions `helperFilter` and `helperNormalize`, which are listed in Appendix 2.

```
function plotSequences(net,threshold)

filenameTrainPredictors = fullfile(tempdir,"turbofan","train_FD001.txt");
filenameTestPredictors = fullfile(tempdir,"turbofan","test_FD001.txt");
filenameTestResponses = fullfile(tempdir,"turbofan","RUL_FD001.txt");

[XTrain,YTrain] = processTurboFanDataTrain(filenameTrainPredictors);
[XTest,YTest] = processTurboFanDataTest(filenameTestPredictors,filenameTestResponses);

[XTrain,XTest] = helperFilter(XTrain,XTest);
[~,XTest] = helperNormalize(XTrain,XTest);

for i = 1:numel(YTrain)
  YTrain{i}(YTrain{i} > threshold) = threshold;
  YTest{i}(YTest{i} > threshold) = threshold;
end

YPred = predict(net,XTest,MiniBatchSize=1);

idx = randperm(100,4);
figure
for i = 1:numel(idx)
    subplot(2,2,i)
```

```
    plot(YTest{idx(i)},'--')
    hold on
    plot(YPred{idx(i)},'.-')
    hold off
    ylim([0 threshold+25])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southwest')
end
```

References

1. Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage Propagation Modeling for Aircraft Engine Run-to-Failure Simulation." *2008 International Conference on Prognostics and Health Management* (2008): 1–9.
2. Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. "An Empirical Exploration of Recurrent Network Architectures." *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2342–2350.
3. Saxena, Abhinav, Kai Goebel. "Turbofan Engine Degradation Simulation Data Set." *NASA Ames Prognostics Data Repository*, https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/, NASA Ames Research Center, Moffett Field, CA.

See Also

Apps
Experiment Manager

Functions
`predict` | `trainNetwork` | `trainingOptions`

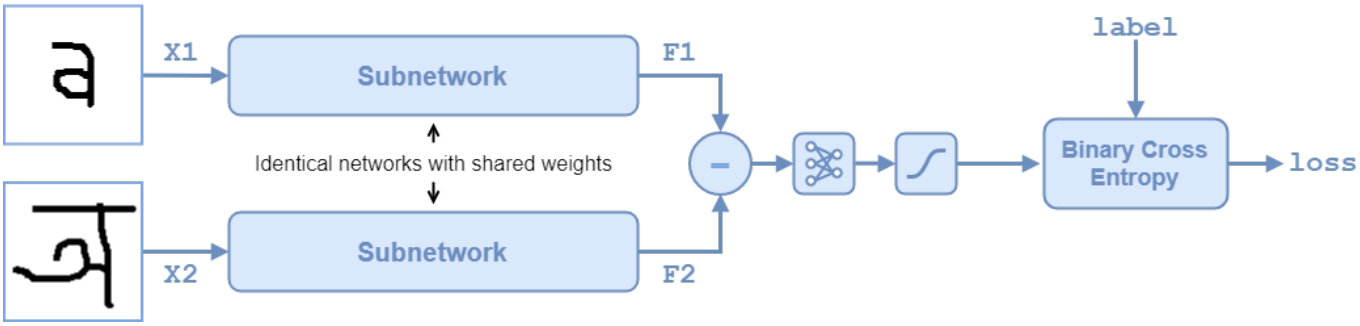
More About

- “Long Short-Term Memory Networks” on page 1-75
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-19

Run a Custom Training Experiment for Image Comparison

This example shows how to create a custom training experiment to train a Siamese network that identifies similar images of handwritten characters. For a custom training experiment, you explicitly define the training procedure used by **Experiment Manager**. In this example, you implement a custom training loop to train a Siamese network, a type of deep learning network that uses two or more identical subnetworks that have the same architecture and share the same parameters and weights. Some common applications for Siamese networks include facial recognition, signature verification, and paraphrase identification.

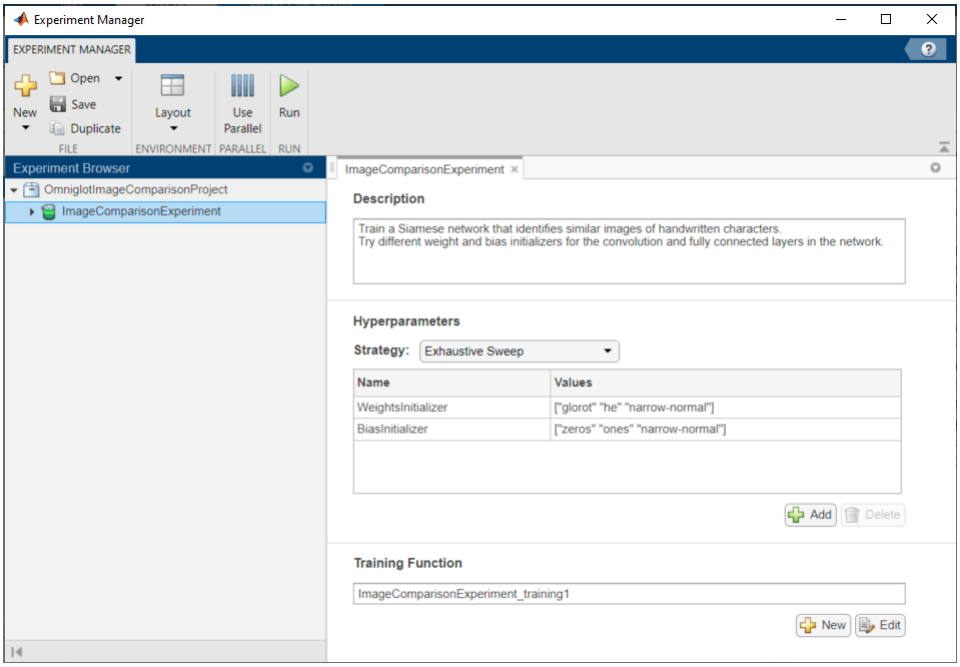
This diagram illustrates the Siamese network architecture in this example.



To compare two images, you pass each image through one of two identical subnetworks that share weights. The subnetworks convert each 105-by-105-by-1 image to a 4096-dimensional feature vector. Images of the same class have similar 4096-dimensional representations. The output feature vectors from each subnetwork are combined through subtraction and the result is passed through a `fullyconnect` operation with a single output. A sigmoid operation converts this value to a probability indicating that the images are similar (when the probability is close to 1) or dissimilar (when the probability is close to 0). The binary cross-entropy loss between the network prediction and the true label updates the network during training. For more information, see “Train a Siamese Network to Compare Images” on page 3-128.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`ImageComparisonExperiment`).



Custom training experiments consist of a description, a table of hyperparameters, and a training function. For more information, see “Configure Custom Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Train a Siamese network to identify similar and dissimilar images of handwritten characters.
Try different weight and bias initializers for the convolution and fully connected layers in the
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses the hyperparameters `WeightsInitializer` and `BiasInitializer` to specify the weight and bias initializers, respectively, for the convolution and fully connected layers in each subnetwork. For more information about these initializers, see “`WeightsInitializer`” and “`BiasInitializer`”.

The **Training Function** specifies the training data, network architecture, training options, and training procedure used by the experiment. The input to the training function is a structure with fields from the hyperparameter table and an `experiments.Monitor` object that you can use to track the progress of the training, record values of the metrics used by the training, and produce training plots. The training function returns a structure that contains the trained network, the weights for the final `fullyconnect` operation for the network, and the execution environment used for training. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete. The training function has five sections.

- **Initialize Output** sets the initial value of the network and `fullyconnect` weights to empty arrays to indicate that the training has not started. The experiment sets the execution environment to `"auto"`, so it trains and validates the network on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
output.network = [];
output.weights = [];
output.executionEnvironment = "auto";
```

- **Load and Preprocess Training and Test Data** defines the training and test data for the experiment as `imageDatastore` objects. The experiment uses the Omniglot data set, which consists of character sets for 50 alphabets, divided into 30 sets for training and 20 sets for testing. For more information on this data set, see “Image Data Sets” on page 19-118.

```
monitor.Status = "Loading Training Data";

url = "https://github.com/brendenlake/omniglot/raw/master/python/images_background.zip";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"images_background.zip");

dataFolderTrain = fullfile(downloadFolder,"images_background");
if ~exist(dataFolderTrain,"dir")
    websave(filename,url);
    unzip(filename,downloadFolder);
end

imdsTrain = imageDatastore(dataFolderTrain, ...
    IncludeSubfolders=true, ...
    LabelSource="none");

files = imdsTrain.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTrain.Labels = categorical(labels);

monitor.Status = "Loading Test Data";

url = "https://github.com/brendenlake/omniglot/raw/master/python/images_evaluation.zip";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"images_evaluation.zip");

dataFolderTest = fullfile(downloadFolder,"images_evaluation");
if ~exist(dataFolderTest,"dir")
    websave(filename,url);
    unzip(filename,downloadFolder);
end

imdsTest = imageDatastore(dataFolderTest, ...
    IncludeSubfolders=true, ...
    LabelSource="none");

files = imdsTest.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTest.Labels = categorical(labels);
```

- **Define Network Architecture** defines the architecture for two identical subnetworks that accept 105-by-105-by-1 images and output a feature vector. The convolution and fully connected layers use the weights and bias initializers specified in the hyperparameter table. To train the network with a custom training loop and enable automatic differentiation, the training function converts the layer graph to a `dlnetwork` object. The weights for the final `fullyconnect` operation are initialized by sampling a random selection from a narrow normal distribution with standard deviation of 0.01.

```
monitor.Status = "Creating Network";

layers = [
    imageInputLayer([105 105 1],Name="input1",Normalization="none")
    convolution2dLayer(10,64,Name="conv1", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu1")
    maxPooling2dLayer(2,Stride=2,Name="maxpool1")
    convolution2dLayer(7,128,Name="conv2", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu2")
    maxPooling2dLayer(2,'Stride',2,Name="maxpool2")
    convolution2dLayer(4,128,'Name','conv3', ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu3")
    maxPooling2dLayer(2,'Stride',2,Name="maxpool3")
    convolution2dLayer(5,256,Name="conv4", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu4")
    fullyConnectedLayer(4096,Name="fc1", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)];

lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph);

fcWeights = dlarray(0.01*randn(1,4096));
fcBias = dlarray(0.01*randn(1,1));
fcParams = struct(...
    "FcWeights",fcWeights,...
    "FcBias",fcBias);

output.network = dlnet;
output.weights = fcParams;
```

- **Specify Training Options** defines the training options used by the experiment. In this example, Experiment Manager trains the network with a mini-batch size of 180 for 1000 iterations, computing the accuracy of the network every 100 iterations. Training can take some time to run. For better results, consider increasing the training to 10,000 iterations.

```
numIterations = 1000;
miniBatchSize = 180;
validationFrequency = 100;
initialLearnRate = 6e-5;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.99;
trailingAvgSubnet = [];
trailingAvgSqSubnet = [];
trailingAvgParams = [];
trailingAvgSqParams = [];
```

- **Train Model** defines the custom training loop used by the experiment. For each iteration, the custom training loop extracts a batch of image pairs and labels, converts the data to `dlarray` objects with underlying type single, and specifies the dimension labels `'SSCB'` (spatial, spatial,

channel, batch) for the image data and `'CB'` (channel, batch) for the labels. If you train on a GPU, the data is converted to `gpuArray` (Parallel Computing Toolbox) objects. Then, the training function evaluates the model gradients and updates the network parameters. To validate, the training function creates a set of five random mini-batches of test pairs, evaluates the network predictions, and calculates the average accuracy over the mini-batches. After each iteration of the custom training loop, the training function saves the trained network and the weights for the `fullyconnect` operation, records the training loss, and updates the training progress.

```
monitor.Metrics = ["TrainingLoss" "ValidationAccuracy"];
monitor.XLabel = "Iteration";
monitor.Status = "Training";

for iteration = 1:numIterations
    [X1,X2,pairLabels] = getSiameseBatch(imdsTrain,miniBatchSize);
    dlX1 = dlarray(single(X1),"SSCB");
    dlX2 = dlarray(single(X2),"SSCB");
    if (output.executionEnvironment == "auto" && canUseGPU) || ...
            output.executionEnvironment == "gpu"
        dlX1 = gpuArray(dlX1);
        dlX2 = gpuArray(dlX2);
    end
    [gradientsSubnet, gradientsParams,loss] = dlfeval(@modelGradients, ...
        dlnet,fcParams,dlX1,dlX2,pairLabels);
    lossValue = double(gather(extractdata(loss)));
    [dlnet,trailingAvgSubnet,trailingAvgSqSubnet] = ...
        adamupdate(dlnet,gradientsSubnet, ...
        trailingAvgSubnet,trailingAvgSqSubnet, ...
        iteration,initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);
    [fcParams,trailingAvgParams,trailingAvgSqParams] = ...
        adamupdate(fcParams,gradientsParams, ...
        trailingAvgParams,trailingAvgSqParams, ...
        iteration,initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);
    if ~rem(iteration,validationFrequency) || iteration == 1 || iteration == numIterations
        monitor.Status = "Validating";
        accuracy = zeros(1,5);
        accuracyBatchSize = 150;
        for i = 1:5
            [XAcc1,XAcc2,pairLabelsAcc] = getSiameseBatch(imdsTest,accuracyBatchSize);
            dlXAcc1 = dlarray(single(XAcc1),"SSCB");
            dlXAcc2 = dlarray(single(XAcc2),"SSCB");
            if (output.executionEnvironment == "auto" && canUseGPU) || ...
                    output.executionEnvironment == "gpu"
                dlXAcc1 = gpuArray(dlXAcc1);
                dlXAcc2 = gpuArray(dlXAcc2);
            end
            dlY = predictSiamese(dlnet,fcParams,dlXAcc1,dlXAcc2);
            Y = gather(extractdata(dlY));
            Y = round(Y);
            accuracy(i) = sum(Y == pairLabelsAcc)/accuracyBatchSize;
        end
        recordMetrics(monitor,iteration, ...
            ValidationAccuracy=mean(accuracy)*100);
        monitor.Status = "Training";
    end
    output.network = dlnet;
    output.weights = fcParams;
    recordMetrics(monitor,iteration, ...
```

```
        TrainingLoss=lossValue);
    monitor.Progress = (iteration/numIterations)*100;
    if monitor.Stop
        return;
    end
end
```

To inspect the training function, under **Training Function**, click **Edit**. The training function opens in MATLAB® Editor. In addition, the code for the training function appears in Appendix 1 at the end of this example.

Run Experiment

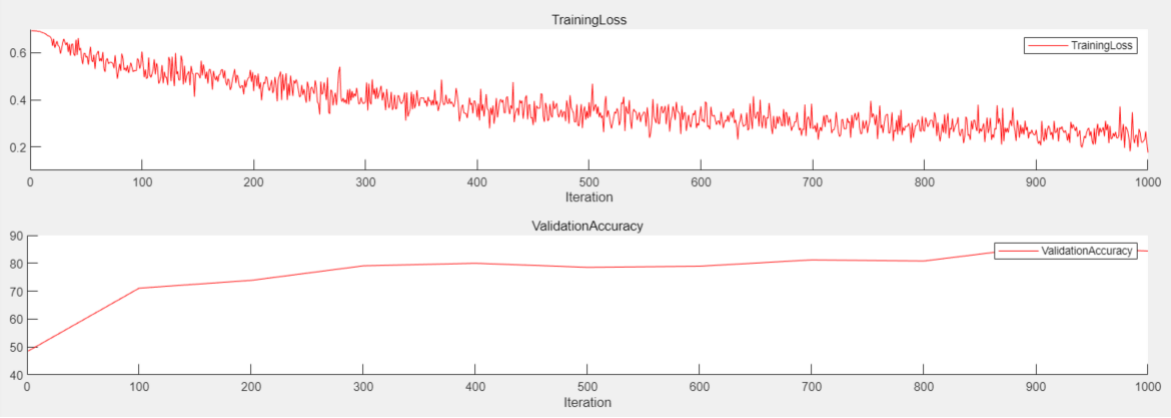
When you run the experiment, Experiment Manager trains the network defined by the training function multiple times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16.

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the training loss and validation accuracy for each trial.

Trial	Status	Progress	Elapsed Time	WeightsInitializer	BiasInitializer	TrainingLoss	ValidationAccuracy
1	Complete	100.0%	0 hr 13 min 16 sec	glorot	zeros	0.1753	84.4000
2	Complete	100.0%	0 hr 12 min 1 sec	he	zeros	0.1766	87.2000
3	Complete	100.0%	0 hr 12 min 18 sec	narrow-normal	zeros	0.2998	80.9333
4	Complete	100.0%	0 hr 12 min 28 sec	glorot	ones	0.2100	82.2667
5	Complete	100.0%	0 hr 12 min 42 sec	he	ones	0.2230	85.3333
6	Complete	100.0%	0 hr 12 min 24 sec	narrow-normal	ones	0.2904	79.4667
7	Complete	100.0%	0 hr 12 min 42 sec	glorot	narrow-normal	0.2280	81.6000
8	Complete	100.0%	0 hr 12 min 37 sec	he	narrow-normal	0.1971	89.2000
9	Complete	100.0%	0 hr 12 min 22 sec	narrow-normal	narrow-normal	0.2832	82.6667

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



Evaluate Results

To find the best result for your experiment, sort the table of results by validation accuracy.

1. Point to the **ValidationAccuracy** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

The trial with the highest validation accuracy appears at the top of the results table.

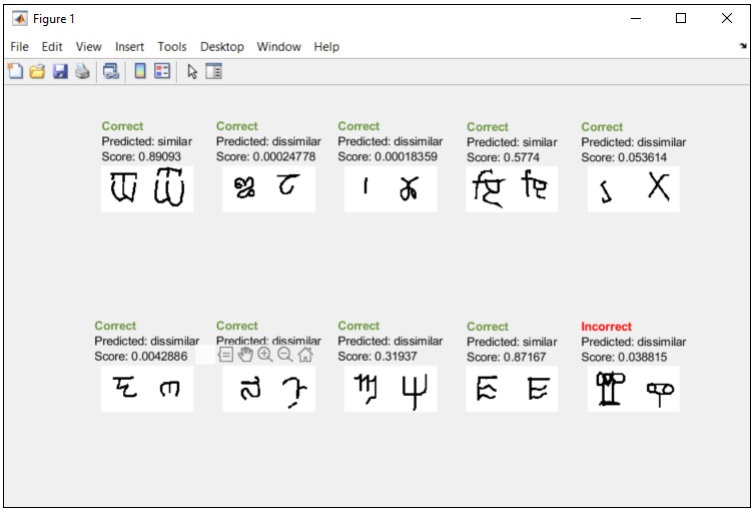
Trial	Status	Progress	Elapsed Time	WeightsInitializer	BiasInitializer	TrainingLoss	ValidationAccuracy
8	Complete	100.0%	0 hr 12 min 37 sec	he	narrow-normal	0.1971	89.2000
2	Complete	100.0%	0 hr 12 min 1 sec	he	zeros	0.1766	87.2000
5	Complete	100.0%	0 hr 12 min 42 sec	he	ones	0.2230	85.3333
1	Complete	100.0%	0 hr 13 min 16 sec	glorot	zeros	0.1753	84.4000
9	Complete	100.0%	0 hr 12 min 22 sec	narrow-normal	narrow-normal	0.2832	82.6667
4	Complete	100.0%	0 hr 12 min 28 sec	glorot	ones	0.2100	82.2667
7	Complete	100.0%	0 hr 12 min 42 sec	glorot	narrow-normal	0.2280	81.6000
3	Complete	100.0%	0 hr 12 min 18 sec	narrow-normal	zeros	0.2998	80.9333
6	Complete	100.0%	0 hr 12 min 24 sec	narrow-normal	ones	0.2904	79.4667

To visually check if the network correctly identifies similar and dissimilar pairs:

1. Select the trial with the highest accuracy.
2. On the **Experiment Manager** toolstrip, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported training output. The default name is `trainingOutput`.
4. Test the network on a small batch of image pairs by calling the `displayTestSet` function, which is listed in Appendix 3 at the end of this example. Use the exported training output as the input to the function. For instance, in the MATLAB Command Window, enter:

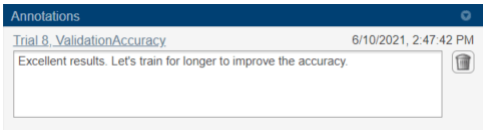
```
displayTestSet(trainingOutput)
```

The function displays 10 randomly selected pairs of test images with the prediction from the trained network, the probability score, and a label indicating whether the prediction is correct or incorrect.



To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **ValidationAccuracy** cell of the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.



For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Training Function

This function specifies the training data, network architecture, training options, and training procedure used by the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.
- `monitor` is an `experiments.Monitor` object that you can use to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots.

Output

- `output` is a structure that contains the trained network, the weights for the final `fullyconnect` operation for the network, and the execution environment used for training. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete.

```
function output = ImageComparisonExperiment_training1(params,monitor)

output.network = [];
output.weights = [];
output.executionEnvironment = "auto";

monitor.Status = "Loading Training Data";

url = "https://github.com/brendenlake/omniglot/raw/master/python/images_background.zip";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"images_background.zip");

dataFolderTrain = fullfile(downloadFolder,"images_background");
if ~exist(dataFolderTrain,"dir")
    websave(filename,url);
    unzip(filename,downloadFolder);
end

imdsTrain = imageDatastore(dataFolderTrain, ...
    IncludeSubfolders=true, ...
```

```
    LabelSource="none");

files = imdsTrain.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTrain.Labels = categorical(labels);

monitor.Status = "Loading Test Data";

url = "https://github.com/brendenlake/omniglot/raw/master/python/images_evaluation.zip";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"images_evaluation.zip");

dataFolderTest = fullfile(downloadFolder,"images_evaluation");
if ~exist(dataFolderTest,"dir")
    websave(filename,url);
    unzip(filename,downloadFolder);
end

imdsTest = imageDatastore(dataFolderTest, ...
    IncludeSubfolders=true, ...
    LabelSource="none");

files = imdsTest.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTest.Labels = categorical(labels);

monitor.Status = "Creating Network";

layers = [
    imageInputLayer([105 105 1],Name="input1",Normalization="none")
    convolution2dLayer(10,64,Name="conv1", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu1")
    maxPooling2dLayer(2,Stride=2,Name="maxpool1")
    convolution2dLayer(7,128,Name="conv2", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu2")
    maxPooling2dLayer(2,'Stride',2,Name="maxpool2")
    convolution2dLayer(4,128,'Name','conv3', ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu3")
    maxPooling2dLayer(2,'Stride',2,Name="maxpool3")
    convolution2dLayer(5,256,Name="conv4", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)
    reluLayer(Name="relu4")
    fullyConnectedLayer(4096,Name="fc1", ...
        WeightsInitializer=params.WeightsInitializer, ...
        BiasInitializer=params.BiasInitializer)];

lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph);
```

```
fcWeights = dlarray(0.01*randn(1,4096));
fcBias = dlarray(0.01*randn(1,1));
fcParams = struct(...
    "FcWeights",fcWeights,...
    "FcBias",fcBias);

output.network = dlnet;
output.weights = fcParams;

numIterations = 1000;
miniBatchSize = 180;
validationFrequency = 100;
initialLearnRate = 6e-5;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.99;
trailingAvgSubnet = [];
trailingAvgSqSubnet = [];
trailingAvgParams = [];
trailingAvgSqParams = [];

monitor.Metrics = ["TrainingLoss" "ValidationAccuracy"];
monitor.XLabel = "Iteration";
monitor.Status = "Training";

for iteration = 1:numIterations
    [X1,X2,pairLabels] = getSiameseBatch(imdsTrain,miniBatchSize);

    dlX1 = dlarray(single(X1),"SSCB");
    dlX2 = dlarray(single(X2),"SSCB");

    if (output.executionEnvironment == "auto" && canUseGPU) || ...
            output.executionEnvironment == "gpu"
        dlX1 = gpuArray(dlX1);
        dlX2 = gpuArray(dlX2);
    end

    [gradientsSubnet, gradientsParams,loss] = dlfeval(@modelGradients, ...
        dlnet,fcParams,dlX1,dlX2,pairLabels);
    lossValue = double(gather(extractdata(loss)));

    [dlnet,trailingAvgSubnet,trailingAvgSqSubnet] = ...
        adamupdate(dlnet,gradientsSubnet, ...
        trailingAvgSubnet,trailingAvgSqSubnet, ...
        iteration,initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);

    [fcParams,trailingAvgParams,trailingAvgSqParams] = ...
        adamupdate(fcParams,gradientsParams, ...
        trailingAvgParams,trailingAvgSqParams, ...
        iteration,initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);

    if ~rem(iteration,validationFrequency) || iteration == 1 || iteration == numIterations
        monitor.Status = "Validating";
        accuracy = zeros(1,5);
        accuracyBatchSize = 150;

        for i = 1:5
            [XAcc1,XAcc2,pairLabelsAcc] = getSiameseBatch(imdsTest,accuracyBatchSize);
```

```
            dlXAcc1 = dlarray(single(XAcc1),"SSCB");
            dlXAcc2 = dlarray(single(XAcc2),"SSCB");

            if (output.executionEnvironment == "auto" && canUseGPU) || ...
                    output.executionEnvironment == "gpu"
                dlXAcc1 = gpuArray(dlXAcc1);
                dlXAcc2 = gpuArray(dlXAcc2);
            end

            dlY = predictSiamese(dlnet,fcParams,dlXAcc1,dlXAcc2);

            Y = gather(extractdata(dlY));
            Y = round(Y);

            accuracy(i) = sum(Y == pairLabelsAcc)/accuracyBatchSize;
        end

        recordMetrics(monitor,iteration, ...
            ValidationAccuracy=mean(accuracy)*100);
        monitor.Status = "Training";
    end

    output.network = dlnet;
    output.weights = fcParams;
    recordMetrics(monitor,iteration, ...
        TrainingLoss=lossValue);
    monitor.Progress = (iteration/numIterations)*100;

    if monitor.Stop
        return;
    end
end
end
```

Appendix 2: Custom Training Helper Functions

The `modelGradients` function takes as input the Siamese `dlnetwork` object `net`, a pair of mini-batch input data `dlX1` and `dlX2`, and the label indicating whether they are similar or dissimilar. The function returns the gradients of the loss with respect to the learnable parameters in the network and the binary cross-entropy loss between the prediction and the ground truth.

```
function [gradientsSubnet,gradientsParams,loss] = modelGradients(dlnet,fcParams,dlX1,dlX2,pairLa
    Y = forwardSiamese(dlnet,fcParams,dlX1,dlX2);
    loss = binarycrossentropy(Y,pairLabels);
    [gradientsSubnet,gradientsParams] = dlgradient(loss,dlnet.Learnables,fcParams);
end
```

This function returns the binary cross-entropy loss value for a prediction from the network.

```
function loss = binarycrossentropy(Y,pairLabels)
    precision = underlyingType(Y);
    Y(Y < eps(precision)) = eps(precision);
    Y(Y > 1 - eps(precision)) = 1 - eps(precision);

     loss = -pairLabels.*log(Y) - (1 - pairLabels).*log(1 - Y);
     loss = sum(loss)/numel(pairLabels);
 end
```

This function defines how the subnetworks and the `fullyconnect` and `sigmoid` operations combine to form the complete Siamese network. The function accepts the network structure and two training images and returns a prediction of the probability of the pair being similar (closer to 1) or dissimilar (closer to 0).

```
function Y = forwardSiamese(dlnet,fcParams,dlX1,dlX2)
    F1 = forward(dlnet,dlX1);
    F1 = sigmoid(F1);

     F2 = forward(dlnet,dlX2);
     F2 = sigmoid(F2);

     Y = abs(F1 - F2);
     Y = fullyconnect(Y,fcParams.FcWeights,fcParams.FcBias);
     Y = sigmoid(Y);
 end
```

This function returns a randomly selected batch of paired images. On average, this function produces a balanced set of similar and dissimilar pairs.

```
function [X1,X2,pairLabels] = getSiameseBatch(imds,miniBatchSize)
    pairLabels = zeros(1,miniBatchSize);
    imgSize = size(readimage(imds,1));
    X1 = zeros([imgSize 1 miniBatchSize]);
    X2 = zeros([imgSize 1 miniBatchSize]);
    for i = 1:miniBatchSize
        choice = rand(1);
        if choice < 0.5
            [pairIdx1,pairIdx2,pairLabels(i)] = getSimilarPair(imds.Labels);
        else
            [pairIdx1,pairIdx2,pairLabels(i)] = getDissimilarPair(imds.Labels);
        end
        X1(:,:,:,i) = imds.readimage(pairIdx1);
        X2(:,:,:,i) = imds.readimage(pairIdx2);
    end
end
```

This function returns a random pair of indices for images that are in the same class and the similar pair label of 1.

```
function [pairIdx1,pairIdx2,label] = getSimilarPair(classLabel)
    classes = unique(classLabel);
    classChoice = randi(numel(classes));
    idxs = find(classLabel==classes(classChoice));
    pairIdxChoice = randperm(numel(idxs),2);
    pairIdx1 = idxs(pairIdxChoice(1));
    pairIdx2 = idxs(pairIdxChoice(2));
    label = 1;
end
```

This function returns a random pair of indices for images that are in different classes and the dissimilar pair label of 0.

```
function  [pairIdx1,pairIdx2,label] = getDissimilarPair(classLabel)
    classes = unique(classLabel);
    classesChoice = randperm(numel(classes),2);
    idxs1 = find(classLabel==classes(classesChoice(1)));
    idxs2 = find(classLabel==classes(classesChoice(2)));
```

```
    pairIdx1Choice = randi(numel(idxs1));
    pairIdx2Choice = randi(numel(idxs2));
    pairIdx1 = idxs1(pairIdx1Choice);
    pairIdx2 = idxs2(pairIdx2Choice);
    label = 0;
end
```

This function uses the trained network to make predictions about the similarity of two images.

```
function Y = predictSiamese(dlnet,fcParams,dlX1,dlX2)
    F1 = predict(dlnet,dlX1);
    F1 = sigmoid(F1);

     F2 = predict(dlnet,dlX2);
     F2 = sigmoid(F2);

     Y = abs(F1 - F2);
     Y = fullyconnect(Y,fcParams.FcWeights,fcParams.FcBias);
     Y = sigmoid(Y);
 end
```

Appendix 3: Display Pairs of Test Images

This function creates a small batch of image pairs to help you visually check if the network correctly identifies similar and dissimilar pairs. This function uses the helper functions `predictSiamese` and `predictSiamese`, which are listed in Appendix 2.

```
function displayTestSet(trainingOutput)

dlnet = trainingOutput.network;
fcParams = trainingOutput.weights;
executionEnvironment = trainingOutput.executionEnvironment;

downloadFolder = tempdir;
dataFolderTest = fullfile(downloadFolder,"images_evaluation");
imdsTest = imageDatastore(dataFolderTest, ...
    IncludeSubfolders=true, ...
    LabelSource="none");

files = imdsTest.Files;
parts = split(files,filesep);
labels = join(parts(:,(end-2):(end-1)),'_');
imdsTest.Labels = categorical(labels);

testBatchSize = 10;

[XTest1,XTest2,pairLabelsTest] = getSiameseBatch(imdsTest,testBatchSize);

dlXTest1 = dlarray(single(XTest1),"SSCB");
dlXTest2 = dlarray(single(XTest2),"SSCB");

if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
   dlXTest1 = gpuArray(dlXTest1);
   dlXTest2 = gpuArray(dlXTest2);
end

dlYScore = predictSiamese(dlnet,fcParams,dlXTest1,dlXTest2);
```

```
YScore = gather(extractdata(dlYScore));

YPred = round(YScore);

XTest1 = extractdata(dlXTest1);
XTest2 = extractdata(dlXTest2);

plotRatio = 16/9;
testingPlot = figure;
testingPlot.Position(3) = plotRatio*testingPlot.Position(4);
testingPlot.Visible = "on";

for i = 1:numel(pairLabelsTest)

    if YPred(i) == 1
        predLabel = "similar";
    else
        predLabel = "dissimilar" ;
    end

    if pairLabelsTest(i) == YPred(i)
        testStr = "\bf\color{darkgreen}Correct\rm\newline";
    else
        testStr = "\bf\color{red}Incorrect\rm\newline";
    end

    subplot(2,5,i)
    imshow([XTest1(:,:,:,i) XTest2(:,:,:,i)]);

    title(testStr + "\color{black}Predicted: " + predLabel + "\newlineScore: " + YScore(i));
end
```

See Also

Apps
Experiment Manager

Functions
`convolution2dLayer` | `fullyConnectedLayer`

Objects
`experiments.Monitor` | `dlnetwork` | `gpuArray`

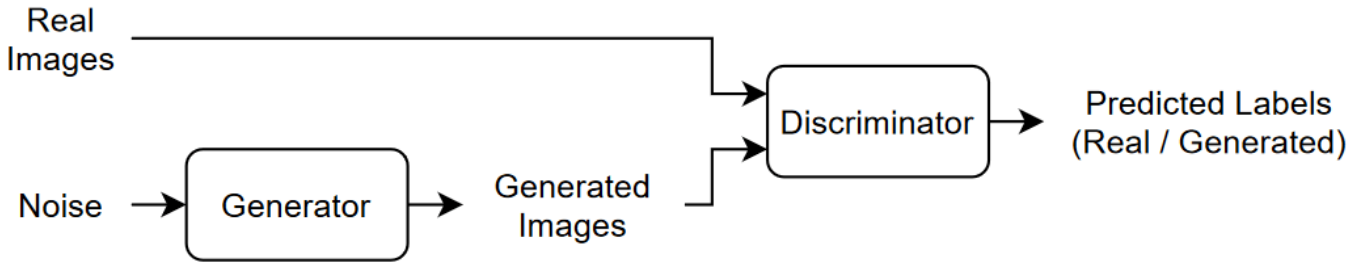
Related Examples

- “Train a Siamese Network to Compare Images” on page 3-128
- “Compare Layer Weight Initializers” on page 18-181
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16

Use Experiment Manager to Train Generative Adversarial Networks (GANs)

This example shows how to create a custom training experiment to train a generative adversarial network (GAN) that generates images of flowers. For a custom training experiment, you explicitly define the training procedure used by **Experiment Manager**. In this example, you implement a custom training loop to train a GAN, a type of deep learning network that can generate data with similar characteristics as the input real data. A GAN consists of two networks that train together:

- Generator — Given a vector of random values (latent inputs) as input, this network generates data with the same structure as the training data.
- Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as "real" or "generated."



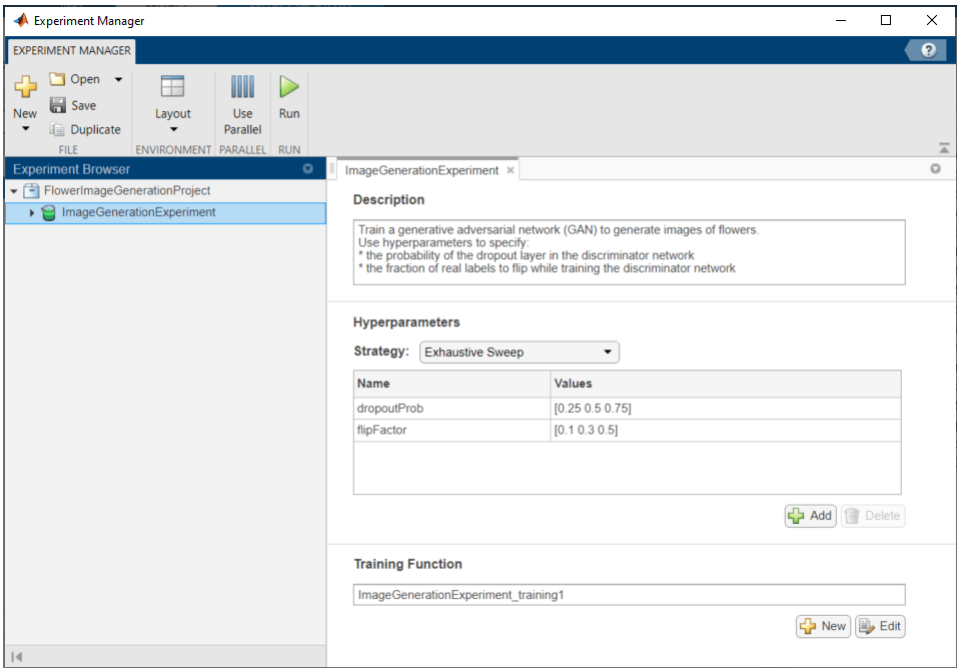
To train a GAN, train both networks simultaneously to maximize the performance of both networks:

- Train the generator to generate data that "fools" the discriminator. To optimize the performance of the generator, maximize the loss of the discriminator when given generated data. In other words, the objective of the generator is to generate data that the discriminator classifies as "real."
- Train the discriminator to distinguish between real and generated data. To optimize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. In other words, the objective of the discriminator is to not be "fooled" by the generator.

Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data. For more information, see “Train Generative Adversarial Network (GAN)” on page 3-76.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`ImageGenerationExperiment`).



Custom training experiments consist of a description, a table of hyperparameters, and a training function. For more information, see “Configure Custom Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Train a generative adversarial network (GAN) to generate images of flowers.
Use hyperparameters to specify:
* the probability of the dropout layer in the discriminator network
* the fraction of real labels to flip while training the discriminator network
```

The **Hyperparameters** section specifies the strategy (`Exhaustive Sweep`) and hyperparameter values to use for the experiment. When you run the experiment, Experiment Manager trains the network using every combination of hyperparameter values specified in the hyperparameter table. This example uses two hyperparameters:

- `dropoutProb` sets the probability of the dropout layer in the discriminator network. By default, the values for this hyperparameter are specified as `[0.25 0.5 0.75]`.
- `flipFactor` sets the fraction of real labels to flip when you train the discriminator network. The experiment uses this hyperparameter to add noise to the real data and better balance the learning of the discriminator and the generator. Otherwise, if the discriminator learns to discriminate between real and generated images too quickly, then the generator can fail to train. The values for this hyperparameter are specified as `[0.1 0.3 0.5]`.

The **Training Function** specifies the training data, network architecture, training options, and training procedure used by the experiment. The input to the training function is a structure with fields from the hyperparameter table and an `experiments.Monitor` object that you can use to track the progress of the training, record values of the metrics used by the training, and produce training plots. The training function returns a structure that contains the trained generator network, the trained discriminator network, and the execution environment used for training. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete. The training function has six sections.

- **Initialize Output** sets the initial value of the networks to empty arrays to indicate that the training has not started. The experiment sets the execution environment to `"auto"`, so it trains the networks on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see "GPU Support by Release" (Parallel Computing Toolbox).

```
output.generator = [];
output.discriminator = [];
output.executionEnvironment = "auto";
```

- **Load Training Data** defines the training data for the experiment as an `imageDatastore` object. The experiment uses the Flowers data set, which contains 3670 images of flowers and is about 218 MB. For more information on this data set, see "Image Data Sets" on page 19-118.

```
monitor.Status = "Loading Data";

url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
    websave(filename,url);
    untar(filename,downloadFolder)
end

datasetFolder = fullfile(imageFolder);
imdsTrain = imageDatastore(datasetFolder, ...
    IncludeSubfolders=true);

augmenter = imageDataAugmenter(RandXReflection=true);
augimdsTrain = augmentedImageDatastore([64 64],imdsTrain, ...
    DataAugmentation=augmenter);
```

- **Define Generator Network** defines the architecture for the generator network as a layer graph that generates images from 1-by-1-by-100 arrays of random values. To train the network with a custom training loop and enable automatic differentiation, the training function converts the layer graph to a `dlnetwork` object.

```
monitor.Status = "Creating Generator";

filterSize = 5;
numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 4 512];

layersGenerator = [
    featureInputLayer(numLatentInputs,Name="in")
    projectAndReshapeLayer(projectionSize,numLatentInputs,Name="proj")
    transposedConv2dLayer(filterSize,4*numFilters,Name="tconv1")
    batchNormalizationLayer(Name="bnorm1")
    reluLayer(Name="relu1")
    transposedConv2dLayer(filterSize,2*numFilters,Stride=2,Cropping="same",Name="tconv2")
    batchNormalizationLayer(Name="bnorm2")
    reluLayer(Name="relu2")
    transposedConv2dLayer(filterSize,numFilters,Stride=2,Cropping="same",Name="tconv3")
    batchNormalizationLayer(Name="bnorm3")
    reluLayer(Name="relu3")
```

```
    transposedConv2dLayer(filterSize,3,Stride=2,Cropping="same",Name="tconv4")
    tanhLayer(Name="tanh")];

lgraphGenerator = layerGraph(layersGenerator);
output.generator = dlnetwork(lgraphGenerator);
```

- **Define Discriminator Network** defines the architecture for the discriminator network as a layer graph that classifies real and generated 64-by-64-by-3 images. The dropout layer uses the dropout probability defined in the hyperparameter table. To train the network with a custom training loop and enable automatic differentiation, the training function converts the layer graph to a `dlnetwork` object.

```
monitor.Status = "Creating Discriminator";

filterSize = 5;
numFilters = 64;
inputSize = [64 64 3];
dropoutProb = params.dropoutProb;
scale = 0.2;

layersDiscriminator = [
    imageInputLayer(inputSize,Normalization="none",Name="in")
    dropoutLayer(dropoutProb,Name="dropout")
    convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same",Name="conv1")
    leakyReluLayer(scale,Name="lrelu1")
    convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same",Name="conv2")
    batchNormalizationLayer(Name="bn2")
    leakyReluLayer(scale,Name="lrelu2")
    convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same",Name="conv3")
    batchNormalizationLayer(Name="bn3")
    leakyReluLayer(scale,Name="lrelu3")
    convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same",Name="conv4")
    batchNormalizationLayer(Name="bn4")
    leakyReluLayer(scale,Name="lrelu4")
    convolution2dLayer(4,1,Name="conv5")];

lgraphDiscriminator = layerGraph(layersDiscriminator);
output.discriminator = dlnetwork(lgraphDiscriminator);
```

- **Specify Training Options** defines the training options used by the experiment. In this example, Experiment Manager trains the networks with a mini-batch size of 128 for 50 epochs using an initial learning rate of 0.0002, a gradient decay factor of 0.5, and a squared gradient decay factor of 0.999.

```
numEpochs = 50;
miniBatchSize = 128;
initialLearnRate = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
flipFactor = params.flipFactor;
```

- **Train Model** defines the custom training loop used by the experiment. The custom training loop uses `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch, the `minibatchqueue` object rescales the images in the range [-1,1], discards any partial mini-

batches with fewer than 128 observations, and formats the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. For each epoch, the custom training loop shuffles the datastore and loops over mini-batches of data. If you train on a GPU, the data is converted to `gpuArray` (Parallel Computing Toolbox) objects. Then, the training function evaluates the model gradients and updates the discriminator and generator network parameters. After each iteration of the custom training loop, the training function saves the trained networks and updates the training progress.

```
monitor.Metrics = ["scoreGenerator","scoreDiscriminator","scoreCombined"];
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Combined Score","scoreCombined");
groupSubPlot(monitor,"Generator and Discriminator Scores", ...
    ["scoreGenerator","scoreDiscriminator"]);
monitor.Status = "Training";
```

```
augimdsTrain.MiniBatchSize = miniBatchSize;
mbq = minibatchqueue(augimdsTrain,...
    MiniBatchSize=miniBatchSize,...
    PartialMiniBatch="discard",...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat="SSCB",...
    OutputEnvironment=output.executionEnvironment);
```

```
iteration = 0;
for epoch = 1:numEpochs
    shuffle(mbq);
    while hasdata(mbq)
        iteration = iteration + 1;
        dlX = next(mbq);
        Z = randn(numLatentInputs,miniBatchSize,"single");
        dlZ = dlarray(Z,"CB");
        if (output.executionEnvironment == "auto" && canUseGPU) || ...
                output.executionEnvironment == "gpu"
            dlZ = gpuArray(dlZ);
        end
        [gradientsGenerator,gradientsDiscriminator,stateGenerator,scoreGenerator,scoreDiscrimina
            dlfeval(@modelGradients,output.generator,output.discriminator,dlX,dlZ,flipFactor);
        output.generator.State = stateGenerator;
        [output.discriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
            adamupdate(output.discriminator,gradientsDiscriminator, ...
            trailingAvgDiscriminator,trailingAvgSqDiscriminator,iteration, ...
            initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);
        [output.generator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
            adamupdate(output.generator,gradientsGenerator, ...
            trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
            initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);
        scoreGeneratorValue = ...
            double(gather(extractdata(scoreGenerator)));
        scoreDiscriminatorValue = ...
            double(gather(extractdata(scoreDiscriminator)));
        scoreCombinedValue = 1-2*max(abs(scoreDiscriminatorValue-0.5),abs(scoreGeneratorValue-0.5
        recordMetrics(monitor,iteration, ...
            scoreGenerator=scoreGeneratorValue, ...
            scoreDiscriminator=scoreDiscriminatorValue, ...
            scoreCombined=scoreCombinedValue);
        if monitor.Stop || isnan(scoreGeneratorValue) || isnan(scoreDiscriminatorValue)
            return;
```

```
        end
    end
    monitor.Progress = (epoch/numEpochs)*100;
end
```

Training GANs can be a challenging task because the generator and the discriminator networks compete against each other during the training. If one network learns too quickly, then the other network can fail to learn. To help you diagnose issues and monitor how well the generator and discriminator achieve their respective goals, this experiment displays a pair of scores in the training plot. The generator score `scoreGenerator` measures the likelihood that the discriminator can correctly distinguish generated images. The discriminator score `scoreDiscriminator` measures the likelihood that the discriminator can correctly distinguish all input images, assuming that the numbers of real and generated images passed to the discriminator are equal. In the ideal case, both scores are 0.5. Scores that are too close to zero or one can indicate that one network dominates the other. See “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182.

To help you decide which trial produces the best results, this experiment combines the generator score and discriminator scores into a single numeric value, `scoreCombined`. This metric uses the *L*-∞ norm to determine how close the two networks are from the ideal scenario. It takes a value of one if both network scores equal 0.5, and zero if one of the network scores equals zero or one.

To inspect the training function, under **Training Function**, click **Edit**. The training function opens in MATLAB® Editor. In addition, the code for the training function appears in Appendix 1 at the end of this example.

Run Experiment

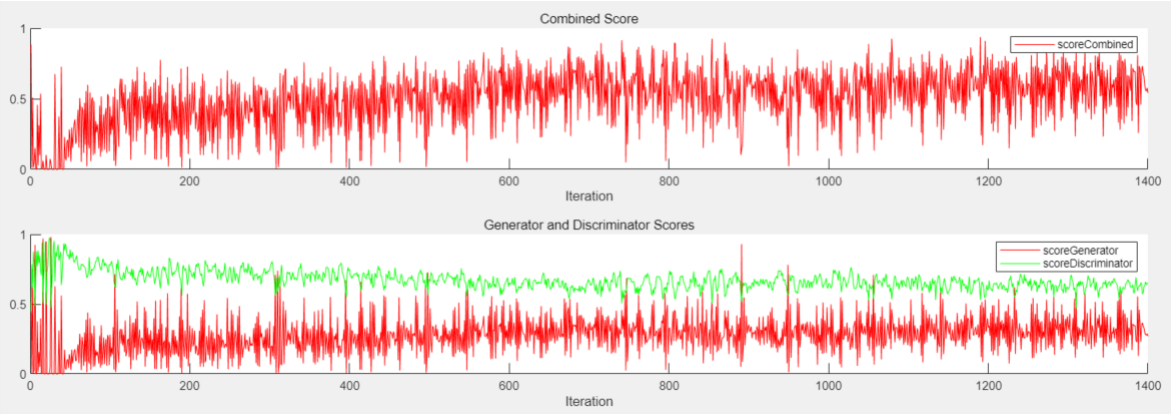
When you run the experiment, Experiment Manager trains the network defined by the training function multiple times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16.

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

A table of results displays the training loss and validation accuracy for each trial.

Trial	Status	Progress	Elapsed Time	dropoutProb	flipFactor	scoreGenerator	scoreDiscriminator	scoreCombined
1	Complete	100.0%	0 hr 7 min 0 sec	0.2500	0.1000	0.2703	0.6532	0.5407
2	Complete	100.0%	0 hr 6 min 45 sec	0.5000	0.1000	0.2413	0.6581	0.4827
3	Complete	100.0%	0 hr 6 min 44 sec	0.7500	0.1000	0.3999	0.6095	0.7811
4	Complete	100.0%	0 hr 6 min 46 sec	0.2500	0.3000	0.3076	0.6068	0.6152
5	Complete	100.0%	0 hr 6 min 49 sec	0.5000	0.3000	0.3753	0.5412	0.7505
6	Complete	100.0%	0 hr 6 min 50 sec	0.7500	0.3000	0.2782	0.6111	0.5564
7	Complete	100.0%	0 hr 6 min 53 sec	0.2500	0.5000	0.2335	0.5065	0.4670
8	Complete	100.0%	0 hr 6 min 53 sec	0.5000	0.5000	0.1647	0.5504	0.3293
9	Complete	100.0%	0 hr 7 min 1 sec	0.7500	0.5000	0.1872	0.5548	0.3744

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



Evaluate Results

To find the best result for your experiment, sort the table of results using the combined score.

1. Point to the **scoreCombined** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

The trial with the highest combined score appears at the top of the results table.

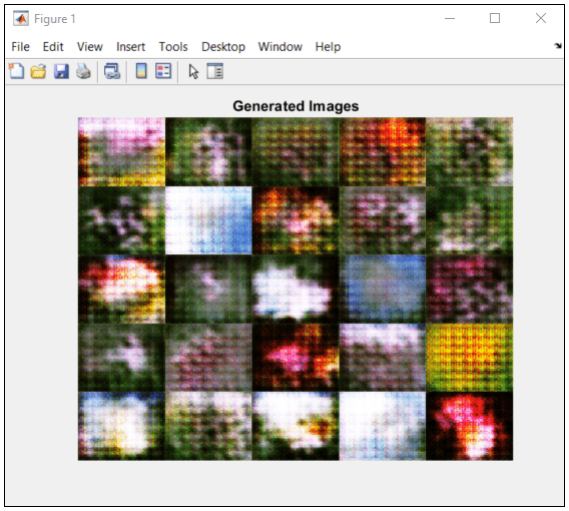
Trial	Status	Progress	Elapsed Time	dropoutProb	flipFactor	scoreGenerator	scoreDiscriminator	scoreCombined
3	Complete	100.0%	0 hr 6 min 44 sec	0.7500	0.1000	0.3999	0.6095	0.7811
5	Complete	100.0%	0 hr 6 min 49 sec	0.5000	0.3000	0.3753	0.5412	0.7505
4	Complete	100.0%	0 hr 6 min 46 sec	0.2500	0.3000	0.3076	0.6068	0.6152
6	Complete	100.0%	0 hr 6 min 50 sec	0.7500	0.3000	0.2782	0.6111	0.5564
1	Complete	100.0%	0 hr 7 min 0 sec	0.2500	0.1000	0.2703	0.6532	0.5407
2	Complete	100.0%	0 hr 6 min 45 sec	0.5000	0.1000	0.2413	0.6581	0.4827
7	Complete	100.0%	0 hr 6 min 53 sec	0.2500	0.5000	0.2335	0.5065	0.4670
9	Complete	100.0%	0 hr 7 min 1 sec	0.7500	0.5000	0.1872	0.5548	0.3744
8	Complete	100.0%	0 hr 6 min 53 sec	0.5000	0.5000	0.1647	0.5504	0.3293

Evaluate the quality of the GAN by generating and inspecting the images produced by the trained generator.

1. Select the trial with the highest combined score.
2. On the **Experiment Manager** toolstrip, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported training output. The default name is `trainingOutput`.
4. Test the trained generator network by calling the `generateTestImages` function, which is listed in Appendix 3 at the end of this example. Use the exported training output as the input to the function. For instance, in the MATLAB Command Window, enter:

```
generateTestImages(trainingOutput)
```

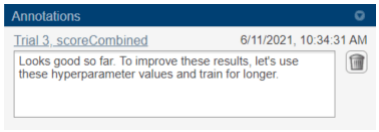
The function creates a batch of 25 random vectors to input to the generator network and displays the resulting images.



Using the combined score to sort your results might not identify the best trial in all cases. For best results, repeat this process for each trial with a high combined score, visually checking that the generator produces a variety of images without many duplicates. If the images have little diversity and some of them are almost identical, then your generator is likely affected by mode collapse. For more information, see “Mode Collapse” on page 5-184.

To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **scoreCombined** cell for the best trial.
2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.

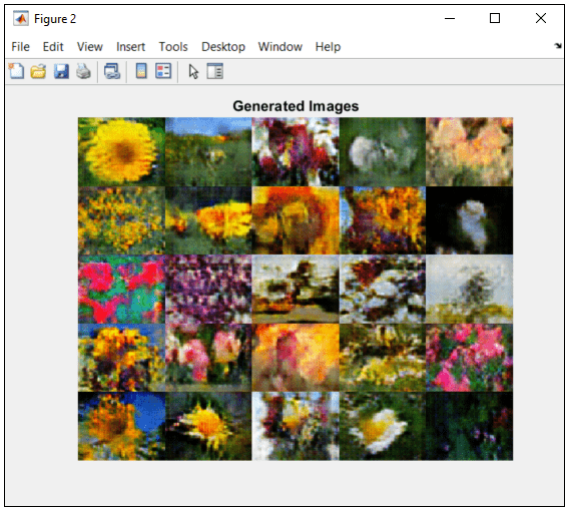


For more information, see “Sort, Filter, and Annotate Experiment Results”.

Rerun Experiment

After you identify the combination of hyperparameters that generates the best images, run the experiment a second time to train the network for a longer period of time.

1. Return to the experiment definition pane.
2. In the hyperparameter table, enter the hyperparameter values from your best trial. For example, to use the values from trial 3, change the value of `dropoutProb` to `0.75` and `flipFactor` to `0.1`.
3. Open the training function and specify a longer training time. Under **Specify Training Options**, change the value of `numEpochs` to `500`.
4. Run the experiment using the new hyperparameter values and training function. Experiment Manager runs a single trial. Training takes about 10 times longer than the previous trials.
5. When the experiment finishes, export the training output and run the `generateTestImages` function to test the new generator network. As before, visually check that the generator produces a variety of images without many duplicates.



Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Training Function

This function specifies the training data, network architecture, training options, and training procedure used by the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.
- `monitor` is an `experiments.Monitor` object that you can use to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots.

Output

- `output` is a structure that contains the trained generator network, the trained discriminator network, and the execution environment used for training. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete.

```
function output = ImageGenerationExperiment_training1(params,monitor)

output.generator = [];
output.discriminator = [];
output.executionEnvironment = "auto";

monitor.Status = "Loading Data";

url = "http://download.tensorflow.org/example_images/flower_photos.tgz";
downloadFolder = tempdir;
filename = fullfile(downloadFolder,"flower_dataset.tgz");

imageFolder = fullfile(downloadFolder,"flower_photos");
if ~exist(imageFolder,"dir")
```

```
    websave(filename,url);
    untar(filename,downloadFolder)
end

datasetFolder = fullfile(imageFolder);
imdsTrain = imageDatastore(datasetFolder, ...
    IncludeSubfolders=true);

augmenter = imageDataAugmenter(RandXReflection=true);
augimdsTrain = augmentedImageDatastore([64 64],imdsTrain, ...
    DataAugmentation=augmenter);

monitor.Status = "Creating Generator";

filterSize = 5;
numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 4 512];

layersGenerator = [
    featureInputLayer(numLatentInputs,Name="in")
    projectAndReshapeLayer(projectionSize,numLatentInputs,Name="proj")
    transposedConv2dLayer(filterSize,4*numFilters,Name="tconv1")
    batchNormalizationLayer(Name="bnorm1")
    reluLayer(Name="relu1")
    transposedConv2dLayer(filterSize,2*numFilters,Stride=2,Cropping="same",Name="tconv2")
    batchNormalizationLayer(Name="bnorm2")
    reluLayer(Name="relu2")
    transposedConv2dLayer(filterSize,numFilters,Stride=2,Cropping="same",Name="tconv3")
    batchNormalizationLayer(Name="bnorm3")
    reluLayer(Name="relu3")
    transposedConv2dLayer(filterSize,3,Stride=2,Cropping="same",Name="tconv4")
    tanhLayer(Name="tanh")];

lgraphGenerator = layerGraph(layersGenerator);
output.generator = dlnetwork(lgraphGenerator);

monitor.Status = "Creating Discriminator";

filterSize = 5;
numFilters = 64;
inputSize = [64 64 3];
dropoutProb = params.dropoutProb;
scale = 0.2;

layersDiscriminator = [
    imageInputLayer(inputSize,Normalization="none",Name="in")
    dropoutLayer(dropoutProb,Name="dropout")
    convolution2dLayer(filterSize,numFilters,Stride=2,Padding="same",Name="conv1")
    leakyReluLayer(scale,Name="lrelu1")
    convolution2dLayer(filterSize,2*numFilters,Stride=2,Padding="same",Name="conv2")
    batchNormalizationLayer(Name="bn2")
    leakyReluLayer(scale,Name="lrelu2")
    convolution2dLayer(filterSize,4*numFilters,Stride=2,Padding="same",Name="conv3")
    batchNormalizationLayer(Name="bn3")
    leakyReluLayer(scale,Name="lrelu3")
    convolution2dLayer(filterSize,8*numFilters,Stride=2,Padding="same",Name="conv4")
    batchNormalizationLayer(Name="bn4")
```

```
    leakyReluLayer(scale,Name="lrelu4")
    convolution2dLayer(4,1,Name="conv5")];

lgraphDiscriminator = layerGraph(layersDiscriminator);
output.discriminator = dlnetwork(lgraphDiscriminator);

numEpochs = 50;
miniBatchSize = 128;
initialLearnRate = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
flipFactor = params.flipFactor;

monitor.Metrics = ["scoreGenerator","scoreDiscriminator","scoreCombined"];
monitor.XLabel = "Iteration";
groupSubPlot(monitor,"Combined Score","scoreCombined");
groupSubPlot(monitor,"Generator and Discriminator Scores", ...
    ["scoreGenerator","scoreDiscriminator"]);
monitor.Status = "Training";

augimdsTrain.MiniBatchSize = miniBatchSize;
mbq = minibatchqueue(augimdsTrain,...
    MiniBatchSize=miniBatchSize,...
    PartialMiniBatch="discard",...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat="SSCB",...
    OutputEnvironment=output.executionEnvironment);

iteration = 0;
for epoch = 1:numEpochs
    shuffle(mbq);
    while hasdata(mbq)
        iteration = iteration + 1;
        dlX = next(mbq);

        Z = randn(numLatentInputs,miniBatchSize,"single");
        dlZ = dlarray(Z,"CB");

        if (output.executionEnvironment == "auto" && canUseGPU) || ...
                output.executionEnvironment == "gpu"
            dlZ = gpuArray(dlZ);
        end

        [gradientsGenerator,gradientsDiscriminator,stateGenerator,scoreGenerator,scoreDiscrimina
            dlfeval(@modelGradients,output.generator,output.discriminator,dlX,dlZ,flipFactor);
        output.generator.State = stateGenerator;

        [output.discriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
            adamupdate(output.discriminator,gradientsDiscriminator, ...
            trailingAvgDiscriminator,trailingAvgSqDiscriminator,iteration, ...
            initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);

        [output.generator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
            adamupdate(output.generator,gradientsGenerator, ...
```

```
            trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
            initialLearnRate,gradientDecayFactor,squaredGradientDecayFactor);

        scoreGeneratorValue = ...
            double(gather(extractdata(scoreGenerator)));
        scoreDiscriminatorValue = ...
            double(gather(extractdata(scoreDiscriminator)));
        scoreCombinedValue = 1-2*max(abs(scoreDiscriminatorValue-0.5),abs(scoreGeneratorValue-0.5

        recordMetrics(monitor,iteration, ...
            scoreGenerator=scoreGeneratorValue, ...
            scoreDiscriminator=scoreDiscriminatorValue, ...
            scoreCombined=scoreCombinedValue);

        if monitor.Stop || isnan(scoreGeneratorValue) || isnan(scoreDiscriminatorValue)
            return;
        end
    end
    monitor.Progress = (epoch/numEpochs)*100;
end
end
```

Appendix 2: Custom Training Helper Functions

This function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array.
2. Rescale the images to be in the range [-1,1].

```
function X = preprocessMiniBatch(data)
    X = cat(4,data{:});
    X = rescale(X,-1,1,InputMin=0,InputMax=255);
end
```

This function takes as input the generator and discriminator `dlnetwork` objects (`dlnetGenerator` and `dlnetDiscriminator`), a mini-batch of input data (`dlX`), an array of random values (`dlZ`), and the percentage of real labels to flip (`flipFactor`), and returns the gradients of the loss with respect to the learnable parameters in the networks, the generator state, and the scores of the two networks. Because the discriminator output is not in the range [0,1], `modelGradients` applies the `sigmoid` function to convert this value into a probability.

```
function [gradientsGenerator,gradientsDiscriminator,stateGenerator,scoreGenerator,scoreDiscrimina
    modelGradients(dlnetGenerator,dlnetDiscriminator,dlX,dlZ,flipFactor)
    dlYPred = forward(dlnetDiscriminator,dlX);
    [dlXGenerated,stateGenerator] = forward(dlnetGenerator,dlZ);
    dlYPredGenerated = forward(dlnetDiscriminator,dlXGenerated);
    probGenerated = sigmoid(dlYPredGenerated);
    probReal = sigmoid(dlYPred);
    scoreDiscriminator = ((mean(probReal)+mean(1-probGenerated))/2);
    scoreGenerator = mean(probGenerated);
    numObservations = size(probReal,4);
    idx = randperm(numObservations,floor(flipFactor*numObservations));
    probReal(:,:,:,idx) = 1-probReal(:,:,:,idx);
    [lossGenerator,lossDiscriminator] = GANLoss(probReal,probGenerated);
    gradientsGenerator = dlgradient(lossGenerator, ...
        dlnetGenerator.Learnables,RetainData=true);
```

```
    gradientsDiscriminator = dlgradient(lossDiscriminator, ...
        dlnetDiscriminator.Learnables);
end
```

This function returns the loss for the discriminator and generator networks.

```
function [lossGenerator,lossDiscriminator] = ...
    GANLoss(probReal,probGenerated)
    lossDiscriminator = -mean(log(probReal))-mean(log(1-probGenerated));
    lossGenerator = -mean(log(probGenerated));
end
```

Appendix 3: Generate Test Images

This function creates a batch of 25 random vectors to input to the generator network and displays the resulting images. Use this function to visually check that the generator produces a variety of images without many duplicates. If the images have little diversity and some of them are almost identical, then your generator is likely affected by mode collapse.

```
function generateTestImages(trainingOutput)

dlnetGenerator = trainingOutput.generator;
executionEnvironment = trainingOutput.executionEnvironment;

numLatentInputs = 100;
numTestImages = 25;

ZTest = randn(numLatentInputs,numTestImages,"single");
dlZTest = dlarray(ZTest,"CB");

if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZTest = gpuArray(dlZTest);
end

dlXGeneratedTest = predict(dlnetGenerator,dlZTest);

I = imtile(extractdata(dlXGeneratedTest));
I = rescale(I);
figure
image(I)
axis off
title("Generated Images")

end
```

See Also

Apps
Experiment Manager

Objects
`dlarray` | `dlnetwork` | `experiments.Monitor` | `gpuArray` | `minibatchqueue`

Related Examples

- “Train Network Using Custom Training Loop” on page 18-225
- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182

Use Bayesian Optimization in Custom Training Experiments

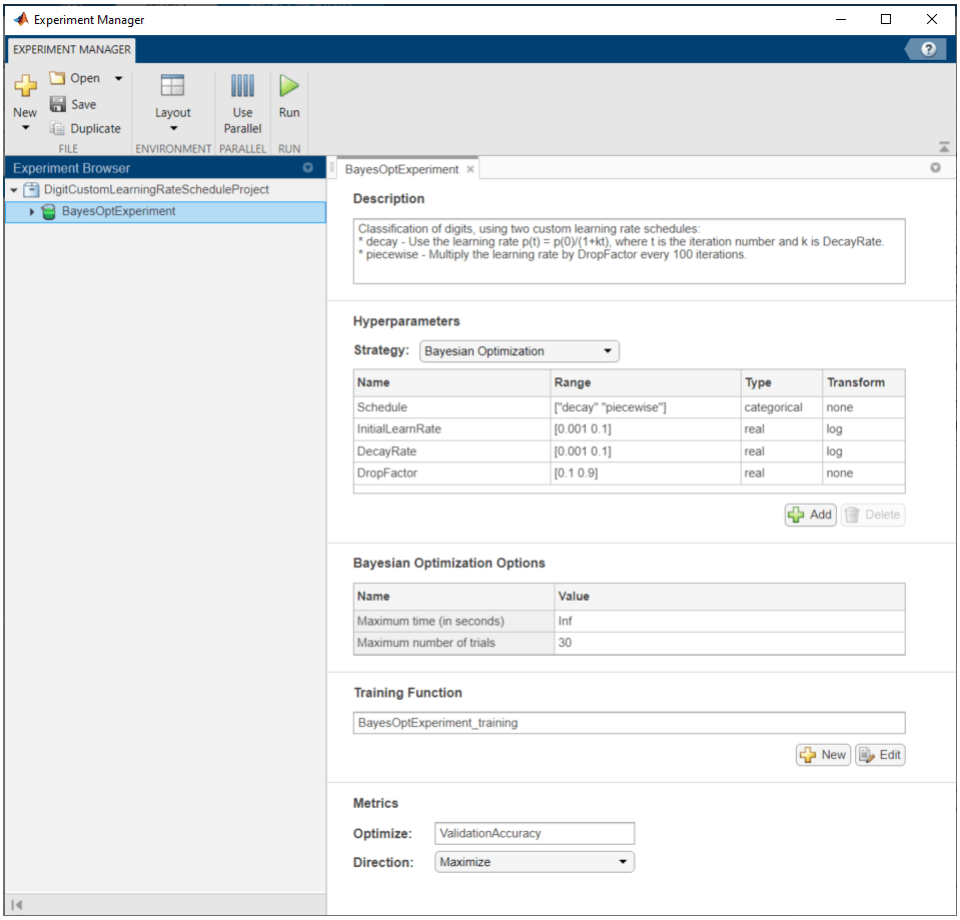
This example shows how to use Bayesian optimization to find optimal hyperparameter values for custom training experiments in **Experiment Manager**. Bayesian optimization provides an alternative strategy to sweeping hyperparameters in an experiment. You specify a range of values for each hyperparameter and select a metric to optimize, and Experiment Manager searches for a combination of hyperparameters that optimizes your selected metric. Bayesian optimization requires Statistics and Machine Learning Toolbox™.

In this example, you train a network to classify images of handwritten digits using a custom learning rate schedule. The experiment uses Bayesian optimization to find the type of schedule and the combination of hyperparameters that maximizes the validation accuracy. For more information on using a custom learning rate schedule, see “Train Network Using Custom Training Loop” on page 18-225 and “Piecewise Learn Rate Schedule” on page 18-217.

Alternatively, you can find optimal hyperparameter values programmatically by calling the `bayesopt` function. For more information, see “Deep Learning Using Bayesian Optimization” on page 5-99.

Open Experiment

First, open the example. Experiment Manager loads a project with a preconfigured experiment that you can inspect and run. To open the experiment, in the **Experiment Browser** pane, double-click the name of the experiment (`BayesOptExperiment`).



Custom training experiments consist of a description, a table of hyperparameters, and a training function. Experiments that use Bayesian optimization include additional options to limit the duration of the experiment. For more information, see “Configure Custom Training Experiment”.

The **Description** field contains a textual description of the experiment. For this example, the description is:

```
Classification of digits, using two custom learning rate schedules:
* decay - Use the learning rate p(t) = p(0)/(1+kt), where t is the iteration number and k is Deca
* piecewise - Multiply the learning rate by DropFactor every 100 iterations.
```

The **Hyperparameters** section specifies the strategy (`Bayesian Optimization`) and hyperparameter options to use for the experiment. For each hyperparameter, specify these options:

- **Range** — Enter a two-element vector that gives the lower bound and upper bound of a real- or integer-valued hyperparameter, or a string array or cell array that lists the possible values of a categorical hyperparameter.
- **Type** — Select `real` (real-valued hyperparameter), `integer` (integer-valued hyperparameter), or `categorical` (categorical hyperparameter).
- **Transform** — Select `none` (no transform) or `log` (logarithmic transform). For `log`, the hyperparameter must be `real` or `integer` and positive. With this option, the hyperparameter is searched and modeled on a logarithmic scale.

When you run the experiment, Experiment Manager searches for the best combination of hyperparameters. Each trial in the experiment uses a new combination of hyperparameter values based on the results of the previous trials. This example uses the hyperparameters `Schedule`, `InitialLearnRate`, `DecayRate`, and `DropFactor` to specify the custom learning rate schedule used for training. The options for `Schedule` are:

- `decay` — For each iteration, use the time-based learning rate $\rho_t = \frac{\rho_0}{1+kt}$, where t is the iteration number, ρ_0 is the initial learning rate specified by `InitialLearnRate`, and k is the decay rate specified by `DecayRate`. This option ignores the value of the hyperparameter `DropFactor`.
- `piecewise` — Start with the initial learning rate specified by `InitialLearnRate` and periodically drop the learning rate by multiplying by the drop factor specified by `DropFactor`. In this example, the learning rate drops every 100 iterations. This option ignores the value of the hyperparameter `DecayRate`.

The experiment models `InitialLearnRate` and `DecayRate` on a logarithmic scale because the range of values for these hyperparameters (`0.001` to `0.1`) spans several orders of magnitude. In contrast, the values for `DropFactor` range from `0.1` to `0.9`, so the experiment models `DropFactor` on a linear scale.

Under **Bayesian Optimization Options**, you can specify the duration of the experiment by entering the maximum time (in seconds) and the maximum number of trials to run. To best use the power of Bayesian optimization, perform at least 30 objective function evaluations.

The **Training Function** specifies the training data, network architecture, training options, and training procedure used by the experiment. The input to the training function is a structure with fields from the hyperparameter table and an `experiments.Monitor` object that you can use to track the progress of the training, record values of the metrics used by the training, and produce training plots. The training function returns a structure that contains the trained network, the training loss, the validation accuracy, and the execution environment used for training. Experiment Manager saves

this output, so you can export it to the MATLAB workspace when the training is complete. The training function has five sections.

- **Initialize Output** sets the initial value of the network, loss, and accuracy to empty arrays to indicate that the training has not started. The experiment sets the execution environment to `"auto"`, so it trains the network on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For more information, see "GPU Support by Release" (Parallel Computing Toolbox).

```
output.trainedNet = [];
output.trainingInfo.loss = [];
output.trainingInfo.accuracy = [];
output.executionEnvironment = "auto";
```

- **Load Training Data** defines the training and validation data for the experiment as augmented image datastores using the Digits data set. For each image in the training set, the experiment applies a random translation of up to 5 pixels on the horizontal and vertical axes. For more information on this data set, see "Image Data Sets" on page 19-118.

```
dataFolder = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imds = imageDatastore(dataFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9,'randomize');
inputSize = [28 28 1];
pixelRange = [-5 5];
imageAugmenter = imageDataAugmenter( ...
    RandXTranslation = pixelRange, ...
    RandYTranslation = pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    DataAugmentation = imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
classes = categories(imdsTrain.Labels);
numClasses = numel(classes);
```

- **Define Network Architecture** defines the architecture for the image classification network. To train the network with a custom training loop and enable automatic differentiation, the training function converts the layer graph to a `dlnetwork` object.

```
layers = [
    imageInputLayer(inputSize,Normalization="none",Name="input")
    convolution2dLayer(5,20,Name ="conv1")
    batchNormalizationLayer(Name="bn1")
    reluLayer(Name="relu1")
    convolution2dLayer(3,20,Padding="same",Name="conv2")
    batchNormalizationLayer(Name="bn2")
    reluLayer(Name="relu2")
    convolution2dLayer(3,20,Padding="same",Name="conv3")
    batchNormalizationLayer(Name="bn3")
    reluLayer(Name="relu3")
    fullyConnectedLayer(numClasses,Name="fc")
    softmaxLayer(Name="softmax")];
lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph);
```

- **Specify Training Options** defines the training options used by the experiment. In this example, Experiment Manager trains the networks with a mini-batch size of 128 for 10 epochs using the custom learning rate schedule defined by the hyperparameters.

```
numEpochs = 10;
miniBatchSize = 128;
momentum = 0.9;

learnRateSchedule = params.Schedule;
initialLearnRate = params.InitialLearnRate;
learnRateDecay = params.DecayRate;
learnRateDropFactor = params.DropFactor;
learnRateDropPeriod = 100;
learnRate = initialLearnRate;
```

- **Train Model** defines the custom training loop used by the experiment. The custom training loop uses `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch, the `minibatchqueue` object converts the labels to one-hot encoded variables and formats the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. If you train on a GPU, the data is converted to `gpuArray` (Parallel Computing Toolbox) objects. For each epoch, the custom training loop shuffles the datastore, loops over mini-batches of data, and evaluates the model gradients, state, and loss. Then, the training function determines the learning rate for the selected schedule and updates the network parameters. After each iteration of the custom training loop, the training function computes the validation accuracy, saves the trained network, and updates the training progress.

```
monitor.Metrics = ["LearnRate" "TrainingLoss" "ValidationAccuracy"];
monitor.XLabel = "Iteration";

mbq = minibatchqueue(augimdsTrain,...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat={'SSCB',''},...
    OutputEnvironment=output.executionEnvironment);

iteration = 0;
velocity = [];
for epoch = 1:numEpochs
    shuffle(mbq);
    while hasdata(mbq)
        iteration = iteration + 1;
        [dlX, dlY] = next(mbq);
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);
        dlnet.State = state;
        switch learnRateSchedule
            case "decay"
                learnRate = initialLearnRate/(1 + learnRateDecay*iteration);
            case "piecewise"
                if mod(iteration,learnRateDropPeriod) == 0
                    learnRate = learnRate*learnRateDropFactor;
                end
        end
        recordMetrics(monitor,iteration, ...
            LearnRate=learnRate, ...
            TrainingLoss=loss);
        output.trainingInfo.loss = [output.trainingInfo.loss; iteration loss];
```

```
        [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate,momentum);
        if monitor.Stop
            return;
        end
    end
    numOutputs = 1;
    mbqTest = minibatchqueue(augimdsValidation,numOutputs, ...
        MiniBatchSize=miniBatchSize, ...
        MiniBatchFcn=@preprocessMiniBatchPredictors, ...
        MiniBatchFormat="SSCB");
    predictions = modelPredictions(dlnet,mbqTest,classes);
    YTest = imdsValidation.Labels;
    accuracy = mean(predictions == YTest)*100.0;
    output.trainedNet = dlnet;
    monitor.Progress = (epoch*100.0)/numEpochs;
    recordMetrics(monitor,iteration, ...
        ValidationAccuracy=accuracy);
    output.trainingInfo.accuracy = [output.trainingInfo.accuracy; iteration accuracy];
end
```

To inspect the training function, under **Training Function**, click **Edit**. The training function opens in MATLAB® Editor. In addition, the code for the training function appears in Appendix 1 at the end of this example.

In the **Metrics** section, the **Optimize** and **Direction** fields indicate the metric that the Bayesian optimization algorithm uses as an objective function. For this experiment, Experiment Manager seeks to maximize the value of the validation accuracy.

Run Experiment

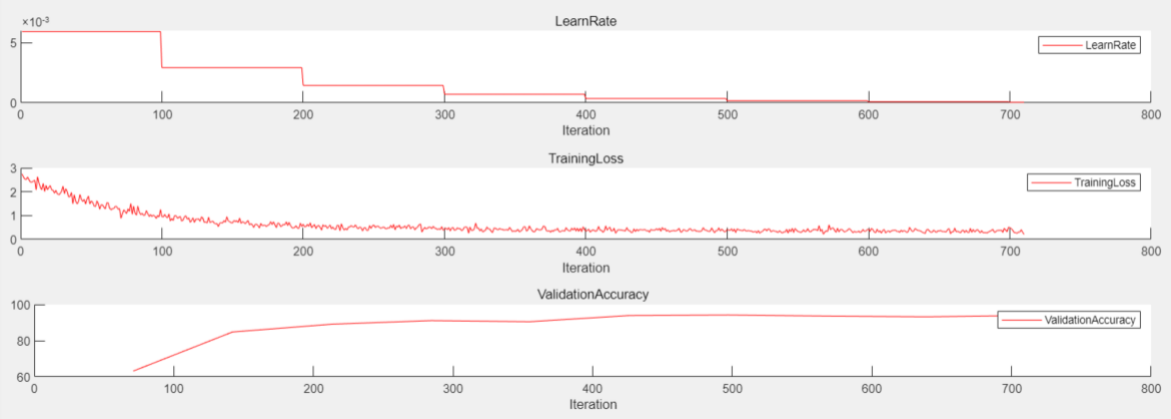
When you run the experiment, Experiment Manager trains the network defined by the training function multiple times. Each trial uses a different combination of hyperparameter values. By default, Experiment Manager runs one trial at a time. If you have Parallel Computing Toolbox, you can run multiple trials at the same time. For best results, before you run your experiment, start a parallel pool with as many workers as GPUs. For more information, see “Use Experiment Manager to Train Networks in Parallel” on page 6-16.

- To run one trial of the experiment at a time, on the Experiment Manager toolstrip, click **Run**.
- To run multiple trials at the same time, click **Use Parallel** and then **Run**. If there is no current parallel pool, Experiment Manager starts one using the default cluster profile. Experiment Manager then executes multiple simultaneous trials, depending on the number of parallel workers available.

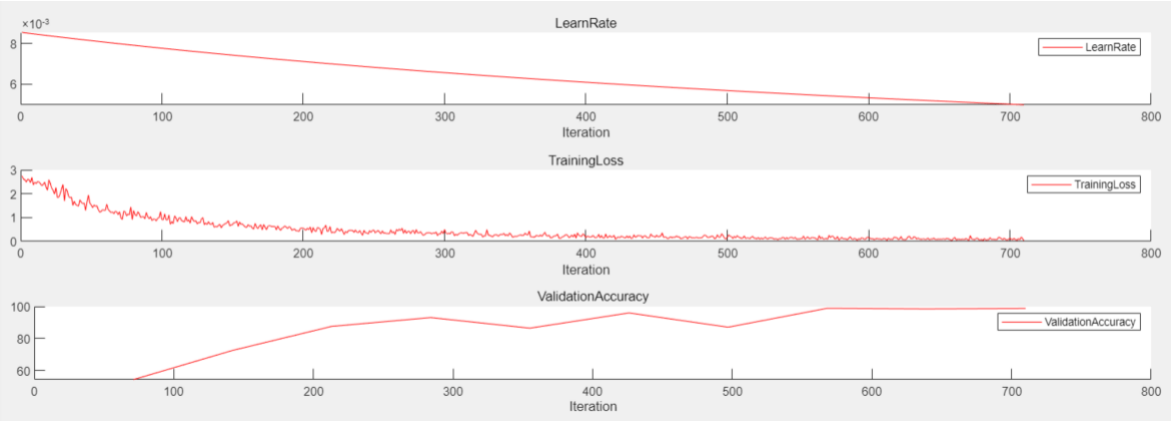
A table of results displays the training loss and validation accuracy for each trial. Experiment Manager indicates the trial with the optimal value for the selected metric. For example, in this experiment, the third trial produces the greatest validation accuracy.

Trial	Status	Progress	Elapsed Time	Schedule	InitialLearnRate	DecayRate	DropFactor	LearnRate	TrainingLoss	ValidationAccuracy
1	Complete	100.0%	0 hr 1 min 25 sec	piecewise	0.0059	0.0467	0.4938	4.2303e-5	0.2173	93.9000
2	Complete	100.0%	0 hr 1 min 20 sec	decay	0.0013	0.0233	0.8366	0.0001	1.0960	79.6000
3	Complete	100.0%	0 hr 1 min 19 sec	piecewise	0.0319	0.0017	0.7229	0.0033	1.6278	49.3000
4	Complete	100.0%	0 hr 1 min 19 sec	decay	0.0674	0.0065	0.2849	0.0120	2.2960	15.4000
5	Complete	100.0%	0 hr 1 min 20 sec	piecewise	0.0040	0.0407	0.3786	4.4626e-6	0.3610	92.4000
6	Complete	100.0%	0 hr 1 min 22 sec	piecewise	0.0020	0.0924	0.3758	2.0986e-6	0.6665	86.1000
7	Complete	100.0%	0 hr 1 min 22 sec	piecewise	0.0057	0.0999	0.2362	2.3262e-7	0.3664	89.9000
8	Complete	100.0%	0 hr 1 min 20 sec	piecewise	0.0997	0.0810	0.7806	0.0176	2.3010	10.0000
9	Complete	100.0%	0 hr 1 min 18 sec	piecewise	0.0042	0.0256	0.8998	0.0020	0.0551	98.3000
10	Complete	100.0%	0 hr 1 min 19 sec	piecewise	0.0054	0.0116	0.8981	0.0025	0.0400	97.6000
11	Complete	100.0%	0 hr 1 min 17 sec	decay	0.0010	0.0723	0.1033	1.9182e-5	1.8155	51.6000
12	Complete	100.0%	0 hr 1 min 16 sec	decay	0.0050	0.0084	0.8983	0.0007	0.1829	91.3000
13	Complete	100.0%	0 hr 1 min 29 sec	piecewise	0.0040	0.0118	0.8988	0.0019	0.0841	94.4000
14	Complete	100.0%	0 hr 1 min 19 sec	piecewise	0.0058	0.0029	0.8986	0.0027	0.0290	98.0000
15	Complete	100.0%	0 hr 1 min 19 sec	piecewise	0.0058	0.0056	0.8995	0.0027	0.0474	96.5000
16	Complete	100.0%	0 hr 1 min 19 sec	piecewise	0.0055	0.0104	0.8988	0.0026	0.0531	98.0000
17	Complete	100.0%	0 hr 1 min 26 sec	decay	0.0128	0.0013	0.1000	0.0067	0.0119	97.3000
18	Complete	100.0%	0 hr 1 min 22 sec	decay	0.0078	0.0053	0.1007	0.0016	0.0644	97.0000
19	Complete	100.0%	0 hr 1 min 18 sec	decay	0.0105	0.0227	0.8975	0.0006	0.2876	95.0000
20	Complete	100.0%	0 hr 1 min 23 sec	decay	0.0101	0.0096	0.1050	0.0013	0.0916	93.5000
21	Complete	100.0%	0 hr 1 min 19 sec	decay	0.0094	0.0010	0.8997	0.0055	0.0376	96.3000
22	Complete	100.0%	0 hr 1 min 18 sec	decay	0.0023	0.0109	0.8920	0.0003	0.5264	90.3000
23	Complete	100.0%	0 hr 1 min 27 sec	piecewise	0.0055	0.0956	0.8962	0.0026	0.0630	90.6000
24	Complete	100.0%	0 hr 1 min 24 sec	piecewise	0.0043	0.0010	0.8716	0.0017	0.0491	96.9000
25	Complete	100.0%	0 hr 1 min 20 sec	piecewise	0.0049	0.0010	0.8916	0.0022	0.0729	93.4000
26	Complete	100.0%	0 hr 1 min 20 sec	decay	0.0085	0.0010	0.1175	0.0050	0.0286	98.8000
27	Complete	100.0%	0 hr 1 min 27 sec	piecewise	0.0010	0.0011	0.8650	0.0004	0.4047	89.3000
28	Complete	100.0%	0 hr 1 min 24 sec	decay	0.0025	0.0010	0.8967	0.0015	0.1354	96.1000
29	Complete	100.0%	0 hr 1 min 19 sec	decay	0.0089	0.0010	0.1423	0.0052	0.0189	98.2000
30	Complete	100.0%	0 hr 1 min 16 sec	decay	0.0086	0.0010	0.1728	0.0050	0.0284	95.7000

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial. The training plot shows the learning rate, training loss, and validation accuracy for each trial. For example, this training plot is for a trial that uses a piecewise learning rate schedule.



In contrast, this training plot is for a trial that uses a time-based decay learning rate schedule.



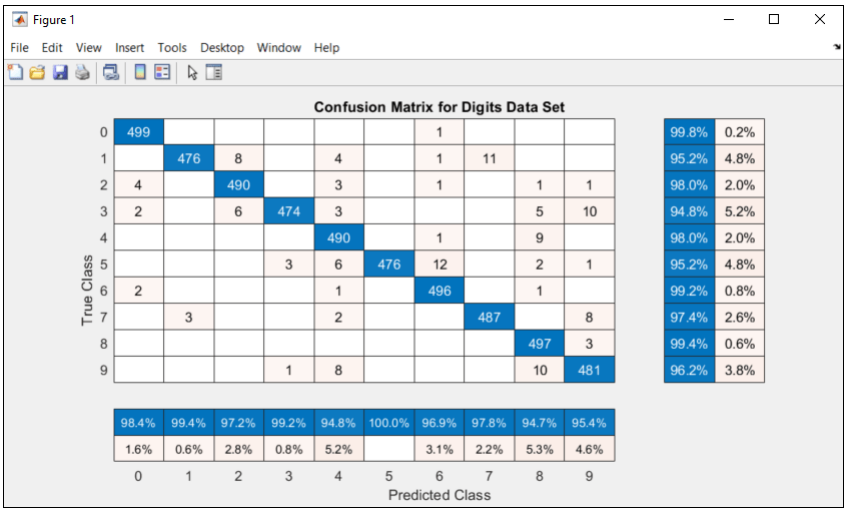
Evaluate Results

To test the best trial in your experiment, plot a confusion matrix.

1. In the results table, select the trial with the highest validation accuracy.
2. On the **Experiment Manager** toolstrip, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported training output. The default name is `trainingOutput`.
4. Create a confusion matrix by calling the `drawConfusionMatrix` function, which is listed in Appendix 3 at the end of this example. As the input to the function, use the exported training output and the fraction of the Digits data set to use as a test set. For instance, in the MATLAB Command Window, enter:

```
drawConfusionMatrix(trainingOutput,0.5)
```

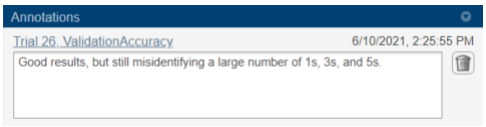
The function creates a confusion matrix using half of the images in the data set rotated by a small, random angle.



To record observations about the results of your experiment, add an annotation.

1. In the results table, right-click the **ValidationAccuracy** cell for the best trial.

2. Select **Add Annotation**.
3. In the **Annotations** pane, enter your observations in the text box.



For more information, see “Sort, Filter, and Annotate Experiment Results”.

Close Experiment

In the **Experiment Browser** pane, right-click the name of the project and select **Close Project**. Experiment Manager closes all of the experiments and results contained in the project.

Appendix 1: Training Function

This function specifies the training data, network architecture, training options, and training procedure used by the experiment.

Input

- `params` is a structure with fields from the Experiment Manager hyperparameter table.
- `monitor` is an `experiments.Monitor` object that you can use to track the progress of the training, update information fields in the results table, record values of the metrics used by the training, and produce training plots.

Output

- `output` is a structure that contains the trained network, the values of the training loss and validation accuracy, and the execution environment used for training. Experiment Manager saves this output, so you can export it to the MATLAB workspace when the training is complete.

```
function output = BayesOptExperiment_training(params,monitor)

output.trainedNet = [];
output.trainingInfo.loss = [];
output.trainingInfo.accuracy = [];
output.executionEnvironment = "auto";

dataFolder = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imds = imageDatastore(dataFolder, ...
    IncludeSubfolders=true, ...
    LabelSource="foldernames");
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9,'randomize');
inputSize = [28 28 1];
pixelRange = [-5 5];
imageAugmenter = imageDataAugmenter( ...
    RandXTranslation = pixelRange, ...
    RandYTranslation = pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    DataAugmentation = imageAugmenter);
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
classes = categories(imdsTrain.Labels);
```

```
numClasses = numel(classes);

layers = [
    imageInputLayer(inputSize,Normalization="none",Name="input")
    convolution2dLayer(5,20,Name ="conv1")
    batchNormalizationLayer(Name="bn1")
    reluLayer(Name="relu1")
    convolution2dLayer(3,20,Padding="same",Name="conv2")
    batchNormalizationLayer(Name="bn2")
    reluLayer(Name="relu2")
    convolution2dLayer(3,20,Padding="same",Name="conv3")
    batchNormalizationLayer(Name="bn3")
    reluLayer(Name="relu3")
    fullyConnectedLayer(numClasses,Name="fc")
    softmaxLayer(Name="softmax")];
lgraph = layerGraph(layers);
dlnet = dlnetwork(lgraph);

numEpochs = 10;
miniBatchSize = 128;
momentum = 0.9;

learnRateSchedule = params.Schedule;
initialLearnRate = params.InitialLearnRate;
learnRateDecay = params.DecayRate;
learnRateDropFactor = params.DropFactor;
learnRateDropPeriod = 100;
learnRate = initialLearnRate;

monitor.Metrics = ["LearnRate" "TrainingLoss" "ValidationAccuracy"];
monitor.XLabel = "Iteration";

mbq = minibatchqueue(augimdsTrain,...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat={'SSCB',''},...
    OutputEnvironment=output.executionEnvironment);

iteration = 0;
velocity = [];
for epoch = 1:numEpochs
    shuffle(mbq);

    while hasdata(mbq)
        iteration = iteration + 1;

        [dlX, dlY] = next(mbq);

        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);
        dlnet.State = state;

        switch learnRateSchedule
            case "decay"
                learnRate = initialLearnRate/(1 + learnRateDecay*iteration);
            case "piecewise"
                if mod(iteration,learnRateDropPeriod) == 0
                    learnRate = learnRate*learnRateDropFactor;
                end
```

```
        end

        recordMetrics(monitor,iteration, ...
            LearnRate=learnRate, ...
            TrainingLoss=loss);
        output.trainingInfo.loss = [output.trainingInfo.loss; iteration loss];

        [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate,momentum);

        if monitor.Stop
            return;
        end
    end

    numOutputs = 1;
    mbqTest = minibatchqueue(augimdsValidation,numOutputs, ...
        MiniBatchSize=miniBatchSize, ...
        MiniBatchFcn=@preprocessMiniBatchPredictors, ...
        MiniBatchFormat="SSCB");
    predictions = modelPredictions(dlnet,mbqTest,classes);
    YTest = imdsValidation.Labels;
    accuracy = mean(predictions == YTest)*100.0;

    output.trainedNet = dlnet;
    monitor.Progress = (epoch*100.0)/numEpochs;
    recordMetrics(monitor,iteration, ...
        ValidationAccuracy=accuracy);
    output.trainingInfo.accuracy = [output.trainingInfo.accuracy; iteration accuracy];
end
end
```

Appendix 2: Custom Training Helper Functions

The `modelGradients` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,Y)
[dlYPred,state] = forward(dlnet,dlX);
loss = crossentropy(dlYPred,Y);
gradients = dlgradient(loss,dlnet.Learnables);
loss = double(gather(extractdata(loss)));
end
```

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)
predictions = [];
while hasdata(mbq)
    dlXTest = next(mbq);
    dlYPred = predict(dlnet,dlXTest);
    YPred = onehotdecode(dlYPred,classes,1)';
    predictions = [predictions; YPred];
```

```
end
end
```

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using these steps:

1. Preprocess the images using the `preprocessMiniBatchPredictors` function.
2. Extract the label data from the incoming cell array and concatenate the data into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)
X = preprocessMiniBatchPredictors(XCell);
Y = cat(2,YCell{1:end});
Y = onehotencode(Y,1);
end
```

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenating the data into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, for use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)
X = cat(4,XCell{1:end});
end
```

Appendix 3: Create Confusion Matrix

This function takes as input a trained network and the fraction of the Digits data set to use as a test set and creates a confusion matrix chart. This function uses the helper functions `modelPredictions` and `preprocessMiniBatchPredictors`, which are listed in Appendix 2.

```
function drawConfusionMatrix(trainingOutput,testSize)

dataFolder = fullfile(toolboxdir("nnet"), ...
    "nndemos","nndatasets","DigitDataset");
imds = imageDatastore(dataFolder, ...
    IncludeSubfolders=true, ....
    LabelSource="foldernames");
imdsTest = splitEachLabel(imds,testSize,"randomize");
inputSize = [28 28 1];
imageAugmenter = imageDataAugmenter(RandRotation=[-15 15]);
augimdsTest = augmentedImageDatastore(inputSize(1:2),imdsTest, ...
    DataAugmentation=imageAugmenter);
classes = categories(imdsTest.Labels);

trainedNet = trainingOutput.trainedNet;
numOutputs = 1;
miniBatchSize = 128;
mbqTest = minibatchqueue(augimdsTest,numOutputs, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@preprocessMiniBatchPredictors, ...
    MiniBatchFormat="SSCB");
predictedLabels = modelPredictions(trainedNet,mbqTest,classes);
trueLabels = imdsTest.Labels;
```

```
figure
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title="Confusion Matrix for Digits Data Set");
cm = gcf;
cm.Position(3) = cm.Position(3)*1.5;

end
```

See Also

Apps
Experiment Manager

Objects
`experiments.Monitor` | `dlnetwork` | `gpuArray`

More About

- “Deep Learning Using Bayesian Optimization” on page 5-99
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Tune Experiment Hyperparameters by Using Bayesian Optimization” on page 6-26
- “Use Experiment Manager to Train Networks in Parallel” on page 6-16
- “Bayesian Optimization Algorithm” (Statistics and Machine Learning Toolbox)

Keyboard Shortcuts for Experiment Manager

Use these keyboard shortcuts when working with Experiment Manager.

Shortcuts for General Navigation

Action	Shortcut
Show the access keys for the toolstrip	**Alt+E** Not supported on macOS systems.
Move forward through the different areas of the Experiment Manager app, including the toolstrip, Experiment Browser, and experiment or results panes.	**Ctrl+F6** On macOS systems, use **Command+F6** instead.
Move backward through the different areas of the Experiment Manager app, including the toolstrip, Experiment Browser, and experiment or results panes.	**Ctrl+Shift+F6** On macOS systems, use **Command+Shift+F6** instead.
Move forward through the different elements in the start page, toolstrip, experiment pane, or results pane	**Tab**
Move backward through the different elements in the start page, toolstrip, experiment pane, or results pane	**Shift+Tab**
Select the current option in the start page or a menu	**Enter**
Cancel the current action, for example, hide the access keys for the toolstrip or close a menu	**Esc**

Shortcuts for Experiment Browser

Use **Ctrl+F6** or **Ctrl+Shift+F6** to navigate to the Experiment Browser. Then press **Tab** to bring into focus the project, experiment, or result that is currently selected.

Action	Shortcut
Show the experiments in the project or the results for an experiment	**Right** arrow If the contents of the project or experiment are already visible, pressing the **Right** arrow selects the first experiment in the project or the first result for the experiment.
Hide the experiments in the project or the results for an experiment	**Left** arrow If the contents of an experiment are already hidden, pressing the **Left** arrow selects the project that contains the experiment.
Move to the next item in the browser	**Down** arrow

Action	Shortcut
Move to the previous item in the browser	**Up** arrow
Open an experiment or result	**Shift+F10** and select **Open** Not supported on macOS systems.
Rename an experiment or result	**F2** On macOS systems, use **Fn+F2** instead.

Shortcuts for Results Table

Use **Ctrl+F6** or **Ctrl+Shift+F6** to navigate to the results pane. Then press **Tab** to navigate down to the results table.

Action	Shortcut
Move to the next trial in the table	**Down** arrow
Move to the previous trial in the table	**Up** arrow
Move to the next column in the table	**Right** arrow
Move to the previous column in the table	**Left** arrow

See Also

Apps
Experiment Manager

Deep Learning in Parallel and the Cloud

Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud

In this section...

Training deep networks is computationally intensive and can take many hours of computing time; however, neural networks are inherently parallel algorithms. You can take advantage of this parallelism by running in parallel using high-performance GPUs and computer clusters.

It is recommended to train using a GPU or multiple GPUs. Only use single CPU or multiple CPUs if you do not have a GPU. CPUs are normally much slower that GPUs for both training and inference. Running on a single GPU typically offers much better performance than running on multiple CPU cores.

If you do not have a suitable GPU, you can rent high-performance GPUs and clusters in the cloud. For more information on hot to access MATLAB in the cloud for deep learning, see “Deep Learning in the Cloud” on page 7-10.

Using a GPU or parallel options requires Parallel Computing Toolbox. Using a GPU also requires a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Using a remote cluster also requires MATLAB Parallel Server.

Tip For `trainNetwork` workflows, GPU support is automatic. By default, the `trainNetwork` function uses a GPU if one is available. If you have access to a machine with multiple GPUs, specify the `ExecutionEnvironment` training option as `"multi-gpu"`.

To run custom training workflows, including `dlnetwork` workflows, on the GPU, use `minibatchqueue` to automatically convert data to `gpuArray` objects.

You can use parallel resources to scale up deep learning for a single network. You can also train multiple networks simultaneously. The following sections show the available options for deep learning in parallel in MATLAB:

- “Train Single Network in Parallel” on page 7-3
  - “Use Local Resources to Train Single Network in Parallel” on page 7-3
  - “Use Remote Cluster Resources to Train Single Network in Parallel” on page 7-4
  - “Use Deep Network Designer and Experiment Manager to Train Single Network in Parallel” on page 7-5
- “Train Multiple Networks in Parallel” on page 7-6
  - “Use Local or Remote Cluster Resources to Train Multiple Network in Parallel” on page 7-6
  - “Use Experiment Manager to Train Multiple Networks in Parallel” on page 7-7

- “Batch Deep Learning” on page 7-7

Note If you run MATLAB on a single remote machine for example, a cloud machine that you connect to via ssh or remote desktop protocol, then follow the steps for local resources. For more information on connecting to cloud resources, see “Deep Learning in the Cloud” on page 7-10.

Train Single Network in Parallel

Use Local Resources to Train Single Network in Parallel

The following table shows you the available options for training and inference with single network on your local workstation.

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Single CPU	Automatic if no GPU is available. Training using a single CPU is not recommended.	Training using a single CPU is not recommended.	• MATLAB • Deep Learning Toolbox
Multiple CPU cores	Training using multiple CPU cores is not recommended if you have access to a GPU.	Training using multiple CPU cores is not recommended if you have access to a GPU.	• MATLAB • Deep Learning Toolbox • Parallel Computing Toolbox
Single GPU	Automatic. By default, training and inference run on the GPU if one is available. Alternatively, specify the `ExecutionEnvironment` training option as `"gpu"`	Use `minibatchqueue` to automatically convert data to `gpuArray` objects. For more information, see “Run Custom Training Loops on a GPU and in Parallel” on page 7-20 For an example, see “Train Network Using Custom Training Loop” on page 18-225.	

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Multiple GPUs	Specify the `ExecutionEnvironment` training option as `"multi-gpu"`. For an example, see "Train Network Using Automatic Multi-GPU Support" on page 7-42.	Start a local parallel pool with as many workers as available GPUs. For more information, see "Deep Learning with MATLAB on Multiple GPUs" on page 7-13 Use `parpool` to execute training or inference with portion of a mini-batch on each worker. Convert each partial mini-batch of data to `gpuArray` objects. For training, aggregate gradients, loss and state parameters after each iteration. For more information, see "Run Custom Training Loops on a GPU and in Parallel" on page 7-20. For an example, see "Train Network in Parallel with Custom Training Loop" on page 7-55. Set the `executionEnvironment` variable to `"auto"` or `"gpu"`.	

Use Remote Cluster Resources to Train Single Network in Parallel

The following table shows you the available options for training and inference with single network on a remote cluster.

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Multiple CPUs	Training using multiple CPU cores is not recommended if you have access to a GPU.	Training using multiple CPU cores is not recommended if you have access to a GPU.	• MATLAB • Deep Learning Toolbox • Parallel Computing Toolbox • MATLAB Parallel Server

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Multiple GPUs	Specify the desired cluster as your default cluster profile. For more information, see "Manage Cluster Profiles and Automatic Pool Creation" on page 7-8 Specify the `ExecutionEnvironment` training option as `"parallel"`. For an example, see "Train Network in the Cloud Using Automatic Parallel Support" on page 7-28.	Start a parallel pool in the desired cluster with as many workers as available GPUs. For more information, see "Deep Learning with MATLAB on Multiple GPUs" on page 7-13 Use `parpool` to execute training or inference with a portion of a mini-batch on each worker. Convert each partial mini-batch of data to `gpuArray` objects. For training, aggregate gradients, loss and state parameters after each iteration. For more information, see "Run Custom Training Loops on a GPU and in Parallel" on page 7-20 For an example, see "Train Network in Parallel with Custom Training Loop" on page 7-55. Set the `executionEnvironment` variable to `"auto"` or `"gpu"`.	

Use Deep Network Designer and Experiment Manager to Train Single Network in Parallel

You can train a single network in parallel using Deep Network Designer. You can train using local resources or a remote cluster.

- To train locally using multiple GPUs, set the `ExectionEnvironment` option to `multi-gpu` in the Training Options dialog.
- To train using a remote cluster, set the `ExectionEnvironment` option to `parallel` in the Training Options dialog. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

You can use Experiment Manager to run a single trial using multiple parallel workers. For more information, see "Use Experiment Manager to Train Networks in Parallel" on page 6-16.

Train Multiple Networks in Parallel

Use Local or Remote Cluster Resources to Train Multiple Network in Parallel

To train multiple networks in parallel, train each network on a different parallel worker. You can modify the network or training parameters on each worker to perform parameter sweeps in parallel.

Use `parfor` or `parfeval` to train a single network on each worker. To run in the background without blocking your local MATLAB, use `parfeval`. You can plot results using the `OutputFcn` training option.

You can run locally or using a remote cluster. Using a remote cluster requires MATLAB Parallel Server.

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Multiple CPUs	Specify the desired cluster as your default cluster profile. For more information, see “Manage Cluster Profiles and Automatic Pool Creation” on page 7-8 Use `parfor` or `parfeval` to simultaneously execute training or inference on each worker. Specify the `ExecutionEnvironment` training option as `"cpu"` for each network. For examples, see • “Use parfor to Train Multiple Deep Learning Networks” on page 7-46 • “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32	Specify the desired cluster as your default cluster profile. For more information, see “Manage Cluster Profiles and Automatic Pool Creation” on page 7-8 Use `parfor` or `parfeval` to simultaneously execute training or inference on each worker. For more information, see “Run Custom Training Loops on a GPU and in Parallel” on page 7-20	• MATLAB • Deep Learning Toolbox • Parallel Computing Toolbox • (optional) MATLAB Parallel Server

Resource	`trainNetwork` Workflows	Custom Training Workflows	Required Products
Multiple GPUs	Start a parallel pool in the desired cluster with as many workers as available GPUs. For more information, see "Deep Learning with MATLAB on Multiple GPUs" on page 7-13 Use `parfor` or `parfeval` to simultaneously execute a network on each worker. Specify the `ExecutionEnvironment` training option as `"gpu"` for each network. For examples, see • "Use parfor to Train Multiple Deep Learning Networks" on page 7-46 • "Use parfeval to Train Multiple Deep Learning Networks" on page 7-32	Start a parallel pool in the desired cluster with as many workers as available GPUs. For more information, see "Deep Learning with MATLAB on Multiple GPUs" on page 7-13 Use `parfor` or `parfeval` to simultaneously execute training or inference on each worker. For more information, see "Run Custom Training Loops on a GPU and in Parallel" on page 7-20 Convert each mini-batch of data to `gpuArray`. Use `minibatchqueue` and set `OutputEnvironment` property to `'gpu'` to automatically convert data to `gpuArray` objects.	

Use Experiment Manager to Train Multiple Networks in Parallel

You can use Experiment Manager to run trials on multiple parallel workers simultaneously. Set up your parallel environment and enable the **Use Parallel** option before running your experiment. Experiment Manager runs as many simultaneous trials as there are workers in your parallel pool. For more information, see "Use Experiment Manager to Train Networks in Parallel" on page 6-16.

Batch Deep Learning

You can offload deep learning computations to run in the background using the `batch` function. This means that you can continue using MATLAB while your computation runs in the background, or you can close your client MATLAB and fetch results later.

You can run batch jobs in a local or remote cluster. To offload your deep learning computations, use `batch` to submit a script or function that runs in the cluster. You can perform any kind of deep learning computation as a batch job, including parallel computations. For an example, see "Send Deep Learning Batch Job to Cluster" on page 7-39

To run in parallel, use a script or function that contains the same code that you would use to run in parallel locally or in a cluster. For example, your script or function can run `trainNetwork` using the

`"ExecutionEnvironment","parallel"` option, or run a custom training loop in parallel. Use `batch` to submit the script or function to the cluster and use the `Pool` option to specify the number of workers you want to use. For more information on running parallel computations with `batch`, see "Run Batch Parallel Jobs" (Parallel Computing Toolbox).

To run deep learning computation on multiple networks, it is recommended to submit a single batch job for each network. Doing so avoids the overhead required to start a parallel pool in the cluster and allows you to use the job monitor to observe the progress of each network computation individually.

You can submit multiple batch jobs. If the submitted jobs require more workers than are currently available in the cluster, then later jobs are queued until earlier jobs have finished. Queued jobs start when enough workers are available to run the job.

The default search paths of the workers might not be the same as that of your client MATLAB. To ensure that workers in the cluster have access to the needed files, such as code files, data files, or model files, specify paths to add to workers using the `AdditionalPaths` option.

To retrieve results after the job is finished, use the `fetchOutputs` function. `fetchOutputs` retrieves all variables in the batch worker workspace. When you submit batch jobs as a script, by default, workspace variables are copied from the client to workers. To avoid recursion of workspace variables, submit batch jobs as functions instead of as scripts.

You can use the `diary` to capture command line output while running batch jobs. This can be useful when executing the `trainNetwork` function with the `Verbose` option set to `true`.

Manage Cluster Profiles and Automatic Pool Creation

Parallel Computing Toolbox comes pre-configured with the cluster profile `local` for running parallel code on your local desktop machine. By default, MATLAB starts all parallel pools using the `local` cluster profile. If you want to run code on a remote cluster, you must start a parallel pool using the remote cluster profile. You can manage cluster profiles using the Cluster Profile Manager. For more information about managing cluster profiles, see "Discover Clusters and Use Cluster Profiles" (Parallel Computing Toolbox).

Some functions, including `trainNetwork`, `predict`, `classify`, `parfor`, and `parfeval` can automatically start a parallel pool. To take advantage of automatic parallel pool creation, set your desired cluster as the default cluster profile in the Cluster Profile Manager. Alternatively, you can create the pool manually and specify the desired cluster resource when you create the pool.

If you want to use multiple GPUs in a remote cluster to train multiple networks in parallel or for custom training loops, best practice is to manually start a parallel pool in the desired cluster with as many workers as available GPUs. For more information, see "Deep Learning with MATLAB on Multiple GPUs" on page 7-13.

Deep Learning Precision

For best performance, it is recommended to use a GPU for all deep learning workflows. Because single-precision and double-precision performance of GPUs can differ substantially, it is important to know in which precision computations are performed. Typically, GPUs offer much better performance for calculations in single precision.

If you only use a GPU for deep learning, then single-precision performance is one of the most important characteristics of a GPU. If you also use a GPU for other computations using Parallel

Computing Toolbox, then high double-precision performance is important. This is because many functions in MATLAB use double-precision arithmetic by default. For more information, see “Improve Performance Using Single Precision Calculations” (Parallel Computing Toolbox)

When you train a network using the `trainNetwork` function, or when you use prediction or validation functions with `DAGNetwork` and `SeriesNetwork` objects, the software performs these computations using single-precision, floating-point arithmetic. Functions for training, prediction, and validation include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

For custom training workflows, it is recommended to convert data to single precision for training and inference. If you use `minibatchqueue` to manage mini-batches, your data is converted to single precision by default.

See Also

`trainingOptions` | `minibatchqueue` | `trainNetwork` | **Deep Network Designer** | **Experiment Manager**

More About

- “Deep Learning with MATLAB on Multiple GPUs” on page 7-13
- “Deep Learning with Big Data” on page 7-17
- “Deep Learning in the Cloud” on page 7-10
- “Train Deep Learning Networks in Parallel” on page 5-109
- “Send Deep Learning Batch Job to Cluster” on page 7-39
- “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46
- “Upload Deep Learning Data to the Cloud” on page 7-53
- “Run Custom Training Loops on a GPU and in Parallel” on page 7-20

Deep Learning in the Cloud

If you do not have a suitable GPU available for training your deep neural networks, you can speed up your deep learning applications with one or more high-performance GPUs in the cloud. Working in the cloud requires some initial setup, but using cloud resources can significantly reduce training time or allow you to train more networks in the same amount of time.

You can accelerate training using one or more GPUs on a single machine or using a cluster of machines with GPUs. Train a single network using multiple GPUs, or train multiple models at once.

After you set up MATLAB or MATLAB Parallel Server in your chosen cloud platform, you can perform deep learning with minimal changes to the code you run on your local machine. For more information about adapting your deep learning code for different parallel environments, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2.

Note If you run MATLAB on a single machine in the cloud and you connect via ssh or remote desktop protocol (RDP), then network execution and training uses the same code as if you were running on your local machine.

Using a GPU or parallel options requires Parallel Computing Toolbox. Using a GPU also requires a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Using a remote cluster also requires MATLAB Parallel Server.

Access MATLAB in the Cloud

MathWorks® provides several ways of accessing MATLAB in public clouds such as Amazon® Web Services (AWS®) and Azure® that are configurable depending on your needs. To utilize public cloud offerings, you must have an account with your chosen cloud platform.

These cloud offerings make it easy for you to run MATLAB in the cloud by using pre-configured machine templates. You do not have to install MATLAB yourself.

The following tables shows some of the options for accessing MATLAB in the cloud.

Type of Resource	Cloud Solution	Additional Information	Learn More
Single machine	MATLAB Deep Learning Container	• Run container anywhere, including in the cloud or local hardware. • Customize and save container image. • Includes commonly used toolboxes for deep learning applications and workflows.	• MATLAB Deep Learning Container on NVIDIA GPU Cloud for Amazon Web Services • MATLAB Deep Learning Container on NVIDIA GPU Cloud for NVIDIA DGX

Type of Resource	Cloud Solution	Additional Information	Learn More
	Azure Marketplace	• Fully customizable. • Configure region and network settings. • Deploy into existing cloud infrastructure.	Run MATLAB from Azure Marketplace
	Reference architecture templates for AWS and Azure	• Fully customizable. • Configure region and network settings. • Deploy into existing cloud infrastructure.	• Run MATLAB on Amazon Web Services (Windows) • Run MATLAB on Amazon Web Services (Linux) • Run MATLAB on Microsoft Azure using Reference Architecture (Windows) • Run MATLAB on Microsoft Azure using Reference Architecture(Linux)
Cluster	MathWorks Cloud Center	• Cloud Center creates and manages clusters in your AWS account. • Connect to Cloud Center clusters in MATLAB Online. For more information, see “Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online” (Parallel Computing Toolbox).	MathWorks Cloud Center
	Azure Marketplace	• Fully customizable. • Configure region and network settings. • Deploy into existing cloud infrastructure.	“Run MATLAB Parallel Server from Microsoft Azure Marketplace” (MATLAB Parallel Server)

Type of Resource	Cloud Solution	Additional Information	Learn More
	Reference architecture templates for AWS and Azure	• Fully customizable. • Configure region and network settings. • Deploy into existing cloud infrastructure.	• "Run MATLAB Parallel Server on Amazon Web Services" (MATLAB Parallel Server) • "Run MATLAB Parallel Server on Microsoft Azure Using Reference Architecture" (MATLAB Parallel Server)

Work with Big Data in the Cloud

Storing data in the cloud can make it easier for you to access for cloud applications without needing to upload or download large amounts of data each time you create cloud resources. Both AWS and Azure offer data storage services, such as AWS S3 and Azure Blob Storage, respectively.

To avoid the time and cost associated with transferring large quantities of data, it is recommended that you set up cloud resources for your deep learning applications using the same cloud provider and region that you use to store your data in the cloud.

To access data stored in the cloud from MATLAB, you must configure your machine with your access credentials. You can configure access from inside MATLAB using environment variables. For more information on how to set environment variables to access cloud data from your client MATLAB, see "Work with Remote Data". For more information on how to set environment variables on parallel workers in a remote cluster, see "Set Environment Variables on Workers" (Parallel Computing Toolbox).

For an example showing how to upload data to the cloud, see "Upload Deep Learning Data to the Cloud" on page 7-53.

See Also

More About

- "Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud" on page 7-2
- "Deep Learning with MATLAB on Multiple GPUs" on page 7-13
- "Train Deep Learning Networks in Parallel" on page 5-109
- "Send Deep Learning Batch Job to Cluster" on page 7-39
- "Use parfeval to Train Multiple Deep Learning Networks" on page 7-32
- "Use parfor to Train Multiple Deep Learning Networks" on page 7-46
- "Upload Deep Learning Data to the Cloud" on page 7-53

Deep Learning with MATLAB on Multiple GPUs

MATLAB supports training a single deep neural network using multiple GPUs in parallel. This can be achieved using multiple GPUs on your local machine, or on a cluster or cloud using parallel workers with GPUs. Using multiple GPUs can speed up training significantly. To decide if you expect multi-GPU training to deliver a performance gain, consider the following factors:

- How long is the iteration on each GPU? If each GPU iteration is short, then the added overhead of communication between GPUs can dominate. Try increasing the computation per iteration by using a larger batch size.
- Are all the GPUs on a single machine? Communication between GPUs on different machines introduces a significant communication delay. You can mitigate this if you have suitable hardware. For more information, see “Advanced Support for Fast Multi-Node GPU Communication” on page 7-16.

Tip To train a single network using multiple GPUs on your local machine, you can simply specify the `ExecutionEnvironment` option as `"multi-gpu"` without changing the rest of your code. `trainNetwork` automatically uses your available GPUs for training computations.

When you train on a remote cluster, specify the `ExecutionEnvironment` option as `"parallel"`. If the cluster has access to one or more GPUs, then `trainNetwork` only uses the GPUs for training. Workers without a unique GPU are never used for training computation.

If you want to use more resources, you can scale up deep learning training to clusters or the cloud. To learn more about parallel options, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2. To try an example, see “Train Network in the Cloud Using Automatic Parallel Support” on page 7-28.

Using a GPU or parallel options requires Parallel Computing Toolbox. Using a GPU also requires a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Using a remote cluster also requires MATLAB Parallel Server.

Use Multiple GPUs in Local Machine

Note If you run MATLAB on a single machine in the cloud that you connect to via ssh or remote desktop protocol (RDP), then network execution and training uses the same code as if you were running on your local machine.

If you have access to a machine with multiple GPUs, you can simply specify the `ExecutionEnvironment` option as `"multi-gpu"`:

- For training using `trainNetwork`, use the `trainingOptions` function to set the `ExecutionEnvironment` name-value option to `"multi-gpu"`.
- For inference using `classify` and `predict`, set the `ExecutionEnvironment` name-value option to `"multi-gpu"`.

The `"multi-gpu"` option allows you to use multiple GPUs in a local parallel pool. If there is no current parallel pool, `trainNetwork`, `predict`, and `classify` automatically start a local parallel

pool using your default cluster profile settings. The pool has as many workers as the number of available GPUs.

For information on how to perform custom training using multiple GPUs in your local machine, see “Run Custom Training Loops on a GPU and in Parallel” on page 7-20.

Use Multiple GPUs in Cluster

For training and inference with multiple GPUs in a remote cluster, use the `"parallel"` option:

- For training using `trainNetwork`, use the `trainingOptions` function to set the `ExecutionEnvironment` name-value option to `"parallel"`.
- For inference using `classify` and `predict`, set the `ExecutionEnvironment` name-value option to `"parallel"`.

If there is no current parallel pool, `trainNetwork`, `predict`, and `classify` automatically start a parallel pool using your default cluster profile settings. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then training takes place on all available CPU workers instead.

For information on how to perform custom training using multiple GPUs in a remote cluster, see “Run Custom Training Loops on a GPU and in Parallel” on page 7-20.

Optimize Mini-Batch Size and Learning Rate

Convolutional neural networks are typically trained iteratively using mini-batches of images. This is because the whole dataset is usually too large to fit into GPU memory. For optimum performance, you can experiment with the mini-batch size by changing the `MiniBatchSize` name-value option using the `trainingOptions` function.

The optimal mini-batch size depends on your exact network, dataset, and GPU hardware. When training with multiple GPUs, each image batch is distributed between the GPUs. This effectively increases the total GPU memory available, allowing larger batch sizes. A recommended practice is to scale up the mini-batch size linearly with the number of GPUs, in order to keep the workload on each GPU constant. For example, if you are training on a single GPU using a mini-batch size of 64, and you want to scale up to training with four GPUs of the same type, you can increase the mini-batch size to 256 so that each GPU processes 64 observations per iteration.

Because increasing the mini-batch size improves the significance of each iteration, you can increase the learning rate. A good general guideline is to increase the learning rate proportionally to the increase in mini-batch size. Depending on your application, a larger mini-batch size and learning rate can speed up training without a decrease in accuracy, up to some limit.

Select Particular GPUs to Use for Training

If you do not want to use all of your GPUs, you can select the GPUs that you want to use for training and inference directly. Doing so can be useful to avoid training on a poor-performance GPU, for example, your display GPU.

If your GPUs are in your local machine, you can use the `gpuDeviceTable` and `gpuDeviceCount` functions to examine your GPU resources and determine the index of the GPUs you want to use.

For single GPU training with the `"auto"` or `"gpu"` options, by default, MATLAB uses the GPU device with index `1`. You can use a different GPU by selecting the device before you start training. Use `gpuDevice` to select the desired GPU using its index:

```
gpuDevice(index)
```

`trainNetwork`, `predict`, and `classify` automatically use the selected GPU when you set the `ExecutionEnvironment` option to `"auto"` or `"gpu"`.

For multiple GPU training with the `"multi-gpu"` option, by default, MATLAB uses all available GPUs in your local machine. If you want to exclude GPUs, you can start the parallel pool in advance and select the devices manually.

For example, suppose you have three GPUs but you only want to use the devices with indices `1` and `3`. You can use the following code to start a parallel pool with two workers and select one GPU on each worker.

```
useGPUs = [1 3];
parpool('local', numel(useGPUs));
spmd
    gpuDevice(useGPUs(labindex));
end
```

`trainNetwork`, `predict`, and `classify` automatically use the current parallel pool when you set the `ExecutionEnvironment` option to `"multi-gpu"` (or `"parallel"` for the same result).

Another option is to select workers using the `WorkerLoad` name-value argument in `trainingOptions`. For example:

```
parpool('local', 5);
opts = trainingOptions('sgdm', 'WorkerLoad', [1 1 1 0 1], ...)
```

In this case, the fourth worker is part of the pool but idle, which is not an ideal use of the parallel resources. It is more efficient to select GPUs for training manually using `gpuDevice`.

Train Multiple Networks on Multiple GPUs

If you want to train multiple models in parallel with one GPU each, start a parallel pool with one worker per available GPU, and train each network on a different worker. Use `parfor` or `parfeval` to simultaneously execute a network on each worker. Use the `trainingOptions` function to set the `ExecutionEnvironment` name-value option to `"gpu"` on each worker.

For example, use code of the following form to train multiple networks in parallel on all available GPUs:

```
options = trainingOptions("sgdm","ExecutionEnvironment","gpu");

parfor i=1:gpuDeviceCount("available")
    trainNetwork(…,options);
end
```

To run in the background without blocking your local MATLAB, use `parfeval`. For examples showing how to train multiple networks using `parfor` and `parfeval`, see

- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46

- “Use parfeval to Train Multiple Deep Learning Networks” on page 7-32

Advanced Support for Fast Multi-Node GPU Communication

Some multi-GPU features in MATLAB, including `trainNetwork`, are optimized for direct communication via fast interconnects for improved performance.

If you have appropriate hardware connections, then data transfer between multiple GPUs uses fast peer-to-peer communication, including NVLink, if available.

If you are using a Linux compute cluster with fast interconnects between machines such as Infiniband, or fast interconnects between GPUs on different machines, such as GPUDirect RDMA, you might be able to take advantage of fast multi-node support in MATLAB. Enable this support on all the workers in your pool by setting the environment variable `PARALLEL_SERVER_FAST_MULTINODE_GPU_COMMUNICATION` to `1`. Set this environment variable in the Cluster Profile Manager.

This feature is part of the NVIDIA NCCL library for GPU communication. To configure it, you must set additional environment variables to define the network interface protocol, especially `NCCL_SOCKET_IFNAME`. For more information, see the NCCL documentation and in particular the section on NCCL Environment Variables.

See Also

`trainNetwork` | `trainingOptions` | `gpuDevice` | `spmd` | `imageDatastore`

Related Examples

- “Train Deep Learning Networks in Parallel” on page 5-109
- “Upload Deep Learning Data to the Cloud” on page 7-53
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46
- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Run Custom Training Loops on a GPU and in Parallel” on page 7-20

Deep Learning with Big Data

Typically, training deep neural networks requires large amounts of data that often do not fit in memory. You do not need multiple computers to solve problems using data sets too large to fit in memory. Instead, you can divide your training data into mini-batches that contain a portion of the data set. By iterating over the mini-batches, networks can learn from large data sets without needing to load all data into memory at once.

If your data is too large to fit in memory, use a datastore to work with mini-batches of data for training and inference. MATLAB provides many different types of datastore tailored for different applications. For more information about datastores for different applications, see “Datastores for Deep Learning” on page 19-2.

`augmentedImageDatastore` is specifically designed to preprocess and augment batches of image data for machine learning and computer vision applications. For an example showing how to use `augmentedImageDatastore` to manage image data during training, see “Train Network with Augmented Images”

Work with Big Data in Parallel

If you want to use large amounts of data to train a network, it can be helpful to train in parallel. Doing so can reduce the time it takes to train a network, because you can train using multiple mini-batches at the same time.

It is recommended to train using a GPU or multiple GPUs. Only use single CPU or multiple CPUs if you do not have a GPU. CPUs are normally much slower that GPUs for both training and inference. Running on a single GPU typically offers much better performance than running on multiple CPU cores.

For more information about training in parallel, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2.

Preprocess Data in Background

When you train in parallel, you can fetch and preprocess your data in the background. This can be particularly useful if you want to preprocess your mini-batches during training, such as when using the `transform` function to apply a mini-batch preprocessing function to your datastore.

When you train a network using the `trainNetwork` function, you can fetch and preprocess data in the background by enabling background dispatch:

- Set the `DispatchInBackground` property of the datastore to `true`.
- Set the `'DispatchInBackground'` training option to `true` using the `trainingOptions` function.

During training, some workers are used for preprocessing data instead of network training computations. You can fine-tune the training computation and data dispatch loads between workers by specifying the `'WorkerLoad'` name-value argument using the `trainingOptions` function. For advanced options, you can try modifying the number of workers of the parallel pool.

You can use a built-in mini-batch datastore, such as `augmentedImageDatastore`, `denoisingImageDatastore`, or `pixelLabelImageDatastore`. You can also use a custom mini-

batch datastore with background dispatch enabled. For more information on creating custom mini-batch datastores, see “Develop Custom Mini-Batch Datastore” on page 19-36.

For more information about datastore requirement for background dispatching, see “Use Datastore for Parallel Training and Background Dispatching” on page 19-8

Work with Big Data in the Cloud

Storing data in the cloud can make it easier for you to access for cloud applications without needing to upload or download large amounts of data each time you create cloud resources. Both AWS and Azure offer data storage services, such as AWS S3 and Azure Blob Storage, respectively.

To avoid the time and cost associated with transferring large quantities of data, it is recommended that you set up cloud resources for your deep learning applications using the same cloud provider and region that you use to store your data in the cloud.

To access data stored in the cloud from MATLAB, you must configure your machine with your access credentials. You can configure access from inside MATLAB using environment variables. For more information on how to set environment variables to access cloud data from your client MATLAB, see “Work with Remote Data”. For more information on how to set environment variables on parallel workers in a remote cluster, see “Set Environment Variables on Workers” (Parallel Computing Toolbox).

For an example showing how to upload data to the cloud, see “Upload Deep Learning Data to the Cloud” on page 7-53.

For more information about deep learning in the cloud, see “Deep Learning in the Cloud” on page 7-10

Preprocess Data for Custom Training Loops

When you train a network using a custom training loop, you can process your data in the background by using `minibatchqueue` and enabling background dispatch. A `minibatchqueue` object iterates over a `datastore` to prepare mini-batches for custom training loops. Enable background dispatch when your mini-batches require heavy preprocessing.

To enable background dispatch, you must:

- Set the `DispatchInBackground` property of the datastore to `true`.
- Set the `DispatchInBackground` property of the `minibatchqueue` to `true`.

When you use this option, MATLAB opens a local parallel pool to use for preprocessing your data. Data preprocessing for custom training loops is supported when training using local resources only. For example, use this option when training using a single GPU in your local machine.

For more information about datastore requirements for background dispatching, see “Use Datastore for Parallel Training and Background Dispatching” on page 19-8

See Also

`trainingOptions` | `minibatchqueue` | `trainNetwork`

More About

- “Datastores for Deep Learning” on page 19-2
- “Data Sets for Deep Learning” on page 19-118
- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Deep Learning in the Cloud” on page 7-10
- “Deep Learning with MATLAB on Multiple GPUs” on page 7-13
- “Train Deep Learning Networks in Parallel” on page 5-109
- “Upload Deep Learning Data to the Cloud” on page 7-53

Run Custom Training Loops on a GPU and in Parallel

You can speed up your custom training loops by running on a GPU, in parallel using multiple GPUs, or on a cluster.

It is recommended to train using a GPU or multiple GPUs. Only use single CPU or multiple CPUs if you do not have a GPU. CPUs are normally much slower that GPUs for both training and inference. Running on a single GPU typically offers much better performance than running on multiple CPU cores.

Note This topic shows you how to perform custom training on GPUs, in parallel, and on the cloud. To learn about parallel and GPU workflows using the `trainNetwork` function, see:

- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Deep Learning with MATLAB on Multiple GPUs” on page 7-13

Using a GPU or parallel options requires Parallel Computing Toolbox. Using a GPU also requires a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Using a remote cluster also requires MATLAB Parallel Server.

Train Network on GPU

By default, custom training loops run on the CPU. Automatic differentiation using `dlgradient` and `dlfeval` supports running on the GPU when your data is on the GPU. To run a custom training loop on a GPU, simply convert your data to `gpuArray` during training.

You can use `minibatchqueue` to manage your data during training. `minibatchqueue` automatically prepares data for training, including custom preprocessing and converting data to `dlarray` and `gpuArray`. By default, `minibatchqueue` returns all mini-batch variables on the GPU if one is available. You can choose which variables to return on the GPU using the `OutputEnvironment` property.

For an example showing how to use `minibatchqueue` to train on the GPU, see “Train Network Using Custom Training Loop” on page 18-225.

Alternatively, you can manually convert your data to `gpuArray` within the training loop.

To easily specify the execution environment, create the variable `executionEnvironment` that contains either `"cpu"`, `"gpu"`, or `"auto"`.

```
executionEnvironment = "auto"
```

During training, after reading a mini-batch, check the execution environment option and convert the data to a `gpuArray` if necessary. The `canUseGPU` function checks for useable GPUs.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end
```

Train Single Network in Parallel

When you train in parallel, each worker trains the network simultaneously using a portion of a mini-batch. This means that you must combine the gradients, loss, and any state parameters after each iteration, according to the proportion of the mini-batch processed by each worker.

You can train in parallel on your local machine, or on a remote cluster, for example, in the cloud. Start a parallel pool in the desired resources and partition your data between the workers. During training, combine the gradients, loss, and state after each iteration so that the learnable parameters on each worker update in synchronization. For an example showing how to perform custom training in parallel, see “Train Network in Parallel with Custom Training Loop” on page 7-55

Set Up Parallel Environment

It is recommended to train using a GPU or multiple GPUs. Only use single CPU or multiple CPUs if you do not have a GPU. CPUs are normally much slower that GPUs for both training and inference. Running on a single GPU typically offers much better performance than running on multiple CPU cores.

Set up the parallel environment that you want to use before training. Start a parallel pool using your desired resources. For training using multiple GPUs, start a parallel pool with as many workers as available GPUs. For best performance, MATLAB automatically assigns a different GPU to each worker.

If you are using your local machine, you can use `canUseGPU` and `gpuDeviceCount` to determine if you have GPUs available. For example, to check availabilities of GPUs and start a parallel pool with as many workers as available GPUs, use the following code:

```
if canUseGPU
    executionEnvironment = "gpu";
    numberOfGPUs = gpuDeviceCount("available");
    pool = parpool(numberOfGPUs);
else
    executionEnvironment = "cpu";
    pool = parpool;
end
```

If you are running using a remote cluster, for example, a cluster in the cloud, start a parallel pool with as many workers as the number of GPUs per machine multiplied by the number of machines.

For more information on selecting specific GPUs, see “Select Particular GPUs to Use for Training” on page 7-14.

Specify Mini-Batch Size and Partition Data

Specify the mini-batch size that you want to use during training. For GPU training, a recommended practice is to scale up the mini-batch size linearly with the number of GPUs, in order to keep the workload on each GPU constant. For example, if you are training on a single GPU using a mini-batch size of 64, and you want to scale up to training with four GPUs of the same type, you can increase the mini-batch size to 256 so that each GPU processes 64 observations per iteration.

You can use the following code to scale up the mini-batch size by the number of workers, where `N` is the number of workers in your parallel pool.

```
if executionEnvironment == "gpu"
    miniBatchSize = miniBatchSize .* N
end
```

If you want to use a mini-batch size that not exactly divisible by the number of workers in your parallel pool, then distribute the remainder across the workers.

```
workerMiniBatchSize = floor(miniBatchSize ./ repmat(N,1,N));
remainder = miniBatchSize - sum(workerMiniBatchSize);
workerMiniBatchSize = workerMiniBatchSize + [ones(1,remainder) zeros(1,N-remainder)]
```

At the start of training, shuffle your data. Partition your data so that each worker has access to a portion of the mini-batch. To partition a datastore, use the `partition` function.

You can use `minibatchqueue` to manage the data on each worker during training. `minibatchqueue` automatically prepares data for training, including custom preprocessing and converting data to `dlarray` and `gpuArray`. Create a minibatchqueue on each worker using the partitioned datastore. Set the `MiniBatchSize` property using the mini-batch sizes calculated for each worker.

At the start of each training iteration, use the `gop` function to check that all worker `minibatchqueue` objects can return data. If any worker runs out of data, training stops. If your overall mini-batch size is not exactly divisible by the number of workers and you do not discard partial mini-batches, some workers might run out of data before others.

Write your training code inside an `spmd` block, so that the training loop executes on each worker.

```
spmd
    % Reset and shuffle the datastore.
    reset(augimdsTrain);
    augimdsTrain = shuffle(augimdsTrain);

    % Partition datastore.
    workerImds = partition(augimdsTrain,N,labindex);

    % Create minibatchqueue using partitioned datastore on each worker
    workerMbq = minibatchqueue(workerImds,...
        "MiniBatchSize",workerMiniBatchSize(labindex),...
        "MiniBatchFcn",@preprocessMiniBatch);

    ...

    for epoch = 1:numEpochs

        % Reset and shuffle minibatchqueue on each worker.
        shuffle(workerMbq);

        % Loop over mini-batches.
        while gop(@and,hasdata(workerMbq))

            % Custom training loop
            ...

        end
        ...
    end
end
```

Aggregate Gradients

To ensure that the network on each worker learns from all data and not just the data on that worker, aggregate the gradients and use the aggregated gradients to update the network on each worker.

For example, suppose you are training the network `dlnet`, using the model gradients function `modelGradients`. Your training loop contains the following code for evaluating the gradient, loss, and statistics on each worker:

```
[workerGradients,dlworkerLoss,workerState] = dlfeval(@modelGradients,dlnet,dlworkerX,workerY);
```

`dlworkerX` and `workerY` are the predictor and true response on each worker, respectively.

To aggregate the gradients, use a weighted sum. Define a helper function to sum the gradients.

```
function gradients = aggregateGradients(dlgradients,factor)
    gradients = extractdata(dlgradients);
    gradients = gplus(factor*gradients);
end
```

Inside the training loop, use `dlupdate` to apply the function to the gradients of each learnable parameter.

```
workerGradients.Value = dlupdate(@aggregateGradients,workerGradients.Value,{workerNormalizationFa
```

Aggregate Loss and Accuracy

To find the network loss and accuracy, for example, to plot them during training to monitor training progress, aggregate the values of the loss and accuracy on all of the workers. Typically, the aggregated value is the sum of the value on each worker, weighted by the proportion of the mini-batch used on each worker. To aggregate the losses and accuracy each iteration, calculate the weight factor for each worker and use `gplus` to sum the values on each worker.

```
workerNormalizationFactor = workerMiniBatchSize(labindex)./miniBatchSize;
loss = gplus(workerNormalizationFactor*extractdata(dlworkerLoss));
accuracy = gplus(workerNormalizationFactor*extractdata(dlworkerAccuracy));
```

Aggregate Statistics

If your network contains layers that track the statistics of your training data, such as batch normalization layers, then you must aggregate the statistics across all workers after each training iteration. Doing so ensures that the network learns statistics that are representative of the entire training set.

You can identify the layers that contain statistics information before training. For example, if you are using a `dlnetwork` with batch normalization layers, you can use the following code to find the relevant layers.

```
batchNormLayers = arrayfun(@(l)isa(l,'nnet.cnn.layer.BatchNormalizationLayer'),dlnet.Layers);
batchNormLayersNames = string({dlnet.Layers(batchNormLayers).Name});
state = dlnet.State;
isBatchNormalizationStateMean = ismember(state.Layer,batchNormLayersNames) & state.Parameter == '
isBatchNormalizationStateVariance = ismember(state.Layer,batchNormLayersNames) & state.Parameter
```

Define a helper function to aggregate the statistics you are using. Batch normalization layers track the mean and variance of the input data. You can aggregate the mean on all the workers using a weighted average. To calculate the aggregated variance s_c^2, use a formula of the following form.

$$s_c^2 = \frac{1}{M}\sum_{j=1}^{N} m_j\left(s_j^2 + \left(\bar{x}_j - \bar{x}_c\right)^2\right)$$

N is the total number of workers, M is the total number of observations in a mini-batch, m_j is the number of observations processed on the jth worker, $\bar{x}_j$ and s_j^2 are the mean and variance statistics calculated on that worker, and $\bar{x}_c$ is the aggregated mean across all workers.

```
function state = aggregateState(state,factor,...
    isBatchNormalizationStateMean,isBatchNormalizationStateVariance)

    stateMeans = state.Value(isBatchNormalizationStateMean);
    stateVariances = state.Value(isBatchNormalizationStateVariance);

    for j = 1:numel(stateMeans)
        meanVal = stateMeans{j};
        varVal = stateVariances{j};

        % Calculate combined mean
        combinedMean = gplus(factor*meanVal);

        % Calculate combined variance terms to sum
        varTerm = factor.*(varVal + (meanVal - combinedMean).^2);

        % Update state
        stateMeans{j} = combinedMean;
        stateVariances{j} = gplus(varTerm);
    end

    state.Value(isBatchNormalizationStateMean) = stateMeans;
    state.Value(isBatchNormalizationStateVariance) = stateVariances;
end
```

Inside the training loop, use the helper function to update the state of the batch normalization layers with the combined mean and variance.

```
dlnet.State = aggregateState(workerState,workerNormalizationFactor,...
                isBatchNormalizationStateMean,isBatchNormalizationStateVariance);
```

Plot Results During Training

If you want to plot results during training, you can send data from the workers to the client using a `DataQueue` object.

To easily specify that the plot should be on or off, create the variable `plots` that contains either `"training-progress"` or `"none"`.

```
plots = "training-progress";
```

Before training, initialize the `DataQueue` and the animated line using the `animatedline` function.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
```

```
    grid on
end
```

Create the `DataQueue` object. Use `afterEach` to call the helper function `displayTrainingProgress` each time data is sent from the worker to the client.

```
Q = parallel.pool.DataQueue;
displayFcn = @(x) displayTrainingProgress(x,lineLossTrain);
afterEach(Q,displayFcn);
```

The `displayTrainingProgress` helper function contains the code used to add points to the animated line and display the training epoch and duration.

```
function displayTrainingProgress (data,line)
     addpoints(line,double(data(3)),double(data(2)))
     D = duration(0,0,data(4),'Format','hh:mm:ss');
     title("Epoch: " + data(1) + ", Elapsed: " + string(D))
     drawnow
end
```

Inside the training loop, at the end of each epoch, use the `DataQueue` to send the training data from the workers to the client. At the end of each iteration, the aggregated loss is the same on each worker, so you can send data from a single worker.

```
% Display training progress information.
if labindex == 1
   data = [epoch loss iteration toc(start)];
   send(Q,gather(data));
end
```

Train Multiple Networks in Parallel

To train multiple networks in parallel, start a parallel pool in your desired resources and use `parfor` to train a single network on each worker.

You can run locally or using a remote cluster. Using a remote cluster requires MATLAB Parallel Server. For more information about managing cluster resources, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox). If you have multiple GPUs and want to exclude some from training, you can choose the GPUs you use to train on. For more information on selecting specific GPUs, see “Select Particular GPUs to Use for Training” on page 7-14.

You can modify the network or training parameters on each worker to perform parameter sweeps in parallel. For example, in `networks` is an array of `dlnetwork` objects, you can use code of the following form to train multiple different networks using the same data.

```
parpool ("local",numNetworks);

parfor idx = 1:numNetworks
    iteration = 0;
    velocity = [];

    % Allocate one network per worker
    dlnet = networks(idx)

    % Loop over epochs.
    for epoch = 1:numEpochs
        % Shuffle data.
```

```
        shuffle(mbq);

        % Loop over mini-batches.
        while hasdata(mbq)
            iteration = iteration + 1;

            % Custom training loop
            ...

        end

    end

    % Send the trained networks back to the client.
    trainedNetworks{idx} = dlnet;
end
```

After `parfor` finishes, `trainedNetworks` contains the resulting networks trained by the workers.

Plot Results During Training

To monitor training progress on the workers, you can use a `DataQueue` to send data back from the workers.

To easily specify that the plot should be on or off, create the variable `plots` that contains either `"training-progress"` or `"none"`.

```
plots = "training-progress";
```

Before training, initialize the `DataQueue` and the animated lines using the `animatedline` function. Create a subplot for each network you are training.

```
if plots == "training-progress"
    f = figure;
    f.Visible = true;
    for i=1:numNetworks
        subplot(numNetworks,1,i)
        xlabel('Iteration');
        ylabel('loss');
        lines(i) = animatedline;
    end
end
```

Create the `DataQueue` object. Use `afterEach` to call the helper function `displayTrainingProgress` each time data is sent from the worker to the client.

```
Q = parallel.pool.DataQueue;
displayFcn = @(x) displayTrainingProgress(x,lines);
afterEach(Q,displayFcn);
```

The `displayTrainingProgress` helper function contains the code used to add points to the animated lines.

```
function displayTrainingProgress (data,lines)
     addpoints(lines(1),double(data(4)),double(data(3)))
     D = duration(0,0,data(5),'Format','hh:mm:ss');
     title("Epoch: " + data(2) + ", Elapsed: " + string(D))
     drawnow limitrate nocallbacks
end
```

Inside the training loop, at the end of each iteration, use the `DataQueue` to send the training data from the workers to the client. Send the `parfor` loop index as well as the training information so that the points are added to the correct line for each worker.

```
% Display training progress information.
data = [idx epoch loss iteration toc(start)];
send(Q,gather(data));
```

Use Experiment Manager to Train in Parallel

You can use Experiment Manager to run your custom training loops in parallel. You can either run multiple trails at the same time, or run a single trial at a time using parallel resources.

To run multiple trials at the same time using one parallel worker for each trial, set up your custom training experiment an enable the **Use Parallel** option before running your experiment.

To run a single trial at a time using multiple parallel workers, define your parallel environment in your experiment training function and use an `spmd` block to train the network in parallel. For more information on training a single network in parallel with a custom training loop, see “Train Single Network in Parallel” on page 7-21.

For more information on training in parallel using Experiment Manager, see “Use Experiment Manager to Train in Parallel” on page 7-27.

See Also

`parfor` | `parfeval` | `gpuArray` | `dlarray` | `dlnetwork`

Related Examples

- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network in Parallel with Custom Training Loop” on page 7-55
- “Define Model Gradients Function for Custom Training Loop” on page 18-231

Train Network in the Cloud Using Automatic Parallel Support

This example shows how to train a convolutional neural network using MATLAB automatic support for parallel training. Deep learning training often takes hours or days. With parallel computing, you can speed up training using multiple graphical processing units (GPUs) locally or in a cluster in the cloud. If you have access to a machine with multiple GPUs, then you can complete this example on a local copy of the data. If you want to use more resources, then you can scale up deep learning training to the cloud. To learn more about your options for parallel training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2. This example guides you through the steps to train a deep learning network in a cluster in the cloud using MATLAB automatic parallel support.

Requirements

Before you can run the example, you need to configure a cluster and upload data to the cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. After that, upload your data to an Amazon S3 bucket and access it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see “Upload Deep Learning Data to the Cloud” on page 7-53.

Set Up Parallel Pool

Start a parallel pool in the cluster and set the number of workers to the number of GPUs in your cluster. If you specify more workers than GPUs, then the remaining workers are idle. This example assumes that the cluster you are using is set as the default cluster profile. Check the default cluster profile on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**.

```
numberOfWorkers = 8;
parpool(numberOfWorkers);
```

```
Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
connected to 8 workers.
```

Load Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. In this example, you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See “Upload Deep Learning Data to the Cloud” on page 7-53.

```
imdsTrain = imageDatastore('s3://cifar10cloud/cifar10/train', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

Train the network with augmented image data by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture and Training Options

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
blockDepth = 4; % blockDepth controls the depth of a convolutional block
netWidth = 32; % netWidth controls the number of filters in a convolutional block

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,blockDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];
```

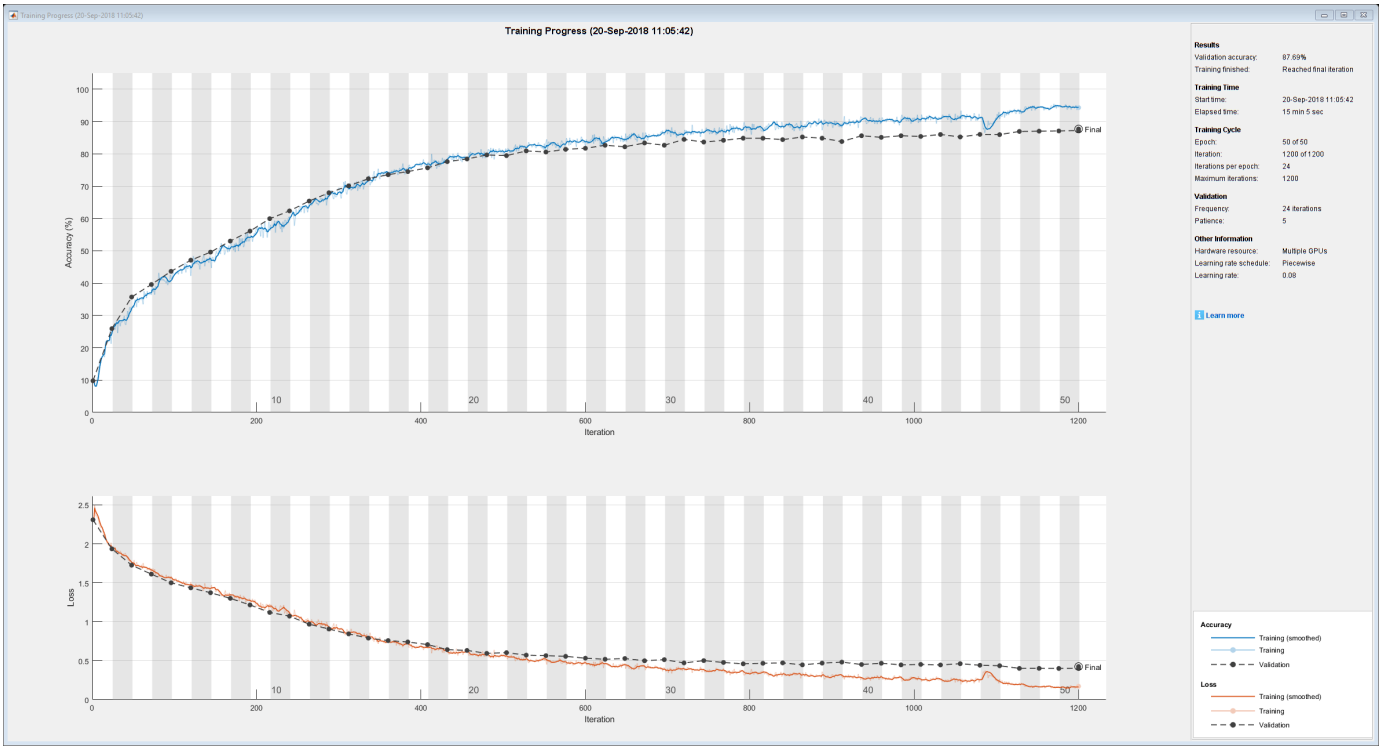
Define the training options. Train the network in parallel using the current cluster, by setting the execution environment to `parallel`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU constant. Scale the learning rate according to the mini-batch size. Use a learning rate schedule to drop the learning rate as the training progresses. Turn on the training progress plot to obtain visual feedback during training.

```
miniBatchSize = 256 * numberOfWorkers;
initialLearnRate = 1e-1 * miniBatchSize/256;

options = trainingOptions('sgdm', ...
    'ExecutionEnvironment','parallel', ... % Turn on automatic parallel support.
    'InitialLearnRate',initialLearnRate, ... % Set the initial learning rate.
    'MiniBatchSize',miniBatchSize, ... % Set the MiniBatchSize.
    'Verbose',false, ... % Do not send command line output.
    'Plots','training-progress', ... % Turn on the training progress plot.
    'L2Regularization',1e-10, ...
    'MaxEpochs',50, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',floor(numel(imdsTrain.Files)/miniBatchSize), ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',45);
```

Train Network and Use for Classification

Train the network in the cluster. During training, the plot displays the progress.

```
net = trainNetwork(augmentedImdsTrain,layers,options)
```




```
net =
  SeriesNetwork with properties:

    Layers: [43×1 nnet.cnn.layer.Layer]
```

Determine the accuracy of the network, by using the trained network to classify the test images on your local machine. Then compare the predicted labels to the actual labels.

```
YPredicted = classify(net,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

Define Helper Function

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same')
        batchNormalizationLayer
        reluLayer
    ];
```

```
    layers = repmat(layers,numConvLayers,1);
end
```

See Also

`trainNetwork` | `trainingOptions` | `imageDatastore`

Related Examples

- “Upload Deep Learning Data to the Cloud” on page 7-53
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46

Use parfeval to Train Multiple Deep Learning Networks

This example shows how to use `parfeval` to perform a parameter sweep on the depth of the network architecture for a deep learning network and retrieve data during training.

Deep learning training often takes hours or days, and searching for good architectures can be difficult. With parallel computing, you can speed up and automate your search for good models. If you have access to a machine with multiple graphical processing units (GPUs), you can complete this example on a local copy of the data set with a local parallel pool. If you want to use more resources, you can scale up deep learning training to the cloud. This example shows how to use `parfeval` to perform a parameter sweep on the depth of a network architecture in a cluster in the cloud. Using `parfeval` allows you to train in the background without blocking MATLAB, and provides options to stop early if results are satisfactory. You can modify the script to do a parameter sweep on any other parameter. Also, this example shows how to obtain feedback from the workers during computation by using `DataQueue`.

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the Cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see “Upload Deep Learning Data to the Cloud” on page 7-53.

Load Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. Split the training data set into training and validation sets, and keep the test data set to test the best network from the parameter sweep. In this example, you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See “Upload Deep Learning Data to the Cloud” on page 7-53.

```
imds = imageDatastore('s3://cifar10cloud/cifar10/train', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
```

Train the network with augmented image data by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
```

```
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Train Several Networks Simultaneously

Define the training options. Set the mini-batch size and scale the initial learning rate linearly according to the mini-batch size. Set the validation frequency so that `trainNetwork` validates the network once per epoch.

```
miniBatchSize = 128;
initialLearnRate = 1e-1 * miniBatchSize/256;
validationFrequency = floor(numel(imdsTrain.Labels)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ... % Set the mini-batch size
    'Verbose',false, ... % Do not send command line output.
    'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
    'L2Regularization',1e-10, ...
    'MaxEpochs',30, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency', validationFrequency);
```

Specify the depths for the network architecture on which to do a parameter sweep. Perform a parallel parameter sweep training several networks simultaneously using `parfeval`. Use a loop to iterate through the different network architectures in the sweep. Create the helper function `createNetworkArchitecture` at the end of the script, which takes an input argument to control the depth of the network and creates an architecture for CIFAR-10. Use `parfeval` to offload the computations performed by `trainNetwork` to a worker in the cluster. `parfeval` returns a future variable to hold the trained networks and training information when computations are done.

```
netDepths = 1:4;
for idx = 1:numel(netDepths)
    networksFuture(idx) = parfeval(@trainNetwork,2, ...
        augmentedImdsTrain,createNetworkArchitecture(netDepths(idx)),options);
end
```

```
Starting parallel pool (parpool) using the 'MyCluster' profile ...
Connected to the parallel pool (number of workers: 4).
```

`parfeval` does not block MATLAB, which means you can continue executing commands. In this case, obtain the trained networks and their training information by using `fetchOutputs` on `networksFuture`. The `fetchOutputs` function waits until the future variables finish.

```
[trainedNetworks,trainingInfo] = fetchOutputs(networksFuture);
```

Obtain the final validation accuracies of the networks by accessing the `trainingInfo` structure.

```
accuracies = [trainingInfo.FinalValidationAccuracy]
```

```
accuracies = 1×4

   72.5600   77.2600   79.4000   78.6800
```

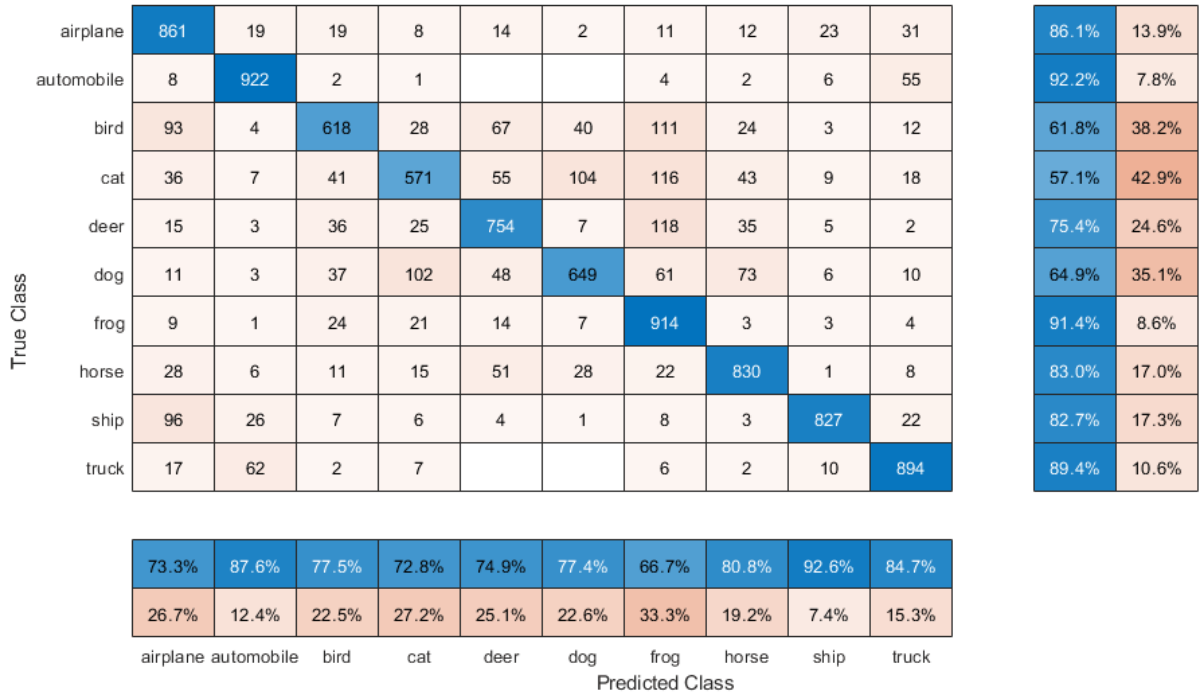
Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, I] = max(accuracies);
bestNetwork = trainedNetworks(I(1));
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.7840
```

Calculate the confusion matrix for the test data.

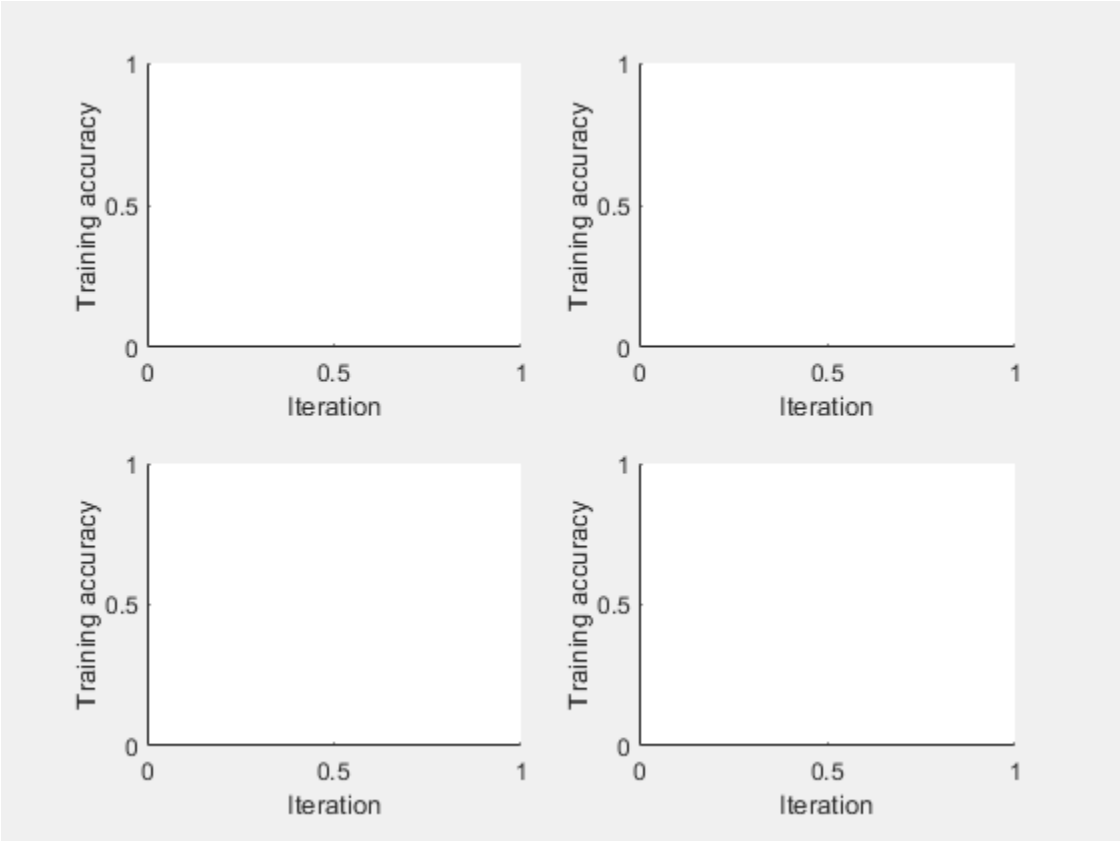
```
figure('Units','normalized','Position',[0.2 0.2 0.4 0.4]);
confusionchart(imdsTest.Labels,YPredicted,'RowSummary','row-normalized','ColumnSummary','column-n
```



Send Feedback Data During Training

Prepare and initialize plots that show the training progress in each of the workers. Use `animatedLine` for a convenient way to show changing data.

```
f = figure;
f.Visible = true;
for i=1:4
    subplot(2,2,i)
    xlabel('Iteration');
    ylabel('Training accuracy');
    lines(i) = animatedline;
end
```



Send the training progress data from the workers to the client by using `DataQueue`, and then plot the data. Update the plots each time the workers send training progress feedback by using `afterEach`. The parameter `opts` contains information about the worker, training iteration, and training accuracy.

```
D = parallel.pool.DataQueue;
afterEach(D, @(opts) updatePlot(lines, opts{:}));
```

Specify the depths for the network architecture on which to do a parameter sweep, and perform the parallel parameter sweep using `parfeval`. Allow the workers to access any helper function in this script, by adding the script to the current pool as an attached file. Define an output function in the training options to send the training progress from the workers to the client. The training options depend on the index of the worker and must be included inside the `for` loop.

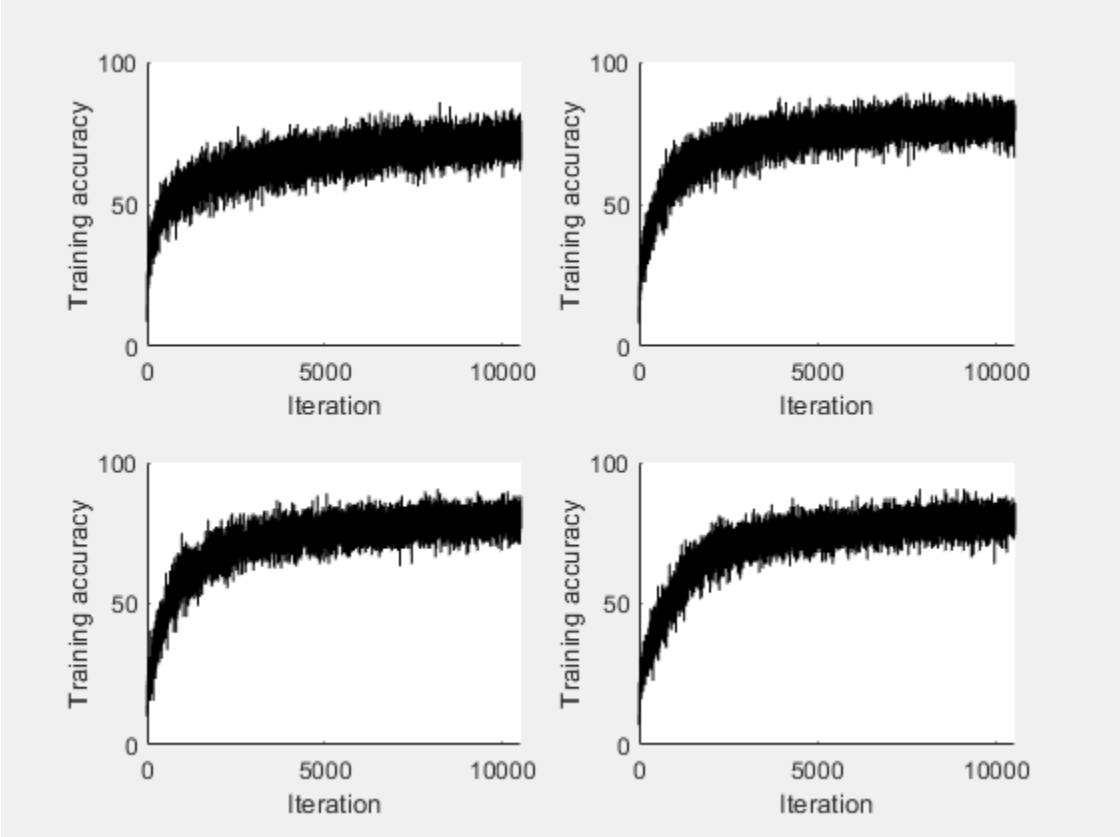
```
netDepths = 1:4;
addAttachedFiles(gcp,mfilename);
for idx = 1:numel(netDepths)

    miniBatchSize = 128;
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the mini
    validationFrequency = floor(numel(imdsTrain.Labels)/miniBatchSize);

    options = trainingOptions('sgdm', ...
        'OutputFcn',@(state) sendTrainingProgress(D,idx,state), ... % Set an output function to s
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
```

```
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'ValidationFrequency', validationFrequency);

    networksFuture(idx) = parfeval(@trainNetwork,2, ...
        augmentedImdsTrain,createNetworkArchitecture(netDepths(idx)),options);
end
```

`parfeval` invokes `trainNetwork` on a worker in the cluster. Computations happen on the background, so you can continue working in MATLAB. If you want to stop a `parfeval` computation, you can call `cancel` on its corresponding future variable. For example, if you observe that a network is underperforming, you can cancel its future. When you do so, the next queued future variable starts its computations.

In this case, fetch the trained networks and their training information by invoking `fetchOutputs` on the future variables.

```
[trainedNetworks,trainingInfo] = fetchOutputs(networksFuture);
```



Obtain the final validation accuracy for each network.

```
accuracies = [trainingInfo.FinalValidationAccuracy]
```

```
accuracies = 1×4
```

```
   72.9200   77.4800   76.9200   77.0400
```

Helper Functions

Define a network architecture for the CIFAR-10 data set with a function, and use an input argument to adjust the depth of the network. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
function layers = createNetworkArchitecture(netDepth)
imageSize = [32 32 3];
netWidth = round(16/sqrt(netDepth)); % netWidth controls the number of filters in a convolutiona

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,netDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
    ];
end
```

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
layers = [
    convolution2dLayer(3,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    ];

layers = repmat(layers,numConvLayers,1);
end
```

Define a function to send the training progress to the client through `DataQueue`.

```
function sendTrainingProgress(D,idx,info)
if info.State == "iteration"
    send(D,{idx,info.Iteration,info.TrainingAccuracy});
end
end
```

Define an update function to update the plots when a worker sends an intermediate result.

```
function updatePlot(lines,idx,iter,acc)
addpoints(lines(idx),iter,acc);
drawnow limitrate nocallbacks
end
```

See Also

`parfeval` | `afterEach` | `trainNetwork` | `trainingOptions` | `imageDatastore`

Related Examples

- “Train Network in the Cloud Using Automatic Parallel Support” on page 7-28
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46
- “Upload Deep Learning Data to the Cloud” on page 7-53

Send Deep Learning Batch Job to Cluster

This example shows how to send deep learning training batch jobs to a cluster so that you can continue working or close MATLAB during training.

Training deep neural networks often takes hours or days. To use time efficiently, you can train neural networks as batch jobs and fetch the results from the cluster when they are ready. You can continue working in MATLAB while computations take place or close MATLAB and obtain the results later using the Job Monitor. This example sends the parallel parameter sweep in “Use parfor to Train Multiple Deep Learning Networks” on page 7-46 as a batch job. After the job is complete, you can fetch the trained networks and compare their accuracies.

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the Cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see “Upload Deep Learning Data to the Cloud” on page 7-53.

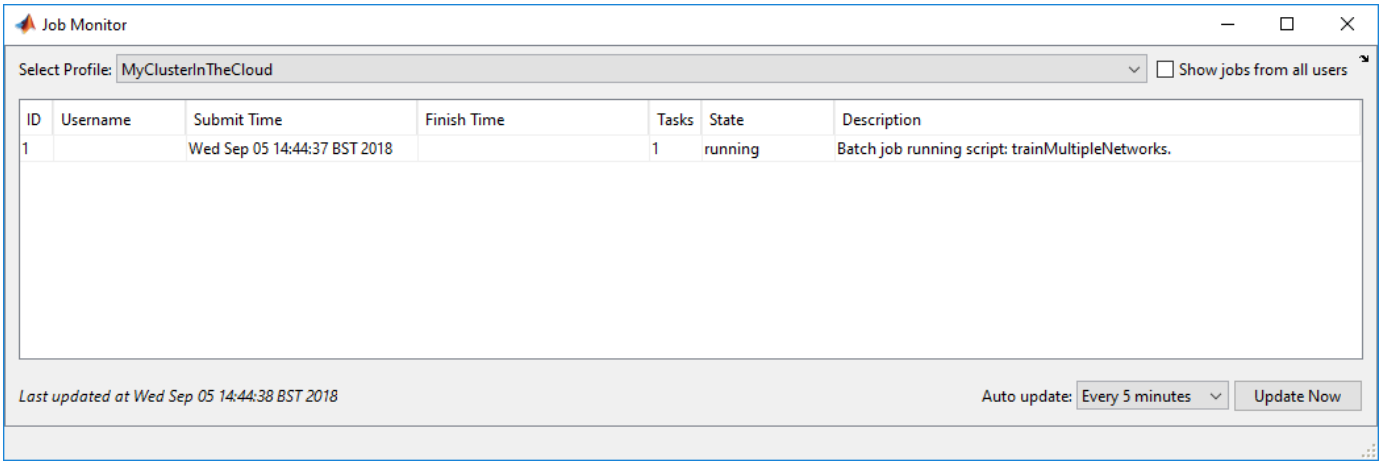
Submit Batch Job

Send a script as a batch job to the cluster by using the `batch` function. The cluster allocates one worker to execute the contents of your script. If the parallel code in the script benefits from extra workers, for example, it includes automatic parallel support or a parfor loop, you need to request the workers explicitly. `batch` uses one worker for the client running the script. You can specify more workers by using the 'Pool' name-value pair argument.

In this case, send the `trainMultipleNetworks` script to the cluster. This script contains the parallel parameter sweep in “Use parfor to Train Multiple Deep Learning Networks” on page 7-46. Because the script contains a `parfor` loop, specify 4 extra workers with the `Pool` name-value pair argument.

```
totalNumberOfWorkers = 5;
job1 = batch('trainMultipleNetworks', ...
    'Pool',totalNumberOfWorkers-1);
```

You can see the current status of your job in the cluster by checking the Job Monitor. In the **Environment** section on the **Home** tab, select **Parallel > Monitor Jobs** to open the Job Monitor.



You can submit additional jobs to the cluster. If the cluster is not available because it is running other jobs, any new job you submit remains queued until the cluster becomes available.

Fetch Results Programmatically

After submitting jobs to the cluster, you can continue working in MATLAB while computations take place. If the rest of your code depends on completion of a job, block MATLAB by using the `wait` command. In this case, wait for the job to finish.

```
wait(job1);
```

After the job finishes, fetch the results by using the `load` function. In this case, fetch the trained networks from the parallel parameter sweep in the submitted script and their accuracies.

```
load(job1,'accuracies');
accuracies
```

```
accuracies = 4×1

    0.8312
    0.8276
    0.8288
    0.8258
```

```
load(job1,'trainedNetworks');
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

To load all the variables in the batch job, use the `load` function without arguments.

```
load(job1);
```

If you close MATLAB, you can still recover the job in the cluster to fetch the results either while the computation is taking place or after the computation is complete. Before closing MATLAB, make a note of the job ID and then retrieve the job later by using the `findJob` function.

To retrieve a job, first create a cluster object for your cluster by using the `parcluster` function. Then, provide the job ID to `findJob`. In this case, the job ID is `1`.

```
c = parcluster('MyClusterInTheCloud');
job = findJob(c,'ID',1);
```

Delete the job when you are done. The job is removed from the Job Monitor.

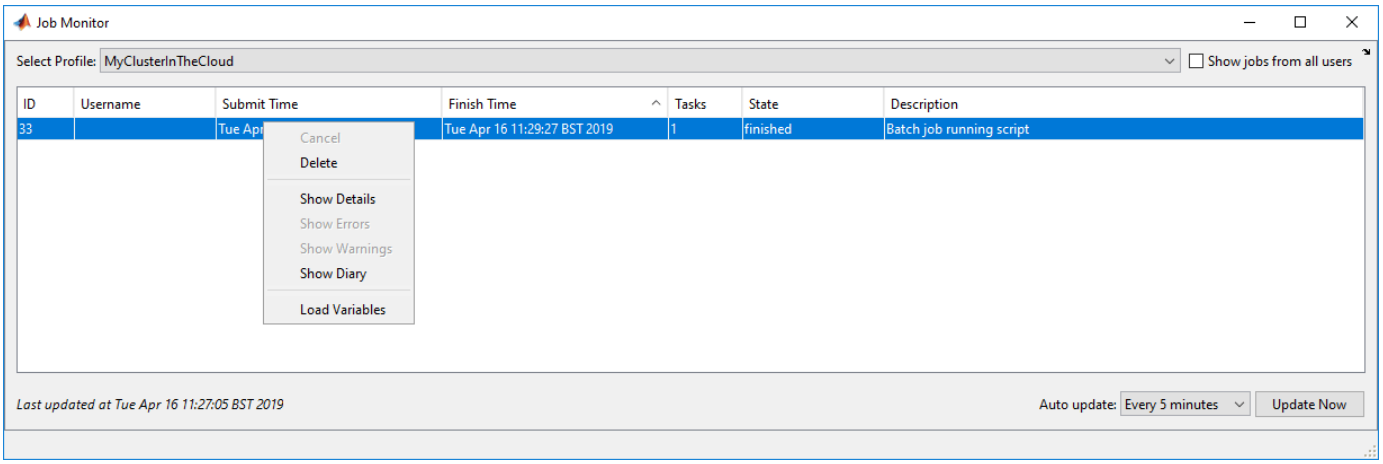
```
delete(job1);
```

Use Job Monitor to Fetch Results

When you submit batch jobs, all the computations happen in the cluster and you can safely close MATLAB. You can check the status of your jobs by using the Job Monitor in another MATLAB session.

When a job is done, you can retrieve the results from the Job Monitor. In the **Environment** section on the **Home** tab, select **Parallel > Monitor Jobs** to open the Job Monitor. Then right-click a job to display the context menu. From this menu, you can:

- Load the job into the workspace by clicking **Show Details**
- Load all variables in the job by clicking **Load Variables**
- Delete the job when you are done by clicking **Delete**



See Also

`batch`

Related Examples

- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46
- “Upload Deep Learning Data to the Cloud” on page 7-53

More About

- “Batch Processing” (Parallel Computing Toolbox)

Train Network Using Automatic Multi-GPU Support

This example shows how to use multiple GPUs on your local machine for deep learning training using automatic parallel support. Training deep learning networks often takes hours or days. With parallel computing, you can speed up training using multiple GPUs. To learn more about options for parallel training, see “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2.

Requirements

Before you can run this example, you must download the CIFAR-10 data set to your local machine. The following code downloads the data set to your current directory. If you already have a local copy of CIFAR-10, then you can skip this section.

```
directory = pwd;
[locationCifar10Train,locationCifar10Test] = downloadCIFARToFolders(directory);
```

```
Downloading CIFAR-10 data set...done.
Copying CIFAR-10 to folders...done.
```

Load Data Set

Load the training and test data sets by using an `imageDatastore` object. In the following code, ensure that the location of the datastores points to CIFAR-10 in your local machine.

```
imdsTrain = imageDatastore(locationCifar10Train, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');

imdsTest = imageDatastore(locationCifar10Test, ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

To train the network with augmented image data, create an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

Define Network Architecture and Training Options

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
blockDepth = 4; % blockDepth controls the depth of a convolutional block.
netWidth = 32; % netWidth controls the number of filters in a convolutional block.

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,blockDepth)
```

```
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,blockDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,blockDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
];
```

Define the training options. Train the network in parallel with multiple GPUs by setting the execution environment to `'multi-gpu'`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU constant. In this example, the number of GPUs is two. Scale the learning rate according to the mini-batch size. Use a learning rate schedule to drop the learning rate as the training progresses. Turn on the training progress plot to obtain visual feedback during training.

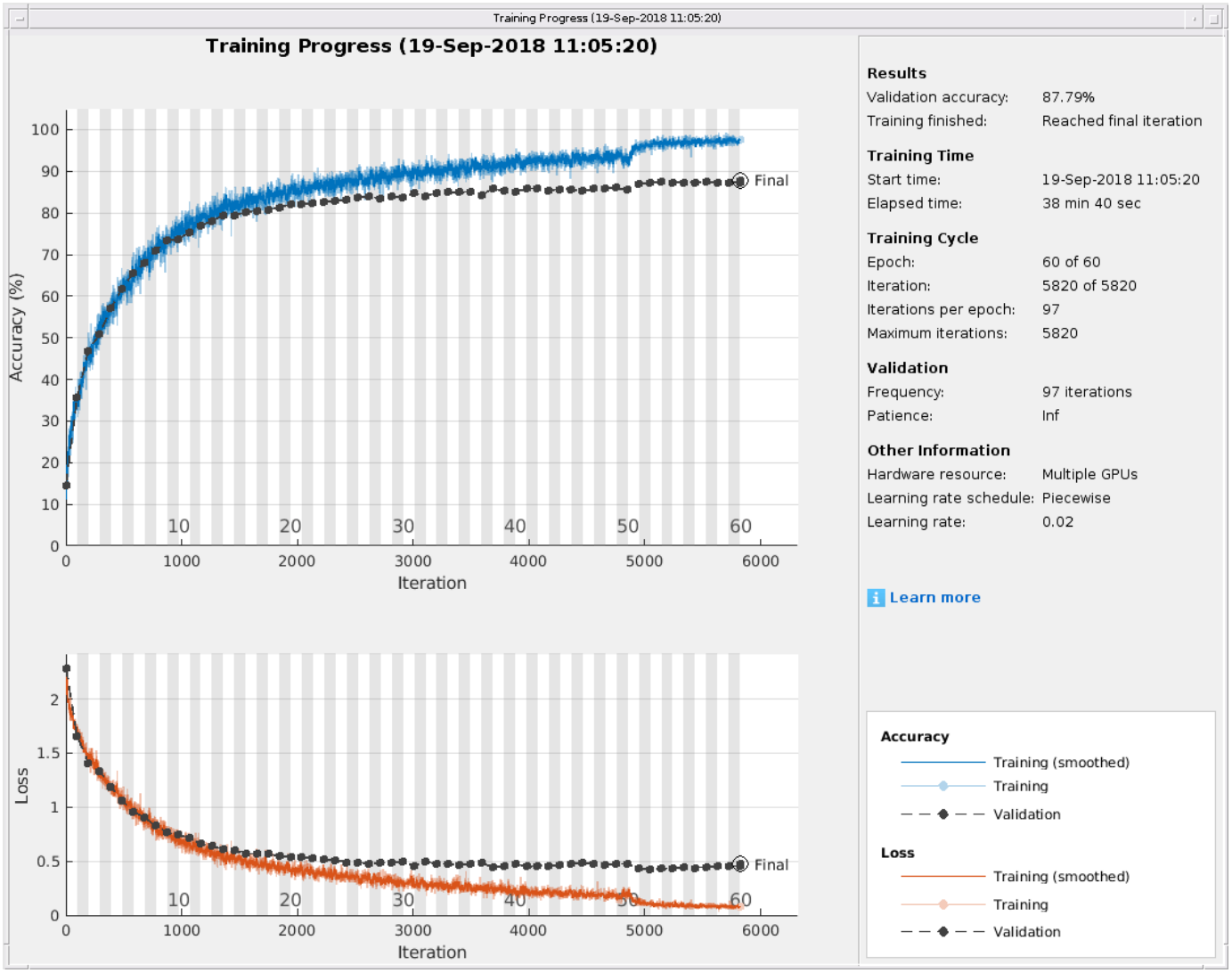
```
numGPUs = 2;
miniBatchSize = 256*numGPUs;
initialLearnRate = 1e-1*miniBatchSize/256;

options = trainingOptions('sgdm', ...
    'ExecutionEnvironment','multi-gpu', ... % Turn on automatic multi-gpu support.
    'InitialLearnRate',initialLearnRate, ... % Set the initial learning rate.
    'MiniBatchSize',miniBatchSize, ... % Set the MiniBatchSize.
    'Verbose',false, ... % Do not send command line output.
    'Plots','training-progress', ... % Turn on the training progress plot.
    'L2Regularization',1e-10, ...
    'MaxEpochs',60, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',floor(numel(imdsTrain.Files)/miniBatchSize), ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',50);
```

Train Network and Use for Classification

Train the network. During training, the plot displays the progress.

```
net = trainNetwork(augmentedImdsTrain,layers,options)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```



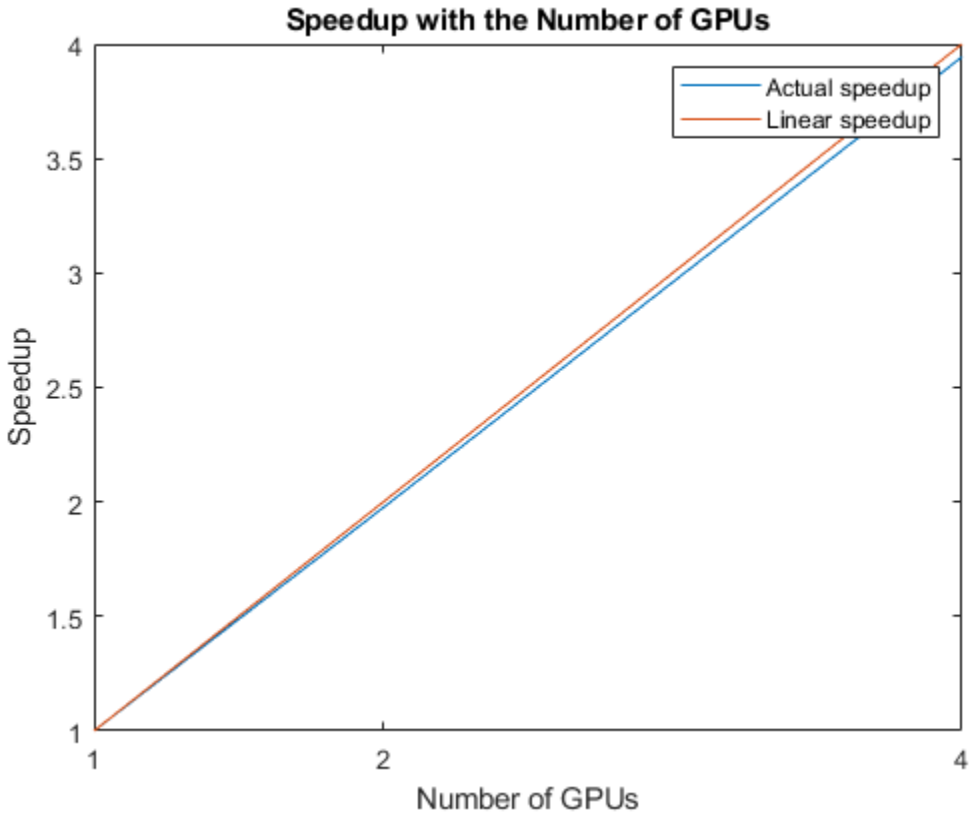

```
net =
  SeriesNetwork with properties:

    Layers: [43×1 nnet.cnn.layer.Layer]
```

Determine the accuracy of the network by using the trained network to classify the test images on your local machine. Then compare the predicted labels to the actual labels.

```
YPredicted = classify(net,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.8779
```

Automatic multi-GPU support can speed up network training by taking advantage of several GPUs. The following plot shows the speedup in the overall training time with the number of GPUs on a Linux machine with four NVIDIA© TITAN Xp GPUs.



Define Helper Function

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same')
        batchNormalizationLayer
        reluLayer];

    layers = repmat(layers,numConvLayers,1);
end
```

See Also

`trainNetwork` | `trainingOptions` | `imageDatastore`

Related Examples

- “Train Network in the Cloud Using Automatic Parallel Support” on page 7-28
- “Scale Up Deep Learning in Parallel, on GPUs, and in the Cloud” on page 7-2

Use parfor to Train Multiple Deep Learning Networks

This example shows how to use a `parfor` loop to perform a parameter sweep on a training option.

Deep learning training often takes hours or days, and searching for good training options can be difficult. With parallel computing, you can speed up and automate your search for good models. If you have access to a machine with multiple graphical processing units (GPUs), you can complete this example on a local copy of the data set with a local parpool. If you want to use more resources, you can scale up deep learning training to the cloud. This example shows how to use a parfor loop to perform a parameter sweep on the training option `MiniBatchSize` in a cluster in the cloud. You can modify the script to do a parameter sweep on any other training option. Also, this example shows how to obtain feedback from the workers during computation using `DataQueue`. You can also send the script as a batch job to the cluster, so you can continue working or close MATLAB and fetch the results later. For more information, see “Send Deep Learning Batch Job to Cluster” on page 7-39.

Requirements

Before you can run this example, you need to configure a cluster and upload your data to the cloud. In MATLAB, you can create clusters in the cloud directly from the MATLAB Desktop. On the **Home** tab, in the **Parallel** menu, select **Create and Manage Clusters**. In the Cluster Profile Manager, click **Create Cloud Cluster**. Alternatively, you can use MathWorks Cloud Center to create and access compute clusters. For more information, see Getting Started with Cloud Center. For this example, ensure that your cluster is set as default on the MATLAB **Home** tab, in **Parallel > Select a Default Cluster**. After that, upload your data to an Amazon S3 bucket and use it directly from MATLAB. This example uses a copy of the CIFAR-10 data set that is already stored in Amazon S3. For instructions, see “Upload Deep Learning Data to the Cloud” on page 7-53.

Load the Data Set from the Cloud

Load the training and test data sets from the cloud using `imageDatastore`. Split the training data set into training and validation sets, and keep the test data set to test the best network from the parameter sweep. In this example you use a copy of the CIFAR-10 data set stored in Amazon S3. To ensure that the workers have access to the datastore in the cloud, make sure that the environment variables for the AWS credentials are set correctly. See “Upload Deep Learning Data to the Cloud” on page 7-53.

```
imds = imageDatastore('s3://cifar10cloud/cifar10/train', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

imdsTest = imageDatastore('s3://cifar10cloud/cifar10/test', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9);
```

Train the network with augmented image data, by creating an `augmentedImageDatastore` object. Use random translations and horizontal reflections. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
```

```
    'RandYTranslation',pixelRange);
augmentedImdsTrain = augmentedImageDatastore(imageSize,imdsTrain, ...
    'DataAugmentation',imageAugmenter, ...
    'OutputSizeMode','randcrop');
```

Define Network Architecture

Define a network architecture for the CIFAR-10 data set. To simplify the code, use convolutional blocks that convolve the input. The pooling layers downsample the spatial dimensions.

```
imageSize = [32 32 3];
netDepth = 2; % netDepth controls the depth of a convolutional block
netWidth = 16; % netWidth controls the number of filters in a convolutional block

layers = [
    imageInputLayer(imageSize)

    convolutionalBlock(netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(2*netWidth,netDepth)
    maxPooling2dLayer(2,'Stride',2)
    convolutionalBlock(4*netWidth,netDepth)
    averagePooling2dLayer(8)

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer
    ];
```

Train Several Networks Simultaneously

Specify the mini-batch sizes on which to do a parameter sweep. Allocate variables for the resulting networks and accuracy.

```
miniBatchSizes = [64 128 256 512];
numMiniBatchSizes = numel(miniBatchSizes);
trainedNetworks = cell(numMiniBatchSizes,1);
accuracies = zeros(numMiniBatchSizes,1);
```

Perform a parallel parameter sweep training several networks inside a `parfor` loop and varying the mini-batch size. The workers in the cluster train the networks simultaneously and send the trained networks and accuracies back when the training is complete. If you want to check that the training is working, set `Verbose` to `true` in the training options. Note that the workers compute independently, so the command line output is not in the same sequential order as the iterations.

```
parfor idx = 1:numMiniBatchSizes

    miniBatchSize = miniBatchSizes(idx);
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the mini

    % Define the training options. Set the mini-batch size.
    options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
```

```
        'ValidationData',imdsValidation, ...
        'LearnRateSchedule','piecewise', ...
        'LearnRateDropFactor',0.1, ...
        'LearnRateDropPeriod',25);

    % Train the network in a worker in the cluster.
    net = trainNetwork(augmentedImdsTrain,layers,options);

    % To obtain the accuracy of this network, use the trained network to
    % classify the validation images on the worker and compare the predicted labels to the
    % actual labels.
    YPredicted = classify(net,imdsValidation);
    accuracies(idx) = sum(YPredicted == imdsValidation.Labels)/numel(imdsValidation.Labels);

    % Send the trained network back to the client.
    trainedNetworks{idx} = net;
end
```

```
Starting parallel pool (parpool) using the 'MyClusterInTheCloud' profile ...
Connected to the parallel pool (number of workers: 4).
```

After `parfor` finishes, `trainedNetworks` contains the resulting networks trained by the workers. Display the trained networks and their accuracies.

```
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

```
accuracies
```

```
accuracies = 4×1

    0.8188
    0.8232
    0.8162
    0.8050
```

Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, I] = max(accuracies);
bestNetwork = trainedNetworks{I(1)};
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

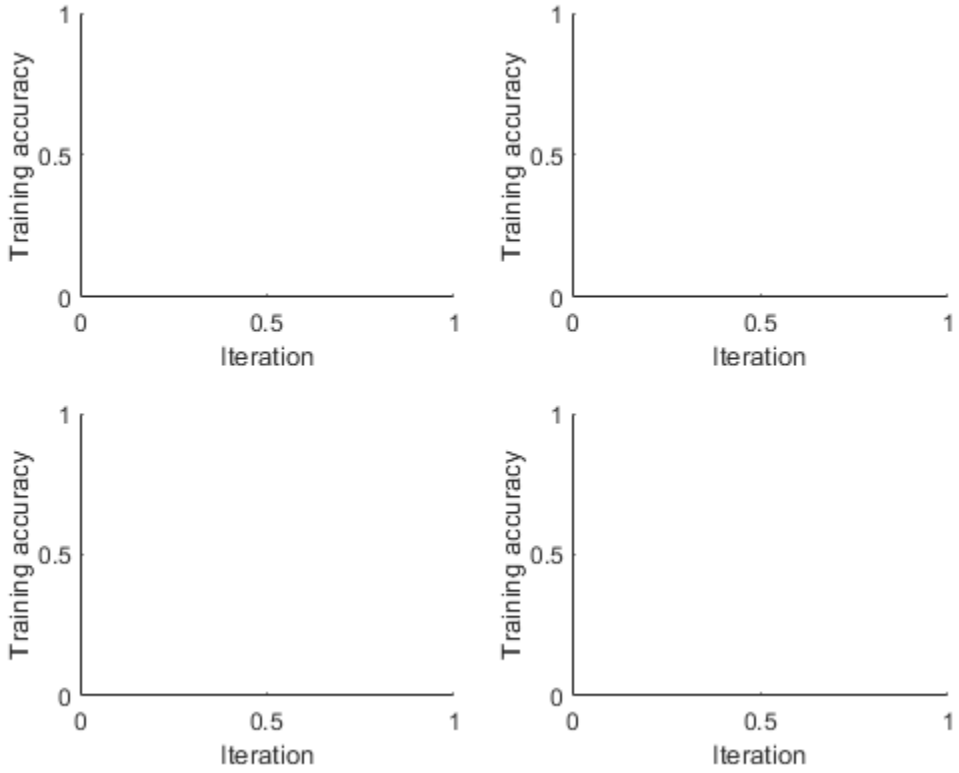
```
accuracy = 0.8173
```

Send Feedback Data During Training

Prepare and initialize plots that show the training progress in each of the workers. Use `animatedLine` for a convenient way to show changing data.

```
f = figure;
f.Visible = true;
```

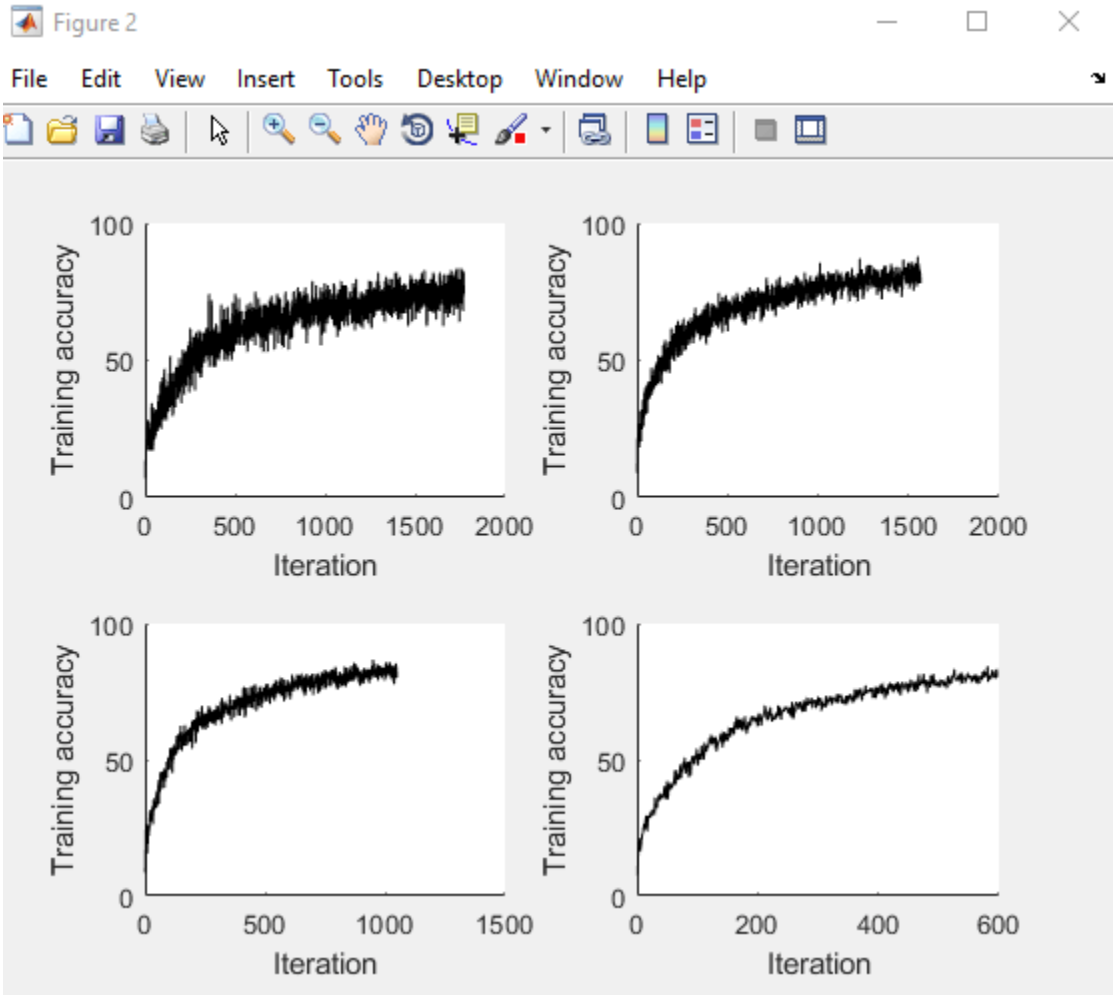
```
for i=1:4
    subplot(2,2,i)
    xlabel('Iteration');
    ylabel('Training accuracy');
    lines(i) = animatedline;
end
```



Send the training progress data from the workers to the client by using `DataQueue`, and then plot the data. Update the plots each time the workers send training progress feedback by using `afterEach`. The parameter `opts` contains information about the worker, training iteration, and training accuracy.

```
D = parallel.pool.DataQueue;
afterEach(D, @(opts) updatePlot(lines, opts{:}));
```

Perform a parallel parameter sweep training several networks inside a parfor loop with different mini-batch sizes. Note the use of `OutputFcn` in the training options to send the training progress to the client each iteration. This figure shows the training progress of four different workers during an execution of the following code.




```
parfor idx = 1:numel(miniBatchSizes)

    miniBatchSize = miniBatchSizes(idx);
    initialLearnRate = 1e-1 * miniBatchSize/256; % Scale the learning rate according to the miniB

    % Define the training options. Set an output function to send data back
    % to the client each iteration.
    options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ... % Set the corresponding MiniBatchSize in the sweep.
        'Verbose',false, ... % Do not send command line output.
        'InitialLearnRate',initialLearnRate, ... % Set the scaled learning rate.
        'OutputFcn',@(state) sendTrainingProgress(D,idx,state), ... % Set an output function to s
        'L2Regularization',1e-10, ...
        'MaxEpochs',30, ...
        'Shuffle','every-epoch', ...
        'ValidationData',imdsValidation, ...
        'LearnRateSchedule','piecewise', ...
        'LearnRateDropFactor',0.1, ...
        'LearnRateDropPeriod',25);

    % Train the network in a worker in the cluster. The workers send
    % training progress information during training to the client.
```

```
    net = trainNetwork(augmentedImdsTrain,layers,options);

    % To obtain the accuracy of this network, use the trained network to
    % classify the validation images on the worker and compare the predicted labels to the
    % actual labels.
    YPredicted = classify(net,imdsValidation);
    accuracies(idx) = sum(YPredicted == imdsValidation.Labels)/numel(imdsValidation.Labels);

    % Send the trained network back to the client.
    trainedNetworks{idx} = net;
end
```

```
Analyzing and transferring files to the workers ...done.
```

After `parfor` finishes, `trainedNetworks` contains the resulting networks trained by the workers. Display the trained networks and their accuracies.

```
trainedNetworks
```

```
trainedNetworks = 4×1 cell array
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
    {1×1 SeriesNetwork}
```

```
accuracies
```

```
accuracies = 4×1

    0.8214
    0.8172
    0.8132
    0.8084
```

Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, I] = max(accuracies);
bestNetwork = trainedNetworks{I(1)};
YPredicted = classify(bestNetwork,imdsTest);
accuracy = sum(YPredicted == imdsTest.Labels)/numel(imdsTest.Labels)
```

```
accuracy = 0.8187
```

Helper Functions

Define a function to create a convolutional block in the network architecture.

```
function layers = convolutionalBlock(numFilters,numConvLayers)
layers = [
    convolution2dLayer(3,numFilters,'Padding','same')
    batchNormalizationLayer
    reluLayer
    ];

layers = repmat(layers,numConvLayers,1);
end
```

Define a function to send the training progress to the client through `DataQueue`.

```
function sendTrainingProgress(D,idx,info)
if info.State == "iteration"
    send(D,{idx,info.Iteration,info.TrainingAccuracy});
end
end
```

Define an update function to update the plots when a worker sends an intermediate result.

```
function updatePlot(lines,idx,iter,acc)
addpoints(lines(idx),iter,acc);
drawnow limitrate nocallbacks
end
```

See Also

`trainNetwork` | `parallel.pool.DataQueue` | `imageDatastore`

Related Examples

- “Upload Deep Learning Data to the Cloud” on page 7-53
- “Send Deep Learning Batch Job to Cloud Cluster” on page 7-39

More About

- “Parallel for-Loops (parfor)” (Parallel Computing Toolbox)

Upload Deep Learning Data to the Cloud

This example shows how to upload data to an Amazon S3 bucket.

Before you can perform deep learning training in the cloud, you need to upload your data to the cloud. The example shows how to download the CIFAR-10 data set to your computer, and then upload the data to an Amazon S3 bucket for later use in MATLAB. The CIFAR-10 data set is a labeled image data set commonly used for benchmarking image classification algorithms. Before running this example, you need access to an Amazon Web Services (AWS) account. After you upload the data set to Amazon S3, you can try any of the examples in “Deep Learning in Parallel and in the Cloud”.

Download CIFAR-10 to Local Machine

Specify a local directory in which to download the data set. The following code creates a folder in your current directory containing all the images in the data set.

```
directory = pwd;
[trainDirectory,testDirectory] = downloadCIFARToFolders(directory);
```

```
Downloading CIFAR-10 data set...done.
Copying CIFAR-10 to folders...done.
```

Upload Local Data Set to Amazon S3 Bucket

To work with data in the cloud, you can upload to Amazon S3 and then use datastores to access the data in S3 from the workers in your cluster. The following steps describe how to upload the CIFAR-10 data set from your local machine to an Amazon S3 bucket.

1. For efficient file transfers to and from Amazon S3, download and install the AWS Command Line Interface tool from `https://aws.amazon.com/cli/`.

2. Specify your AWS Access Key ID, Secret Access Key, and Region of the bucket as system environment variables. Contact your AWS account administrator to obtain your keys.

For example, on Linux, macOS, or Unix, specify these variables:

```
export AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
export AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
export AWS_DEFAULT_REGION="us-east-1"
```

On Windows, specify these variables:

```
set AWS_ACCESS_KEY_ID="YOUR_AWS_ACCESS_KEY_ID"
set AWS_SECRET_ACCESS_KEY="YOUR_AWS_SECRET_ACCESS_KEY"
set AWS_DEFAULT_REGION="us-east-1"
```

To specify these environment variables permanently, set them in your user or system environment.

3. Create a bucket for your data by using either the AWS S3 web page or a command such as the following:

```
aws s3 mb s3://mynewbucket
```

4. Upload your data using a command such as the following:

```
aws s3 cp mylocaldatapath s3://mynewbucket --recursive
```

For example:

```
aws s3 cp path/to/CIFAR10/in/the/local/machine s3://MyExampleCloudData/cifar10/ --recursive
```

5. Copy your AWS credentials to your cluster workers by completing these steps in MATLAB:

a. In the **Environment** section on the **Home** tab, select **Parallel > Create and Manage Clusters**.

b. In the **Cluster Profile** pane of the Cluster Profile Manager, select your cloud cluster profile.

c. In the **Properties** tab, select the **EnvironmentVariables** property, scrolling as necessary to find the property.

d. At the bottom right of the window, click **Edit**.

e. Click in the box to the right of **EnvironmentVariables**, and then type these three variables, each on its own line: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION`.

f. At the bottom right of the window, click **Done**.

For information on how to create a cloud cluster, see “Create Cloud Cluster” (Parallel Computing Toolbox).

Use Data Set in MATLAB

After you store your data in Amazon S3, you can use datastores to access the data from your cluster workers. Simply create a datastore pointing to the URL of the S3 bucket. The following sample code shows how to use an `imageDatastore` to access an S3 bucket. Replace `'s3://MyExampleCloudData/cifar10/train'` with the URL of your S3 bucket.

```
imds = imageDatastore('s3://MyExampleCloudData/cifar10/train', ...
 'IncludeSubfolders',true, ...
 'LabelSource','foldernames');
```

With the CIFAR-10 data set now stored in Amazon S3, you can try any of the examples in “Deep Learning in Parallel and in the Cloud” that show how to use CIFAR-10 in different use cases.

See Also

`imageDatastore`

Related Examples

- “Use parfor to Train Multiple Deep Learning Networks” on page 7-46

Train Network in Parallel with Custom Training Loop

This example shows how to set up a custom training loop to train a network in parallel. In this example, parallel workers train on portions of the overall mini-batch. If you have a GPU, then training happens on the GPU. During training, a `DataQueue` object sends training progress information back to the MATLAB client.

Load Data Set

Load the digit data set and create an image datastore for the data set. Split the datastore into training and test datastores in a randomized way. Create an `augmentedImageDatastore` containing the training data.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsTest] = splitEachLabel(imds,0.9,"randomized");

inputSize = [28 28 1];
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain);
```

Determine the different classes in the training set.

```
classes = categories(imdsTrain.Labels);
numClasses = numel(classes);
```

Define Network

Define your network architecture and make it into a layer graph by using the `layerGraph` function. This network architecture includes batch normalization layers, which track the mean and variance statistics of the data set. When training in parallel, combine the statistics from all of the workers at the end of each iteration step, to ensure the network state reflects the whole mini-batch. Otherwise, the network state can diverge across the workers. If you are training stateful recurrent neural networks (RNNs), for example, using sequence data that has been split into smaller sequences to train networks containing LSTM or GRU layers, you must also manage the state between the workers.

```
layers = [
    imageInputLayer([28 28 1],'Name','input','Normalization','none')
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')];

lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph. `dlnetwork` objects allow for training with custom loops.

```
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [11×1 nnet.cnn.layer.Layer]
    Connections: [10×2 table]
     Learnables: [14×3 table]
          State: [6×3 table]
     InputNames: {'input'}
    OutputNames: {'fc'}
    Initialized: 1
```

Set Up Parallel Environment

Determine if GPUs are available for MATLAB to use with the `canUseGPU` function.

- If there are GPUs available, then train on the GPUs. Create a parallel pool with as many workers as GPUs.
- If there are no GPUs available, then train on the CPUs. Create a parallel pool with the default number of workers.

```
if canUseGPU
    executionEnvironment = "gpu";
    numberOfGPUs = gpuDeviceCount("available");
    pool = parpool(numberOfGPUs);
else
    executionEnvironment = "cpu";
    pool = parpool;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
```

Get the number of workers in the parallel pool. Later in this example, you divide the workload according to this number.

```
N = pool.NumWorkers;
```

Train Model

Specify the training options.

```
numEpochs = 20;
miniBatchSize = 128;
velocity = [];
```

For GPU training, a recommended practice is to scale up the mini-batch size linearly with the number of GPUs, in order to keep the workload on each GPU constant. For more related advice, see “Deep Learning with MATLAB on Multiple GPUs” on page 7-13.

```
if executionEnvironment == "gpu"
    miniBatchSize = miniBatchSize .* N
end
```

```
miniBatchSize = 512
```

Calculate the mini-batch size for each worker by dividing the overall mini-batch size evenly among the workers. Distribute the remainder across the first workers.

```
workerMiniBatchSize = floor(miniBatchSize ./ repmat(N,1,N));
remainder = miniBatchSize - sum(workerMiniBatchSize);
workerMiniBatchSize = workerMiniBatchSize + [ones(1,remainder) zeros(1,N-remainder)]
```

```
workerMiniBatchSize = 1×4

   128   128   128   128
```

This network contains batch normalization layers that keep track of the mean and variance of the data the network is trained on. Since each worker processes a portion of each mini-batch during each iteration, the mean and variance must be aggregated across all the workers. Find the indices of the mean and variance state parameters of the batch normalization layers in the network state property.

```
batchNormLayers = arrayfun(@(l)isa(l,'nnet.cnn.layer.BatchNormalizationLayer'),dlnet.Layers);
batchNormLayersNames = string({dlnet.Layers(batchNormLayers).Name});
state = dlnet.State;
isBatchNormalizationStateMean = ismember(state.Layer,batchNormLayersNames) & state.Parameter == '
isBatchNormalizationStateVariance = ismember(state.Layer,batchNormLayersNames) & state.Parameter
```

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

To send data back from the workers during training, create a `DataQueue` object. Use `afterEach` to set up a function, `displayTrainingProgress`, to call each time a worker sends data. `displayTrainingProgress` is a supporting function, defined at the end of this example, that displays the training progress information that comes from the workers.

```
Q = parallel.pool.DataQueue;
displayFcn = @(x) displayTrainingProgress(x,lineLossTrain);
afterEach(Q,displayFcn);
```

Train the model using a custom parallel training loop, as detailed in the following steps. To execute the code simultaneously on all the workers, use an `spmd` block. Within the `spmd` block, `labindex` gives the index of the worker currently executing the code.

Before training, partition the datastore for each worker by using the `partition` function. Use the partitioned datastore to create a `minibatchqueue` on each worker. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to normalize the data, convert the labels to one-hot encoded variables, and determine the number of observations in the mini-batch.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or the number of observations.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a

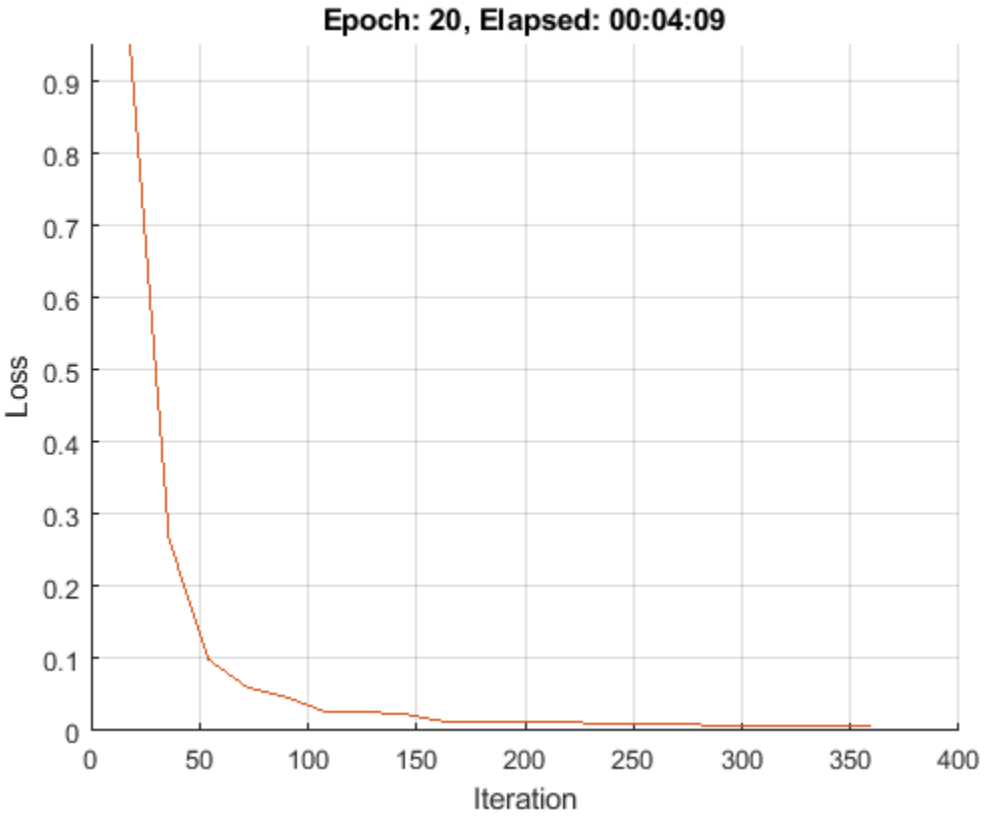
supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox) (Parallel Computing Toolbox).

For each epoch, reset and shuffle the datastore with the `reset` and `shuffle` functions. For each iteration in the epoch:

- Ensure that all workers have data available before beginning processing it in parallel, by performing a global `and` operation (`gop`) on the result of the `hasdata` function.
- Read a mini-batch from the `minibatchqueue` by using the `next` function.
- Compute the gradients and the loss of the network on each worker by calling `dlfeval` on the `modelGradients` function. The `dlfeval` function evaluates the helper function `modelGradients` with automatic differentiation enabled, so `modelGradients` can compute the gradients with respect to the loss in an automatic way. `modelGradients` is defined at the end of the example and returns loss and gradients given a network, mini-batch of data, and true labels.
- To obtain the overall loss, aggregate the losses on all workers. This example uses cross entropy for the loss function, and the aggregated loss is the sum of all losses. Before aggregating, normalize each loss by multiplying by the proportion of the overall mini-batch that the worker is working on. Use `gplus` to add all losses together and replicate the results across workers.
- To aggregate and update the gradients of all workers, use the `dlupdate` function with the `aggregateGradients` function. `aggregateGradients` is a supporting function defined at the end of this example. This function uses `gplus` to add together and replicate gradients across workers, following normalization according to the proportion of the overall mini-batch that each worker is working on.
- Aggregate the state of the network on all workers using the `aggregateState` function. `aggregateState` is a supporting function defined at the end of this example. The batch normalization layers in the network track the mean and variance of the data. Since the complete mini-batch is spread across multiple workers, aggregate the network state after each iteration to compute the mean and variance of the whole minibatch.
- After computing the final gradients, update the network learnable parameters with the `sgdmupdate` function.
- Send training progress information back to the client by using the `send` function with the `DataQueue`. Use only one worker to send data, because all workers have the same loss information. To ensure that data is on the CPU, so that a client machine without a GPU can access it, use `gather` on the `dlarray` before sending it.

```
start = tic;
spmd
    % Reset and shuffle the datastore.
    reset(augimdsTrain);
    augimdsTrain = shuffle(augimdsTrain);

    % Partition datastore.
    workerImds = partition(augimdsTrain,N,labindex);

    % Create minibatchqueue using partitioned datastore on each worker
    workerMbq = minibatchqueue(workerImds,3,...
        "MiniBatchSize",workerMiniBatchSize(labindex),...
        "MiniBatchFcn",@preprocessMiniBatch,...
        "MiniBatchFormat",{'SSCB','',''});

    workerVelocity = velocity;
```

```
    iteration = 0;

    for epoch = 1:numEpochs
        shuffle(workerMbq);

        % Loop over mini-batches.
        while gop(@and,hasdata(workerMbq))
            iteration = iteration + 1;

            % Read a mini-batch of data.
            [dlworkerX,workerY,workerNumObservations] = next(workerMbq);

            % Evaluate the model gradients and loss on the worker.
            [workerGradients,dlworkerLoss,workerState] = dlfeval(@modelGradients,dlnet,dlworkerX

            % Aggregate the losses on all workers.
            workerNormalizationFactor = workerMiniBatchSize(labindex)./miniBatchSize;
            loss = gplus(workerNormalizationFactor*extractdata(dlworkerLoss));

            % Aggregate the network state on all workers
            dlnet.State = aggregateState(workerState,workerNormalizationFactor,...
                isBatchNormalizationStateMean,isBatchNormalizationStateVariance);

            % Aggregate the gradients on all workers.
            workerGradients.Value = dlupdate(@aggregateGradients,workerGradients.Value,{workerNo

            % Update the network parameters using the SGDM optimizer.
            [dlnet.Learnables,workerVelocity] = sgdmupdate(dlnet.Learnables,workerGradients,worke
        end

       % Display training progress information.
       if labindex == 1
           data = [epoch loss iteration toc(start)];
           send(Q,gather(data));
       end
    end
end
```



Test Model

After you train the network, you can test its accuracy.

Load the test images into memory by using `readall` on the test datastore, concatenate them, and normalize them.

```
XTest = readall(imdsTest);
XTest = cat(4,XTest{:});
XTest = single(XTest) ./ 255;
YTest = imdsTest.Labels;
```

After the training is complete, all workers have the same complete trained network. Retrieve any of them.

```
dlnetFinal = dlnet{1};
```

To classify images using a `dlnetwork` object, use the `predict` function on a `dlarray`.

```
dlYPredScores = predict(dlnetFinal,dlarray(XTest,'SSCB'));
```

From the predicted scores, find the class with the highest score with the `max` function. Before you do that, extract the data from the `dlarray` with the `extractdata` function.

```
[~,idx] = max(extractdata(dlYPredScores),[],1);
YPred = classes(idx);
```

To obtain the classification accuracy of the model, compare the predictions on the test set against the true labels.

```
accuracy = mean(YPred==YTest)
```

```
accuracy = 0.8960
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Determine the number of observations in the mini-batch
2. Preprocess the images using the `preprocessMiniBatchPredictors` function.
3. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.
4. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,numObs] = preprocessMiniBatch(XCell,YCell)

numObs = numel(YCell);

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate.
Y = cat(2,YCell{1:end});

% One-hot encode labels.
Y = onehotencode(Y,1);

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension. The data are then normalized.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

% Normalize.
X =  X ./ 255;

end
```

Model Gradients Functions

Define a function, `modelGradients`, to compute the gradients of the loss with respect to the learnable parameters of the network. This function computes the network outputs for a mini-batch `X` with `forward` and `softmax` and calculates the loss, given the true outputs, using cross entropy.

When you call this function with `dlfeval`, automatic differentiation is enabled, and `dlgradient` can compute the gradients of the loss with respect to the learnables automatically.

```
function [dlgradients,dlloss,state] = modelGradients(dlnet,dlX,dlY)
    [dlYPred,state] = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);

    dlloss = crossentropy(dlYPred,dlY);
    dlgradients = dlgradient(dlloss,dlnet.Learnables);
end
```

Display Training Progress Function

Define a function to display training progress information that comes from the workers. The `DataQueue` in this example calls this function every time a worker sends data.

```
function displayTrainingProgress (data,line)
     addpoints(line,double(data(3)),double(data(2)))
     D = duration(0,0,data(4),'Format','hh:mm:ss');
     title("Epoch: " + data(1) + ", Elapsed: " + string(D))
     drawnow
end
```

Aggregate Gradients Function

Define a function that aggregates the gradients on all workers by adding them together. `gplus` adds together and replicates all the gradients on the workers. Before adding them together, normalize them by multiplying them by a factor that represents the proportion of the overall mini-batch that the worker is working on. To retrieve the contents of a `dlarray`, use `extractdata`.

```
function gradients = aggregateGradients(dlgradients,factor)
    gradients = extractdata(dlgradients);
    gradients = gplus(factor*gradients);
end
```

Aggregate State Function

Define a function that aggregates the network state on all workers. The network state contains the trained batch normalization statistics of the data set. Since each worker only sees a portion of the mini-batch, aggregate the network state so that the statistics are representative of the statistics across all the data. For each mini-batch, the combined mean is calculated as a weighted average of the mean across the workers for each iteration. The combined variance is calculated according to the following formula:

$$s_c^2 = \frac{1}{M}\sum_{j=1}^{N} m_j\left[s_j^2 + \left(\bar{x}_j - \bar{x}_c\right)^2\right]$$

where $N$is the total number of workers, $M$is the total number of observations in a mini-batch, m_j is the number of observations processed on the jth worker, $\bar{x}_j$ and s_j^2 are the mean and variance statistics calculated on that worker, and $\bar{x}_c$ is the combined mean across all workers.

```
function state = aggregateState(state,factor,...
    isBatchNormalizationStateMean,isBatchNormalizationStateVariance)

    stateMeans = state.Value(isBatchNormalizationStateMean);
    stateVariances = state.Value(isBatchNormalizationStateVariance);
```

```
    for j = 1:numel(stateMeans)
        meanVal = stateMeans{j};
        varVal = stateVariances{j};

        % Calculate combined mean
        combinedMean = gplus(factor*meanVal);

        % Calculate combined variance terms to sum
        varTerm = factor.*(varVal + (meanVal - combinedMean).^2);

        % Update state
        stateMeans{j} = combinedMean;
        stateVariances{j} = gplus(varTerm);
    end

    state.Value(isBatchNormalizationStateMean) = stateMeans;
    state.Value(isBatchNormalizationStateVariance) = stateVariances;
end
```

See Also

`dlarray` | `dlnetwork` | `sgdmupdate` | `dlupdate` | `dlfeval` | `dlgradient` | `crossentropy` | `softmax` | `forward` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Train Network Using Federated Learning” on page 7-64
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Automatic Differentiation Background” on page 18-200

Train Network Using Federated Learning

This example shows how to train a network using federated learning. Federated learning is a technique that enables you to train a network in a distributed, decentralized way [1].

Federated learning allows you to train a model using data from different sources without moving the data to a central location, even if the individual data sources do not match the overall distribution of the data set. This is known as non-independent and identically distributed (non-IID) data. Federated learning can be especially useful when the training data is large, or when there are privacy concerns about transferring the training data.

Instead of distributing data, the federated learning technique trains multiple models, each in the same location as a data source. You can create a global model that has learned from all the data sources by periodically collecting and combining the learnable parameters of the locally trained models. In this way, you can train a global model without centrally processing any training data.

This example uses federated learning to train a classification model in parallel using a highly non-IID dataset. The model is trained using the digits data set, which consists of 10000 handwritten images of the numbers 0 to 9. The example runs in parallel using 10 workers, each processing images of a single digit. By averaging the learnable parameters of the networks after each round of training, the models on each worker improve performance across all classes, without ever processing data of the other classes.

While data privacy is one of the applications of federated learning, this example does not deal with the details of maintaining data privacy and security. This example demonstrates the basic federated learning algorithm.

Set Up Parallel Environment

Create a parallel pool with the same number of workers as classes in the data set. For this example, use a local parallel pool with 10 workers.

```
pool = parpool('local',10);
numWorkers = pool.NumWorkers;
```

Load Data Set

All data used in this example is initially stored in a centralized location. To make this data highly non-IID, you need to distribute the data among the workers according to class. To create validation and test data sets, transfer a portion of data from the workers to the client. After the data is correctly set up, with training data of individual classes on the workers and test and validation data of all classes on the client, there is no further transfer of data during training.

Specify the folder containing the image data.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos',...
    'nndatasets','DigitDataset');
```

Distribute the data among the workers. Each worker receives images of only one digit, such that worker 1 receives all the images of the number 0, worker 2 receives images of the number 1, etc.

Images of each digit are stored in a separate folder with the name of that digit. On each worker, use the `fullfile` function to specify the path to a specific class folder. Then, create an `imageDatastore` that contains all images of that digit. Next, use the `splitEachLabel` function to

randomly separate 30% of the data for use in validation and testing. Finally, create an `augmentedImageDatastore` containing the training data.

```
inputSize = [28 28 1];
spmd
    digitDatasetPath = fullfile(digitDatasetPath,num2str(labindex - 1));
    imds = imageDatastore(digitDatasetPath,...
        'IncludeSubfolders',true,...
        'LabelSource','foldernames');
    [imdsTrain,imdsTestVal] = splitEachLabel(imds,0.7,"randomized");

    augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain);
end
```

To test the performance of the combined global model during and after training, create test and validation datasets containing images from all classes. Combine the test and validation data from each worker into a single datastore. Then, split this datastore into two datastores that each contain 15% of the overall data - one for validating the network during training and the other for testing the network after training.

```
fileList = [];
labelList = [];

for i = 1:numWorkers
    tmp = imdsTestVal{i};

    fileList = cat(1,fileList,tmp.Files);
    labelList = cat(1,labelList,tmp.Labels);
end

imdsGlobalTestVal = imageDatastore(fileList);
imdsGlobalTestVal.Labels = labelList;

[imdsGlobalTest,imdsGlobalVal] = splitEachLabel(imdsGlobalTestVal,0.5,"randomized");

augimdsGlobalTest = augmentedImageDatastore(inputSize(1:2),imdsGlobalTest);
augimdsGlobalVal = augmentedImageDatastore(inputSize(1:2),imdsGlobalVal);
```

The data is now arranged such that each worker has data from a single class to train on, and the client holds validation and test data from all classes.

Define Network

Determine the number of classes in the data set.

```
classes = categories(imdsGlobalTest.Labels);
numClasses = numel(classes);
```

Define the network architecture.

```
layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','input')
    convolution2dLayer(5,32,'Name','conv1')
    reluLayer('Name','relu1')
    maxPooling2dLayer(2,'Name','maxpool1')
    convolution2dLayer(5,64,'Name','conv2')
    reluLayer('Name','relu2')
    maxPooling2dLayer(2,'Name','maxpool2')
```

```
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
```

Create a dlnetwork object from the layers.

```
dlnet = dlnetwork(layers)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [9×1 nnet.cnn.layer.Layer]
    Connections: [8×2 table]
     Learnables: [6×3 table]
          State: [0×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
    Initialized: 1
```

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 7-0 section of this example, that takes a `dlnetwork` object and a mini-batch of input data with corresponding labels and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Define Federated Averaging Function

Create the function `federatedAveraging`, listed in the the Federated Averaging Function on page 7-0 section of this example, that takes the learnable parameters of the networks on each worker and the normalization factor for each worker, and returns the averaged learnable parameters across all the networks. Use the average learnable parameters to update the global network and the network on each worker.

Define Compute Accuracy Function

Create the function `computeAccuracy`, listed in the Compute Accuracy Function on page 7-0 section of this example, that takes a `dlnetwork` object, a data set inside a `minibatchqueue` object, and the list of classes, and returns the accuracy of the predictions across all observations in the data set.

Specify Training Options

During training, the workers periodically communicate their network learnable parameters to the client, so that the client can update the global model. The training is divided into rounds. At the end of each round of training, the learnable parameters are averaged and the global model is updated. The worker models are then replaced with the new global model, and training continues on the workers.

Train for 300 rounds, with 5 epochs per round. Training for a small numer of epochs per round ensures that the networks on the workers do not diverge too far before they are averaged.

```
numRounds = 300;
numEpochsperRound = 5;
miniBatchSize = 100;
```

Specify the options for SGD optimization. Specify an initial learn rate of 0.001 and momentum 0.

```
learnRate = 0.001;
momentum = 0;
```

Train Model

Create a function handle to the custom mini-batch preprocessing function `preprocessMiniBatch` (defined in the Mini-Batch Preprocessing Function on page 7-0 section of this example).

On each worker, find the total number of training observations processed locally on that worker. Use this number to normalize the learnable parameters on each worker when you find the average learnable parameters after each communicaton round. This helps to balance the average if there is a difference between the amount of data on each worker.

On each worker, create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

- Preprocess the data using the custom mini-batch preprocessing function `preprocessMiniBatch` to convert the labels to one-hot encoded variables.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see GPU Support by Release (Parallel Computing Toolbox).

```
preProcess = @(x,y)preprocessMiniBatch(x,y,classes);

spmd
    sizeOfLocalDataset = augimdsTrain.NumObservations;

    mbq = minibatchqueue(augimdsTrain,...
        'MiniBatchSize',miniBatchSize,...
        'MiniBatchFcn',preProcess,...
        'MiniBatchFormat',{'SSCB',''});
end
```

Create a `minibatchqueue` object that manages the validation data to use during training. Use the same settings as the `minibatchqueue` on each worker.

```
mbqGlobalVal = minibatchqueue(augimdsGlobalVal,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn',preProcess,...
    'MiniBatchFormat',{'SSCB',''});
```

Initialize the training progress plot.

```
figure
lineAccuracyTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Communication rounds")
ylabel("Accuracy (Global)")
grid on
```

Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Initialize the global model. To start, the global model has the same initial parameters as the untrained network on each worker.

```
globalModel = dlnet;
```

Train the model using a custom training loop. For each communication round,

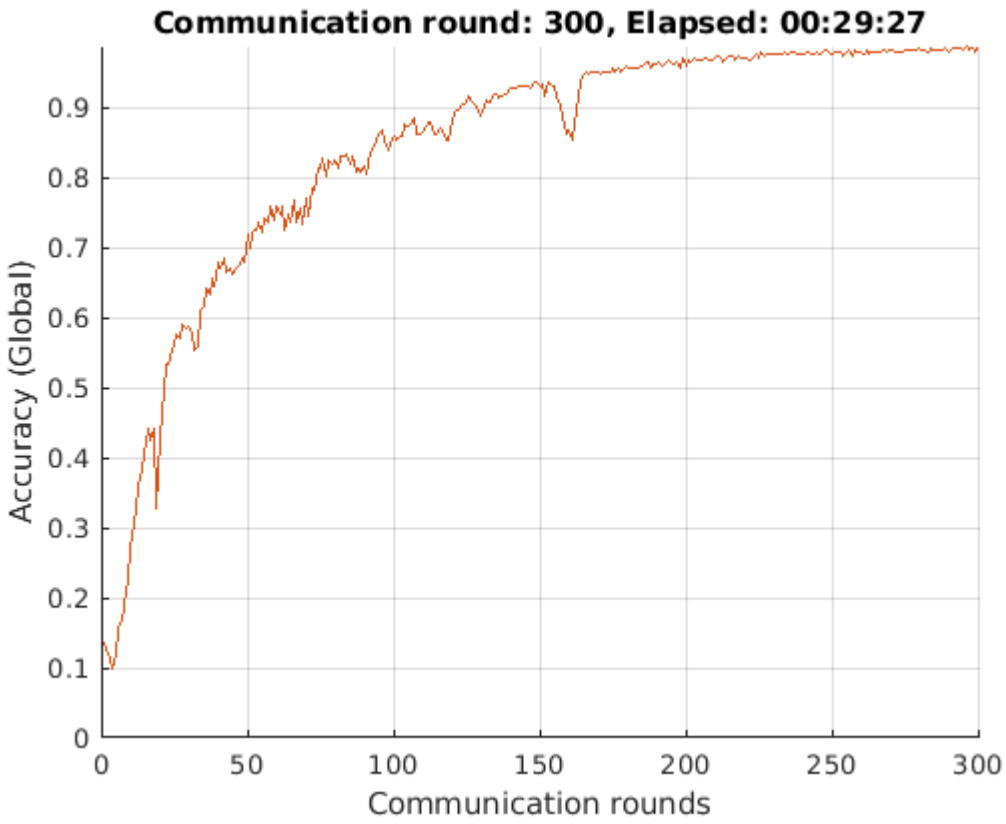
- Update the networks on the workers with the latest global network.
- Train the networks on the workers for five epochs.
- Find the average parameters of all the networks using the `federatedAveraging` function.
- Replace the global network parameters with the average value.
- Calculate the accuracy of the updated global network using the validation data.
- Display the training progress.

For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients and loss using the `dlfeval` and `modelGradients` functions.
- Update the local network parameters using the `sgdmupdate` function.

```
start = tic;
for rounds = 1:numRounds

    spmd
        % Send global updated parameters to each worker.
        dlnet.Learnables.Value = globalModel.Learnables.Value;

        % Loop over epochs.
        for epoch = 1:numEpochsperRound
            % Shuffle data.
            shuffle(mbq);

            % Loop over mini-batches.
            while hasdata(mbq)

                % Read mini-batch of data.
                [dlX,dlT] = next(mbq);

                % Evaluate the model gradients, state, and loss using dlfeval and the
                % modelGradients function and update the network state.
                [gradients,loss] = dlfeval(@modelGradients,dlnet,dlX,dlT);

                % Update the network parameters using the SGDM optimizer.
                [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate,momentum);

            end
        end

        % Collect updated learnable parameters on each worker.
        workerLearnables = dlnet.Learnables.Value;
    end

    % Find normalization factors for each worker based on ratio of data
    % processed on that worker.
    sizeOfAllDatasets = sum([sizeOfLocalDataset{:}]);
    normalizationFactor = [sizeOfLocalDataset{:}]/sizeOfAllDatasets;
```

```
    % Update the global model with new learnable parameters, normalized and
    % averaged across all workers.
    globalModel.Learnables.Value = federatedAveraging(workerLearnables,normalizationFactor);

    % Calculate the accuracy of the global model.
    accuracy = computeAccuracy(globalModel,mbqGlobalVal,classes);

    % Display the training progress of the global model.
    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    addpoints(lineAccuracyTrain,rounds,double(accuracy))
    title("Communication round: " + rounds + ", Elapsed: " + string(D))
    drawnow
end
```



After the final round of training, update the network on each worker with the final average learnable parameters. This is important if you want to continue to use or train the network on the workers.

```
spmd
    dlnet.Learnables.Value = globalModel.Learnables.Value;
end
```

Test Model

Test the classification accuracy of the model by comparing the predictions on the test set with the true labels.

Create a `minibatchqueue` object that manages the test data. Use the same settings as the `minibatchqueue` objects used during training and validation.

```
mbqGlobalTest = minibatchqueue(augimdsGlobalTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn',preProcess,...
    'MiniBatchFormat','SSCB');
```

Use the `computePredictions` function to compute the predicted classes and calculate the accuracy of the predictions across all the test data.

```
accuracy = computeAccuracy(globalModel,mbqGlobalTest,classes)
```

```
accuracy = single
    0.9873
```

After you are done with your computations, you can delete your parallel pool. The `gcp` function returns the current parallel pool object so you can delete the pool.

```
delete(gcp('nocreate'));
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `T` and returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the loss. To compute the gradients automatically, use the `dlgradient` function. To compute predictions of the network during training, use the `forward` function.

```
function [gradients,loss] = modelGradients(dlnet,dlX,T)

    dlYPred = forward(dlnet,dlX);

    loss = crossentropy(dlYPred,T);
    gradients = dlgradient(loss,dlnet.Learnables);

end
```

Compute Accuracy Function

The `computePredictions` function takes a `dlnetwork` object `dlnet`, a `minibatchqueue` object `mbq`, and the list of classes, and returns the accuracy of all the predictions on the data set provided. To compute predictions of the network during validation or after training is finished, use the `predict` function.

```
function accuracy = computeAccuracy(dlnet,mbq,classes)

    correctPredictions = [];

    shuffle(mbq);
    while hasdata(mbq)

        [dlXTest,dlTTest] = next(mbq);

        TTest = onehotdecode(dlTTest,classes,1)';

        dlYPred = predict(dlnet,dlXTest);
        YPred = onehotdecode(dlYPred,classes,1)';
```

```
        correctPredictions = [correctPredictions; YPred == TTest];
    end

    predSum = sum(correctPredictions);
    accuracy = single(predSum./size(correctPredictions,1));

end
```

Mini-Batch Preprocessing Function

The preprocessMiniBatch function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label data from the incoming cell arrays and concatenate into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell,classes)

    % Concatenate.
    X = cat(4,XCell{1:end});

    % Extract label data from cell and concatenate.
    Y = cat(2,YCell{1:end});

    % One-hot encode labels.
    Y = onehotencode(Y,1,'ClassNames',classes);

end
```

Federated Averaging Function

The function `federatedAveraging` function takes the learnable parameters of the networks on each worker and the normalization factor for each worker, and returns the averaged learnable parameters across all the networks. Use the average learnable parameters to update the global network and the network on each worker.

```
function learnables = federatedAveraging(workerLearnables,normalizationFactor)

    numWorkers = size(normalizationFactor,2);

    % Initialize container for averaged learnables with same size as existing
    % learnables. Use learnables of first worker network as an example.
    exampleLearnables = workerLearnables{1};
    learnables = cell(height(exampleLearnables),1);

    for i = 1:height(learnables)
        learnables{i} = zeros(size(exampleLearnables{i}),'like',(exampleLearnables{i}));
    end

    % Add the normalized learnable parameters of all workers to
    % calculate average values.
    for i = 1:numWorkers
        tmp = workerLearnables{i};
```

```
        for values = 1:numel(learnables)
            learnables{values} = learnables{values} + normalizationFactor(i).*tmp{values};
        end
    end

end
```

References

[1] McMahan, H. Brendan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. "Communication-Efficient Learning of Deep Networks from Decentralized Data." Preprint, submitted. February, 2017. https://arxiv.org/abs/1602.05629.

See Also

`dlarray` | `dlnetwork` | `sgdmupdate` | `dlupdate` | `dlfeval` | `dlgradient` | `minibatchqueue`

More About

- “Train Network in Parallel with Custom Training Loop” on page 7-55
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209

Computer Vision Examples

Gesture Recognition using Videos and Deep Learning

This example first shows how to perform gesture recognition using a pretrained SlowFast [1] on page 8-0 video classifier and then shows how to use transfer learning to train a classifier on a custom gesture recognition data set.

Overview

Vision-based human gesture recognition involves predicting a gesture, such as waving hello, sign language gestures, or clapping, using a set of video frames. One of the appealing features of gesture recognition is that they make it possible for humans to communicate with computers and devices without the need for an external input equipment such as a mouse or a remote control. Gesture recognition from videos has many applications, such as control of consumer electronics and mechanical systems, robot learning, and computer games. For example, online prediction of multiple actions for incoming videos from multiple cameras can be important for robot learning. Compared to image classification, human gesture recognition using videos is challenging to model because of the inaccurate ground truth data for video data sets, the variety of gestures that actors in a video can perform, the heavily class imbalanced data sets, and the large amount of data required to train a robust classifier from scratch. Deep learning techniques, such as SlowFast two pathway convolutional networks [1] on page 8-0 , have shown improved performance on smaller data sets using transfer learning with networks pretrained on large video activity recognition data sets.

Note: This example requires the Computer Vision Toolbox™ Model for SlowFast Video Classification. You can install the Computer Vision Toolbox Model for SlowFast Video Classification from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Gesture Recognition Using a Pretrained Video Classifier

Download the pretrained SlowFast video classifier along with a video file on which to perform gesture recognition. The size of the downloaded zip file is around 245 MB.

```
downloadFolder = fullfile(tempdir,"gesture");
if ~isfolder(downloadFolder)
    mkdir(downloadFolder);
end

zipFile = "slowFastPretrained_fourClasses.zip";

if ~isfile(fullfile(downloadFolder,zipFile))
    disp('Downloading the pretrained network...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(downloadFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
    disp("Downloaded.")
end
```

```
Downloading the pretrained network...

Downloaded.
```

Load the pretrained SlowFast video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"slowFastPretrained_fourClasses.mat");
pretrained = load(pretrainedDataFile);
slowFastClassifier = pretrained.data.slowFast;
```

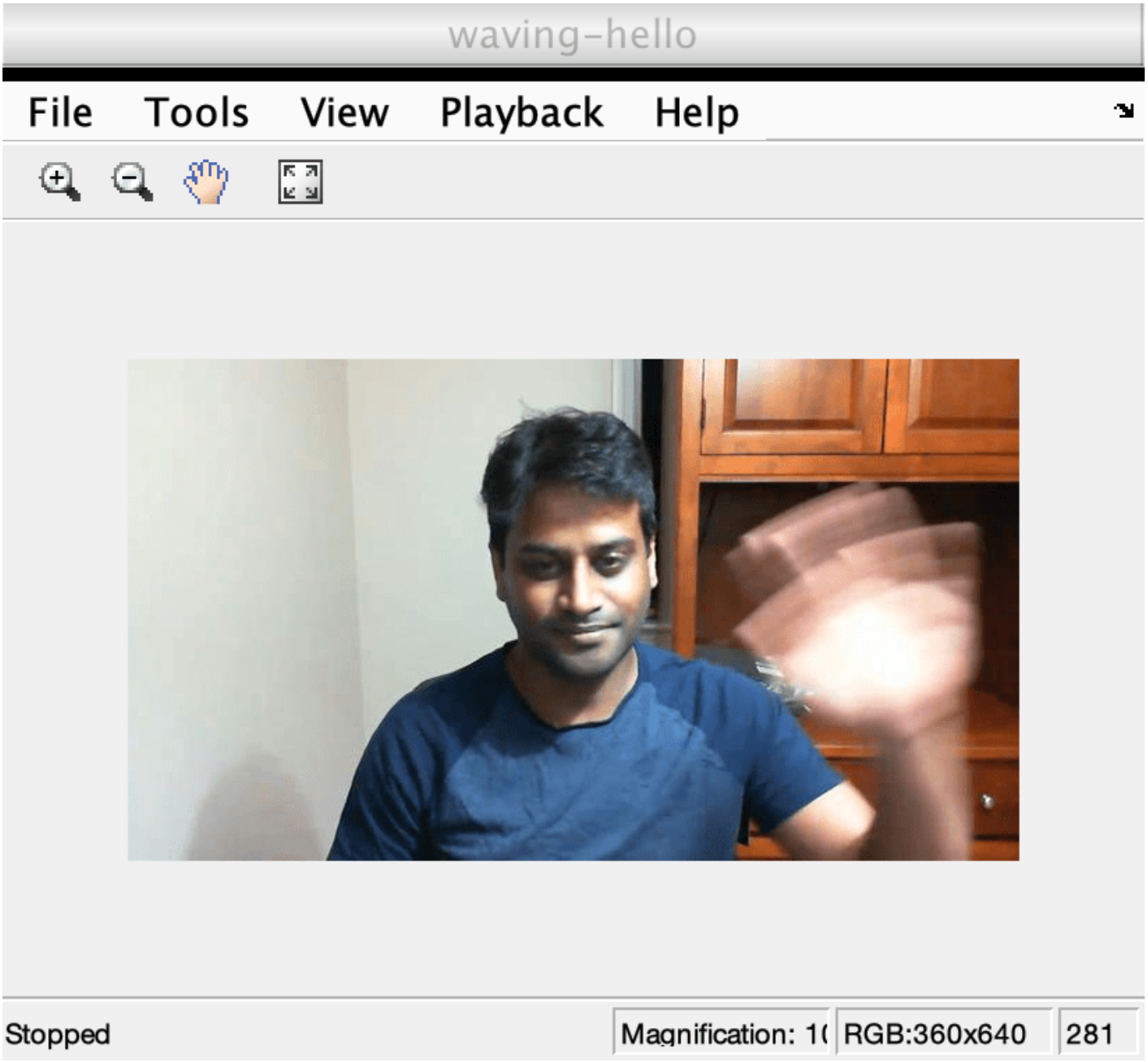
Display the class label names of the pretrained video classifier.

```
classes = slowFastClassifier.Classes
```

```
classes = 4×1 categorical
     clapping
     noAction
     somethingElse
     wavingHello
```

Read and display the video `waving-hello.avi` using `VideoReader` and `vision.VideoPlayer`.

```
videoFilename = fullfile(downloadFolder,"waving-hello.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "waving-hello";

while hasFrame(videoReader)
   frame = readFrame(videoReader);
   step(videoPlayer,frame);
end
release(videoPlayer);
```



Choose 10 randomly selected video sequences to classify the video, to uniformly cover the entirety of the file to find the action class that is predominant in the video.

```
numSequences = 10;
```

Classify the video file using the `classifyVideoFile` function.

```
[gestureLabel,score] = classifyVideoFile(slowFastClassifier,videoFilename,NumSequences=numSequenc
```

```
gestureLabel = categorical
     wavingHello
```

```
score = single
    0.4753
```

The classification can also be applied to a streaming video. To learn how to classify a streaming webcam video, see “Classify Streaming Webcam Video Using SlowFast Video Classifier” (Computer Vision Toolbox).

Train a Video Classifier for Gesture Recognition

This section of the example shows how the video classifier shown above is trained using transfer learning. Set the `doTraining` variable to `false` to use the pretrained video classifier without having to wait for training to complete. Alternatively, if you want to train the video classifier, set the `doTraining` variable to `true`.

```
doTraining = false;
```

Download Ground Truth Training Data

This example trains a SlowFast video classification network using downloadable gesture data set that contains four gestures: `"clapping"`,`"wavingHello"`,`"somethingElse"`, and `"noAction"`. The data set contains videos that are labeled using a Video Labeler and the corresponding ground truth data.

Create directories to store the ground truth training data.

```
groundTruthFolder = fullfile(downloadFolder,"groundTruthFolder");
if ~isfolder(groundTruthFolder)
    mkdir(groundTruthFolder);
end
```

Download the data set and extract the zip archive into the `downloadFolder`.

```
zipFile = 'videoClipsAndSceneLabels.zip';

if ~isfile(fullfile(groundTruthFolder,zipFile))
    disp('Downloading the ground truth training data...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + zipFile;
    zipFile = fullfile(groundTruthFolder,zipFile);
    websave(zipFile,downloadURL);
    unzip(zipFile,groundTruthFolder);
end
```

Extract Training Video Sequences

To train a video classifier, you need a collection of videos and its corresponding collection of scene labels. Use the helper function extractVideoScenes, defined at the end of this example, to extract labeled video scenes from the ground truth data and write them to disk as separate video files. To learn more about extracting training data from videos, see “Extract Training Data for Video Classification” (Computer Vision Toolbox).

```
groundTruthFolder = fullfile(downloadFolder,"groundTruthFolder");
trainingFolder = fullfile(downloadFolder,"videoScenes");

extractVideoScenes(groundTruthFolder,trainingFolder,classes);
```

A total of 40 video scenes are extracted from the downloaded ground truth data.

Load data set

This example uses a datastore to read the videos scenes and labels extracted from the ground truth data.

Specify the number of video frames the datastore should be configured to output for each time data is read from the datastore.

```
numFrames = 16;
```

A value of 16 is used here to balance memory usage and classification time. Common values to consider are 8, 16, 32, 64, or 128. Using more frames helps capture additional temporal information, but requires more memory. Empirical analysis is required to determine the optimal number of frames.

Next, specify the height and width of the frames the datastore should be configured to output. The datastore automatically resizes the raw video frames to the specified size to enable batch processing of multiple video sequences.

```
frameSize = [112,112];
```

A value of [112 112] is used to capture longer temporal relationships in the video scene which help classify gestures with long time durations. Common values for the size are [112 112], [224 224], or [256 256]. Smaller sizes enable the use of more video frames at the cost of memory usage, processing time, and spatial resolution. As with the number of frames, empirical analysis is required to determine the optimal values.

Specify the number of channels as 3, as the videos are RGB.

```
numChannels = 3;
```

Use the helper function, `createFileDatastore`, to configure a `FileDatastore` for loading the data. The helper function is listed at the end of this example.

```
isDataForTraining = true;
dsTrain = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForTraining);
```

Configure SlowFast Video Classifier for Transfer Learning

Create a SlowFast video classifier for transfer learning by using the `slowFastVideoClassifier` function. The `slowFastVideoClassifier` function creates a SlowFast video classifier object that is pretrained on the Kinetics-400 data set [2 on page 8-0].

Specify ResNet-50 as the base network convolution neural network 3D architecture for the SlowFast classifier.

```
baseNetwork = "resnet50-3d";
```

Specify the input size for the SlowFast video classifier.

```
inputSize = [frameSize,numChannels,numFrames];
```

Create a SlowFast video classifier by specifying the classes for the gesture data set and the network input size.

```
slowFast = slowFastVideoClassifier(baseNetwork,string(classes),InputSize=inputSize);
```

Specify a model name for the video classifier.

```
slowFast.ModelName = "Gesture Recognizer Using Deep Learning";
```

Augment and Preprocess Training Data

Data augmentation provides a way to use limited data sets for training. Augmentation on video data must be the same for a collection of frames based on the network input size. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Augment the training video data by using the `augmentVideo` supporting function, defined at the end of this example.

```
dsTrain = transform(dsTrain,@augmentVideo);
```

Preprocess the training video data to resize to the SlowFast video classifier input size, by using the `preprocessVideoClips`, defined at the end of this example. Specify the `InputNormalizationStatistics` property of the video classifier and input size to the preprocessing function as field values in a struct, `preprocessInfo`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
preprocessInfo.Statistics   = slowFast.InputNormalizationStatistics;
preprocessInfo.InputSize    = inputSize;
preprocessInfo.SizingOption = "resize";

dsTrain = transform(dsTrain,@(data)preprocessVideoClips(data,preprocessInfo));
```

Define Model Gradients Function

The `modelGradients` function, listed at the end of this example, takes as input the SlowFast video classifier `slowFast`, a mini-batch of input data `dlRGB`, and a mini-batch of ground truth label data `dlY`. The function returns the training loss value, the gradients of the loss with respect to the learnable parameters of the classifier, and the mini-batch accuracy of the classifier.

The loss is calculated by computing the cross-entropy loss of the predictions from video classifier. The output predictions of the network are probabilities between 0 and 1 for each of the classes.

$$predictions = forward(slowFast, dlRGB);$$

$$loss = crossentropy(predictions)$$

The accuracy of the classifier is calculated by comparing the classifier `predictions` to the ground truth label of the inputs, `dlY`.

Specify Training Options

Train with a mini-batch size of 5 for 600 iterations. Specify the iteration after which to save the model with the best mini-batch loss by using the `SaveBestAfterIteration` parameter.

Specify the cosine-annealing learning rate schedule [3 on page 8-0] parameters:

- A minimum learning rate of 1e-4.
- A maximum learning rate of 1e-3.

- Cosine number of iterations of 200, 300, and 400, after which the learning rate schedule cycle restarts. The option `CosineNumIterations` defines the width of each cosine cycle.

Specify the parameters for SGDM optimization. Initialize the SGDM optimization parameters at the beginning of the training:

- A momentum of 0.9.
- An initial velocity parameter initialized as `[]`.
- An L2 regularization factor of 0.0005.

Specify to dispatch the data in the background using a parallel pool. If `DispatchInBackground` is set to true, open a parallel pool with the specified number of parallel workers, and create a `DispatchInBackgroundDatastore`, provided as part of this example, that dispatches the data in the background to speed up training using asynchronous data loading and preprocessing. By default, this example uses a GPU if one is available. Otherwise, it uses a CPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

```
params.Classes = classes;
params.MiniBatchSize = 5;
params.NumIterations = 600;
params.CosineNumIterations = [100 200 300];
params.SaveBestAfterIteration = 400;
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.Velocity = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = false;
params.Verbose = true;
params.DispatchInBackground = true;
params.NumWorkers = 12;
```

Train Video Classifier

Train the SlowFast video classifier using the video data.

For each epoch:

- Shuffle the data before looping over mini-batches of data.
- Use `minibatchqueue` to loop over the mini-batches. The supporting function `createMiniBatchQueue`, listed at the end of this example, uses the given training datastore to create a `minibatchqueue`.
- Display the loss and accuracy results for each epoch using the supporting function `displayVerboseOutputEveryEpoch`, listed at the end of this example.

For each mini-batch:

- Convert the video data and the labels to `dlarray` objects with the underlying type single.
- To enable processing the time dimension of the the video data using the SlowFast video classifier specify the temporal sequence dimension, `"T"`. Specify the dimension labels `"SSCTB"` (spatial, spatial, channel, temporal, batch) for the video data, and `"CB"` for the label data.

The `minibatchqueue` object uses the supporting function `batchVideo`, listed at the end of this example, to batch the RGB video data.

```
params.ModelFilename = "slowFastPretrained_fourClasses.mat";
if doTraining
    epoch = 1;
    bestLoss = realmax;
    accTrain = [];
    lossTrain = [];

    iteration = 1;
    start = tic;
    trainTime = start;
    shuffled = shuffleTrainDs(dsTrain);

    % Number of outputs is two: One for RGB frames, and one for ground truth labels.
    numOutputs = 2;
    mbq = createMiniBatchQueue(shuffled, numOutputs, params);

    % Use the initializeTrainingProgressPlot and initializeVerboseOutput
    % supporting functions, listed at the end of the example, to initialize
    % the training progress plot and verbose output to display the training
    % loss, training accuracy, and validation accuracy.
    plotters = initializeTrainingProgressPlot(params);
    initializeVerboseOutput(params);

    while iteration <= params.NumIterations

        % Iterate through the data set.
        [dlX1,dlY] = next(mbq);

        % Evaluate the model gradients and loss using dlfeval.
        [gradients,loss,acc,state] = ...
            dlfeval(@modelGradients,slowFast,dlX1,dlY);

        % Accumulate the loss and accuracies.
        lossTrain = [lossTrain, loss];
        accTrain = [accTrain, acc];

        % Update the network state.
        slowFast.State = state;

        % Update the gradients and parameters for the video classifier
        % using the SGDM optimizer.
        [slowFast,params.Velocity,learnRate] = ...
            updateLearnables(slowFast,gradients,params,params.Velocity,iteration);

        if ~hasdata(mbq) || iteration == params.NumIterations
            % Current epoch is complete. Do validation and update progress.
            trainTime = toc(trainTime);

            accTrain = mean(accTrain);
            lossTrain = mean(lossTrain);

            % Update the training progress.
            displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
                accTrain,lossTrain,trainTime);
            updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accTrain);

            % Save the trained video classifier and the parameters, that gave
            % the best training loss so far. Use the saveData supporting function,
```

```
            % listed at the end of this example.
            bestLoss = saveData(slowFast,bestLoss,iteration,lossTrain,params);
        end

        if ~hasdata(mbq) && iteration < params.NumIterations
            % Current epoch is complete. Initialize the training loss, accuracy
            % values, and minibatchqueue for the next epoch.
            accTrain = [];
            lossTrain = [];

            epoch = epoch + 1;
            trainTime = tic;
            shuffled = shuffleTrainDs(dsTrain);
            mbq = createMiniBatchQueue(shuffled, numOutputs, params);
        end

        iteration = iteration + 1;
    end

    % Display a message when training is complete.
    endVerboseOutput(params);

    disp("Model saved to: " + params.ModelFilename);
end
```

Evaluate the Trained Video Classifier

To evaluate the accuracy of the trained SlowFast video classifier, set the `isDataForTraining` variable to false and create a `fileDatastore`. Note that data augmentation is not applied to the evaluation data. Ideally, test and evaluation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
isDataForTraining = false;
dsEval = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForTraining);
dsEval = transform(dsEval,@(data)preprocessVideoClips(data,preprocessInfo));
```

Load the best model saved during training or use the pretrained model.

```
if doTraining
    transferLearned = load(params.ModelFilename);
    slowFastClassifier = transferLearned.data.slowFast;
end
```

Create a `minibatchqueue` object to load batches of the test data.

```
numOutputs = 2;
mbq = createMiniBatchQueue(dsEval,numOutputs,params);
```

For each batch of evaluation data, make predictions using the SlowFast video classifier, and compute the prediction accuracy using a confusion matrix.

```
numClasses = numel(params.Classes);
cmat = sparse(numClasses,numClasses);

while hasdata(mbq)
    [dlVideo,dlY] = next(mbq);

    % Computer the predictions of the trained SlowFast
```

```
    % video classifier.
    dlYPred = predict(slowFastClassifier,dlVideo);
    dlYPred = squeezeIfNeeded(dlYPred,dlY);

    % Aggregate the confusion matrix by using the maximum
    % values of the prediction scores and the ground truth labels.
    [~,YTest] = max(dlY,[],1);
    [~,YPred] = max(dlYPred,[],1);
    cmat = aggregateConfusionMetric(cmat,YTest,YPred);
end
```

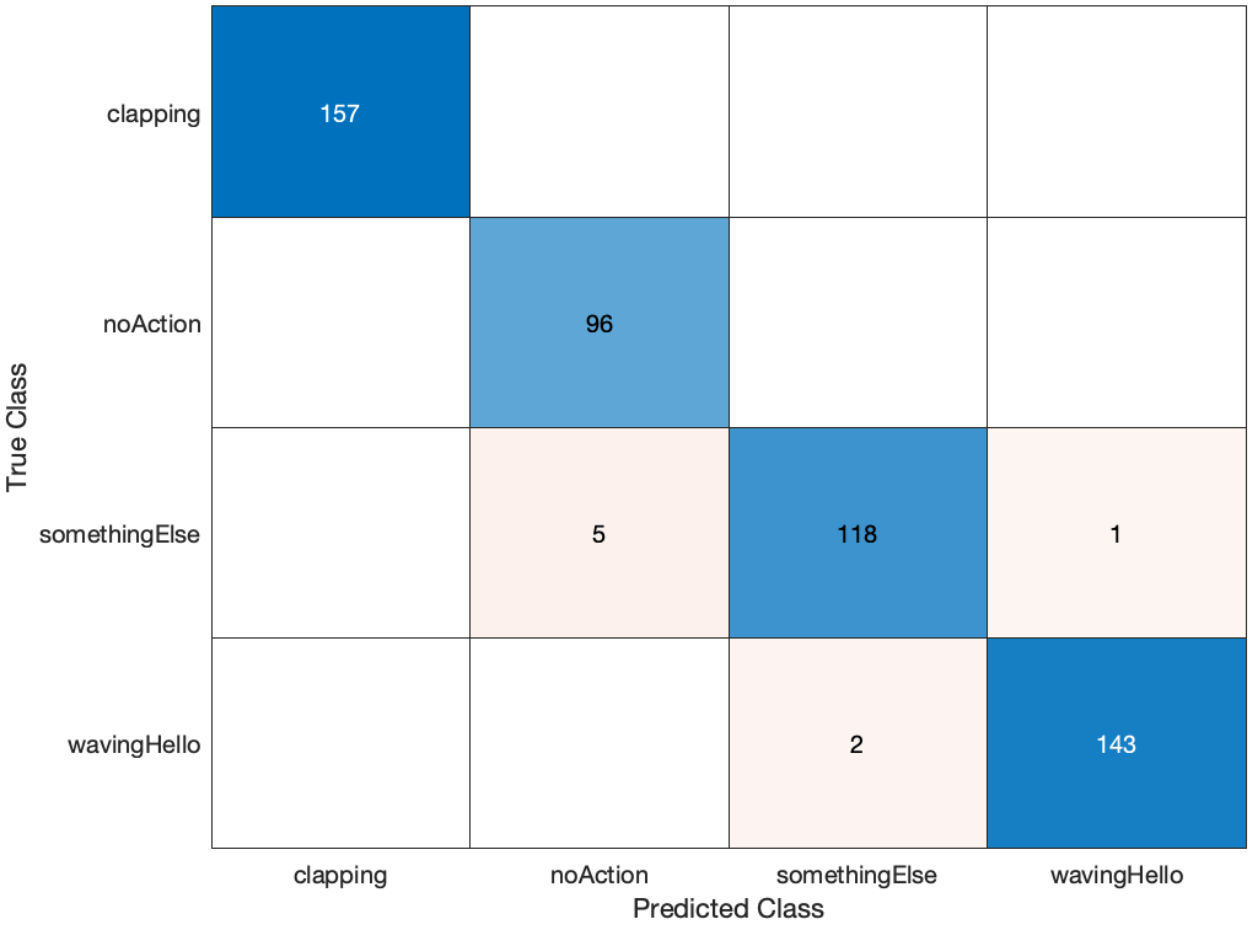
Compute the average clip classification accuracy for the trained SlowFast video classifier.

```
evalClipAccuracy = sum(diag(cmat))./sum(cmat,"all")
```

```
evalClipAccuracy = 0.9847
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



The SlowFast video classifier that is pretrained on the Kinetics-400 data set [2 on page 8-0], provides strong performance for human gesture recognition on transfer learning. The above training was run on 24GB Titan-X GPU for about 60 minutes. When training from scratch on a small gesture recognition video data set, the training time and convergence takes much longer than the pretrained video classifier. Transer learning using the Kinetics-400 pretrained SlowFast video classifier also avoids overfitting the classifier when ran for larger number of epochs on such a small gesture recognition video data set. To learn more about video recognition using deep learning, see “Getting Started with Video Classification Using Deep Learning” (Computer Vision Toolbox).

Supporting Functions

`createFileDatastore`

The `createFileDatastore` function creates a `FileDatastore` object using the given folder name. The `FileDatastore` object reads the data in `'partialfile'` mode, so every read can return partially read frames from videos. This feature helps with reading large video files, if all of the frames do not fit in memory.

```
function datastore = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForT
    readFcn = @(f,u)readVideo(f,u,numFrames,numChannels,classes,isDataForTraining);
    datastore = fileDatastore(trainingFolder,...
        'IncludeSubfolders',true,...
        'FileExtensions','.avi',...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end
```

`shuffleTrainDs`

The `shuffleTrainDs` function shuffles the files present in the training datastore, `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
transformed = isa(shuffled, 'matlab.io.datastore.TransformedDatastore');
if transformed
    files = shuffled.UnderlyingDatastores{1}.Files;
else
    files = shuffled.Files;
end
n = numel(files);
shuffledIndices = randperm(n);
if transformed
    shuffled.UnderlyingDatastores{1}.Files = files(shuffledIndices);
else
    shuffled.Files = files(shuffledIndices);
end

reset(shuffled);

end
```

`readVideo`

The `readVideo` function reads video frames, and the corresponding label values for a given video file. During training, the read function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. During testing, all the frames are sequentially read. The

video frames are resized to the required classifier network input size for training, and for testing and validation.

```
function [data,userdata,done] = readVideo(filename,userdata,numFrames,numChannels,classes,isDataF
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;

        userdata.label = getLabel(filename,classes);

        totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
        totalFrames = min(totalFrames, userdata.reader.NumFrames);
        userdata.totalFrames = totalFrames;
        userdata.datatype = class(read(userdata.reader,1));
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;

    if isDataForTraining
        video = readForTraining(reader,numFrames,totalFrames);
    else
        video = readForEvaluation(reader,userdata.datatype,numChannels,numFrames,totalFrames);
    end

    data = {video, label};

    batchesRead = batchesRead + 1;

    userdata.batchesRead = batchesRead;

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end
    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || isDataForTraining;
end
```

`readForTraining`

The `readForTraining` function reads the video frames for training the video classifier. The function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForTraining(reader,numFrames,totalFrames)
    if numFrames >= totalFrames
        startIdx = 1;
        endIdx = totalFrames;
    else
        startIdx = randperm(totalFrames - numFrames + 1);
        startIdx = startIdx(1);
        endIdx = startIdx + numFrames - 1;
    end
```

```
    video = read(reader,[startIdx,endIdx]);
    if numFrames > totalFrames
        % Add more frames to fill in the network input size.
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

readForEvaluation

The `readForEvaluation` function reads the video frames for evaluating the trained video classifier. The function reads the specific number of frames sequentially as per the network input size. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForEvaluation(reader,datatype,numChannels,numFrames,totalFrames)
    H = reader.Height;
    W = reader.Width;
    toRead = min([numFrames,totalFrames]);
    video = zeros([H,W,numChannels,toRead],datatype);
    frameIndex = 0;
    while hasFrame(reader) && frameIndex < numFrames
        frame = readFrame(reader);
        frameIndex = frameIndex + 1;
        video(:,:,:,frameIndex) = frame;
    end

    if frameIndex < numFrames
        video = video(:,:,:,1:frameIndex);
        additional = ceil(numFrames/frameIndex);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

getLabel

The `getLabel` function obtains the label name from the full path of a filename. The label for a file is the folder in which it exists. For example, for a file path such as `"/path/to/data set/clapping/video_0001.avi"`, the label name is `"clapping"`.

```
function label = getLabel(filename,classes)
    folder = fileparts(string(filename));
    [~,label] = fileparts(folder);
    label = categorical(string(label),string(classes));
end
```

augmentVideo

The `augmentVideo` function augments the video frames for training the video classifier. The function augments a video sequence with the same augmentation technique provided by the `augmentTransform` function.

```
function data = augmentVideo(data)
    numClips = size(data,1);
    for ii = 1:numClips
        video = data{ii,1};
```

```
        % HxWxC
        sz = size(video,[1,2,3]);
        % One augment fcn per clip
        augmentFcn = augmentTransform(sz);
        data{ii,1} = augmentFcn(video);
    end
end
```

augmentTransform

The `augmentTransform` function creates an augmentation method with random left-right flipping and scaling factors.

```
function augmentFcn = augmentTransform(sz)
% Randomly flip and scale the image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');

augmentFcn = @(data)augmentData(data,tform,rout);

    function data = augmentData(data,tform,rout)
        data = imwarp(data,tform,'OutputView',rout);
    end
end
```

preprocessVideoClips

The `preprocessVideoClips` function preprocesses the training video data to resize to the SlowFast video classifier input size. It takes the `InputNormalizationStatistics` and the `InputSize` properties of the video classifier in a struct, `info`. The `InputNormalizationStatistics` property is used to rescale the video frames between 0 and 1, and then normalize the rescaled data using mean and standard deviation. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
function data = preprocessVideoClips(data, info)
    inputSize = info.InputSize(1:2);
    sizingOption = info.SizingOption;
    switch sizingOption
        case "resize"
            sizingFcn = @(x)imresize(x,inputSize);
        case "randomcrop"
            sizingFcn = @(x)cropVideo(x,@randomCropWindow2d,inputSize);
        case "centercrop"
            sizingFcn = @(x)cropVideo(x,@centerCropWindow2d,inputSize);
    end
    numClips = size(data,1);

    minValue  = info.Statistics.Min;
    maxValue  = info.Statistics.Max;
    meanValue = info.Statistics.Mean;
    stdValue  = info.Statistics.StandardDeviation;

    minValue  = reshape(minValue,1,1,3);
    maxValue  = reshape(maxValue,1,1,3);
    meanValue = reshape(meanValue,1,1,3);
```

```
    stdValue  = reshape(stdValue,1,1,3);

    for ii = 1:numClips
        video = data{ii,1};
        resized = sizingFcn(video);

        % Cast the input to single.
        resized = single(resized);

        % Rescale the input between 0 and 1.
        resized = rescale(resized,0,1,InputMin=minValue,InputMax=maxValue);

        % Normalize using mean and standard deviation.
        resized = resized - meanValue;
        resized = resized./stdValue;
        data{ii,1} = resized;
    end

    function outData = cropVideo(data,cropFcn,inputSize)
        imsz = size(data,[1,2]);
        cropWindow = cropFcn(imsz,inputSize);
        numBatches = size(data,4);
        sz = [inputSize, size(data,3),numBatches];
        outData = zeros(sz,'like',data);
        for b = 1:numBatches
            outData(:,:,:,b) = imcrop(data(:,:,:,b),cropWindow);
        end
    end
end
```

createMiniBatchQueue

The `createMiniBatchQueue` function creates a `minibatchqueue` object that provides `miniBatchSize` amount of data from the given datastore. It also creates a `DispatchInBackgroundDatastore` if a parallel pool is open.

```
function mbq = createMiniBatchQueue(datastore, numOutputs, params)
if params.DispatchInBackground && isempty(gcp('nocreate'))
    % Start a parallel pool, if DispatchInBackground is true, to dispatch
    % data in the background using the parallel pool.
    c = parcluster('local');
    c.NumWorkers = params.NumWorkers;
    parpool('local',params.NumWorkers);
end
p = gcp('nocreate');
if ~isempty(p)
    datastore = DispatchInBackgroundDatastore(datastore, p.NumWorkers);
end

inputFormat(1:numOutputs-1) = "SSCTB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MiniBatchSize, ...
    "MiniBatchFcn", @batchVideo, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchVideo

The `batchVideo` function batches the video, and the label data from cell arrays. It uses `onehotencode` function to encode ground truth categorical labels into one-hot arrays. The one-hot encoded array contains a `1` in the position corresponding to the class of the label, and `0` in every other position.

```
function [video,labels] = batchVideo(video,labels)
% Batch dimension: 5
video = cat(5,video{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
labels = onehotencode(labels,1);
end
```

modelGradients

The `modelGradients` function takes as input a mini-batch of RGB data `dlRGB`, and the corresponding target `dlY`, and returns the corresponding loss, the gradients of the loss with respect to the learnable parameters, and the training accuracy. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```
function [gradientsRGB,loss,acc,stateRGB] = modelGradients(slowFast,dlRGB,dlY)
[dlYPredRGB,stateRGB] = forward(slowFast,dlRGB);
dlYPred = squeezeIfNeeded(dlYPredRGB,dlY);

loss = crossentropy(dlYPred,dlY);

gradientsRGB = dlgradient(loss,slowFast.Learnables);

% Calculate the accuracy of the predictions.
[~,YTest] = max(dlY,[],1);
[~,YPred] = max(dlYPred,[],1);

acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));
end
```

squeezeIfNeeded

The `squeezeIfNeeded` function takes as the predicted scores, `dlYPred` and corresponding target `Y`, and returns the predicted scores `dlYPred`, after squeezing the singleton dimensions, if there are any.

```
function dlYPred = squeezeIfNeeded(dlYPred,Y)
if ~isequal(size(Y),size(dlYPred))
    dlYPred = squeeze(dlYPred);
    dlYPred = dlarray(dlYPred,dims(Y));
end
end
```

updateLearnables

The `updateLearnables` function updates the learnable parameters of the SlowFast video classifier with gradients and other parameters using SGDM optimization function `sgdmupdate`.

```
function [slowFast,velocity,learnRate] = updateLearnables(slowFast,gradients,params,velocity,ite
    % Determine the learning rate using the cosine-annealing learning rate schedule.
```

```
    learnRate = cosineAnnealingLearnRate(iteration, params);

    % Apply L2 regularization to the weights.
    learnables = slowFast.Learnables;
    idx = learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w,gradients(idx,:),learnables

    % Update the network parameters using the SGDM optimizer.
    [slowFast, velocity] = sgdmupdate(slowFast,gradients,velocity,learnRate,params.Momentum);
end
```

cosineAnnealingLearnRate

The `cosineAnnealingLearnRate` function computes the learning rate based on the current iteration number, minimum learning rate, maximum learning rate, and number of iterations for annealing [3 on page 8-0].

```
function lr = cosineAnnealingLearnRate(iteration,params)
    if iteration == params.NumIterations
        lr = params.MinLearningRate;
        return;
    end
    cosineNumIter = [0, params.CosineNumIterations];
    csum = cumsum(cosineNumIter);
    block = find(csum >= iteration, 1,'first');
    cosineIter = iteration - csum(block - 1);
    annealingIteration = mod(cosineIter,cosineNumIter(block));
    cosineIteration = cosineNumIter(block);
    minR = params.MinLearningRate;
    maxR = params.MaxLearningRate;
    cosMult = 1 + cos(pi * annealingIteration / cosineIteration);
    lr = minR + ((maxR - minR) *  cosMult / 2);
end
```

aggregateConfusionMetric

The `aggregateConfusionMetric` function incrementally fills a confusion matrix based on the predicted results `YPred` and the expected results `YTest`.

```
function cmat = aggregateConfusionMetric(cmat,YTest,YPred)
YTest = gather(extractdata(YTest));
YPred = gather(extractdata(YPred));
[m,n] = size(cmat);
cmat = cmat + full(sparse(YTest,YPred,1,m,n));
end
```

saveData

The `saveData` function saves the given SlowFast video classifier, loss, and other training parameters to a MAT-file.

```
function bestLoss = saveData(slowFast,bestLoss,iteration,lossTrain,params)
if iteration >= params.SaveBestAfterIteration
    trainingLoss = extractdata(gather(lossTrain));
    if trainingLoss < bestLoss
        bestLoss = trainingLoss;
        slowFast = gatherFromGPUToSave(slowFast);
        data.BestLoss = bestLoss;
```

```
        data.slowFast = slowFast;
        data.Params = params;
        save(params.ModelFilename,'data');
    end
end
end
```

gatherFromGPUToSave

The `gatherFromGPUToSave` function gathers data from the GPU in order to save the model to disk.

```
function slowfast = gatherFromGPUToSave(slowfast)
if ~canUseGPU
    return;
end
slowfast.Learnables = gatherValues(slowfast.Learnables);
slowfast.State = gatherValues(slowfast.State);
    function tbl = gatherValues(tbl)
        for ii = 1:height(tbl)
            tbl.Value{ii} = gather(tbl.Value{ii});
        end
    end
end
```

extractVideoScenes

The `extractVideoScenes` function extracts training video data from a collection of videos and its corresponding collection of scene labels, by using the functions `sceneTimeRanges` and `writeVideoScenes`.

```
function extractVideoScenes(groundTruthFolder,trainingFolder,classes)
% If the video scenes are already extracted, no need to download
% the data set and extract video scenes.
if isfolder(trainingFolder)
    classFolders = fullfile(trainingFolder,string(classes));
    allClassFoldersFound = true;
    for ii = 1:numel(classFolders)
        if ~isfolder(classFolders(ii))
            allClassFoldersFound = false;
            break;
        end
    end
    if allClassFoldersFound
        return;
    end
end
if ~isfolder(groundTruthFolder)
    mkdir(groundTruthFolder);
end
downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/videoClipsAndSceneLabels.zip";

filename = fullfile(groundTruthFolder,"videoClipsAndSceneLabels.zip");
if ~exist(filename,'file')
    disp("Downloading the video clips and the corresponding scene labels to " + groundTruthFolde
    websave(filename,downloadURL);
end
% Unzip the contents to the download folder.
unzip(filename,groundTruthFolder);
```

```
labelDataFiles = dir(fullfile(groundTruthFolder,"*_labelData.mat"));
labelDataFiles = fullfile(groundTruthFolder,{labelDataFiles.name}');
numGtruth = numel(labelDataFiles);
% Load the label data information and create ground truth objects.
gTruth = groundTruth.empty(numGtruth,0);
for ii = 1:numGtruth
    ld = load(labelDataFiles{ii});
    videoFilename = fullfile(groundTruthFolder,ld.videoFilename);
    gds = groundTruthDataSource(videoFilename);
    gTruth(ii) = groundTruth(gds,ld.labelDefs,ld.labelData);
end
% Gather all the scene time ranges and the corresponding scene labels
% using the sceneTimeRanges function.
[timeRanges, sceneLabels] = sceneTimeRanges(gTruth);
% Specify the subfolder names for each duration as the scene label names.
foldernames = sceneLabels;
% Delete the folder if it already exists.
if isfolder(trainingFolder)
    rmdir(trainingFolder,'s');
end
% Video files are written to the folders specified by the folderNames input.
writeVideoScenes(gTruth,timeRanges,trainingFolder,foldernames);
end
```

`initializeTrainingProgressPlot`

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, and the training accuracy.

```
function plotters = initializeTrainingProgressPlot(params)
if params.ProgressPlot
    % Plot the loss, training accuracy, and validation accuracy.
    figure

    % Loss plot
    subplot(2,1,1)
    plotters.LossPlotter = animatedline;
    xlabel("Iteration")
    ylabel("Loss")

    % Accuracy plot
    subplot(2,1,2)
    plotters.TrainAccPlotter = animatedline('Color','b');
    legend('Training Accuracy','Location','northwest');
    xlabel("Iteration")
    ylabel("Accuracy")
else
    plotters = [];
end
end
```

`updateProgressPlot`

The `updateProgressPlot` function updates the progress plot with loss and accuracy information during training.

```
function updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accuracyTrain)
if params.ProgressPlot
```

```
    % Update the training progress.
    D = duration(0,0,toc(start),"Format","hh:mm:ss");
    title(plotters.LossPlotter.Parent,"Epoch: " + epoch + ", Elapsed: " + string(D));
    addpoints(plotters.LossPlotter,iteration,double(gather(extractdata(lossTrain))));
    addpoints(plotters.TrainAccPlotter,iteration,accuracyTrain);
    drawnow
end
end
```

initializeVerboseOutput

The `initializeVerboseOutput` function displays the column headings for the table of training values, which shows the epoch, mini-batch accuracy, and other training values.

```
function initializeVerboseOutput(params)
if params.Verbose
    disp(" ")
    if canUseGPU
        disp("Training on GPU.")
    else
        disp("Training on CPU.")
    end
    p = gcp('nocreate');
    if ~isempty(p)
        disp("Training on parallel cluster '" + p.Cluster.Profile + "'. ")
    end
    disp("NumIterations:" + string(params.NumIterations));
    disp("MiniBatchSize:" + string(params.MiniBatchSize));
    disp("Classes:" + join(string(params.Classes),","));
    disp("|=========================================================================================
    disp("| Epoch | Iteration | Time Elapsed | Mini-Batch | Mini-Batch |  Base Learning  | Train
    disp("|       |           |  (hh:mm:ss)  |  Accuracy  |    Loss    |      Rate       | (hh:m
    disp("|=========================================================================================
end
end
```

displayVerboseOutputEveryEpoch

The `displayVerboseOutputEveryEpoch` function displays the verbose output of the training values, such as the epoch, mini-batch accuracy, and mini-batch loss.

```
function displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
                accTrain,lossTrain,trainTime)
    if params.Verbose
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        trainTime = duration(0,0,trainTime,'Format','hh:mm:ss');

        lossTrain = gather(extractdata(lossTrain));
        lossTrain = compose('%.4f',lossTrain);

        accTrain = composePadAccuracy(accTrain);

        learnRate = compose('%.13f',learnRate);

        disp("| " + ...
            pad(string(epoch),5,'both') + " | " + ...
            pad(string(iteration),9,'both') + " | " + ...
```

```
            pad(string(D),12,'both') + " | " + ...
            pad(string(accTrain),10,'both') + " | " + ...
            pad(string(lossTrain),10,'both') + " | " + ...
            pad(string(learnRate),13,'both') + " | " + ...
            pad(string(trainTime),10,'both') + " |")
    end

    function acc = composePadAccuracy(acc)
        acc = compose('%.2f',acc*100) + "%";
        acc = pad(string(acc),6,'left');
    end

end
```

`endVerboseOutput`

The `endVerboseOutput` function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp("|=======================================================================================
end
end
```

References

[1] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast Networks for Video Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2019.

[2] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, Andrew Zisserman. "The Kinetics Human Action Video data set." *arXiv preprint arXiv:1705.06950*, 2017.

[3] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *International Conferencee on Learning Representations 2017*. Toulon, France: ICLR, 2017.

Code Generation for Object Detection by Using Single Shot Multibox Detector

This example shows how to generate CUDA® code for an SSD network (ssdObjectDetector object) and take advantage of the NVIDIA® cuDNN and TensorRT libraries. An SSD network is based on a feed-forward convolutional neural network that detect multiple objects within the image in a single shot. SSD network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

This example generates code for the network trained in the *Object Detection Using SSD Deep Learning* example from Computer Vision Toolbox™. For more information, see “Object Detection Using SSD Deep Learning” (Computer Vision Toolbox). The *Object Detection Using SSD Deep Learning* example uses ResNet-50 for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAGNetwork

```
net = getSSDNW();
```

The DAG network contains 180 layers including convolution, ReLU, and batch normalization layers, anchor box, SSD merge, focal loss, and other layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(net);
```

The `ssdObj_detect` Entry-Point Function

The ssdObj_detect.m entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `ssdResNet50VehicleExample_20a.mat` file. The function loads the network object from the `ssdResNet50VehicleExample_20a.mat` file into a persistent variable *ssdObj* and reuses the persistent object on subsequent detection calls.

```
type('ssdObj_detect.m')
```

```
function outImg = ssdObj_detect(in)

%   Copyright 2019-2021 The MathWorks, Inc.

persistent ssdObj;

if isempty(ssdObj)
    ssdObj = coder.loadDeepLearningNetwork('ssdResNet50VehicleExample_20a.mat');
end

% Pass in input
[bboxes,~,labels] = ssdObj.detect(in,'Threshold',0.7);

% Convert categorical labels to cell array of charactor vectors for
% execution
labels = cellstr(labels);

% Annotate detections in the image.
if ~isempty(labels)
    outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
else
    outImg = in;
end
```

Run MEX Code Generation

To generate CUDA code for the ssdObj_detect.m entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [300,300,3]. This value corresponds to the input layer size of SSD Network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg ssdObj_detect -args {ones(300,300,3,'uint8')} -report
```

```
Code generation successful: View report
```

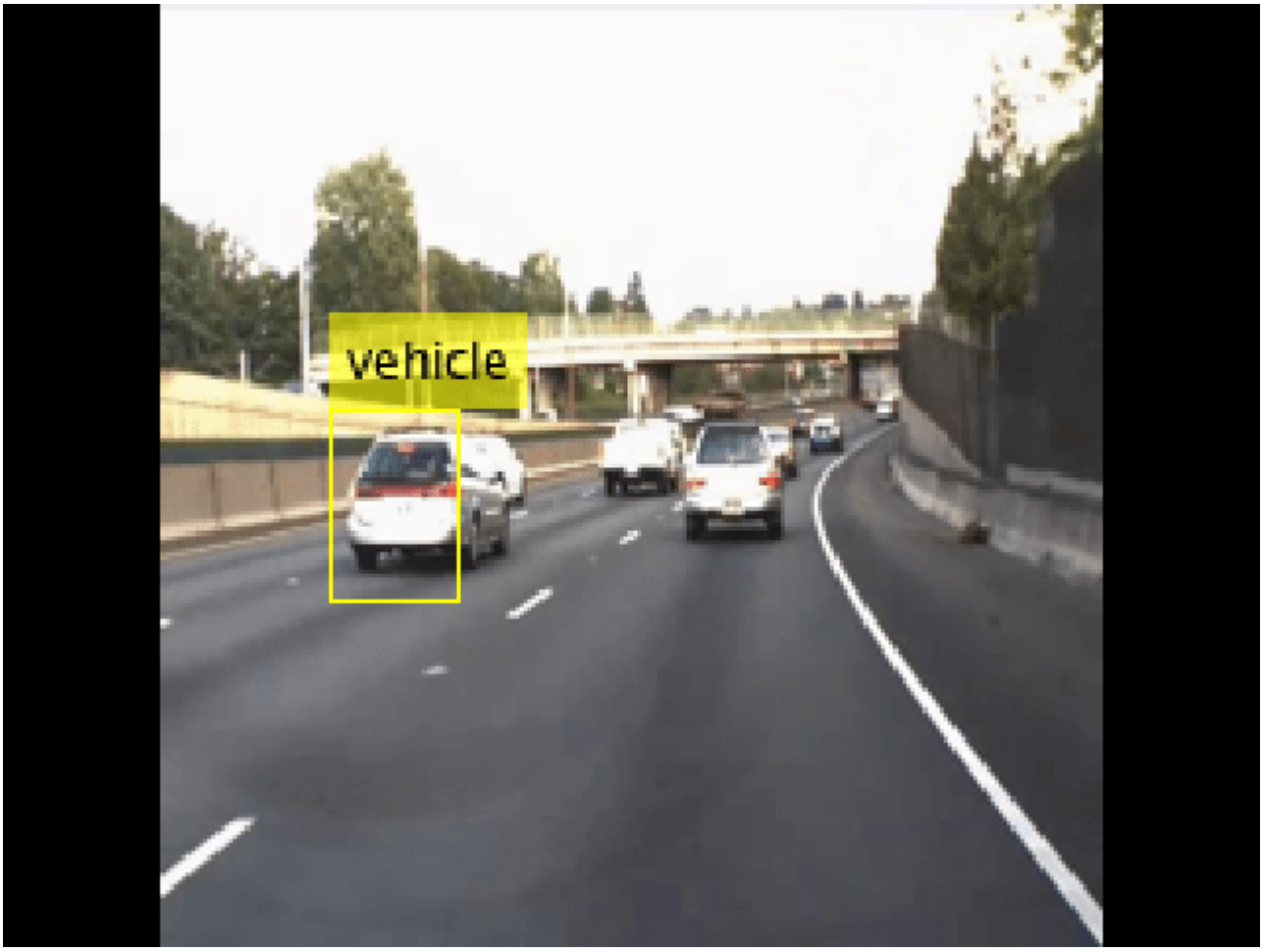
Run Generated MEX

Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[300,300]);
    out = ssdObj_detect_mex(in);
    step(depVideoPlayer, out);
    % Exit the loop if the video player figure window is closed
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer);
end
```



References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.

Point Cloud Classification Using PointNet Deep Learning

This example shows how to train a PointNet network for point cloud classification.

Point cloud data is acquired by a variety of sensors, such as lidar, radar, and depth cameras. These sensors capture 3-D position information about objects in a scene, which is useful for many applications in autonomous driving and augmented reality. For example, discriminating vehicles from pedestrians is critical for planning the path of an autonomous vehicle. However, training robust classifiers with point cloud data is challenging because of the sparsity of data per object, object occlusions, and sensor noise. Deep learning techniques have been shown to address many of these challenges by learning robust feature representations directly from point cloud data. One of the seminal deep learning techniques for point cloud classification is PointNet [1 on page 8-0].

This example trains a PointNet classifier on the Sydney Urban Objects data set created by the University of Sydney [2 on page 8-0]. This data set provides a collection of point cloud data acquired from an urban environment using a lidar sensor. The data set has 100 labeled objects from 14 different categories, such as car, pedestrian, and bus.

Load data set

Download and extract the Sydney Urban Objects data set to a temporary directory.

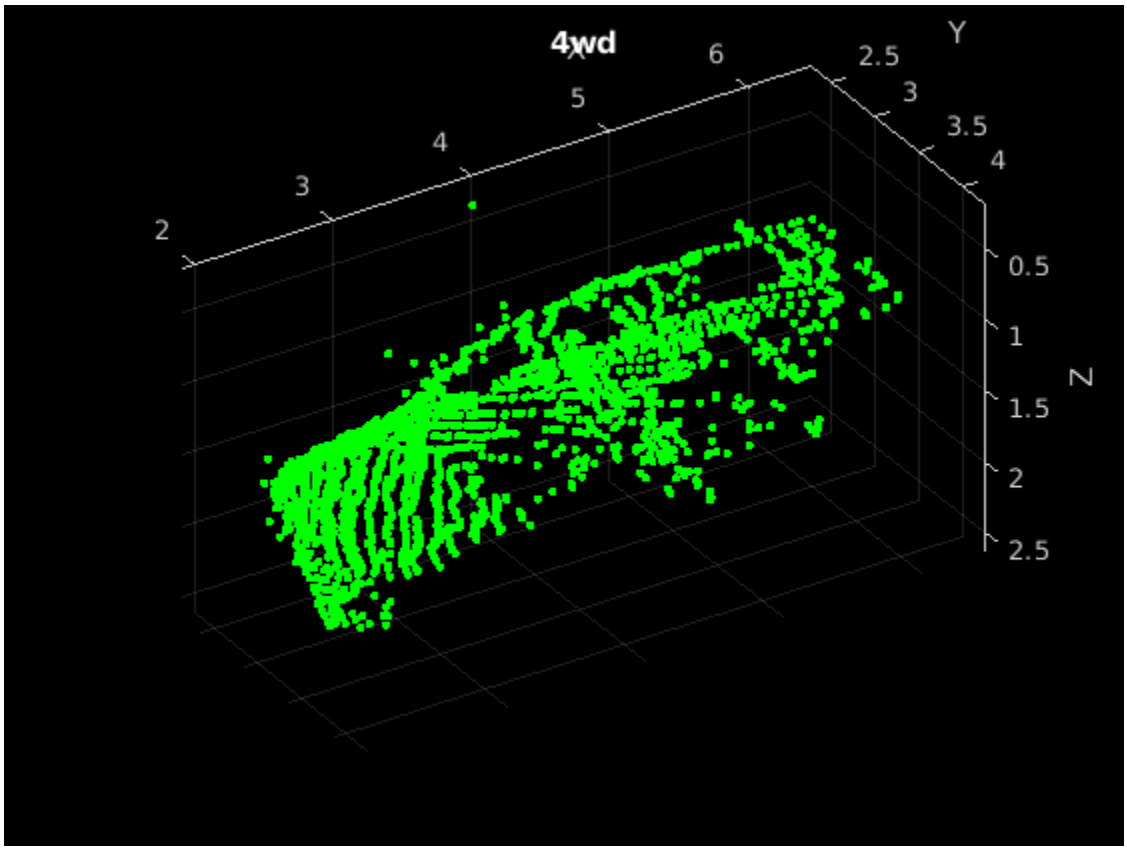
```
downloadDirectory = tempdir;
datapath = downloadSydneyUrbanObjects(downloadDirectory);
```

Load the downloaded training and validation data set using the `loadSydneyUrbanObjectsData` helper function listed at the end of this example. Use the first three data folds for training and the fourth for validation.

```
foldsTrain = 1:3;
foldsVal = 4;
dsTrain = loadSydneyUrbanObjectsData(datapath,foldsTrain);
dsVal = loadSydneyUrbanObjectsData(datapath,foldsVal);
```

Read one of the training samples and visualize it using `pcshow`.

```
data = read(dsTrain);
ptCloud = data{1,1};
label = data{1,2};

figure
pcshow(ptCloud.Location,[0 1 0],"MarkerSize",40,"VerticalAxisDir","down")
xlabel("X")
ylabel("Y")
zlabel("Z")
title(label)
```

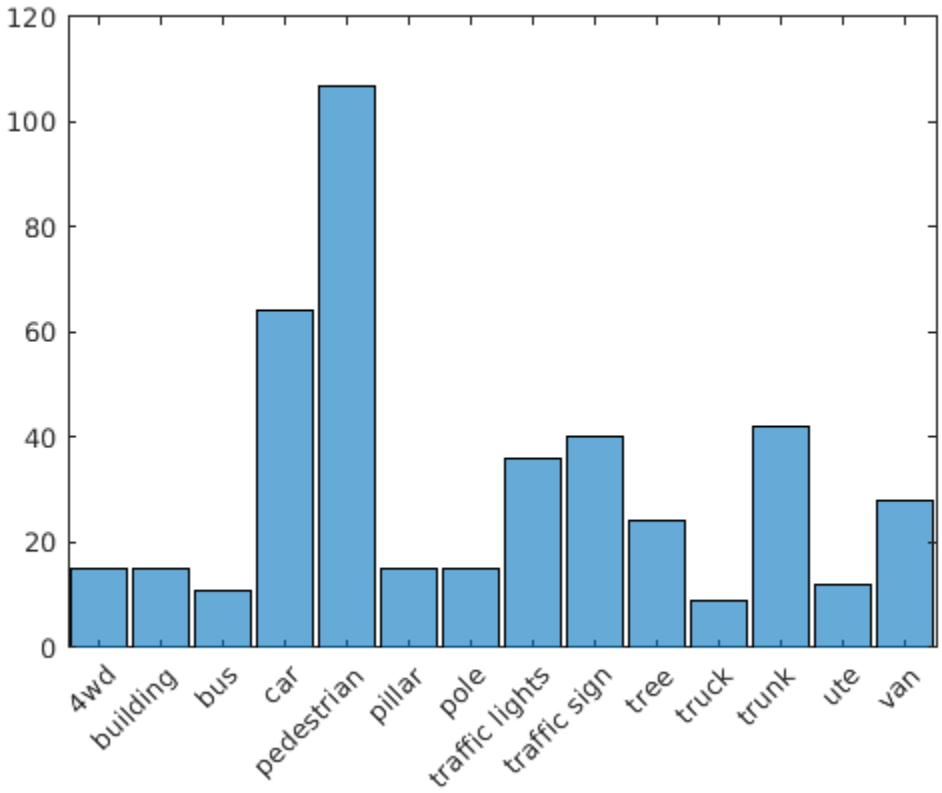


Read the labels and count the number of points assigned to each label to better understand the distribution of labels within the data set.

```
dsLabelCounts = transform(dsTrain,@(data){data{2} data{1}.Count});
labelCounts = readall(dsLabelCounts);
labels = vertcat(labelCounts{:,1});
counts = vertcat(labelCounts{:,2});
```

Next, use a histogram to visualize the class distribution.

```
figure
histogram(labels)
```



The label histogram shows that the data set is imbalanced and biased towards cars and pedestrians, which can prevent the training of a robust classifier. You can address class imbalance by oversampling the infrequent classes. For the Sydney Urban Objects data set, duplicating files corresponding to the infrequent classes is a simple method to address the class imbalance.

Group the files by label, count the number of observations per class, and use the `randReplicateFiles` helper function, listed at the end of this example, to randomly oversample the files to the desired number of observations per class.

```
rng(0)
[G,classes] = findgroups(labels);
numObservations = splitapply(@numel,labels,G);
desiredNumObservationsPerClass = max(numObservations);
files = splitapply(@(x){randReplicateFiles(x,desiredNumObservationsPerClass)},dsTrain.Files,G);
files = vertcat(files{:});
dsTrain.Files = files;
```

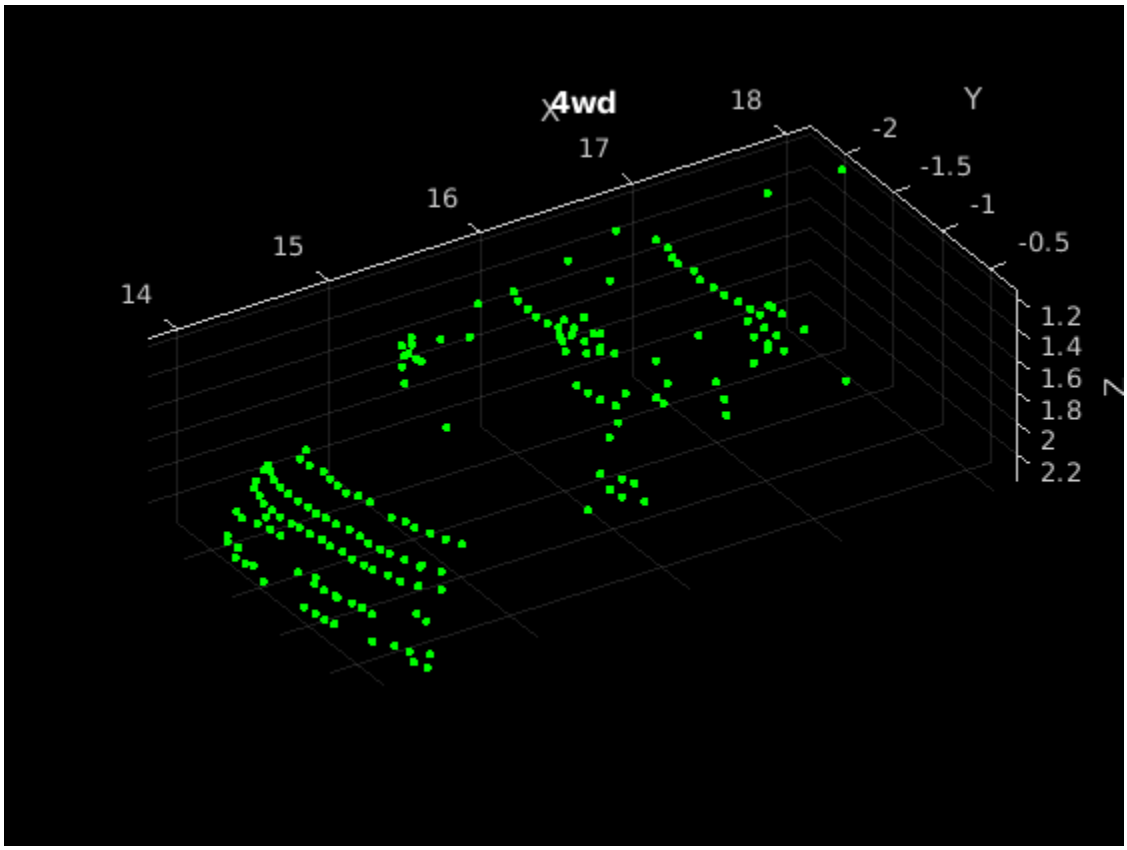
Data Augmentation

Duplicating the files to address class imbalance increases the likelihood of overfitting the network because much of the training data is identical. To offset this effect, apply data augmentation to the training data using the `transform` and `augmentPointCloud` helper function, which randomly rotates the point cloud, randomly removes points, and randomly jitters points with Gaussian noise.

```
dsTrain = transform(dsTrain,@augmentPointCloud);
```

Preview one of the augmented training samples.

```
data = preview(dsTrain);
ptCloud = data{1,1};
label = data{1,2};

figure
pcshow(ptCloud.Location,[0 1 0],"MarkerSize",40,"VerticalAxisDir","down")
xlabel("X")
ylabel("Y")
zlabel("Z")
title(label)
```



Note that because the data for measuring the performance of the trained network must be representative of the original data set, data augmentation is not applied to validation or test data.

Data Preprocessing

Two preprocessing steps are required to prepare the point cloud data for training and prediction.

First, to enable batch processing during training, select a fixed number of points from each point cloud. The optimal number of points depends on the data set and the number of points required to accurately capture the shape of the object. To help select the appropriate number of points, compute the minimum, maximum, and mean number of points per class.

```
minPointCount = splitapply(@min,counts,G);
maxPointCount = splitapply(@max,counts,G);
meanPointCount = splitapply(@(x)round(mean(x)),counts,G);

stats = table(classes,numObservations,minPointCount,maxPointCount,meanPointCount)
```

```
stats=14×5 table
       classes        numObservations    minPointCount    maxPointCount    meanPointCount
    ______________    _______________    _____________    _____________    ______________

    4wd                        15              140             1955             751
    building                   15              193             8455            2708
    bus                        11              126            11767            2190
    car                        64               52             2377             528
    pedestrian                107               20              297             110
    pillar                     15               80              751             357
    pole                       15               13              253              90
    traffic lights             36               38              352             161
    traffic sign               40               18              736             126
    tree                       24               53             2953             470
    truck                       9              445             3013            1376
    trunk                      42               32              766             241
    ute                        12               90             1380             580
    van                        28               91             5809            1125
```

Because of the large amount of intra-class and inter-class variability in the number of points per class, choosing a value that fits all classes is difficult. One heuristic is to choose enough points to adequately capture the shape of the objects while not increasing the computational cost by processing too many points. A value of 1024 provides a good tradeoff between these two facets. You can also select the optimal number of points based on empirical analysis. However, that is beyond the scope of this example. Use the `transform` function to select 1024 points in the training and validation sets.

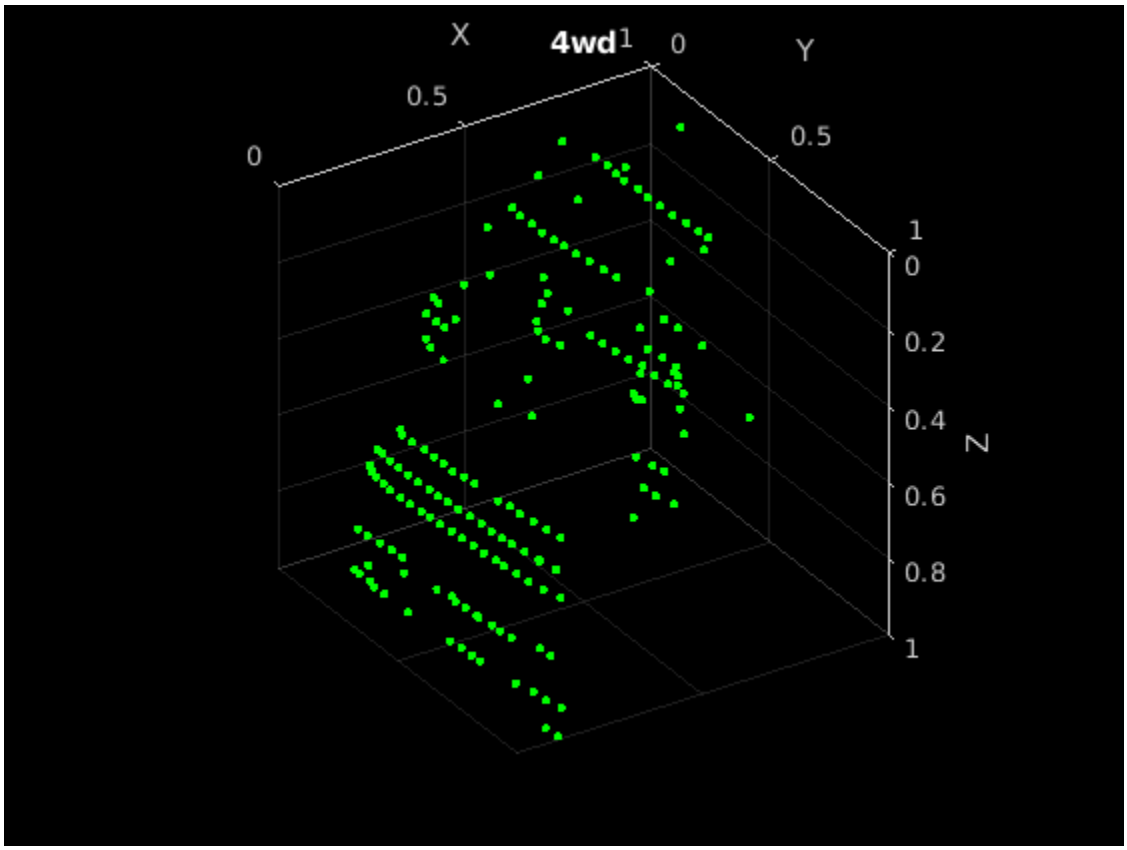
```
numPoints = 1024;
dsTrain = transform(dsTrain,@(data)selectPoints(data,numPoints));
dsVal = transform(dsVal,@(data)selectPoints(data,numPoints));
```

The last preprocessing step is to normalize the point cloud data between 0 and 1 to account for large differences in the range of data values. For example, objects closer to the lidar sensor have smaller values compared to objects that are further away. These differences can hinder the convergence of the network during training. Use `transform` to normalize the point cloud data in the training and validation sets.

```
dsTrain = transform(dsTrain,@preprocessPointCloud);
dsVal = transform(dsVal,@preprocessPointCloud);
```

Preview the augmented and preprocessed training data.

```
data = preview(dsTrain);
figure
pcshow(data{1,1},[0 1 0],"MarkerSize",40,"VerticalAxisDir","down");
xlabel("X")
ylabel("Y")
zlabel("Z")
title(data{1,2})
```



Define PointNet Model

The PointNet classification model consists of two components. The first component is a point cloud encoder that learns to encode sparse point cloud data into a dense feature vector. The second component is a classifier that predicts the categorical class of each encoded point cloud.

The PointNet encoder model is further composed of four models followed by a max operation.

1. Input transform model
2. Shared MLP model
3. Feature transform model
4. Shared MLP model

The shared MLP model is implemented using a series of convolution, batch normalization, and ReLU operations. The convolution operation is configured such that the weights are shared across the input point cloud. The transform model is composed of a shared MLP and a learnable transform matrix that is applied to each point cloud. The shared MLP and the max operation make the PointNet encoder invariant to the order in which the points are processed, while the transform model provides invariance to orientation changes.

Define PointNet Encoder Model Parameters

The shared MLP and transform models are parameterized by the number of input channels and the hidden channel sizes. The values chosen in this example are selected by tuning these hyperparameters on the Sydney Urban Objects data set. Note that if you want to apply PointNet to a different data set, you must perform additional hyperparameter tuning.

Set the input transform model input channel size to three and the hidden channel sizes to 64, 128, and 256 and use the `initializeTransform` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 3;
hiddenChannelSize1 = [64,128];
hiddenChannelSize2 = 256;
[parameters.InputTransform, state.InputTransform] = initializeTransform(inputChannelSize,hiddenCh
```

Set the first shared MLP model input channel size to three and the hidden channel size to 64 and use the `initializeSharedMLP` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 3;
hiddenChannelSize = [64 64];
[parameters.SharedMLP1,state.SharedMLP1] = initializeSharedMLP(inputChannelSize,hiddenChannelSize
```

Set the feature transformation model input channel size to 64 and hidden channel sizes to 64, 128, and 256 and use the `initializeTransform` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize1 = [64,128];
hiddenChannelSize2 = 256;
[parameters.FeatureTransform, state.FeatureTransform] = initializeTransform(inputChannelSize,hidd
```

Set the second shared MLP model input channel size to 64 and the hidden channel size to 64 and use the `initializeSharedMLP` function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize = 64;
[parameters.SharedMLP2,state.SharedMLP2] = initializeSharedMLP(inputChannelSize,hiddenChannelSize
```

Define PointNet Classifier Model Parameters

The PointNet classifier model consists of a shared MLP, a fully connected operation, and a softmax activation. Set the classifier model input size to 64 and the hidden channel size to 512 and 256 and use the `initalizeClassifier` helper function, listed at the end of this example, to initialize the model parameters.

```
inputChannelSize = 64;
hiddenChannelSize = [512,256];
numClasses = numel(classes);
[parameters.ClassificationMLP, state.ClassificationMLP] = initializeClassificationMLP(inputChanne
```

Define PointNet Function

Create the function `pointnetClassifier`, listed in the Model Function section at the end of the example, to compute the outputs of the PointNet model. The function model takes as input the point cloud data, the learnable model parameters, the model state, and a flag that specifies whether the model returns outputs for training or prediction. The network returns the predictions for classifying the input point cloud.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function section of the example, that takes as input the model parameters, the model state, and a mini-batch of input data, and

returns the gradients of the loss with respect to the learnable parameters in the models and the corresponding loss.

Specify Training Options

Train for 10 epochs and load data in batches of 128. Set the initial learning rate to 0.002 and the L2 regularization factor to 0.01.

```
numEpochs = 10;
learnRate = 0.002;
miniBatchSize = 128;
l2Regularization = 0.01;
learnRateDropPeriod = 15;
learnRateDropFactor = 0.5;
```

Initialize the options for Adam optimization.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Train PointNet

Train the model using a custom training loop.

Shuffle the data at the beginning of training.

For each iteration:

- Read a batch of data.
- Evaluate the model gradients.
- Apply L2 weight regularization.
- Use `adamupdate` to update the model parameters.
- Update the training progress plot.

At the end of each epoch, evaluate the model against the validation data set and collect confusion metrics to measure classification accuracy as training progresses.

After completing `learnRateDropPeriod` epochs, reduce the learning rate by a factor of `learnRateDropFactor`.

Initialize the moving average of the parameter gradients and the element-wise squares of the gradients used by the Adam optimizer.

```
avgGradients = [];
avgSquaredGradients = [];
```

Train the model if `doTraining` is true. Otherwise, load a pretrained network.

Note that training was verified on an NVIDIA Titan X with 12 GB of GPU memory. If your GPU has less memory, you may run out of memory during training. If this happens, lower the `miniBatchSize`. Training this network takes about 5 minutes. Depending on your GPU hardware, it can take longer.

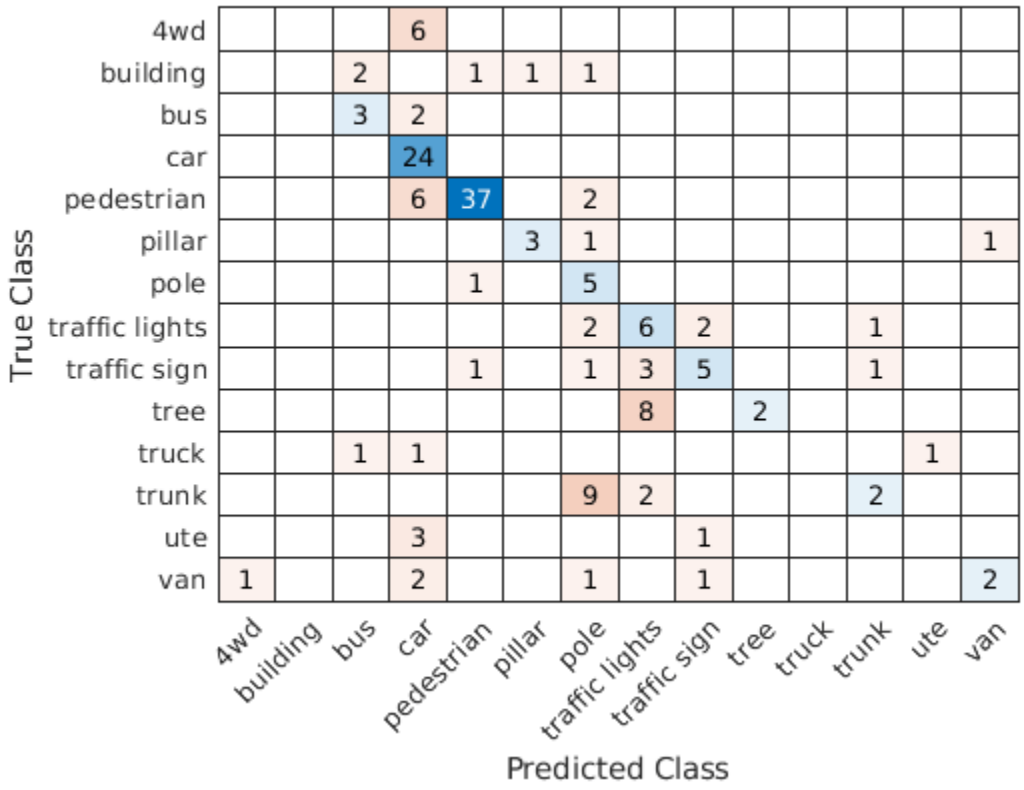
```
doTraining = false;

if doTraining
```

```
% Create a minibatchqueue to batch data from training and validation
% datastores. Use the batchData function, listed at the end of the
% example, to batch the point cloud data and one-hot encode the label
% data.
numOutputsFromDSRead = 2;
mbqTrain = minibatchqueue(dsTrain,numOutputsFromDSRead,...
    "MiniBatchSize", miniBatchSize,...
    "MiniBatchFcn",@batchData,...
    "MiniBatchFormat",["SCSB" "BC"]);

mbqVal = minibatchqueue(dsVal,numOutputsFromDSRead,...
    "MiniBatchSize", miniBatchSize,...
    "MiniBatchFcn",@batchData,...
    "MiniBatchFormat",["SCSB" "BC"]);

% Use the configureTrainingProgressPlot function, listed at the end of the
% example, to initialize the training progress plot to display the training
% loss, training accuracy, and validation accuracy.
[lossPlotter, trainAccPlotter,valAccPlotter] = initializeTrainingProgressPlot;

numClasses = numel(classes);
iteration = 0;
start = tic;
for epoch = 1:numEpochs

    % Shuffle data every epoch.
    shuffle(mbqTrain);

    % Iterate through data set.
    while hasdata(mbqTrain)
        iteration = iteration + 1;

        % Read next batch of training data.
        [XTrain, YTrain] = next(mbqTrain);

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients, loss, state, acc] = dlfeval(@modelGradients,XTrain,YTrain,parameters,sta

        % L2 regularization.
        gradients = dlupdate(@(g,p) g + l2Regularization*p,gradients,parameters);

        % Update the network parameters using the Adam optimizer.
        [parameters, avgGradients, avgSquaredGradients] = adamupdate(parameters, gradients,
            avgGradients, avgSquaredGradients, iteration,...
            learnRate,gradientDecayFactor, squaredGradientDecayFactor);

        % Update the training progress.
        D = duration(0,0,toc(start),"Format","hh:mm:ss");
        title(lossPlotter.Parent,"Epoch: " + epoch + ", Elapsed: " + string(D))
        addpoints(lossPlotter,iteration,double(gather(extractdata(loss))))
        addpoints(trainAccPlotter,iteration,acc);
        drawnow
    end

    % Evaluate the model on validation data.
    cmat = sparse(numClasses,numClasses);
    while hasdata(mbqVal)
```

```
            % Read next batch of validation data.
            [XVal, YVal] = next(mbqVal);

            % Compute label predictions.
            isTraining = false;
            YPred = pointnetClassifier(XVal,parameters,state,isTraining);

            % Choose prediction with highest score as the class label for
            % XTest.
            [~,YValLabel] = max(YVal,[],1);
            [~,YPredLabel] = max(YPred,[],1);

            % Collect confusion metrics.
            cmat = aggreateConfusionMetric(cmat,YValLabel,YPredLabel);
        end

        % Update training progress plot with average classification accuracy.
        acc = sum(diag(cmat))./sum(cmat,"all");
        addpoints(valAccPlotter,iteration,acc);

        % Upate the learning rate.
        if mod(epoch,learnRateDropPeriod) == 0
            learnRate = learnRate * learnRateDropFactor;
        end

        % Reset training and validation data queues.
        reset(mbqTrain);
        reset(mbqVal);
    end

else
    % Download pretrained model parameters, model state, and validation
    % results.
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/pointnetSydneyUrbanObjec
    pretrainedResults = downloadPretrainedPointNet(pretrainedURL);

    parameters = pretrainedResults.parameters;
    state = pretrainedResults.state;
    cmat = pretrainedResults.cmat;

    % Move model parameters to the GPU if possible and convert to a dlarray.
    parameters = prepareForPrediction(parameters,@(x)dlarray(toDevice(x,canUseGPU)));

    % Move model state to the GPU if possible.
    state = prepareForPrediction(state,@(x)toDevice(x,canUseGPU));
end
```

Display the validation confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



Compute the mean training and validation accuracy.

```
acc = sum(diag(cmat))./sum(cmat,"all")
```

```
acc = 0.5742
```

Due to the limited number of training samples in the Sydney Urban Objects data set, increasing the validation accuracy beyond 60% is challenging. The model easily overfits the training data in the absence of the augmentation defined in the `augmentPointCloudData` helper function. To improve the robustness of the PointNet classifier, additional training is required.

Classify Point Cloud Data Using PointNet

Load point cloud data with `pcread`, preprocess the point cloud using the same function used during training, and convert the result to a `dlarray`.

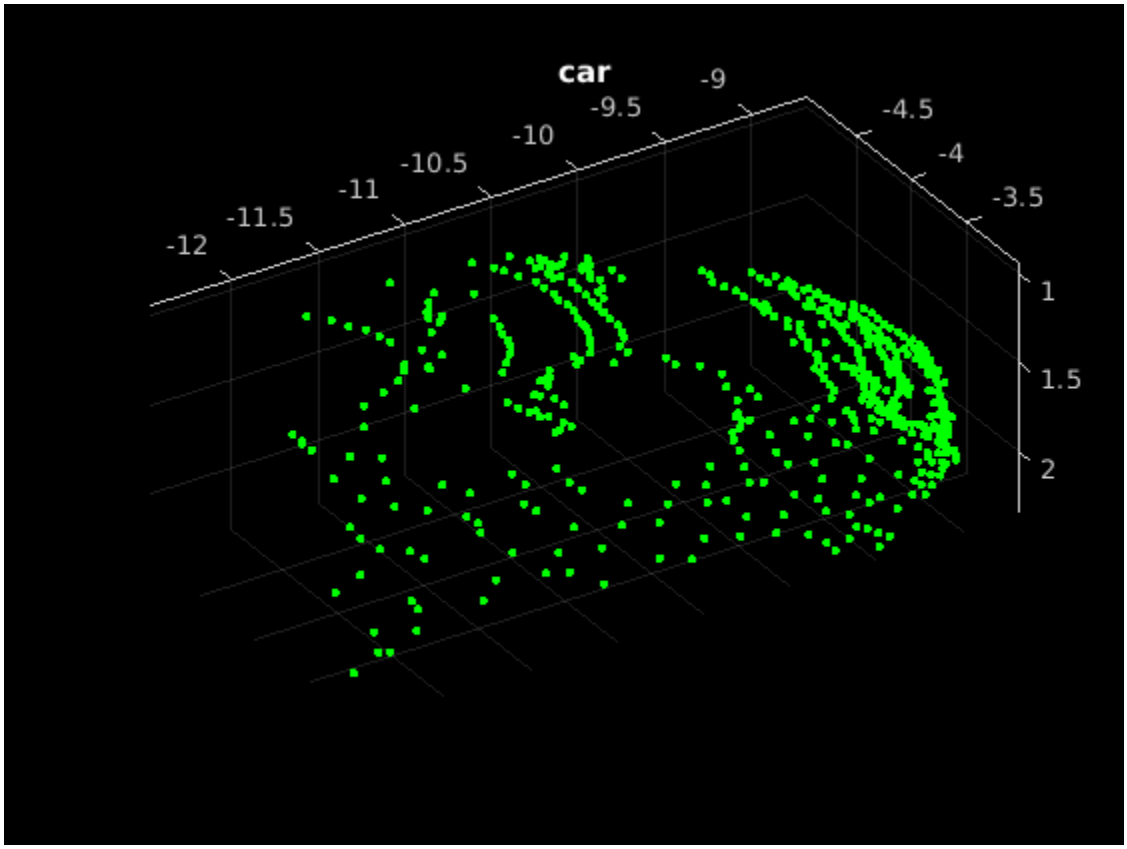
```
ptCloud = pcread("car.pcd");
X = preprocessPointCloud(ptCloud);
dlX = dlarray(X{1},"SCSB");
```

Predict point cloud labels with the `pointnetClassifier` model function.

```
YPred = pointnetClassifier(dlX,parameters,state,false);
[~,classIdx] = max(YPred,[],1);
```

Display the point cloud and the predicted label with the highest score.

```
figure
pcshow(ptCloud.Location,[0 1 0],"MarkerSize",40,"VerticalAxisDir","down")
title(classes(classIdx))
```



Model Gradients Function

The modelGradients function takes as input a mini-batch of data `dlX`, the corresponding target `dlY`, and the learnable parameters, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. The loss includes a regularization term designed to ensure the feature transformation matrix predicted by the PointNet encoder is approximately orthogonal. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```
function [gradients, loss, state, acc] = modelGradients(X,Y,parameters,state)

% Execute the model function.
isTraining = true;
[YPred,state,dlT] = pointnetClassifier(X,parameters,state,isTraining);

% Add regularization term to ensure feature transform matrix is
% approximately orthogonal.
K = size(dlT,1);
B = size(dlT, 4);
I = repelem(eye(K),1,1,1,B);
dlI = dlarray(I,"SSCB");
treg = mse(dlI,pagemtimes(dlT,permute(dlT,[2 1 3 4])));
factor = 0.001;
```

```
% Compute the loss.
loss = crossentropy(YPred,Y) + factor*treg;

% Compute the parameter gradients with respect to the loss.
gradients = dlgradient(loss, parameters);

% Compute training accuracy metric.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(YPred,[],1);
acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));

end
```

PointNet Classifier Function

The `pointnetClassifier` function takes as input the point cloud data dlX, the learnable model parameters, the model state, and the flag isTraining, which specifies whether the model returns outputs for training or prediction. Then, the function invokes the PointNet encoder and a multilayer perceptron to extract classification features. During training, dropout is applied after each perceptron operation. After the last perceptron, a `fullyconnect` operation maps the classification features to the number of classes and a softmax activation is used to normalize the output into a probability distribution of labels. The probability distribution, the updated model state, and the feature transformation matrix predicted by the PointNet encoder are returned as outputs.

```
function [dlY,state,dlT] = pointnetClassifier(dlX,parameters,state,isTraining)

% Invoke the PointNet encoder.
[dlY,state,dlT] = pointnetEncoder(dlX,parameters,state,isTraining);

% Invoke the classifier.
p = parameters.ClassificationMLP.Perceptron;
s = state.ClassificationMLP.Perceptron;
for k = 1:numel(p)

    [dlY, s(k)] = perceptron(dlY,p(k),s(k),isTraining);

    % If training, apply inverted dropout with a probability of 0.3.
    if isTraining
        probability = 0.3;
        dropoutScaleFactor = 1 - probability;
        dropoutMask = ( rand(size(dlY), "like", dlY) > probability ) / dropoutScaleFactor;
        dlY = dlY.*dropoutMask;
    end

end
state.ClassificationMLP.Perceptron = s;

% Apply final fully connected and softmax operations.
weights = parameters.ClassificationMLP.FC.Weights;
bias = parameters.ClassificationMLP.FC.Bias;
dlY = fullyconnect(dlY,weights,bias);
dlY = softmax(dlY);
end
```

PointNet Encoder Function

The `pointnetEncoder` function processes the input dlX using an input transform, a shared MLP, a feature transform, a second shared MLP, and a max operation, and returns the result of the max operation.

```
function [dlY,state,T] = pointnetEncoder(dlX,parameters,state,isTraining)
% Input transform.
[dlY,state.InputTransform] = dataTransform(dlX,parameters.InputTransform,state.InputTransform,is

% Shared MLP.
[dlY,state.SharedMLP1.Perceptron] = sharedMLP(dlY,parameters.SharedMLP1.Perceptron,state.SharedMl

% Feature transform.
[dlY,state.FeatureTransform,T] = dataTransform(dlY,parameters.FeatureTransform,state.FeatureTrans

% Shared MLP.
[dlY,state.SharedMLP2.Perceptron] = sharedMLP(dlY,parameters.SharedMLP2.Perceptron,state.SharedMl

% Max operation.
dlY = max(dlY,[],1);
end
```

Shared Multilayer Perceptron Function

The shared multilayer `perceptron` function processes the input dlX using a series of perceptron operations and returns the result of the last perceptron.

```
function [dlY,state] = sharedMLP(dlX,parameters,state,isTraining)
dlY = dlX;
for k = 1:numel(parameters)
    [dlY, state(k)] = perceptron(dlY,parameters(k),state(k),isTraining);
end
end
```

Perceptron Function

The perceptron function processes the input dlX using a convolution, a batch normalization, and a relu operation and returns the output of the ReLU operation.

```
function [dlY,state] = perceptron(dlX,parameters,state,isTraining)
% Convolution.
W = parameters.Conv.Weights;
B = parameters.Conv.Bias;
dlY = dlconv(dlX,W,B);

% Batch normalization. Update batch normalization state when training.
offset = parameters.BatchNorm.Offset;
scale = parameters.BatchNorm.Scale;
trainedMean = state.BatchNorm.TrainedMean;
trainedVariance = state.BatchNorm.TrainedVariance;
if isTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state.
    state.BatchNorm.TrainedMean = trainedMean;
    state.BatchNorm.TrainedVariance = trainedVariance;
else
```

```
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

% ReLU.
dlY = relu(dlY);
end
```

Data Transform Function

The `dataTransform` function processes the input `dlX` using a shared MLP, a max operation, and another shared MLP to predict a transformation matrix T. The transformation matrix is applied to the input dlX using a batched matrix multiply operation. The function returns the result of the batched matrix multiply and the transformation matrix.

```
function [dlY,state,T] = dataTransform(dlX,parameters,state,isTraining)

% Shared MLP.
[dlY,state.Block1.Perceptron] = sharedMLP(dlX,parameters.Block1.Perceptron,state.Block1.Perceptr

% Max operation.
dlY = max(dlY,[],1);

% Shared MLP.
[dlY,state.Block2.Perceptron] = sharedMLP(dlY,parameters.Block2.Perceptron,state.Block2.Perceptr

% Transform net (T-Net). Apply last fully connected operation as W*X to
% predict tranformation matrix T.
dlY = squeeze(dlY); % N-by-B
T = parameters.Transform * stripdims(dlY); % K^2-by-B

% Reshape T into a square matrix.
K = sqrt(size(T,1));
T = reshape(T,K,K,1,[]); % [K K 1 B]
T = T + eye(K);

% Apply to input dlX using batch matrix multiply.
[C,B] = size(dlX,[3 4]);  % [M 1 K B]
dlX = reshape(dlX,[],C,1,B); % [M K 1 B]
Y = pagemtimes(dlX,T);
dlY = dlarray(Y,"SCSB");
end
```

Model Parameter Initialization Functions

`initializeTransform` Function

The `initializeTransform` function takes as input the channel size and the number of hidden channels for the two shared MLPs, and returns the initialized parameters in a struct. The parameters are initialized using He weight initialization [3 on page 8-0].

```
function [params,state] = initializeTransform(inputChannelSize,block1,block2)
[params.Block1,state.Block1] = initializeSharedMLP(inputChannelSize,block1);
[params.Block2,state.Block2] = initializeSharedMLP(block1(end),block2);

% Parameters for the transform matrix.
params.Transform = dlarray(zeros(inputChannelSize^2,block2(end)));
end
```

`initializeSharedMLP` Function

The initializeSharedMLP function takes as input the channel size and the hidden channel size, and returns the initialized parameters in a struct. The parameters are initialized using He weight initialization.

```
function [params,state] = initializeSharedMLP(inputChannelSize,hiddenChannelSize)
weights = initializeWeightsHe([1 1 inputChannelSize hiddenChannelSize(1)]);
bias = zeros(hiddenChannelSize(1),1,"single");
p.Conv.Weights = dlarray(weights);
p.Conv.Bias = dlarray(bias);

p.BatchNorm.Offset = dlarray(zeros(hiddenChannelSize(1),1,"single"));
p.BatchNorm.Scale = dlarray(ones(hiddenChannelSize(1),1,"single"));

s.BatchNorm.TrainedMean = zeros(hiddenChannelSize(1),1,"single");
s.BatchNorm.TrainedVariance = ones(hiddenChannelSize(1),1,"single");

params.Perceptron(1) = p;
state.Perceptron(1) = s;

for k = 2:numel(hiddenChannelSize)
    weights = initializeWeightsHe([1 1 hiddenChannelSize(k-1) hiddenChannelSize(k)]);
    bias = zeros(hiddenChannelSize(k),1,"single");
    p.Conv.Weights = dlarray(weights);
    p.Conv.Bias = dlarray(bias);

    p.BatchNorm.Offset = dlarray(zeros(hiddenChannelSize(k),1,"single"));
    p.BatchNorm.Scale = dlarray(ones(hiddenChannelSize(k),1,"single"));

    s.BatchNorm.TrainedMean = zeros(hiddenChannelSize(k),1,"single");
    s.BatchNorm.TrainedVariance = ones(hiddenChannelSize(k),1,"single");

    params.Perceptron(k) = p;
    state.Perceptron(k) = s;
end
end
```

`initializeClassificationMLP` Function

The `initializeClassificationMLP` function takes as input the channel size, the hidden channel size, and the number of classes and returns the initialized parameters in a struct. The shared MLP is initialized using He weight initialization and the final fully connected operation is initialized using random Gaussian values.

```
function [params,state] = initializeClassificationMLP(inputChannelSize,hiddenChannelSize,numClass
[params,state] = initializeSharedMLP(inputChannelSize,hiddenChannelSize);

weights = initializeWeightsGaussian([numClasses hiddenChannelSize(end)]);
bias = zeros(numClasses,1,"single");
params.FC.Weights = dlarray(weights);
params.FC.Bias = dlarray(bias);
end
```

`initializeWeightsHe` Function

The `initializeWeightsHe` function initializes parameters using He initialization.

```
function x = initializeWeightsHe(sz)
fanIn = prod(sz(1:3));
stddev = sqrt(2/fanIn);
x = stddev .* randn(sz);
end
```

`initializeWeightsGaussian` Function

The `initializeWeightsGaussian` function initializes parameters using Gaussian initialization with a standard deviation of 0.01.

```
function x = initializeWeightsGaussian(sz)
x = randn(sz,"single") .* 0.01;
end
```

Data Preprocessing Functions

`preprocessPointCloudData` Function

The `preprocessPointCloudData` function extracts the X, Y, Z point data from the input data and normalizes the data between 0 and 1. The function returns the normalized X, Y, Z data.

```
function data = preprocessPointCloud(data)

if ~iscell(data)
    data = {data};
end

numObservations = size(data,1);
for i = 1:numObservations
    % Scale points between 0 and 1.
    xlim = data{i,1}.XLimits;
    ylim = data{i,1}.YLimits;
    zlim = data{i,1}.ZLimits;

    xyzMin = [xlim(1) ylim(1) zlim(1)];
    xyzDiff = [diff(xlim) diff(ylim) diff(zlim)];

    data{i,1} = (data{i,1}.Location - xyzMin) ./ xyzDiff;
end
end
```

`selectPoints` Function

The `selectPoints` function samples the desired number of points. When the point cloud contains more than the desired number of points, the function uses `pcdownsample` to randomly select points. Otherwise, the function replicates data to produce the desired number of points.

```
function data = selectPoints(data,numPoints)
% Select the desired number of points by downsampling or replicating
% point cloud data.
numObservations = size(data,1);
for i = 1:numObservations
    ptCloud = data{i,1};
    if ptCloud.Count > numPoints
        percentage = numPoints/ptCloud.Count;
        data{i,1} = pcdownsample(ptCloud,"random",percentage);
    else
```

```
        replicationFactor = ceil(numPoints/ptCloud.Count);
        ind = repmat(1:ptCloud.Count,1,replicationFactor);
        data{i,1} = select(ptCloud,ind(1:numPoints));
    end
end
end
```

Data Augmentation Functions

The `augmentPointCloudData` function randomly rotates a point cloud about the z-axis, randomly drops 30% of the points, and randomly jitters the point location with Gaussian noise.

```
function data = augmentPointCloud(data)

numObservations = size(data,1);
for i = 1:numObservations

    ptCloud = data{i,1};

    % Rotate the point cloud about "up axis", which is Z for this data set.
    tform = randomAffine3d(...
        "XReflection", true,...
        "YReflection", true,...
        "Rotation",@randomRotationAboutZ);

    ptCloud = pctransform(ptCloud,tform);

    % Randomly drop out 30% of the points.
    if rand > 0.5
        ptCloud = pcdownsample(ptCloud,'random',0.3);
    end

    if rand > 0.5
        % Jitter the point locations with Gaussian noise with a mean of 0 and
        % a standard deviation of 0.02 by creating a random displacement field.
        D = 0.02 * randn(size(ptCloud.Location));
        ptCloud = pctransform(ptCloud,D);
    end

    data{i,1} = ptCloud;
end
end

function [rotationAxis,theta] = randomRotationAboutZ()
rotationAxis = [0 0 1];
theta = 2*pi*rand;
end
```

Supporting Functions

`aggregateConfusionMetric` Function

The `aggregateConfusionMetric` function incrementally fills a confusion matrix based on the predicted results `YPred` and the expected results `YTest`.

```
function cmat = aggreateConfusionMetric(cmat,YTest,YPred)
YTest = gather(extractdata(YTest));
YPred = gather(extractdata(YPred));
```

```
[m,n] = size(cmat);
cmat = cmat + full(sparse(YTest,YPred,1,m,n));
end
```

`initializeTrainingProgressPlot` Function

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, training accuracy, and validation accuracy.

```
function [plotter,trainAccPlotter,valAccPlotter] = initializeTrainingProgressPlot()
% Plot the loss, training accuracy, and validation accuracy.
figure

% Loss plot
subplot(2,1,1)
plotter = animatedline;
xlabel("Iteration")
ylabel("Loss")

% Accuracy plot
subplot(2,1,2)
trainAccPlotter = animatedline('Color','b');
valAccPlotter = animatedline('Color','g');
legend('Training Accuracy','Validation Accuracy','Location','northwest');
xlabel("Iteration")
ylabel("Accuracy")
end
```

`replicateFiles` Function

The `replicateFiles` function randomly oversamples a set of files and returns a set of files with `numDesired` elements.

```
function files = randReplicateFiles(files,numDesired)
n = numel(files);
ind = randi(n,numDesired,1);
files = files(ind);
end
```

`downloadSydneyUrbanObjects` Function

The `downloadSydneyUrbanObjects` function downloads the data set and saves it to a temporary directory.

```
function datapath = downloadSydneyUrbanObjects(dataLoc)

if nargin == 0
    dataLoc = pwd;
end

dataLoc = string(dataLoc);

url = "http://www.acfr.usyd.edu.au/papers/data/";
name = "sydney-urban-objects-dataset.tar.gz";

datapath = fullfile(dataLoc,'sydney-urban-objects-dataset');
if ~exist(datapath,'dir')
    disp('Downloading Sydney Urban Objects data set...');
```

```
    untar(url+name,dataLoc);
end

end
```

`loadSydneyUrbanObjectsData` Function

The `loadSydneyUrbanObjectsData` function creates a datastore for loading point cloud and label data from the Sydney Urban Objects data set.

```
function ds = loadSydneyUrbanObjectsData(datapath,folds)

if nargin == 0
    return;
end

if nargin < 2
    folds = 1:4;
end

datapath = string(datapath);
path = fullfile(datapath,'objects',filesep);

% Add folds to datastore.
foldNames{1} = importdata(fullfile(datapath,'folds','fold0.txt'));
foldNames{2} = importdata(fullfile(datapath,'folds','fold1.txt'));
foldNames{3} = importdata(fullfile(datapath,'folds','fold2.txt'));
foldNames{4} = importdata(fullfile(datapath,'folds','fold3.txt'));
names = foldNames(folds);
names = vertcat(names{:});

fullFilenames = append(path,names);
ds = fileDatastore(fullFilenames,'ReadFcn',@extractTrainingData,'FileExtensions','.bin');
end
```

`batchData` Function

The `batchData` function collates data into batches and moves data to the GPU for processing.

```
function [X,Y] = batchData(ptCloud,labels)
X = cat(4,ptCloud{:});
labels = cat(1,labels{:});
Y = onehotencode(labels,2);
end
```

`extractTrainingData` Function

The extractTrainingData function extracts point cloud and label data from the Sydney Urban Objects data set.

```
function dataOut = extractTrainingData(fname)

[pointData,intensity] = readbin(fname);

[~,name] = fileparts(fname);
name = string(name);
name = extractBefore(name,'.');
name = replace(name,'_',' ');
```

```
labelNames = ["4wd","building","bus","car","pedestrian","pillar",...
    "pole","traffic lights","traffic sign","tree","truck","trunk","ute","van"];

label = categorical(name,labelNames);

dataOut = {pointCloud(pointData,'Intensity',intensity),label};

end
```

`readbin` Function

The `readbin` function reads the point cloud data from Sydney Urban Object binary files.

```
function [pointData,intensity] = readbin(fname)
% readbin Read point and intensity data from Sydney Urban Object binary
% files.

% names = ['t','intensity','id',...
%          'x','y','z',...
%          'azimuth','range','pid']
%
% formats = ['int64', 'uint8', 'uint8',...
%            'float32', 'float32', 'float32',...
%            'float32', 'float32', 'int32']

fid = fopen(fname, 'r');
c = onCleanup(@() fclose(fid));

fseek(fid,10,-1); % Move to the first X point location 10 bytes from beginning
X = fread(fid,inf,'single',30);
fseek(fid,14,-1);
Y = fread(fid,inf,'single',30);
fseek(fid,18,-1);
Z = fread(fid,inf,'single',30);

fseek(fid,8,-1);
intensity = fread(fid,inf,'uint8',33);

pointData = [X,Y,Z];
end
```

`downloadPretrainedPointNet` Function

The `downloadPretrainedPointNet` function downloads a pretrained pointnet model.

```
function data = downloadPretrainedPointNet(pretrainedURL)
% Download and load a pretrained pointnet model.
if ~exist('pointnetSydneyUrbanObjects.mat', 'file')
    if ~exist('pointnetSydneyUrbanObjects.zip', 'file')
        disp('Downloading pretrained detector (5 MB)...');
        websave('pointnetSydneyUrbanObjects.zip', pretrainedURL);
    end
    unzip('pointnetSydneyUrbanObjects.zip');
end
data = load("pointnetSydneyUrbanObjects.mat");
end
```

`prepareForPrediction` Function

The `prepareForPrediction` function is used to apply a user-defined function to nested structure data. It is a used to move model parameter and state data to the GPU.

```
function p = prepareForPrediction(p,fcn)

for i = 1:numel(p)
    p(i) = structfun(@(x)invoke(fcn,x),p(i),'UniformOutput',0);
end

    function data = invoke(fcn,data)
        if isstruct(data)
            data = prepareForPrediction(data,fcn);
        else
            data = fcn(data);
        end
    end
end

% Move data to the GPU.
function x = toDevice(x,useGPU)
if useGPU
    x = gpuArray(x);
end
end
```

References

[1] Charles, R. Qi, Hao Su, Mo Kaichun, and Leonidas J. Guibas. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation.” In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 77–85. Honolulu, HI: IEEE, 2017. https://doi.org/10.1109/CVPR.2017.16.

[2] de Deuge, Mark, Alastair Quadras, Calvin Hung, and Bertrand Douillard. "Unsupervised Feature Learning for Classification of Outdoor 3D Scans." In *Australasian Conference on Robotics and Automation 2013 (ACRA 13)*. Sydney, Australia: ACRA, 2013.

[3] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.” In *2015 IEEE International Conference on Computer Vision (ICCV)*, 1026–34. Santiago, Chile: IEEE, 2015. https://doi.org/10.1109/ICCV.2015.123.

See Also

More About

- “Getting Started with Point Clouds Using Deep Learning” (Computer Vision Toolbox)
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

- “Automatic Differentiation Background” on page 18-200

Activity Recognition from Video and Optical Flow Data Using Deep Learning

This example first shows how to perform activity recognition using a pretrained Inflated 3-D (I3D) two-stream convolutional neural network based video classifier and then shows how to use transfer learning to train such a video classifier using RGB and optical flow data from videos [1] on page 8-0 .

Overview

Vision-based activity recognition involves predicting the action of an object, such as walking, swimming, or sitting, using a set of video frames. Activity recognition from video has many applications, such as human-computer interaction, robot learning, anomaly detection, surveillance, and object detection. For example, online prediction of multiple actions for incoming videos from multiple cameras can be important for robot learning. Compared to image classification, action recognition using videos is challenging to model because of the inaccurate ground truth data for video data sets, the variety of gestures that actors in a video can perform, the heavily class imbalanced datasets, and the large amount of data required to train a robust classifier from scratch. Deep learning techniques, such as I3D two-stream convolutional networks [1] on page 8-0 , R(2+1)D [4 on page 8-0], and SlowFast [5 on page 8-0] have shown improved performance on smaller datasets using transfer learning with networks pretrained on large video activity recognition datasets, such as Kinetics-400 [6 on page 8-0].

Note: This example requires the Computer Vision Toolbox™ Model for Inflated-3D Video Classification. You can install the Computer Vision Toolbox Model for Inflated-3D Video Classification from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Activity Recognition Using a Pretrained Inflated-3D Video Classifier

Download the pretrained Inflated-3D video classifier along with a video file on which to perform activity recognition. The size of the downloaded zip file is around 89 MB.

```
downloadFolder = fullfile(tempdir,"hmdb51","pretrained","I3D");
if ~isfolder(downloadFolder)
    mkdir(downloadFolder);
end

filename = "activityRecognition-I3D-HMDB51-21b.zip";

zipFile = fullfile(downloadFolder,filename);
if ~isfile(zipFile)
    disp('Downloading the pretrained network...');
    downloadURL = "https://ssd.mathworks.com/supportfiles/vision/data/" + filename;
    websave(zipFile,downloadURL);
    unzip(zipFile,downloadFolder);
end
```

Load the pretrained Inflated-3D video classifier.

```
pretrainedDataFile = fullfile(downloadFolder,"inflated3d-FiveClasses-hmdb51.mat");
pretrained = load(pretrainedDataFile);
inflated3dPretrained = pretrained.data.inflated3d;
```

Display the class label names of the pretrained video classifier.

```
classes = inflated3dPretrained.Classes
```

```
classes = 5×1 categorical
     kiss
     laugh
     pick
     pour
     pushup
```

Read and display the video `pour.avi` using `VideoReader` and `vision.VideoPlayer`.

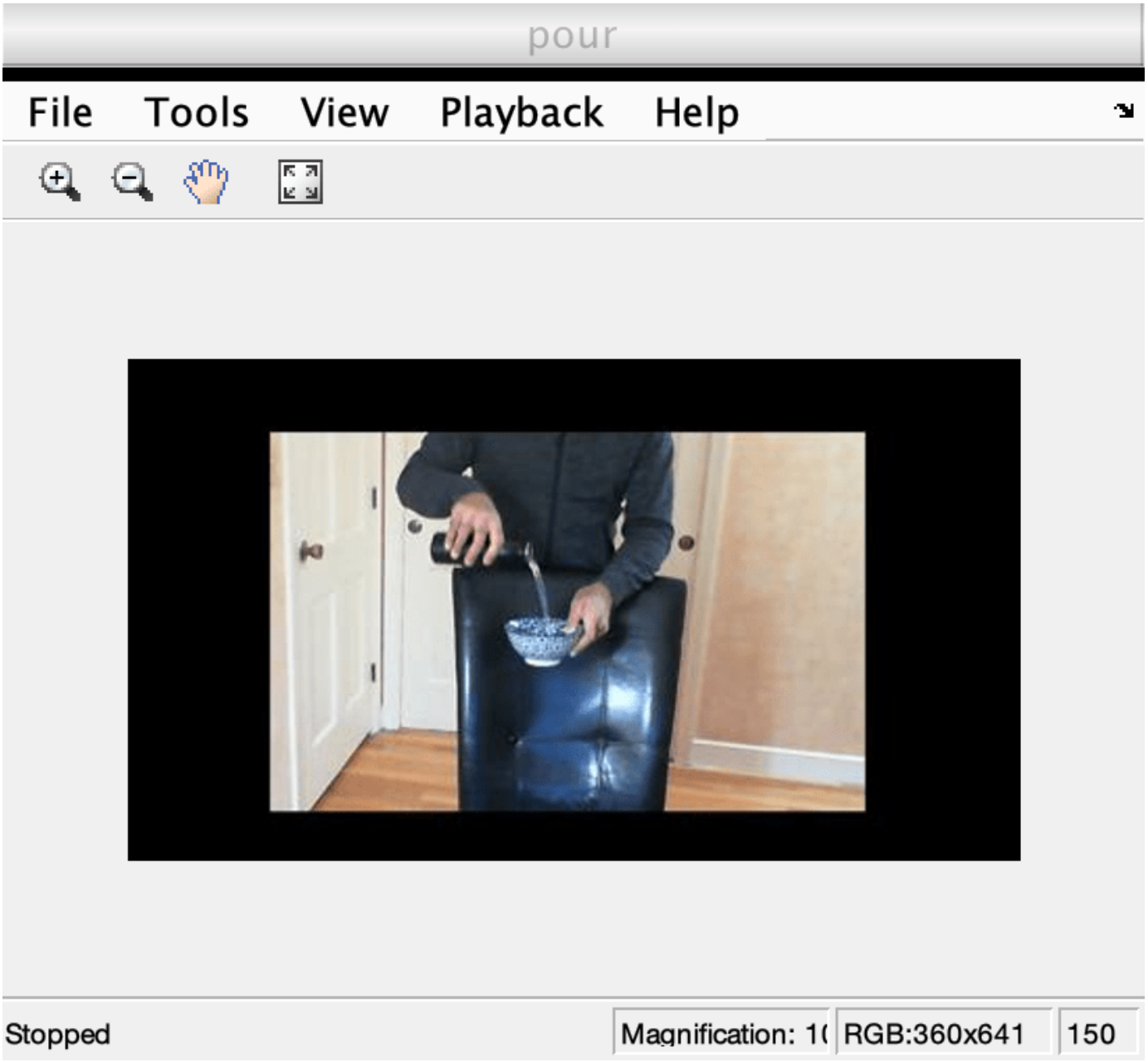
```
videoFilename = fullfile(downloadFolder, "pour.avi");

videoReader = VideoReader(videoFilename);
videoPlayer = vision.VideoPlayer;
videoPlayer.Name = "pour";

while hasFrame(videoReader)
   frame = readFrame(videoReader);
   % Resize the frame for display.
   frame = imresize(frame, 1.5);
   step(videoPlayer,frame);
end
release(videoPlayer);
```

Choose 10 randomly selected video sequences to classify the video, to uniformly cover the entirety of the file to find the action class that is predominant in the video.

```
numSequences = 10;
```

Classify the video file using the `classifyVideoFile` function.

```
[actionLabel,score] = classifyVideoFile(inflated3dPretrained, videoFilename, "NumSequences", numS
```




```
actionLabel = categorical
     pour


score = single
    0.4482
```

Train a Video Classifier for Gesture Recognition

This section of the example shows how the video classifier shown above is trained using transfer learning. Set the `doTraining` variable to `false` to use the pretrained video classifier without having

to wait for training to complete. Alternatively, if you want to train the video classifier, set the `doTraining` variable to `true`.

```
doTraining = false;
```

Download Training and Validation Data

This example trains an Inflated-3D (I3D) Video Classifier using the HMDB51 data set. Use the `downloadHMDB51` supporting function, listed at the end of this example, to download the HMDB51 data set to a folder named `hmdb51`.

```
downloadFolder = fullfile(tempdir,"hmdb51");
downloadHMDB51(downloadFolder);
```

After the download is complete, extract the RAR file `hmdb51_org.rar` to the `hmdb51` folder. Next, use the `checkForHMDB51Folder` supporting function, listed at the end of this example, to confirm that the downloaded and extracted files are in place.

```
allClasses = checkForHMDB51Folder(downloadFolder);
```

The data set contains about 2 GB of video data for 7000 clips over 51 classes, such as *drink*, *run*, and *shake hands*. Each video frame has a height of 240 pixels and a minimum width of 176 pixels. The number of frames ranges from 18 to approximately 1000.

To reduce training time, this example trains an activity recognition network to classify 5 action classes instead of all 51 classes in the data set. Set `useAllData` to `true` to train with all 51 classes.

```
useAllData = false;

if useAllData
    classes = allClasses;
end
dataFolder = fullfile(downloadFolder, "hmdb51_org");
```

Split the data set into a training set for training the classifier, and a test set for evaluating the classifier. Use 80% of the data for the training set and the rest for the test set. Use `folders2labels` and `splitlabels` to create label information from folders and split the data based on each label into training and test data sets by randomly selecting a proportion of files from each label.

```
[labels,files] = folders2labels(fullfile(dataFolder,string(classes)),...
    "IncludeSubfolders",true,...
    "FileExtensions",'.avi');

indices = splitlabels(labels,0.8,'randomized');

trainFilenames = files(indices{1});
testFilenames  = files(indices{2});
```

To normalize the input data for the network, the minimum and maximum values for the data set are provided in the MAT file `inputStatistics.mat`, attached to this example. To find the minimum and maximum values for a different data set, use the `inputStatistics` supporting function, listed at the end of this example.

```
inputStatsFilename = 'inputStatistics.mat';
if ~exist(inputStatsFilename, 'file')
    disp("Reading all the training data for input statistics...")
    inputStats = inputStatistics(dataFolder);
```

```
else
    d = load(inputStatsFilename);
    inputStats = d.inputStats;
end
```

Load Dataset

This example uses a datastore to read the videos scenes, the corresponding optical flow data, and the corresponding labels from the video files.

Specify the number of video frames the datastore should be configured to output for each time data is read from the datastore.

```
numFrames = 64;
```

A value of 64 is used here to balance memory usage and classification time. Common values to consider are 16, 32, 64, or 128. Using more frames helps capture additional temporal information, but requires more memory. You might need to lower this value depending on your system resources. Empirical analysis is required to determine the optimal number of frames.

Next, specify the height and width of the frames the datastore should be configured to output. The datastore automatically resizes the raw video frames to the specified size to enable batch processing of multiple video sequences.

```
frameSize = [112,112];
```

A value of [112 112] is used to capture longer temporal relationships in the video scene which help classify activities with long time durations. Common values for the size are [112 112], [224 224], or [256 256]. Smaller sizes enable the use of more video frames at the cost of memory usage, processing time, and spatial resolution. The minimum height and width of the video frames in the HMDB51 data set are 240 and 176, respectively. If you want to specify a frame size for the datastore to read that is larger than the minimum values, such as [256, 256], first resize the frames using `imresize`. As with the number of frames, empirical analysis is required to determine the optimal values.

Specify the number of channels as 3 for the RGB video subnetwork, and 2 for the optical flow subnetwork of the I3D video classifier. The two channels for optical flow data are the x and y components of velocity, V_x and V_y, respectively.

```
rgbChannels = 3;
flowChannels = 2;
```

Use the helper function, `createFileDatastore`, to configure two `FileDatastore` objects for loading the data, one for training and another for validation. The helper function is listed at the end of this example. Each datastore reads a video file to provide RGB data and the corresponding label information.

```
isDataForTraining = true;
dsTrain = createFileDatastore(trainFilenames,numFrames,rgbChannels,classes,isDataForTraining);

isDataForTraining = false;
dsVal = createFileDatastore(testFilenames,numFrames,rgbChannels,classes,isDataForTraining);
```

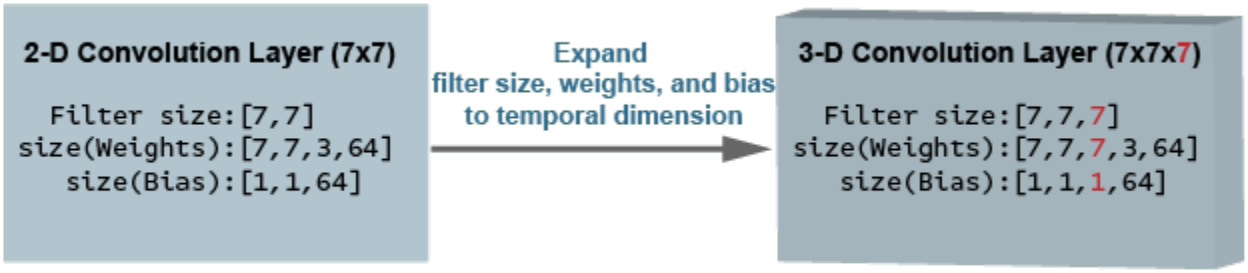
Define Network Architecture

I3D network

Using a 3-D CNN is a natural approach to extracting spatio-temporal features from videos. You can create an I3D network from a pretrained 2-D image classification network such as Inception v1 or

ResNet-50 by expanding 2-D filters and pooling kernels into 3-D. This procedure reuses the weights learned from the image classification task to bootstrap the video recognition task.

The following figure is a sample showing how to inflate a 2-D convolution layer to a 3-D convolution layer. The inflation involves expanding the filter size, weights, and bias by adding a third dimension (the temporal dimension).

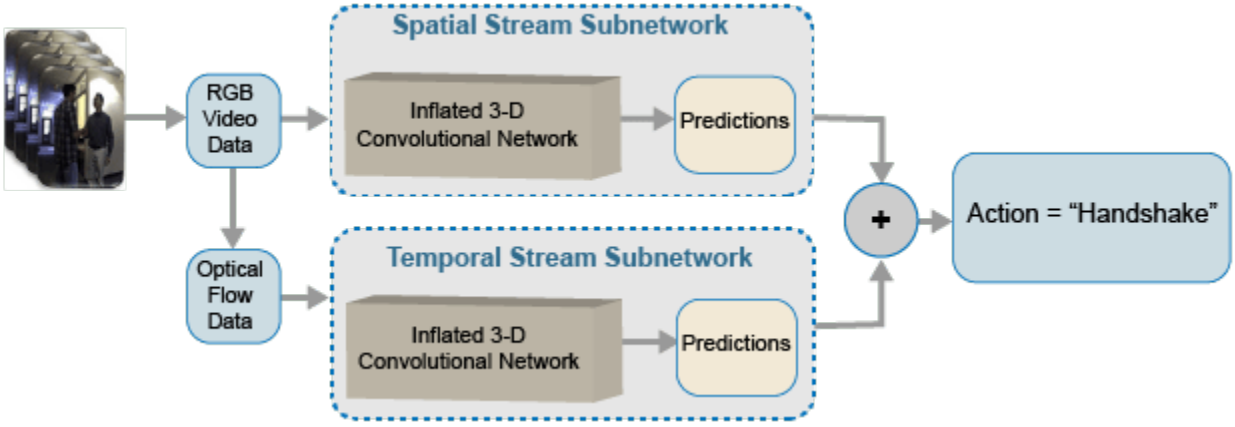


Two-Stream I3D Network

Video data can be considered to have two parts: a spatial component and a temporal component.

- The spatial component comprises information about the shape, texture, and color of objects in video. RGB data contains this information.
- The temporal component comprises information about the motion of objects across the frames and depicts important movements between the camera and the objects in a scene. Computing optical flow is a common technique for extracting temporal information from video.

A two-stream CNN incorporates a spatial subnetwork and a temporal subnetwork [2] on page 8-0 . A convolutional neural network trained on dense optical flow and a video data stream can achieve better performance with limited training data than with raw stacked RGB frames. The following illustration shows a typical two-stream I3D network.



Configure Inflated-3D (I3D) Video Classifier for Transfer Learning

In this example, you create an I3D video classifier based on the GoogLeNet architecture, a 3D Convolution Neural Network Video Classifier pretrained on the Kinetics-400 dataset.

Specify GoogLeNet as the backbone convolution neural network architecture for the I3D video classifier that contains two subnetworks, one for video data and another for optical flow data.

```
baseNetwork = "googlenet-video-flow";
```

Specify the input size for the Inflated-3D Video Classifier.

```
inputSize = [frameSize, rgbChannels, numFrames];
```

Obtain the minimum and maximum values for the RGB and optical flow data from the `inputStats` structure loaded from the `inputStatistics.mat` file. These values are needed to normalize the input data.

```
oflowMin = squeeze(inputStats.oflowMin)';
oflowMax = squeeze(inputStats.oflowMax)';
rgbMin   = squeeze(inputStats.rgbMin)';
rgbMax   = squeeze(inputStats.rgbMax)';

stats.Video.Min               = rgbMin;
stats.Video.Max               = rgbMax;
stats.Video.Mean              = [];
stats.Video.StandardDeviation = [];

stats.OpticalFlow.Min               = oflowMin(1:flowChannels);
stats.OpticalFlow.Max               = oflowMax(1:flowChannels);
stats.OpticalFlow.Mean              = [];
stats.OpticalFlow.StandardDeviation = [];
```

Create the I3D Video Classifier by using the `inflated3dVideoClassifier` function.

```
i3d = inflated3dVideoClassifier(baseNetwork,string(classes),...
    "InputSize",inputSize,...
    "InputNormalizationStatistics",stats);
```

Specify a model name for the video classifier.

```
i3d.ModelName = "Inflated-3D Activity Recognizer Using Video and Optical Flow";
```

Augment and Preprocess Training Data

Data augmentation provides a way to use limited data sets for training. Augmentation on video data must be the same for a collection of frames, i.e. a video sequence, based on the network input size. Minor changes, such as translation, cropping, or transforming an image, provide, new, distinct, and unique images that you can use to train a robust video classifier. Datastores are a convenient way to read and augment collections of data. Augment the training video data by using the `augmentVideo` supporting function, defined at the end of this example.

```
dsTrain = transform(dsTrain, @augmentVideo);
```

Preprocess the training video data to resize to the Inflated-3D Video Classifier input size, by using the `preprocessVideoClips`, defined at the end of this example. Specify the `InputNormalizationStatistics` property of the video classifier and input size to the preprocessing function as field values in a struct, `preprocessInfo`. The `InputNormalizationStatistics` property is used to rescale the video frames and optical flow data between -1 and 1. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` to random crop or center crop the input data to the input size of the video classifier. Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
preprocessInfo.Statistics = i3d.InputNormalizationStatistics;
preprocessInfo.InputSize = inputSize;
preprocessInfo.SizingOption = "resize";
dsTrain = transform(dsTrain, @(data)preprocessVideoClips(data, preprocessInfo));
dsVal = transform(dsVal, @(data)preprocessVideoClips(data, preprocessInfo));
```

Define Model Gradients Function

Create the supporting function `modelGradients`, listed at the end of this example. The `modelGradients` function takes as input the I3D video classifier `i3d`, a mini-batch of input data `dlRGB` and `dlFlow`, and a mini-batch of ground truth label data `dlY`. The function returns the training loss value, the gradients of the loss with respect to the learnable parameters of the classifier, and the mini-batch accuracy of the classifier.

The loss is calculated by computing the average of the cross-entropy losses of the predictions from each of the subnetworks. The output predictions of the network are probabilities between 0 and 1 for each of the classes.

$$rgbLoss = crossentropy(rgbPrediction)$$

$$flowLoss = crossentropy(flowPrediction)$$

$$loss = mean([rgbLoss, flowLoss])$$

The accuracy of each of the classifier is calculated by taking the average of the RGB and optical flow predictions, and comparing it to the ground truth label of the inputs.

Specify Training Options

Train with a mini-batch size of 20 for 600 iterations. Specify the iteration after which to save the video classifier with the best validation accuracy by using the `SaveBestAfterIteration` parameter.

Specify the cosine-annealing learning rate schedule [3 on page 8-0] parameters:

- A minimum learning rate of 1e-4.
- A maximum learning rate of 1e-3.
- Cosine number of iterations of 100, 200, and 300, after which the learning rate schedule cycle restarts. The option `CosineNumIterations` defines the width of each cosine cycle.

Specify the parameters for SGDM optimization. Initialize the SGDM optimization parameters at the beginning of the training:

- A momentum of 0.9.
- An initial velocity parameter initialized as `[]`.
- An L2 regularization factor of 0.0005.

Specify to dispatch the data in the background using a parallel pool. If `DispatchInBackground` is set to true, open a parallel pool with the specified number of parallel workers, and create a `DispatchInBackgroundDatastore`, provided as part of this example, that dispatches the data in the background to speed up training using asynchronous data loading and preprocessing. By default, this example uses a GPU if one is available. Otherwise, it uses a CPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

```
params.Classes = classes;
params.MiniBatchSize = 20;
params.NumIterations = 600;
params.SaveBestAfterIteration = 400;
params.CosineNumIterations = [100, 200, 300];
params.MinLearningRate = 1e-4;
params.MaxLearningRate = 1e-3;
params.Momentum = 0.9;
params.VelocityRGB = [];
params.VelocityFlow = [];
params.L2Regularization = 0.0005;
params.ProgressPlot = true;
params.Verbose = true;
params.ValidationData = dsVal;
params.DispatchInBackground = false;
params.NumWorkers = 4;
```

Train I3D Video Classifier

Train the I3D video classifier using the RGB video data and optical flow data.

For each epoch:

- Shuffle the data before looping over mini-batches of data.
- Use `minibatchqueue` to loop over the mini-batches. The supporting function `createMiniBatchQueue`, listed at the end of this example, uses the given training datastore to create a `minibatchqueue`.
- Use the validation data `dsVal` to validate the networks.
- Display the loss and accuracy results for each epoch using the supporting function `displayVerboseOutputEveryEpoch`, listed at the end of this example.

For each mini-batch:

- Convert the video data or optical flow data and the labels to `dlarray` objects with the underlying type single.
- To enable processing the time dimension of the the video data using the I3D Video Classifier specify the temporal sequence dimension, `"T"`. Specify the dimension labels `"SSCTB"` (spatial, spatial, channel, temporal, batch) for the video data, and `"CB"` for the label data.

The `minibatchqueue` object uses the supporting function `batchVideoAndFlow`, listed at the end of this example, to batch the RGB video and optical flow data.

```
params.ModelFilename = "inflated3d-FiveClasses-hmdb51.mat";
if doTraining
    epoch     = 1;
    bestLoss  = realmax;

    accTrain     = [];
    accTrainRGB  = [];
    accTrainFlow = [];
    lossTrain    = [];

    iteration = 1;
    start     = tic;
    trainTime = start;
```

```
shuffled  = shuffleTrainDs(dsTrain);

% Number of outputs is three: One for RGB frames, one for optical flow
% data, and one for ground truth labels.
numOutputs = 3;
mbq        = createMiniBatchQueue(shuffled, numOutputs, params);

% Use the initializeTrainingProgressPlot and initializeVerboseOutput
% supporting functions, listed at the end of the example, to initialize
% the training progress plot and verbose output to display the training
% loss, training accuracy, and validation accuracy.
plotters = initializeTrainingProgressPlot(params);
initializeVerboseOutput(params);

while iteration <= params.NumIterations

    % Iterate through the data set.
    [dlVideo,dlFlow,dlY] = next(mbq);

    % Evaluate the model gradients and loss using dlfeval.
    [gradRGB,gradFlow,loss,acc,accRGB,accFlow,stateRGB,stateFlow] = ...
        dlfeval(@modelGradients,i3d,dlVideo,dlFlow,dlY);

    % Accumulate the loss and accuracies.
    lossTrain    = [lossTrain, loss];
    accTrain     = [accTrain, acc];
    accTrainRGB  = [accTrainRGB, accRGB];
    accTrainFlow = [accTrainFlow, accFlow];

    % Update the network state.
    i3d.VideoState       = stateRGB;
    i3d.OpticalFlowState = stateFlow;

    % Update the gradients and parameters for the RGB and optical flow
    % subnetworks using the SGDM optimizer.
    [i3d.VideoLearnables,params.VelocityRGB] = ...
        updateLearnables(i3d.VideoLearnables,gradRGB,params,params.VelocityRGB,iteration);
    [i3d.OpticalFlowLearnables,params.VelocityFlow,learnRate] = ...
        updateLearnables(i3d.OpticalFlowLearnables,gradFlow,params,params.VelocityFlow,itera

    if ~hasdata(mbq) || iteration == params.NumIterations
        % Current epoch is complete. Do validation and update progress.
        trainTime = toc(trainTime);

        [validationTime,cmat,lossValidation,accValidation,accValidationRGB,accValidationFlow
            doValidation(params, i3d);

        accTrain     = mean(accTrain);
        accTrainRGB  = mean(accTrainRGB);
        accTrainFlow = mean(accTrainFlow);
        lossTrain    = mean(lossTrain);

        % Update the training progress.
        displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
            accTrain,accTrainRGB,accTrainFlow,...
            accValidation,accValidationRGB,accValidationFlow,...
            lossTrain,lossValidation,trainTime,validationTime);
        updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accTrain,accValida
```

```
            % Save the trained video classifier and the parameters, that gave
            % the best validation loss so far. Use the saveData supporting function,
            % listed at the end of this example.
            bestLoss = saveData(i3d,bestLoss,iteration,cmat,lossTrain,lossValidation,...
                accTrain,accValidation,params);
        end

        if ~hasdata(mbq) && iteration < params.NumIterations
            % Current epoch is complete. Initialize the training loss, accuracy
            % values, and minibatchqueue for the next epoch.
            accTrain     = [];
            accTrainRGB  = [];
            accTrainFlow = [];
            lossTrain    = [];

            trainTime  = tic;
            epoch      = epoch + 1;
            shuffled   = shuffleTrainDs(dsTrain);
            numOutputs = 3;
            mbq        = createMiniBatchQueue(shuffled, numOutputs, params);

        end

        iteration = iteration + 1;
    end

    % Display a message when training is complete.
    endVerboseOutput(params);

    disp("Model saved to: " + params.ModelFilename);
end
```

Evaluate Trained Network

Use the test data set to evaluate the accuracy of the trained video classifier.

Load the best model saved during training or use the pretrained model.

```
if doTraining
    transferLearned = load(params.ModelFilename);
    inflated3dPretrained = transferLearned.data.inflated3d;
end
```

Create a `minibatchqueue` object to load batches of the test data.

```
numOutputs = 3;
mbq = createMiniBatchQueue(params.ValidationData, numOutputs, params);
```

For each batch of test data, make predictions using the RGB and optical flow networks, take the average of the predictions, and compute the prediction accuracy using a confusion matrix.

```
numClasses = numel(classes);
cmat = sparse(numClasses,numClasses);
while hasdata(mbq)
    [dlRGB, dlFlow, dlY] = next(mbq);

    % Pass the video input as RGB and optical flow data through the
```

```
    % two-stream I3D Video Classifier to get the separate predictions.
    [dlYPredRGB,dlYPredFlow] = predict(inflated3dPretrained,dlRGB,dlFlow);

    % Fuse the predictions by calculating the average of the predictions.
    dlYPred = (dlYPredRGB + dlYPredFlow)/2;

    % Calculate the accuracy of the predictions.
    [~,YTest] = max(dlY,[],1);
    [~,YPred] = max(dlYPred,[],1);

    cmat = aggregateConfusionMetric(cmat,YTest,YPred);
end
```

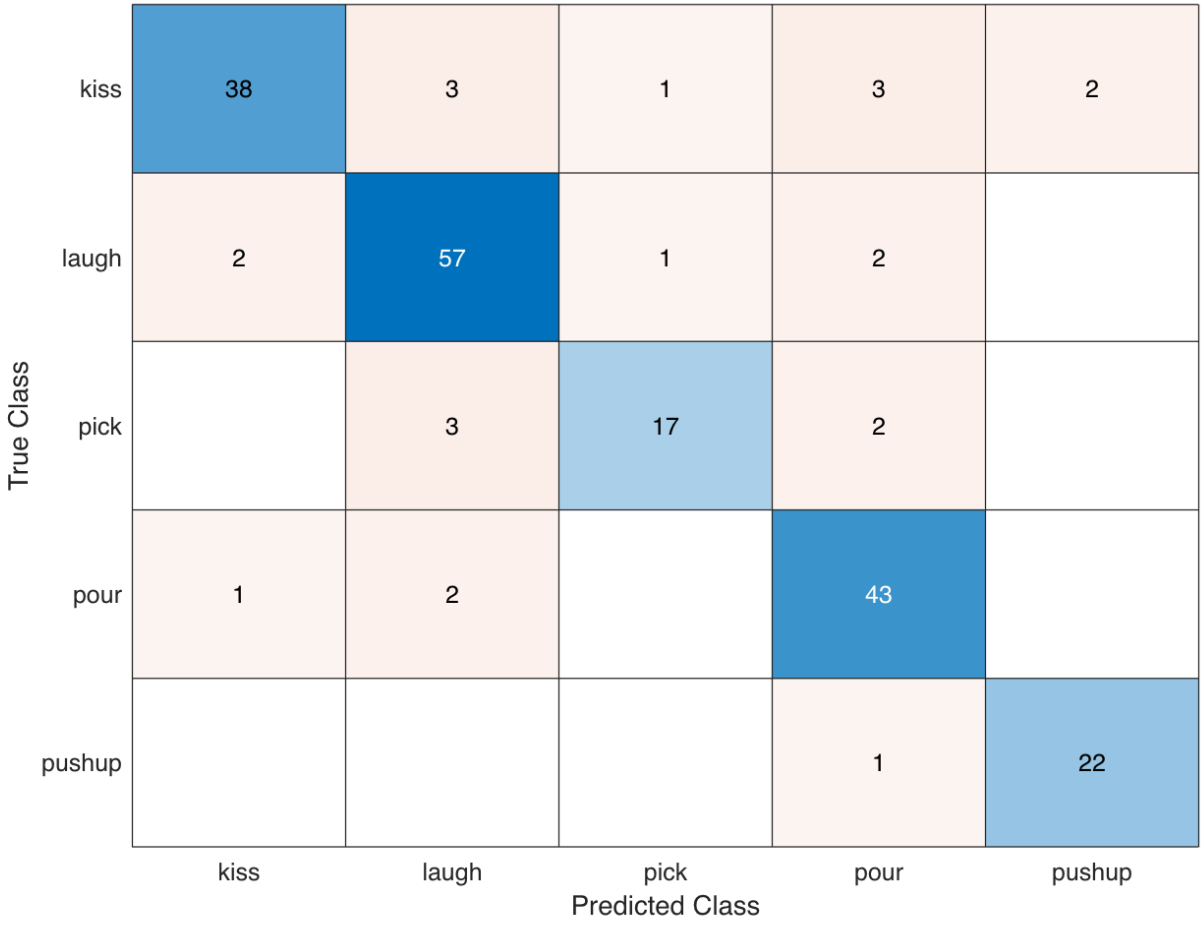
Compute the average classification accuracy for the trained networks.

```
accuracyEval = sum(diag(cmat))./sum(cmat,"all")
```

```
accuracyEval = 0.8850
```

Display the confusion matrix.

```
figure
chart = confusionchart(cmat,classes);
```



The Inflated-3D video classifier that is pretrained on the Kinetics-400 dataset, provides better performance for human activity recognition on transfer learning. The above training was run on 24GB Titan-X GPU for about 100 minutes. When training from scratch on a small activity recognition video dataset, the training time and convergence takes much longer than the pretrained video classifier. Transer learning using the Kinetics-400 pretrained Inflated-3D video classifier also avoids overfitting the classifier when ran for larger number of epochs. However, the SlowFast Video Classifier and R(2+1)D Video Classifier that are pretrained on the Kinetics-400 dataset provide better performance and faster convergence during training compared to the Inflated-3D Video Classifier. To learn more about video recognition using deep learning, see “Getting Started with Video Classification Using Deep Learning” (Computer Vision Toolbox).

Supporting Functions

`inputStatistics`

The `inputStatistics` function takes as input the name of the folder containing the HMDB51 data, and calculates the minimum and maximum values for the RGB data and the optical flow data. The minimum and maximum values are used as normalization inputs to the input layer of the networks. This function also obtains the number of frames in each of the video files to use later during training

and testing the network. In order to find the minimum and maximum values for a different data set, use this function with a folder name containing the data set.

```
function inputStats = inputStatistics(dataFolder)
    ds = createDatastore(dataFolder);
    ds.ReadFcn = @getMinMax;

    tic;
    tt = tall(ds);
    varnames = {'rgbMax','rgbMin','oflowMax','oflowMin'};
    stats = gather(groupsummary(tt,[],{'max','min'}, varnames));
    inputStats.Filename = gather(tt.Filename);
    inputStats.NumFrames = gather(tt.NumFrames);
    inputStats.rgbMax = stats.max_rgbMax;
    inputStats.rgbMin = stats.min_rgbMin;
    inputStats.oflowMax = stats.max_oflowMax;
    inputStats.oflowMin = stats.min_oflowMin;
    save('inputStatistics.mat','inputStats');
    toc;
end

function data = getMinMax(filename)
    reader = VideoReader(filename);
    opticFlow = opticalFlowFarneback;
    data = [];
    while hasFrame(reader)
        frame = readFrame(reader);
        [rgb,oflow] = findMinMax(frame,opticFlow);
        data = assignMinMax(data, rgb, oflow);
    end

    totalFrames = floor(reader.Duration * reader.FrameRate);
    totalFrames = min(totalFrames, reader.NumFrames);

    [labelName, filename] = getLabelFilename(filename);
    data.Filename = fullfile(labelName, filename);
    data.NumFrames = totalFrames;

    data = struct2table(data,'AsArray',true);
end

function data = assignMinMax(data, rgb, oflow)
    if isempty(data)
        data.rgbMax = rgb.Max;
        data.rgbMin = rgb.Min;
        data.oflowMax = oflow.Max;
        data.oflowMin = oflow.Min;
        return;
    end
    data.rgbMax = max(data.rgbMax, rgb.Max);
    data.rgbMin = min(data.rgbMin, rgb.Min);

    data.oflowMax = max(data.oflowMax, oflow.Max);
    data.oflowMin = min(data.oflowMin, oflow.Min);
end

function [rgbMinMax,oflowMinMax] = findMinMax(rgb, opticFlow)
    rgbMinMax.Max = max(rgb,[],[1,2]);
```

```
    rgbMinMax.Min = min(rgb,[],[1,2]);

    gray = rgb2gray(rgb);
    flow = estimateFlow(opticFlow,gray);
    oflow = cat(3,flow.Vx,flow.Vy,flow.Magnitude);

    oflowMinMax.Max = max(oflow,[],[1,2]);
    oflowMinMax.Min = min(oflow,[],[1,2]);
end

function ds = createDatastore(folder)
    ds = fileDatastore(folder,...
        'IncludeSubfolders', true,...
        'FileExtensions', '.avi',...
        'UniformRead', true,...
        'ReadFcn', @getMinMax);
    disp("NumFiles: " + numel(ds.Files));
end
```

createFileDatastore

The `createFileDatastore` function creates a `FileDatastore` object using the given file names. The `FileDatastore` object reads the data in `'partialfile'` mode, so every read can return partially read frames from videos. This feature helps with reading large video files, if all of the frames do not fit in memory.

```
function datastore = createFileDatastore(trainingFolder,numFrames,numChannels,classes,isDataForT
    readFcn = @(f,u)readVideo(f,u,numFrames,numChannels,classes,isDataForTraining);
    datastore = fileDatastore(trainingFolder,...
        'IncludeSubfolders',true,...
        'FileExtensions','.avi',...
        'ReadFcn',readFcn,...
        'ReadMode','partialfile');
end
```

shuffleTrainDs

The `shuffleTrainDs` function shuffles the files present in the training datastore `dsTrain`.

```
function shuffled = shuffleTrainDs(dsTrain)
shuffled = copy(dsTrain);
transformed = isa(shuffled, 'matlab.io.datastore.TransformedDatastore');
if transformed
    files = shuffled.UnderlyingDatastores{1}.Files;
else
    files = shuffled.Files;
end
n = numel(files);
shuffledIndices = randperm(n);
if transformed
    shuffled.UnderlyingDatastores{1}.Files = files(shuffledIndices);
else
    shuffled.Files = files(shuffledIndices);
end

reset(shuffled);
end
```

readVideo

The `readVideo` function reads video frames, and the corresponding label values for a given video file. During training, the read function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. During testing, all the frames are sequentially read. The video frames are resized to the required classifier network input size for training, and for testing and validation.

```
function [data,userdata,done] = readVideo(filename,userdata,numFrames,numChannels,classes,isDataF
    if isempty(userdata)
        userdata.reader      = VideoReader(filename);
        userdata.batchesRead = 0;

        userdata.label = getLabel(filename,classes);

        totalFrames = floor(userdata.reader.Duration * userdata.reader.FrameRate);
        totalFrames = min(totalFrames, userdata.reader.NumFrames);
        userdata.totalFrames = totalFrames;
        userdata.datatype = class(read(userdata.reader,1));
    end
    reader      = userdata.reader;
    totalFrames = userdata.totalFrames;
    label       = userdata.label;
    batchesRead = userdata.batchesRead;

    if isDataForTraining
        video = readForTraining(reader, numFrames, totalFrames);
    else
        video = readForValidation(reader, userdata.datatype, numChannels, numFrames, totalFrames);
    end

    data = {video, label};

    batchesRead = batchesRead + 1;

    userdata.batchesRead = batchesRead;

    if numFrames > totalFrames
        numBatches = 1;
    else
        numBatches = floor(totalFrames/numFrames);
    end
    % Set the done flag to true, if the reader has read all the frames or
    % if it is training.
    done = batchesRead == numBatches || isDataForTraining;
end
```

readForTraining

The `readForTraining` function reads the video frames for training the video classifier. The function reads the specific number of frames as per the network input size, with a randomly chosen starting frame. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForTraining(reader, numFrames, totalFrames)
    if numFrames >= totalFrames
        startIdx = 1;
```

```
        endIdx = totalFrames;
    else
        startIdx = randperm(totalFrames - numFrames + 1);
        startIdx = startIdx(1);
        endIdx = startIdx + numFrames - 1;
    end
    video = read(reader,[startIdx,endIdx]);
    if numFrames > totalFrames
        % Add more frames to fill in the network input size.
        additional = ceil(numFrames/totalFrames);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`readForValidation`

The `readForValidation` function reads the video frames for evaluating the trained video classifier. The function reads the specific number of frames sequentially as per the network input size. If there are not enough frames left over, the video sequence is repeated to pad the required number of frames.

```
function video = readForValidation(reader, datatype, numChannels, numFrames, totalFrames)
    H = reader.Height;
    W = reader.Width;
    toRead = min([numFrames,totalFrames]);
    video = zeros([H,W,numChannels,toRead], datatype);
    frameIndex = 0;
    while hasFrame(reader) && frameIndex < numFrames
        frame = readFrame(reader);
        frameIndex = frameIndex + 1;
        video(:,:,:,frameIndex) = frame;
    end

    if frameIndex < numFrames
        video = video(:,:,:,1:frameIndex);
        additional = ceil(numFrames/frameIndex);
        video = repmat(video,1,1,1,additional);
        video = video(:,:,:,1:numFrames);
    end
end
```

`getLabel`

The `getLabel` function obtains the label name from the full path of a filename. The label for a file is the folder in which it exists. For example, for a file path such as `"/path/to/dataset/clapping/video_0001.avi"`, the label name is `"clapping"`.

```
function label = getLabel(filename,classes)
    folder = fileparts(string(filename));
    [~,label] = fileparts(folder);
    label = categorical(string(label), string(classes));
end
```

`augmentVideo`

The `augmentVideo` function uses the augment transform function provided by the `augmentTransform` supporting function to apply the same augmentation across a video sequence.

```
function data = augmentVideo(data)
    numSequences = size(data,1);
    for ii = 1:numSequences
        video = data{ii,1};
        % HxWxC
        sz = size(video,[1,2,3]);
        % One augmentation per sequence
        augmentFcn = augmentTransform(sz);
        data{ii,1} = augmentFcn(video);
    end
end
```

augmentTransform

The `augmentTransform` function creates an augmentation method with random left-right flipping and scaling factors.

```
function augmentFcn = augmentTransform(sz)
% Randomly flip and scale the image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');

augmentFcn = @(data)augmentData(data,tform,rout);

    function data = augmentData(data,tform,rout)
        data = imwarp(data,tform,'OutputView',rout);
    end
end
```

preprocessVideoClips

The `preprocessVideoClips` function preprocesses the training video data to resize to the Inflated-3D Video Classifier input size. It takes the `InputNormalizationStatistics` and the `InputSize` properties of the video classifier in a struct, `info`. The `InputNormalizationStatistics` property is used to rescale the video frames and optical flow data between -1 and 1. The input size is used to resize the video frames using `imresize` based on the `SizingOption` value in the `info` struct. Alternatively, you could use `"randomcrop"` or `"centercrop"` as values for `SizingOption` to random crop or center crop the input data to the input size of the video classifier.

```
function preprocessed = preprocessVideoClips(data, info)
inputSize = info.InputSize(1:2);
sizingOption = info.SizingOption;
switch sizingOption
    case "resize"
        sizingFcn = @(x)imresize(x,inputSize);
    case "randomcrop"
        sizingFcn = @(x)cropVideo(x,@randomCropWindow2d,inputSize);
    case "centercrop"
        sizingFcn = @(x)cropVideo(x,@centerCropWindow2d,inputSize);
end
numClips = size(data,1);

rgbMin   = info.Statistics.Video.Min;
rgbMax   = info.Statistics.Video.Max;
oflowMin = info.Statistics.OpticalFlow.Min;
oflowMax = info.Statistics.OpticalFlow.Max;
```

```
numChannels = length(rgbMin);
rgbMin   = reshape(rgbMin, 1, 1, numChannels);
rgbMax   = reshape(rgbMax, 1, 1, numChannels);

numChannels = length(oflowMin);
oflowMin = reshape(oflowMin, 1, 1, numChannels);
oflowMax = reshape(oflowMax, 1, 1, numChannels);

preprocessed = cell(numClips, 3);
for ii = 1:numClips
    video   = data{ii,1};
    resized = sizingFcn(video);
    oflow   = computeFlow(resized,inputSize);

    % Cast the input to single.
    resized = single(resized);
    oflow   = single(oflow);

    % Rescale the input between -1 and 1.
    resized = rescale(resized,-1,1,"InputMin",rgbMin,"InputMax",rgbMax);
    oflow   = rescale(oflow,-1,1,"InputMin",oflowMin,"InputMax",oflowMax);

    preprocessed{ii,1} = resized;
    preprocessed{ii,2} = oflow;
    preprocessed{ii,3} = data{ii,2};
end
end

function outData = cropVideo(data, cropFcn, inputSize)
imsz = size(data,[1,2]);
cropWindow = cropFcn(imsz, inputSize);
numFrames = size(data,4);
sz = [inputSize, size(data,3), numFrames];
outData = zeros(sz, 'like', data);
for f = 1:numFrames
    outData(:,:,:,f) = imcrop(data(:,:,:,f), cropWindow);
end
end
```

`computeFlow`

The `computeFlow` function takes as input a video sequence, `videoFrames`, and computes the the corresponding optical flow data `opticalFlowData` using `opticalFlowFarneback`. The optical flow data contains two channels, which correspond to the *x*- and *y*- components of velocity.

```
function opticalFlowData = computeFlow(videoFrames, inputSize)
opticalFlow = opticalFlowFarneback;
numFrames = size(videoFrames,4);
sz = [inputSize, 2, numFrames];
opticalFlowData = zeros(sz, 'like', videoFrames);
for f = 1:numFrames
    gray = rgb2gray(videoFrames(:,:,:,f));
    flow = estimateFlow(opticalFlow,gray);

    opticalFlowData(:,:,:,f) = cat(3,flow.Vx,flow.Vy);
end
end
```

createMiniBatchQueue

The `createMiniBatchQueue` function creates a `minibatchqueue` object that provides `miniBatchSize` amount of data from the given datastore. It also creates a `DispatchInBackgroundDatastore` if a parallel pool is open.

```
function mbq = createMiniBatchQueue(datastore, numOutputs, params)
if params.DispatchInBackground && isempty(gcp('nocreate'))
    % Start a parallel pool, if DispatchInBackground is true, to dispatch
    % data in the background using the parallel pool.
    c = parcluster('local');
    c.NumWorkers = params.NumWorkers;
    parpool('local',params.NumWorkers);
end
p = gcp('nocreate');
if ~isempty(p)
    datastore = DispatchInBackgroundDatastore(datastore, p.NumWorkers);
end
inputFormat(1:numOutputs-1) = "SSCTB";
outputFormat = "CB";
mbq = minibatchqueue(datastore, numOutputs, ...
    "MiniBatchSize", params.MiniBatchSize, ...
    "MiniBatchFcn", @batchVideoAndFlow, ...
    "MiniBatchFormat", [inputFormat,outputFormat]);
end
```

batchVideoAndFlow

The `batchVideoAndFlow` function batches the video, optical flow, and label data from cell arrays. It uses `onehotencode` function to encode ground truth categorical labels into one-hot arrays. The one-hot encoded array contains a `1` in the position corresponding to the class of the label, and `0` in every other position.

```
function [video,flow,labels] = batchVideoAndFlow(video, flow, labels)
% Batch dimension: 5
video = cat(5,video{:});
flow = cat(5,flow{:});

% Batch dimension: 2
labels = cat(2,labels{:});

% Feature dimension: 1
labels = onehotencode(labels,1);
end
```

modelGradients

The `modelGradients` function takes as input a mini-batch of RGB data `dlRGB`, the corresponding optical flow data `dlFlow`, and the corresponding target `dlY`, and returns the corresponding loss, the gradients of the loss with respect to the learnable parameters, and the training accuracy. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```
function [gradientsRGB,gradientsFlow,loss,acc,accRGB,accFlow,stateRGB,stateFlow] = modelGradients

% Pass video input as RGB and optical flow data through the two-stream
% network.
```

```
[dlYPredRGB,dlYPredFlow,stateRGB,stateFlow] = forward(i3d,dlRGB,dlFlow);

% Calculate fused loss, gradients, and accuracy for the two-stream
% predictions.
rgbLoss = crossentropy(dlYPredRGB,Y);
flowLoss = crossentropy(dlYPredFlow,Y);
% Fuse the losses.
loss = mean([rgbLoss,flowLoss]);

gradientsRGB = dlgradient(rgbLoss,i3d.VideoLearnables);
gradientsFlow = dlgradient(flowLoss,i3d.OpticalFlowLearnables);

% Fuse the predictions by calculating the average of the predictions.
dlYPred = (dlYPredRGB + dlYPredFlow)/2;

% Calculate the accuracy of the predictions.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(dlYPred,[],1);

acc = gather(extractdata(sum(YTest == YPred)./numel(YTest)));

% Calculate the accuracy of the RGB and flow predictions.
[~,YTest] = max(Y,[],1);
[~,YPredRGB] = max(dlYPredRGB,[],1);
[~,YPredFlow] = max(dlYPredFlow,[],1);

accRGB = gather(extractdata(sum(YTest == YPredRGB)./numel(YTest)));
accFlow = gather(extractdata(sum(YTest == YPredFlow)./numel(YTest)));
end
```

`updateLearnables`

The `updateLearnables` function updates the provided `learnables` with gradients and other parameters using SGDM optimization function `sgdmupdate`.

```
function [learnables,velocity,learnRate] = updateLearnables(learnables,gradients,params,velocity
    % Determine the learning rate using the cosine-annealing learning rate schedule.
    learnRate = cosineAnnealingLearnRate(iteration, params);

    % Apply L2 regularization to the weights.
    idx = learnables.Parameter == "Weights";
    gradients(idx,:) = dlupdate(@(g,w) g + params.L2Regularization*w, gradients(idx,:), learnable

    % Update the network parameters using the SGDM optimizer.
    [learnables, velocity] = sgdmupdate(learnables, gradients, velocity, learnRate, params.Momen
end
```

`cosineAnnealingLearnRate`

The `cosineAnnealingLearnRate` function computes the learning rate based on the current iteration number, minimum learning rate, maximum learning rate, and number of iterations for annealing [3 on page 8-0].

```
function lr = cosineAnnealingLearnRate(iteration, params)
    if iteration == params.NumIterations
        lr = params.MinLearningRate;
        return;
    end
```

```
    cosineNumIter = [0, params.CosineNumIterations];
    csum = cumsum(cosineNumIter);
    block = find(csum >= iteration, 1,'first');
    cosineIter = iteration - csum(block - 1);
    annealingIteration = mod(cosineIter, cosineNumIter(block));
    cosineIteration = cosineNumIter(block);
    minR = params.MinLearningRate;
    maxR = params.MaxLearningRate;
    cosMult = 1 + cos(pi * annealingIteration / cosineIteration);
    lr = minR + ((maxR - minR) *  cosMult / 2);
end
```

aggregateConfusionMetric

The `aggregateConfusionMetric` function incrementally fills a confusion matrix based on the predicted results `YPred` and the expected results `YTest`.

```
function cmat = aggregateConfusionMetric(cmat,YTest,YPred)
YTest = gather(extractdata(YTest));
YPred = gather(extractdata(YPred));
[m,n] = size(cmat);
cmat = cmat + full(sparse(YTest,YPred,1,m,n));
end
```

doValidation

The `doValidation` function validates the video classifier using the validation data.

```
function [validationTime, cmat, lossValidation, accValidation, accValidationRGB, accValidationFl

validationTime = tic;

numOutputs = 3;
mbq = createMiniBatchQueue(params.ValidationData, numOutputs, params);

lossValidation = [];
numClasses = numel(params.Classes);
cmat = sparse(numClasses,numClasses);
cmatRGB = sparse(numClasses,numClasses);
cmatFlow = sparse(numClasses,numClasses);
while hasdata(mbq)

    [dlX1,dlX2,dlY] = next(mbq);

    [loss,YTest,YPred,YPredRGB,YPredFlow] = predictValidation(i3d,dlX1,dlX2,dlY);

    lossValidation = [lossValidation,loss];
    cmat = aggregateConfusionMetric(cmat,YTest,YPred);
    cmatRGB = aggregateConfusionMetric(cmatRGB,YTest,YPredRGB);
    cmatFlow = aggregateConfusionMetric(cmatFlow,YTest,YPredFlow);
end
lossValidation = mean(lossValidation);
accValidation = sum(diag(cmat))./sum(cmat,"all");
accValidationRGB = sum(diag(cmatRGB))./sum(cmatRGB,"all");
accValidationFlow = sum(diag(cmatFlow))./sum(cmatFlow,"all");

validationTime = toc(validationTime);
end
```

predictValidation

The `predictValidation` function calculates the loss and prediction values using the provided video classifier for RGB and optical flow data.

```
function [loss,YTest,YPred,YPredRGB,YPredFlow] = predictValidation(i3d,dlRGB,dlFlow,Y)

% Pass the video input through the two-stream Inflated-3D video classifier.
[dlYPredRGB,dlYPredFlow] = predict(i3d,dlRGB,dlFlow);

% Calculate the cross-entropy separately for the two-stream outputs.
rgbLoss = crossentropy(dlYPredRGB,Y);
flowLoss = crossentropy(dlYPredFlow,Y);

% Fuse the losses.
loss = mean([rgbLoss,flowLoss]);

% Fuse the predictions by calculating the average of the predictions.
dlYPred = (dlYPredRGB + dlYPredFlow)/2;

% Calculate the accuracy of the predictions.
[~,YTest] = max(Y,[],1);
[~,YPred] = max(dlYPred,[],1);

[~,YPredRGB] = max(dlYPredRGB,[],1);
[~,YPredFlow] = max(dlYPredFlow,[],1);

end
```

saveData

The `saveData` function saves the given Inflated-3d Video Classifier, accuracy, loss, and other training parameters to a MAT-file.

```
function bestLoss = saveData(inflated3d,bestLoss,iteration,cmat,lossTrain,lossValidation,...
                accTrain,accValidation,params)
if iteration >= params.SaveBestAfterIteration
    lossValidtion = extractdata(gather(lossValidation));
    if lossValidtion < bestLoss
        params = rmfield(params, 'VelocityRGB');
        params = rmfield(params, 'VelocityFlow');
        bestLoss = lossValidtion;
        inflated3d = gatherFromGPUToSave(inflated3d);
        data.BestLoss = bestLoss;
        data.TrainingLoss = extractdata(gather(lossTrain));
        data.TrainingAccuracy = accTrain;
        data.ValidationAccuracy = accValidation;
        data.ValidationConfmat= cmat;
        data.inflated3d = inflated3d;
        data.Params = params;
        save(params.ModelFilename, 'data');
    end
end
end
```

gatherFromGPUToSave

The `gatherFromGPUToSave` function gathers data from the GPU in order to save the video classifier to disk.

```
function classifier = gatherFromGPUToSave(classifier)
if ~canUseGPU
    return;
end
p = string(properties(classifier));
p = p(endsWith(p, ["Learnables","State"]));
for jj = 1:numel(p)
    prop = p(jj);
    classifier.(prop) = gatherValues(classifier.(prop));
end
    function tbl = gatherValues(tbl)
        for ii = 1:height(tbl)
            tbl.Value{ii} = gather(tbl.Value{ii});
        end
    end
end
```

checkForHMDB51Folder

The `checkForHMDB51Folder` function checks for the downloaded data in the download folder.

```
function classes = checkForHMDB51Folder(dataLoc)
hmdbFolder = fullfile(dataLoc, "hmdb51_org");
if ~isfolder(hmdbFolder)
    error("Download 'hmdb51_org.rar' file using the supporting function 'downloadHMDB51' before r
end

classes = ["brush_hair","cartwheel","catch","chew","clap","climb","climb_stairs",...
    "dive","draw_sword","dribble","drink","eat","fall_floor","fencing",...
    "flic_flac","golf","handstand","hit","hug","jump","kick","kick_ball",...
    "kiss","laugh","pick","pour","pullup","punch","push","pushup","ride_bike",...
    "ride_horse","run","shake_hands","shoot_ball","shoot_bow","shoot_gun",...
    "sit","situp","smile","smoke","somersault","stand","swing_baseball","sword",...
    "sword_exercise","talk","throw","turn","walk","wave"];
expectFolders = fullfile(hmdbFolder, classes);
if ~all(arrayfun(@(x)exist(x,'dir'),expectFolders))
    error("Download hmdb51_org.rar using the supporting function 'downloadHMDB51' before running
end
end
```

downloadHMDB51

The `downloadHMDB51` function downloads the data set and saves it to a directory.

```
function downloadHMDB51(dataLoc)

if nargin == 0
    dataLoc = pwd;
end
dataLoc = string(dataLoc);

if ~isfolder(dataLoc)
    mkdir(dataLoc);
```

```
end

dataUrl     = "http://serre-lab.clps.brown.edu/wp-content/uploads/2013/10/hmdb51_org.rar";
options     = weboptions('Timeout', Inf);
rarFileName = fullfile(dataLoc, 'hmdb51_org.rar');

% Download the RAR file and save it to the download folder.
if ~isfile(rarFileName)
    disp("Downloading hmdb51_org.rar (2 GB) to the folder:")
    disp(dataLoc)
    disp("This download can take a few minutes...")
    websave(rarFileName, dataUrl, options);
    disp("Download complete.")
    disp("Extract the hmdb51_org.rar file contents to the folder: ")
    disp(dataLoc)
end
end
```

initializeTrainingProgressPlot

The `initializeTrainingProgressPlot` function configures two plots for displaying the training loss, training accuracy, and validation accuracy.

```
function plotters = initializeTrainingProgressPlot(params)
if params.ProgressPlot
    % Plot the loss, training accuracy, and validation accuracy.
    figure

    % Loss plot
    subplot(2,1,1)
    plotters.LossPlotter = animatedline;
    xlabel("Iteration")
    ylabel("Loss")

    % Accuracy plot
    subplot(2,1,2)
    plotters.TrainAccPlotter = animatedline('Color','b');
    plotters.ValAccPlotter = animatedline('Color','g');
    legend('Training Accuracy','Validation Accuracy','Location','northwest');
    xlabel("Iteration")
    ylabel("Accuracy")
else
    plotters = [];
end
end
```

updateProgressPlot

The `updateProgressPlot` function updates the progress plot with loss and accuracy information during training.

```
function updateProgressPlot(params,plotters,epoch,iteration,start,lossTrain,accuracyTrain,accurac
if params.ProgressPlot

    % Update the training progress.
    D = duration(0,0,toc(start),"Format","hh:mm:ss");
    title(plotters.LossPlotter.Parent,"Epoch: " + epoch + ", Elapsed: " + string(D));
    addpoints(plotters.LossPlotter,iteration,double(gather(extractdata(lossTrain))));
```

```
    addpoints(plotters.TrainAccPlotter,iteration,accuracyTrain);
    addpoints(plotters.ValAccPlotter,iteration,accuracyValidation);
    drawnow
end
end
```

initializeVerboseOutput

The `initializeVerboseOutput` function displays the column headings for the table of training values, which shows the epoch, mini-batch accuracy, and other training values.

```
function initializeVerboseOutput(params)
if params.Verbose
    disp(" ")
    if canUseGPU
        disp("Training on GPU.")
    else
        disp("Training on CPU.")
    end
    p = gcp('nocreate');
    if ~isempty(p)
        disp("Training on parallel cluster '" + p.Cluster.Profile + "'. ")
    end
    disp("NumIterations:" + string(params.NumIterations));
    disp("MiniBatchSize:" + string(params.MiniBatchSize));
    disp("Classes:" + join(string(params.Classes), ","));
    disp("|=========================================================================================
    disp("| Epoch | Iteration | Time Elapsed |     Mini-Batch Accuracy    |    Validation Accurac
    disp("|       |           |  (hh:mm:ss)  |       (Avg:RGB:Flow)       |       (Avg:RGB:Flow)
    disp("|=========================================================================================
end
end
```

displayVerboseOutputEveryEpoch

The `displayVerboseOutputEveryEpoch` function displays the verbose output of the training values, such as the epoch, mini-batch accuracy, validation accuracy, and mini-batch loss.

```
function displayVerboseOutputEveryEpoch(params,start,learnRate,epoch,iteration,...
    accTrain,accTrainRGB,accTrainFlow,accValidation,accValidationRGB,accValidationFlow,lossTrain
if params.Verbose
    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    trainTime = duration(0,0,trainTime,'Format','hh:mm:ss');
    validationTime = duration(0,0,validationTime,'Format','hh:mm:ss');

    lossValidation = gather(extractdata(lossValidation));
    lossValidation = compose('%.4f',lossValidation);

    accValidation = composePadAccuracy(accValidation);
    accValidationRGB = composePadAccuracy(accValidationRGB);
    accValidationFlow = composePadAccuracy(accValidationFlow);

    accVal = join([accValidation,accValidationRGB,accValidationFlow], " : ");

    lossTrain = gather(extractdata(lossTrain));
    lossTrain = compose('%.4f',lossTrain);

    accTrain = composePadAccuracy(accTrain);
```

```
    accTrainRGB = composePadAccuracy(accTrainRGB);
    accTrainFlow = composePadAccuracy(accTrainFlow);

    accTrain = join([accTrain,accTrainRGB,accTrainFlow], " : ");
    learnRate = compose('%.13f',learnRate);

    disp("| " + ...
        pad(string(epoch),5,'both') + " | " + ...
        pad(string(iteration),9,'both') + " | " + ...
        pad(string(D),12,'both') + " | " + ...
        pad(string(accTrain),26,'both') + " | " + ...
        pad(string(accVal),26,'both') + " | " + ...
        pad(string(lossTrain),10,'both') + " | " + ...
        pad(string(lossValidation),10,'both') + " | " + ...
        pad(string(learnRate),13,'both') + " | " + ...
        pad(string(trainTime),10,'both') + " | " + ...
        pad(string(validationTime),15,'both') + " |")
end

    function acc = composePadAccuracy(acc)
        acc = compose('%.2f',acc*100) + "%";
        acc = pad(string(acc),6,'left');
    end

end
```

`endVerboseOutput`

The `endVerboseOutput` function displays the end of verbose output during training.

```
function endVerboseOutput(params)
if params.Verbose
    disp("|========================================================================================
end
end
```

References

[1] Carreira, Joao, and Andrew Zisserman. "Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR): 6299??6308. Honolulu, HI: IEEE, 2017.

[2] Simonyan, Karen, and Andrew Zisserman. "Two-Stream Convolutional Networks for Action Recognition in Videos." *Advances in Neural Information Processing Systems* 27, Long Beach, CA: NIPS, 2017.

[3] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *International Conferencee on Learning Representations 2017*. Toulon, France: ICLR, 2017.

[4] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, Manohar Paluri. "A Closer Look at Spatiotemporal Convolutions for Action Recognition". Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 6450-6459.

[5] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. "SlowFast Networks for Video Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), 2019.

[6] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, Andrew Zisserman. "The Kinetics Human Action Video Dataset." *arXiv preprint arXiv:1705.06950*, 2017.

Import Pretrained ONNX YOLO v2 Object Detector

This example shows how to import a pretrained ONNX™ (Open Neural Network Exchange) you only look once (YOLO) v2 [1] on page 8-0 object detection network and use it to detect objects. After you import the network, you can deploy it to embedded platforms using GPU Coder™ or retrain it on custom data using transfer learning with `trainYOLOv2ObjectDetector`.

Download ONNX YOLO v2 Network

Download files related to the pretrained Tiny YOLO v2 network.

```
pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/deeplearning/models/yolov2/tiny_yo
pretrainedNetTar = 'yolov2Tiny.tar';
if ~exist(pretrainedNetTar,'file')
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetTar,pretrainedURL);
end
```

```
Downloading pretrained network (58 MB)...
```

Extract YOLO v2 Network

Untar the downloaded file to extract the Tiny YOLO v2 network. Load the `'Model.onnx'` model from `tiny_yolov2` folder, which is an ONNX YOLO v2 network pretrained on the PASCAL VOC data set. The network can detect objects from 20 different classes [4] on page 8-0 .

```
onnxfiles = untar(pretrainedNetTar);
pretrainedNet = 'tiny_yolov2/Model.onnx';
```

Import ONNX YOLO v2 Layers

Use the `importONNXLayers` function to import the downloaded network.

```
lgraph = importONNXLayers(pretrainedNet,'ImportWeights',true);
```

```
Warning: Imported layers have no output layer because ONNX files do not specify the network's ou
```

In this example you add an output layer to the imported layers, so you can ignore this warning. The Add YOLO v2 Transform and Output Layers on page 8-0 section shows how to add YOLO v2 output layer along with YOLO v2 Transform layer to the imported layers.

The network in this example contains no unsupported layers. Note that if the network you want to import has unsupported layers, the function imports them as placeholder layers. Before you can use your imported network, you must replace these layers. For more information on replacing placeholder layers, see `findPlaceholderLayers`.

Define YOLO v2 Anchor Boxes

YOLO v2 uses predefined anchor boxes to predict object location. The anchor boxes used in the imported network are defined in the Tiny YOLO v2 network configuration file [5] on page 8-0 . The ONNX anchors are defined with respect to the output size of the final convolution layer, which is 13-by-13. To use the anchors with `yolov2ObjectDetector`, resize the anchor boxes to the network input size, which is 416-by-416. The anchor boxes for `yolov2ObjectDetector` must be specified in the form [height, width].

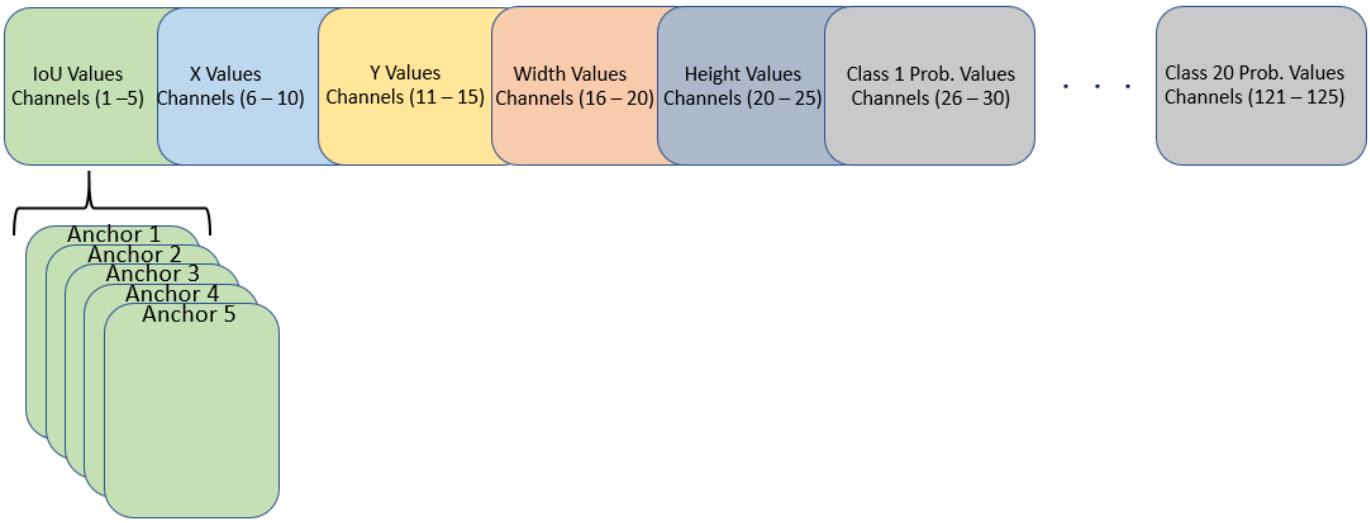
```
onnxAnchors = [1.08,1.19; 3.42,4.41; 6.63,11.38; 9.42,5.11; 16.62,10.52];
```

```
inputSize = lgraph.Layers(1,1).InputSize(1:2);
lastActivationSize = [13,13];
upScaleFactor = inputSize./lastActivationSize;
anchorBoxesTmp = upScaleFactor.* onnxAnchors;
anchorBoxes = [anchorBoxesTmp(:,2),anchorBoxesTmp(:,1)];
```

Reorder Detection Layer Weights

For efficient processing, you must reorder the weights and biases of the last convolution layer in the imported network to obtain the activations in the arrangement that `yolov2ObjectDetector` requires. `yolov2ObjectDetector` expects the 125 channels of the feature map of the last convolution layer in the following arrangement:

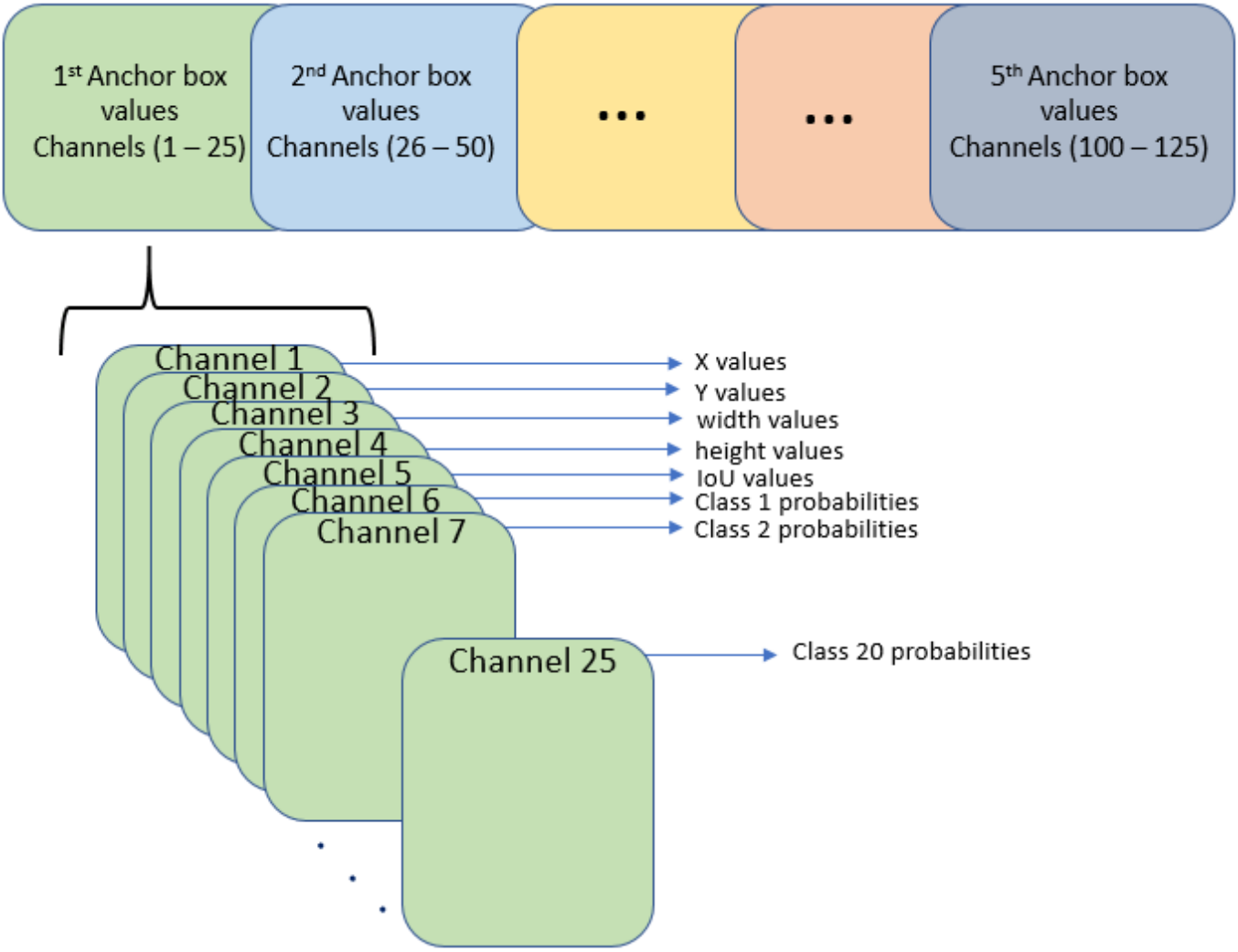
- Channels 1 to 5 - IoU values for five anchors
- Channels 6 to 10 - X values for five anchors
- Channels 11 to 15 - Y values for five anchors
- Channels 16 to 20 - Width values for five anchors
- Channels 21 to 25 - Height values for five anchors
- Channels 26 to 30 - Class 1 probability values for five anchors
- Channels 31 to 35 - Class 2 probability values for five anchors
- Channels 121 to 125 - Class 20 probability values for five anchors



However, in the last convolution layer, which is of size 13-by-13, the activations are arranged differently. Each of the 25 channels in the feature map corresponds to:

- Channel 1 - X values
- Channel 2 - Y values
- Channel 3 - Width values
- Channel 4 - Height values
- Channel 5 - IoU values
- Channel 6 - Class 1 probability values

- Channel 7 - Class 2 probability values
- Channel 25 - Class 20 probability values



Use the supporting function `rearrangeONNXWeights`, listed at the end of this example, to reorder the weights and biases of the last convolution layer in the imported network and obtain the activations in the format required by `yolov2ObjectDetector`.

```
weights = lgraph.Layers(end,1).Weights;
bias = lgraph.Layers(end,1).Bias;
layerName = lgraph.Layers(end,1).Name;

numAnchorBoxes = size(onnxAnchors,1);
[modWeights,modBias] = rearrangeONNXWeights(weights,bias,numAnchorBoxes);
```

Replace the weights and biases of the last convolution layer in the imported network with the new convolution layer using the reordered weights and biases.

```
filterSize = size(modWeights,[1 2]);
numFilters = size(modWeights,4);
```

```
modConvolution8 = convolution2dLayer(filterSize,numFilters,...
    'Name',layerName,'Bias',modBias,'Weights',modWeights);
lgraph = replaceLayer(lgraph,'convolution8',modConvolution8);
```

Add YOLO v2 Transform and Output Layers

A YOLO v2 detection network requires the YOLO v2 transform and YOLO v2 output layers. Create both of these layers, stack them in series, and attach the YOLO v2 transform layer to the last convolution layer.

```
classNames = tinyYOLOv2Classes;

layersToAdd = [
    yolov2TransformLayer(numAnchorBoxes,'Name','yolov2Transform');
    yolov2OutputLayer(anchorBoxes,'Classes',classNames,'Name','yolov2Output');
    ];

lgraph = addLayers(lgraph, layersToAdd);
lgraph = connectLayers(lgraph,layerName,'yolov2Transform');
```

The `ElementwiseAffineLayer` in the imported network duplicates the preprocessing step performed by `yolov2ObjectDetector`. Hence, remove the `ElementwiseAffineLayer` from the imported network.

```
yoloScaleLayerIdx = find(...
    arrayfun( @(x)isa(x,'nnet.onnx.layer.ElementwiseAffineLayer'), ...
    lgraph.Layers));

if ~isempty(yoloScaleLayerIdx)
    for i = 1:size(yoloScaleLayerIdx,1)
        layerNames {i} = lgraph.Layers(yoloScaleLayerIdx(i,1),1).Name;
    end
    lgraph = removeLayers(lgraph,layerNames);
    lgraph = connectLayers(lgraph,'Input_image','convolution');
end
```

Create YOLO v2 Object Detector

Assemble the layer graph using the `assembleNetwork` function and create a YOLO v2 object detector using the `yolov2ObjectDetector` function.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'Input_image'}
    OutputNames: {'yolov2Output'}
```

```
yolov2Detector = yolov2ObjectDetector(net)
```

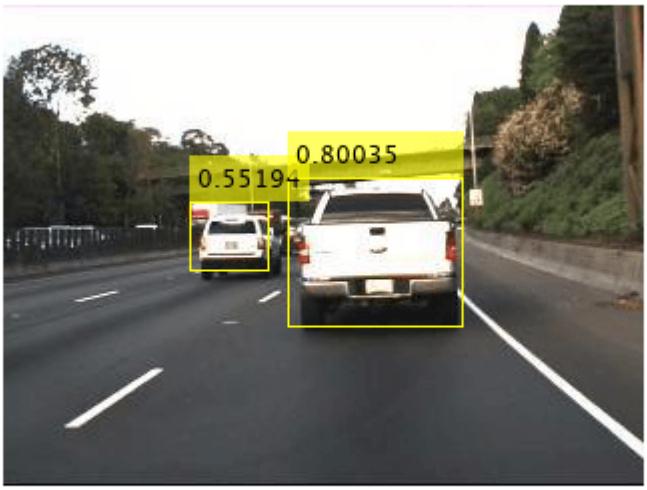
```
yolov2Detector =
  yolov2ObjectDetector with properties:

            ModelName: 'importedNetwork'
```

```
           Network: [1×1 DAGNetwork]
    TrainingImageSize: [416 416]
         AnchorBoxes: [5×2 double]
          ClassNames: [aeroplane    bicycle    bird    boat    bottle    bus    car    cat    cl
```

Detect Objects Using Imported YOLO v2 Detector

Use the imported detector to detect objects in a test image. Display the results.

```
I = imread('highway.png');
% Convert image to BGR format.
Ibgr = cat(3,I(:,:,3),I(:,:,2),I(:,:,1));
[bboxes, scores, labels] = detect(yolov2Detector, Ibgr);
detectedImg = insertObjectAnnotation(I, 'rectangle', bboxes, scores);
figure
imshow(detectedImg);
```



Supporting Functions

```
function [modWeights,modBias] = rearrangeONNXWeights(weights,bias,numAnchorBoxes)
%rearrangeONNXWeights rearranges the weights and biases of an imported YOLO
%v2 network as required by yolov2ObjectDetector. numAnchorBoxes is a scalar
%value containing the number of anchors that are used to reorder the weights and
%biases. This function performs the following operations:
%   * Extract the weights and biases related to IoU, boxes, and classes.
%   * Reorder the extracted weights and biases as expected by yolov2ObjectDetector.
%   * Combine and reshape them back to the original dimensions.

weightsSize = size(weights);
biasSize = size(bias);
sizeOfPredictions = biasSize(3)/numAnchorBoxes;

% Reshape the weights with regard to the size of the predictions and anchors.
```

```
reshapedWeights = reshape(weights,prod(weightsSize(1:3)),sizeOfPredictions,numAnchorBoxes);

% Extract the weights related to IoU, boxes, and classes.
weightsIou = reshapedWeights(:,5,:);
weightsBoxes = reshapedWeights(:,1:4,:);
weightsClasses = reshapedWeights(:,6:end,:);

% Combine the weights of the extracted parameters as required by
% yolov2ObjectDetector.
reorderedWeights = cat(2,weightsIou,weightsBoxes,weightsClasses);
permutedWeights = permute(reorderedWeights,[1 3 2]);

% Reshape the new weights to the original size.
modWeights = reshape(permutedWeights,weightsSize);

% Reshape the biases with regared to the size of the predictions and anchors.
reshapedBias = reshape(bias,sizeOfPredictions,numAnchorBoxes);

% Extract the biases related to IoU, boxes, and classes.
biasIou = reshapedBias(5,:);
biasBoxes = reshapedBias(1:4,:);
biasClasses = reshapedBias(6:end,:);

% Combine the biases of the extracted parameters as required by yolov2ObjectDetector.
reorderedBias = cat(1,biasIou,biasBoxes,biasClasses);
permutedBias = permute(reorderedBias,[2 1]);

% Reshape the new biases to the original size.
modBias = reshape(permutedBias,biasSize);
end


function classes = tinyYOLOv2Classes()
% Return the class names corresponding to the pretrained ONNX tiny YOLO v2
% network.
%
% The tiny YOLO v2 network is pretrained on the Pascal VOC data set,
% which contains images from 20 different classes [4].

classes = [ ...
    " aeroplane", "bicycle", "bird", "boat", "bottle", "bus", "car",...
    "cat", "chair", "cow", "diningtable", "dog", "horse", "motorbike",...
    "person", "pottedplant", "sheep", "sofa", "train", "tvmonitor"];
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLO9000: Better, Faster, Stronger.” In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 6517–25. Honolulu, HI: IEEE, 2017. https://doi.org/10.1109/CVPR.2017.690.

[2] "Tiny YOLO v2 Model." https://github.com/onnx/models/tree/master/vision/object_detection_segmentation/tiny-yolov2

[3] "Tiny YOLO v2 Model License." https://github.com/onnx/onnx/blob/master/LICENSE.

[4] Everingham, Mark, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. “The Pascal Visual Object Classes (VOC) Challenge.” International Journal of Computer Vision 88, no. 2 (June 2010): 303–38. https://doi.org/10.1007/s11263-009-0275-4.

[5] "yolov2-tiny-voc.cfg" https://github.com/pjreddie/darknet/blob/master/cfg/yolov2-tiny-voc.cfg.

See Also

Functions
`importONNXNetwork` | `assembleNetwork` | `convolution2dLayer` | `replaceLayer` | `removeLayers` | `connectLayers` | `findPlaceholderLayers` | `detect` | `trainYOLOv2ObjectDetector` | `addLayers`

Objects
`yolov2ObjectDetector`

More About

- “Export to and Import from ONNX” on page 1-15

Export YOLO v2 Object Detector to ONNX

This example shows how to export a YOLO v2 object detection network to ONNX™ (Open Neural Network Exchange) model format. After exporting the YOLO v2 network, you can import the network into other deep learning frameworks for inference. This example also presents the workflow that you can follow to perform inference using the imported ONNX model.

Export YOLO v2 Network

Export the detection network to ONNX and gather the metadata required to generate object detection results.

First, load a pretrained YOLO v2 object detector into the workspace.

```
input = load('yolov2VehicleDetector.mat');
net = input.detector.Network;
```

Next, obtain the YOLO v2 detector metadata to use for inference. The detector metadata includes the network input image size, anchor boxes, and activation size of last convolution layer.

Read the network input image size from the input YOLO v2 network.

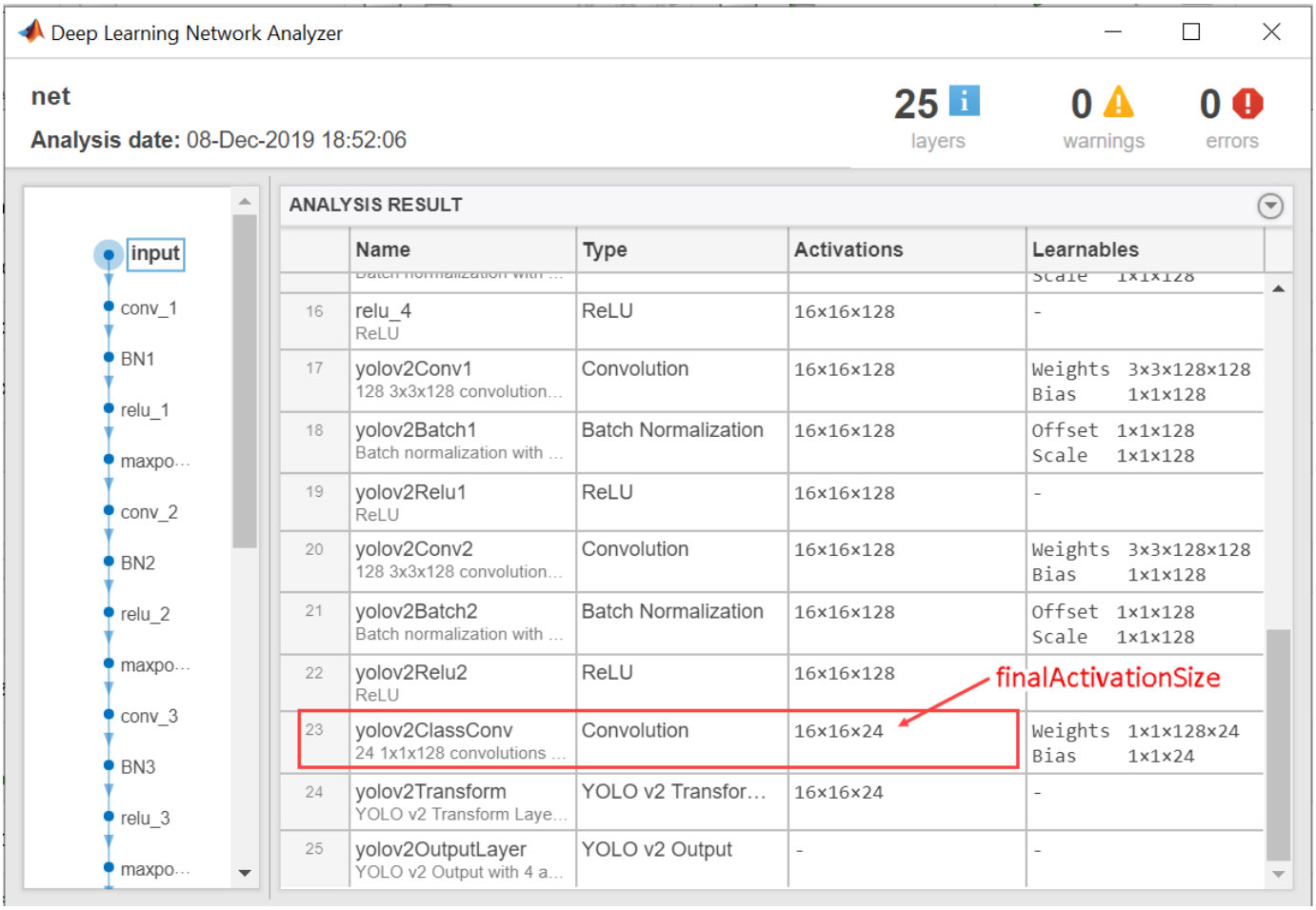
```
inputImageSize = net.Layers(1,1).InputSize;
```

Read the anchor boxes used for training from the input detector.

```
anchorBoxes = input.detector.AnchorBoxes;
```

Get the activation size of the last convolution layer in the input network by using the `analyzeNetwork` function.

```
analyzeNetwork(net);
```




```
finalActivationSize = [16 16 24];
```

Export to ONNX Model Format

Export the YOLO v2 object detection network as an ONNX format file by using the `exportONNXNetwork` function. Specify the file name as `yolov2.onnx`. The function saves the exported ONNX file to the current working folder.

```
filename = 'yolov2.onnx';
exportONNXNetwork(net,filename);
```

The `exportONNXNetwork` function maps the `yolov2TransformLayer` (Computer Vision Toolbox) and `yolov2OutputLayer` (Computer Vision Toolbox) in the input YOLO v2 network to the basic ONNX operator and identity operator, respectively. After you export the network, you can import the `yolov2.onnx` file into any deep learning framework that supports ONNX import.

Using the `exportONNXNetwork`, requires Deep Learning Toolbox™ and the Deep Learning Toolbox Converter for ONNX Model Format support package. If this support package is not installed, then the function provides a download link.

Object Detection Using Exported YOLO v2 Network

When exporting is complete, you can import the ONNX model into any deep learning framework and use the following workflow to perform object detection. Along with the ONNX network, this workflow

also requires the YOLO v2 detector metadata `inputImageSize`, `anchorBoxes`, and `finalActivationSize` obtained from the MATLAB workspace. The following code is a MATLAB implementation of the workflow that you must translate into the equivalent code for the framework of your choice.

Preprocess Input Image

Preprocess the image to use for inference. The image must be an RGB image and must be resized to the network input image size, and its pixel values must lie in the interval [0 1].

```
I = imread('highway.png');
resizedI = imresize(I,inputImageSize(1:2));
rescaledI = rescale(resizedI);
```

Pass Input and Run ONNX Model

Run the ONNX model in the deep learning framework of your choice with the preprocessed image as input to the imported ONNX model.

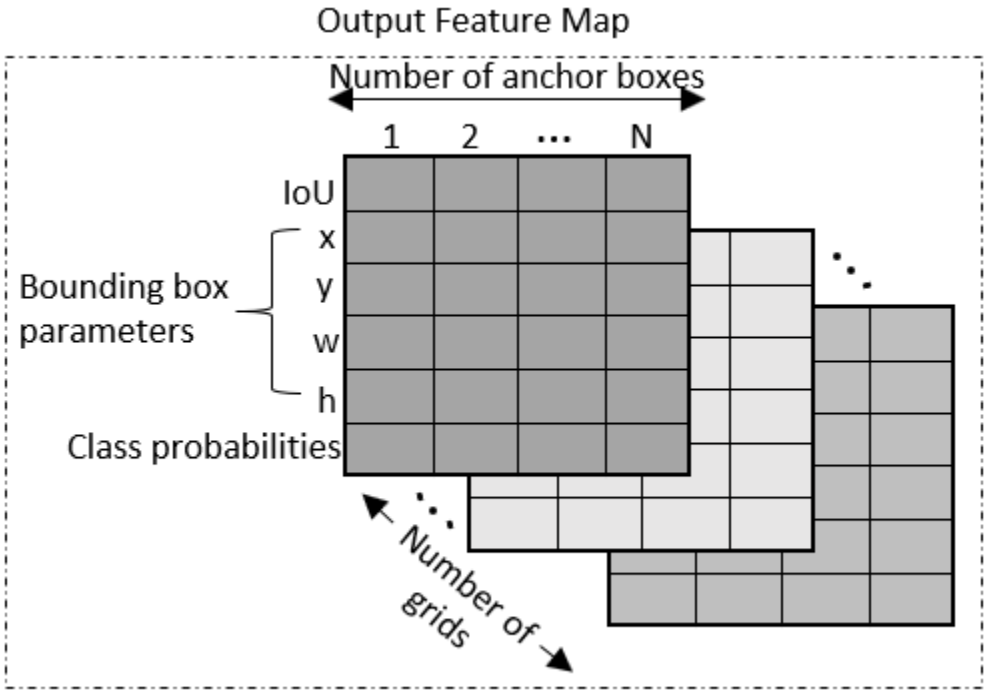
Extract Predictions from Output of ONNX Model

The model predicts the following:

- Intersection over union (IoU) with ground truth boxes
- `x`, `y`, `w`, and `h` bounding box parameters for each anchor box
- Class probabilities for each anchor box

The output of the ONNX model is a feature map that contains the predictions and is of size `predictionsPerAnchor`-by-`numAnchors`-by-`numGrids`.

- `numAnchors` is the number of anchor boxes.
- `numGrids` is the number of grids calculated as the product of the height and width of the last convolution layer.
- `predictionsPerAnchor` is the output predictions in the form [`IoU;x;y;w;h;class probabilities`].



- The first row in the feature map contains IoU predictions for each anchor box.
- The second and third rows in the feature map contain predictions for the centroid coordinates (x,y) of each anchor box.
- The fourth and fifth rows in the feature map contain the predictions for the width and height of each anchor box.
- The sixth row in the feature map contains the predictions for class probabilities of each anchor box.

Compute Final Detections

To compute final detections for the preprocessed test image, you must:

- Rescale the bounding box parameters with respect to the size of the input layer of the network.
- Compute object confidence scores from the predictions.
- Obtain predictions with high object confidence scores.
- Perform nonmaximum suppression.

As an implementation guide, use the code for `yolov2PostProcess` on page 8-0 function in Postprocessing Functions on page 8-0 .

```
[bboxes,scores,labels] = yolov2PostProcess(featureMap,inputImageSize,finalActivationsSize,anchorB
```

Display Detection Results

```
Idisp = insertObjectAnnotation(resizedI,'rectangle',bboxes,scores);
figure
imshow(Idisp)
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLO9000: Better, Faster, Stronger.” In 2017 *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517–25. Honolulu, HI: IEEE, 2017. https://doi.org/10.1109/CVPR.2017.690.

Postprocessing Functions

```
function  [bboxes,scores,labels] = yolov2PostProcess(featureMap,inputImageSize,finalActivationsSi

% Extract prediction values from the feature map.
iouPred = featureMap(1,:,:);
xyPred = featureMap(2:3,:,:);
whPred = featureMap(4:5,:,:);
probPred = featureMap(6,:,:);

% Rescale the bounding box parameters.
bBoxes = rescaleBbox(xyPred,whPred,anchorBoxes,finalActivationsSize,inputImageSize);

% Rearrange the feature map as a two-dimensional matrix for efficient processing.
predVal = [bBoxes;iouPred;probPred];
predVal = reshape(predVal,size(predVal,1),[]);

% Compute object confidence scores from the rearranged prediction values.
[confScore,idx] = computeObjectScore(predVal);

% Obtain predictions with high object confidence scores.
[bboxPred,scorePred,classPred] = selectMaximumPredictions(confScore,idx,predVal);

% To get the final detections, perform nonmaximum suppression with an overlap threshold of 0.5.
[bboxes,scores,labels] = selectStrongestBboxMulticlass(bboxPred', scorePred', classPred','RatioTy

end

function bBoxes = rescaleBbox(xyPred,whPred,anchorBoxes,finalActivationsSize,inputImageSize)

% To rescale the bounding box parameters, compute the scaling factor by using the network paramet
scaleY = inputImageSize(1)/finalActivationsSize(1);
scaleX = inputImageSize(2)/finalActivationsSize(2);
scaleFactor = [scaleY scaleX];

bBoxes = zeros(size(xyPred,1)+size(whPred,1),size(anchors,1),size(xyPred,3),'like',xyPred);
for rowIdx=0:finalActivationsSize(1,1)-1
    for colIdx=0:finalActivationsSize(1,2)-1
        ind = rowIdx*finalActivationsSize(1,2)+colIdx+1;
        for anchorIdx = 1 : size(anchorBoxes,1)

            % Compute the center with respect to image.
            cx = (xyPred(1,anchorIdx,ind)+colIdx)* scaleFactor(1,2);
            cy = (xyPred(2,anchorIdx,ind)+rowIdx)* scaleFactor(1,1);

            % Compute the width and height with respect to the image.
            bw = whPred(1,anchorIdx,ind)* anchorBoxes(anchorIdx,1);
            bh = whPred(2,anchorIdx,ind)* anchorBoxes(anchorIdx,2);

            bBoxes(1,anchorIdx,ind) = (cx-bw/2);
            bBoxes(2,anchorIdx,ind) = (cy-bh/2);
            bBoxes(3,anchorIdx,ind) = bw;
```

```
            bBoxes(4,anchorIdx,ind) = bh;
        end
    end
end
end

function [confScore,idx] = computeObjectScore(predVal)
iouPred = predVal(5,:);
probPred = predVal(6:end,:);
[imax,idx] = max(probPred,[],1);
confScore = iouPred.*imax;
end

function [bboxPred,scorePred,classPred] = selectMaximumPredictions(confScore,idx,predVal)
% Specify the threshold for confidence scores.
confScoreId = confScore >= 0.5;
% Obtain the confidence scores greater than or equal to 0.5.
scorePred = confScore(:,confScoreId);
% Obtain the class IDs for predictions with confidence scores greater than
% or equal to 0.5.
classPred = idx(:,confScoreId);
% Obtain the bounding box parameters for predictions with confidence scores
% greater than or equal to 0.5.
bboxesXYWH = predVal(1:4,:);
bboxPred = bboxesXYWH(:,confScoreId);
end
```

See Also

Functions
`exportONNXNetwork` | `analyzeNetwork`

More About

- “Export to and Import from ONNX” on page 1-15

Object Detection Using SSD Deep Learning

This example shows how to train a Single Shot Detector (SSD).

Overview

Deep learning is a powerful machine learning technique that automatically learns image features required for detection tasks. There are several techniques for object detection using deep learning such as Faster R-CNN, You Only Look Once (YOLO v2), and SSD. This example trains an SSD vehicle detector using the `trainSSDObjectDetector` function. For more information, see “Object Detection” (Computer Vision Toolbox).

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('ssdResNet50VehicleExample_20a.mat','file')
    disp('Downloading pretrained detector (44 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/ssdResNet50VehicleExample
    websave('ssdResNet50VehicleExample_20a.mat',pretrainedURL);
end
```

```
Downloading pretrained detector (44 MB)...
```

Load Dataset

This example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the SSD training procedure, but in practice, more labeled images are needed to train a robust detector.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The training data is stored in a table. The first column contains the path to the image files. The remaining columns contain the ROI labels for vehicles. Display the first few rows of the data.

```
vehicleDataset(1:4,:)
```

```
ans=4×2 table
              imageFilename                    vehicle
    ___________________________________    _______________

    {'vehicleImages/image_00001.jpg'}    {[220 136 35 28]}
    {'vehicleImages/image_00002.jpg'}    {[175 126 61 45]}
    {'vehicleImages/image_00003.jpg'}    {[108 120 45 33]}
    {'vehicleImages/image_00004.jpg'}    {[124 112 38 36]}
```

Split the data set into a training set for training the detector and a test set for evaluating the detector. Select 60% of the data for training. Use the rest for evaluation.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
```

```
idx = floor(0.6 * length(shuffledIndices) );
trainingData = vehicleDataset(shuffledIndices(1:idx),:);
testData = vehicleDataset(shuffledIndices(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDatastore` to load the image and label data during training and evaluation.

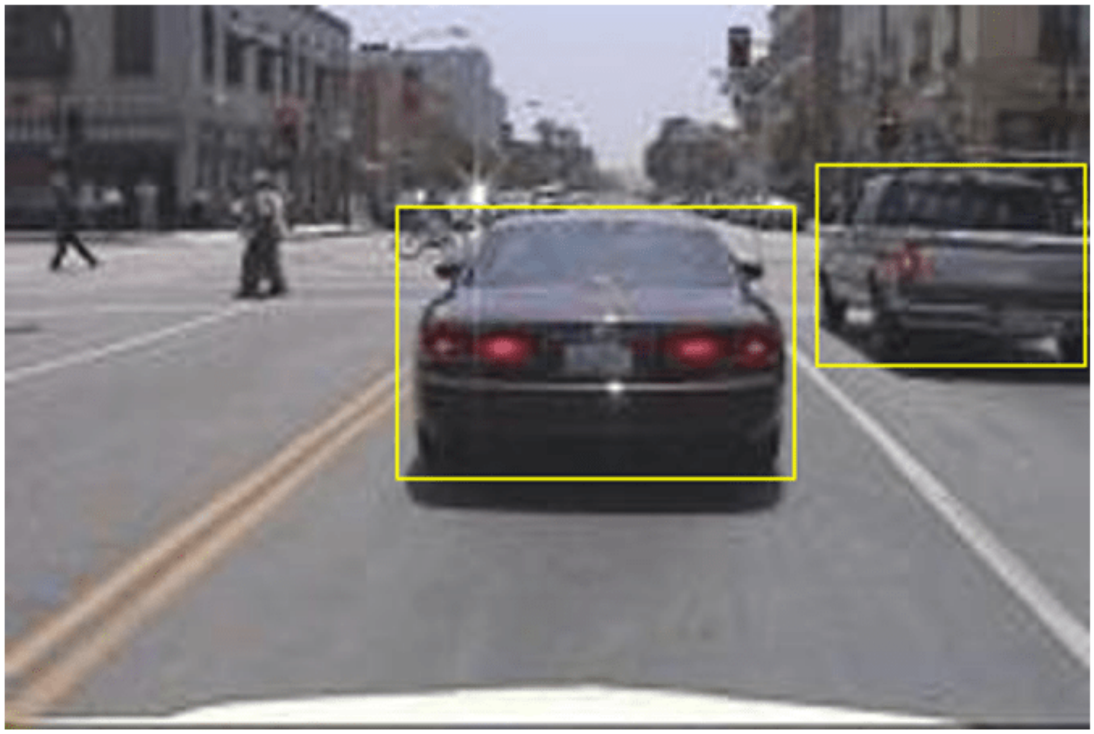
```
imdsTrain = imageDatastore(trainingData{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingData(:,'vehicle'));

imdsTest = imageDatastore(testData{:,'imageFilename'});
bldsTest = boxLabelDatastore(testData(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest, bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Create a SSD Object Detection Network

The SSD object detection network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

The feature extraction network is typically a pretrained CNN (see “Pretrained Deep Neural Networks” on page 1-8 for more details). This example uses ResNet-50 for feature extraction. Other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Use the `ssdLayers` function to automatically modify a pretrained ResNet-50 network into a SSD object detection network. `ssdLayers` requires you to specify several inputs that parameterize the SSD network, including the network input size and the number of classes. When choosing the network input size, consider the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image. However, to reduce the computational cost of running this example, the network input size is chosen to be [300 300 3]. During training, `trainSSDObjectDetector` automatically resizes the training images to the network input size.

```
inputSize = [300 300 3];
```

Define number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the SSD object detection network.

```
lgraph = ssdLayers(inputSize, numClasses, 'resnet50');
```

You can visualize the network using `analyzeNetwork` or `DeepNetworkDesigner` from Deep Learning Toolbox™. Note that you can also create a custom SSD network layer-by-layer. For more information, see “Create SSD Object Detection Network” (Computer Vision Toolbox).

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples. Use `transform` to augment the training data by

- Randomly flipping the image and associated box labels horizontally.
- Randomly scale the image, associated box labels.
- Jitter image color.

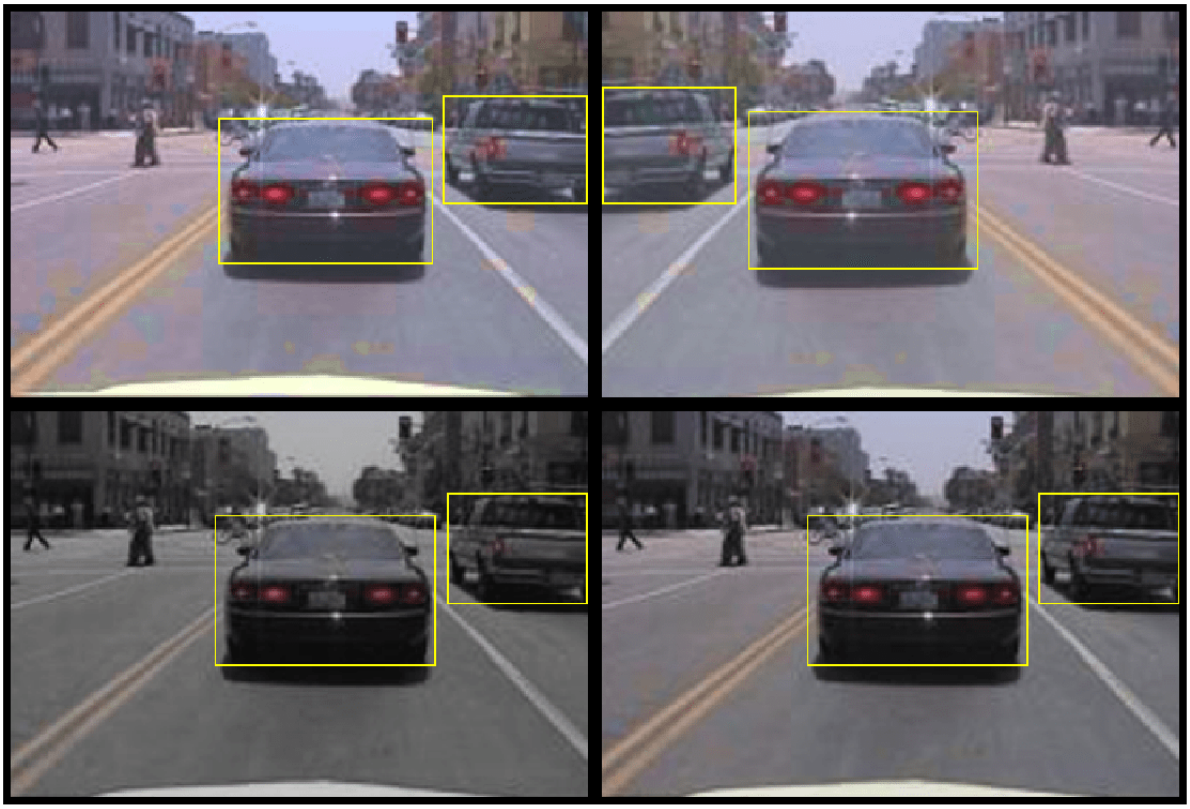
Note that data augmentation is not applied to the test data. Ideally, test data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Visualize augmented training data by reading the same image multiple times.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
```

```
    reset(augmentedTrainingData);
end

figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data to prepare for training.

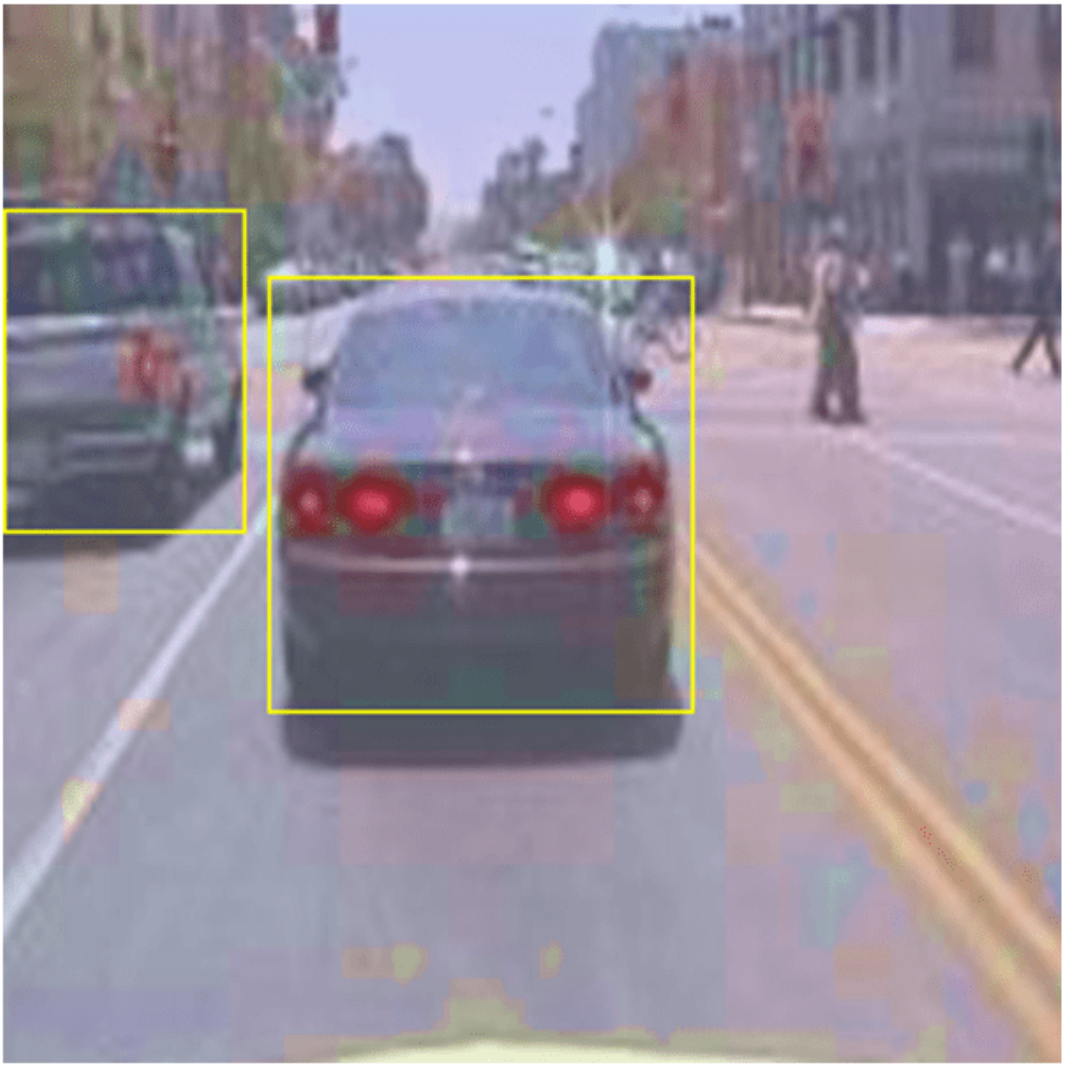
```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Train SSD Object Detector

Use `trainingOptions` to specify network training options. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
        'MiniBatchSize', 16, ....
        'InitialLearnRate',1e-1, ...
        'LearnRateSchedule', 'piecewise', ...
        'LearnRateDropPeriod', 30, ...
        'LearnRateDropFactor', 0.8, ...
        'MaxEpochs', 300, ...
```

```
        'VerboseFrequency', 50, ...
        'CheckpointPath', tempdir, ...
        'Shuffle','every-epoch');
```

Use `trainSSDObjectDetector` (Computer Vision Toolbox) function to train SSD object detector if `doTraining` to true. Otherwise, load a pretrained network.

```
if doTraining
    % Train the SSD detector.
    [detector, info] = trainSSDObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load('ssdResNet50VehicleExample_20a.mat');
    detector = pretrained.detector;
end
```

This example is verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the '`MiniBatchSize`' using the `trainingOptions` function. Training this network took approximately 2 hours using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on one test image.

```
data = read(testData);
I = data{1,1};
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I, 'Threshold', 0.4);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (`precision`) and the ability of the detector to find all relevant objects (`recall`).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

```
detectionResults = detect(detector, preprocessedTestData, 'Threshold', 0.4);
```

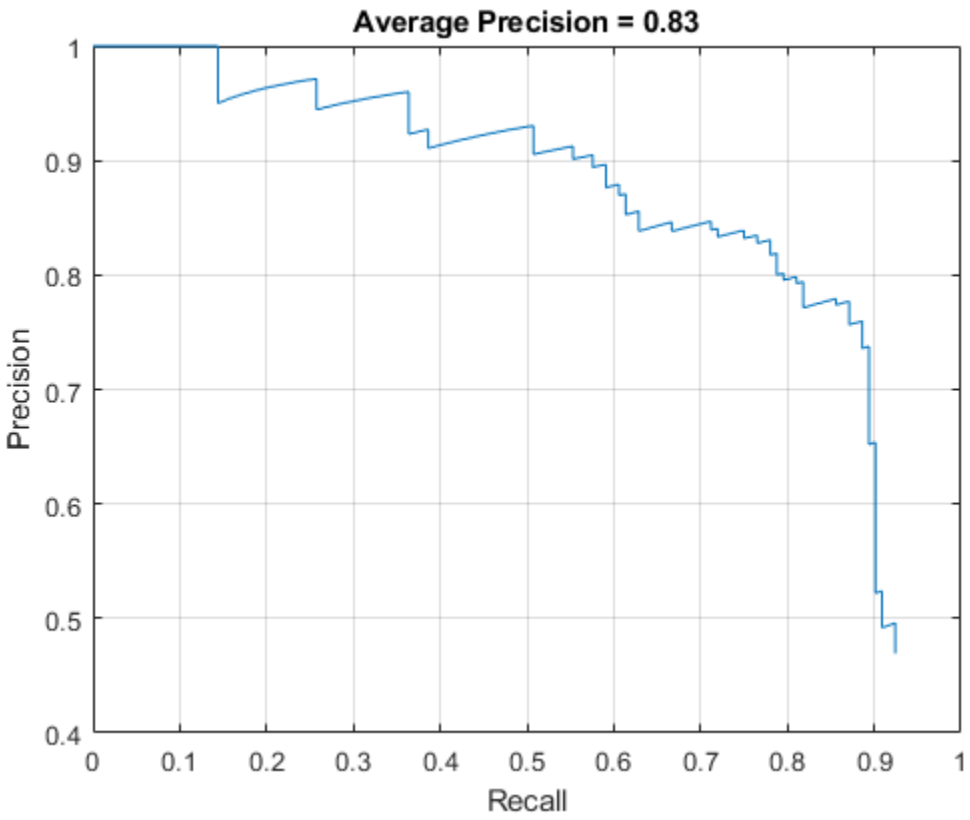
Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. Ideally, the precision would be 1 at all recall levels. The use of more data can help improve the average precision, but might require more training time Plot the PR curve.

```
figure
plot(recall,precision)
```

```
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `ssdObjectDetector` using GPU Coder™. For more details, see "Code Generation for Object Detection by Using Single Shot Multibox Detector" (Computer Vision Toolbox) example.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.
B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end
```

```
% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
B{1} = imwarp(I,tform,'OutputView',rout);

% Sanitize boxes, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize boxes, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.

See Also

Apps
Deep Network Designer

Functions
`trainSSDObjectDetector` | `analyzeNetwork` | `combine` | `transform` | `evaluateDetectionPrecision` | `ssdLayers` | `trainingOptions` | `detect` | `read`

Objects
`ssdObjectDetector` | `boxLabelDatastore` | `imageDatastore`

More About

- “Code Generation for Object Detection by Using Single Shot Multibox Detector” (Computer Vision Toolbox)
- “Create SSD Object Detection Network” (Computer Vision Toolbox)

- “Getting Started with SSD Multibox Detection” (Computer Vision Toolbox)

Object Detection Using YOLO v3 Deep Learning

This example shows how to train a YOLO v3 on page 8-0 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN, you only look once (YOLO) v2, and single shot detector (SSD). This example shows how to train a YOLO v3 object detector. YOLO v3 improves upon YOLO v2 by adding detection at multiple scales to help detect smaller objects. The loss function used for training is separated into mean squared error for bounding box regression and binary cross-entropy for object classification to help improve detection accuracy.

Note: This example requires the Computer Vision Toolbox™ Model for YOLO v3 Object Detection. You can install the Computer Vision Toolbox Model for YOLO v3 Object Detection from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Download Pretrained Network

Download a pretrained network using the helper function `downloadPretrainedYOLOv3Detector` to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
doTraining = false;

if ~doTraining
    preTrainedDetector = downloadPretrainedYOLOv3Detector();
end
```

Load Data

This example uses a small labeled data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the YOLO v3 training procedure, but in practice, more labeled images are needed to train a robust network.

Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

% Add the full path to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd, vehicleDataset.imageFilename);
```

Note: In case of multiple classes, the data can also organized as three columns where the first column contains the image file names with paths, the second column contains the bounding boxes and the third column must be a cell vector that contains the label names corresponding to each bounding box. For more information on how to arrange the bounding boxes and labels, see `boxLabelDatastore` (Computer Vision Toolbox).

All the bounding boxes must be in the form `[x y width height]`. This vector specifies the upper left corner and the size of the bounding box in pixels.

Split the data set into a training set for training the network, and a test set for evaluating the network. Use 60% of the data for training set and the rest for the test set.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices));
trainingDataTbl = vehicleDataset(shuffledIndices(1:idx), :);
testDataTbl = vehicleDataset(shuffledIndices(idx+1:end), :);
```

Create an image datastore for loading the images.

```
imdsTrain = imageDatastore(trainingDataTbl.imageFilename);
imdsTest = imageDatastore(testDataTbl.imageFilename);
```

Create a datastore for the ground truth bounding boxes.

```
bldsTrain = boxLabelDatastore(trainingDataTbl(:, 2:end));
bldsTest = boxLabelDatastore(testDataTbl(:, 2:end));
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain, bldsTrain);
testData = combine(imdsTest, bldsTest);
```

Use `validateInputData` to detect invalid images, bounding boxes or labels i.e.,

- Samples with invalid image format or containing NaNs
- Bounding boxes containing zeros/NaNs/Infs/empty
- Missing/non-categorical labels.

The values of the bounding boxes should be finite, positive, non-fractional, non-NaN and should be within the image boundary with a positive height and width. Any invalid samples must either be discarded or fixed for proper training.

```
validateInputData(trainingData);
validateInputData(testData);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` function to apply custom data augmentations to the training data. The `augmentData` helper function, listed at the end of the example, applies the following augmentations to the input data.

- Color jitter augmentation in HSV space
- Random horizontal flip
- Random scaling by 10 percent

```
augmentedTrainingData = transform(trainingData, @augmentData);
```

Read the same image four times and display the augmented training data.

```
% Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
```

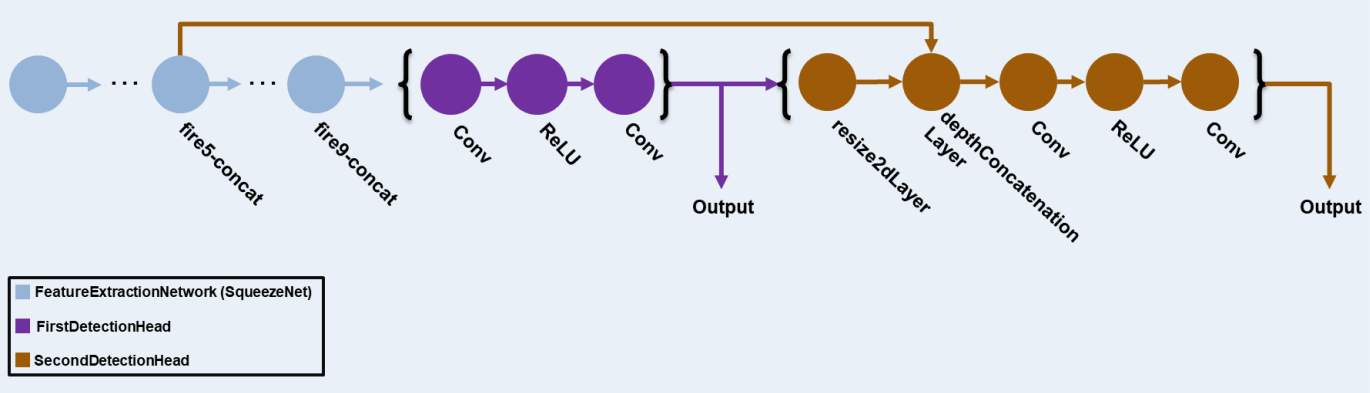
```
    augmentedData{k} = insertShape(data{1,1}, 'Rectangle', data{1,2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData, 'BorderSize', 10)
```

Define YOLO v3 Object Detector

The YOLO v3 detector in this example is based on SqueezeNet, and uses the feature extraction network in SqueezeNet with the addition of two detection heads at the end. The second detection head is twice the size of the first detection head, so it is better able to detect small objects. Note that you can specify any number of detection heads of different sizes based on the size of the objects that you want to detect. The YOLO v3 detector uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the detector learn to predict the boxes accurately. For information about anchor boxes, see “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

The YOLO v3 network present in the YOLO v3 detector is illustrated in the following diagram.

You can use Deep Network Designer to create the network shown in the diagram.



Specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [227 227 3].

```
networkInputSize = [227 227 3];
```

First, use `transform` to preprocess the training data for computing the anchor boxes, as the training images used in this example are bigger than 227-by-227 and vary in size. Specify the number of anchors as 6 to achieve a good tradeoff between number of anchors and mean IoU. Use the `estimateAnchorBoxes` function to estimate the anchor boxes. For details on estimating anchor boxes, see “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox). In case of using a pretrained YOLOv3 object detector, the anchor boxes calculated on that particular training dataset need to be specified. Note that the estimation process is not deterministic. To prevent the estimated anchor boxes from changing while tuning other hyperparameters set the random seed prior to estimation using rng.

```
rng(0)
trainingDataForEstimation = transform(trainingData, @(data)preprocessData(data, networkInputSize)
numAnchors = 6;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchors = 6×2

    41    34
   163   130
    98    93
   144   125
    33    24
    69    66


meanIoU = 0.8507
```

Specify `anchorBoxes` to use in both the detection heads. `anchorBoxes` is a cell array of [Mx1], where M denotes the number of detection heads. Each detection head consists of a [Nx2] matrix of `anchors`, where N is the number of anchors to use. Select `anchorBoxes` for each detection head based on the feature map size. Use larger `anchors` at lower scale and smaller `anchors` at higher scale. To do so, sort the `anchors` with the larger anchor boxes first and assign the first three to the first detection head and the next three to the second detection head.

```
area = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxes = {anchors(1:3,:)
    anchors(4:6,:)
    };
```

Load the SqueezeNet network pretrained on Imagenet data set and then specify the class names. You can also choose to load a different pretrained network trained on COCO data set such as `tiny-yolov3-coco` or `darknet53-coco` or Imagenet data set such as MobileNet-v2 or ResNet-18. YOLO v3 performs better and trains faster when you use a pretrained network.

```
baseNetwork = squeezenet;
classNames = trainingDataTbl.Properties.VariableNames(2:end);
```

Next, create the `yolov3ObjectDetector` object by adding the detection network source. Choosing the optimal detection network source requires trial and error, and you can use `analyzeNetwork` to find the names of potential detection network source within a network. For this example, use the `fire9-concat` and `fire5-concat` layers as `DetectionNetworkSource`.

```
yolov3Detector = yolov3ObjectDetector(baseNetwork, classNames, anchorBoxes, 'DetectionNetworkSou
```

Alternatively, instead of the network created above using SqueezeNet, other pretrained YOLOv3 architectures trained using larger datasets like MS-COCO can be used to transfer learn the detector on custom object detection task. Transfer learning can be realized by changing the classNames and anchorBoxes.

Preprocess Training Data

Preprocess the augmented training data to prepare for training. The `preprocess` (Computer Vision Toolbox) method in `yolov3ObjectDetector` (Computer Vision Toolbox), applies the following preprocessing operations to the input data.

- Resize the images to the network input size by maintaining the aspect ratio.
- Scale the image pixels in the range `[0 1]`.

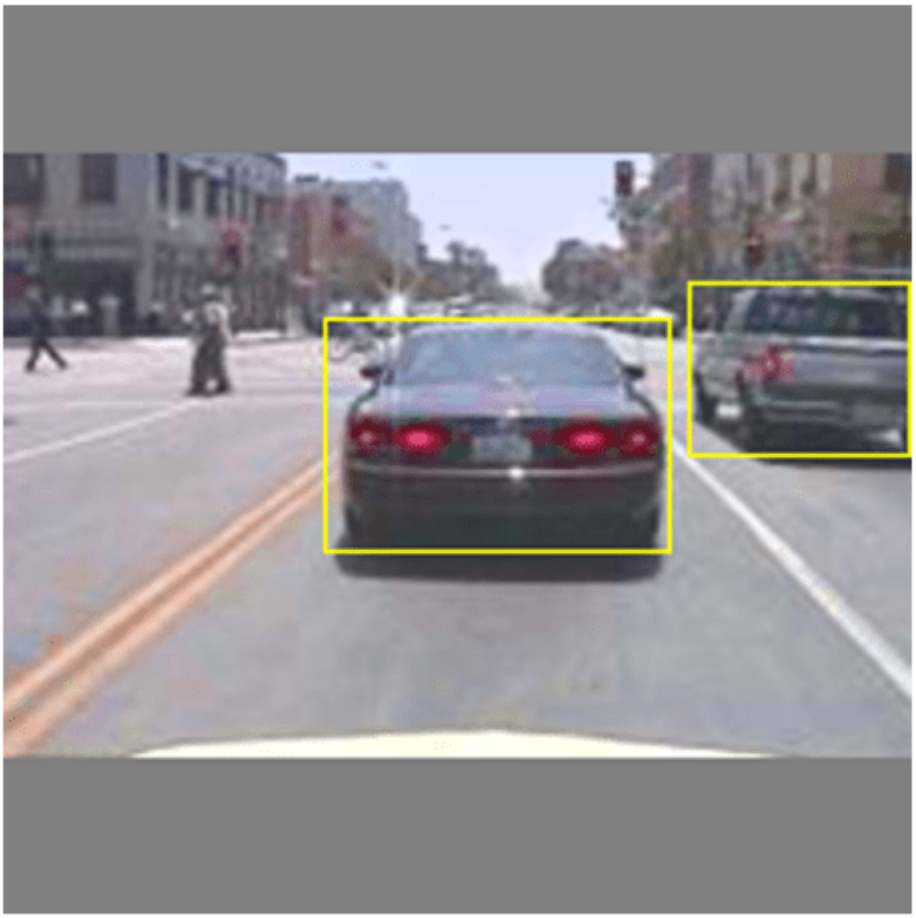
```
preprocessedTrainingData = transform(augmentedTrainingData, @(data)preprocess(yolov3Detector, da
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image with the bounding boxes.

```
I = data{1,1};
bbox = data{1,2};
annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Reset the datastore.

```
reset(preprocessedTrainingData);
```

Specify Training Options

Specify these training options.

- Set the number of epochs to be 80.
- Set the mini batch size as `8`. Stable training can be possible with higher learning rates when higher mini batch size is used. Although, this should be set depending on the available memory.
- Set the learning rate to 0.001.
- Set the warmup period as `1000` iterations. This parameter denotes the number of iterations to increase the learning rate exponentially based on the formula learningRate $\times \left(\frac{\text{iteration}}{\text{warmupPeriod}}\right)^4$. It helps in stabilizing the gradients at higher learning rates.

- Set the L2 regularization factor to 0.0005.
- Specify the penalty threshold as 0.5. Detections that overlap less than 0.5 with the ground truth are penalized.
- Initialize the velocity of gradient as `[]`. This is used by SGDM to store the velocity of gradients.

```
numEpochs = 80;
miniBatchSize = 8;
learningRate = 0.001;
warmupPeriod = 1000;
l2Regularization = 0.0005;
penaltyThreshold = 0.5;
velocity = [];
```

Train Model

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

Use the `minibatchqueue` function to split the preprocessed training data into batches with the supporting function `createBatchData` which returns the batched images and bounding boxes combined with the respective class IDs. For faster extraction of the batch data for training, `dispatchInBackground` should be set to "true" which ensures the usage of parallel pool.

`minibatchqueue` automatically detects the availability of a GPU. If you do not have a GPU, or do not want to use one for training, set the `OutputEnvironment` parameter to `"cpu"`.

```
if canUseParallelPool
   dispatchInBackground = true;
else
   dispatchInBackground = false;
end

mbqTrain = minibatchqueue(preprocessedTrainingData, 2,...
        "MiniBatchSize", miniBatchSize,...
        "MiniBatchFcn", @(images, boxes, labels) createBatchData(images, boxes, labels, className
        "MiniBatchFormat", ["SSCB", ""],...
        "DispatchInBackground", dispatchInBackground,...
        "OutputCast", ["", "double"]);
```

Create the training progress plotter using supporting function `configureTrainingProgressPlotter` to see the plot while training the detector object with a custom training loop.

Finally, specify the custom training loop. For each iteration:

- Read data from the `minibatchqueue`. If it doesn't have any more data, reset the `minibatchqueue` and shuffle.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function. The function `modelGradients`, listed as a supporting function, returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.
- Apply a weight decay factor to the gradients to regularization for more robust training.
- Determine the learning rate based on the iterations using the `piecewiseLearningRateWithWarmup` supporting function.

- Update the detector parameters using the `sgdmupdate` function.
- Update the `state` parameters of detector with the moving average.
- Display the learning rate, total loss, and the individual losses (box loss, object loss and class loss) for every iteration. These can be used to interpret how the respective losses are changing in each iteration. For example, a sudden spike in the box loss after few iterations implies that there are Inf or NaNs in the predictions.
- Update the training progress plot.

The training can also be terminated if the loss has saturated for few epochs.

```
if doTraining

    % Create subplots for the learning rate and mini-batch loss.
    fig = figure;
    [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(fig);

    iteration = 0;
    % Custom training loop.
    for epoch = 1:numEpochs

        reset(mbqTrain);
        shuffle(mbqTrain);

        while(hasdata(mbqTrain))
            iteration = iteration + 1;

            [XTrain, YTrain] = next(mbqTrain);

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients function.
            [gradients, state, lossInfo] = dlfeval(@modelGradients, yolov3Detector, XTrain, YTra

            % Apply L2 regularization.
            gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients, yolov3Detector.Learnab

            % Determine the current learning rate value.
            currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPe

            % Update the detector learnable parameters using the SGDM optimizer.
            [yolov3Detector.Learnables, velocity] = sgdmupdate(yolov3Detector.Learnables, gradie

            % Update the state parameters of dlnetwork.
            yolov3Detector.State = state;

            % Display progress.
            displayLossInfo(epoch, iteration, currentLR, lossInfo);

            % Update training plot with new points.
            updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR, lossInfo.totalLos
        end
    end
else
    yolov3Detector = preTrainedDetector;
end
```

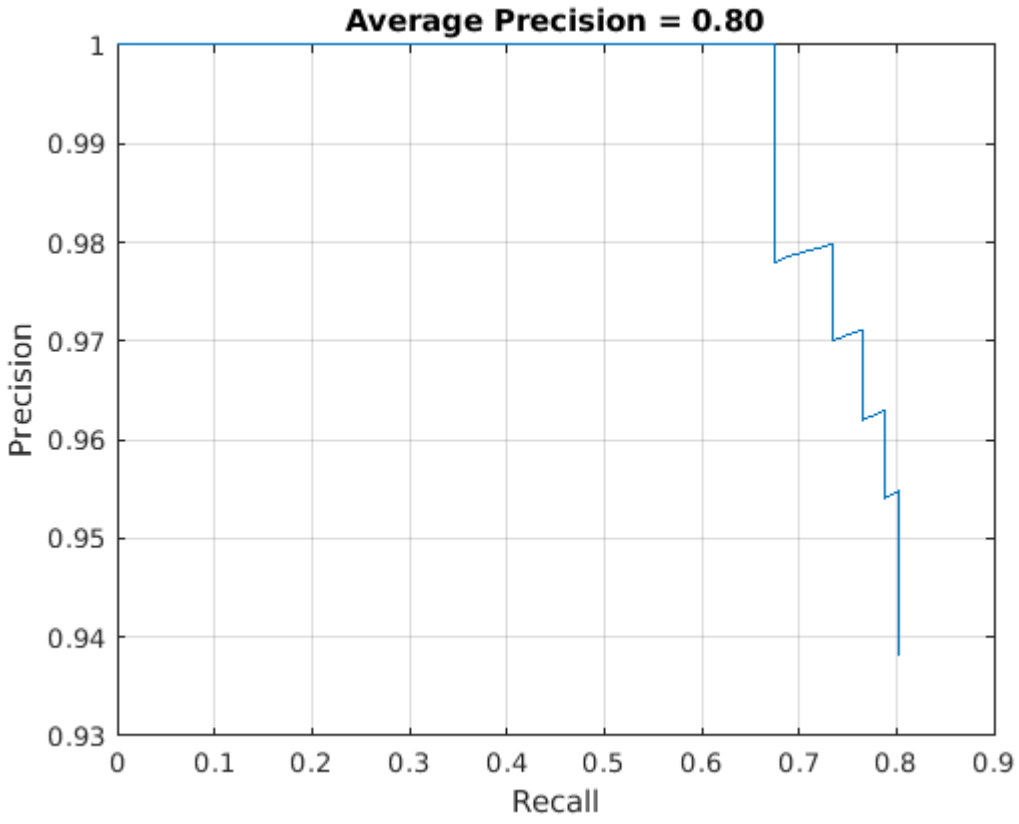
Evaluate Model

Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). In this example, the average precision metric is used. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

```
results = detect(yolov3Detector,testData,'MiniBatchSize',8);

% Evaluate the object detector using Average Precision metric.
[ap,recall,precision] = evaluateDetectionPrecision(results,testData);
```

The precision-recall (PR) curve shows how precise a detector is at varying levels of recall. Ideally, the precision is 1 at all recall levels.

```
% Plot precision-recall curve.
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```



Detect Objects Using YOLO v3

Use the detector for object detection.

```
% Read the datastore.
data = read(testData);

% Get the image.
I = data{1};

[bboxes,scores,labels] = detect(yolov3Detector,I);

% Display the detections on image.
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);

figure
imshow(I)
```

Supporting Functions

Model Gradients Function

The function `modelGradients` takes the `yolov3ObjectDetector` object, a mini-batch of input data `XTrain` with corresponding ground truth boxes `YTrain`, the specified penalty threshold as input arguments and returns the gradients of the loss with respect to the learnable parameters in `yolov3ObjectDetector`, the corresponding mini-batch loss information, and the state of the current batch.

The model gradients function computes the total loss and gradients by performing these operations.

- Generate predictions from the input batch of images using the `forward` method.
- Collect predictions on the CPU for postprocessing.
- Convert the predictions from the YOLO v3 grid cell coordinates to bounding box coordinates to allow easy comparison with the ground truth data by using the `anchorBoxGenerator` method of `yolov3ObjectDetector`.

- Generate targets for loss computation by using the converted predictions and ground truth data. These targets are generated for bounding box positions (x, y, width, height), object confidence, and class probabilities. See the supporting function `generateTargets`.
- Calculates the mean squared error of the predicted bounding box coordinates with target boxes. See the supporting function `bboxOffsetLoss`.
- Determines the binary cross-entropy of the predicted object confidence score with target object confidence score. See the supporting function `objectnessLoss`.
- Determines the binary cross-entropy of the predicted class of object with the target. See the supporting function `classConfidenceLoss`.
- Computes the total loss as the sum of all losses.
- Computes the gradients of learnables with respect to the total loss.

```
function [gradients, state, info] = modelGradients(detector, XTrain, YTrain, penaltyThreshold)
inputImageSize = size(XTrain,1:2);

% Gather the ground truths in the CPU for post processing
YTrain = gather(extractdata(YTrain));

% Extract the predictions from the detector.
[gatheredPredictions, YPredCell, state] = forward(detector, XTrain);

% Generate target for predictions from the ground truth data.
[boxTarget, objectnessTarget, classTarget, objectMaskTarget, boxErrorScale] = generateTargets(gat
    YTrain, inputImageSize, detector.AnchorBoxes, penaltyThreshold);

% Compute the loss.
boxLoss = bboxOffsetLoss(YPredCell(:,[2 3 7 8]),boxTarget,objectMaskTarget,boxErrorScale);
objLoss = objectnessLoss(YPredCell(:,1),objectnessTarget,objectMaskTarget);
clsLoss = classConfidenceLoss(YPredCell(:,6),classTarget,objectMaskTarget);
totalLoss = boxLoss + objLoss + clsLoss;

info.boxLoss = boxLoss;
info.objLoss = objLoss;
info.clsLoss = clsLoss;
info.totalLoss = totalLoss;

% Compute gradients of learnables with regard to loss.
gradients = dlgradient(totalLoss, detector.Learnables);
end

function boxLoss = bboxOffsetLoss(boxPredCell, boxDeltaTarget, boxMaskTarget, boxErrorScaleTarget
% Mean squared error for bounding box position.
lossX = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,1),boxDeltaTarget(:,1),boxMask
lossY = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,2),boxDeltaTarget(:,2),boxMask
lossW = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,3),boxDeltaTarget(:,3),boxMask
lossH = sum(cellfun(@(a,b,c,d) mse(a.*c.*d,b.*c.*d),boxPredCell(:,4),boxDeltaTarget(:,4),boxMask
boxLoss = lossX+lossY+lossW+lossH;
end

function objLoss = objectnessLoss(objectnessPredCell, objectnessDeltaTarget, boxMaskTarget)
% Binary cross-entropy loss for objectness score.
objLoss = sum(cellfun(@(a,b,c) crossentropy(a.*c,b.*c,'TargetCategories','independent'),objectnes
end

function clsLoss = classConfidenceLoss(classPredCell, classTarget, boxMaskTarget)
```

```
% Binary cross-entropy loss for class confidence score.
clsLoss = sum(cellfun(@(a,b,c) crossentropy(a.*c,b.*c,'TargetCategories','independent'),classPred
end
```

Augmentation and Data Processing Functions

```
function data = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.

data = cell(size(A));
for ii = 1:size(A,1)
    I = A{ii,1};
    bboxes = A{ii,2};
    labels = A{ii,3};
    sz = size(I);

    if numel(sz) == 3 && sz(3) == 3
        I = jitterColorHSV(I,...
            'Contrast',0.0,...
            'Hue',0.1,...
            'Saturation',0.2,...
            'Brightness',0.2);
    end

    % Randomly flip image.
    tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
    rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');
    I = imwarp(I,tform,'OutputView',rout);

    % Apply same transform to boxes.
    [bboxes,indices] = bboxwarp(bboxes,tform,rout,'OverlapThreshold',0.25);
    labels = labels(indices);

    % Return original data only when all boxes are removed by warping.
    if isempty(indices)
        data(ii,:) = A(ii,:);
    else
        data(ii,:) = {I, bboxes, labels};
    end
end
end


function data = preprocessData(data, targetSize)
% Resize the images and scale the pixels to between 0 and 1. Also scale the
% corresponding bounding boxes.

for ii = 1:size(data,1)
    I = data{ii,1};
    imgSize = size(I);

    % Convert an input image with single channel to 3 channels.
    if numel(imgSize) < 3
        I = repmat(I,1,1,3);
    end
    bboxes = data{ii,2};
```

```
    I = im2single(imresize(I,targetSize(1:2)));
    scale = targetSize(1:2)./imgSize(1:2);
    bboxes = bboxresize(bboxes,scale);

    data(ii, 1:2) = {I, bboxes};
end
end

function [XTrain, YTrain] = createBatchData(data, groundTruthBoxes, groundTruthClasses, className
% Returns images combined along the batch dimension in XTrain and
% normalized bounding boxes concatenated with classIDs in YTrain

% Concatenate images along the batch dimension.
XTrain = cat(4, data{:,1});

% Get class IDs from the class names.
classNames = repmat({categorical(classNames')}, size(groundTruthClasses));
[~, classIndices] = cellfun(@(a,b)ismember(a,b), groundTruthClasses, classNames, 'UniformOutput'

% Append the label indexes and training image size to scaled bounding boxes
% and create a single cell array of responses.
combinedResponses = cellfun(@(bbox, classid)[bbox, classid], groundTruthBoxes, classIndices, 'Uni
len = max( cellfun(@(x)size(x,1), combinedResponses ) );
paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'), combinedResponses, 'Uniform
YTrain = cat(4, paddedBBoxes{:,1});
end
```

Learning Rate Schedule Function

```
function currentLR = piecewiseLearningRateWithWarmup(iteration, epoch, learningRate, warmupPerioc
% The piecewiseLearningRateWithWarmup function computes the current
% learning rate based on the iteration number.
persistent warmUpEpoch;

if iteration <= warmupPeriod
    % Increase the learning rate for number of iterations in warmup period.
    currentLR = learningRate * ((iteration/warmupPeriod)^4);
    warmUpEpoch = epoch;
elseif iteration >= warmupPeriod && epoch < warmUpEpoch+floor(0.6*(numEpochs-warmUpEpoch))
    % After warm up period, keep the learning rate constant if the remaining number of epochs is
    currentLR = learningRate;

elseif epoch >= warmUpEpoch + floor(0.6*(numEpochs-warmUpEpoch)) && epoch < warmUpEpoch+floor(0.9
    % If the remaining number of epochs is more than 60 percent but less
    % than 90 percent multiply the learning rate by 0.1.
    currentLR = learningRate*0.1;

else
    % If remaining epochs are more than 90 percent multiply the learning
    % rate by 0.01.
    currentLR = learningRate*0.01;
end

end
```

Utility Functions

```
function [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(f)
% Create the subplots to display the loss and learning rate.
figure(f);
clf
subplot(2,1,1);
ylabel('Learning Rate');
xlabel('Iteration');
learningRatePlotter = animatedline;
subplot(2,1,2);
ylabel('Total Loss');
xlabel('Iteration');
lossPlotter = animatedline;
end

function displayLossInfo(epoch, iteration, currentLR, lossInfo)
% Display loss information for each iteration.
disp("Epoch : " + epoch + " | Iteration : " + iteration + " | Learning Rate : " + currentLR + ..
   " | Total Loss : " + double(gather(extractdata(lossInfo.totalLoss))) + ...
   " | Box Loss : " + double(gather(extractdata(lossInfo.boxLoss))) + ...
   " | Object Loss : " + double(gather(extractdata(lossInfo.objLoss))) + ...
   " | Class Loss : " + double(gather(extractdata(lossInfo.clsLoss))));
end

function updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR, totalLoss)
% Update loss and learning rate plots.
addpoints(lossPlotter, iteration, double(extractdata(gather(totalLoss))));
addpoints(learningRatePlotter, iteration, currentLR);
drawnow
end

function detector = downloadPretrainedYOLOv3Detector()
% Download a pretrained yolov3 detector.
if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.mat', 'file')
    if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.zip', 'file')
        disp('Downloading pretrained detector...');
        pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehi
        websave('yolov3SqueezeNetVehicleExample_21aSPKG.zip', pretrainedURL);
    end
    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');
end
pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
detector = pretrained.detector;
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” Preprint, submitted April 8, 2018. https://arxiv.org/abs/1804.02767.

See Also

Apps
Deep Network Designer

Functions
`estimateAnchorBoxes` | `analyzeNetwork` | `combine` | `transform` | `dlfeval` | `read` | `evaluateDetectionPrecision` | `sgdmupdate` | `dlupdate`

Objects
`boxLabelDatastore` | `imageDatastore` | `dlnetwork` | `dlarray`

More About

- “Anchor Boxes for Object Detection” (Computer Vision Toolbox)
- “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox)
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Getting Started with Object Detection Using Deep Learning” (Computer Vision Toolbox)

Object Detection Using YOLO v2 Deep Learning

This example shows how to train a you only look once (YOLO) v2 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a YOLO v2 vehicle detector using the `trainYOLOv2ObjectDetector` function. For more information, see "Getting Started with YOLO v2" (Computer Vision Toolbox).

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('yolov2ResNet50VehicleExample_19b.mat','file')
    disp('Downloading pretrained detector (98 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50VehicleExar
    websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small vehicle dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the YOLO v2 training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

```
% Display first few rows of the data set.
vehicleDataset(1:4,:)
```

```
ans=4×2 table
              imageFilename                vehicle
    ___________________________________    ___________

    {'vehicleImages/image_00001.jpg'}    {1×4 double}
    {'vehicleImages/image_00002.jpg'}    {1×4 double}
    {'vehicleImages/image_00003.jpg'}    {1×4 double}
    {'vehicleImages/image_00004.jpg'}    {1×4 double}
```

```
% Add the fullpath to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd,vehicleDataset.imageFilename);
```

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

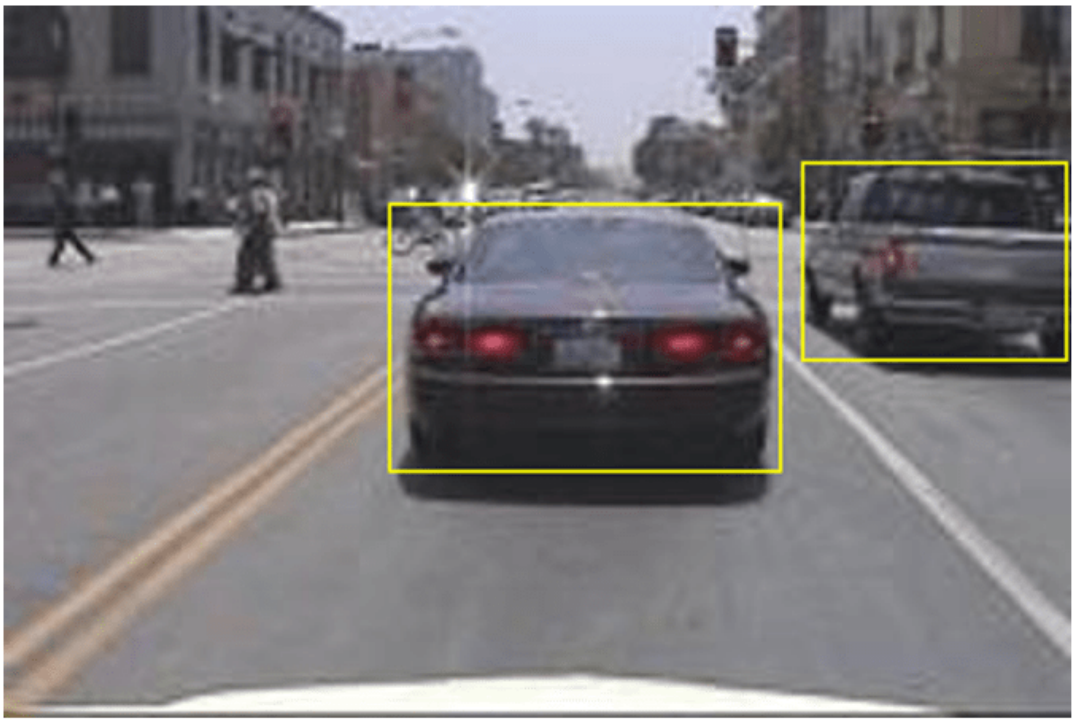
```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:,'imageFilename'});
bldsValidation = boxLabelDatastore(validationDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
validationData = combine(imdsValidation,bldsValidation);
testData = combine(imdsTest,bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Create a YOLO v2 Object Detection Network

A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” on page 1-8). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific for YOLO v2.

Use the `yolov2Layers` (Computer Vision Toolbox) function to create a YOLO v2 object detection network automatically given a pretrained ResNet-50 feature extraction network. `yolov2Layers` requires you to specify several inputs that parameterize a YOLO v2 network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size and the number of classes. When choosing the network input size, consider the minimum size required by the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` (Computer Vision Toolbox) to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes. Resize the training data to the input image size of the network using the supporting function `preprocessData`.

```
trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,inputSize));
numAnchors = 7;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchorBoxes = 7×2

   162   136
    85    80
   149   123
    43    32
    65    63
   117   105
    33    27


meanIoU = 0.8472
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox) (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer to replace the layers after `'activation_40_relu'` with the detection subnetwork. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis.

```
featureLayer = 'activation_40_relu';
```

Create the YOLO v2 object detection network.

```
lgraph = yolov2Layers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the YOLO v2 network architecture, use Deep Network Designer to design the YOLO v2 detection network manually. For more information, see “Design a YOLO v2 Detection Network” (Computer Vision Toolbox).

Data Augmentation

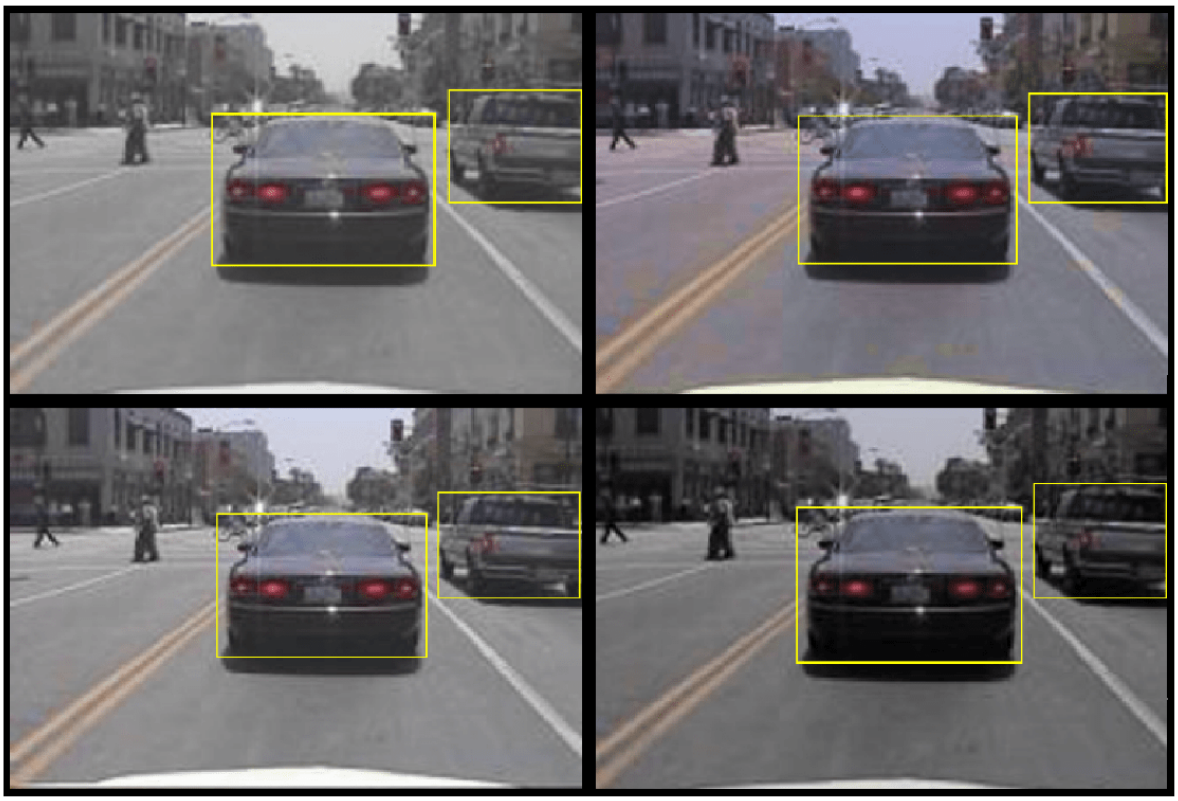
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
% Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
preprocessedValidationData = transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Train YOLO v2 Object Detector

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
        'MiniBatchSize',16, ....
        'InitialLearnRate',1e-3, ...
        'MaxEpochs',20, ...
        'CheckpointPath',tempdir, ...
        'ValidationData',preprocessedValidationData);
```

Use `trainYOLOv2ObjectDetector` (Computer Vision Toolbox) function to train YOLO v2 object detector if `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % Train the YOLO v2 detector.
```

```
    [detector,info] = trainYOLOv2ObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load('yolov2ResNet50VehicleExample_19b.mat');
    detector = pretrained.detector;
end
```

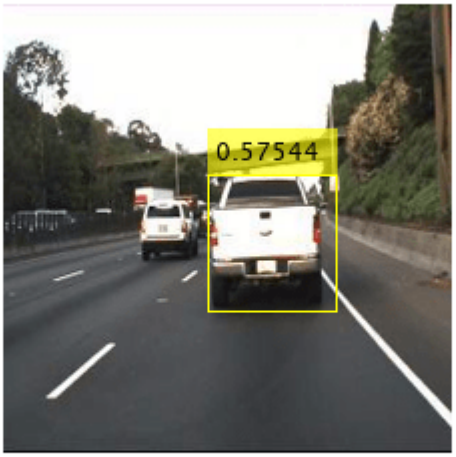
This example was verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the `'MiniBatchSize'` using the `trainingOptions` function. Training this network took approximately 7 minutes using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on a test image. Make sure you resize the image to the same size as the training images.

```
I = imread('highway.png');
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

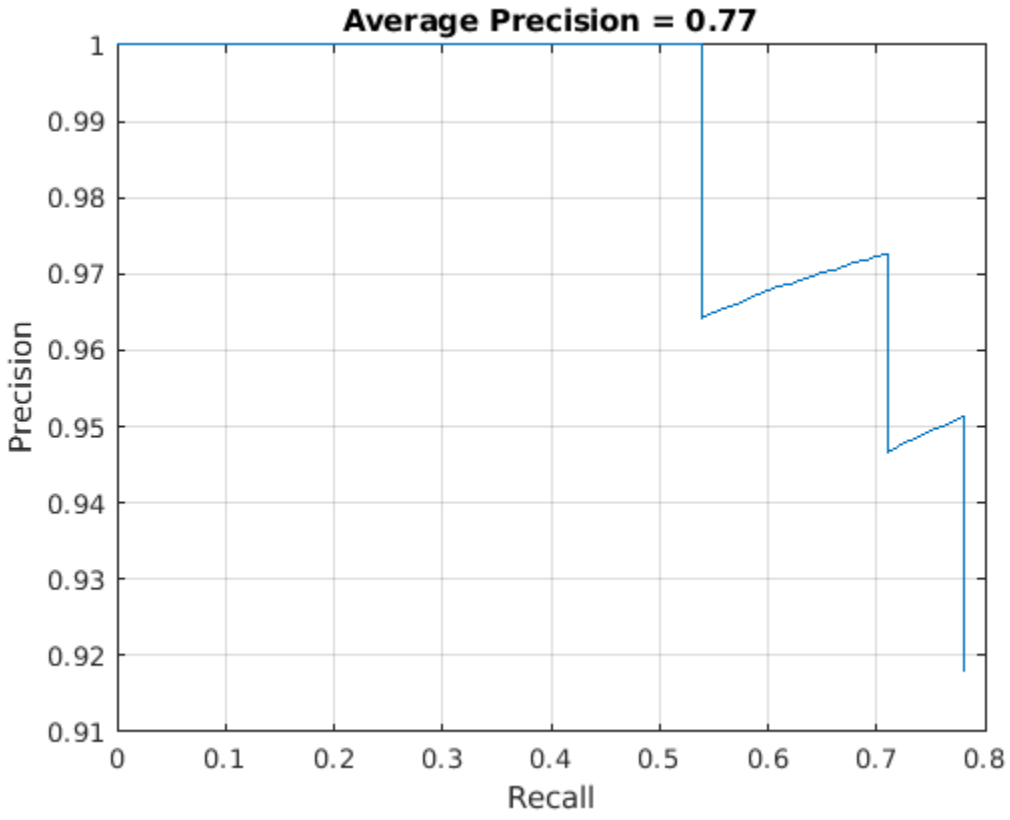
```
detectionResults = detect(detector, preprocessedTestData);
```

Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `yolov2ObjectDetector` using GPU Coder™. See “Code Generation for Object Detection by Using YOLO v2” (GPU Coder) example for more details.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.

B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end

% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
B{1} = imwarp(I,tform,'OutputView',rout);

% Sanitize box data, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2},sz);

% Resize boxes to new image size.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. “YOLO9000: Better, Faster, Stronger.” In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517–25. Honolulu, HI: IEEE, 2017. https://doi.org/10.1109/CVPR.2017.690.

Semantic Segmentation Using Deep Learning

This example shows how to train a semantic segmentation network using deep learning.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

To illustrate the training procedure, this example trains Deeplab v3+ [1], one type of convolutional neural network (CNN) designed for semantic image segmentation. Other types of networks for semantic segmentation include fully convolutional networks (FCN), SegNet, and U-Net. The training procedure shown here can be applied to those networks too.

This example uses the CamVid dataset [2] from the University of Cambridge for training. This dataset is a collection of images containing street-level views obtained while driving. The dataset provides pixel-level labels for 32 semantic classes including car, pedestrian, and road.

Setup

This example creates the Deeplab v3+ network with weights initialized from a pre-trained Resnet-18 network. ResNet-18 is an efficient network that is well suited for applications with limited processing resources. Other pretrained networks such as MobileNet v2 or ResNet-50 can also be used depending on application requirements. For more details, see “Pretrained Deep Neural Networks” on page 1-8.

To get a pretrained Resnet-18, install `resnet18`. After installation is complete, run the following code to verify that the installation is correct.

```
resnet18();
```

In addition, download a pretrained version of DeepLab v3+. The pretrained model allows you to run the entire example without having to wait for training to complete.

```
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/deeplabv3plusResnet18CamVid.m
pretrainedFolder = fullfile(tempdir,'pretrainedNetwork');
pretrainedNetwork = fullfile(pretrainedFolder,'deeplabv3plusResnet18CamVid.mat');
if ~exist(pretrainedNetwork,'file')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetwork,pretrainedURL);
end
```

A CUDA-capable NVIDIA™ GPU is highly recommended for running this example. Use of a GPU requires Parallel Computing Toolbox™. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

Download CamVid Dataset

Download the CamVid dataset from the following URLs.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

outputFolder = fullfile(tempdir,'CamVid');
labelsZip = fullfile(outputFolder,'labels.zip');
imagesZip = fullfile(outputFolder,'images.zip');
```

```
if ~exist(labelsZip, 'file') || ~exist(imagesZip,'file')
    mkdir(outputFolder)

    disp('Downloading 16 MB CamVid dataset labels...');
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder,'labels'));

    disp('Downloading 557 MB CamVid dataset images...');
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder,'images'));
end
```

Note: Download time of the data depends on your Internet connection. The commands used above block MATLAB until the download is complete. Alternatively, you can use your web browser to first download the dataset to your local disk. To use the file you downloaded from the web, change the `outputFolder` variable above to the location of the downloaded file.

Load CamVid Images

Use `imageDatastore` to load CamVid images. The `imageDatastore` enables you to efficiently load a large collection of images on disk.

```
imgDir = fullfile(outputFolder,'images','701_StillsRaw_full');
imds = imageDatastore(imgDir);
```

Display one of the images.

```
I = readimage(imds,559);
I = histeq(I);
imshow(I)
```

Load CamVid Pixel-Labeled Images

Use `pixelLabelDatastore` (Computer Vision Toolbox) to load CamVid pixel label image data. A `pixelLabelDatastore` encapsulates the pixel label data and the label ID to a class name mapping.

We make training easier, we group the 32 original classes in CamVid to 11 classes. Specify these classes.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];
```

To reduce 32 classes into 11, multiple classes from the original dataset are grouped together. For example, "Car" is a combination of "Car", "SUVPickupTruck", "Truck_Bus", "Train", and "OtherMoving". Return the grouped label IDs by using the supporting function `camvidPixelLabelIDs`, which is listed at the end of this example.

```
labelIDs = camvidPixelLabelIDs();
```

Use the classes and label IDs to create the `pixelLabelDatastore.`

```
labelDir = fullfile(outputFolder,'labels');
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Read and display one of the pixel-labeled images by overlaying it on top of an image.

```
C = readimage(pxds,559);
cmap = camvidColorMap;
B = labeloverlay(I,C,'ColorMap',cmap);
imshow(B)
pixelLabelColorbar(cmap,classes);
```

Areas with no color overlay do not have pixel labels and are not used during training.

Analyze Dataset Statistics

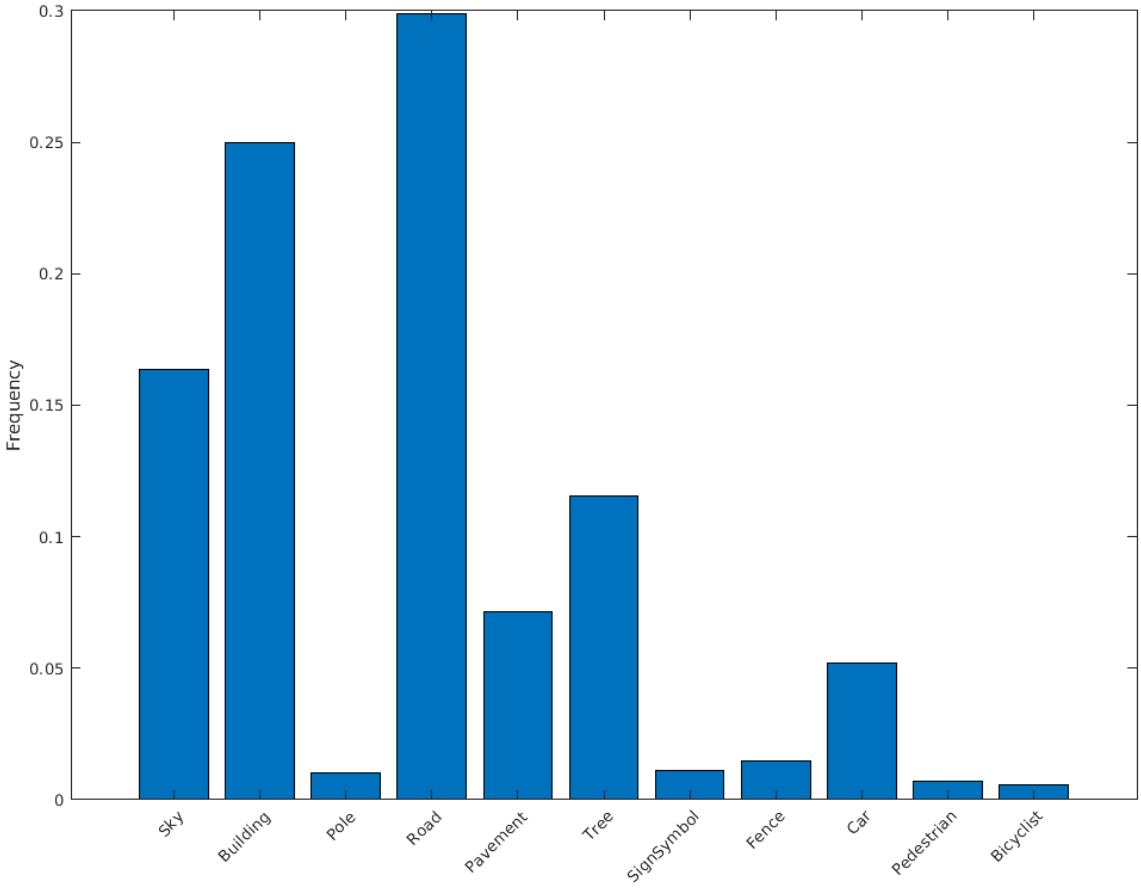
To see the distribution of class labels in the CamVid dataset, use `countEachLabel` (Computer Vision Toolbox). This function counts the number of pixels by class label.

```
tbl = countEachLabel(pxds)
```

```
tbl=11×3 table
         Name          PixelCount    ImagePixelCount
    _______________    __________    _______________

    {'Sky'       }    7.6801e+07      4.8315e+08
    {'Building'  }    1.1737e+08      4.8315e+08
    {'Pole'      }    4.7987e+06      4.8315e+08
    {'Road'      }    1.4054e+08      4.8453e+08
    {'Pavement'  }    3.3614e+07      4.7209e+08
    {'Tree'      }    5.4259e+07       4.479e+08
    {'SignSymbol'}    5.2242e+06      4.6863e+08
    {'Fence'     }    6.9211e+06       2.516e+08
    {'Car'       }    2.4437e+07      4.8315e+08
    {'Pedestrian'}    3.4029e+06      4.4444e+08
    {'Bicyclist' }    2.5912e+06      2.6196e+08
```

Visualize the pixel counts by class.

```
frequency = tbl.PixelCount/sum(tbl.PixelCount);

bar(1:numel(classes),frequency)
xticks(1:numel(classes))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes would have an equal number of observations. However, the classes in CamVid are imbalanced, which is a common issue in automotive data-sets of street scenes. Such scenes have more sky, building, and road pixels than pedestrian and bicyclist pixels because sky, buildings and roads cover more area in the image. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes. Later on in this example, you will use class weighting to handle this issue.

The images in the CamVid data set are 720 by 960 in size. Image size is chosen such that a large enough batch of images can fit in memory during training on an NVIDIA™ Titan X with 12 GB of memory. You may need to resize the images to smaller sizes if your GPU does not have sufficient memory or reduce the training batch size.

Prepare Training, Validation, and Test Sets

Deeplab v3+ is trained using 60% of the images from the dataset. The rest of the images are split evenly in 20% and 20% for validation and testing respectively. The following code randomly splits the image and pixel label data into a training, validation and test set.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = partitionCamVidData(imds,pxds);
```

The 60/20/20 split results in the following number of training, validation and test images:

```
numTrainingImages = numel(imdsTrain.Files)
```

```
numTrainingImages = 421
```

```
numValImages = numel(imdsVal.Files)
```

```
numValImages = 140
```

```
numTestingImages = numel(imdsTest.Files)
```

```
numTestingImages = 140
```

Create the Network

Use the `deeplabv3plusLayers` function to create a DeepLab v3+ network based on ResNet-18. Choosing the best network for your application requires empirical analysis and is another level of hyperparameter tuning. For example, you can experiment with different base networks such as ResNet-50 or MobileNet v2, or you can try other semantic segmentation network architectures such as SegNet, fully convolutional networks (FCN), or U-Net.

```
% Specify the network image size. This is typically the same as the traing image sizes.
imageSize = [720 960 3];

% Specify the number of classes.
numClasses = numel(classes);

% Create DeepLab v3+.
lgraph = deeplabv3plusLayers(imageSize, numClasses, "resnet18");
```

Balance Classes Using Class Weighting

As shown earlier, the classes in CamVid are not balanced. To improve training, you can use class weighting to balance the classes. Use the pixel label counts computed earlier with `countEachLabel` (Computer Vision Toolbox) and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq
```

```
classWeights = 11×1

    0.3182
    0.2082
    5.0924
    0.1744
    0.7103
    0.4175
    4.5371
    1.8386
    1.0000
    6.6059
      ⋮
```

Specify the class weights using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights
lgraph = replaceLayer(lgraph,"classification",pxLayer);
```

Select Training Options

The optimization algorithm used for training is stochastic gradient descent with momentum (SGDM). Use `trainingOptions` to specify the hyper-parameters used for SGDM.

```
% Define validation data.
dsVal = combine(imdsVal,pxdsVal);

% Define training options.
options = trainingOptions('sgdm', ...
    'LearnRateSchedule','piecewise',...
    'LearnRateDropPeriod',10,...
    'LearnRateDropFactor',0.3,...
    'Momentum',0.9, ...
    'InitialLearnRate',1e-3, ...
    'L2Regularization',0.005, ...
    'ValidationData',dsVal,...
    'MaxEpochs',30, ...
    'MiniBatchSize',8, ...
    'Shuffle','every-epoch', ...
    'CheckpointPath', tempdir, ...
    'VerboseFrequency',2,...
    'Plots','training-progress',...
    'ValidationPatience', 4);
```

The learning rate uses a piecewise schedule. The learning rate is reduced by a factor of 0.3 every 10 epochs. This allows the network to learn quickly with a higher initial learning rate, while being able to find a solution close to the local optimum once the learning rate drops.

The network is tested against the validation data every epoch by setting the `'ValidationData'` parameter. The `'ValidationPatience'` is set to 4 to stop training early when the validation accuracy converges. This prevents the network from overfitting on the training dataset.

A mini-batch size of 8 is used to reduce memory usage while training. You can increase or decrease this value based on the amount of GPU memory you have on your system.

In addition, `'CheckpointPath'` is set to a temporary location. This name-value pair enables the saving of network checkpoints at the end of every training epoch. If training is interrupted due to a system failure or power outage, you can resume training from the saved checkpoint. Make sure that the location specified by `'CheckpointPath'` has enough space to store the network checkpoints. For example, saving 100 Deeplab v3+ checkpoints requires ~6 GB of disk space because each checkpoint is 61 MB.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without increasing the number of labeled training samples. To apply the same random transformation to both image and pixel label data use datastore `combine` and `transform`. First, combine `imdsTrain` and `pxdsTrain`.

```
dsTrain = combine(imdsTrain, pxdsTrain);
```

Next, use datastore `transform` to apply the desired data augmentation defined in the supporting function `augmentImageAndLabel`. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation.

```
xTrans = [-10 10];
yTrans = [-10 10];
dsTrain = transform(dsTrain, @(data)augmentImageAndLabel(data,xTrans,yTrans));
```

Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

Start Training

Start training using `trainNetwork` if the `doTraining` flag is true. Otherwise, load a pretrained network.

Note: The training was verified on an NVIDIA™ Titan X with 12 GB of GPU memory. If your GPU has less memory, you may run out of memory during training. If this happens, try setting `'MiniBatchSize'` to 1 in `trainingOptions`, or reducing the network input and resizing the training data. Training this network takes about 5 hours. Depending on your GPU hardware, it can take even longer.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(dsTrain,lgraph,options);
else
    data = load(pretrainedNetwork);
    net = data.net;
end
```

Test Network on One Image

As a quick sanity check, run the trained network on one test image.

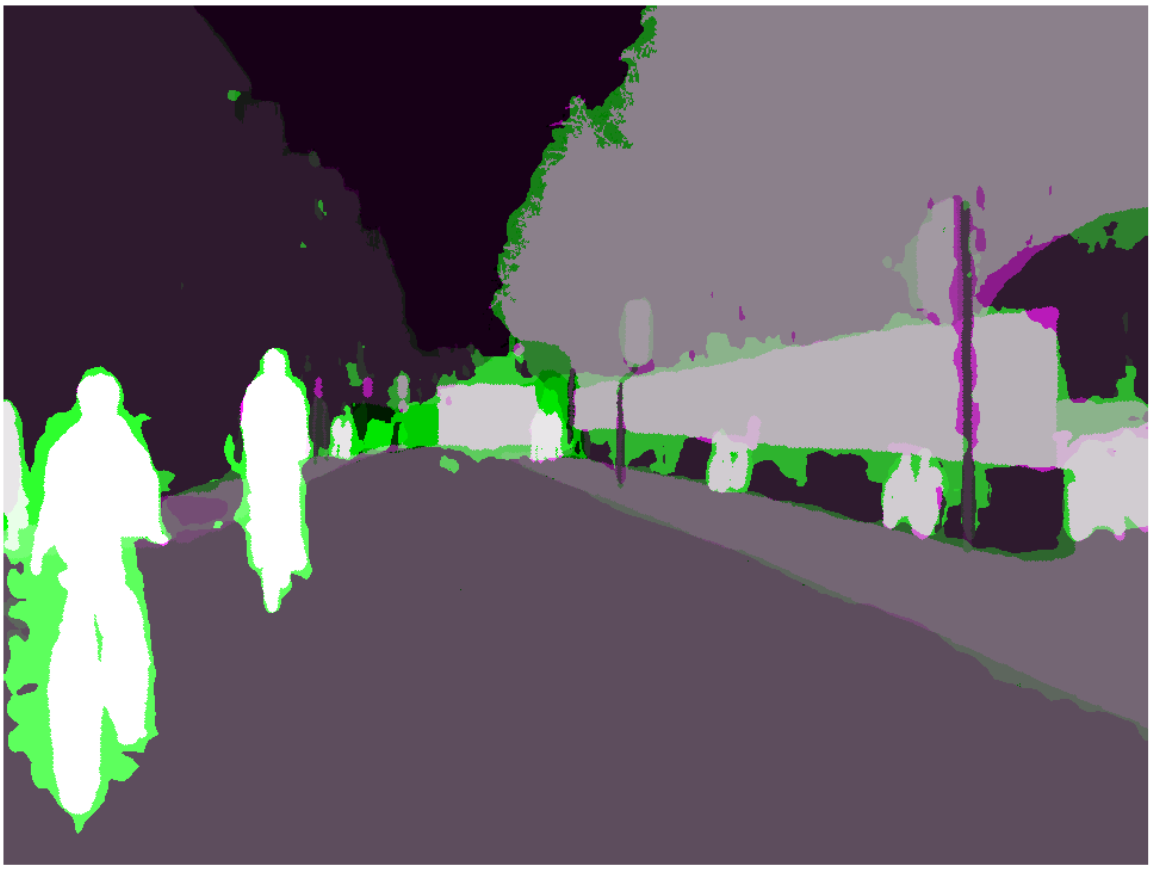
```
I = readimage(imdsTest,35);
C = semanticseg(I, net);
```

Display the results.

```
B = labeloverlay(I,C,'Colormap',cmap,'Transparency',0.4);
imshow(B)
pixelLabelColorbar(cmap, classes);
```

Compare the results in `C` with the expected ground truth stored in `pxdsTest`. The green and magenta regions highlight areas where the segmentation results differ from the expected ground truth.

```
expectedResult = readimage(pxdsTest,35);
actual = uint8(C);
expected = uint8(expectedResult);
imshowpair(actual, expected)
```



Visually, the semantic segmentation results overlap well for classes such as road, sky, and building. However, smaller objects like pedestrians and cars are not as accurate. The amount of overlap per class can be measured using the intersection-over-union (IoU) metric, also known as the Jaccard index. Use the `jaccard` (Image Processing Toolbox) function to measure IoU.

```
iou = jaccard(C,expectedResult);
table(classes,iou)
```

```
ans=11×2 table
      classes         iou
    ____________    _______

    "Sky"           0.91837
    "Building"      0.84479
    "Pole"          0.31203
    "Road"          0.93698
    "Pavement"      0.82838
    "Tree"          0.89636
    "SignSymbol"    0.57644
    "Fence"         0.71046
    "Car"           0.66688
    "Pedestrian"    0.48417
```

```
    "Bicyclist"     0.68431
```

The IoU metric confirms the visual results. Road, sky, and building classes have high IoU scores, while classes such as pedestrian and car have low scores. Other common segmentation metrics include the `dice` (Image Processing Toolbox) and the `bfscore` (Image Processing Toolbox) contour matching score.

Evaluate Trained Network

To measure accuracy for multiple test images, run`semanticseg` (Computer Vision Toolbox) on the entire test set. A mini-batch size of 4 is used to reduce memory usage while segmenting images. You can increase or decrease this value based on the amount of GPU memory you have on your system.

```
pxdsResults = semanticseg(imdsTest,net, ...
    'MiniBatchSize',4, ...
    'WriteLocation',tempdir, ...
    'Verbose',false);
```

`semanticseg` returns the results for the test set as a `pixelLabelDatastore` object. The actual pixel label data for each test image in `imdsTest` is written to disk in the location specified by the `'WriteLocation'` parameter. Use `evaluateSemanticSegmentation` (Computer Vision Toolbox) to measure semantic segmentation metrics on the test set results.

```
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTest,'Verbose',false);
```

`evaluateSemanticSegmentation` returns various metrics for the entire dataset, for individual classes, and for each test image. To see the dataset level metrics, inspect `metrics.DataSetMetrics` .

```
metrics.DataSetMetrics
```

```
ans=1×5 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.87695          0.85392       0.6302       0.80851        0.65051
```

The dataset metrics provide a high-level overview of the network performance. To see the impact each class has on the overall performance, inspect the per-class metrics using `metrics.ClassMetrics`.

```
metrics.ClassMetrics
```

```
ans=11×3 table
                  Accuracy      IoU      MeanBFScore
                  ________    _______    ___________

    Sky           0.93112     0.90209       0.8952
    Building      0.78453     0.76098      0.58511
    Pole          0.71586     0.21477      0.51439
    Road          0.93024     0.91465      0.76696
    Pavement      0.88466     0.70571      0.70919
    Tree          0.87377     0.76323      0.70875
    SignSymbol    0.79358     0.39309      0.48302
    Fence         0.81507     0.46484      0.48566
    Car           0.90956     0.76799      0.69233
```

```
    Pedestrian    0.87629      0.4366      0.60792
    Bicyclist     0.87844     0.60829      0.55089
```

Although the overall dataset performance is quite high, the class metrics show that underrepresented classes such as `Pedestrian`, `Bicyclist`, and `Car` are not segmented as well as classes such as `Road`, `Sky`, and `Building`. Additional data that includes more samples of the underrepresented classes might help improve the results.

Supporting Functions

```
function labelIDs = camvidPixelLabelIDs()
% Return the label IDs corresponding to each class.
%
% The CamVid dataset has 32 classes. Group them into 11 classes following
% the original SegNet training methodology [1].
%
% The 11 classes are:
%   "Sky" "Building", "Pole", "Road", "Pavement", "Tree", "SignSymbol",
%   "Fence", "Car", "Pedestrian",  and "Bicyclist".
%
% CamVid pixel label IDs are provided as RGB color values. Group them into
% 11 classes and return them as a cell array of M-by-3 matrices. The
% original CamVid class names are listed alongside each RGB value. Note
% that the Other/Void class are excluded below.
labelIDs = { ...

    % "Sky"
    [
    128 128 128; ... % "Sky"
    ]

    % "Building"
    [
    000 128 064; ... % "Bridge"
    128 000 000; ... % "Building"
    064 192 000; ... % "Wall"
    064 000 064; ... % "Tunnel"
    192 000 128; ... % "Archway"
    ]

    % "Pole"
    [
    192 192 128; ... % "Column_Pole"
    000 000 064; ... % "TrafficCone"
    ]

    % Road
    [
    128 064 128; ... % "Road"
    128 000 192; ... % "LaneMkgsDriv"
    192 000 064; ... % "LaneMkgsNonDriv"
    ]

    % "Pavement"
    [
    000 000 192; ... % "Sidewalk"
    064 192 128; ... % "ParkingBlock"
```

```
    128 128 192; ... % "RoadShoulder"
    ]

    % "Tree"
    [
    128 128 000; ... % "Tree"
    192 192 000; ... % "VegetationMisc"
    ]

    % "SignSymbol"
    [
    192 128 128; ... % "SignSymbol"
    128 128 064; ... % "Misc_Text"
    000 064 064; ... % "TrafficLight"
    ]

    % "Fence"
    [
    064 064 128; ... % "Fence"
    ]

    % "Car"
    [
    064 000 128; ... % "Car"
    064 128 192; ... % "SUVPickupTruck"
    192 128 192; ... % "Truck_Bus"
    192 064 128; ... % "Train"
    128 064 064; ... % "OtherMoving"
    ]

    % "Pedestrian"
    [
    064 064 000; ... % "Pedestrian"
    192 128 064; ... % "Child"
    064 000 192; ... % "CartLuggagePram"
    064 128 064; ... % "Animal"
    ]

    % "Bicyclist"
    [
    000 128 192; ... % "Bicyclist"
    192 000 192; ... % "MotorcycleScooter"
    ]

    };
end


function pixelLabelColorbar(cmap, classNames)
% Add a colorbar to the current axis. The colorbar is formatted
% to display the class names with the color.

colormap(gca,cmap)

% Add colorbar to current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
```

```
c.TickLabels = classNames;
numClasses = size(cmap,1);

% Center tick labels.
c.Ticks = 1/(numClasses*2):1/numClasses:1;

% Remove tick mark.
c.TickLength = 0;
end

function cmap = camvidColorMap()
% Define the colormap used by CamVid dataset.

cmap = [
    128 128 128   % Sky
    128 0 0       % Building
    192 192 192   % Pole
    128 64 128    % Road
    60 40 222     % Pavement
    128 128 0     % Tree
    192 128 128   % SignSymbol
    64 64 128     % Fence
    64 0 128      % Car
    64 64 0       % Pedestrian
    0 128 192     % Bicyclist
    ];

% Normalize between [0 1].
cmap = cmap ./ 255;
end

function [imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = partitionCamVidData(imds
% Partition CamVid data by randomly selecting 60% of the data for training. The
% rest is used for testing.

% Set initial random state for example reproducibility.
rng(0);
numFiles = numel(imds.Files);
shuffledIndices = randperm(numFiles);

% Use 60% of the images for training.
numTrain = round(0.60 * numFiles);
trainingIdx = shuffledIndices(1:numTrain);

% Use 20% of the images for validation
numVal = round(0.20 * numFiles);
valIdx = shuffledIndices(numTrain+1:numTrain+numVal);

% Use the rest for testing.
testIdx = shuffledIndices(numTrain+numVal+1:end);

% Create image datastores for training and test.
trainingImages = imds.Files(trainingIdx);
valImages = imds.Files(valIdx);
testImages = imds.Files(testIdx);

imdsTrain = imageDatastore(trainingImages);
imdsVal = imageDatastore(valImages);
```

```
imdsTest = imageDatastore(testImages);

% Extract class and label IDs info.
classes = pxds.ClassNames;
labelIDs = camvidPixelLabelIDs();

% Create pixel label datastores for training and test.
trainingLabels = pxds.Files(trainingIdx);
valLabels = pxds.Files(valIdx);
testLabels = pxds.Files(testIdx);

pxdsTrain = pixelLabelDatastore(trainingLabels, classes, labelIDs);
pxdsVal = pixelLabelDatastore(valLabels, classes, labelIDs);
pxdsTest = pixelLabelDatastore(testLabels, classes, labelIDs);
end

function data = augmentImageAndLabel(data, xTrans, yTrans)
% Augment images and pixel label images using random reflection and
% translation.

for i = 1:size(data,1)

    tform = randomAffine2d(...
        'XReflection',true,...
        'XTranslation', xTrans, ...
        'YTranslation', yTrans);

    % Center the view at the center of image in the output space while
    % allowing translation to move the output image out of view.
    rout = affineOutputView(size(data{i,1}), tform, 'BoundsStyle', 'centerOutput');

    % Warp the image and pixel labels using the same transform.
    data{i,1} = imwarp(data{i,1}, tform, 'OutputView', rout);
    data{i,2} = imwarp(data{i,2}, tform, 'OutputView', rout);

end
end
```

References

[1] Chen, Liang-Chieh et al. “Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation.” ECCV (2018).

[2] Brostow, G. J., J. Fauqueur, and R. Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

See Also

`pixelLabelDatastore` | `pixelLabelImageDatastore` | `semanticseg` | `labeloverlay` | `countEachLabel` | `segnetLayers` | `pixelClassificationLayer` | `trainingOptions` | `imageDataAugmenter` | `trainNetwork` | `evaluateSemanticSegmentation`

More About

- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection” (Computer Vision Toolbox)

- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 8-154
- “Semantic Segmentation Using Dilated Convolutions” on page 8-143
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox)
- “Pretrained Deep Neural Networks” on page 1-8

Semantic Segmentation Using Dilated Convolutions

Train a semantic segmentation network using dilated convolutions.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

Semantic segmentation networks like DeepLab [1] make extensive use of dilated convolutions (also known as atrous convolutions) because they can increase the receptive field of the layer (the area of the input which the layers can see) without increasing the number of parameters or computations.

Load Training Data

The example uses a simple dataset of 32-by-32 triangle images for illustration purposes. The dataset includes accompanying pixel label ground truth data. Load the training data using an `imageDatastore` and a `pixelLabelDatastore`.

```
dataFolder = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageFolderTrain = fullfile(dataFolder,'trainingImages');
labelFolderTrain = fullfile(dataFolder,'trainingLabels');
```

Create an `imageDatastore` for the images.

```
imdsTrain = imageDatastore(imageFolderTrain);
```

Create a `pixelLabelDatastore` for the ground truth pixel labels.

```
classNames = ["triangle" "background"];
labels = [255 0];
pxdsTrain = pixelLabelDatastore(labelFolderTrain,classNames,labels)
```

```
pxdsTrain =
  PixelLabelDatastore with properties:

                      Files: {200x1 cell}
                 ClassNames: {2x1 cell}
                   ReadSize: 1
                    ReadFcn: @readDatastoreImage
    AlternateFileSystemRoots: {}
```

Create Semantic Segmentation Network

This example uses a simple semantic segmentation network based on dilated convolutions.

Create a data source for training data and get the pixel counts for each label.

```
ds = combine(imdsTrain,pxdsTrain);
tbl = countEachLabel(pxdsTrain)
```

```
tbl=2×3 table
         Name          PixelCount    ImagePixelCount
    _______________    __________    _______________

    {'triangle'  }         10326       2.048e+05
```

```
    {'background'}    1.9447e+05       2.048e+05
```

The majority of pixel labels are for background. This class imbalance biases the learning process in favor of the dominant class. To fix this, use class weighting to balance the classes. You can use several methods to compute class weights. One common method is inverse frequency weighting where the class weights are the inverse of the class frequencies. This method increases the weight given to under represented classes. Calculate the class weights using inverse frequency weighting.

```
numberPixels = sum(tbl.PixelCount);
frequency = tbl.PixelCount / numberPixels;
classWeights = 1 ./ frequency;
```

Create a network for pixel classification by using an image input layer with an input size corresponding to the size of the input images. Next, specify three blocks of convolution, batch normalization, and ReLU layers. For each convolutional layer, specify 32 3-by-3 filters with increasing dilation factors and pad the inputs so they are the same size as the outputs by setting the `'Padding'` option to `'same'`. To classify the pixels, include a convolutional layer with K 1-by-1 convolutions, where K is the number of classes, followed by a softmax layer and a `pixelClassificationLayer` with the inverse class weights.

```
inputSize = [32 32 1];
filterSize = 3;
numFilters = 32;
numClasses = numel(classNames);

layers = [
    imageInputLayer(inputSize)

    convolution2dLayer(filterSize,numFilters,'DilationFactor',1,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(filterSize,numFilters,'DilationFactor',2,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(filterSize,numFilters,'DilationFactor',4,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(1,numClasses)
    softmaxLayer
    pixelClassificationLayer('Classes',classNames,'ClassWeights',classWeights)];
```

Train Network

Specify the training options.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs', 100, ...
    'MiniBatchSize', 64, ...
    'InitialLearnRate', 1e-3);
```

Train the network using `trainNetwork`.

```
net = trainNetwork(ds,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:01 |       91.62% |       1.6825 |          0.0010 |
|      17 |          50 |       00:00:25 |       88.56% |       0.2393 |          0.0010 |
|      34 |         100 |       00:00:49 |       92.08% |       0.1672 |          0.0010 |
|      50 |         150 |       00:01:14 |       93.17% |       0.1472 |          0.0010 |
|      67 |         200 |       00:01:42 |       94.15% |       0.1313 |          0.0010 |
|      84 |         250 |       00:02:14 |       94.47% |       0.1167 |          0.0010 |
|     100 |         300 |       00:02:38 |       95.04% |       0.1100 |          0.0010 |
|========================================================================================|
Training finished: Max epochs completed.
```

Test Network

Load the test data. Create an `imageDatastore` for the images. Create a `pixelLabelDatastore` for the ground truth pixel labels.

```
imageFolderTest = fullfile(dataFolder,'testImages');
imdsTest = imageDatastore(imageFolderTest);
labelFolderTest = fullfile(dataFolder,'testLabels');
pxdsTest = pixelLabelDatastore(labelFolderTest,classNames,labels);
```

Make predictions using the test data and trained network.

```
pxdsPred = semanticseg(imdsTest,net,'MiniBatchSize',32,'WriteLocation',tempdir);
```

```
Running semantic segmentation network
-------------------------------------
* Processed 100 images.
```

Evaluate the prediction accuracy using `evaluateSemanticSegmentation`.

```
metrics = evaluateSemanticSegmentation(pxdsPred,pxdsTest);
```

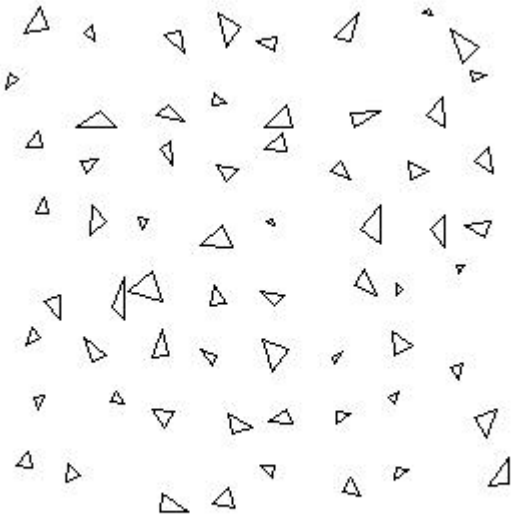
```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 100 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.95237           0.97352      0.72081      0.92889        0.46416
```

For more information on evaluating semantic segmentation networks, see `evaluateSemanticSegmentation` (Computer Vision Toolbox).

Segment New Image

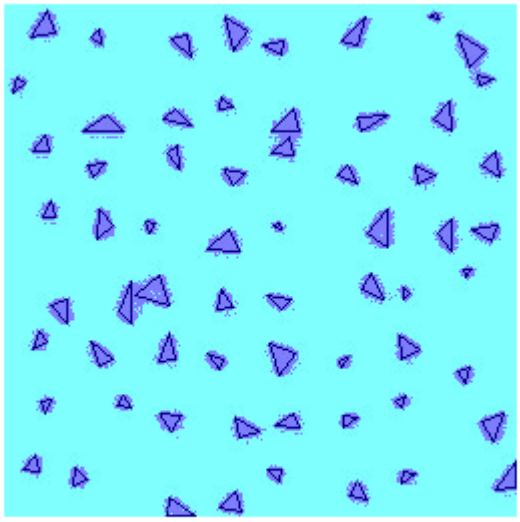
Read and display the test image `triangleTest.jpg`.

```
imgTest = imread('triangleTest.jpg');
figure
imshow(imgTest)
```



Segment the test image using `semanticseg` and display the results using `labeloverlay`.

```
C = semanticseg(imgTest,net);
B = labeloverlay(imgTest,C);
figure
imshow(B)
```



See Also

`pixelLabelDatastore` | `pixelLabelImageDatastore` | `semanticseg` | `labeloverlay` | `countEachLabel` | `pixelClassificationLayer` | `trainingOptions` | `trainNetwork` | `evaluateSemanticSegmentation` | `convolution2dLayer`

More About

- “Semantic Segmentation Using Deep Learning” on page 8-126
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 8-154
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox)
- “Pretrained Deep Neural Networks” on page 1-8

Train Simple Semantic Segmentation Network in Deep Network Designer

This example shows how to create and train a simple semantic segmentation network using Deep Network Designer.

Semantic segmentation describes the process of associating each pixel of an image with a class label (such as *flower*, *person*, *road*, *sky*, *ocean*, or *car*). Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

Preprocess Training Data

To train a semantic segmentation network, you need a collection of images and its corresponding collection of pixel-labeled images. A pixel-labeled image is an image where every pixel value represents the categorical label of that pixel. This example uses a simple data set of 32-by-32 images of triangles for illustration purposes. You can interactively label pixels and export the label data for computer vision applications using Image Labeler (Computer Vision Toolbox). For more information on creating training data for semantic segmentation applications, see “Label Pixels for Semantic Segmentation” (Computer Vision Toolbox).

Load the training data.

```
dataFolder  = fullfile(toolboxdir('vision'), ...
'visiondata','triangleImages');

imageDir = fullfile(dataFolder,'trainingImages');
labelDir = fullfile(dataFolder,'trainingLabels');
```

Create an `ImageDatastore` containing the images.

```
imds = imageDatastore(imageDir);
```

Create a `PixelLabelDatastore` containing the ground truth pixel labels. This data set has two classes: `"triangle"` and `"background"`.

```
classNames = ["triangle","background"];
labelIDs   = [255 0];

pxds = pixelLabelDatastore(labelDir,classNames,labelIDs);
```

Combine the image datastore and the pixel label datastore into a `CombinedDatastore` object using the `combine` function. A combined datastore maintains parity between the pair of images in the underlying datastores.

```
cds = combine(imds,pxds);
```

Build Network

Open Deep Network Designer.

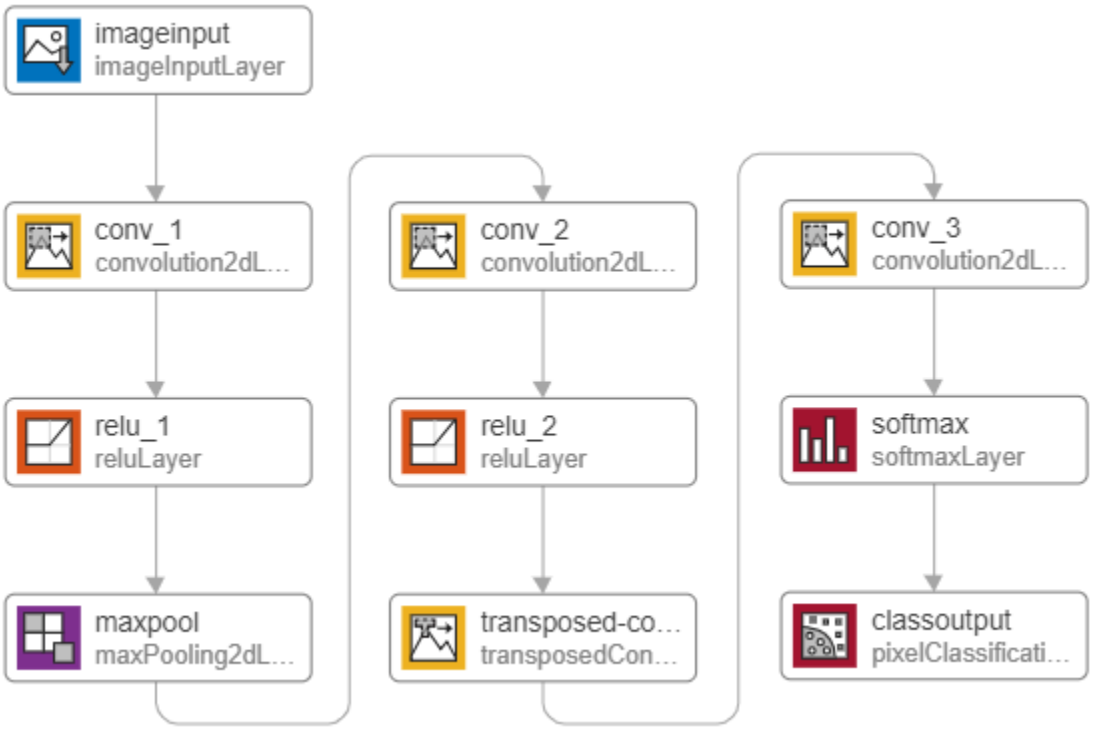
```
deepNetworkDesigner
```

In Deep Network Designer, you can build, edit, and train deep learning networks. Pause on **Blank Network** and click **New**.

Create a semantic segmentation network by dragging layers from the **Layer Library** to the **Designer** pane.

Connect the layers in this order:

1. `imageInputLayer` with `InputSize` set to `32,32,1`
2. `convolution2dLayer` with `FilterSize` set to `3,3`, `NumFilters` set to `64`, and `Padding` set to `1,1,1,1`
3. `reluLayer`
4. `maxPooling2dLayer` with `PoolSize` set to `2,2`, `Stride` set to `2,2`, and `Padding` set to `0,0,0,0`
5. `convolution2dLayer` with `FilterSize` set to `3,3`, `NumFilters` set to `64`, and `Padding` set to `1,1,1,1`
6. `reluLayer`
7. `transposedConv2dLayer` with `FilterSize` set to `4,4`, `NumFilters` set to `64`, `Stride` set to `2,2`, and `Cropping` set to `1,1,1,1`
8. `convolution2dLayer` with `FilterSize` set to `1,1`, `NumFilters` set to `2`, and `Padding` set to `0,0,0,0`
9. `softmaxLayer`
10. `pixelClassificationLayer`



You can also create this network at the command line and then import the network into Deep Network Designer using `deepNetworkDesigner(layers)`.

```
layers = [
    imageInputLayer([32 32 1])
```

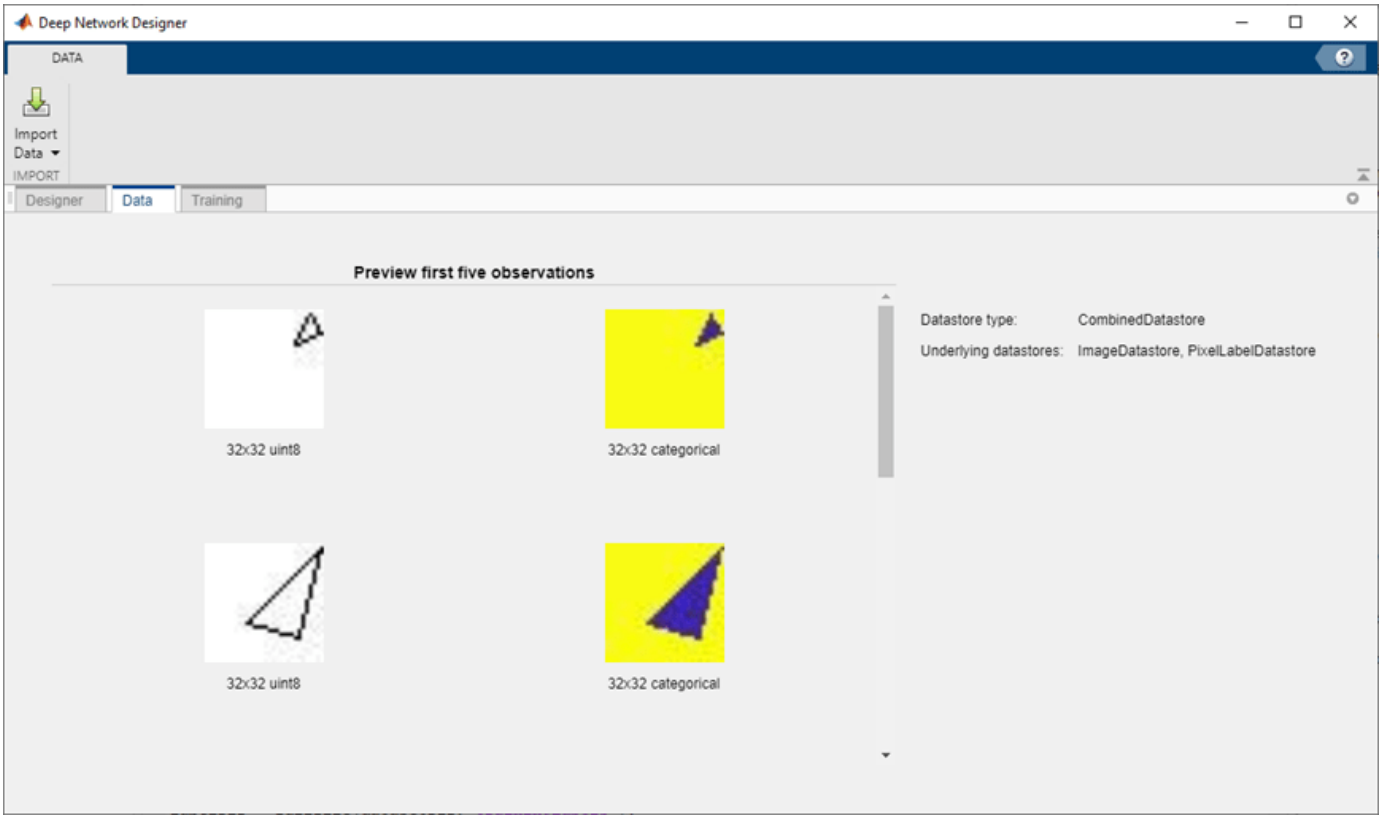
```
convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
reluLayer
maxPooling2dLayer([2,2],'Stride',[2,2])
convolution2dLayer([3,3],64,'Padding',[1,1,1,1])
reluLayer
transposedConv2dLayer([4,4],64,'Stride',[2,2],'Cropping',[1,1,1,1])
convolution2dLayer([1,1],2)
softmaxLayer
pixelClassificationLayer
];
```

This network is a simple semantic segmentation network based on a downsampling and upsampling design. For more information on constructing a semantic segmentation network, see “Create a Semantic Segmentation Network” (Computer Vision Toolbox).

Import Data

To import the training datastore, on the **Data** tab, select **Import Data** > **Import Datastore**. Select the `CombinedDatastore` object `cds` as the training data. For the validation data, select `None`. Import the training data by clicking **Import**.

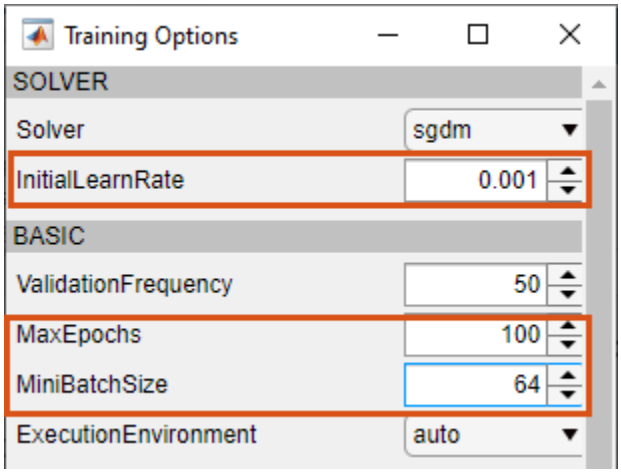
Deep Network Designer displays a preview of the imported semantic segmentation data. The preview displays the training images and the ground truth pixel labels. The network requires input images (left) and returns a classification for each pixel as either triangle or background (right).



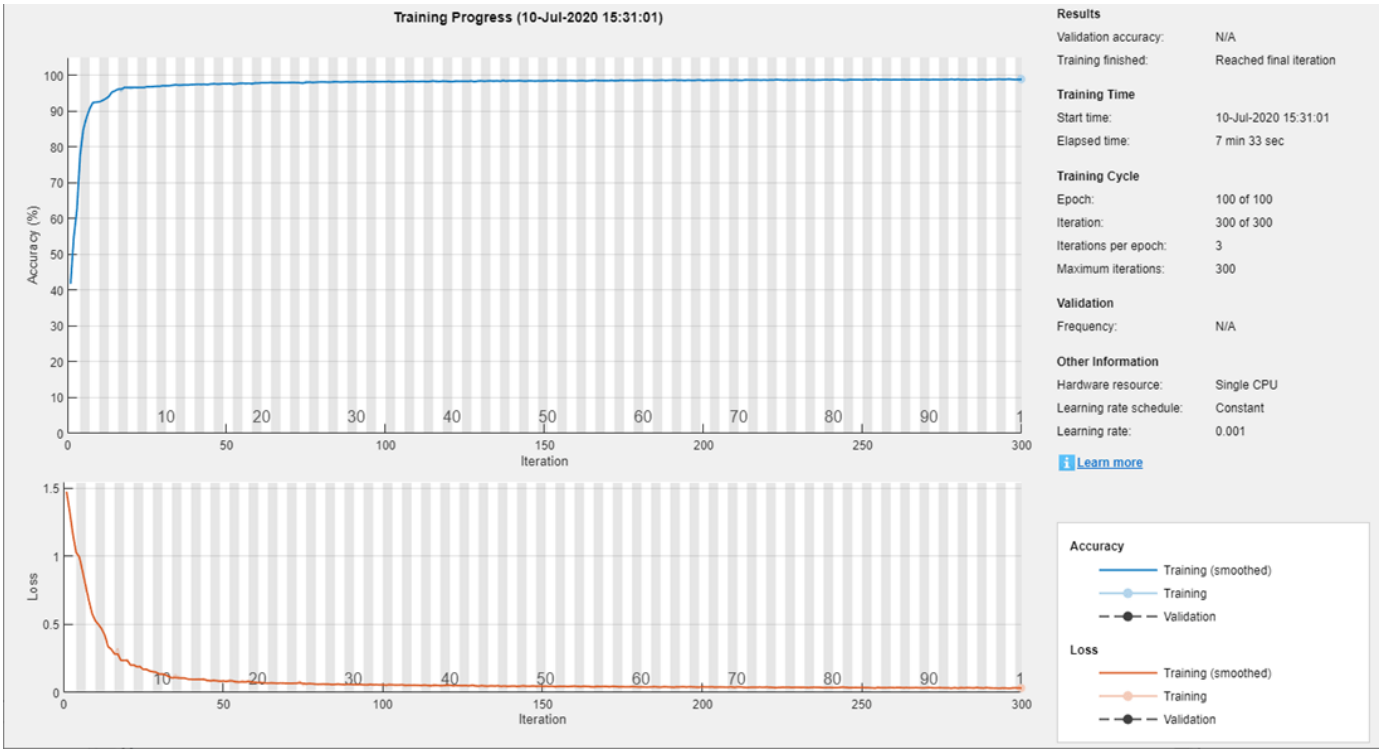
Train Network

Set the training options and train the network.

On the **Training** tab, click **Training Options**. Set **InitialLearnRate** to `0.001`, **MaxEpochs** to `100`, and **MiniBatchSize** to `64`. Set the training options by clicking **Close**.



Train the network by clicking **Train**.



Once training is complete, click **Export** to export the trained network to the workspace. The trained network is stored in the variable `trainedNetwork_1`.

Test Network

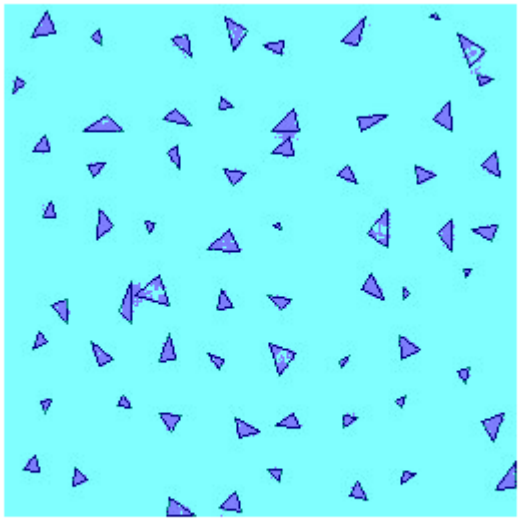
Make predictions using test data and the trained network.

Segment the test image using `semanticseg`. Display the labels over the image by using the `labeloverlay` function.

```
imgTest = imread('triangleTest.jpg');
testSeg = semanticseg(imgTest,trainedNetwork_1);
testImageSeg = labeloverlay(imgTest,testSeg);
```

Display the results.

```
figure
imshow(testImageSeg)
```



The network successfully labels the triangles in the test image.

The semantic segmentation network trained in this example is very simple. To construct more complex semantic segmentation networks, you can use the Computer Vision Toolbox functions `segnetLayers` (Computer Vision Toolbox), `deeplabv3plusLayers` (Computer Vision Toolbox), and `unetLayers` (Computer Vision Toolbox). For an example showing how to use the `deeplabv3plusLayers` function to create a DeepLab v3+ network, see "Semantic Segmentation With Deep Learning" (Computer Vision Toolbox).

See Also

Deep Network Designer | **Image Labeler** | `pixelLabelDatastore` | `semanticseg` | `labeloverlay` | `pixelClassificationLayer` | `trainingOptions` | `deeplabv3plusLayers` | `segnetLayers` | `unetLayers`

More About

- "Semantic Segmentation Using Deep Learning" on page 8-126
- "Getting Started with Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
- "Label Pixels for Semantic Segmentation" (Computer Vision Toolbox)

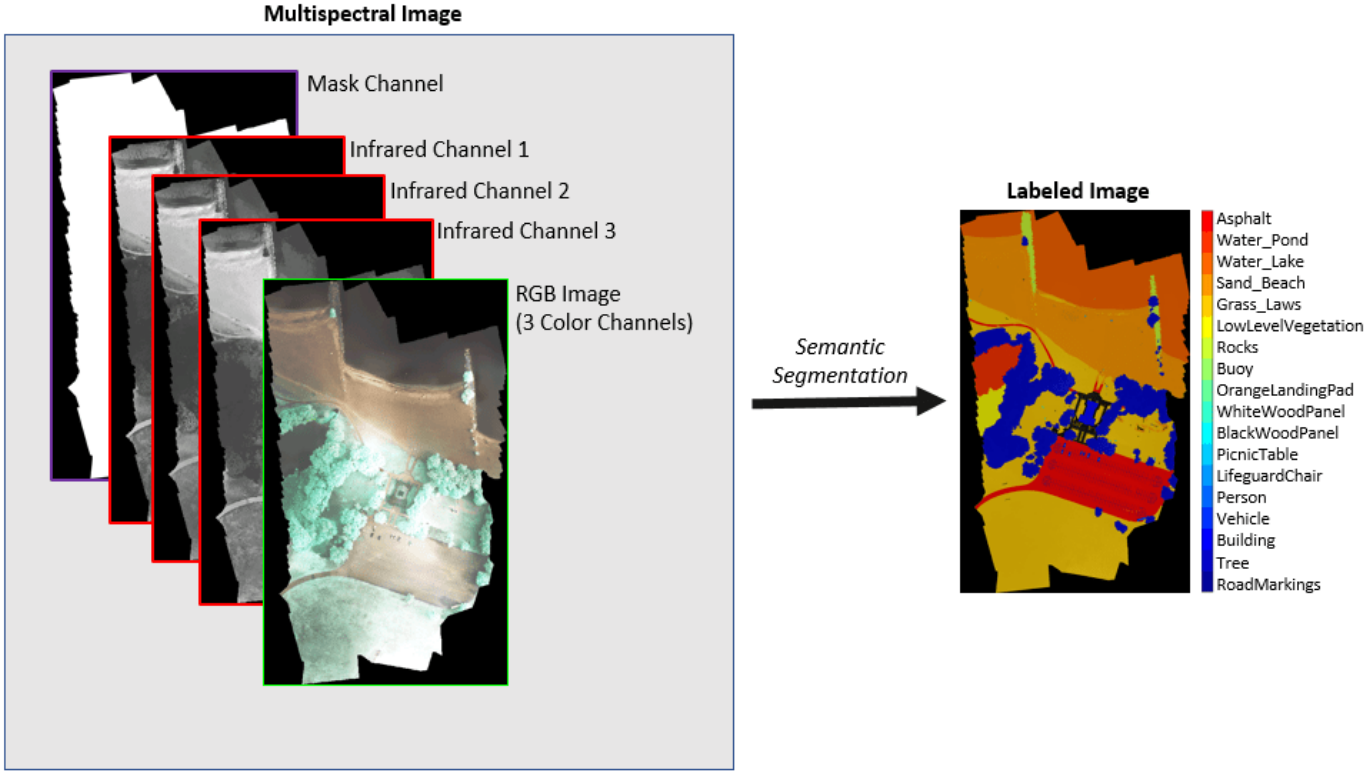
- “Pretrained Deep Neural Networks” on page 1-8
- “Get Started with Deep Network Designer”

Semantic Segmentation of Multispectral Images Using Deep Learning

This example shows how to perform semantic segmentation of a multispectral image with seven channels using a U-Net.

Semantic segmentation involves labeling each pixel in an image with a class. One application of semantic segmentation is tracking deforestation, which is the change in forest cover over time. Environmental agencies track deforestation to assess and quantify the environmental and ecological health of a region.

Deep learning based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One challenge is differentiating classes with similar visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with three near-infrared channels that provide a clearer separation of the classes.



This example shows how to use deep-learning-based semantic segmentation techniques to calculate the percentage vegetation cover in a region from a set of multispectral images.

Download Data

This example uses a high-resolution multispectral data set to train the network [1 on page 8-0]. The image set was captured using a drone over the Hamlin Beach State Park, NY. The data contains labeled training, validation, and test sets, with 18 object class labels. The size of the data file is ~3.0 GB.

Download the MAT-file version of the data set using the `downloadHamlinBeachMSIData` helper function. This function is attached to the example as a supporting file.

```
imageDir = tempdir;
url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
downloadHamlinBeachMSIData(url,imageDir);
```

Inspect Training Data

Load the data set into the workspace.

```
load(fullfile(imageDir,'rit18_data','rit18_data.mat'));
```

Examine the structure of the data.

```
whos train_data val_data test_data
```

```
  Name            Size                         Bytes  Class     Attributes

  test_data       7x12446x7654            1333663576  uint16
  train_data      7x9393x5642              741934284  uint16
  val_data        7x8833x6918              855493716  uint16
```

The multispectral image data is arranged as *numChannels*-by-*width*-by-*height* arrays. However, in MATLAB®, multichannel images are arranged as *width*-by-*height*-by-*numChannels* arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`. This function is attached to the example as a supporting file.

```
train_data = switchChannelsToThirdPlane(train_data);
val_data   = switchChannelsToThirdPlane(val_data);
test_data  = switchChannelsToThirdPlane(test_data);
```

Confirm that the data has the correct structure.

```
whos train_data val_data test_data
```

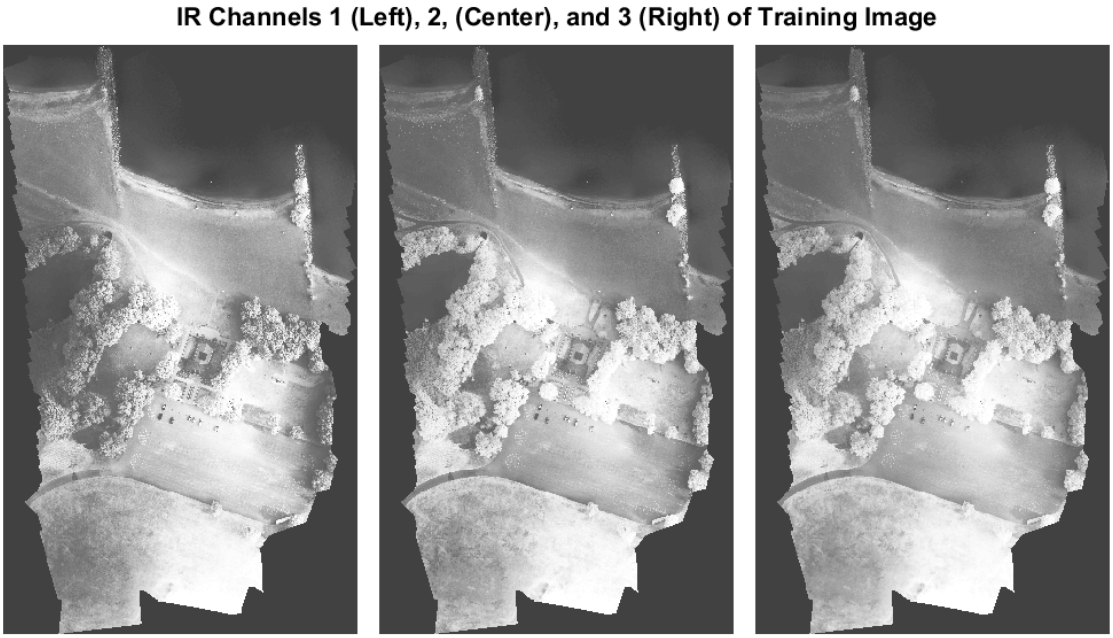
```
  Name               Size                         Bytes  Class     Attributes

  test_data       12446x7654x7            1333663576  uint16
  train_data       9393x5642x7             741934284  uint16
  val_data         8833x6918x7             855493716  uint16
```

The RGB color channels are the 3rd, 2nd, and 1st image channels. Display the color component of the training, validation, and test images as a montage. To make the images appear brighter on the screen, equalize their histograms by using the `histeq` (Image Processing Toolbox) function.

```
figure
montage(...
    {histeq(train_data(:,:,[3 2 1])), ...
    histeq(val_data(:,:,[3 2 1])), ...
    histeq(test_data(:,:,[3 2 1]))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('RGB Component of Training Image (Left), Validation Image (Center), and Test Image (Right)
```

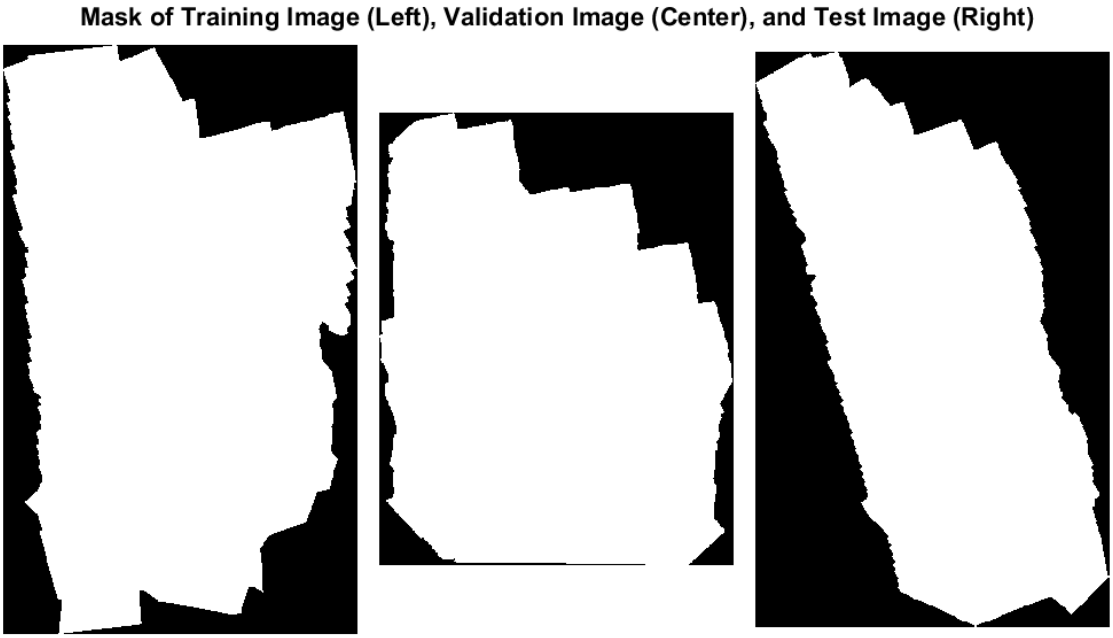
Display the last three histogram-equalized channels of the training data as a montage. These channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. For example, the trees near the center of the second channel image show more detail than the trees in the other two channels.

```
figure
montage(...
    {histeq(train_data(:,:,4)), ...
    histeq(train_data(:,:,5)), ...
    histeq(train_data(:,:,6))}, ...
    'BorderSize',10,'BackgroundColor','white')
title('IR Channels 1 (Left), 2, (Center), and 3 (Right) of Training Image')
```



Channel 7 is a mask that indicates the valid segmentation region. Display the mask for the training, validation, and test images.

```
figure
montage(...
    {train_data(:,:,7), ...
    val_data(:,:,7), ...
    test_data(:,:,7)}, ...
    'BorderSize',10,'BackgroundColor','white')
title('Mask of Training Image (Left), Validation Image (Center), and Test Image (Right)')
```



The labeled images contain the ground truth data for the segmentation, with each pixel assigned to one of the 18 classes. Get a list of the classes with their corresponding IDs.

```
disp(classes)
```

```
0. Other Class/Image Border
1. Road Markings
2. Tree
3. Building
4. Vehicle (Car, Truck, or Bus)
5. Person
6. Lifeguard Chair
7. Picnic Table
8. Black Wood Panel
9. White Wood Panel
10. Orange Landing Pad
11. Water Buoy
12. Rocks
13. Other Vegetation
14. Grass
15. Sand
16. Water (Lake)
17. Water (Pond)
18. Asphalt (Parking Lot/Walkway)
```

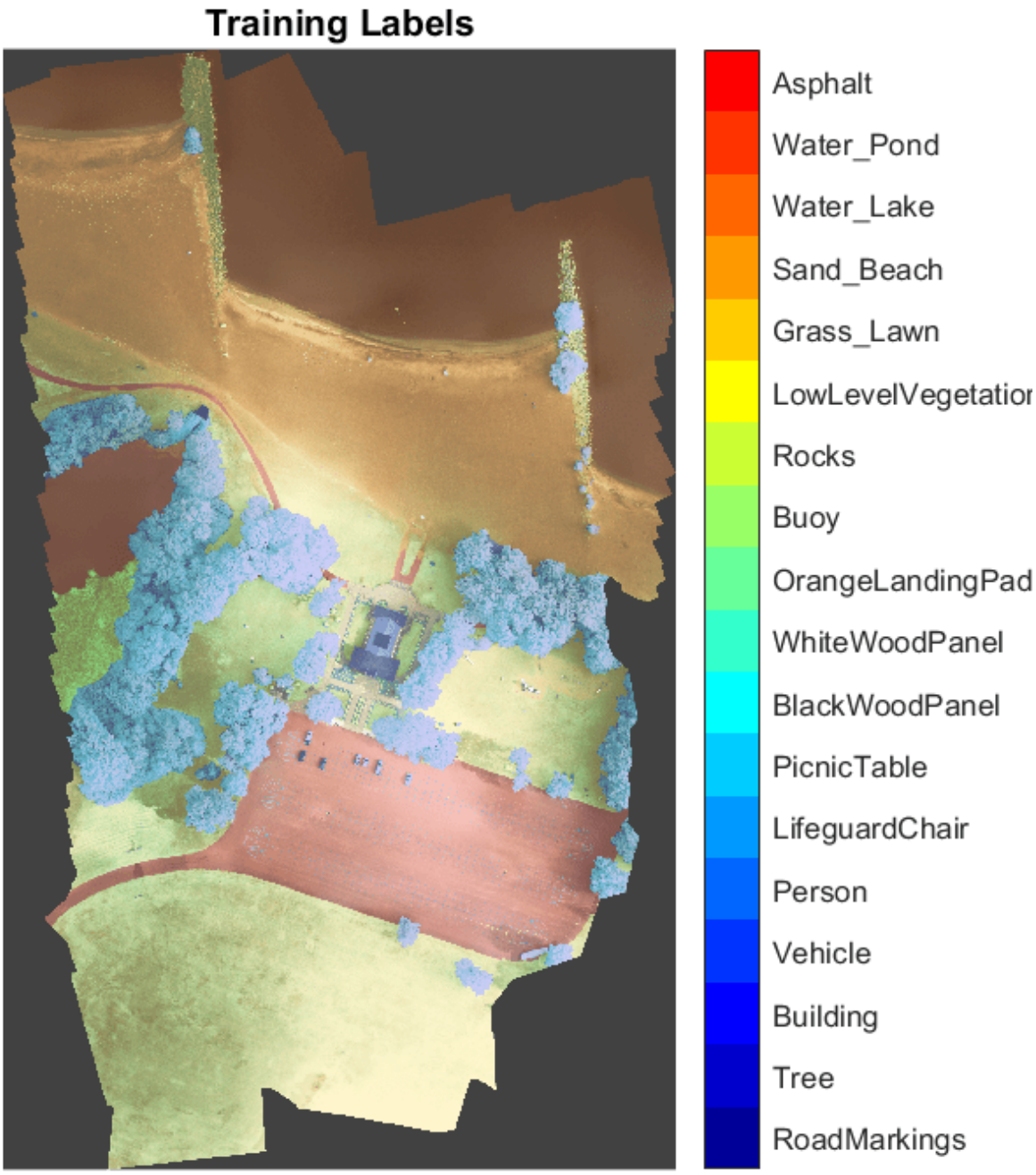
Create a vector of class names.

```
classNames = [ "RoadMarkings","Tree","Building","Vehicle","Person", ...
               "LifeguardChair","PicnicTable","BlackWoodPanel",...
```

```
    "WhiteWoodPanel","OrangeLandingPad","Buoy","Rocks",...
    "LowLevelVegetation","Grass_Lawn","Sand_Beach",...
    "Water_Lake","Water_Pond","Asphalt"];
```

Overlay the labels on the histogram-equalized RGB training image. Add a color bar to the image.

```
cmap = jet(numel(classNames));
B = labeloverlay(histeq(train_data(:,:,4:6)),train_labels,'Transparency',0.8,'Colormap',cmap);

figure
imshow(B)
title('Training Labels')
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
```


Training Labels
Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetatior
Rocks
Buoy
OrangeLandingPad
WhiteWoodPanel
BlackWoodPanel
PicnicTable
LifeguardChair
Person
Vehicle
Building
Tree
RoadMarkings

Save the training data as a MAT file and the training labels as a PNG file.

```
save('train_data.mat','train_data');
imwrite(train_labels,'train_labels.png');
```

Create Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts multiple corresponding random patches from an image datastore and pixel label datastore that contain ground truth images and pixel label data. Patching is a common technique to prevent running out of memory for large images and to effectively increase the amount of available training data.

Begin by storing the training images from `'train_data.mat'` in an `imageDatastore`. Because the MAT file format is a nonstandard image format, you must use a MAT file reader to enable reading the image data. You can use the helper MAT file reader, `matReader`, that extracts the first six channels from the training data and omits the last channel containing the mask. This function is attached to the example as a supporting file.

```
imds = imageDatastore('train_data.mat','FileExtensions','.mat','ReadFcn',@matReader);
```

Create a `pixelLabelDatastore` (Computer Vision Toolbox) to store the label patches containing the 18 labeled regions.

```
pixelLabelIds = 1:18;
pxds = pixelLabelDatastore('train_labels.png',classNames,pixelLabelIds);
```

Create a `randomPatchExtractionDatastore` (Image Processing Toolbox) from the image datastore and the pixel label datastore. Each mini-batch contains 16 patches of size 256-by-256 pixels. One thousand mini-batches are extracted at each iteration of the epoch.

```
dsTrain = randomPatchExtractionDatastore(imds,pxds,[256,256],'PatchesPerImage',16000);
```

The random patch extraction datastore `dsTrain` provides mini-batches of data to the network at each iteration of the epoch. Preview the datastore to explore the data.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
        InputImage         ResponsePixelLabelImage
    ___________________    _______________________

    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
    {256×256×6 uint16}     {256×256 categorical}
```

Create U-Net Network Layers

This example uses a variation of the U-Net network. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image [2 on page 8-0]. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

This example modifies the U-Net to use zero-padding in the convolutions, so that the input and the output to the convolutions have the same size. Use the helper function, `createUnet`, to create a U-Net with a few preselected hyperparameters. This function is attached to the example as a supporting file.

```
inputTileSize = [256,256,6];
lgraph = createUnet(inputTileSize);
disp(lgraph.Layers)
```

```
  58×1 Layer array with layers:

     1   'ImageInputLayer'                          Image Input                 256×256×6 images w
     2   'Encoder-Section-1-Conv-1'                 Convolution                 64 3×3×6 convoluti
     3   'Encoder-Section-1-ReLU-1'                 ReLU                        ReLU
     4   'Encoder-Section-1-Conv-2'                 Convolution                 64 3×3×64 convolut
     5   'Encoder-Section-1-ReLU-2'                 ReLU                        ReLU
     6   'Encoder-Section-1-MaxPool'                Max Pooling                 2×2 max pooling wi
     7   'Encoder-Section-2-Conv-1'                 Convolution                 128 3×3×64 convolu
     8   'Encoder-Section-2-ReLU-1'                 ReLU                        ReLU
     9   'Encoder-Section-2-Conv-2'                 Convolution                 128 3×3×128 convol
    10   'Encoder-Section-2-ReLU-2'                 ReLU                        ReLU
    11   'Encoder-Section-2-MaxPool'                Max Pooling                 2×2 max pooling wi
    12   'Encoder-Section-3-Conv-1'                 Convolution                 256 3×3×128 convol
    13   'Encoder-Section-3-ReLU-1'                 ReLU                        ReLU
    14   'Encoder-Section-3-Conv-2'                 Convolution                 256 3×3×256 convol
    15   'Encoder-Section-3-ReLU-2'                 ReLU                        ReLU
    16   'Encoder-Section-3-MaxPool'                Max Pooling                 2×2 max pooling wi
    17   'Encoder-Section-4-Conv-1'                 Convolution                 512 3×3×256 convol
    18   'Encoder-Section-4-ReLU-1'                 ReLU                        ReLU
    19   'Encoder-Section-4-Conv-2'                 Convolution                 512 3×3×512 convol
    20   'Encoder-Section-4-ReLU-2'                 ReLU                        ReLU
    21   'Encoder-Section-4-DropOut'                Dropout                     50% dropout
    22   'Encoder-Section-4-MaxPool'                Max Pooling                 2×2 max pooling wi
    23   'Mid-Conv-1'                               Convolution                 1024 3×3×512 convo
    24   'Mid-ReLU-1'                               ReLU                        ReLU
    25   'Mid-Conv-2'                               Convolution                 1024 3×3×1024 conv
    26   'Mid-ReLU-2'                               ReLU                        ReLU
    27   'Mid-DropOut'                              Dropout                     50% dropout
    28   'Decoder-Section-1-UpConv'                 Transposed Convolution      512 2×2×1024 trans
    29   'Decoder-Section-1-UpReLU'                 ReLU                        ReLU
    30   'Decoder-Section-1-DepthConcatenation'     Depth concatenation         Depth concatenatio
    31   'Decoder-Section-1-Conv-1'                 Convolution                 512 3×3×1024 convo
    32   'Decoder-Section-1-ReLU-1'                 ReLU                        ReLU
    33   'Decoder-Section-1-Conv-2'                 Convolution                 512 3×3×512 convol
    34   'Decoder-Section-1-ReLU-2'                 ReLU                        ReLU
    35   'Decoder-Section-2-UpConv'                 Transposed Convolution      256 2×2×512 transp
    36   'Decoder-Section-2-UpReLU'                 ReLU                        ReLU
    37   'Decoder-Section-2-DepthConcatenation'     Depth concatenation         Depth concatenatio
    38   'Decoder-Section-2-Conv-1'                 Convolution                 256 3×3×512 convol
    39   'Decoder-Section-2-ReLU-1'                 ReLU                        ReLU
    40   'Decoder-Section-2-Conv-2'                 Convolution                 256 3×3×256 convol
    41   'Decoder-Section-2-ReLU-2'                 ReLU                        ReLU
    42   'Decoder-Section-3-UpConv'                 Transposed Convolution      128 2×2×256 transp
    43   'Decoder-Section-3-UpReLU'                 ReLU                        ReLU
    44   'Decoder-Section-3-DepthConcatenation'     Depth concatenation         Depth concatenatio
    45   'Decoder-Section-3-Conv-1'                 Convolution                 128 3×3×256 convol
    46   'Decoder-Section-3-ReLU-1'                 ReLU                        ReLU
    47   'Decoder-Section-3-Conv-2'                 Convolution                 128 3×3×128 convol
```

```
    48   'Decoder-Section-3-ReLU-2'              ReLU                          ReLU
    49   'Decoder-Section-4-UpConv'              Transposed Convolution        64 2×2×128 transpo
    50   'Decoder-Section-4-UpReLU'              ReLU                          ReLU
    51   'Decoder-Section-4-DepthConcatenation'  Depth concatenation           Depth concatenatio
    52   'Decoder-Section-4-Conv-1'              Convolution                   64 3×3×128 convolu
    53   'Decoder-Section-4-ReLU-1'              ReLU                          ReLU
    54   'Decoder-Section-4-Conv-2'              Convolution                   64 3×3×64 convolut
    55   'Decoder-Section-4-ReLU-2'              ReLU                          ReLU
    56   'Final-ConvolutionLayer'                Convolution                   18 1×1×64 convolut
    57   'Softmax-Layer'                         Softmax                       softmax
    58   'Segmentation-Layer'                    Pixel Classification Layer    Cross-entropy loss
```

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` function.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by specifying `'GradientThreshold'` as `0.05`, and specify `'GradientThresholdMethod'` to use the L2-norm of the gradients.

```
initialLearningRate = 0.05;
maxEpochs = 150;
minibatchSize = 16;
l2reg = 0.0001;

options = trainingOptions('sgdm',...
    'InitialLearnRate',initialLearningRate, ...
    'Momentum',0.9,...
    'L2Regularization',l2reg,...
    'MaxEpochs',maxEpochs,...
    'MiniBatchSize',minibatchSize,...
    'LearnRateSchedule','piecewise',...
    'Shuffle','every-epoch',...
    'GradientThresholdMethod','l2norm',...
    'GradientThreshold',0.05, ...
    'Plots','training-progress', ...
    'VerboseFrequency',20);
```

Train the Network

By default, the example downloads a pretrained version of U-Net for this dataset using the `downloadTrainedUnet` helper function. This function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without having to wait for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model by using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 20 hours on an NVIDIA Titan X.

```
doTraining = false;
if doTraining
```

```
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    save(strcat("multispectralUnet-",modelDateTime,"-Epoch-",num2str(maxEpochs),".mat"),'net');

else
    trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
    downloadTrainedUnet(trainedUnet_url,imageDir);
    load(fullfile(imageDir,'trainedUnet','multispectralUnet.mat'));
end
```

You can now use the U-Net to semantically segment the multispectral image.

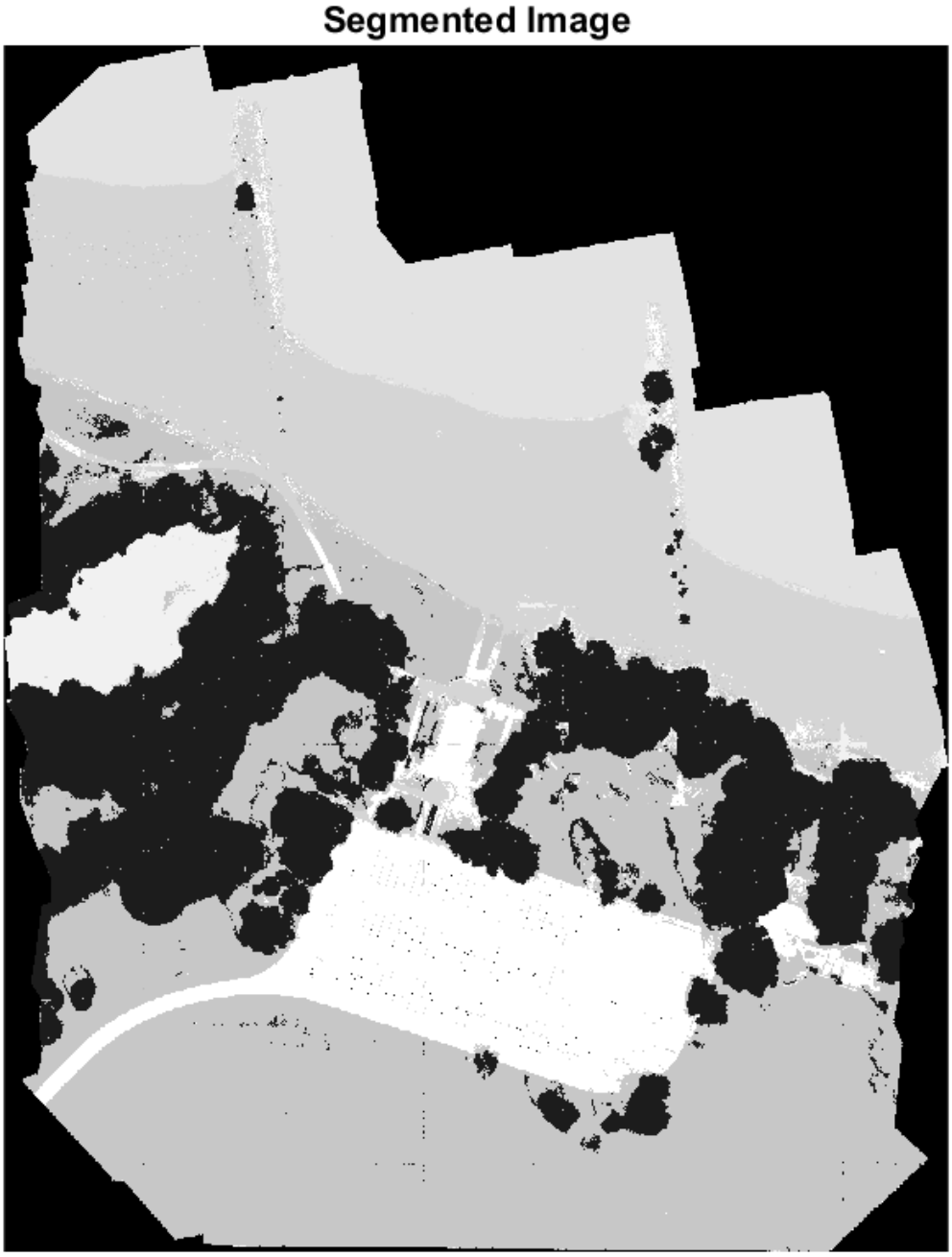
Predict Results on Test Data

To perform the forward pass on the trained network, use the helper function, `segmentImage`, with the validation data set. This function is attached to the example as a supporting file. `segmentImage` performs segmentation on image patches using the `semanticseg` (Computer Vision Toolbox) function.

```
predictPatchSize = [1024 1024];
segmentedImage = segmentImage(val_data,net,predictPatchSize);
```

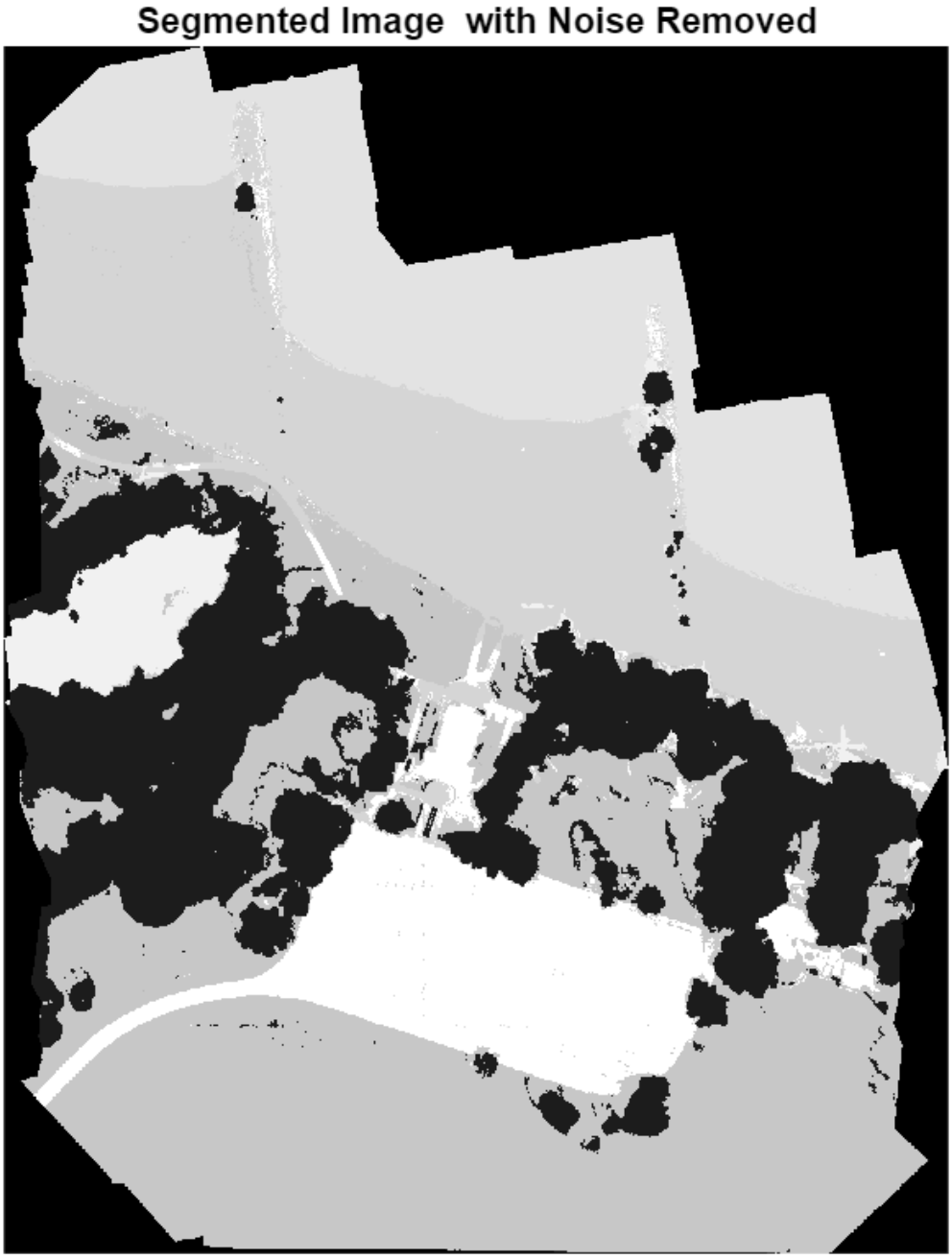
To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the validation data.

```
segmentedImage = uint8(val_data(:,:,7)~=0) .* segmentedImage;

figure
imshow(segmentedImage,[])
title('Segmented Image')
```



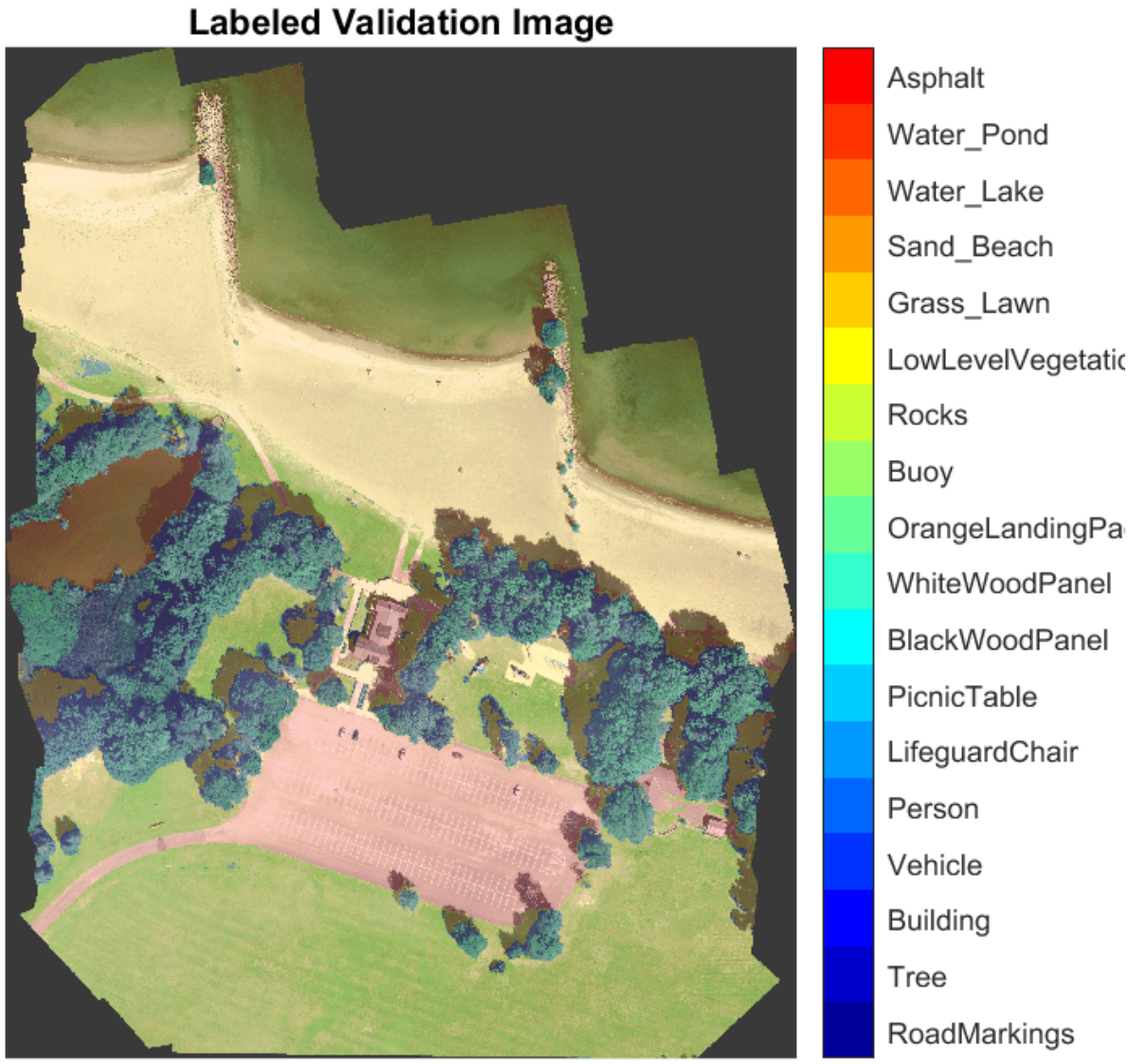
The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` (Image Processing Toolbox) function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
segmentedImage = medfilt2(segmentedImage,[7,7]);
imshow(segmentedImage,[]);
title('Segmented Image  with Noise Removed')
```



Overlay the segmented image on the histogram-equalized RGB validation image.

```
B = labeloverlay(histeq(val_data(:,:,[3 2 1])),segmentedImage,'Transparency',0.8,'Colormap',cmap)

figure
imshow(B)
title('Labeled Validation Image')
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
```



Save the segmented image and ground truth labels as PNG files. These will be used to compute accuracy metrics.

```
imwrite(segmentedImage,'results.png');
imwrite(val_labels,'gtruth.png');
```

Quantify Segmentation Accuracy

Create a `pixelLabelDatastore` (Computer Vision Toolbox) for the segmentation results and the ground truth labels.

```
pxdsResults = pixelLabelDatastore('results.png',classNames,pixelLabelIds);
pxdsTruth = pixelLabelDatastore('gtruth.png',classNames,pixelLabelIds);
```

Measure the global accuracy of the semantic segmentation by using the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function.

```
ssm = evaluateSemanticSegmentation(pxdsResults,pxdsTruth,'Metrics','global-accuracy');
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy.
* Processed 1 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy
    ______________

       0.90698
```

The global accuracy score indicates that just over 90% of the pixels are classified correctly.

Calculate Extent of Vegetation Cover

The final goal of this example is to calculate the extent of vegetation cover in the multispectral image.

Find the number of pixels labeled vegetation. The label IDs 2 ("Trees"), 13 ("LowLevelVegetation"), and 14 ("Grass_Lawn") are the vegetation classes. Also find the total number of valid pixels by summing the pixels in the ROI of the mask image.

```
vegetationClassIds = uint8([2,13,14]);
vegetationPixels = ismember(segmentedImage(:),vegetationClassIds);
validPixels = (segmentedImage~=0);

numVegetationPixels = sum(vegetationPixels(:));
numValidPixels = sum(validPixels(:));
```

Calculate the percentage of vegetation cover by dividing the number of vegetation pixels by the number of valid pixels.

```
percentVegetationCover = (numVegetationPixels/numValidPixels)*100;
fprintf('The percentage of vegetation cover is %3.2f%%.',percentVegetationCover);
```

```
The percentage of vegetation cover is 51.72%.
```

References

[1] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918. 2017.

[2] Ronneberger, O., P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." CoRR, abs/1505.04597. 2015.

See Also

`trainingOptions` | `trainNetwork` | `randomPatchExtractionDatastore` | `pixelLabelDatastore` | `semanticseg` | `evaluateSemanticSegmentation` | `imageDatastore` | `histeq` | `unetLayers`

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation Using Deep Learning” on page 8-126
- “Semantic Segmentation Using Dilated Convolutions” on page 8-143
- “Datastores for Deep Learning” on page 19-2

External Websites

- https://github.com/rmkemker/RIT-18

3-D Brain Tumor Segmentation Using Deep Learning

This example shows how to train a 3-D U-Net neural network and perform semantic segmentation of brain tumors from 3-D medical images.

Semantic segmentation involves labeling each pixel in an image or voxel of a 3-D volume with a class. This example illustrates the use of deep learning methods to perform binary semantic segmentation of brain tumors in magnetic resonance imaging (MRI) scans. In this binary segmentation, each pixel is labeled as tumor or background.

This example performs brain tumor segmentation using a 3-D U-Net architecture [1 on page 8-0]. U-Net is a fast, efficient and simple network that has become popular in the semantic segmentation domain.

One challenge of medical image segmentation is the amount of memory needed to store and process 3-D volumes. Training a network on the full input volume is impractical due to GPU resource constraints. This example solves the problem by training the network on image patches. The example uses an overlap-tile strategy to stitch test patches into a complete segmented test volume. The example avoids border artifacts by using the valid part of the convolution in the neural network [5 on page 8-0].

A second challenge of medical image segmentation is class imbalance in the data that hampers training when using conventional cross entropy loss. This example solves the problem by using a weighted multiclass Dice loss function [4 on page 8-0]. Weighting the classes helps to counter the influence of larger regions on the Dice score, making it easier for the network to learn how to segment smaller regions.

Download Training, Validation, and Test Data

This example uses the BraTS data set [2 on page 8-0]. The BraTS data set contains MRI scans of brain tumors, namely gliomas, which are the most common primary brain malignancies. The size of the data file is ~7 GB. If you do not want to download the BraTS data set, then go directly to the Download Pretrained Network and Sample Test Set on page 8-0 section in this example.

Create a directory to store the BraTS data set.

```
imageDir = fullfile(tempdir,'BraTS');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

To download the BraTS data, go to the Medical Segmentation Decathlon website and click the "Download Data" link. Download the "Task01_BrainTumour.tar" file [3 on page 8-0]. Unzip the TAR file into the directory specified by the `imageDir` variable. When unzipped successfully, `imageDir` will contain a directory named `Task01_BrainTumour` that has three subdirectories: `imagesTr`, `imagesTs`, and `labelsTr`.

The data set contains 750 4-D volumes, each representing a stack of 3-D images. Each 4-D volume has size 240-by-240-by-155-by-4, where the first three dimensions correspond to height, width, and depth of a 3-D volumetric image. The fourth dimension corresponds to different scan modalities. The data set is divided into 484 training volumes with voxel labels and 266 test volumes, The test volumes do not have labels so this example does not use the test data. Instead, the example splits the 484 training volumes into three independent sets that are used for training, validation, and testing.

Preprocess Training and Validation Data

To train the 3-D U-Net network more efficiently, preprocess the MRI data using the helper function `preprocessBraTSdataset`. This function is attached to the example as a supporting file.

The helper function performs these operations:

- Crop the data to a region containing primarily the brain and tumor. Cropping the data reduces the size of data while retaining the most critical part of each MRI volume and its corresponding labels.
- Normalize each modality of each volume independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.
- Split the 484 training volumes into 400 training, 29 validation, and 55 test sets.

Preprocessing the data can take about 30 minutes to complete.

```
sourceDataLoc = [imageDir filesep 'Task01_BrainTumour'];
preprocessDataLoc = fullfile(tempdir,'BraTS','preprocessedDataset');
preprocessBraTSdataset(preprocessDataLoc,sourceDataLoc);
```

Create Random Patch Extraction Datastore for Training and Validation

Use a random patch extraction datastore to feed the training data to the network and to validate the training progress. This datastore extracts random patches from ground truth images and corresponding pixel label data. Patching is a common technique to prevent running out of memory when training with arbitrarily large volumes.

Create an `imageDatastore` to store the 3-D image data. Because the MAT-file format is a nonstandard image format, you must use a MAT-file reader to enable reading the image data. You can use the helper MAT-file reader, `matRead`. This function is attached to the example as a supporting file.

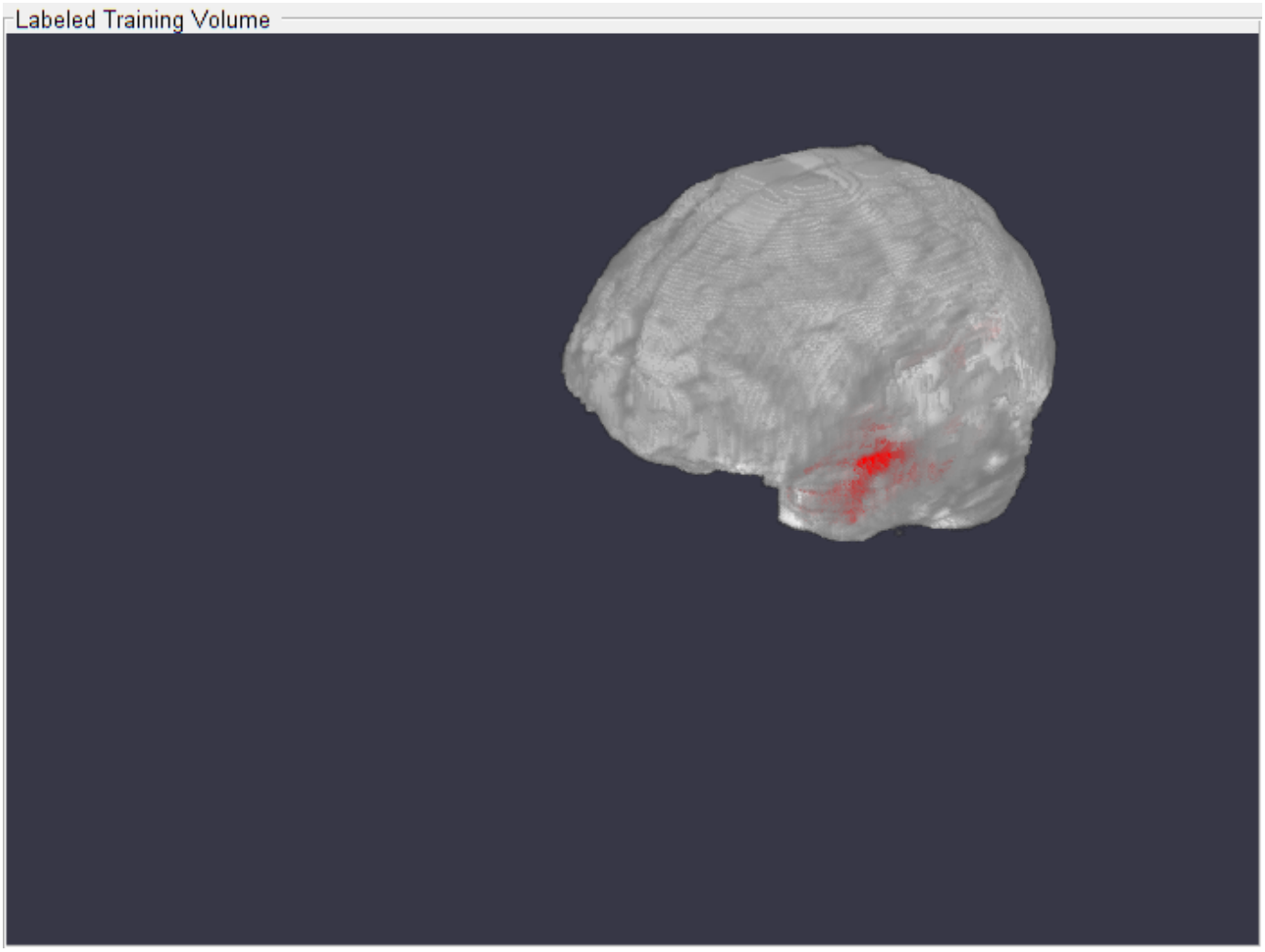
```
volReader = @(x) matRead(x);
volLoc = fullfile(preprocessDataLoc,'imagesTr');
volds = imageDatastore(volLoc, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Create a `pixelLabelDatastore` (Computer Vision Toolbox) to store the labels.

```
lblLoc = fullfile(preprocessDataLoc,'labelsTr');
classNames = ["background","tumor"];
pixelLabelID = [0 1];
pxds = pixelLabelDatastore(lblLoc,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Preview one image volume and label. Display the labeled volume using the `labelvolshow` (Image Processing Toolbox) function. Make the background fully transparent by setting the visibility of the background label (`1`) to `0`.

```
volume = preview(volds);
label = preview(pxds);

viewPnl = uipanel(figure,'Title','Labeled Training Volume');
hPred = labelvolshow(label,volume(:,:,:,1),'Parent',viewPnl, ...
    'LabelColor',[0 0 0;1 0 0]);
hPred.LabelVisibility(1) = 0;
```



Create a `randomPatchExtractionDatastore` (Image Processing Toolbox) that contains the training image and pixel label data. Specify a patch size of 132-by-132-by-132 voxels. Specify `'PatchesPerImage'` to extract 16 randomly positioned patches from each pair of volumes and labels during training. Specify a mini-batch size of 8.

```
patchSize = [132 132 132];
patchPerImage = 16;
miniBatchSize = 8;
patchds = randomPatchExtractionDatastore(volds,pxds,patchSize, ...
    'PatchesPerImage',patchPerImage);
patchds.MiniBatchSize = miniBatchSize;
```

Follow the same steps to create a `randomPatchExtractionDatastore` that contains the validation image and pixel label data. You can use validation data to evaluate whether the network is continuously learning, underfitting, or overfitting as time progresses.

```
volLocVal = fullfile(preprocessDataLoc,'imagesVal');
voldsVal = imageDatastore(volLocVal, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

```
lblLocVal = fullfile(preprocessDataLoc,'labelsVal');
pxdsVal = pixelLabelDatastore(lblLocVal,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);

dsVal = randomPatchExtractionDatastore(voldsVal,pxdsVal,patchSize, ...
    'PatchesPerImage',patchPerImage);
dsVal.MiniBatchSize = miniBatchSize;
```

Set Up 3-D U-Net Layers

This example uses the 3-D U-Net network [1 on page 8-0]. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. A batch normalization layer is introduced before each ReLU layer. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

Create a default 3-D U-Net network by using the `unetLayers` (Computer Vision Toolbox) function. Specify two class segmentation. Also specify valid convolution padding to avoid border artifacts when using the overlap-tile strategy for prediction of the test volumes.

```
numChannels = 4;
inputPatchSize = [patchSize numChannels];
numClasses = 2;
[lgraph,outPatchSize] = unet3dLayers(inputPatchSize,numClasses,'ConvolutionPadding','valid');
```

Augment the training and validation data by using the `transform` function with custom preprocessing operations specified by the helper function `augmentAndCrop3dPatch`. This function is attached to the example as a supporting file.

The `augmentAndCrop3dPatch` function performs these operations:

1. Randomly rotate and reflect training data to make the training more robust. The function does not rotate or reflect validation data.
2. Crop response patches to the output size of the network, 44-by-44-by-44 voxels.

```
dataSource = 'Training';
dsTrain = transform(patchds,@(patchIn)augmentAndCrop3dPatch(patchIn,outPatchSize,dataSource));

dataSource = 'Validation';
dsVal = transform(dsVal,@(patchIn)augmentAndCrop3dPatch(patchIn,outPatchSize,dataSource));
```

To better segment smaller tumor regions and reduce the influence of larger background regions, this example uses a `dicePixelClassificationLayer` (Computer Vision Toolbox). Replace the pixel classification layer with the Dice pixel classification layer.

```
outputLayer = dicePixelClassificationLayer('Name','Output');
lgraph = replaceLayer(lgraph,'Segmentation-Layer',outputLayer);
```

The data has already been normalized in the Preprocess Training and Validation Data on page 8-0 section of this example. Data normalization in the `image3dInputLayer` is unnecessary, so replace the input layer with an input layer that does not have data normalization.

```
inputLayer = image3dInputLayer(inputPatchSize,'Normalization','none','Name','ImageInputLayer');
lgraph = replaceLayer(lgraph,'ImageInputLayer',inputLayer);
```

Alternatively, you can modify the 3-D U-Net network by using Deep Network Designer App from Deep Learning Toolbox™.

Plot the graph of the updated 3-D U-Net network.

```
analyzeNetwork(lgraph)
```

Specify Training Options

Train the network using the `adam` optimization solver. Specify the hyperparameter settings using the `trainingOptions` function. The initial learning rate is set to 5e-4 and gradually decreases over the span of training. You can experiment with the `MiniBatchSize` property based on your GPU memory. To maximize GPU memory utilization, favor large input patches over a large batch size. Note that batch normalization layers are less effective for smaller values of `MiniBatchSize`. Tune the initial learning rate based on the `MiniBatchSize`.

```
options = trainingOptions('adam', ...
    'MaxEpochs',50, ...
    'InitialLearnRate',5e-4, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',5, ...
    'LearnRateDropFactor',0.95, ...
    'ValidationData',dsVal, ...
    'ValidationFrequency',400, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'MiniBatchSize',miniBatchSize);
```

Download Pretrained Network and Sample Test Set

Download a pretrained version of 3-D U-Net and five sample test volumes and their corresponding labels from the BraTS data set [3 on page 8-0]. The pretrained model and sample data enable you to perform segmentation on test data without downloading the full data set or waiting for the network to train.

```
trained3DUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/brainTumor3DUNetValid.ma
sampleData_url = 'https://www.mathworks.com/supportfiles/vision/data/sampleBraTSTestSetValid.tar

imageDir = fullfile(tempdir,'BraTS');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end

downloadTrained3DUnetSampleData(trained3DUnet_url,sampleData_url,imageDir);
```

Train Network

By default, the example loads a pretrained 3-D U-Net network. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 30 hours on a multi-GPU system with 4 NVIDIA™ Titan Xp GPUs and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    save(strcat("trained3DUNet-",modelDateTime,"-Epoch-",num2str(options.MaxEpochs),".mat"),'net
else
    inputPatchSize = [132 132 132 4];
    outPatchSize = [44 44 44 2];
    load(fullfile(imageDir,'trained3DUNet','brainTumor3DUNetValid.mat'));
end
```

Perform Segmentation of Test Data

A GPU is highly recommended for performing semantic segmentation of the image volumes (requires Parallel Computing Toolbox™).

Select the source of test data that contains ground truth volumes and labels for testing. If you keep the `useFullTestSet` variable in the following code as `false`, then the example uses five volumes for testing. If you set the `useFullTestSet` variable to `true`, then the example uses 55 test images selected from the full data set.

```
useFullTestSet = false;
if useFullTestSet
    volLocTest = fullfile(preprocessDataLoc,'imagesTest');
    lblLocTest = fullfile(preprocessDataLoc,'labelsTest');
else
    volLocTest = fullfile(imageDir,'sampleBraTSTestSetValid','imagesTest');
    lblLocTest = fullfile(imageDir,'sampleBraTSTestSetValid','labelsTest');
    classNames = ["background","tumor"];
    pixelLabelID = [0 1];
end
```

The `voldsTest` variable stores the ground truth test images. The `pxdsTest` variable stores the ground truth labels.

```
volReader = @(x) matRead(x);
voldsTest = imageDatastore(volLocTest, ...
    'FileExtensions','.mat','ReadFcn',volReader);
pxdsTest = pixelLabelDatastore(lblLocTest,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',volReader);
```

Use the overlap-tile strategy to predict the labels for each test volume. Each test volume is padded to make the input size a multiple of the output size of the network and compensates for the effects of valid convolution. The overlap-tile algorithm selects overlapping patches, predicts the labels for each patch by using the `semanticseg` (Computer Vision Toolbox) function, and then recombines the patches.

```
id = 1;
while hasdata(voldsTest)
    disp(['Processing test volume ' num2str(id)]);

    tempGroundTruth = read(pxdsTest);
    groundTruthLabels{id} = tempGroundTruth{1};
    vol{id} = read(voldsTest);

    % Use reflection padding for the test image.
    % Avoid padding of different modalities.
```

```
    volSize = size(vol{id},(1:3));
    padSizePre  = (inputPatchSize(1:3)-outPatchSize(1:3))/2;
    padSizePost = (inputPatchSize(1:3)-outPatchSize(1:3))/2 + (outPatchSize(1:3)-mod(volSize,outP
    volPaddedPre = padarray(vol{id},padSizePre,'symmetric','pre');
    volPadded = padarray(volPaddedPre,padSizePost,'symmetric','post');
    [heightPad,widthPad,depthPad,~] = size(volPadded);
    [height,width,depth,~] = size(vol{id});

    tempSeg = categorical(zeros([height,width,depth],'uint8'),[0;1],classNames);

    % Overlap-tile strategy for segmentation of volumes.
    for k = 1:outPatchSize(3):depthPad-inputPatchSize(3)+1
        for j = 1:outPatchSize(2):widthPad-inputPatchSize(2)+1
            for i = 1:outPatchSize(1):heightPad-inputPatchSize(1)+1
                patch = volPadded( i:i+inputPatchSize(1)-1,...
                    j:j+inputPatchSize(2)-1,...
                    k:k+inputPatchSize(3)-1,:);
                patchSeg = semanticseg(patch,net);
                tempSeg(i:i+outPatchSize(1)-1, ...
                    j:j+outPatchSize(2)-1, ...
                    k:k+outPatchSize(3)-1) = patchSeg;
            end
        end
    end

    % Crop out the extra padded region.
    tempSeg = tempSeg(1:height,1:width,1:depth);

    % Save the predicted volume result.
    predictedLabels{id} = tempSeg;
    id=id+1;
end
```

Compare Ground Truth Against Network Prediction

Select one of the test images to evaluate the accuracy of the semantic segmentation. Extract the first modality from the 4-D volumetric data and store this 3-D volume in the variable `vol3d`.

```
volId = 1;
vol3d = vol{volId}(:,:,:,1);
```

Display in a montage the center slice of the ground truth and predicted labels along the depth direction.

```
zID = size(vol3d,3)/2;
zSliceGT = labeloverlay(vol3d(:,:,zID),groundTruthLabels{volId}(:,:,zID));
zSlicePred = labeloverlay(vol3d(:,:,zID),predictedLabels{volId}(:,:,zID));

figure
montage({zSliceGT,zSlicePred},'Size',[1 2],'BorderSize',5)
title('Labeled Ground Truth (Left) vs. Network Prediction (Right)')
```

Display the ground-truth labeled volume using the `labelvolshow` (Image Processing Toolbox) function. Make the background fully transparent by setting the visibility of the background label (`1`) to `0`. Because the tumor is inside the brain tissue, make some of the brain voxels transparent, so that the tumor is visible. To make some brain voxels transparent, specify the volume threshold as a number in the range [0, 1]. All normalized volume intensities below this threshold value are fully

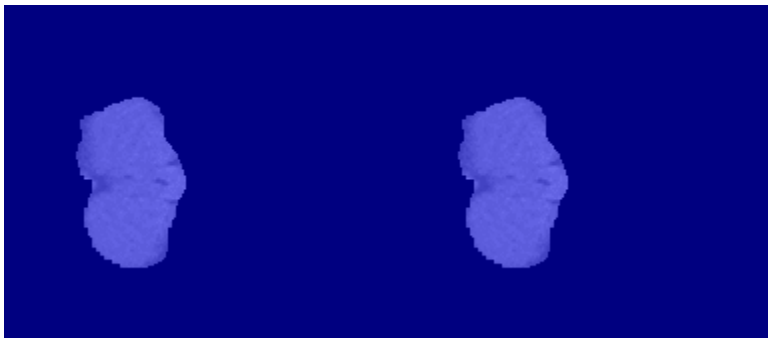
transparent. This example sets the volume threshold as less than 1 so that some brain pixels remain visible, to give context to the spatial location of the tumor inside the brain.

```
viewPnlTruth = uipanel(figure,'Title','Ground-Truth Labeled Volume');
hTruth = labelvolshow(groundTruthLabels{volId},vol3d,'Parent',viewPnlTruth, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
hTruth.LabelVisibility(1) = 0;
```

For the same volume, display the predicted labels.

```
viewPnlPred = uipanel(figure,'Title','Predicted Labeled Volume');
hPred = labelvolshow(predictedLabels{volId},vol3d,'Parent',viewPnlPred, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
hPred.LabelVisibility(1) = 0;
```

This image shows the result of displaying slices sequentially across the one of the volume. The labeled ground truth is on the left and the network prediction is on the right.



Quantify Segmentation Accuracy

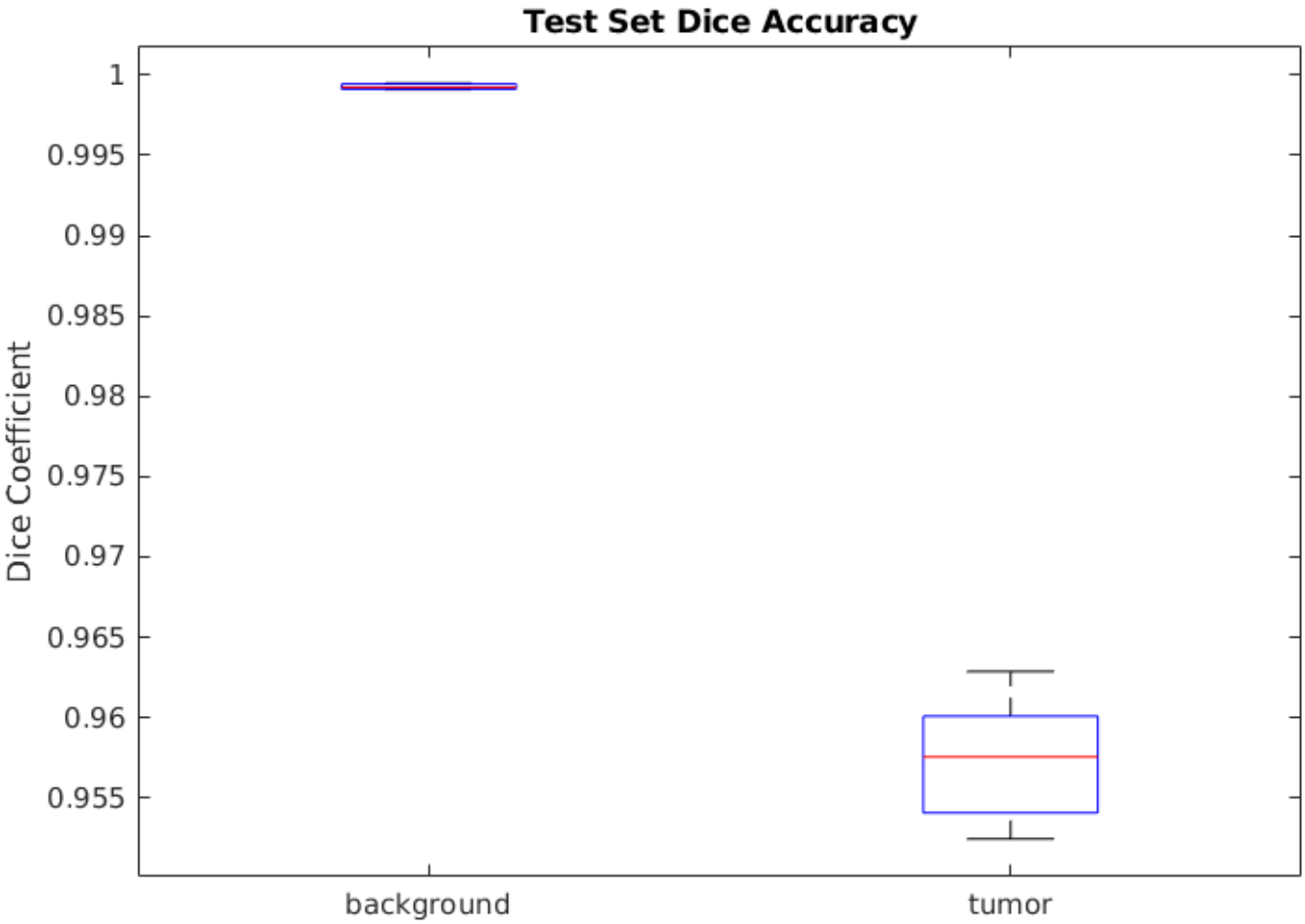
Measure the segmentation accuracy using the `dice` (Image Processing Toolbox) function. This function computes the Dice similarity coefficient between the predicted and ground truth segmentations.

```
diceResult = zeros(length(voldsTest.Files),2);

for j = 1:length(vol)
    diceResult(j,:) = dice(groundTruthLabels{j},predictedLabels{j});
end
```

Calculate the average Dice score across the set of test volumes.

```
meanDiceBackground = mean(diceResult(:,1));
disp(['Average Dice score of background across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceBackground)])
meanDiceTumor = mean(diceResult(:,2));
disp(['Average Dice score of tumor across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceTumor)])
```

The figure shows a `boxplot` (Statistics and Machine Learning Toolbox) that visualizes statistics about the Dice scores across the set of five sample test volumes. The red lines in the plot show the median Dice value for the classes. The upper and lower bounds of the blue box indicate the 25th and 75th percentiles, respectively. Black whiskers extend to the most extreme data points not considered outliers.



If you have Statistics and Machine Learning Toolbox™, then you can use the `boxplot` function to visualize statistics about the Dice scores across all your test volumes. To create a `boxplot`, set the `createBoxplot` variable in the following code to `true`.

```
createBoxplot = false;
if createBoxplot
    figure
    boxplot(diceResult)
    title('Test Set Dice Accuracy')
    xticklabels(classNames)
    ylabel('Dice Coefficient')
end
```

References

[1] Çiçek, Ö., A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger. "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016*. Athens, Greece, Oct. 2016, pp. 424-432.

[2] Isensee, F., P. Kickingereder, W. Wick, M. Bendszus, and K. H. Maier-Hein. "Brain Tumor Segmentation and Radiomics Survival Prediction: Contribution to the BRATS 2017 Challenge." In

Proceedings of BrainLes: International MICCAI Brainlesion Workshop. Quebec City, Canada, Sept. 2017, pp. 287-297.

[3] "Brain Tumours". *Medical Segmentation Decathlon*. http://medicaldecathlon.com/

The BraTS dataset is provided by Medical Segmentation Decathlon under the CC-BY-SA 4.0 license. All warranties and representations are disclaimed; see the license for details. MathWorks® has modified the data set linked in the Download Pretrained Network and Sample Test Set on page 8-0 section of this example. The modified sample dataset has been cropped to a region containing primarily the brain and tumor and each channel has been normalized independently by subtracting the mean and dividing by the standard deviation of the cropped brain region.

[4] Sudre, C. H., W. Li, T. Vercauteren, S. Ourselin, and M. J. Cardoso. "Generalised Dice Overlap as a Deep Learning Loss Function for Highly Unbalanced Segmentations." *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support: Third International Workshop*. Quebec City, Canada, Sept. 2017, pp. 240-248.

[5] Ronneberger, O., P. Fischer, and T. Brox. "U-Net:Convolutional Networks for Biomedical Image Segmentation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015*. Munich, Germany, Oct. 2015, pp. 234-241. Available at arXiv:1505.04597.

See Also

`randomPatchExtractionDatastore` | `trainNetwork` | `trainingOptions` | `transform` | `pixelLabelDatastore` | `imageDatastore` | `semanticseg` | `dicePixelClassificationLayer`

More About

- “Preprocess Volumes for Deep Learning” on page 19-20
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

Define Custom Pixel Classification Layer with Tversky Loss

This example shows how to define and create a custom pixel classification layer that uses Tversky loss.

This layer can be used to train semantic segmentation networks. To learn more about creating custom deep learning layers, see “Define Custom Deep Learning Layers” on page 18-9.

Tversky Loss

The Tversky loss is based on the Tversky index for measuring overlap between two segmented images [1 on page 8-0]. The Tversky index TI_c between one image Y and the corresponding ground truth T is given by

$$\mathrm{TI}_c = \frac{\sum_{m=1}^{M} Y_{cm} T_{cm}}{\sum_{m=1}^{M} Y_{cm} T_{cm} + \alpha \sum_{m=1}^{M} Y_{cm} T_{\bar{c}m} + \beta \sum_{m=1}^{M} Y_{\bar{c}m} T_{cm}}$$

- c corresponds to the class and $\bar{c}$ corresponds to not being in class c.
- M is the number of elements along the first two dimensions of Y.
- α and β are weighting factors that control the contribution that false positives and false negatives for each class make to the loss.

The loss L over the number of classes C is given by

$$L = \sum_{c=1}^{C} 1 - \mathrm{TI}_c$$

Classification Layer Template

Copy the classification layer template into a new file in MATLAB®. This template outlines the structure of a classification layer and includes the functions that define the layer behavior. The rest of the example shows how to complete the `tverskyPixelClassificationLayer`.

```
classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % Optional properties
    end

    methods

         function loss = forwardLoss(layer, Y, T)
             % Layer forward loss function goes here
         end

     end
end
```

Declare Layer Properties

By default, custom output layers have the following properties:

- `Name` – Layer name, specified as a character vector or a string scalar. To include this layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with this layer and `Name` is set to `''`, then the software automatically assigns a name at training time.
- `Description` – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
- `Type` – Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays `'Classification layer'` or `'Regression layer'`.

Custom classification layers also have the following property:

- `Classes` – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. If `Classes` is `'auto'`, then the software automatically sets the classes at training time. If you specify a string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`. The default value is `'auto'`.

If the layer has no other properties, then you can omit the `properties` section.

The Tversky loss requires a small constant value to prevent division by zero. Specify the property, `Epsilon`, to hold this value. It also requires two variable properties `Alpha` and `Beta` that control the weighting of false positives and false negatives, respectively.

```
classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer

    properties(Constant)
       % Small constant to prevent division by zero.
       Epsilon = 1e-8;
    end

    properties
       % Default weighting coefficients for false positives and false negatives
       Alpha = 0.5;
       Beta = 0.5;
    end

    ...
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

Specify an optional input argument name to assign to the `Name` property at creation.

```
function layer = tverskyPixelClassificationLayer(name, alpha, beta)
    % layer =  tverskyPixelClassificationLayer(name) creates a Tversky
    % pixel classification layer with the specified name.

    % Set layer name
    layer.Name = name;

    % Set layer properties
    layer.Alpha = alpha;
```

```
    layer.Beta = beta;

    % Set layer description
    layer.Description = 'Tversky loss';
end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the weighted cross entropy loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer,Y,T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

For semantic segmentation problems, the dimensions of `T` match the dimension of `Y`, where `Y` is a 4-D array of size `H`-by-`W`-by-`K`-by-`N`, where `K` is the number of classes, and `N` is the mini-batch size.

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, to ensure that `Y` is a 4-D array of prediction scores for `K` classes, you can include a fully connected layer of size `K` or a convolutional layer with `K` filters followed by a softmax layer before the output layer.

```
function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the Tversky loss between
    % the predictions Y and the training targets T.

    Pcnot = 1-Y;
    Gcnot = 1-T;
    TP = sum(sum(Y.*T,1),2);
    FP = sum(sum(Y.*Gcnot,1),2);
    FN = sum(sum(Pcnot.*T,1),2);

    numer = TP + layer.Epsilon;
    denom = TP + layer.Alpha*FP + layer.Beta*FN + layer.Epsilon;

    % Compute Tversky index
    lossTIc = 1 - numer./denom;
    lossTI = sum(lossTIc,3);

    % Return average Tversky index loss
    N = size(Y,4);
    loss = sum(lossTI)/N;

end
```

Backward Loss Function

As the `forwardLoss` function fully supports automatic differentiation, there is no need to create a function for the backward loss.

For a list of functions that support automatic differentiation, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

The completed layer is provided in `tverskyPixelClassificationLayer.m`.

```
classdef tverskyPixelClassificationLayer < nnet.layer.ClassificationLayer
    % This layer implements the Tversky loss function for training
```

```
% semantic segmentation networks.

% References
% Salehi, Seyed Sadegh Mohseni, Deniz Erdogmus, and Ali Gholipour.
% "Tversky loss function for image segmentation using 3D fully
% convolutional deep networks." International Workshop on Machine
% Learning in Medical Imaging. Springer, Cham, 2017.
% ----------


properties(Constant)
    % Small constant to prevent division by zero.
    Epsilon = 1e-8;
end

properties
    % Default weighting coefficients for False Positives and False
    % Negatives
    Alpha = 0.5;
    Beta = 0.5;
end


methods

    function layer = tverskyPixelClassificationLayer(name, alpha, beta)
        % layer =  tverskyPixelClassificationLayer(name, alpha, beta) creates a Tversky
        % pixel classification layer with the specified name and properties alpha and beta.

        % Set layer name.
        layer.Name = name;

        layer.Alpha = alpha;
        layer.Beta = beta;

        % Set layer description.
        layer.Description = 'Tversky loss';
    end


    function loss = forwardLoss(layer, Y, T)
        % loss = forwardLoss(layer, Y, T) returns the Tversky loss between
        % the predictions Y and the training targets T.

        Pcnot = 1-Y;
        Gcnot = 1-T;
        TP = sum(sum(Y.*T,1),2);
        FP = sum(sum(Y.*Gcnot,1),2);
        FN = sum(sum(Pcnot.*T,1),2);

        numer = TP + layer.Epsilon;
        denom = TP + layer.Alpha*FP + layer.Beta*FN + layer.Epsilon;

        % Compute tversky index
        lossTIc = 1 - numer./denom;
        lossTI = sum(lossTIc,3);

        % Return average tversky index loss.
```

```
            N = size(Y,4);
            loss = sum(lossTI)/N;

        end
    end
end
```

GPU Compatibility

The MATLAB functions used in `forwardLoss` in `tverskyPixelClassificationLayer` all support `gpuArray` inputs, so the layer is GPU compatible.

Check Output Layer Validity

Create an instance of the layer.

```
layer = tverskyPixelClassificationLayer('tversky',0.7,0.3);
```

Check the validity of the layer by using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a H-by-W-by-K-by-N array inputs, where K is the number of classes, and N is the number of observations in the mini-batch.

```
numClasses = 2;
validInputSize = [4 4 numClasses];
checkLayer(layer,validInputSize, 'ObservationDimension',4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 1.7623 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Use Custom Layer in Semantic Segmentation Network

Create a semantic segmentation network that uses the `tverskyPixelClassificationLayer`.

```
layers = [
    imageInputLayer([32 32 1])
    convolution2dLayer(3,64,'Padding',1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    convolution2dLayer(3,64,'Padding',1)
    reluLayer
    transposedConv2dLayer(4,64,'Stride',2,'Cropping',1)
    convolution2dLayer(1,2)
    softmaxLayer
    tverskyPixelClassificationLayer('tversky',0.3,0.7)];
```

Load training data for semantic segmentation using `imageDatastore` and `pixelLabelDatastore`.

```
dataSetDir = fullfile(toolboxdir('vision'),'visiondata','triangleImages');
imageDir = fullfile(dataSetDir,'trainingImages');
labelDir = fullfile(dataSetDir,'trainingLabels');

imds = imageDatastore(imageDir);

classNames = ["triangle" "background"];
labelIDs = [255 0];
pxds = pixelLabelDatastore(labelDir, classNames, labelIDs);
```

Associate the image and pixel label data by using datastore `combine`.

```
ds = combine(imds,pxds);
```

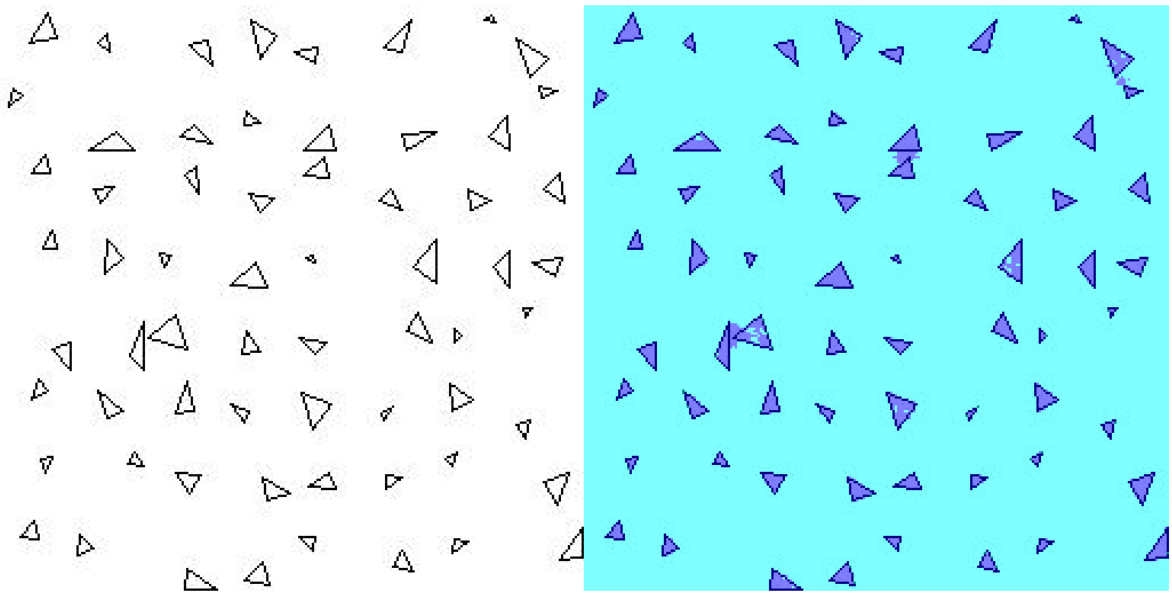
Set the training options and train the network.

```
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',100, ...
    'LearnRateDropFactor',5e-1, ...
    'LearnRateDropPeriod',20, ...
    'LearnRateSchedule','piecewise', ...
    'MiniBatchSize',50);

net = trainNetwork(ds,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:01 |       50.32% |       1.2933 |          0.0010 |
|      13 |          50 |       00:00:16 |       98.82% |       0.0985 |          0.0010 |
|      25 |         100 |       00:00:32 |       99.32% |       0.0545 |          0.0005 |
|      38 |         150 |       00:00:52 |       99.37% |       0.0472 |          0.0005 |
|      50 |         200 |       00:01:11 |       99.48% |       0.0401 |          0.0003 |
|      63 |         250 |       00:01:31 |       99.48% |       0.0379 |          0.0001 |
|      75 |         300 |       00:01:51 |       99.54% |       0.0348 |          0.0001 |
|      88 |         350 |       00:02:10 |       99.51% |       0.0351 |      6.2500e-05 |
|     100 |         400 |       00:02:29 |       99.56% |       0.0330 |      6.2500e-05 |
|========================================================================================|
Training finished: Max epochs completed.
```

Evaluate the trained network by segmenting a test image and displaying the segmentation result.

```
I = imread('triangleTest.jpg');
[C,scores] = semanticseg(I,net);

B = labeloverlay(I,C);
montage({I,B})
```



References

[1] Salehi, Seyed Sadegh Mohseni, Deniz Erdogmus, and Ali Gholipour. "Tversky loss function for image segmentation using 3D fully convolutional deep networks." *International Workshop on Machine Learning in Medical Imaging*. Springer, Cham, 2017.

See Also

`checkLayer` | `trainingOptions` | `trainNetwork` | `pixelLabelDatastore` | `semanticseg`

More About

- “Define Custom Deep Learning Layers” on page 18-9
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation Using Deep Learning” on page 8-126

Train Object Detector Using R-CNN Deep Learning

This example shows how to train an object detector using deep learning and R-CNN (Regions with Convolutional Neural Networks).

Overview

This example shows how to train an R-CNN object detector for detecting stop signs. R-CNN is an object detection framework, which uses a convolutional neural network (CNN) to classify image regions within an image [1]. Instead of classifying every region using a sliding window, the R-CNN detector only processes those regions that are likely to contain an object. This greatly reduces the computational cost incurred when running a CNN.

To illustrate how to train an R-CNN stop sign detector, this example follows the transfer learning workflow that is commonly used in deep learning applications. In transfer learning, a network trained on a large collection of images, such as ImageNet [2], is used as the starting point to solve a new classification or detection task. The advantage of using this approach is that the pretrained network has already learned a rich set of image features that are applicable to a wide range of images. This learning is transferable to the new task by fine-tuning the network. A network is fine-tuned by making small adjustments to the weights such that the feature representations learned for the original task are slightly adjusted to support the new task.

The advantage of transfer learning is that the number of images required for training and the training time are reduced. To illustrate these advantages, this example trains a stop sign detector using the transfer learning workflow. First a CNN is pretrained using the CIFAR-10 data set, which has 50,000 training images. Then this pretrained CNN is fine-tuned for stop sign detection using just 41 training images. Without pretraining the CNN, training the stop sign detector would require many more images.

Note: This example requires Computer Vision Toolbox™, Image Processing Toolbox™, Deep Learning Toolbox™, and Statistics and Machine Learning Toolbox™.

Using a CUDA-capable NVIDIA™ GPU is highly recommended for running this example. Use of a GPU requires the Parallel Computing Toolbox™. For information about the supported compute capabilities, see “GPU Support by Release” (Parallel Computing Toolbox).

Download CIFAR-10 Image Data

Download the CIFAR-10 data set [3]. This dataset contains 50,000 training images that will be used to train a CNN.

Download CIFAR-10 data to a temporary directory

```
cifar10Data = tempdir;

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

helperCIFAR10Data.download(url,cifar10Data);
```

Load the CIFAR-10 training and test data.

```
[trainingImages,trainingLabels,testImages,testLabels] = helperCIFAR10Data.load(cifar10Data);
```

Each image is a 32x32 RGB image and there are 50,000 training samples.

```
size(trainingImages)
```

```
ans = 1×4

          32          32           3       50000
```

CIFAR-10 has 10 image categories. List the image categories:

```
numImageCategories = 10;
categories(trainingLabels)
```

```
ans = 10×1 cell
    {'airplane'  }
    {'automobile'}
    {'bird'      }
    {'cat'       }
    {'deer'      }
    {'dog'       }
    {'frog'      }
    {'horse'     }
    {'ship'      }
    {'truck'     }
```

You can display a few of the training images using the following code.

```
figure
thumbnails = trainingImages(:,:,:,1:100);
montage(thumbnails)
```

Create A Convolutional Neural Network (CNN)

A CNN is composed of a series of layers, where each layer defines a specific computation. The Deep Learning Toolbox™ provides functionality to easily design a CNN layer-by-layer. In this example, the following layers are used to create a CNN:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - 2D convolution layer for Convolutional Neural Networks
- `reluLayer` - Rectified linear unit (ReLU) layer
- `maxPooling2dLayer` - Max pooling layer
- `fullyConnectedLayer` - Fully connected layer
- `softmaxLayer` - Softmax layer
- `classificationLayer` - Classification output layer for a neural network

The network defined here is similar to the one described in [4] and starts with an `imageInputLayer`. The input layer defines the type and size of data the CNN can process. In this example, the CNN is used to process CIFAR-10 images, which are 32x32 RGB images:

```
% Create the image input layer for 32x32x3 CIFAR-10 images.
[height,width,numChannels, ~] = size(trainingImages);

imageSize = [height width numChannels];
inputLayer = imageInputLayer(imageSize)
```

```
inputLayer =
  ImageInputLayer with properties:
```

```
                    Name: ''
               InputSize: [32 32 3]
   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                    Mean: []
```

Next, define the middle layers of the network. The middle layers are made up of repeated blocks of convolutional, ReLU (rectified linear units), and pooling layers. These 3 layers form the core building blocks of convolutional neural networks. The convolutional layers define sets of filter weights, which are updated during network training. The ReLU layer adds non-linearity to the network, which allow the network to approximate non-linear functions that map image pixels to the semantic content of the image. The pooling layers downsample data as it flows through the network. In a network with lots of layers, pooling layers should be used sparingly to avoid downsampling the data too early in the network.

```
% Convolutional layer parameters
filterSize = [5 5];
numFilters = 32;

middleLayers = [

% The first convolutional layer has a bank of 32 5x5x3 filters. A
% symmetric padding of 2 pixels is added to ensure that image borders
% are included in the processing. This is important to avoid
% information at the borders being washed away too early in the
% network.
convolution2dLayer(filterSize,numFilters,'Padding',2)

% Note that the third dimension of the filter can be omitted because it
% is automatically deduced based on the connectivity of the network. In
% this case because this layer follows the image layer, the third
% dimension must be 3 to match the number of channels in the input
% image.

% Next add the ReLU layer:
reluLayer()

% Follow it with a max pooling layer that has a 3x3 spatial pooling area
% and a stride of 2 pixels. This down-samples the data dimensions from
% 32x32 to 15x15.
maxPooling2dLayer(3,'Stride',2)

% Repeat the 3 core layers to complete the middle of the network.
convolution2dLayer(filterSize,numFilters,'Padding',2)
reluLayer()
maxPooling2dLayer(3, 'Stride',2)

convolution2dLayer(filterSize,2 * numFilters,'Padding',2)
reluLayer()
maxPooling2dLayer(3,'Stride',2)

]
```

```
middleLayers =
  9x1 Layer array with layers:
```

```
    1   ''   Convolution   32 5x5 convolutions with stride [1  1] and padding [2  2  2  2]
    2   ''   ReLU          ReLU
    3   ''   Max Pooling   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
    4   ''   Convolution   32 5x5 convolutions with stride [1  1] and padding [2  2  2  2]
    5   ''   ReLU          ReLU
    6   ''   Max Pooling   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
    7   ''   Convolution   64 5x5 convolutions with stride [1  1] and padding [2  2  2  2]
    8   ''   ReLU          ReLU
    9   ''   Max Pooling   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
```

A deeper network may be created by repeating these 3 basic layers. However, the number of pooling layers should be reduced to avoid downsampling the data prematurely. Downsampling early in the network discards image information that is useful for learning.

The final layers of a CNN are typically composed of fully connected layers and a softmax loss layer.

```
finalLayers = [

% Add a fully connected layer with 64 output neurons. The output size of
% this layer will be an array with a length of 64.
fullyConnectedLayer(64)

% Add an ReLU non-linearity.
reluLayer

% Add the last fully connected layer. At this point, the network must
% produce 10 signals that can be used to measure whether the input image
% belongs to one category or another. This measurement is made using the
% subsequent loss layers.
fullyConnectedLayer(numImageCategories)

% Add the softmax loss layer and classification layer. The final layers use
% the output of the fully connected layer to compute the categorical
% probability distribution over the image classes. During the training
% process, all the network weights are tuned to minimize the loss over this
% categorical distribution.
softmaxLayer
classificationLayer
]
```

```
finalLayers =
  5x1 Layer array with layers:

    1   ''   Fully Connected         64 fully connected layer
    2   ''   ReLU                   ReLU
    3   ''   Fully Connected         10 fully connected layer
    4   ''   Softmax                softmax
    5   ''   Classification Output   crossentropyex
```

Combine the input, middle, and final layers.

```
layers = [
    inputLayer
    middleLayers
    finalLayers
    ]
```

```
layers =
  15x1 Layer array with layers:

     1   ''   Image Input             32x32x3 images with 'zerocenter' normalization
     2   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding [2  2  2
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             3x3 max pooling with stride [2  2] and padding [0  0  0  0]
     5   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding [2  2  2
     6   ''   ReLU                    ReLU
     7   ''   Max Pooling             3x3 max pooling with stride [2  2] and padding [0  0  0  0]
     8   ''   Convolution             64 5x5 convolutions with stride [1  1] and padding [2  2  2
     9   ''   ReLU                    ReLU
    10   ''   Max Pooling             3x3 max pooling with stride [2  2] and padding [0  0  0  0]
    11   ''   Fully Connected         64 fully connected layer
    12   ''   ReLU                    ReLU
    13   ''   Fully Connected         10 fully connected layer
    14   ''   Softmax                 softmax
    15   ''   Classification Output   crossentropyex
```

Initialize the first convolutional layer weights using normally distributed random numbers with standard deviation of 0.0001. This helps improve the convergence of training.

```
layers(2).Weights = 0.0001 * randn([filterSize numChannels numFilters]);
```

Train CNN Using CIFAR-10 Data

Now that the network architecture is defined, it can be trained using the CIFAR-10 training data. First, set up the network training algorithm using the `trainingOptions` function. The network training algorithm uses Stochastic Gradient Descent with Momentum (SGDM) with an initial learning rate of 0.001. During training, the initial learning rate is reduced every 8 epochs (1 epoch is defined as one complete pass through the entire training data set). The training algorithm is run for 40 epochs.

Note that the training algorithm uses a mini-batch size of 128 images. If using a GPU for training, this size may need to be lowered due to memory constraints on the GPU.

```
% Set the network training options
opts = trainingOptions('sgdm', ...
    'Momentum', 0.9, ...
    'InitialLearnRate', 0.001, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 8, ...
    'L2Regularization', 0.004, ...
    'MaxEpochs', 40, ...
    'MiniBatchSize', 128, ...
    'Verbose', true);
```

Train the network using the `trainNetwork` function. This is a computationally intensive process that takes 20-30 minutes to complete. To save time while running this example, a pretrained network is loaded from disk. If you wish to train the network yourself, set the `doTraining` variable shown below to true.

Note that a CUDA-capable NVIDIA™ GPU is highly recommended for training.

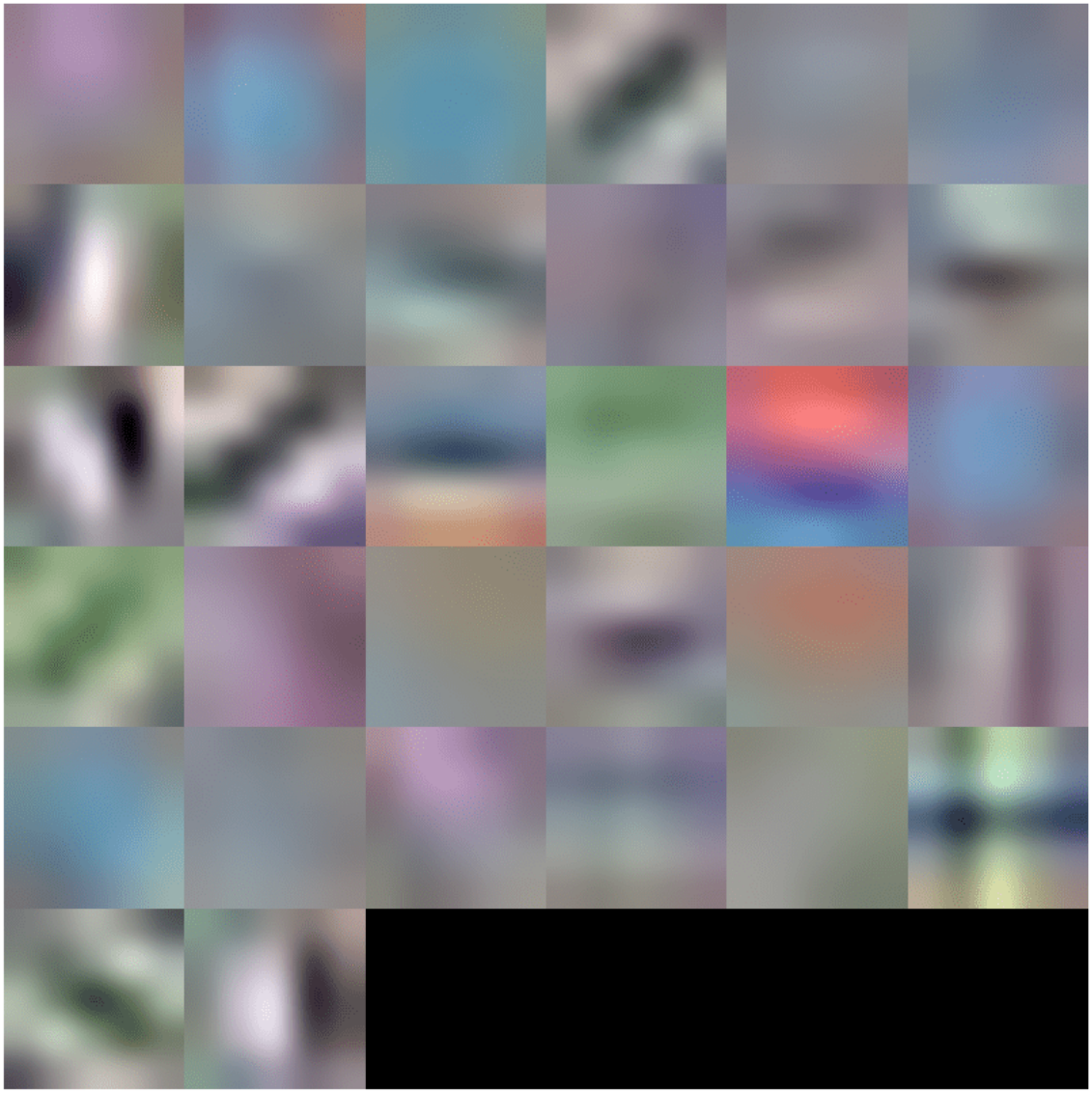
```
% A trained network is loaded from disk to save time when running the
% example. Set this flag to true to train the network.
```

```
doTraining = false;

if doTraining
    % Train a network.
    cifar10Net = trainNetwork(trainingImages, trainingLabels, layers, opts);
else
    % Load pre-trained detector for the example.
    load('rcnnStopSigns.mat','cifar10Net')
end
```

Validate CIFAR-10 Network Training

After the network is trained, it should be validated to ensure that training was successful. First, a quick visualization of the first convolutional layer's filter weights can help identify any immediate issues with training.

```
% Extract the first convolutional layer weights
w = cifar10Net.Layers(2).Weights;

% rescale the weights to the range [0, 1] for better visualization
w = rescale(w);

figure
montage(w)
```



The first layer weights should have some well defined structure. If the weights still look random, then that is an indication that the network may require additional training. In this case, as shown above, the first layer filters have learned edge-like features from the CIFAR-10 training data.

To completely validate the training results, use the CIFAR-10 test data to measure the classification accuracy of the network. A low accuracy score indicates additional training or additional training data is required. The goal of this example is not necessarily to achieve 100% accuracy on the test set, but to sufficiently train a network for use in training an object detector.

```
% Run the network on the test set.
YTest = classify(cifar10Net, testImages);
```

```
% Calculate the accuracy.
accuracy = sum(YTest == testLabels)/numel(testLabels)

accuracy = 0.7456
```

Further training will improve the accuracy, but that is not necessary for the purpose of training the R-CNN object detector.

Load Training Data

Now that the network is working well for the CIFAR-10 classification task, the transfer learning approach can be used to fine-tune the network for stop sign detection.

Start by loading the ground truth data for stop signs.

```
% Load the ground truth data
data = load('stopSignsAndCars.mat', 'stopSignsAndCars');
stopSignsAndCars = data.stopSignsAndCars;

% Update the path to the image files to match the local file system
visiondata = fullfile(toolboxdir('vision'),'visiondata');
stopSignsAndCars.imageFilename = fullfile(visiondata, stopSignsAndCars.imageFilename);

% Display a summary of the ground truth data
summary(stopSignsAndCars)

Variables:
    imageFilename: 41×1 cell array of character vectors
    stopSign: 41×1 cell
    carRear: 41×1 cell
    carFront: 41×1 cell
```

The training data is contained within a table that contains the image filename and ROI labels for stop signs, car fronts, and rears. Each ROI label is a bounding box around objects of interest within an image. For training the stop sign detector, only the stop sign ROI labels are needed. The ROI labels for car front and rear must be removed:

```
% Only keep the image file names and the stop sign ROI labels
stopSigns = stopSignsAndCars(:, {'imageFilename','stopSign'});

% Display one training image and the ground truth bounding boxes
I = imread(stopSigns.imageFilename{1});
I = insertObjectAnnotation(I,'Rectangle',stopSigns.stopSign{1},'stop sign','LineWidth',8);

figure
imshow(I)
```

Note that there are only 41 training images within this data set. Training an R-CNN object detector from scratch using only 41 images is not practical and would not produce a reliable stop sign detector. Because the stop sign detector is trained by fine-tuning a network that has been pre-trained on a larger dataset (CIFAR-10 has 50,000 training images), using a much smaller dataset is feasible.

Train R-CNN Stop Sign Detector

Finally, train the R-CNN object detector using `trainRCNNObjectDetector` (Computer Vision Toolbox). The input to this function is the ground truth table which contains labeled stop sign images, the pre-trained CIFAR-10 network, and the training options. The training function automatically modifies the original CIFAR-10 network, which classified images into 10 categories, into a network that can classify images into 2 classes: stop signs and a generic background class.

During training, the input network weights are fine-tuned using image patches extracted from the ground truth data. The 'PositiveOverlapRange' and 'NegativeOverlapRange' parameters control which image patches are used for training. Positive training samples are those that overlap with the ground truth boxes by 0.5 to 1.0, as measured by the bounding box intersection over union metric. Negative training samples are those that overlap by 0 to 0.3. The best values for these parameters should be chosen by testing the trained detector on a validation set.

For R-CNN training, **the use of a parallel pool of MATLAB workers is highly recommended to reduce training time**. `trainRCNNObjectDetector` automatically creates and uses a parallel pool based on your parallel preference settings. Ensure that the use of the parallel pool is enabled prior to training.

To save time while running this example, a pretrained network is loaded from disk. If you wish to train the network yourself, set the `doTraining` variable shown below to true.

Note that a CUDA-capable NVIDIA™ GPU is highly recommended for training.

```
% A trained detector is loaded from disk to save time when running the
% example. Set this flag to true to train the detector.
doTraining = false;

if doTraining

    % Set training options
    options = trainingOptions('sgdm', ...
        'MiniBatchSize', 128, ...
        'InitialLearnRate', 1e-3, ...
        'LearnRateSchedule', 'piecewise', ...
        'LearnRateDropFactor', 0.1, ...
        'LearnRateDropPeriod', 100, ...
        'MaxEpochs', 100, ...
        'Verbose', true);

    % Train an R-CNN object detector. This will take several minutes.
    rcnn = trainRCNNObjectDetector(stopSigns, cifar10Net, options, ...
    'NegativeOverlapRange', [0 0.3], 'PositiveOverlapRange',[0.5 1])
else
    % Load pre-trained network for the example.
    load('rcnnStopSigns.mat','rcnn')
end
```

Test R-CNN Stop Sign Detector

The R-CNN object detector can now be used to detect stop signs in images. Try it out on a test image:

```
% Read test image
testImage = imread('stopSignTest.jpg');

% Detect stop signs
[bboxes,score,label] = detect(rcnn,testImage,'MiniBatchSize',128)
```

```
bboxes = 1×4

   419   147    31    20


score = single
    0.9955

label = categorical categorical
     stopSign
```

The R-CNN object `detect` (Computer Vision Toolbox) method returns the object bounding boxes, a detection score, and a class label for each detection. The labels are useful when detecting multiple objects, e.g. stop, yield, or speed limit signs. The scores, which range between 0 and 1, indicate the confidence in the detection and can be used to ignore low scoring detections.

```
% Display the detection results
[score, idx] = max(score);

bbox = bboxes(idx, :);
annotation = sprintf('%s: (Confidence = %f)', label(idx), score);

outputImage = insertObjectAnnotation(testImage, 'rectangle', bbox, annotation);
```

```
figure
imshow(outputImage)
```

Debugging Tips

The network used within the R-CNN detector can also be used to process the entire test image. By directly processing the entire image, which is larger than the network's input size, a 2-D heat-map of classification scores can be generated. This is a useful debugging tool because it helps identify items in the image that are confusing the network, and may help provide insight into improving training.

```
% The trained network is stored within the R-CNN detector
rcnn.Network
```

```
ans =
  SeriesNetwork with properties:

    Layers: [15×1 nnet.cnn.layer.Layer]
```

Extract the `activations` from the softmax layer, which is the 14th layer in the network. These are the classification scores produced by the network as it scans the image.

```
featureMap = activations(rcnn.Network, testImage, 14);

% The softmax activations are stored in a 3-D array.
size(featureMap)
```

```
ans = 1×3

    43    78     2
```

The 3rd dimension in featureMap corresponds to the object classes.

```
rcnn.ClassNames
```

```
ans = 2×1 cell
    {'stopSign'  }
    {'Background'}
```

The stop sign feature map is stored in the first channel.

```
stopSignMap = featureMap(:, :, 1);
```

The size of the activations output is smaller than the input image due to the downsampling operations in the network. To generate a nicer visualization, resize `stopSignMap` to the size of the input image. This is a very crude approximation that maps activations to image pixels and should only be used for illustrative purposes.

```
% Resize stopSignMap for visualization
[height, width, ~] = size(testImage);
stopSignMap = imresize(stopSignMap, [height, width]);

% Visualize the feature map superimposed on the test image.
featureMapOnImage = imfuse(testImage, stopSignMap);

figure
imshow(featureMapOnImage)
```

The stop sign in the test image corresponds nicely with the largest peak in the network activations. This helps verify that the CNN used within the R-CNN detector has effectively learned to identify stop

signs. Had there been other peaks, this may indicate that the training requires additional negative data to help prevent false positives. If that's the case, then you can increase 'MaxEpochs' in the trainingOptions and re-train.

Summary

This example showed how to train an R-CNN stop sign object detector using a network trained with CIFAR-10 data. Similar steps may be followed to train other object detectors using deep learning.

References

[1] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[2] Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database." *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL, June 2009, pp. 248-255.

[3] Krizhevsky, A., and G. Hinton. "Learning multiple layers of features from tiny images." Master's Thesis. University of Toronto, Toronto, Canada, 2009.

[4] https://code.google.com/p/cuda-convnet/

See Also

`rcnnObjectDetector` | `trainingOptions` | `trainNetwork` | `trainRCNNObjectDetector` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `classify` | `detect` | `activations`

More About

- “Object Detection Using Faster R-CNN Deep Learning” on page 8-201
- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection” (Computer Vision Toolbox)

Object Detection Using Faster R-CNN Deep Learning

This example shows how to train a Faster R-CNN (regions with convolutional neural networks) object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several deep learning techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a Faster R-CNN vehicle detector using the `trainFasterRCNNObjectDetector` function. For more information, see “Object Detection” (Computer Vision Toolbox).

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToEn
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Data Set

This example uses a small labeled dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the Faster R-CNN training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

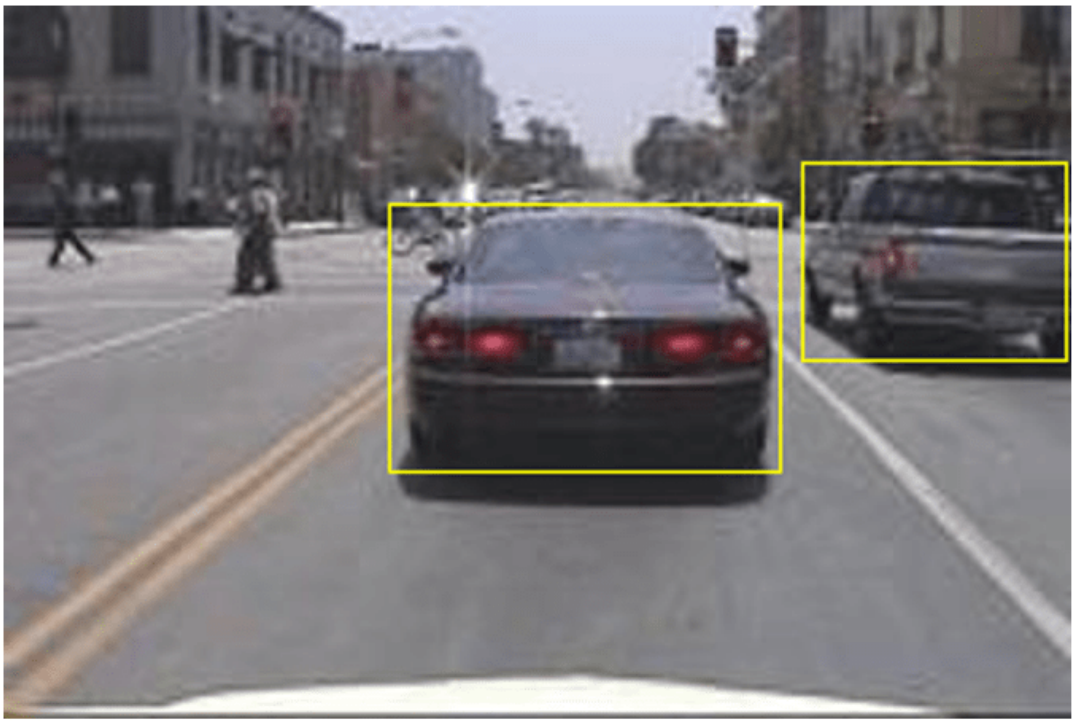
```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:,'imageFilename'});
bldsValidation = boxLabelDatastore(validationDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
validationData = combine(imdsValidation,bldsValidation);
testData = combine(imdsTest,bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Create Faster R-CNN Detection Network

A Faster R-CNN object detection network is composed of a feature extraction network followed by two subnetworks. The feature extraction network is typically a pretrained CNN, such as ResNet-50 or Inception v3. The first subnetwork following the feature extraction network is a region proposal network (RPN) trained to generate object proposals - areas in the image where objects are likely to exist. The second subnetwork is trained to predict the actual class of each object proposal.

The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” on page 1-8). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18, depending on your application requirements.

Use `fasterRCNNLayers` to create a Faster R-CNN network automatically given a pretrained feature extraction network. `fasterRCNNLayers` requires you to specify several inputs that parameterize a Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes.

```
preprocessedTrainingData = transform(trainingData, @(data)preprocessData(data,inputSize));
numAnchors = 3;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

```
anchorBoxes = 3×2

    29    17
    46    39
   136   116
```

For more information on choosing anchor boxes, see“Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox) (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis. You can use `analyzeNetwork` to find the names of other potential feature extraction layers within a network.

```
featureLayer = 'activation_40_relu';
```

Define the number of classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the Faster R-CNN object detection network.

```
lgraph = fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the Faster R-CNN network architecture, use Deep Network Designer to design the Faster R-CNN detection network manually. For more information, see “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” (Computer Vision Toolbox).

Data Augmentation

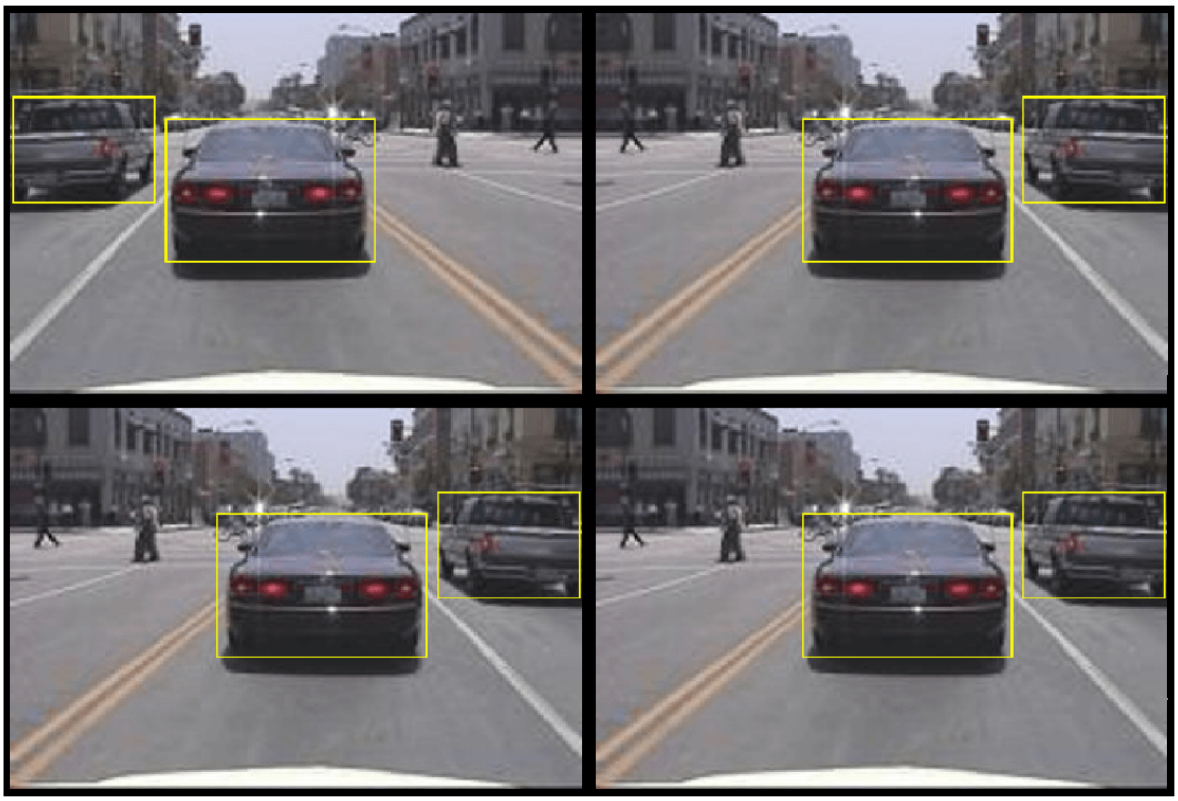
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to test and validation data. Ideally, test and validation data are representative of the original data and are left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
trainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
validationData = transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed data.

```
data = read(trainingData);
```

Display the image and box bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Train Faster R-CNN

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',10,...
    'MiniBatchSize',2,...
    'InitialLearnRate',1e-3,...
    'CheckpointPath',tempdir,...
    'ValidationData',validationData);
```

Use `trainFasterRCNNObjectDetector` to train Faster R-CNN object detector if `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % Train the Faster R-CNN detector.
```

```
    % * Adjust NegativeOverlapRange and PositiveOverlapRange to ensure
    %   that training samples tightly overlap with ground truth.
    [detector, info] = trainFasterRCNNObjectDetector(trainingData,lgraph,options, ...
        'NegativeOverlapRange',[0 0.3], ...
        'PositiveOverlapRange',[0.6 1]);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detector = pretrained.detector;
end
```

This example was verified on an Nvidia(TM) Titan X GPU with 12 GB of memory. Training the network took approximately 20 minutes. The training time varies depending on the hardware you use.

As a quick check, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{3});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the

detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data.

```
testData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

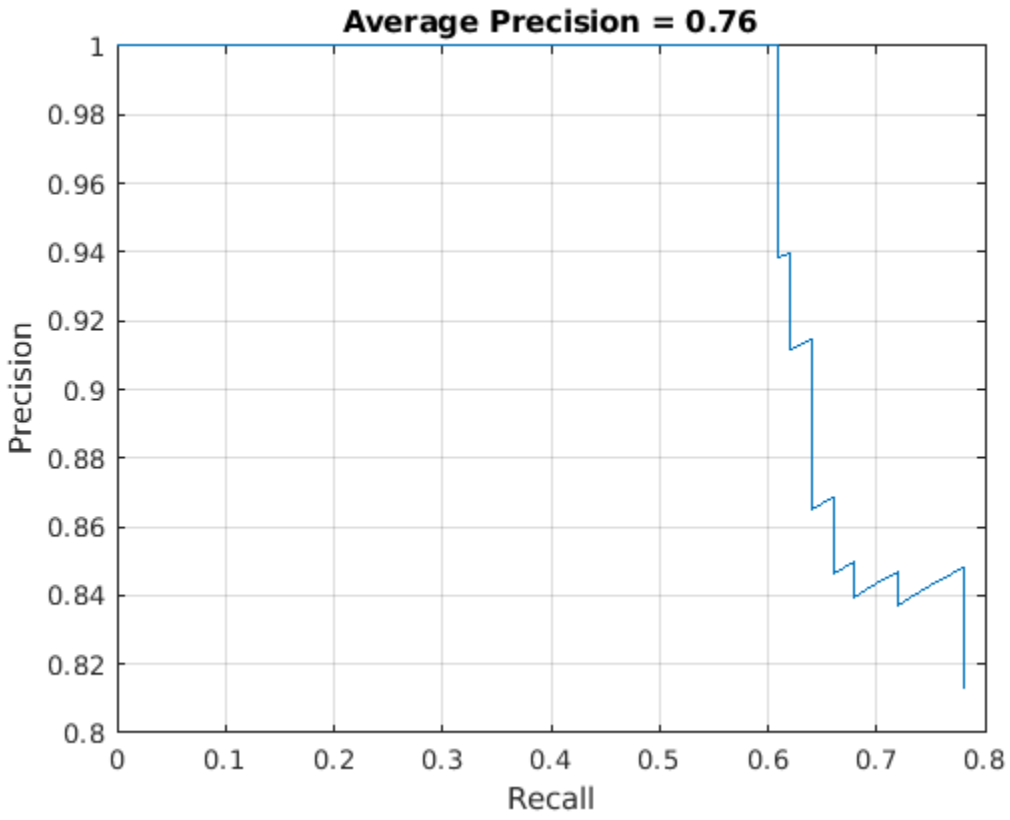
```
detectionResults = detect(detector,testData,'MinibatchSize',4);
```

Evaluate the object detector using the average precision metric.

```
[ap, recall, precision] = evaluateDetectionPrecision(detectionResults,testData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```



Supporting Functions

```
function data = augmentData(data)
% Randomly flip images and bounding boxes horizontally.
tform = randomAffine2d('XReflection',true);
sz = size(data{1});
rout = affineOutputView(sz,tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Warp boxes.
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Ren, S., K. He, R. Gershick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions of Pattern Analysis and Machine Intelligence*. Vol. 39, Issue 6, June 2017, pp. 1137-1149.

[2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[3] Girshick, R. "Fast R-CNN." *Proceedings of the 2015 IEEE International Conference on Computer Vision*. Santiago, Chile, Dec. 2015, pp. 1440-1448.

[4] Zitnick, C. L., and P. Dollar. "Edge Boxes: Locating Object Proposals from Edges." *European Conference on Computer Vision*. Zurich, Switzerland, Sept. 2014, pp. 391-405.

[5] Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. "Selective Search for Object Recognition." *International Journal of Computer Vision*. Vol. 104, Number 2, Sept. 2013, pp. 154-171.

See Also

`rcnnObjectDetector` | `trainingOptions` | `trainNetwork` | `trainRCNNObjectDetector` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `detect` | `insertObjectAnnotation` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision`

More About

- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection” (Computer Vision Toolbox)

Perform Instance Segmentation Using Mask R-CNN

This example shows how to segment individual instances of people and cars using a multiclass Mask region-based convolutional neural network (R-CNN).

Instance segmentation is a computer vision technique in which you detect and localize objects while simultaneously generating a segmentation map for each of the detected instances.

This example first shows how to perform instance segmentation using a pretrained Mask R-CNN that detects two classes. Then, you can optionally download a data set and train a multiclass Mask R-CNN using transfer learning.

Note: This example requires the Computer Vision Toolbox™ Model for Mask R-CNN. You can install the Computer Vision Toolbox Model for Mask R-CNN from the Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Perform Instance Segmentation Using Pretrained Mask R-CNN

Download the pretrained Mask R-CNN. The network is stored as a `maskrcnn` (Computer Vision Toolbox) object.

```
dataFolder = fullfile(tempdir,"coco");
trainedMaskRCNN_url = 'https://www.mathworks.com/supportfiles/vision/data/maskrcnn_object_person_
helper.downloadTrainedMaskRCNN(trainedMaskRCNN_url,dataFolder);
```

```
Pretrained MaskRCNN network already exists.
```

```
pretrained = load(fullfile(dataFolder,'maskrcnn_object_person_car.mat'));
net = pretrained.net;
```

Read a test image that contains objects of the target classes.

```
imTest = imread('visionteam.jpg');
```

Segment the objects and their masks using the `segmentObjects` (Computer Vision Toolbox) function. The `segmentObjects` function performs these preprocessing steps on the input image before performing prediction:

1. Zero center the images using the COCO data set mean.
2. Resize the image to the input size of the network, while maintaining the aspect ratio (letter boxing).

```
[masks,labels,scores,boxes] = segmentObjects(net,imTest);
```

Visualize the predictions by overlaying the detected masks on the image using the `insertObjectMask` (Computer Vision Toolbox) function.

```
overlayedImage = insertObjectMask(imTest,masks);
imshow(overlayedImage)
```

Show the bounding boxes and labels on the objects.

```
showShape("rectangle",gather(boxes),"Label",labels,"LineColor",'r')
```

Download Training Data

The COCO 2014 train images data set [2] on page 8-0 consists of 82,783 images. The annotations data contains at least five captions corresponding to each image.

Create directories to store the COCO training images and annotation data.

```
imageFolder = fullfile(dataFolder,"images");
captionsFolder = fullfile(dataFolder,"annotations");
if ~exist(imageFolder,'dir')
    mkdir(imageFolder)
    mkdir(captionsFolder)
end
```

Download the COCO 2014 training images and captions from https://cocodataset.org/#download by clicking the "2014 Train images" and "2014 Train/Val annotations" links, respectively. Extract the image files into the folder specified by `imageFolder`. Extract the annotation files into the folder specified by `captionsFolder`.

```
annotationFile = fullfile(captionsFolder,"instances_train2014.json");
str = fileread(annotationFile);
```

Read and Preprocess Training Data

To train a Mask R-CNN, you need this data.

- RGB images that serve as input to the network, specified as *H*-by-*W*-by-3 numeric arrays.
- Bounding boxes for objects in the RGB images, specified as *NumObjects*-by-4 matrices, with rows in the format [*x y w h*]).
- Instance labels, specified as *NumObjects*-by-1 string vectors.

- Instance masks. Each mask is the segmentation of one instance in the image. The COCO data set specifies object instances using polygon coordinates formatted as *NumObjects*-by-2 cell arrays. Each row of the array contains the (*x*,*y*) coordinates of a polygon along the boundary of one instance in the image. However, the Mask R-CNN in this example requires binary masks specified as logical arrays of size *H*-by-*W*-by-*NumObjects*.

Initialize Training Data Parameters

```
trainClassNames = {'person', 'car'};
numClasses = length(trainClassNames);
imageSizeTrain = [800 800 3];
```

Format COCO Annotation Data as MAT Files

The COCO API for MATLAB enables you to access the annotation data. Download the COCO API for MATLAB from https://github.com/cocodataset/cocoapi by clicking the "Code" button and selecting "Download ZIP." Extract the `cocoapi-master` directory and its contents to the folder specified by `dataFolder`. If needed for your operating system, compile the gason parser by following the instructions in the `gason.m` file within the `MatlabAPI` subdirectory.

Specify the directory location for the COCO API for MATLAB and add the directory to the path.

```
cocoAPIDir = fullfile(dataFolder,"cocoapi-master","MatlabAPI");
addpath(cocoAPIDir);
```

Specify the folder in which to store the MAT files.

```
unpackAnnotationDir = fullfile(dataFolder,"annotations_unpacked","matFiles");
if ~exist(unpackAnnotationDir,'dir')
    mkdir(unpackAnnotationDir)
end
```

Extract the COCO annotations to MAT files using the `unpackAnnotations` helper function, which is attached to this example as a supporting file in the folder `helper`. Each MAT file corresponds to a single training image and contains the file name, bounding boxes, instance labels, and instance masks for each training image. The function converts object instances specified as polygon coordinates to binary masks using the `poly2mask` (Image Processing Toolbox) function.

```
helper.unpackAnnotations(trainClassNames,annotationFile,imageFolder,unpackAnnotationDir);
```

```
Loading and preparing annotations... DONE (t=9.11s).
Unpacking annotations into MAT files...
Done!
```

Create Datastore

The Mask R-CNN expects input data as a 1-by-4 cell array containing the RGB training image, bounding boxes, instance labels, and instance masks.

Create a file datastore with a custom read function, `cocoAnnotationMATReader`, that reads the content of the unpacked annotation MAT files, converts grayscale training images to RGB, and returns the data as a 1-by-4 cell array in the required format. The custom read function is attached to this example as a supporting file in the folder `helper`.

```
ds = fileDatastore(unpackAnnotationDir, ...
    'ReadFcn',@(x)helper.cocoAnnotationMATReader(x,imageFolder));
```

Preprocess the training images, bounding boxes, and instance masks to the size expected by the network using the `transform` function. The `transform` function processes the data using the operations specified in the `preprocessData` helper function. The helper function is attached to the example as a supporting file in the folder `helper`.

The `preprocessData` helper function performs these operations on the training images, bounding boxes, and instance masks:

- Resize the RGB images and masks using the `imresize` function and rescale the bounding boxes using the `bboxresize` (Computer Vision Toolbox) function. The helper function selects a homogenous scale factor such that the smaller dimension of the image, bounding box, or mask is equal to the target network input size.
- Crop the RGB images and masks using the `imcrop` (Image Processing Toolbox) function and crop the bounding boxes using the `bboxcrop` (Computer Vision Toolbox) function. The helper function crops the image, bounding box, or mask such that the larger dimension is equal to the target network input size.
- Zero center the images using the COCO data set image mean. (The standard deviation normalization factor is included in the weights of the first convolutional layer.)

```
dsTrain = transform(ds,@(x)helper.preprocessData(x,imageSizeTrain));
```

Preview the data returned by the transformed datastore.

```
data = preview(dsTrain)
```

```
data=1×4 cell array
    {800×800×3 single}    {14×4 double}    {14×1 categorical}    {800×800×14 logical}
```

Create Mask R-CNN Network Layers

The Mask R-CNN builds upon a Faster R-CNN with a ResNet-50 base network. To transfer learn on the pretrained Mask R-CNN network, use the `maskrcnn` object to load the pretrained network and customize the network for the new set of classes and input size. By default, the `maskrcnn` object uses the same anchor boxes as the COCO data set.

```
net = maskrcnn("resnet50-coco",trainClassNames,"InputSize",imageSizeTrain)
```

```
net =
  maskrcnn with properties:

      ModelName: 'maskrcnn'
     ClassNames: {'person'  'car'}
      InputSize: [800 800 3]
    AnchorBoxes: [15×2 double]
```

Create a structure containing configuration paramaters for the Mask R-CNN network.

```
params = createMaskRCNNConfig(imageSizeTrain,numClasses,[trainClassNames {'background'}]);
params.ClassAgnosticMasks = false;
params.AnchorBoxes = net.AnchorBoxes;
params.FreezeBackbone = true;
```

Specify Training Options

Specify the options for SGDM optimization. Train the network for 10 epochs.

```
initialLearnRate = 0.0012;
momentum = 0.9;
decay = 0.01;
velocity = [];
maxEpochs = 10;
miniBatchSize = 2;
```

Batch Training Data

Create a `minibatchqueue` object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` object that enables automatic differentiation in deep learning applications.

Define a custom batching function named `miniBatchFcn`. The images are concatenated along the fourth dimension to get an *H*-by-*W*-by-*C*-by-*miniBatchSize* shaped batch. The other ground truth data is configured a cell array of length equal to the mini-batch size.

```
miniBatchFcn = @(img,boxes,labels,masks) deal(cat(4,img{:}),boxes,labels,masks);
```

Specify the mini-batch data extraction format for the image data as `"SSCB"` (spatial, spatial, channel, batch). If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
mbqTrain = minibatchqueue(dsTrain,4, ...
    "MiniBatchFormat",["SSCB","","",""], ...
    "MiniBatchSize",miniBatchSize, ...
    "OutputCast",["single","","",""], ...
    "OutputAsDlArray",[true,false,false,false], ...
    "MiniBatchFcn",miniBatchFcn, ...
    "OutputEnvironment",["auto","cpu","cpu","cpu"]);
```

Train Network

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `networkGradients` helper function. The function `networkGradients`, listed as a supporting function, returns the gradients of the loss with respect to the learnable parameters, the corresponding mini-batch loss, and the state of the current batch.
- Update the network parameters using the `sgdmupdate` function.
- Update the `state` parameters of the network with the moving average.
- Update the training progress plot.

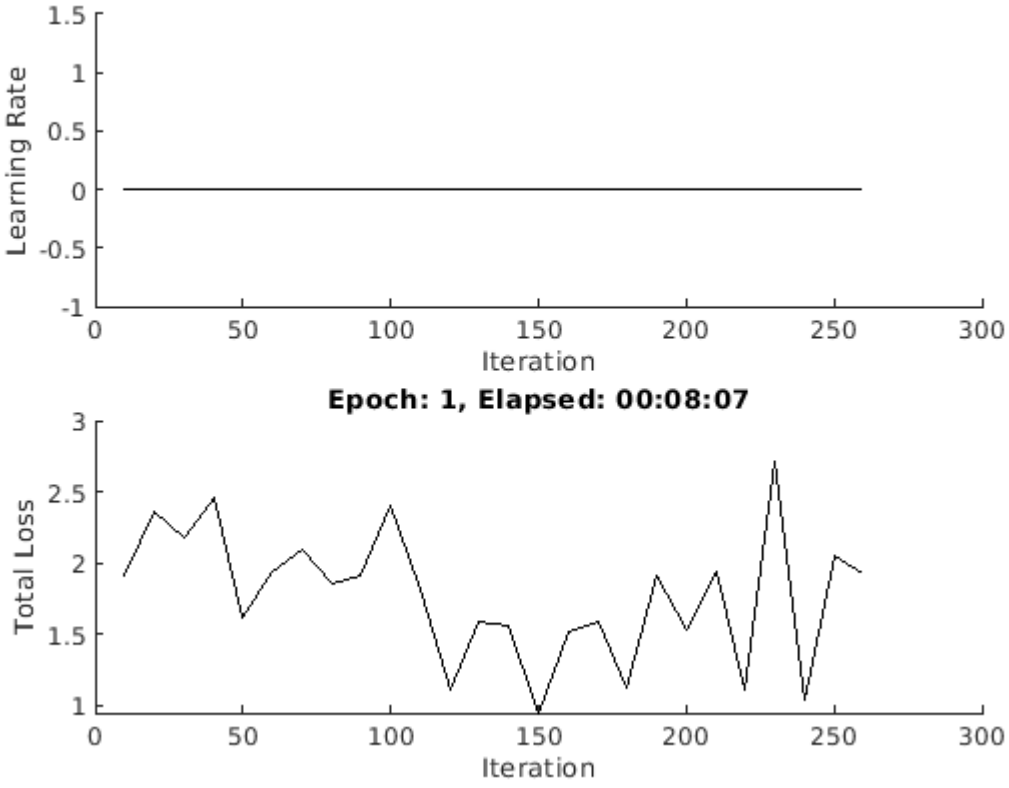
Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
doTraining = true;
if doTraining

    iteration = 1;
    start = tic;
```

```
        % Create subplots for the learning rate and mini-batch loss
        fig = figure;
        [lossPlotter, learningratePlotter] = helper.configureTrainingProgressPlotter(fig);

        % Initialize verbose output
        helper.initializeVerboseOutput([]);

        % Custom training loop
        for epoch = 1:maxEpochs
            reset(mbqTrain)
            shuffle(mbqTrain)

            while hasdata(mbqTrain)
                % Get next batch from minibatchqueue
                [X,gtBox,gtClass,gtMask] = next(mbqTrain);

                % Evaluate the model gradients and loss using dlfeval
                [gradients,loss,state,learnables] = dlfeval(@networkGradients,X,gtBox,gtClass,gtMask
                %dlnet.State = state;

                % Compute the learning rate for the current iteration
                learnRate = initialLearnRate/(1 + decay*(epoch-1));

                if(~isempty(gradients) && ~isempty(loss))
                    [net.AllLearnables,velocity] = sgdmupdate(learnables,gradients,velocity,learnRate
                else
                    continue;
                end

                % Plot loss/accuracy metric every 10 iterations
                if(mod(iteration,10)==0)
                    helper.displayVerboseOutputEveryEpoch(start,learnRate,epoch,iteration,loss);
                    D = duration(0,0,toc(start),'Format','hh:mm:ss');
                    addpoints(learningratePlotter,iteration,learnRate)
                    addpoints(lossPlotter,iteration,double(gather(extractdata(loss))))
                    subplot(2,1,2)
                    title(strcat("Epoch: ",num2str(epoch),", Elapsed: "+string(D)))
                    drawnow
                end

                iteration = iteration + 1;
            end

        end

        % Save the trained network
        modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
        save(strcat("trainedMaskRCNN-",modelDateTime,"-Epoch-",num2str(epoch),".mat"),'net');

    end
```

```
Training on GPU.
|=========================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     Loss     |      Rate       |
|=========================================================================|
```

```
|     1     |      10       |    00:00:26     |     1.9042     |      0.0012       |
|     1     |      20       |    00:00:45     |     2.3645     |      0.0012       |
|     1     |      30       |    00:01:03     |     2.1728     |      0.0012       |
|     1     |      40       |    00:01:22     |     2.4587     |      0.0012       |
|     1     |      50       |    00:01:40     |     1.6101     |      0.0012       |
|     1     |      60       |    00:01:59     |     1.9428     |      0.0012       |
|     1     |      70       |    00:02:17     |     2.0966     |      0.0012       |
|     1     |      80       |    00:02:35     |     1.8483     |      0.0012       |
|     1     |      90       |    00:02:53     |     1.9071     |      0.0012       |
|     1     |     100       |    00:03:11     |     2.3982     |      0.0012       |
|     1     |     110       |    00:03:29     |     1.8156     |      0.0012       |
|     1     |     120       |    00:03:48     |     1.1133     |      0.0012       |
|     1     |     130       |    00:04:07     |     1.5866     |      0.0012       |
|     1     |     140       |    00:04:24     |     1.5608     |      0.0012       |
|     1     |     150       |    00:04:43     |     0.9455     |      0.0012       |
|     1     |     160       |    00:05:01     |     1.5179     |      0.0012       |
|     1     |     170       |    00:05:20     |     1.5809     |      0.0012       |
|     1     |     180       |    00:05:39     |     1.1198     |      0.0012       |
|     1     |     190       |    00:05:58     |     1.9142     |      0.0012       |
|     1     |     200       |    00:06:17     |     1.5293     |      0.0012       |
|     1     |     210       |    00:06:35     |     1.9376     |      0.0012       |
|     1     |     220       |    00:06:53     |     1.1024     |      0.0012       |
|     1     |     230       |    00:07:11     |     2.7115     |      0.0012       |
|     1     |     240       |    00:07:29     |     1.0415     |      0.0012       |
|     1     |     250       |    00:07:48     |     2.0512     |      0.0012       |
|     1     |     260       |    00:08:07     |     1.9210     |      0.0012       |
```



Using the trained network, you can perform instance segmentation on test images, such as demonstrated in the section Perform Instance Segmentation Using Pretrained Mask R-CNN on page 8-0 .

References

[1] He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. “Mask R-CNN.” Preprint, submitted January 24, 2018. https://arxiv.org/abs/1703.06870.

[2] Lin, Tsung-Yi, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. “Microsoft COCO: Common Objects in Context,” May 1, 2014. https://arxiv.org/abs/1405.0312v3.

See Also

`fasterRCNNLayers` | `dlfeval` | `sgdmupdate` | `transform` | `insertObjectMask` | `dlarray` | `dlnetwork` | `FileDatastore` | `roiAlignLayer` | `minibatchqueue`

More About

- “Getting Started with Mask R-CNN for Instance Segmentation” (Computer Vision Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

External Websites

- Multiclass Instance Segmentation using Mask R-CNN

Estimate Body Pose Using Deep Learning

This example shows how to estimate the body pose of one or more people using the OpenPose algorithm and a pretrained network.

The goal of body pose estimation is to identify the location of people in an image and the orientation of their body parts. When multiple people are present in a scene, pose estimation can be more difficult because of occlusion, body contact, and proximity of similar body parts.

There are two strategies to estimating body pose. A top-down strategy first identifies individual people using object detection and then estimates the pose of each person. A bottom-up strategy first identifies body parts in an image, such as noses and left elbows, and then assembles individuals based on likely pairings of body parts. The bottom-up strategy is more robust to occlusion and body contact, but the strategy is more difficult to implement. OpenPose is a multi-person human pose estimation algorithm that uses a bottom-up strategy [1 on page 8-0].

To identify body parts in an image, OpenPose uses a pretrained neural network that predicts heatmaps and part affinity fields (PAFs) for body parts in an input image [2 on page 8-0]. Each heatmap shows the probability that a particular type of body part is located at each pixel in the image. The PAFs are vector fields that indicate whether two body parts are connected. For each defined type of body part pairing, such as neck to left shoulder, there are two PAFs that show the x- and y-component of the vector field between instances of the body parts.

To assemble body parts into individual people, the OpenPose algorithm performs a series of post-processing operations. The first operation identifies and localized body parts using the heatmaps returned by the network. Subsequent operations identify actual connections between body parts, resulting in the individual poses. For more details about the algorithm, see Identify Poses from Heatmaps and PAFs on page 8-0 .

Import the Network

Import a pretrained network from an ONNX file.

```
dataDir = fullfile(tempdir,'OpenPose');
trainedOpenPoseNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/human-pose-estimatio
downloadTrainedOpenPoseNet(trainedOpenPoseNet_url,dataDir)
unzip(fullfile(dataDir,'human-pose-estimation.zip'),dataDir);
```

Download and install the Deep Learning Toolbox™ Converter for ONNX Model Format support package.

If Deep Learning Toolbox Converter™ for ONNX Model Format is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click Install. If the support package is installed, then the `importONNXLayers` function returns a `LayerGraph` object.

```
modelfile = fullfile(dataDir,'human-pose-estimation.onnx');
layers = importONNXLayers(modelfile,"ImportWeights",true);
```

Remove the unused output layers.

```
layers = removeLayers(layers,["Output_node_95" "Output_node_98" "Output_node_147" "Output_node_15
net = dlnetwork(layers);
```

Predict Heatmaps and PAFs of Test Image

Read and display a test image.

```
im = imread("visionteam.jpg");
imshow(im)
```

The network expects image data of data type `single` in the range [-0.5, 0.5]. Shift and rescale the data to this range.

```
netInput = im2single(im)-0.5;
```

The network expects the color channels in the order blue, green, red. Switch the order of the image color channels.

```
netInput = netInput(:,:,[3 2 1]);
```

Store the image data as a `dlarray`.

```
netInput = dlarray(netInput,"SSC");
```

Predict the heatmaps, which are output from the 2-D convolutional layer named 'node_147'.

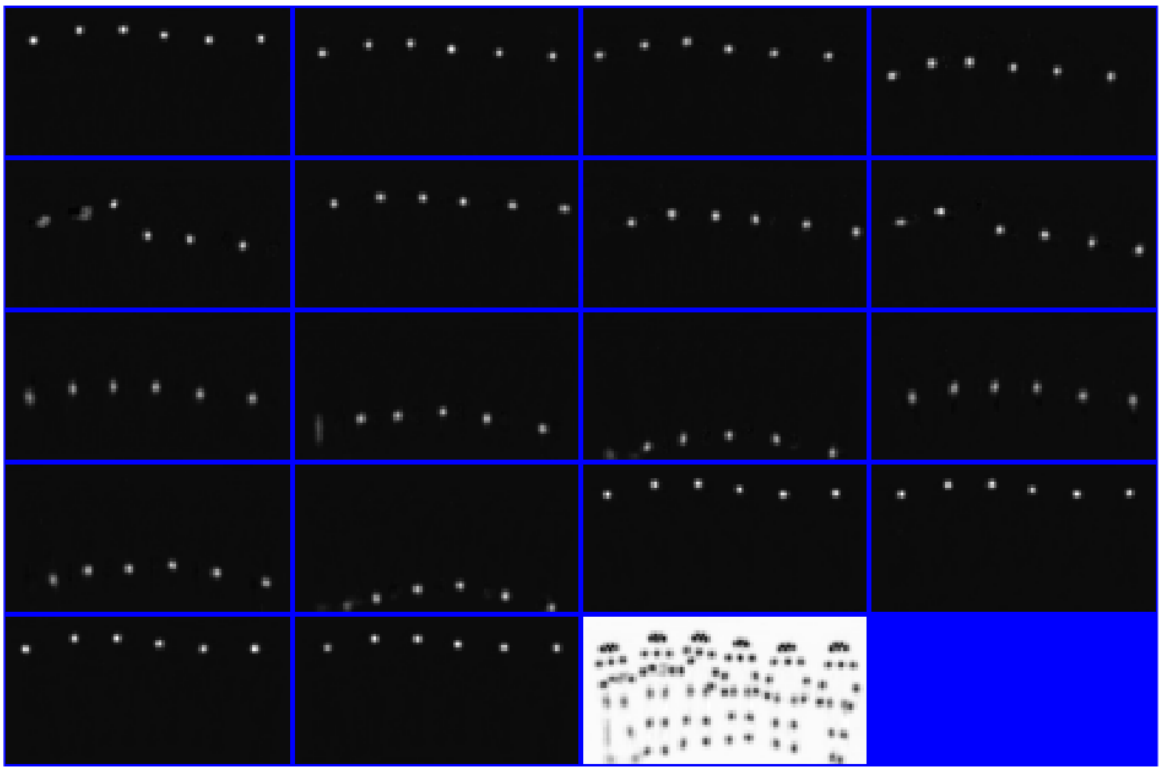
```
heatmaps = predict(net,netInput,"Outputs","node_147");
```

Get the numeric heatmap data stored in the `dlarray`. The data has 19 channels. Each channel corresponds to a heatmap for a unique body part, with one additional heatmap for the background.

```
heatmaps = extractdata(heatmaps);
```

Display the heatmaps in a montage, rescaling the data to the range [0, 1] expected of images of data type `single`. The scene has six people, and there are six bright spots in each heatmap.

```
montage(rescale(heatmaps),"BackgroundColor","b","BorderSize",3)
```



To visualize the correspondence of bright spots with the bodies, display the first heatmap in falsecolor over the test image.

```
idx = 1;
hmap = heatmaps(:,:,idx);
hmap = imresize(hmap,size(im,[1 2]));
imshowpair(hmap,im);
```

The OpenPose algorithm does not use the background heatmap to determine the location of body parts. Remove the background heatmap.

```
heatmaps = heatmaps(:,:,1:end-1);
```

Predict the PAFs, which are output from the 2-D convolutional layer named 'node_150'.

```
pafs = predict(net,netInput,"Outputs","node_150");
```

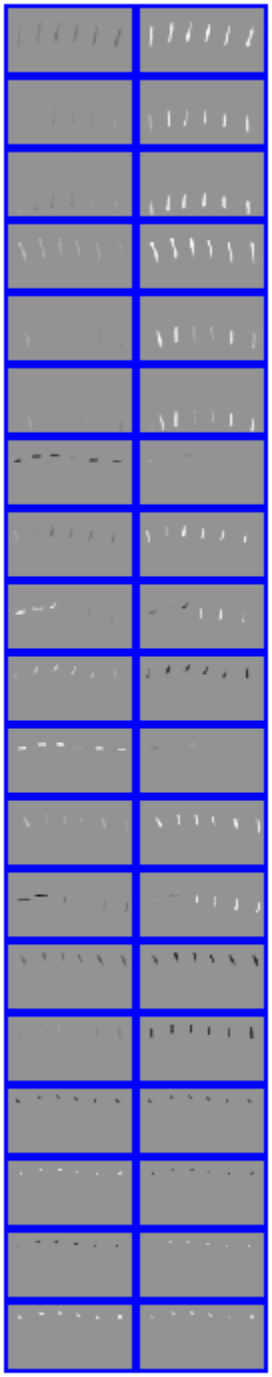
Get the numeric PAF data stored in the `dlarray`. The data has 38 channels. There are two channels for each type of body part pairing, which represent the x- and y-component of the vector field.

```
pafs = extractdata(pafs);
```

Display the PAFs in a montage, rescaling the data to the range [0, 1] expected of images of data type `single`. The two columns show the x- and y-components of the vector field, respectively. The body part pairings are in the order determined by the `params.PAF_INDEX` value.

- Pairs of body parts with a mostly vertical connection have large magnitudes for the y-component pairings and negligible values for the x-component pairings. One example is the right hip to right knee connection, which appears in the second row. Note that the PAFs depend on the actual poses in the image. An image with a body in a different orientation, such as lying down, will not necessarily have a large y-component magnitude for the right hip to right knee connection.
- Pairs of body parts with a mostly horizontal connection have large magnitudes for the x-component pairings and negligible values for the y-component pairings. One example is the neck to left shoulder connection, which appears in the seventh row.
- Pairs of body part at an angle have values for both x- and y-components of the vector field. One example is the neck to left hip, which appears in the first row.

```
montage(rescale(pafs),"Size",[19 2],"BackgroundColor","b","BorderSize",3)
```



To visualize the correspondence of the PAFs with the bodies, display the x- and y-component of the first type of body part pair in falsecolor over the test image.

```
idx = 1;
impair = horzcat(im,im);
pafpair = horzcat(pafs(:,:,2*idx-1),pafs(:,:,2*idx));
pafpair = imresize(pafpair,size(impair,[1 2]));
imshowpair(pafpair,impair);
```

Identify Poses from Heatmaps and PAFs

The post-processing part of the algorithm identifies the individual poses of the people in the image using the heatmaps and PAFs returned by the neural network.

Get parameters of the OpenPose algorithm using the `getBodyPoseParameters` helper function. The function is attached to the example as a supporting file. The function returns a struct with parameters such as the number of body parts and connections between body part types to consider. The parameters also include thresholds that you can adjust to improve the performance of the algorithm.

```
params = getBodyPoseParameters;
```

Identify individual people and their poses by using the `getBodyPoses` helper function. This function is attached to the example as a supporting file. The helper function performs all post-processing steps for pose estimation:

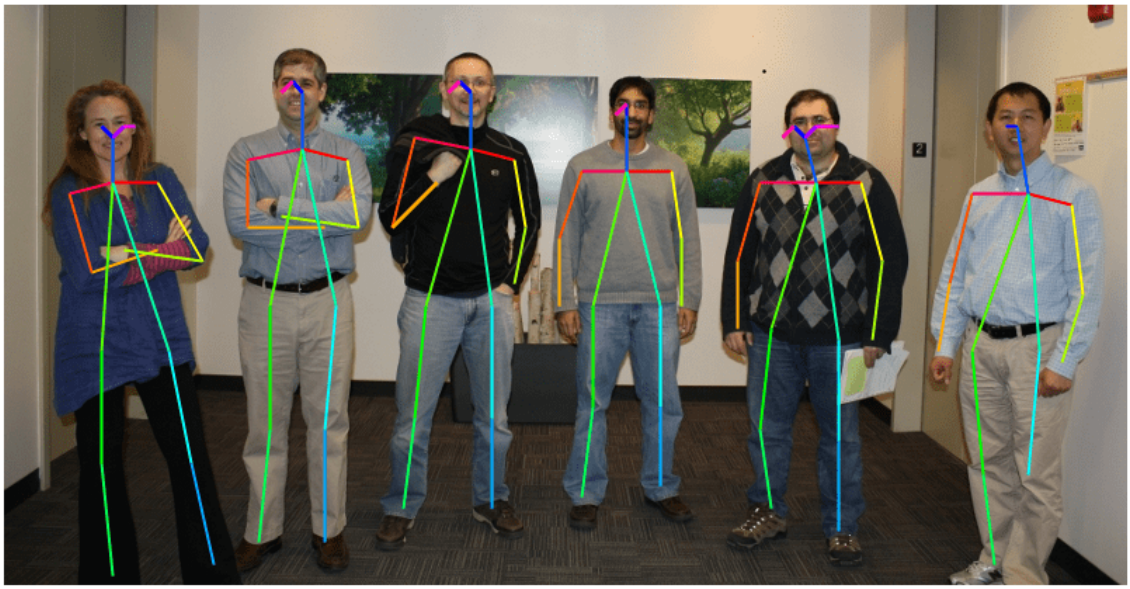
1. Detect the precise body part locations from the heatmaps using nonmaximum suppression.
2. For each type of body part pairing, generate all possible pairs between detected body parts. For instance, generate all possible pairs between the six necks and the six left shoulders. The result is a bipartite graph.
3. Score the pairs by computing the line integral of the straight line connecting the two detected body parts through the PAF vector field. A large score indicates a strong connection between detected body parts.
4. Sort the possible pairs by their scores and find the valid pairs. Valid body part pairs are pairs that connect two body parts that belong to the same person. Typically, pairs with the largest score are considered first because they are most likely to be a valid pair. However, the algorithm compensates for occlusion and proximity using additional constraints. For example, the same person cannot have duplicate pairs of body parts, and one body part cannot belong to two different people.
5. Knowing which body parts are connected, assemble the body parts into separate poses for each individual person.

The helper function returns a 3-D matrix. The first dimension represents the number of identified people in the image. The second dimension represents the number of body part types. The third dimension indicates the x- and y-coordinates for each body part of each person. If a body part is not detected in the image, then the coordinates for that part are [NaN NaN].

```
poses = getBodyPoses(heatmaps,pafs,params);
```

Display the body poses using the `renderBodyPoses` helper function. This function is attached to the example as a supporting file.

```
renderBodyPoses(im,poses,size(heatmaps,1),size(heatmaps,2),params);
```



References

[1] Cao, Zhe, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. “OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields.” *ArXiv:1812.08008 [Cs]*, May 30, 2019. https://arxiv.org/abs/1812.08008.

[2] Osokin, Daniil. “Real-Time 2D Multi-Person Pose Estimation on CPU: Lightweight OpenPose.” *ArXiv:1811.12004 [Cs]*, November 29, 2018. https://arxiv.org/abs/1811.12004.

See Also

`importONNXLayers` | `dlnetwork` | `predict`

Generate Image from Segmentation Map Using Deep Learning

This example shows how to generate a synthetic image of a scene from a semantic segmentation map using a pix2pixHD conditional generative adversarial network (CGAN).

Pix2pixHD [1 on page 8-0] consists of two networks that are trained simultaneously to maximize the performance of both.

1. The generator is an encoder-decoder style neural network that generates a scene image from a semantic segmentation map. A CGAN network trains the generator to generate a scene image that the discriminator misclassifies as real.
2. The discriminator is a fully convolutional neural network that compares a generated scene image and the corresponding real image and attempts to classify them as fake and real, respectively. A CGAN network trains the discriminator to correctly distinguish between generated and real image.

The generator and discriminator networks compete against each other during training. The training converges when neither network can improve further.

Download CamVid Data Set

This example uses the CamVid data set [2 on page 8-0] from the University of Cambridge for training. This data set is a collection of 701 images containing street-level views obtained while driving. The data set provides pixel labels for 32 semantic classes including car, pedestrian, and road.

Download the CamVid data set from these URLs. The download time depends on your internet connection.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

dataDir = fullfile(tempdir,'CamVid');
downloadCamVidData(dataDir,imageURL,labelURL);
imgDir = fullfile(dataDir,"images","701_StillsRaw_full");
labelDir = fullfile(dataDir,'labels');
```

Preprocess Training Data

Create an `imageDatastore` to store the images in the CamVid data set.

```
imds = imageDatastore(imgDir);
imageSize = [576 768];
```

Define the class names and pixel label IDs of the 32 classes in the CamVid data set using the helper function `defineCamVid32ClassesAndPixelLabelIDs`. Get a standard colormap for the CamVid data set using the helper function `camvid32ColorMap`. The helper functions are attached to the example as supporting files.

```
numClasses = 32;
[classes,labelIDs] = defineCamVid32ClassesAndPixelLabelIDs;
cmap = camvid32ColorMap;
```

Create a `pixelLabelDatastore` (Computer Vision Toolbox) to store the pixel label images.

```
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Preview a pixel label image and the corresponding ground truth scene image. Convert the labels from categorical labels to RGB colors by using the `label2rgb` (Image Processing Toolbox) function, then display the pixel label image and ground truth image in a montage.

```
im = preview(imds);
px = preview(pxds);
px = label2rgb(px,cmap);
montage({px,im})
```

Partition the data into training and test sets using the helper function `partitionCamVidForPix2PixHD`. This function is attached to the example as a supporting file. The helper function splits the data into 648 training files and 32 test files.

```
[imdsTrain,imdsTest,pxdsTrain,pxdsTest] = partitionCamVidForPix2PixHD(imds,pxds,classes,labelIDs)
```

Use the `combine` function to combine the pixel label images and ground truth scene images into a single datastore.

```
dsTrain = combine(pxdsTrain,imdsTrain);
```

Augment the training data by using the `transform` function with custom preprocessing operations specified by the helper function `preprocessCamVidForPix2PixHD`. This helper function is attached to the example as a supporting file.

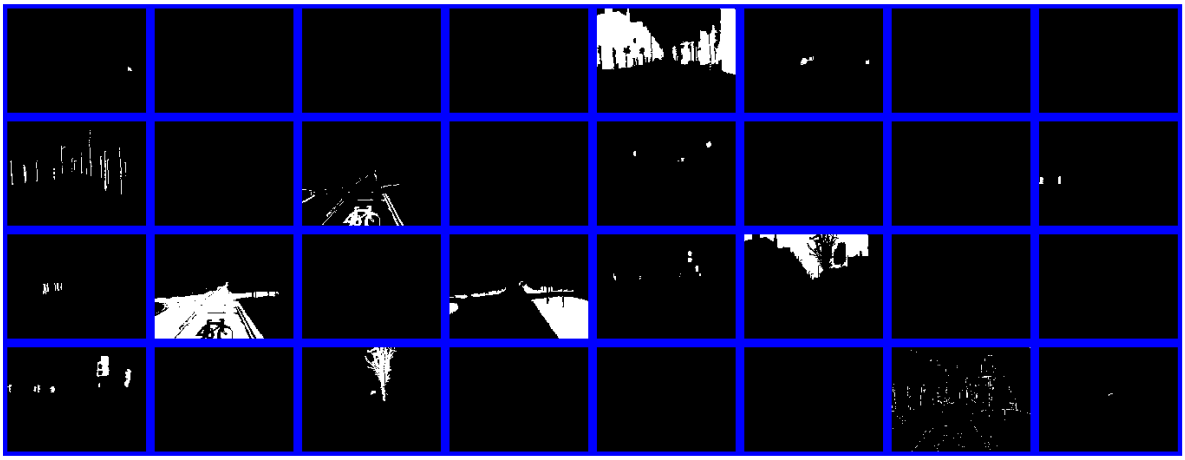
The `preprocessCamVidForPix2PixHD` function performs these operations:

1. Scale the ground truth data to the range [-1, 1]. This range matches the range of the final `tanhLayer` in the generator network.
2. Resize the image and labels to the output size of the network, 576-by-768 pixels, using bicubic and nearest neighbor downsampling, respectively.
3. Convert the single channel segmentation map to a 32-channel one-hot encoded segmentation map using the `onehotencode` function.
4. Randomly flip image and pixel label pairs in the horizontal direction.

```
dsTrain = transform(dsTrain,@(x) preprocessCamVidForPix2PixHD(x,imageSize));
```

Preview the channels of a one-hot encoded segmentation map in a montage. Each channel represents a one-hot map corresponding to pixels of a unique class.

```
map = preview(dsTrain);
montage(map{1},'Size',[4 8],'Bordersize',5,'BackgroundColor','b')
```



Create Generator Network

Define a pix2pixHD generator network that generates a scene image from a depth-wise one-hot encoded segmentation map. This input has same height and width as the original segmentation map and the same number of channels as classes.

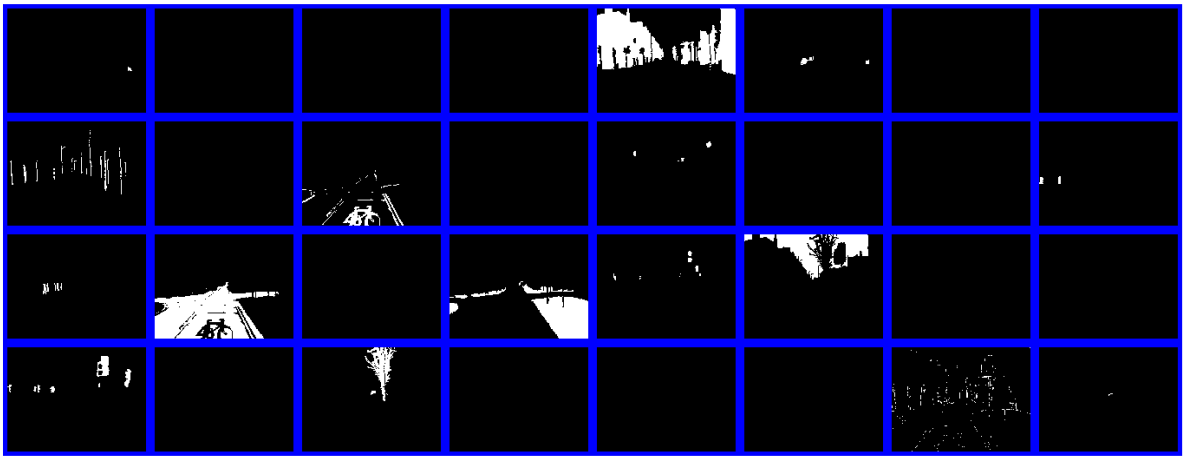
```
generatorInputSize = [imageSize numClasses];
```

Create the pix2pixHD generator network using the `pix2pixHDGlobalGenerator` (Image Processing Toolbox) function.

```
dlnetGenerator = pix2pixHDGlobalGenerator(generatorInputSize);
```

Display the network architecture.

```
analyzeNetwork(dlnetGenerator)
```



Note that this example shows the use of pix2pixHD global generator for generating images of size 576-by-768 pixels. To create local enhancer networks that generate images at higher resolution such as 1152-by-1536 pixels or even higher, you can use the `addPix2PixHDLocalEnhancer` (Image Processing Toolbox) function. The local enhancer networks help generate fine level details at very high resolutions.

Create Discriminator Network

Define the patch GAN discriminator networks that classifies an input image as either real (1) or fake (0). This example uses two discriminator networks at different input scales, also known as multiscale discriminators. The first scale is the same size as the image size, and the second scale is half the size of image size.

The input to the discriminator is the depth-wise concatenation of the one-hot encoded segmentation maps and the scene image to be classified. Specify the number of channels input to the discriminator as the total number of labeled classes and image color channels.

```
numImageChannels = 3;
numChannelsDiscriminator = numClasses + numImageChannels;
```

Specify the input size of the first discriminator. Create the patch GAN discriminator with instance normalization using the `patchGANDiscriminator` (Image Processing Toolbox) function.

```
discriminatorInputSizeScale1 = [imageSize numChannelsDiscriminator];
dlnetDiscriminatorScale1 = patchGANDiscriminator(discriminatorInputSizeScale1,"NormalizationLaye
```

Specify the input size of the second discriminator as half the image size, then create the second patch GAN discriminator.

```
discriminatorInputSizeScale2 = [floor(imageSize)./2 numChannelsDiscriminator];
dlnetDiscriminatorScale2 = patchGANDiscriminator(discriminatorInputSizeScale2,"NormalizationLaye
```

Visualize the networks.

```
analyzeNetwork(dlnetDiscriminatorScale1);
analyzeNetwork(dlnetDiscriminatorScale2);
```

Define Model Gradients and Loss Functions

The helper function `modelGradients` calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator. This function is defined in Supporting Functions on page 8-0 section of this example.

Generator Loss

The objective of the generator is to generate images that the discriminator classifies as real (1). The generator loss consists of three losses.

- The adversarial loss is computed as the squared difference between a vector of ones and the discriminator predictions on the generated image. $\widehat{Y}_{generated}$ are discriminator predictions on the image generated by the generator. This loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions on page 8-0 section of this example.

  $$lossAdversarialGenerator = (1 - \widehat{Y}_{generated})^2$$

- The feature matching loss penalises the L^1 distance between the real and generated feature maps obtained as predictions from the discriminator network. T is total number of discriminator feature layers. Y_{real} and $\widehat{Y}_{generated}$ are the ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdFeatureMatchingLoss` helper function defined in the Supporting Functions on page 8-0 section of this example

  $$lossFeatureMatching = \sum_{i=1}^{T} ||Y_{real} - \widehat{Y}_{generated}||_1$$

- The perceptual loss penalises the L^1 distance between real and generated feature maps obtained as predictions from a feature extraction network. T is total number of feature layers. $Y_{VggReal}$ and $\widehat{Y}_{VggGenerated}$ are network predictions for ground truth images and generated images, respectively. This loss is implemented using the `pix2pixhdVggLoss` helper function defined in the Supporting Functions on page 8-0 section of this example. The feature extraction network is created in Load Feature Extraction Network on page 8-0 .

  $$lossVgg = \sum_{i=1}^{T} ||Y_{VggReal} - \widehat{Y}_{VggGenerated}||_1$$

The overall generator loss is a weighted sum of all three losses. λ_1, λ_2, and λ_3 are the weight factors for adversarial loss, feature matching loss, and perceptual loss, respectively.

$$lossGenerator = \lambda_1 * lossAdversarialGenerator + \lambda_2 * lossFeatureMatching + \lambda_3 * lossPerceptual$$

Note that the adversarial loss and feature matching loss for the generator are computed for two different scales.

Discriminator Loss

The objective of the discriminator is to correctly distinguish between ground truth images and generated images. The discriminator loss is a sum of two components:

- The squared difference between a vector of ones and the predictions of the discriminator on real images
- The squared difference between a vector of zeros and the predictions of the discriminator on generated images

$$lossDiscriminator = (1 - Y_{real})^2 + (0 - \hat{Y}_{generated})^2$$

The discriminator loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions on page 8-0 section of this example. Note that adversarial loss for the discriminator is computed for two different discriminator scales.

Load Feature Extraction Network

This example modifies a pretrained VGG-19 deep neural network to extract the features of the real and generated images at various layers. These multilayer features are used to compute the perceptual loss of the generator.

To get a pretrained VGG-19 network, install `vgg19`. If you do not have the required support packages installed, then the software provides a download link.

```
netVGG = vgg19;
```

Visualize the network architecture using the Deep Network Designer app.

```
deepNetworkDesigner(netVGG)
```

To make the VGG-19 network suitable for feature extraction, keep the layers up to 'pool5' and remove all of the fully connected layers from the network. The resulting network is a fully convolutional network.

```
netVGG = layerGraph(netVGG.Layers(1:38));
```

Create a new image input layer with no normalization. Replace the original image input layer with the new layer.

```
inp = imageInputLayer([imageSize 3],"Normalization","None","Name","Input");
netVGG = replaceLayer(netVGG,"input",inp);
netVGG = dlnetwork(netVGG);
```

Specify Training Options

Specify the options for Adam optimization. Train for 60 epochs. Specify identical options for the generator and discriminator networks.

- Specify an equal learning rate of 0.0002.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.
- Use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.
- Use a mini-batch size of 1 for training.

```
numEpochs = 60;
learningRate = 0.0002;
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminatorScale1 = [];
trailingAvgSqDiscriminatorScale1 = [];
```

```
trailingAvgDiscriminatorScale2 = [];
trailingAvgSqDiscriminatorScale2 = [];
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
miniBatchSize = 1;
```

Create a `minibatchqueue` object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` object that enables auto differentiation in deep learning applications.

Specify the mini-batch data extraction format as `SSCB` (spatial, spatial, channel, batch). Set the `DispatchInBackground` name-value pair argument as the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
mbqTrain = minibatchqueue(dsTrain,"MiniBatchSize",miniBatchSize, ...
   "MiniBatchFormat","SSCB","DispatchInBackground",canUseGPU);
```

Train the Network

By default, the example downloads a pretrained version of the pix2pixHD generator network for the CamVid data set by using the helper function `downloadTrainedPix2PixHDNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradients` helper function.
- Update the network parameters using the `adamupdate` function.
- Update the training progress plot for every iteration and display various computed losses.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

Training takes about 22 hours on an NVIDIA™ Titan RTX and can take even longer depending on your GPU hardware. If your GPU device has less memory, try reducing the size of the input images by specifying the `imageSize` variable as [480 640] in the Preprocess Training Data on page 8-0 section of the example.

```
doTraining = false;
if doTraining
    fig = figure;

    lossPlotter = configureTrainingProgressPlotter(fig);
    iteration = 0;

    % Loop over epochs
    for epoch = 1:numEpochs

        % Reset and shuffle the data
        reset(mbqTrain);
```

```
            shuffle(mbqTrain);

            % Loop over each image
            while hasdata(mbqTrain)
                iteration = iteration + 1;

                % Read data from current mini-batch
                [dlInputSegMap,dlRealImage] = next(mbqTrain);

                % Evaluate the model gradients and the generator state using
                % dlfeval and the GANLoss function listed at the end of the
                % example
                [gradParamsG,gradParamsDScale1,gradParamsDScale2,lossGGAN,lossGFM,lossGVGG,lossD] = d
                    @modelGradients,dlInputSegMap,dlRealImage,dlnetGenerator,dlnetDiscriminatorScale1

                % Update the generator parameters
                [dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = adamupdate( ...
                    dlnetGenerator,gradParamsG, ...
                    trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Update the discriminator scale1 parameters
                [dlnetDiscriminatorScale1,trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorSc
                    dlnetDiscriminatorScale1,gradParamsDScale1, ...
                    trailingAvgDiscriminatorScale1,trailingAvgSqDiscriminatorScale1,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Update the discriminator scale2 parameters
                [dlnetDiscriminatorScale2,trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorSc
                    dlnetDiscriminatorScale2,gradParamsDScale2, ...
                    trailingAvgDiscriminatorScale2,trailingAvgSqDiscriminatorScale2,iteration, ...
                    learningRate,gradientDecayFactor,squaredGradientDecayFactor);

                % Plot and display various losses
                lossPlotter = updateTrainingProgressPlotter(lossPlotter,iteration, ...
                    epoch,numEpochs,lossD,lossGGAN,lossGFM,lossGVGG);
            end
        end
        save('trainedPix2PixHDNet.mat','dlnetGenerator');

    else
        trainedPix2PixHDNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedPix2PixH
        netDir = fullfile(tempdir,'CamVid');
        downloadTrainedPix2PixHDNet(trainedPix2PixHDNet_url,netDir);
        load(fullfile(netDir,'trainedPix2PixHDv2.mat'));
    end
```

Evaluate Generated Images from Test Data

The performance of this trained Pix2PixHD network is limited because the number of CamVid training images is relatively small. Additionally, some images belong to an image sequence and therefore are correlated with other images in the training set. To improve the effectiveness of the Pix2PixHD network, train the network using a different data set that has a larger number of training images without correlation.

Because of the limitations, this Pix2PixHD network generates more realistic images for some test images than for others. To demonstrate the difference in results, compare the generated images for

the first and third test image. The camera angle of the first test image has an uncommon vantage point that faces more perpendicular to the road than the typical training image. In contrast, the camera angle of the third test image has a typical vantage point that faces along the road and shows two lanes with lane markers. The network has significantly better performance generating a realistic image for the third test image than for the first test image.

Get the first ground truth scene image from the test data. Resize the image using bicubic interpolation.

```
idxToTest = 1;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");
```

Get the corresponding pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.

```
segMap = readimage(pxdsTest,idxToTest);
segMap = imresize(segMap,imageSize,"nearest");
```

Convert the pixel label image to a multichannel one-hot segmentation map by using the `onehotencode` function.

```
segMapOneHot = onehotencode(segMap,3,'single');
```

Create `dlarray` objects that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a `gpuArray` object.

```
dlSegMap = dlarray(segMapOneHot,'SSCB');
if canUseGPU
    dlSegMap = gpuArray(dlSegMap);
end
```

Generate a scene image from the generator and one-hot segmentation map using the `predict` function.

```
dlGeneratedImage = predict(dlnetGenerator,dlSegMap);
generatedImage = extractdata(gather(dlGeneratedImage));
```

The final layer of the generator network produces activations in the range [-1, 1]. For display, rescale the activations to the range [0, 1].

```
generatedImage = rescale(generatedImage);
```

For display, convert the labels from categorical labels to RGB colors by using the `label2rgb` (Image Processing Toolbox) function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and Ground Truth Scene Image
```

Get the third ground truth scene image from the test data. Resize the image using bicubic interpolation.

```
idxToTest = 3;
gtImage = readimage(imdsTest,idxToTest);
gtImage = imresize(gtImage,imageSize,"bicubic");
```

To get the third pixel label image from the test data and to generate the corresponding scene image, you can use the helper function `evaluatePix2PixHD`. This helper function is attached to the example as a supporting file.

The `evaluatePix2PixHD` function performs the same operations as the evaluation of the first test image:

- Get a pixel label image from the test data. Resize the pixel label image using nearest neighbor interpolation.
- Convert the pixel label image to a multichannel one-hot segmentation map using the `onehotencode` function.
- Create a `dlarray` object to input data to the generator. For GPU inference, convert the data to a `gpuArray` object.
- Generate a scene image from the generator and one-hot segmentation map using the `predict` function.
- Rescale the activations to the range [0, 1].

```
[generatedImage,segMap] = evaluatePix2PixHD(pxdsTest,idxToTest,imageSize,dlnetGenerator);
```

For display, convert the labels from categorical labels to RGB colors by using the `label2rgb` (Image Processing Toolbox) function.

```
coloredSegMap = label2rgb(segMap,cmap);
```

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
figure
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
title(['Test Pixel Label Image ',num2str(idxToTest),' with Generated and Ground Truth Scene Image
```

Evaluate Generated Images from Custom Pixel Label Images

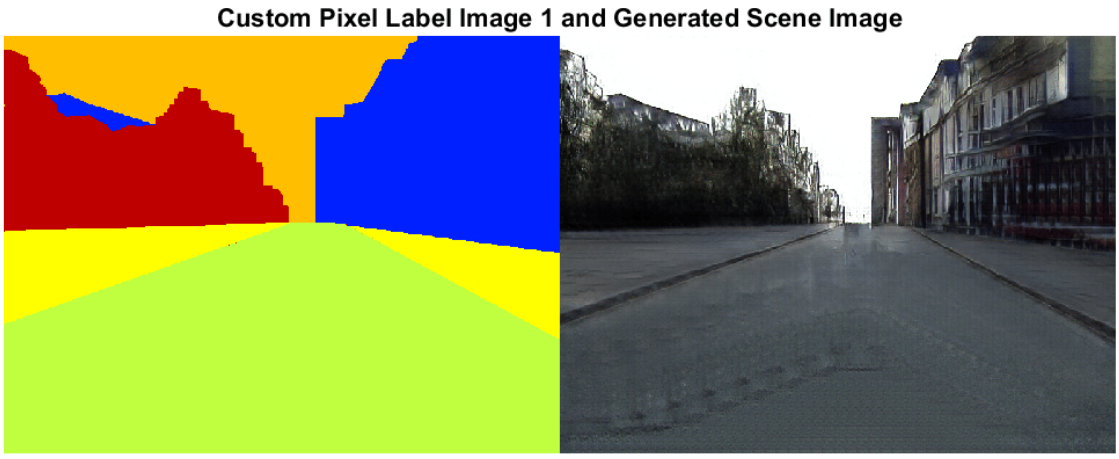
To evaluate how well the network generalizes to pixel label images outside the CamVid data set, generate scene images from custom pixel label images. This example uses pixel label images that were created using the Image Labeler (Computer Vision Toolbox) app. The pixel label images are attached to the example as supporting files. No ground truth images are available.

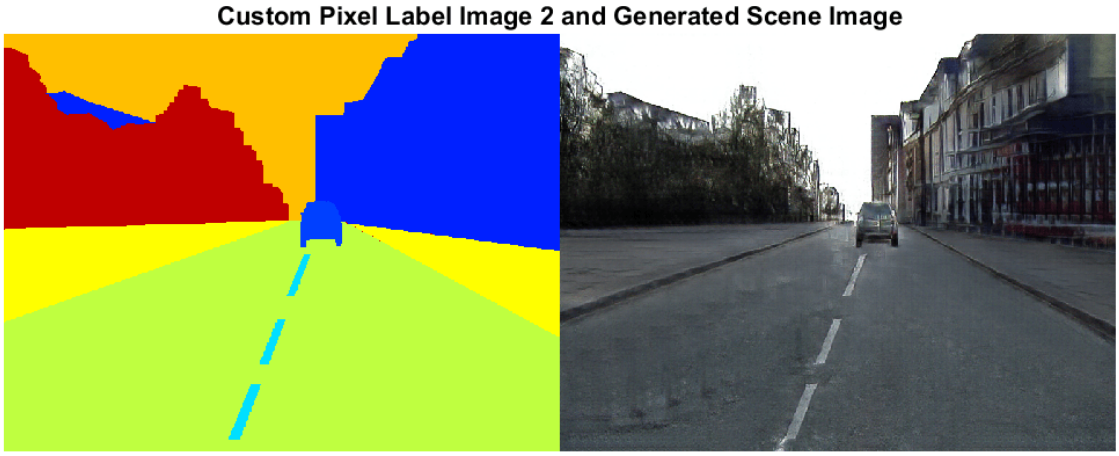
Create a pixel label datastore that reads and processes the pixel label images in the current example directory.

```
cpxds = pixelLabelDatastore(pwd,classes,labelIDs);
```

For each pixel label image in the datastore, generate a scene image using the helper function `evaluatePix2PixHD`.

```
for idx = 1:length(cpxds.Files)

    % Get the pixel label image and generated scene image
    [generatedImage,segMap] = evaluatePix2PixHD(cpxds,idx,imageSize,dlnetGenerator);

    % For display, convert the labels from categorical labels to RGB colors
    coloredSegMap = label2rgb(segMap);

    % Display the pixel label image and generated scene image in a montage
    figure
    montage({coloredSegMap generatedImage})
    title(['Custom Pixel Label Image ',num2str(idx),' and Generated Scene Image'])

end
```





Supporting Functions

Model Gradients Function

The `modelGradients` helper function calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the feature matching loss and VGG loss for the generator.

```
function [gradParamsG,gradParamsDScale1,gradParamsDScale2,lossGGAN,lossGFM,lossGVGG,lossD] = mode

    % Compute the image generated by the generator given the input semantic
    % map.
    generatedImage = forward(generator,inputSegMap);

    % Define the loss weights
    lambdaDiscriminator = 1;
```

```
lambdaGenerator = 1;
lambdaFeatureMatching = 5;
lambdaVGG = 5;

% Concatenate the image to be classified and the semantic map
inpDiscriminatorReal = cat(3,inputSegMap,realImage);
inpDiscriminatorGenerated = cat(3,inputSegMap,generatedImage);

% Compute the adversarial loss for the discriminator and the generator
% for first scale.
[DLossScale1,GLossScale1,realPredScale1D,fakePredScale1G] = pix2pixHDAdverserialLoss(inpDisc

% Scale the generated image, the real image, and the input semantic map to
% half size
resizedRealImage = dlresize(realImage, 'Scale',0.5, 'Method',"linear");
resizedGeneratedImage = dlresize(generatedImage,'Scale',0.5,'Method',"linear");
resizedinputSegMap = dlresize(inputSegMap,'Scale',0.5,'Method',"nearest");

% Concatenate the image to be classified and the semantic map
inpDiscriminatorReal = cat(3,resizedinputSegMap,resizedRealImage);
inpDiscriminatorGenerated = cat(3,resizedinputSegMap,resizedGeneratedImage);

% Compute the adversarial loss for the discriminator and the generator
% for second scale.
[DLossScale2,GLossScale2,realPredScale2D,fakePredScale2G] = pix2pixHDAdverserialLoss(inpDisc

% Compute the feature matching loss for first scale.
FMLossScale1 = pix2pixHDFeatureMatchingLoss(realPredScale1D,fakePredScale1G);
FMLossScale1 = FMLossScale1 * lambdaFeatureMatching;

% Compute the feature matching loss for second scale.
FMLossScale2 = pix2pixHDFeatureMatchingLoss(realPredScale2D,fakePredScale2G);
FMLossScale2 = FMLossScale2 * lambdaFeatureMatching;

% Compute the VGG loss
VGGLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG);
VGGLoss = VGGLoss * lambdaVGG;

% Compute the combined generator loss
lossGCombined = GLossScale1 + GLossScale2 + FMLossScale1 + FMLossScale2 + VGGLoss;
lossGCombined = lossGCombined * lambdaGenerator;

% Compute gradients for the generator
gradParamsG = dlgradient(lossGCombined,generator.Learnables,'RetainData',true);

% Compute the combined discriminator loss
lossDCombined = (DLossScale1 + DLossScale2)/2 * lambdaDiscriminator;

% Compute gradients for the discriminator scale1
gradParamsDScale1 = dlgradient(lossDCombined,discriminatorScale1.Learnables,'RetainData',true

% Compute gradients for the discriminator scale2
gradParamsDScale2 = dlgradient(lossDCombined,discriminatorScale2.Learnables);

% Log the values for displaying later
lossD = gather(extractdata(lossDCombined));
lossGGAN = gather(extractdata(GLossScale1 + GLossScale2));
lossGFM  = gather(extractdata(FMLossScale1 + FMLossScale2));
```

```
    lossGVGG = gather(extractdata(VGGLoss));
end
```

Adversarial Loss Function

The helper function `pix2pixHDAdverserialLoss` computes the adversarial loss gradients for the generator and the discriminator. The function also returns feature maps of the real image and synthetic images.

```
function [DLoss,GLoss,realPredFtrsD,genPredFtrsD] = pix2pixHDAdverserialLoss(inpReal,inpGenerated

    % Discriminator layer names containing feature maps
    featureNames = {'act_top','act_mid_1','act_mid_2','act_tail','conv2d_final'};

    % Get the feature maps for the real image from the discriminator
    realPredFtrsD = cell(size(featureNames));
    [realPredFtrsD{:}] = forward(discriminator,inpReal,"Outputs",featureNames);

    % Get the feature maps for the generated image from the discriminator
    genPredFtrsD = cell(size(featureNames));
    [genPredFtrsD{:}] = forward(discriminator,inpGenerated,"Outputs",featureNames);

    % Get the feature map from the final layer to compute the loss
    realPredD = realPredFtrsD{end};
    genPredD = genPredFtrsD{end};

    % Compute the discriminator loss
    DLoss = (1 - realPredD).^2 + (genPredD).^2;
    DLoss = mean(DLoss,"all");

    % Compute the generator loss
    GLoss = (1 - genPredD).^2;
    GLoss = mean(GLoss,"all");
end
```

Feature Matching Loss Function

The helper function `pix2pixHDFeatureMatchingLoss` computes the feature matching loss between a real image and a synthetic image generated by the generator.

```
function featureMatchingLoss = pix2pixHDFeatureMatchingLoss(realPredFtrs,genPredFtrs)

    % Number of features
    numFtrsMaps = numel(realPredFtrs);

    % Initialize the feature matching loss
    featureMatchingLoss = 0;

    for i = 1:numFtrsMaps
        % Get the feature maps of the real image
        a = extractdata(realPredFtrs{i});
        % Get the feature maps of the synthetic image
        b = genPredFtrs{i};

        % Compute the feature matching loss
        featureMatchingLoss = featureMatchingLoss + mean(abs(a - b),"all");
    end
end
```

Perceptual VGG Loss Function

The helper function `pix2pixHDVGGLoss` computes the perceptual VGG loss between a real image and a synthetic image generated by the generator.

```
function vggLoss = pix2pixHDVGGLoss(realImage,generatedImage,netVGG)

    featureWeights = [1.0/32 1.0/16 1.0/8 1.0/4 1.0];

    % Initialize the VGG loss
    vggLoss = 0;

    % Specify the names of the layers with desired feature maps
    featureNames = ["relu1_1","relu2_1","relu3_1","relu4_1","relu5_1"];

    % Extract the feature maps for the real image
    activReal = cell(size(featureNames));
    [activReal{:}] = forward(netVGG,realImage,"Outputs",featureNames);

    % Extract the feature maps for the synthetic image
    activGenerated = cell(size(featureNames));
    [activGenerated{:}] = forward(netVGG,generatedImage,"Outputs",featureNames);

    % Compute the VGG loss
    for i = 1:numel(featureNames)
        vggLoss = vggLoss + featureWeights(i)*mean(abs(activReal{i} - activGenerated{i}),"all");
    end
end
```

References

[1] Wang, Ting-Chun, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs." In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 8798–8807, 2018. https://doi.org/10.1109/CVPR.2018.00917.

[2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

See Also

`vgg19` | `imageDatastore` | `pixelLabelDatastore` | `trainNetwork` | `transform` | `combine` | `minibatchqueue` | `adamupdate` | `predict` | `dlfeval`

More About

- “Preprocess Images for Deep Learning” on page 19-16
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Image Processing Examples

Remove Noise from Color Image Using Pretrained Neural Network

This example shows how to remove Gaussian noise from an RGB image using a denoising convolutional neural network.

Read a color image into the workspace and convert the data to data type `double`. Display the pristine color image.

```
pristineRGB = imread('lighthouse.png');
pristineRGB = im2double(pristineRGB);
imshow(pristineRGB)
title('Pristine Image')
```

Add zero-mean Gaussian white noise with a variance of 0.01 to the image. The `imnoise` function adds noise to each color channel independently. Display the noisy color image.

```
noisyRGB = imnoise(pristineRGB,'gaussian',0,0.01);
imshow(noisyRGB)
title('Noisy Image')
```

The pretrained denoising convolutional neural network, DnCNN, operates on single-channel images. Split the noisy RGB image into its three individual color channels.

```
[noisyR,noisyG,noisyB] = imsplit(noisyRGB);
```

Load the pretrained DnCNN network.

```
net = denoisingNetwork('dncnn');
```

Use the DnCNN network to remove noise from each color channel.

```
denoisedR = denoiseImage(noisyR,net);
denoisedG = denoiseImage(noisyG,net);
denoisedB = denoiseImage(noisyB,net);
```

Recombine the denoised color channels to form the denoised RGB image. Display the denoised color image.

```
denoisedRGB = cat(3,denoisedR,denoisedG,denoisedB);
imshow(denoisedRGB)
title('Denoised Image')
```

Calculate the peak signal-to-noise ratio (PSNR) for the noisy and denoised images. A larger PSNR indicates that noise has a smaller relative signal, and is associated with higher image quality.

```
noisyPSNR = psnr(noisyRGB,pristineRGB);
fprintf('\n The PSNR value of the noisy image is %0.4f.',noisyPSNR);
```

```
 The PSNR value of the noisy image is 20.6395.
```

```
denoisedPSNR = psnr(denoisedRGB,pristineRGB);
fprintf('\n The PSNR value of the denoised image is %0.4f.',denoisedPSNR);
```

```
 The PSNR value of the denoised image is 29.6857.
```

Calculate the structural similarity (SSIM) index for the noisy and denoised images. An SSIM index close to 1 indicates good agreement with the reference image, and higher image quality.

```
noisySSIM = ssim(noisyRGB,pristineRGB);
fprintf('\n The SSIM value of the noisy image is %0.4f.',noisySSIM);
```

```
 The SSIM value of the noisy image is 0.7393.
```

```
denoisedSSIM = ssim(denoisedRGB,pristineRGB);
fprintf('\n The SSIM value of the denoised image is %0.4f.',denoisedSSIM);
```

```
 The SSIM value of the denoised image is 0.9507.
```

In practice, image color channels frequently have correlated noise. To remove correlated image noise, first convert the RGB image to a color space with a luminance channel, such as the L*a*b* color space. Remove noise on the luminance channel only, then convert the denoised image back to the RGB color space.

See Also

`denoisingNetwork` | `denoiseImage` | `rgb2lab` | `lab2rgb` | `psnr` | `ssim` | `imnoise`

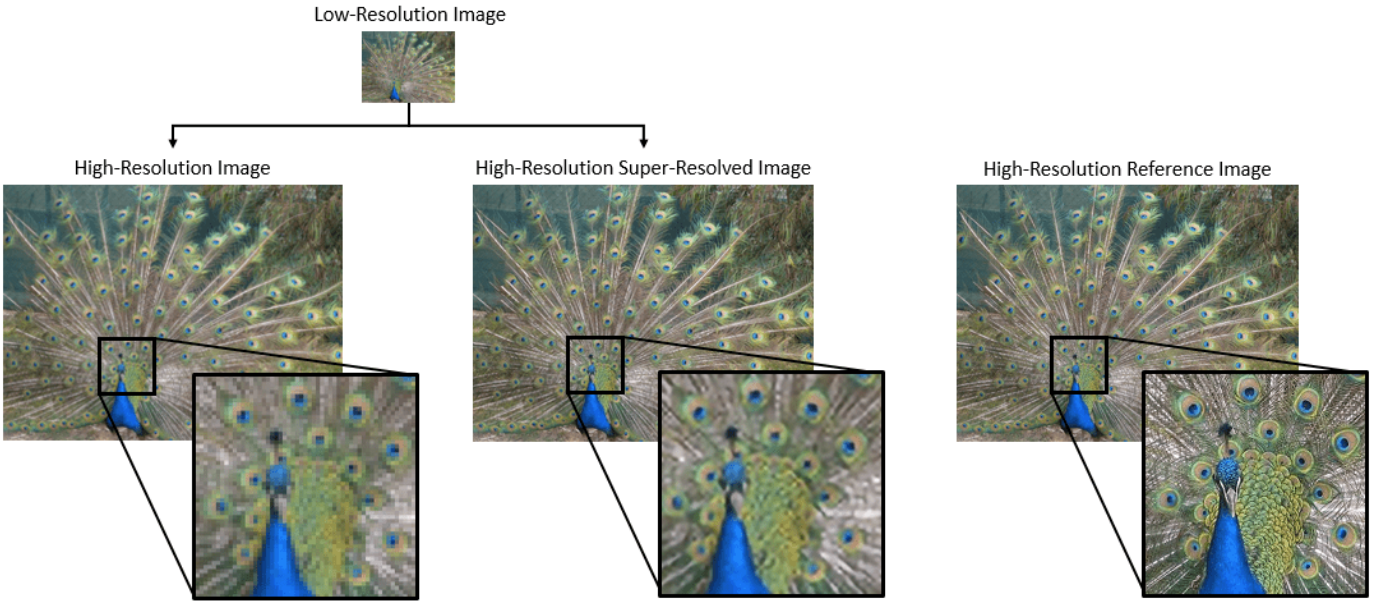
More About

- “Train and Apply Denoising Neural Networks” (Image Processing Toolbox)

Single Image Super-Resolution Using Deep Learning

This example shows how to estimate a high-resolution image from a low-resolution image using a very-deep super-resolution (VDSR) neural network.

Super-resolution is the process of creating high-resolution images from low-resolution images. This example considers single image super-resolution (SISR), where the goal is to recover one high-resolution image from one low-resolution image. SISR is challenging because high-frequency image content typically cannot be recovered from the low-resolution image. Without high-frequency information, the quality of the high-resolution image is limited. Further, SISR is an ill-posed problem because one low-resolution image can yield several possible high-resolution images.



Several techniques, including deep learning algorithms, have been proposed to perform SISR. This example explores one deep learning algorithm for SISR, called very-deep super-resolution (VDSR) [1 on page 9-0].

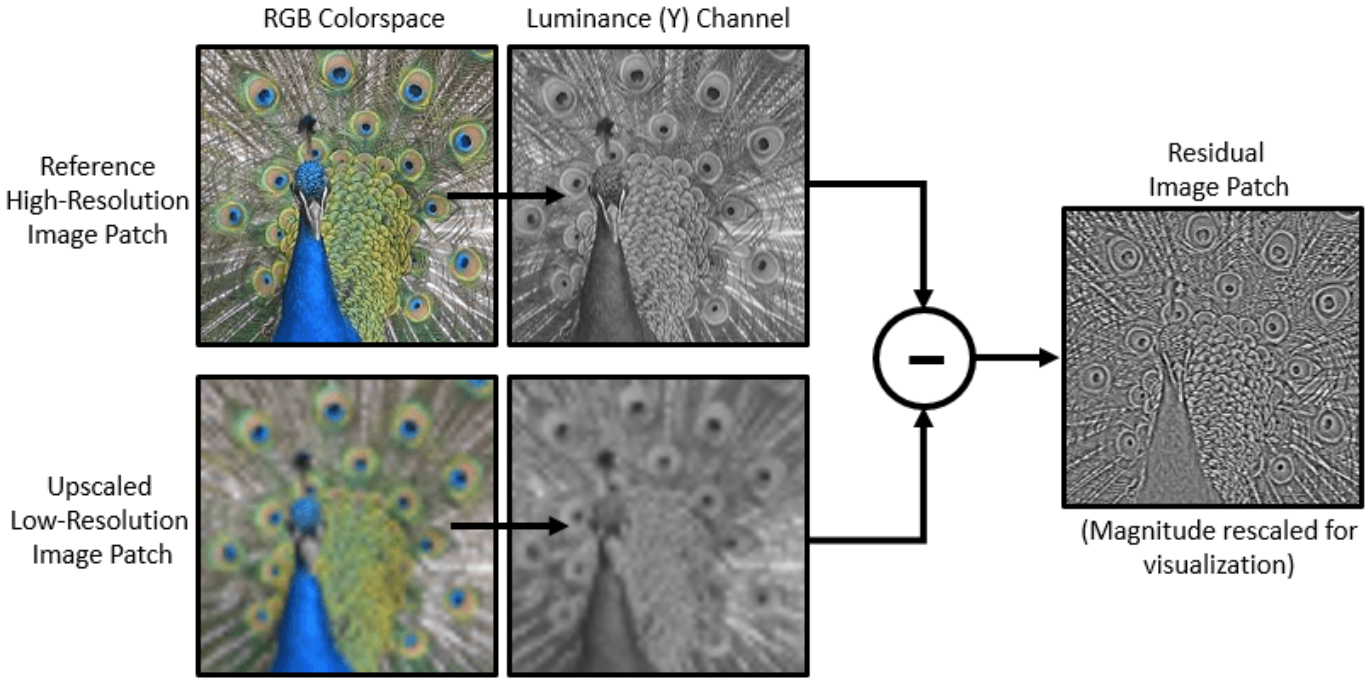
The VDSR Network

VDSR is a convolutional neural network architecture designed to perform single image super-resolution [1 on page 9-0]. The VDSR network learns the mapping between low- and high-resolution images. This mapping is possible because low-resolution and high-resolution images have similar image content and differ primarily in high-frequency details.

VDSR employs a residual learning strategy, meaning that the network learns to estimate a residual image. In the context of super-resolution, a residual image is the difference between a high-resolution reference image and a low-resolution image that has been upscaled using bicubic interpolation to match the size of the reference image. A residual image contains information about the high-frequency details of an image.

The VDSR network detects the residual image from the luminance of a color image. The luminance channel of an image, Y, represents the brightness of each pixel through a linear combination of the red, green, and blue pixel values. In contrast, the two chrominance channels of an image, *Cb* and *Cr*, are different linear combinations of the red, green, and blue pixel values that represent color-

difference information. VDSR is trained using only the luminance channel because human perception is more sensitive to changes in brightness than to changes in color.



If Y_{highres} is the luminance of the high-resolution image and Y_{lowres} is the luminance a low-resolution image that has been upscaled using bicubic interpolation, then the input to the VDSR network is Y_{lowres} and the network learns to predict $Y_{\text{residual}} = Y_{\text{highres}} - Y_{\text{lowres}}$ from the training data.

After the VDSR network learns to estimate the residual image, you can reconstruct high-resolution images by adding the estimated residual image to the upsampled low-resolution image, then converting the image back to the RGB color space.

A scale factor relates the size of the reference image to the size of the low-resolution image. As the scale factor increases, SISR becomes more ill-posed because the low-resolution image loses more information about the high-frequency image content. VDSR solves this problem by using a large receptive field. This example trains a VDSR network with multiple scale factors using scale augmentation. Scale augmentation improves the results at larger scale factors because the network can take advantage of the image context from smaller scale factors. Additionally, the VDSR network can generalize to accept images with noninteger scale factors.

Download Training and Test Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2 on page 9-0]. The data set includes photos of people, animals, cities, and more. The size of the data file is ~1.8 GB. If you do not want to download the training data set, then you can load the pretrained VDSR network by typing `load('trainedVDSR-Epoch-100-ScaleFactors-234.mat');` at the command line. Then, go directly to the Perform Single Image Super-Resolution Using VDSR Network on page 9-0 section in this example.

Use the helper function, `downloadIAPRTC12Data`, to download the data. This function is attached to the example as a supporting file.

```
imagesDir = tempdir;
url = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz';
downloadIAPRTC12Data(url,imagesDir);
```

This example will train the network with a small subset of the IAPR TC-12 Benchmark data. Load the imageCLEF training data. All images are 32-bit JPEG color images.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','02');
exts = {'.jpg','.bmp','.png'};
pristineImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(pristineImages.Files)
```

```
ans = 616
```

Prepare Training Data

To create a training data set, generate pairs of images consisting of upsampled images and the corresponding residual images.

The upsampled images are stored on disk as MAT files in the directory `upsampledDirName`. The computed residual images representing the network responses are stored on disk as MAT files in the directory `residualDirName`. The MAT files are stored as data type `double` for greater precision when training the network.

```
upsampledDirName = [trainImagesDir filesep 'upsampledImages'];
residualDirName = [trainImagesDir filesep 'residualImages'];
```

Use the helper function `createVDSRTrainingSet` to preprocess the training data. This function is attached to the example as a supporting file.

The helper function performs these operations for each pristine image in `trainImages`:

- Convert the image to the YCbCr color space
- Downsize the luminance (Y) channel by different scale factors to create sample low-resolution images, then resize the images to the original size using bicubic interpolation
- Calculate the difference between the pristine and resized images.
- Save the resized and residual images to disk.

```
scaleFactors = [2 3 4];
createVDSRTrainingSet(pristineImages,scaleFactors,upsampledDirName,residualDirName);
```

Define Preprocessing Pipeline for Training Set

In this example, the network inputs are low-resolution images that have been upsampled using bicubic interpolation. The desired network responses are the residual images. Create an image datastore called `upsampledImages` from the collection of input image files. Create an image datastore called `residualImages` from the collection of computed residual image files. Both datastores require a helper function, `matRead`, to read the image data from the image files. This function is attached to the example as a supporting file.

```
upsampledImages = imageDatastore(upsampledDirName,'FileExtensions','.mat','ReadFcn',@matRead);
residualImages = imageDatastore(residualDirName,'FileExtensions','.mat','ReadFcn',@matRead);
```

Create an `imageDataAugmenter` that specifies the parameters of data augmentation. Use data augmentation during training to vary the training data, which effectively increases the amount of available training data. Here, the augmenter specifies random rotation by 90 degrees and random reflections in the *x*-direction.

```
augmenter = imageDataAugmenter( ...
    'RandRotation',@()randi([0,1],1)*90, ...
    'RandXReflection',true);
```

Create a `randomPatchExtractionDatastore` (Image Processing Toolbox) that performs randomized patch extraction from the upsampled and residual image datastores. Patch extraction is the process of extracting a large set of small image patches, or tiles, from a single larger image. This type of data augmentation is frequently used in image-to-image regression problems, where many network architectures can be trained on very small input image sizes. This means that a large number of patches can be extracted from each full-sized image in the original training set, which greatly increases the size of the training set.

```
patchSize = [41 41];
patchesPerImage = 64;
dsTrain = randomPatchExtractionDatastore(upsampledImages,residualImages,patchSize, ...
    "DataAugmentation",augmenter,"PatchesPerImage",patchesPerImage);
```

The resulting datastore, `dsTrain`, provides mini-batches of data to the network at each iteration of the epoch. Preview the result of reading from the datastore.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
      InputImage      ResponseImage
    ______________    ______________

    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
    {41×41 double}    {41×41 double}
```

Set Up VDSR Layers

This example defines the VDSR network using 41 individual layers from Deep Learning Toolbox™, including:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - 2-D convolution layer for convolutional neural networks
- `reluLayer` - Rectified linear unit (ReLU) layer
- `regressionLayer` - Regression output layer for a neural network

The first layer, `imageInputLayer`, operates on image patches. The patch size is based on the network receptive field, which is the spatial image region that affects the response of the top-most layer in the network. Ideally, the network receptive field is the same as the image size so that the field can see all the high-level features in the image. In this case, for a network with D convolutional layers, the receptive field is $(2D+1)$-by-$(2D+1)$.

VDSR has 20 convolutional layers so the receptive field and the image patch size are 41-by-41. The image input layer accepts images with one channel because VDSR is trained using only the luminance channel.

```
networkDepth = 20;
firstLayer = imageInputLayer([41 41 1],'Name','InputLayer','Normalization','none');
```

The image input layer is followed by a 2-D convolutional layer that contains 64 filters of size 3-by-3. The mini-batch size determines the number of filters. Zero-pad the inputs to each convolutional layer so that the feature maps remain the same size as the input after each convolution. He's method [3 on page 9-0] initializes the weights to random values so that there is asymmetry in neuron learning. Each convolutional layer is followed by a ReLU layer, which introduces nonlinearity in the network.

```
convLayer = convolution2dLayer(3,64,'Padding',1, ...
    'WeightsInitializer','he','BiasInitializer','zeros','Name','Conv1');
```

Specify a ReLU layer.

```
relLayer = reluLayer('Name','ReLU1');
```

The middle layers contain 18 alternating convolutional and rectified linear unit layers. Every convolutional layer contains 64 filters of size 3-by-3-by-64, where a filter operates on a 3-by-3 spatial region across 64 channels. As before, a ReLU layer follows every convolutional layer.

```
middleLayers = [convLayer relLayer];
for layerNumber = 2:networkDepth-1
    convLayer = convolution2dLayer(3,64,'Padding',[1 1], ...
        'WeightsInitializer','he','BiasInitializer','zeros', ...
        'Name',['Conv' num2str(layerNumber)]);

    relLayer = reluLayer('Name',['ReLU' num2str(layerNumber)]);
    middleLayers = [middleLayers convLayer relLayer];
end
```

The penultimate layer is a convolutional layer with a single filter of size 3-by-3-by-64 that reconstructs the image.

```
convLayer = convolution2dLayer(3,1,'Padding',[1 1], ...
    'WeightsInitializer','he','BiasInitializer','zeros', ...
    'NumChannels',64,'Name',['Conv' num2str(networkDepth)]);
```

The last layer is a regression layer instead of a ReLU layer. The regression layer computes the mean-squared error between the residual image and network prediction.

```
finalLayers = [convLayer regressionLayer('Name','FinalRegressionLayer')];
```

Concatenate all the layers to form the VDSR network.

```
layers = [firstLayer middleLayers finalLayers];
```

Alternatively, you can use the `vdsrLayers` helper function to create VDSR layers. This function is attached to the example as a supporting file.

```
layers = vdsrLayers;
```

Specify Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` function. The learning rate is initially `0.1` and decreased by a factor of 10 every 10 epochs. Train for 100 epochs.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range, enable gradient clipping by specifying `'GradientThreshold'` as `0.01`, and specify `'GradientThresholdMethod'` to use the L2-norm of the gradients.

```
maxEpochs = 100;
epochIntervals = 1;
initLearningRate = 0.1;
learningRateFactor = 0.1;
l2reg = 0.0001;
miniBatchSize = 64;
options = trainingOptions('sgdm', ...
    'Momentum',0.9, ...
    'InitialLearnRate',initLearningRate, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',10, ...
    'LearnRateDropFactor',learningRateFactor, ...
    'L2Regularization',l2reg, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThresholdMethod','l2norm', ...
    'GradientThreshold',0.01, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

By default, the example loads a pretrained version of the VDSR network that has been trained to super-resolve images for scale factors 2, 3 and 4. The pretrained network enables you to perform super-resolution of test images without waiting for training to complete.

To train the VDSR network, set the `doTraining` variable in the following code to `true`. Train the network using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 6 hours on an NVIDIA Titan X.

```
doTraining = false;
if doTraining
    net = trainNetwork(dsTrain,layers,options);
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    save(strcat("trainedVDSR-",modelDateTime,"-Epoch-",num2str(maxEpochs),"-ScaleFactors-234.mat'
else
    load('trainedVDSR-Epoch-100-ScaleFactors-234.mat');
end
```

Perform Single Image Super-Resolution Using VDSR Network

To perform single image super-resolution (SISR) using the VDSR network, follow the remaining steps of this example. The remainder of the example shows how to:

- Create a sample low-resolution image from a high-resolution reference image.
- Perform SISR on the low-resolution image using bicubic interpolation, a traditional image processing solution that does not rely on deep learning.
- Perform SISR on the low-resolution image using the VDSR neural network.
- Visually compare the reconstructed high-resolution images using bicubic interpolation and VDSR.
- Evaluate the quality of the super-resolved images by quantifying the similarity of the images to the high-resolution reference image.

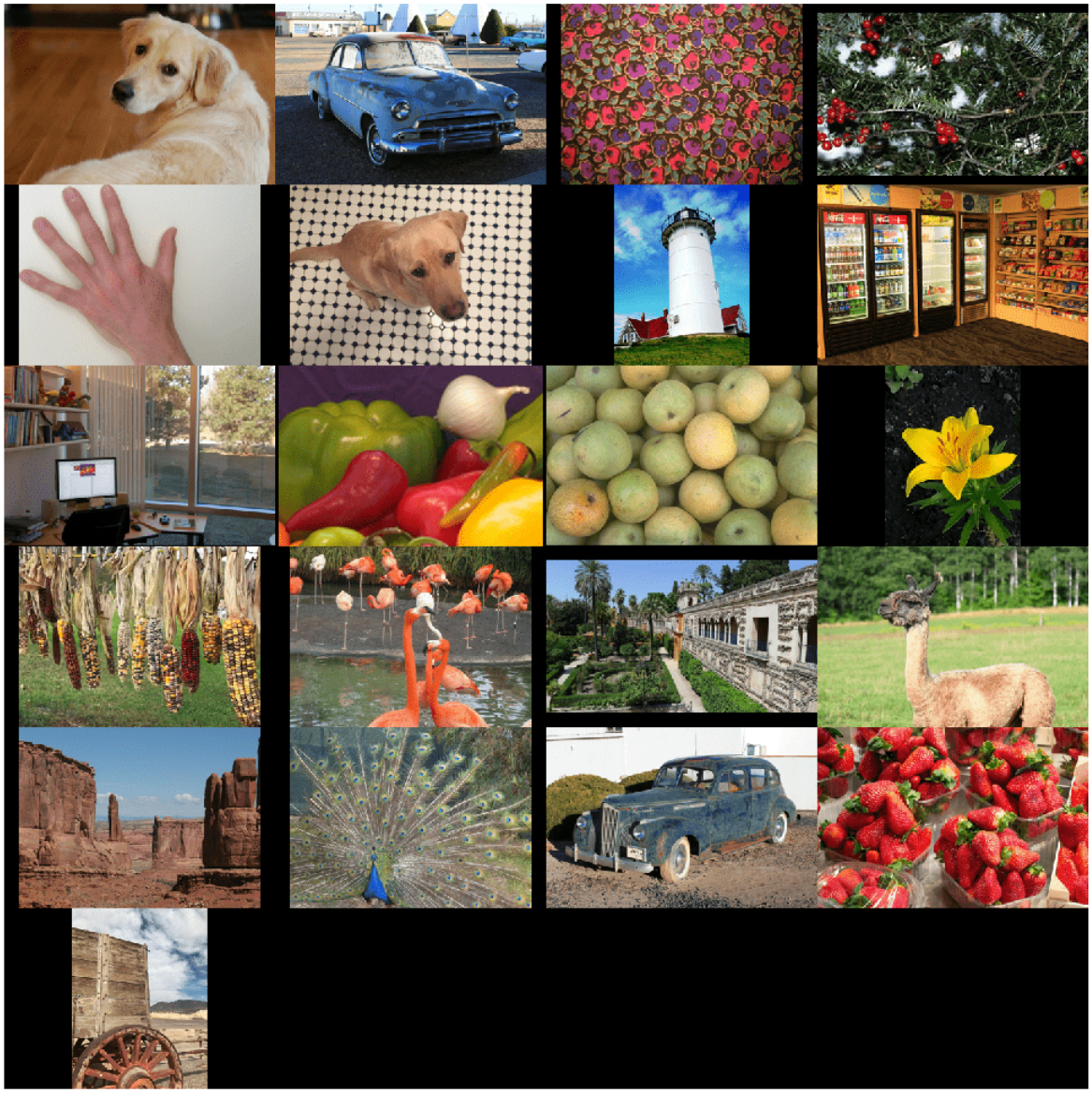
Create Sample Low-Resolution Image

Create a low-resolution image that will be used to compare the results of super-resolution using deep-learning to the result using traditional image processing techniques such as bicubic interpolation. The test data set, `testImages`, contains 21 undistorted images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png', ...
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowlily.jpg', ...
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg', ...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the testing images as a montage.

```
montage(testImages)
```



Select one of the images to use as the reference image for super-resolution. You can optionally use your own high-resolution image as the reference image.

```
indx = 1; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
Ireference = im2double(Ireference);
imshow(Ireference)
title('High-Resolution Reference Image')
```

Create a low-resolution version of the high-resolution reference image by using `imresize` with a scaling factor of 0.25. The high-frequency components of the image are lost during the downscaling.

```
scaleFactor = 0.25;
Ilowres = imresize(Ireference,scaleFactor,'bicubic');
imshow(Ilowres)
title('Low-Resolution Image')
```

Improve Image Resolution Using Bicubic Interpolation

A standard way to increase image resolution without deep learning is to use bicubic interpolation. Upscale the low-resolution image using bicubic interpolation so that the resulting high-resolution image is the same size as the reference image.

```
[nrows,ncols,np] = size(Ireference);
Ibicubic = imresize(Ilowres,[nrows ncols],'bicubic');
imshow(Ibicubic)
title('High-Resolution Image Obtained Using Bicubic Interpolation')
```

Improve Image Resolution Using Pretrained VDSR Network

Recall that VDSR is trained using only the luminance channel of an image because human perception is more sensitive to changes in brightness than to changes in color.

Convert the low-resolution image from the RGB color space to luminance (`Iy`) and chrominance (`Icb` and `Icr`) channels by using the `rgb2ycbcr` (Image Processing Toolbox) function.

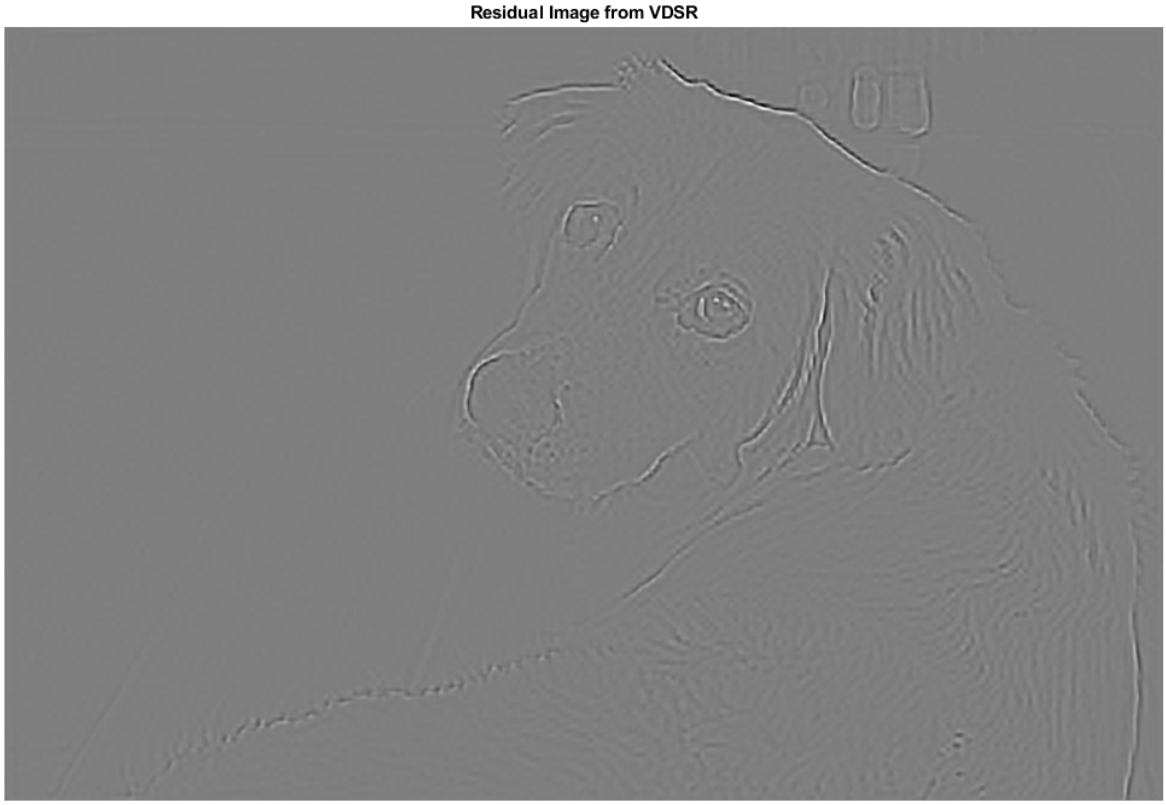
```
Iycbcr = rgb2ycbcr(Ilowres);
Iy = Iycbcr(:,:,1);
Icb = Iycbcr(:,:,2);
Icr = Iycbcr(:,:,3);
```

Upscale the luminance and two chrominance channels using bicubic interpolation. The upsampled chrominance channels, `Icb_bicubic` and `Icr_bicubic`, require no further processing.

```
Iy_bicubic = imresize(Iy,[nrows ncols],'bicubic');
Icb_bicubic = imresize(Icb,[nrows ncols],'bicubic');
Icr_bicubic = imresize(Icr,[nrows ncols],'bicubic');
```

Pass the upscaled luminance component, `Iy_bicubic`, through the trained VDSR network. Observe the `activations` from the final layer (a regression layer). The output of the network is the desired residual image.

```
Iresidual = activations(net,Iy_bicubic,41);
Iresidual = double(Iresidual);
imshow(Iresidual,[])
title('Residual Image from VDSR')
```



Add the residual image to the upscaled luminance component to get the high-resolution VDSR luminance component.

```
Isr = Iy_bicubic + Iresidual;
```

Concatenate the high-resolution VDSR luminance component with the upscaled color components. Convert the image to the RGB color space by using the `ycbcr2rgb` (Image Processing Toolbox) function. The result is the final high-resolution color image using VDSR.

```
Ivdsr = ycbcr2rgb(cat(3,Isr,Icb_bicubic,Icr_bicubic));
imshow(Ivdsr)
title('High-Resolution Image Obtained Using VDSR')
```

Visual and Quantitative Comparison

To get a better visual understanding of the high-resolution images, examine a small region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

```
roi = [320 30 480 400];
```

Crop the high-resolution images to this ROI, and display the result as a montage. The VDSR image has clearer details and sharper edges than the high-resolution image created using bicubic interpolation.

```
montage({imcrop(Ibicubic,roi),imcrop(Ivdsr,roi)})
title('High-Resolution Results Using Bicubic Interpolation (Left) vs. VDSR (Right)');
```

Use image quality metrics to quantitatively compare the high-resolution image using bicubic interpolation to the VDSR image. The reference image is the original high-resolution image, `Ireference`, before preparing the sample low-resolution image.

Measure the peak signal-to-noise ratio (PSNR) of each image against the reference image. Larger PSNR values generally indicate better image quality. See `psnr` (Image Processing Toolbox) for more information about this metric.

```
bicubicPSNR = psnr(Ibicubic,Ireference)
```

```
bicubicPSNR = 38.4747
```

```
vdsrPSNR = psnr(Ivdsr,Ireference)
```

```
vdsrPSNR = 39.2346
```

Measure the structural similarity index (SSIM) of each image. SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. See `ssim` (Image Processing Toolbox) for more information about this metric.

```
bicubicSSIM = ssim(Ibicubic,Ireference)
```

```
bicubicSSIM = 0.9861
```

```
vdsrSSIM = ssim(Ivdsr,Ireference)
```

```
vdsrSSIM = 0.9874
```

Measure perceptual image quality using the Naturalness Image Quality Evaluator (NIQE). Smaller NIQE scores indicate better perceptual quality. See `niqe` (Image Processing Toolbox) for more information about this metric.

```
bicubicNIQE = niqe(Ibicubic)
```

```
bicubicNIQE = 5.1721
```

```
vdsrNIQE = niqe(Ivdsr)

vdsrNIQE = 4.7611
```

Calculate the average PSNR and SSIM of the entire set of test images for the scale factors 2, 3, and 4. For simplicity, you can use the helper function, `superResolutionMetrics`, to compute the average metrics. This function is attached to the example as a supporting file.

```
scaleFactors = [2 3 4];
superResolutionMetrics(net,testImages,scaleFactors);

Results for Scale factor 2

Average PSNR for Bicubic = 31.809683
Average PSNR for VDSR = 31.921784
Average SSIM for Bicubic = 0.938194
Average SSIM for VDSR = 0.949404

Results for Scale factor 3

Average PSNR for Bicubic = 28.170441
Average PSNR for VDSR = 28.563952
Average SSIM for Bicubic = 0.884381
Average SSIM for VDSR = 0.895830

Results for Scale factor 4

Average PSNR for Bicubic = 27.010839
Average PSNR for VDSR = 27.837260
Average SSIM for Bicubic = 0.861604
Average SSIM for VDSR = 0.877132
```

VDSR has better metric scores than bicubic interpolation for each scale factor.

References

[1] Kim, J., J. K. Lee, and K. M. Lee. "Accurate Image Super-Resolution Using Very Deep Convolutional Networks." *Proceedings of the IEEE® Conference on Computer Vision and Pattern Recognition*. 2016, pp. 1646-1654.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

[3] He, K., X. Zhang, S. Ren, and J. Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026-1034.

See Also

`trainingOptions` | `trainNetwork` | `transform` | `combine` | `activations` | `imageDataAugmenter` | `imageDatastore`

More About

- “Datastores for Deep Learning” on page 19-2

- “Preprocess Images for Deep Learning” on page 19-16
- “List of Deep Learning Layers” on page 1-21

JPEG Image Deblocking Using Deep Learning

This example shows how to train a denoising convolutional neural network (DnCNN), then use the network to reduce JPEG compression artifacts in an image.

Image compression is used to reduce the memory footprint of an image. One popular and powerful compression method is employed by the JPEG image format, which uses a quality factor to specify the amount of compression. Reducing the quality value results in higher compression and a smaller memory footprint, at the expense of visual quality of the image.

JPEG compression is *lossy*, meaning that the compression process causes the image to lose information. For JPEG images, this information loss appears as blocking artifacts in the image. As shown in the figure, more compression results in more information loss and stronger artifacts. Textured regions with high-frequency content, such as the grass and clouds, look blurry. Sharp edges, such as the roof of the house and the guardrails atop the lighthouse, exhibit ringing.

JPEG deblocking is the process of reducing the effects of compression artifacts in JPEG images. Several JPEG deblocking methods exist, including more effective methods that use deep learning. This example implements one such deep learning-based method that attempts to minimize the effect of JPEG compression artifacts.

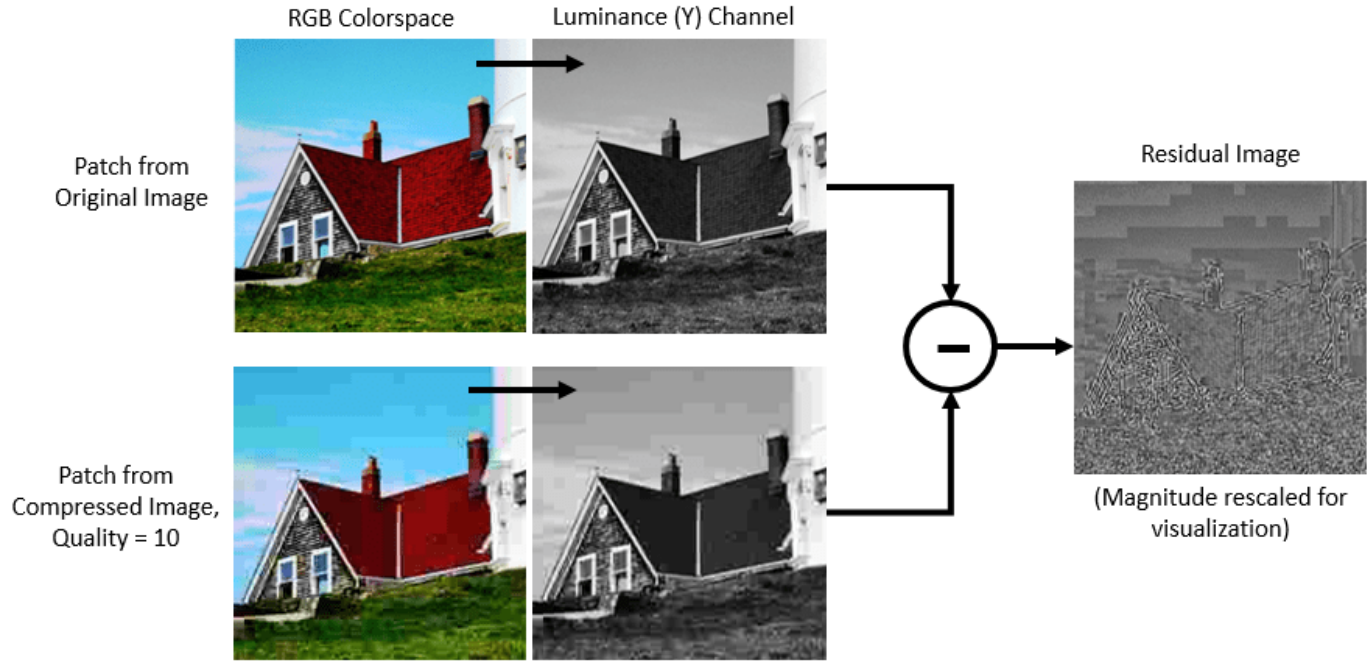
The DnCNN Network

This example uses a built-in deep feed-forward convolutional neural network, called `DnCNN`. The network was primarily designed to remove noise from images. However, the DnCNN architecture can also be trained to remove JPEG compression artifacts or increase image resolution.

The reference paper [1 on page 9-0] employs a residual learning strategy, meaning that the DnCNN network learns to estimate the residual image. A residual image is the difference between a pristine image and a distorted copy of the image. The residual image contains information about the image distortion. For this example, distortion appears as JPEG blocking artifacts.

The DnCNN network is trained to detect the residual image from the luminance of a color image. The luminance channel of an image, *Y*, represents the brightness of each pixel through a linear combination of the red, green, and blue pixel values. In contrast, the two chrominance channels of an

image, *Cb* and *Cr*, are different linear combinations of the red, green, and blue pixel values that represent color-difference information. DnCNN is trained using only the luminance channel because human perception is more sensitive to changes in brightness than changes in color.



If Y_{Original} is the luminance of the pristine image and $Y_{\text{Compressed}}$ is the luminance of the image containing JPEG compression artifacts, then the input to the DnCNN network is $Y_{\text{Compressed}}$ and the network learns to predict $Y_{\text{Residual}} = Y_{\text{Compressed}} - Y_{\text{Original}}$ from the training data.

Once the DnCNN network learns how to estimate a residual image, it can reconstruct an undistorted version of a compressed JPEG image by adding the residual image to the compressed luminance channel, then converting the image back to the RGB color space.

Download Training Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2 on page 9-0]. The data set includes photos of people, animals, cities, and more. The size of the data file is ~1.8 GB. If you do not want to download the training data set needed to train the network, then you can load the pretrained DnCNN network by typing `load('pretrainedJPEGDnCNN.mat')` at the command line. Then, go directly to the Perform JPEG Deblocking Using DnCNN Network on page 9-0 section in this example.

Use the helper function, `downloadIAPRTC12Data`, to download the data. This function is attached to the example as a supporting file.

```
imagesDir = tempdir;
url = "http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz";
downloadIAPRTC12Data(url,imagesDir);
```

This example will train the network with a small subset of the IAPR TC-12 Benchmark data. Load the imageCLEF training data. All images are 32-bit JPEG color images.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','00');
exts = {'.jpg','.bmp','.png'};
imdsPristine = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(imdsPristine.Files)
```

```
ans = 251
```

Prepare Training Data

To create a training data set, read in pristine images and write out images in the JPEG file format with various levels of compression.

Specify the JPEG image quality values used to render image compression artifacts. Quality values must be in the range [0, 100]. Small quality values result in more compression and stronger compression artifacts. Use a denser sampling of small quality values so the training data has a broad range of compression artifacts.

```
JPEGQuality = [5:5:40 50 60 70 80];
```

The compressed images are stored on disk as MAT files in the directory `compressedImagesDir`. The computed residual images are stored on disk as MAT files in the directory `residualImagesDir`. The MAT files are stored as data type `double` for greater precision when training the network.

```
compressedImagesDir = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','compressedImages');
residualImagesDir = fullfile(imagesDir,'iaprtc12','JPEGDeblockingData','residualImages');
```

Use the helper function `createJPEGDeblockingTrainingSet` to preprocess the training data. This function is attached to the example as a supporting file.

For each pristine training image, the helper function writes a copy of the image with quality factor 100 to use as a reference image and copies of the image with each quality factor to use as the network inputs. The function computes the luminance (Y) channel of the reference and compressed images in data type `double` for greater precision when calculating the residual images. The compressed images are stored on disk as .MAT files in the directory `compressedDirName`. The computed residual images are stored on disk as .MAT files in the directory `residualDirName`.

```
[compressedDirName,residualDirName] = createJPEGDeblockingTrainingSet(imdsPristine,JPEGQuality);
```

Create Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts random corresponding patches from two image datastores that contain the network inputs and desired network responses.

In this example, the network inputs are the compressed images. The desired network responses are the residual images. Create an image datastore called `imdsCompressed` from the collection of compressed image files. Create an image datastore called `imdsResidual` from the collection of computed residual image files. Both datastores require a helper function, `matRead`, to read the image data from the image files. This function is attached to the example as a supporting file.

```
imdsCompressed = imageDatastore(compressedDirName,'FileExtensions','.mat','ReadFcn',@matRead);
imdsResidual = imageDatastore(residualDirName,'FileExtensions','.mat','ReadFcn',@matRead);
```

Create an `imageDataAugmenter` that specifies the parameters of data augmentation. Use data augmentation during training to vary the training data, which effectively increases the amount of

available training data. Here, the augmenter specifies random rotation by 90 degrees and random reflections in the x-direction.

```
augmenter = imageDataAugmenter( ...
    'RandRotation',@()randi([0,1],1)*90, ...
    'RandXReflection',true);
```

Create the `randomPatchExtractionDatastore` (Image Processing Toolbox) from the two image datastores. Specify a patch size of 50-by-50 pixels. Each image generates 128 random patches of size 50-by-50 pixels. Specify a mini-batch size of 128.

```
patchSize = 50;
patchesPerImage = 128;
dsTrain = randomPatchExtractionDatastore(imdsCompressed,imdsResidual,patchSize, ...
    'PatchesPerImage',patchesPerImage, ...
    'DataAugmentation',augmenter);
dsTrain.MiniBatchSize = patchesPerImage;
```

The random patch extraction datastore `dsTrain` provides mini-batches of data to the network at iteration of the epoch. Preview the result of reading from the datastore.

```
inputBatch = preview(dsTrain);
disp(inputBatch)
```

```
      InputImage       ResponseImage
    _______________    _______________

    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
    {50×50 double}    {50×50 double}
```

Set up DnCNN Layers

Create the layers of the built-in DnCNN network by using the `dnCNNLayers` (Image Processing Toolbox) function. By default, the network depth (the number of convolution layers) is 20.

```
layers = dnCNNLayers
```

```
layers =
  1x59 Layer array with layers:

     1   'InputLayer'            Image Input           50x50x1 images
     2   'Conv1'                 Convolution           64 3x3x1 convolutions with stride [1  1]
     3   'ReLU1'                 ReLU                  ReLU
     4   'Conv2'                 Convolution           64 3x3x64 convolutions with stride [1  1]
     5   'BNorm2'                Batch Normalization   Batch normalization with 64 channels
     6   'ReLU2'                 ReLU                  ReLU
     7   'Conv3'                 Convolution           64 3x3x64 convolutions with stride [1  1]
     8   'BNorm3'                Batch Normalization   Batch normalization with 64 channels
     9   'ReLU3'                 ReLU                  ReLU
    10   'Conv4'                 Convolution           64 3x3x64 convolutions with stride [1  1]
    11   'BNorm4'                Batch Normalization   Batch normalization with 64 channels
    12   'ReLU4'                 ReLU                  ReLU
```

```
 13   'Conv5'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 14   'BNorm5'                Batch Normalization   Batch normalization with 64 channels
 15   'ReLU5'                 ReLU                  ReLU
 16   'Conv6'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 17   'BNorm6'                Batch Normalization   Batch normalization with 64 channels
 18   'ReLU6'                 ReLU                  ReLU
 19   'Conv7'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 20   'BNorm7'                Batch Normalization   Batch normalization with 64 channels
 21   'ReLU7'                 ReLU                  ReLU
 22   'Conv8'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 23   'BNorm8'                Batch Normalization   Batch normalization with 64 channels
 24   'ReLU8'                 ReLU                  ReLU
 25   'Conv9'                 Convolution           64 3x3x64 convolutions with stride [1  1]
 26   'BNorm9'                Batch Normalization   Batch normalization with 64 channels
 27   'ReLU9'                 ReLU                  ReLU
 28   'Conv10'                Convolution           64 3x3x64 convolutions with stride [1  1]
 29   'BNorm10'               Batch Normalization   Batch normalization with 64 channels
 30   'ReLU10'                ReLU                  ReLU
 31   'Conv11'                Convolution           64 3x3x64 convolutions with stride [1  1]
 32   'BNorm11'               Batch Normalization   Batch normalization with 64 channels
 33   'ReLU11'                ReLU                  ReLU
 34   'Conv12'                Convolution           64 3x3x64 convolutions with stride [1  1]
 35   'BNorm12'               Batch Normalization   Batch normalization with 64 channels
 36   'ReLU12'                ReLU                  ReLU
 37   'Conv13'                Convolution           64 3x3x64 convolutions with stride [1  1]
 38   'BNorm13'               Batch Normalization   Batch normalization with 64 channels
 39   'ReLU13'                ReLU                  ReLU
 40   'Conv14'                Convolution           64 3x3x64 convolutions with stride [1  1]
 41   'BNorm14'               Batch Normalization   Batch normalization with 64 channels
 42   'ReLU14'                ReLU                  ReLU
 43   'Conv15'                Convolution           64 3x3x64 convolutions with stride [1  1]
 44   'BNorm15'               Batch Normalization   Batch normalization with 64 channels
 45   'ReLU15'                ReLU                  ReLU
 46   'Conv16'                Convolution           64 3x3x64 convolutions with stride [1  1]
 47   'BNorm16'               Batch Normalization   Batch normalization with 64 channels
 48   'ReLU16'                ReLU                  ReLU
 49   'Conv17'                Convolution           64 3x3x64 convolutions with stride [1  1]
 50   'BNorm17'               Batch Normalization   Batch normalization with 64 channels
 51   'ReLU17'                ReLU                  ReLU
 52   'Conv18'                Convolution           64 3x3x64 convolutions with stride [1  1]
 53   'BNorm18'               Batch Normalization   Batch normalization with 64 channels
 54   'ReLU18'                ReLU                  ReLU
 55   'Conv19'                Convolution           64 3x3x64 convolutions with stride [1  1]
 56   'BNorm19'               Batch Normalization   Batch normalization with 64 channels
 57   'ReLU19'                ReLU                  ReLU
 58   'Conv20'                Convolution           1 3x3x64 convolutions with stride [1  1]
 59   'FinalRegressionLayer'  Regression Output     mean-squared-error
```

Select Training Options

Train the network using stochastic gradient descent with momentum (SGDM) optimization. Specify the hyperparameter settings for SGDM by using the `trainingOptions` function.

Training a deep network is time-consuming. Accelerate the training by specifying a high learning rate. However, this can cause the gradients of the network to explode or grow uncontrollably, preventing the network from training successfully. To keep the gradients in a meaningful range,

enable gradient clipping by setting `'GradientThreshold'` to `0.005`, and specify `'GradientThresholdMethod'` to use the absolute value of the gradients.

```
maxEpochs = 30;
initLearningRate = 0.1;
l2reg = 0.0001;
batchSize = 64;

options = trainingOptions('sgdm', ...
    'Momentum',0.9, ...
    'InitialLearnRate',initLearningRate, ...
    'LearnRateSchedule','piecewise', ...
    'GradientThresholdMethod','absolute-value', ...
    'GradientThreshold',0.005, ...
    'L2Regularization',l2reg, ...
    'MiniBatchSize',batchSize, ...
    'MaxEpochs',maxEpochs, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

By default, the example loads a pretrained DnCNN network. The pretrained network enables you to perform JPEG deblocking without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the DnCNN network using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 40 hours on an NVIDIA™ Titan X.

```
doTraining = false;
if doTraining
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    [net,info] = trainNetwork(dsTrain,layers,options);
    save(strcat("trainedJPEGDnCNN-",modelDateTime,"-Epoch-",num2str(maxEpochs),".mat"),'net');
else
    load('pretrainedJPEGDnCNN.mat');
end
```

You can now use the DnCNN network to remove JPEG compression artifacts from images.

Perform JPEG Deblocking Using DnCNN Network

To perform JPEG deblocking using DnCNN, follow the remaining steps of this example. The remainder of the example shows how to:

- Create sample test images with JPEG compression artifacts at three different quality levels.
- Remove the compression artifacts using the DnCNN network.
- Visually compare the images before and after deblocking.
- Evaluate the quality of the compressed and deblocked images by quantifying their similarity to the undistorted reference image.

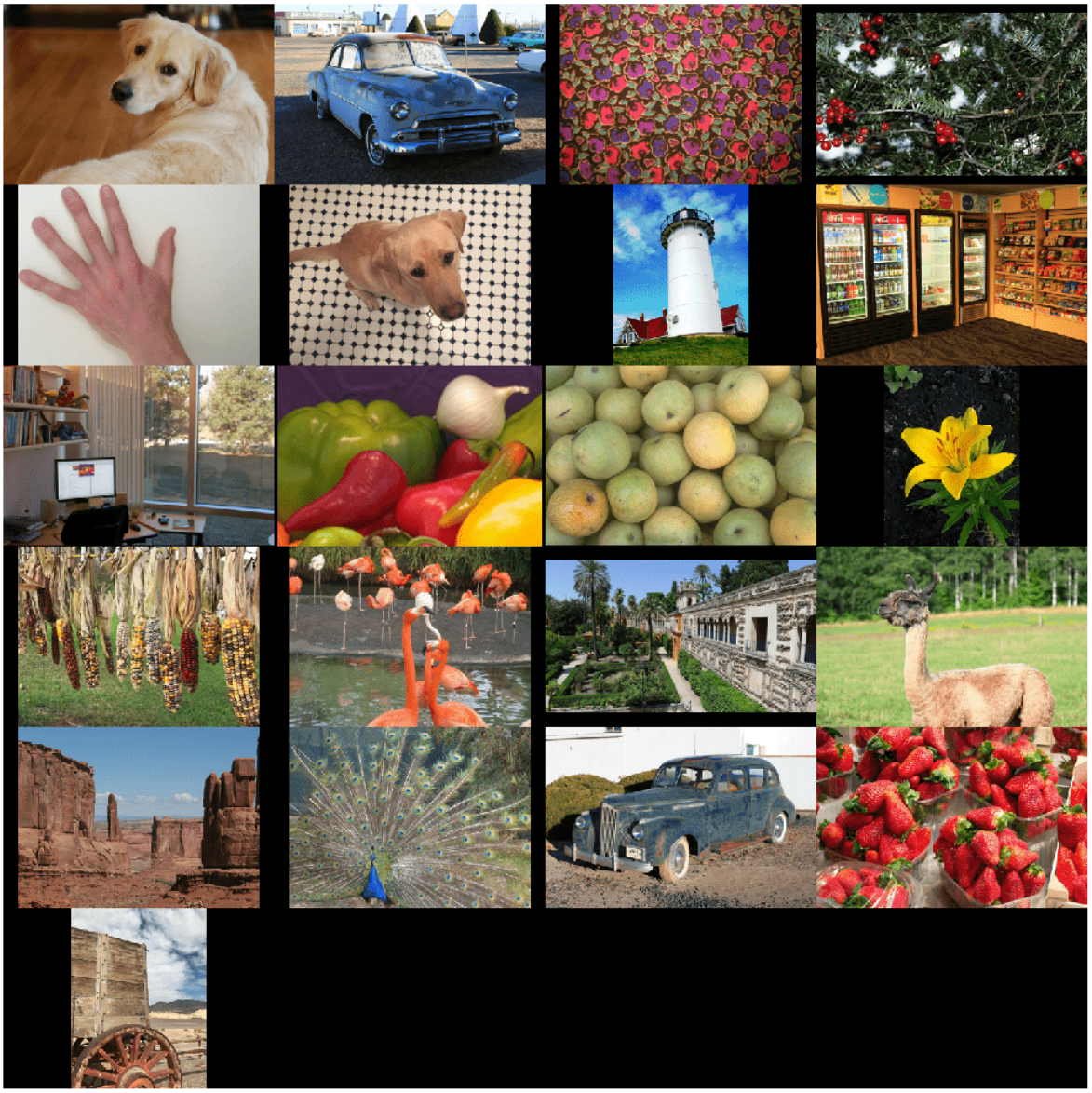
Create Sample Images with Blocking Artifacts

Create sample images to evaluate the result of JPEG image deblocking using the DnCNN network. The test data set, `testImages`, contains 21 undistorted images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png',...
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowlily.jpg',...
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg',...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the testing images as a montage.

```
montage(testImages)
```



Select one of the images to use as the reference image for JPEG deblocking. You can optionally use your own uncompressed image as the reference image.

```
indx = 7; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
imshow(Ireference)
title('Uncompressed Reference Image')
```

Create three compressed test images with the JPEG `Quality` values of 10, 20, and 50.

```
imwrite(Ireference,fullfile(tempdir,'testQuality10.jpg'),'Quality',10);
imwrite(Ireference,fullfile(tempdir,'testQuality20.jpg'),'Quality',20);
imwrite(Ireference,fullfile(tempdir,'testQuality50.jpg'),'Quality',50);
```

Preprocess Compressed Images

Read the compressed versions of the image into the workspace.

```
I10 = imread(fullfile(tempdir,'testQuality10.jpg'));
I20 = imread(fullfile(tempdir,'testQuality20.jpg'));
I50 = imread(fullfile(tempdir,'testQuality50.jpg'));
```

Display the compressed images as a montage.

```
montage({I50,I20,I10},'Size',[1 3])
title('JPEG-Compressed Images with Quality Factor: 50, 20 and 10 (left to right)')
```

Recall that DnCNN is trained using only the luminance channel of an image because human perception is more sensitive to changes in brightness than changes in color. Convert the JPEG-compressed images from the RGB color space to the YCbCr color space using the `rgb2ycbcr` (Image Processing Toolbox) function.

```
I10ycbcr = rgb2ycbcr(I10);
I20ycbcr = rgb2ycbcr(I20);
I50ycbcr = rgb2ycbcr(I50);
```

Apply the DnCNN Network

In order to perform the forward pass of the network, use the `denoiseImage` (Image Processing Toolbox) function. This function uses exactly the same training and testing procedures for denoising an image. You can think of the JPEG compression artifacts as a type of image noise.

```
I10y_predicted = denoiseImage(I10ycbcr(:,:,1),net);
I20y_predicted = denoiseImage(I20ycbcr(:,:,1),net);
I50y_predicted = denoiseImage(I50ycbcr(:,:,1),net);
```

The chrominance channels do not need processing. Concatenate the deblocked luminance channel with the original chrominance channels to obtain the deblocked image in the YCbCr color space.

```
I10ycbcr_predicted = cat(3,I10y_predicted,I10ycbcr(:,:,2:3));
I20ycbcr_predicted = cat(3,I20y_predicted,I20ycbcr(:,:,2:3));
I50ycbcr_predicted = cat(3,I50y_predicted,I50ycbcr(:,:,2:3));
```

Convert the deblocked YCbCr image to the RGB color space by using the `ycbcr2rgb` (Image Processing Toolbox) function.

```
I10_predicted = ycbcr2rgb(I10ycbcr_predicted);
I20_predicted = ycbcr2rgb(I20ycbcr_predicted);
I50_predicted = ycbcr2rgb(I50ycbcr_predicted);
```

Display the deblocked images as a montage.

```
montage({I50_predicted,I20_predicted,I10_predicted},'Size',[1 3])
title('Deblocked Images with Quality Factor 50, 20 and 10 (Left to Right)')
```

To get a better visual understanding of the improvements, examine a smaller region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

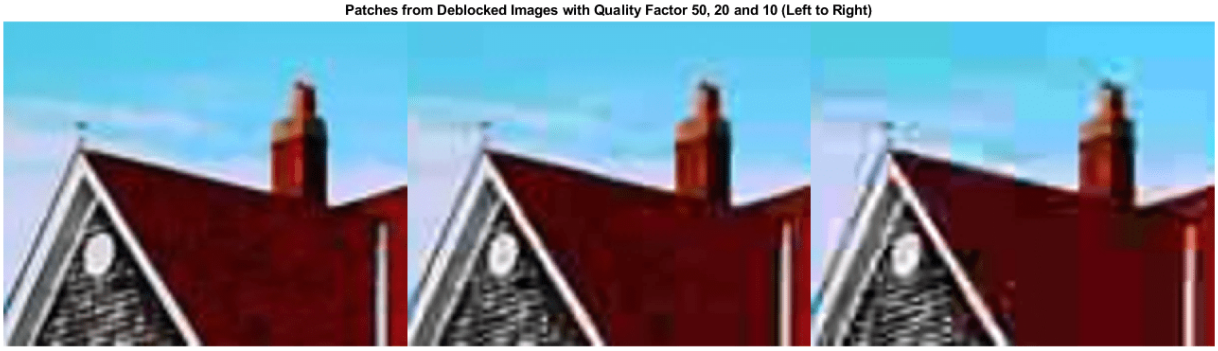
```
roi = [30 440 100 80];
```

Crop the compressed images to this ROI, and display the result as a montage.

```
i10 = imcrop(I10,roi);
i20 = imcrop(I20,roi);
i50 = imcrop(I50,roi);
montage({i50 i20 i10},'Size',[1 3])
title('Patches from JPEG-Compressed Images with Quality Factor 50, 20 and 10 (Left to Right)')
```

Crop the deblocked images to this ROI, and display the result as a montage.

```
i10predicted = imcrop(I10_predicted,roi);
i20predicted = imcrop(I20_predicted,roi);
i50predicted = imcrop(I50_predicted,roi);
montage({i50predicted,i20predicted,i10predicted},'Size',[1 3])
title('Patches from Deblocked Images with Quality Factor 50, 20 and 10 (Left to Right)')
```



Quantitative Comparison

Quantify the quality of the deblocked images through four metrics. You can use the `displayJPEGResults` helper function to compute these metrics for compressed and deblocked images at the quality factors 10, 20, and 50. This function is attached to the example as a supporting file.

- Structural Similarity Index (SSIM). SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. Here, the reference image is the undistorted original image, `Ireference`, before JPEG compression. See `ssim` (Image Processing Toolbox) for more information about this metric.
- Peak signal-to-noise ratio (PSNR). The larger the PSNR value, the stronger the signal compared to the distortion. See `psnr` (Image Processing Toolbox) for more information about this metric.
- Naturalness Image Quality Evaluator (NIQE). NIQE measures perceptual image quality using a model trained from natural scenes. Smaller NIQE scores indicate better perceptual quality. See `niqe` (Image Processing Toolbox) for more information about this metric.

- Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE). BRISQUE measures perceptual image quality using a model trained from natural scenes with image distortion. Smaller BRISQUE scores indicate better perceptual quality. See `brisque` (Image Processing Toolbox) for more information about this metric.

```
displayJPEGResults(Ireference,I10,I20,I50,I10_predicted,I20_predicted,I50_predicted)
```

```
----------------------------------------
SSIM Comparison
===============
I10: 0.90624    I10_predicted: 0.91286
I20: 0.94904    I20_predicted: 0.95444
I50: 0.97238    I50_predicted: 0.97482
----------------------------------------
PSNR Comparison
===============
I10: 26.6046    I10_predicted: 27.0793
I20: 28.8015    I20_predicted: 29.3378
I50: 31.4512    I50_predicted: 31.8584
----------------------------------------
NIQE Comparison
===============
I10: 7.2194    I10_predicted: 3.9478
I20: 4.5158    I20_predicted: 3.0685
I50: 2.8874    I50_predicted: 2.4106
NOTE: Smaller NIQE score signifies better perceptual quality
----------------------------------------
BRISQUE Comparison
==================
I10: 52.372    I10_predicted: 38.9271
I20: 45.3772    I20_predicted: 30.8991
I50: 27.7093    I50_predicted: 24.3845
NOTE: Smaller BRISQUE score signifies better perceptual quality
```

References

[1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." *IEEE® Transactions on Image Processing*. Feb 2017.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

See Also

`rgb2ycbcr` | `ycbcr2rgb` | `dnCNNLayers` | `denoiseImage` | `trainingOptions` | `trainNetwork` | `randomPatchExtractionDatastore`

More About

- “Preprocess Images for Deep Learning” on page 19-16
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

Image Processing Operator Approximation Using Deep Learning

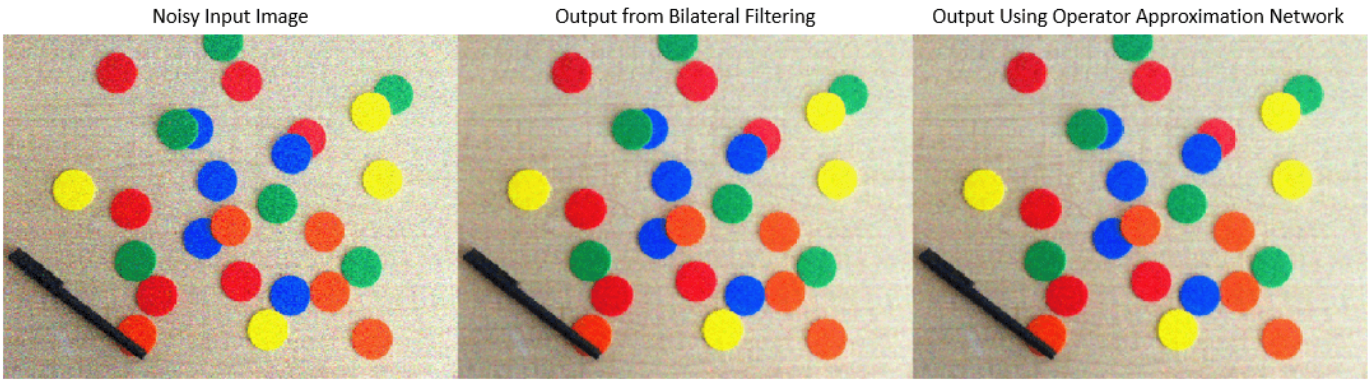
This example shows how to approximate an image filtering operation using a multiscale context aggregation network (CAN).

Operator approximation finds alternative ways to process images such that the result resembles the output from a conventional image processing operation or pipeline. The goal of operator approximation is often to reduce the time required to process an image.

Several classical and deep learning techniques have been proposed to perform operator approximation. Some classical techniques improve the efficiency of a single algorithm but cannot be generalized to other operations. Another common technique approximates a wide range of operations by applying the operator to a low resolution copy of an image, but the loss of high-frequency content limits the accuracy of the approximation.

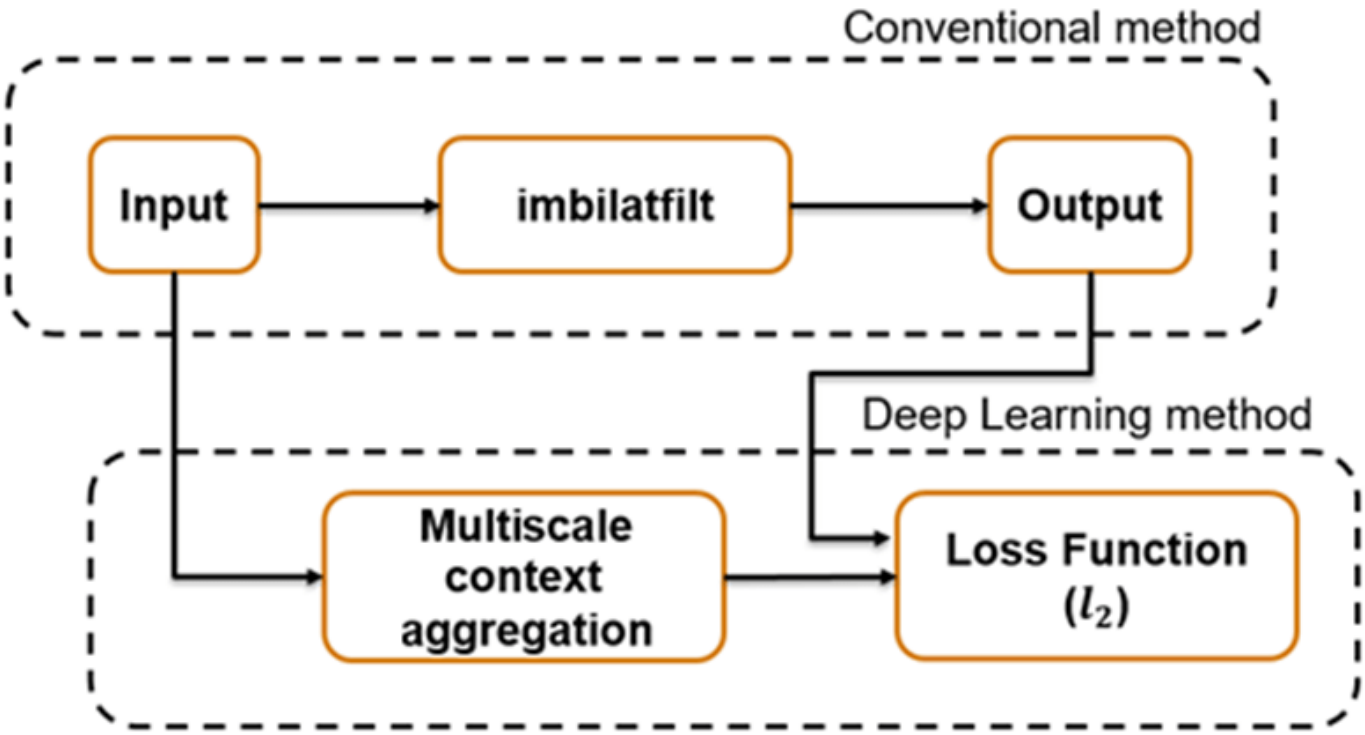
Deep learning solutions enable the approximation of more general and complex operations. For example, the multiscale context aggregation network (CAN) presented by Q. Chen [1 on page 9-0] can approximate multiscale tone mapping, photographic style transfer, nonlocal dehazing, and pencil drawing. Multiscale CAN trains on full-resolution images for greater accuracy in processing high-frequency details. After the network is trained, the network can bypass the conventional processing operation and process images directly.

This example explores how to train a multiscale CAN to approximate a bilateral image filtering operation, which reduces image noise while preserving edge sharpness. The example presents the complete training and inference workflow, which includes the process of creating a training datastore, selecting training options, training the network, and using the network to process test images.



The Operator Approximation Network

The multiscale CAN is trained to minimize the l_2 loss between the conventional output of an image processing operation and the network response after processing the input image using multiscale context aggregation. Multiscale context aggregation looks for information about each pixel from across the entire image, rather than limiting the search to a small neighborhood surrounding the pixel.



To help the network learn global image properties, the multiscale CAN architecture has a large receptive field. The first and last layers have the same size because the operator should not change the size of the image. Successive intermediate layers are dilated by exponentially increasing scale factors (hence the "multiscale" nature of the CAN). Dilation enables the network to look for spatially separated features at various spatial frequencies, without reducing the resolution of the image. After each convolution layer, the network uses adaptive normalization to balance the impact of batch normalization and the identity mapping on the approximated operator.

Download Training and Test Data

Download the IAPR TC-12 Benchmark, which consists of 20,000 still natural images [2 on page 9-0]. The data set includes photos of people, animals, cities, and more. The size of the data file is ~1.8 GB. If you do not want to download the training data set needed to train the network, then you can load the pretrained CAN by typing `load('trainedOperatorLearning-Epoch-181.mat');` at the command line. Then, go directly to the Perform Bilateral Filtering Approximation Using Multiscale CAN on page 9-0 section in this example.

```
imagesDir = tempdir;
url_1 = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.tgz';
downloadIAPRTC12Data(url_1,imagesDir);
```

This example trains the network with small subset of the IAPRTC-12 Benchmark data.

```
trainImagesDir = fullfile(imagesDir,'iaprtc12','images','39');
exts = {'.jpg','.bmp','.png'};
pristineImages = imageDatastore(trainImagesDir,'FileExtensions',exts);
```

List the number of training images.

```
numel(pristineImages.Files)
```

```
ans = 916
```

Prepare Training Data

To create a training data set, read in pristine images and write out images that have been bilateral filtered. The filtered images are stored on disk in the directory specified by `preprocessDataDir`.

```
preprocessDataDir = [trainImagesDir filesep 'preprocessedDataset'];
```

Use the helper function `bilateralFilterDataset` to preprocess the training data. This function is attached to the example as a supporting file.

The helper function performs these operations for each pristine image in `inputImages`:

- Calculate the degree of smoothing for bilateral filtering. Smoothing the filtered image reduces image noise.
- Perform bilateral filtering using `imbilatfilt` (Image Processing Toolbox).
- Save the filtered image to disk using `imwrite`.

```
bilateralFilterDataset(pristineImages,preprocessDataDir);
```

Define Random Patch Extraction Datastore for Training

Use a random patch extraction datastore to feed the training data to the network. This datastore extracts random corresponding patches from two image datastores that contain the network inputs and desired network responses.

In this example, the network inputs are the pristine images in `pristineImages`. The desired network responses are the processed images after bilateral filtering. Create an image datastore called `bilatFilteredImages` from the collection of bilateral filtered image files.

```
bilatFilteredImages = imageDatastore(preprocessDataDir,'FileExtensions',exts);
```

Create a `randomPatchExtractionDatastore` (Image Processing Toolbox) from the two image datastores. Specify a patch size of 256-by-256 pixels. Specify '`PatchesPerImage`' to extract one randomly-positioned patch from each pair of images during training. Specify a mini-batch size of one.

```
miniBatchSize = 1;
patchSize = [256 256];
dsTrain = randomPatchExtractionDatastore(pristineImages,bilatFilteredImages,patchSize, ....
    'PatchesPerImage',1);
dsTrain.MiniBatchSize = miniBatchSize;
```

The `randomPatchExtractionDatastore` provides mini-batches of data to the network at each iteration of the epoch. Perform a read operation on the datastore to explore the data.

```
inputBatch = read(dsTrain);
disp(inputBatch)
```

```
       InputImage          ResponseImage
    ________________    ________________

    {256×256×3 uint8}    {256×256×3 uint8}
```

Set Up Multiscale CAN Layers

This example defines the multiscale CAN using layers from Deep Learning Toolbox™, including:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - 2D convolution layer for convolutional neural networks
- `batchNormalizationLayer` - Batch normalization layer
- `leakyReluLayer` - Leaky rectified linear unit layer
- `regressionLayer` - Regression output layer for a neural network

Two custom scale layers are added to implement an adaptive batch normalization layer. These layers are attached as supporting files to this example.

- **adaptiveNormalizationMu** - Scale layer that adjusts the strengths of the batch-normalization branch
- **adaptiveNormalizationLambda** - Scale layer that adjusts the strengths of the identity branch

The first layer, `imageInputLayer`, operates on image patches. The patch size is based on the network receptive field, which is the spatial image region that affects the response of top-most layer in the network. Ideally, the network receptive field is the same as the image size so that it can see all the high level features in the image. For a bilateral filter, the approximation image patch size is fixed to 256-by-256.

```
networkDepth = 10;
numberOfFilters = 32;
firstLayer = imageInputLayer([256 256 3],'Name','InputLayer','Normalization','none');
```

The image input layer is followed by a 2-D convolution layer that contains 32 filters of size 3-by-3. Zero-pad the inputs to each convolution layer so that feature maps remain the same size as the input after each convolution. Initialize the weights to the identity matrix.

```
Wgts = zeros(3,3,3,numberOfFilters);
for ii = 1:3
    Wgts(2,2,ii,ii) = 1;
end
convolutionLayer = convolution2dLayer(3,numberOfFilters,'Padding',1, ...
    'Weights',Wgts,'Name','Conv1');
```

Each convolution layer is followed by a batch normalization layer and an adaptive normalization scale layer that adjusts the strengths of the batch-normalization branch. Later, this example will create the corresponding adaptive normalization scale layer that adjusts the strength of the identity branch. For now, follow the `adaptiveNormalizationMu` layer with an addition layer. Finally, specify a leaky ReLU layer with a scalar multiplier of `0.2` for negative inputs.

```
batchNorm = batchNormalizationLayer('Name','BN1');
adaptiveMu = adaptiveNormalizationMu(numberOfFilters,'Mu1');
addLayer = additionLayer(2,'Name','add1');
leakyrelLayer = leakyReluLayer(0.2,'Name','Leaky1');
```

Specify the middle layers of the network following the same pattern. Successive convolution layers have a dilation factor that scales exponentially with the network depth.

```
middleLayers = [convolutionLayer batchNorm adaptiveMu addLayer leakyrelLayer];

Wgts = zeros(3,3,numberOfFilters,numberOfFilters);
for ii = 1:numberOfFilters
    Wgts(2,2,ii,ii) = 1;
end
```

```
for layerNumber = 2:networkDepth-2
    dilationFactor = 2^(layerNumber-1);
    padding = dilationFactor;
    conv2dLayer = convolution2dLayer(3,numberOfFilters, ...
        'Padding',padding,'DilationFactor',dilationFactor, ...
        'Weights',Wgts,'Name',['Conv' num2str(layerNumber)]);
    batchNorm = batchNormalizationLayer('Name',['BN' num2str(layerNumber)]);
    adaptiveMu = adaptiveNormalizationMu(numberOfFilters,['Mu' num2str(layerNumber)]);
    addLayer = additionLayer(2,'Name',['add' num2str(layerNumber)]);
    leakyrelLayer = leakyReluLayer(0.2, 'Name', ['Leaky' num2str(layerNumber)]);
    middleLayers = [middleLayers conv2dLayer batchNorm adaptiveMu addLayer leakyrelLayer];
end
```

Do not apply a dilation factor to the second-to-last convolution layer.

```
conv2dLayer = convolution2dLayer(3,numberOfFilters, ...
    'Padding',1,'Weights',Wgts,'Name','Conv9');

batchNorm = batchNormalizationLayer('Name','AN9');
adaptiveMu = adaptiveNormalizationMu(numberOfFilters,'Mu9');
addLayer = additionLayer(2,'Name','add9');
leakyrelLayer = leakyReluLayer(0.2,'Name','Leaky9');
middleLayers = [middleLayers conv2dLayer batchNorm adaptiveMu addLayer leakyrelLayer];
```

The last convolution layer has a single filter of size 1-by-1-by-32-by-3 that reconstructs the image.

```
Wgts = sqrt(2/(9*numberOfFilters))*randn(1,1,numberOfFilters,3);
conv2dLayer = convolution2dLayer(1,3,'NumChannels',numberOfFilters, ...
    'Weights',Wgts,'Name','Conv10');
```

The last layer is a regression layer instead of a leaky ReLU layer. The regression layer computes the mean-squared error between the bilateral-filtered image and the network prediction.

```
finalLayers = [conv2dLayer
    regressionLayer('Name','FinalRegressionLayer')
];
```

Concatenate all the layers.

```
layers = [firstLayer middleLayers finalLayers'];
lgraph = layerGraph(layers);
```

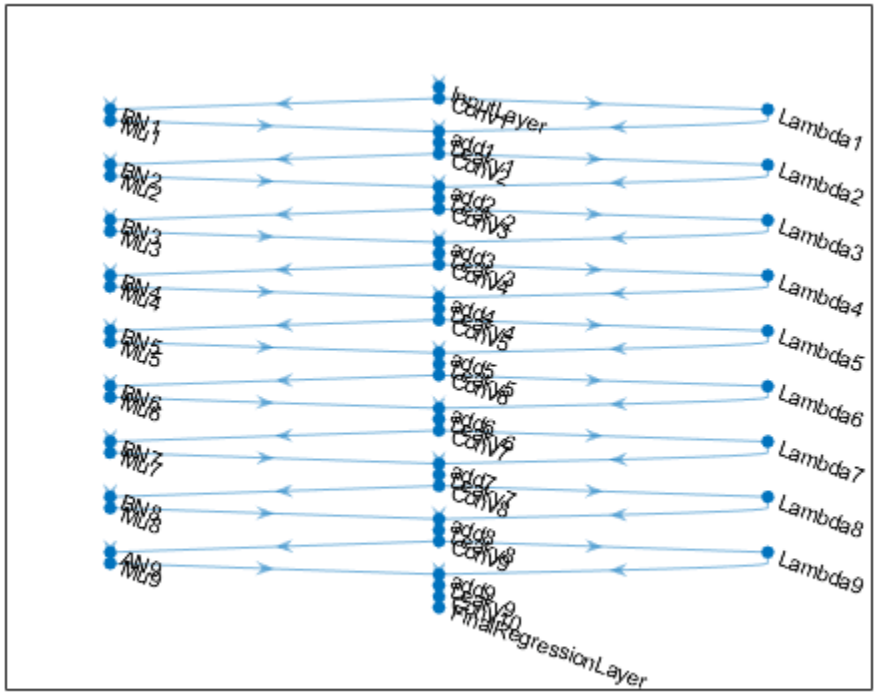
Create skip connections, which act as the identity branch for the adaptive normalization equation. Connect the skip connections to the addition layers.

```
skipConv1 = adaptiveNormalizationLambda(numberOfFilters,'Lambda1');
skipConv2 = adaptiveNormalizationLambda(numberOfFilters,'Lambda2');
skipConv3 = adaptiveNormalizationLambda(numberOfFilters,'Lambda3');
skipConv4 = adaptiveNormalizationLambda(numberOfFilters,'Lambda4');
skipConv5 = adaptiveNormalizationLambda(numberOfFilters,'Lambda5');
skipConv6 = adaptiveNormalizationLambda(numberOfFilters,'Lambda6');
skipConv7 = adaptiveNormalizationLambda(numberOfFilters,'Lambda7');
skipConv8 = adaptiveNormalizationLambda(numberOfFilters,'Lambda8');
skipConv9 = adaptiveNormalizationLambda(numberOfFilters,'Lambda9');

lgraph = addLayers(lgraph,skipConv1);
lgraph = connectLayers(lgraph,'Conv1','Lambda1');
lgraph = connectLayers(lgraph,'Lambda1','add1/in2');
```

```
lgraph = addLayers(lgraph,skipConv2);
lgraph = connectLayers(lgraph,'Conv2','Lambda2');
lgraph = connectLayers(lgraph,'Lambda2','add2/in2');

lgraph = addLayers(lgraph,skipConv3);
lgraph = connectLayers(lgraph,'Conv3','Lambda3');
lgraph = connectLayers(lgraph,'Lambda3','add3/in2');

lgraph = addLayers(lgraph,skipConv4);
lgraph = connectLayers(lgraph,'Conv4','Lambda4');
lgraph = connectLayers(lgraph,'Lambda4','add4/in2');

lgraph = addLayers(lgraph,skipConv5);
lgraph = connectLayers(lgraph,'Conv5','Lambda5');
lgraph = connectLayers(lgraph,'Lambda5','add5/in2');

lgraph = addLayers(lgraph,skipConv6);
lgraph = connectLayers(lgraph,'Conv6','Lambda6');
lgraph = connectLayers(lgraph,'Lambda6','add6/in2');

lgraph = addLayers(lgraph,skipConv7);
lgraph = connectLayers(lgraph,'Conv7','Lambda7');
lgraph = connectLayers(lgraph,'Lambda7','add7/in2');

lgraph = addLayers(lgraph,skipConv8);
lgraph = connectLayers(lgraph,'Conv8','Lambda8');
lgraph = connectLayers(lgraph,'Lambda8','add8/in2');

lgraph = addLayers(lgraph,skipConv9);
lgraph = connectLayers(lgraph,'Conv9','Lambda9');
lgraph = connectLayers(lgraph,'Lambda9','add9/in2');
```

Plot the layer graph.

```
plot(lgraph)
```



Specify Training Options

Train the network using the Adam optimizer. Specify the hyperparameter settings by using the `trainingOptions` function. Use the default values of `0.9` for '`Momentum`' and `0.0001` for '`L2Regularization`' (weight decay). Specify a constant learning rate of `0.0001`. Train for 181 epochs.

```
maxEpochs = 181;
initLearningRate = 0.0001;
miniBatchSize = 1;

options = trainingOptions('adam', ...
    'InitialLearnRate',initLearningRate, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

By default, the example loads a pretrained multiscale CAN that approximates a bilateral filter. The pretrained network enables you to perform an approximation of bilateral filtering without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the multiscale CAN using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 15 hours on an NVIDIA™ Titan X.

```
doTraining = false;
if doTraining
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    net = trainNetwork(dsTrain,lgraph,options);
    save(strcat("trainedOperatorLearning-",modelDateTime,"-Epoch-",num2str(maxEpochs),".mat"),'ne
else
    load('trainedOperatorLearning-Epoch-181.mat');
end
```

Perform Bilateral Filtering Approximation Using Multiscale CAN

To process an image using a trained multiscale CAN network that approximates a bilateral filter, follow the remaining steps of this example. The remainder of the example shows how to:

- Create a sample noisy input image from a reference image.
- Perform conventional bilateral filtering of the noisy image using the `imbilatfilt` (Image Processing Toolbox) function.
- Perform an approximation to bilateral filtering on the noisy image using the CAN.
- Visually compare the denoised images from operator approximation and conventional bilateral filtering.
- Evaluate the quality of the denoised images by quantifying the similarity of the images to the pristine reference image.

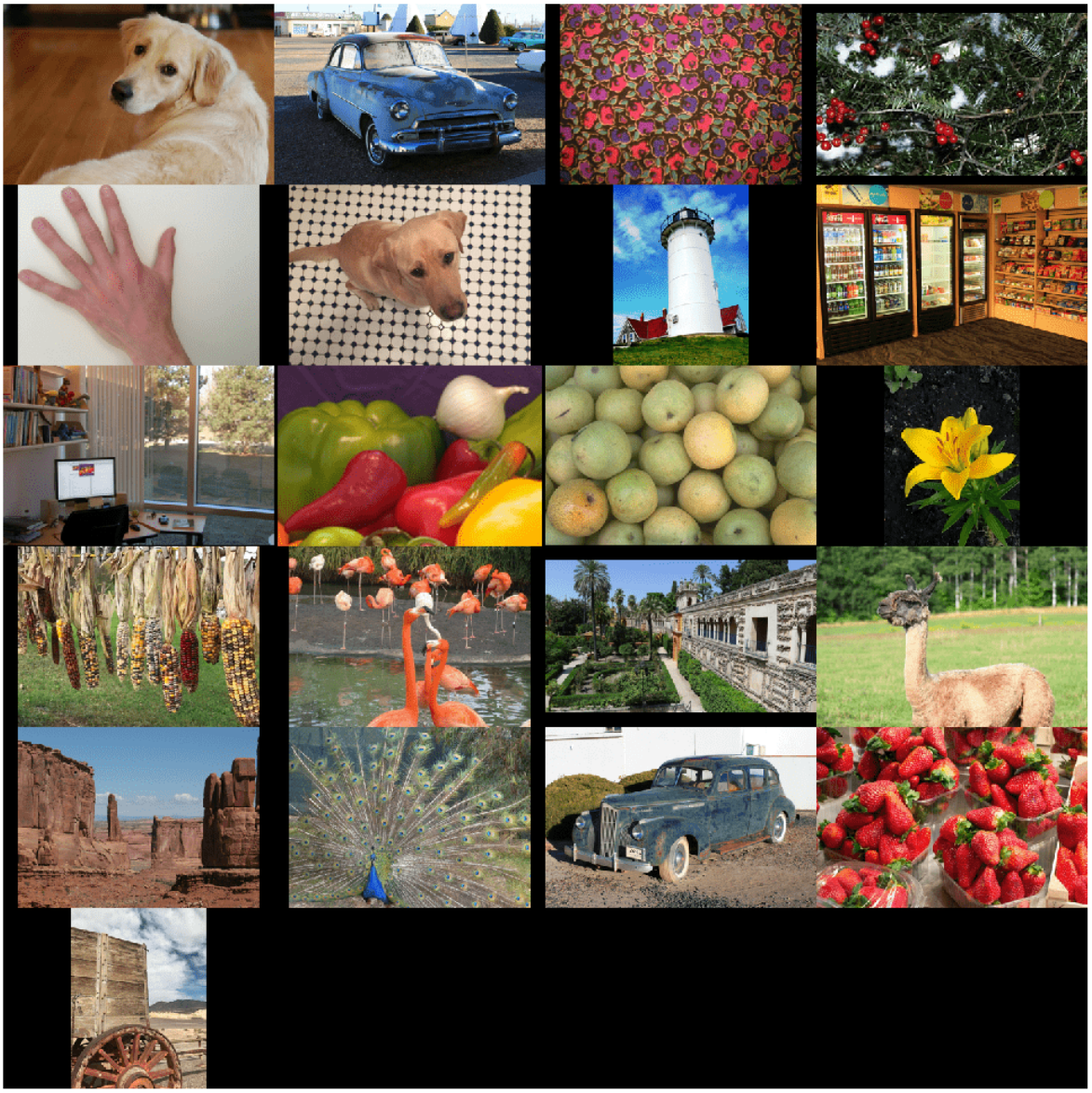
Create Sample Noisy Image

Create a sample noisy image that will be used to compare the results of operator approximation to conventional bilateral filtering. The test data set, `testImages`, contains 21 pristine images shipped in Image Processing Toolbox™. Load the images into an `imageDatastore`.

```
exts = {'.jpg','.png'};
fileNames = {'sherlock.jpg','car2.jpg','fabric.png','greens.jpg','hands1.jpg','kobi.png',...
    'lighthouse.png','micromarket.jpg','office_4.jpg','onion.png','pears.png','yellowlily.jpg',.
    'indiancorn.jpg','flamingos.jpg','sevilla.jpg','llama.jpg','parkavenue.jpg',...
    'peacock.jpg','car1.jpg','strawberries.jpg','wagon.jpg'};
filePath = [fullfile(matlabroot,'toolbox','images','imdata') filesep];
filePathNames = strcat(filePath,fileNames);
testImages = imageDatastore(filePathNames,'FileExtensions',exts);
```

Display the test images as a montage.

```
montage(testImages)
```



Select one of the images to use as the reference image for bilateral filtering. Convert the image to data type `uint8`.

```
indx = 3; % Index of image to read from the test image datastore
Ireference = readimage(testImages,indx);
Ireference = im2uint8(Ireference);
```

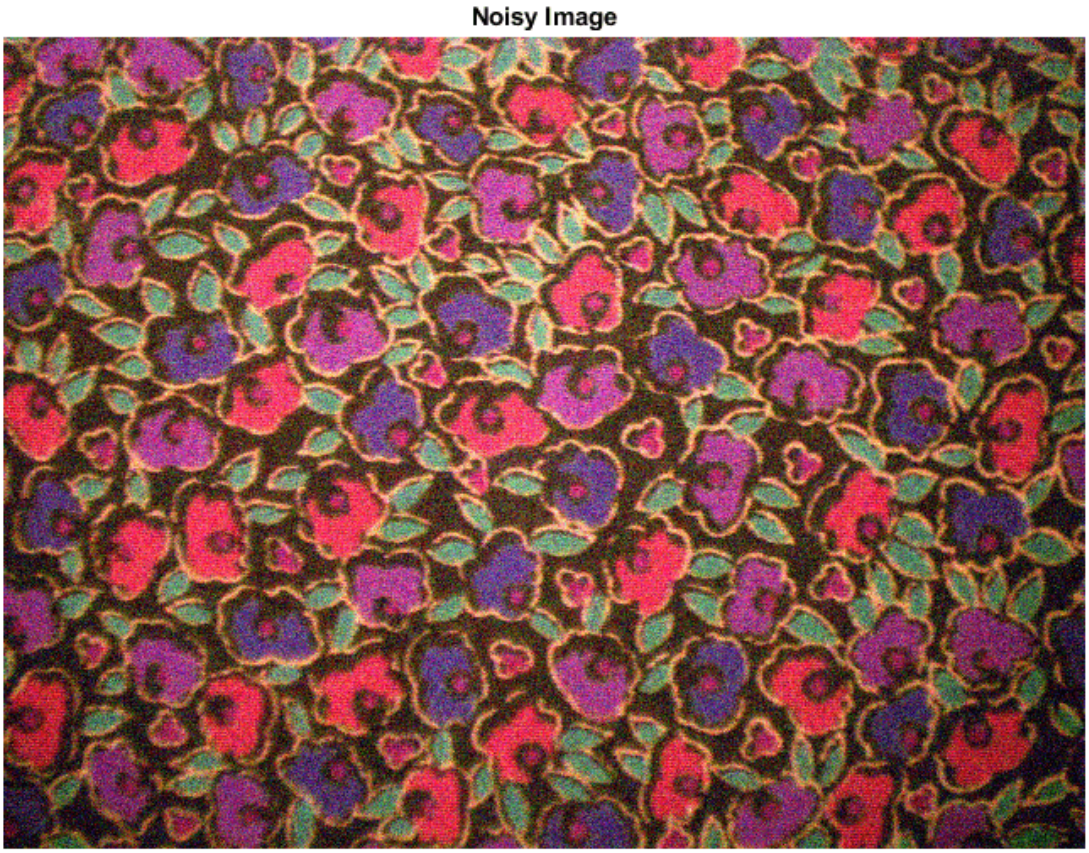
You can optionally use your own image as the reference image. Note that the size of the test image must be at least 256-by-256. If the test image is smaller than 256-by-256, then increase the image size by using the `imresize` function. The network also requires an RGB test image. If the test image is grayscale, then convert the image to RGB by using the `cat` function to concatenate three copies of the original image along the third dimension.

Display the reference image.

```
imshow(Ireference)
title('Pristine Reference Image')
```

Use the `imnoise` (Image Processing Toolbox) function to add zero-mean Gaussian white noise with a variance of 0.00001 to the reference image.

```
Inoisy = imnoise(Ireference,'gaussian',0.00001);
imshow(Inoisy)
title('Noisy Image')
```



Filter Image Using Bilateral Filtering

Conventional bilateral filtering is a standard way to reduce image noise while preserving edge sharpness. Use the `imbilatfilt` (Image Processing Toolbox) function to apply a bilateral filter to the noisy image. Specify a degree of smoothing equal to the variance of pixel values.

```
degreeOfSmoothing = var(double(Inoisy(:)));
Ibilat = imbilatfilt(Inoisy,degreeOfSmoothing);
imshow(Ibilat)
title('Denoised Image Obtained Using Bilateral Filtering')
```

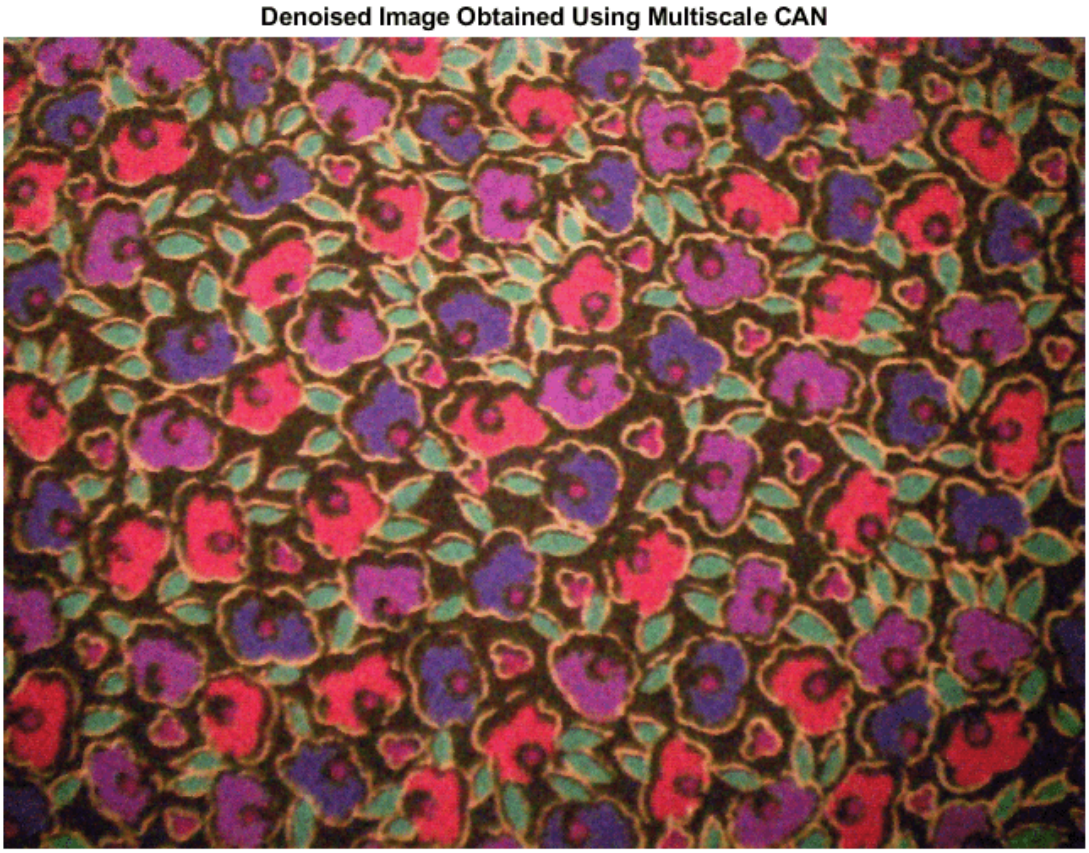
Process Image Using Trained Network

Pass the normalized input image through the trained network and observe the `activations` from the final layer (a regression layer). The output of the network is the desired denoised image.

```
Iapprox = activations(net,Inoisy,'FinalRegressionLayer');
```

Image Processing Toolbox™ requires floating point images to have pixel values in the range [0, 1]. Use the `rescale` function to scale the pixel values to this range, then convert the image to `uint8`.

```
Iapprox = rescale(Iapprox);
Iapprox = im2uint8(Iapprox);
imshow(Iapprox)
title('Denoised Image Obtained Using Multiscale CAN')
```



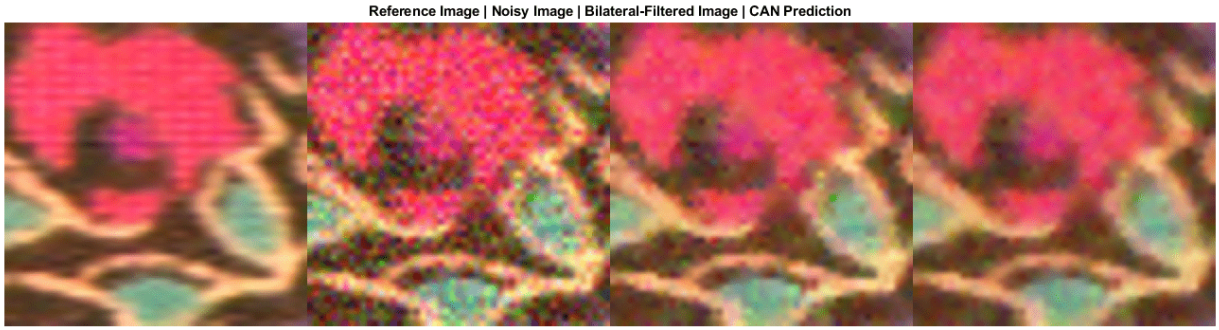
Visual and Quantitative Comparison

To get a better visual understanding of the denoised images, examine a small region inside each image. Specify a region of interest (ROI) using vector `roi` in the format [*x y width height*]. The elements define the x- and y-coordinate of the top left corner, and the width and height of the ROI.

```
roi = [300 30 50 50];
```

Crop the images to this ROI, and display the result as a montage.

```
montage({imcrop(Ireference,roi),imcrop(Inoisy,roi), ...
    imcrop(Ibilat,roi),imcrop(Iapprox,roi)}, ...
    'Size',[1 4]);
title('Reference Image | Noisy Image | Bilateral-Filtered Image | CAN Prediction');
```



The CAN removes more noise than conventional bilateral filtering. Both techniques preserve edge sharpness.

Use image quality metrics to quantitatively compare the noisy input image, the bilateral-filtered image, and the operator-approximated image. The reference image is the original reference image, `Ireference`, before adding noise.

Measure the peak signal-to-noise ratio (PSNR) of each image against the reference image. Larger PSNR values generally indicate better image quality. See `psnr` (Image Processing Toolbox) for more information about this metric.

```
noisyPSNR = psnr(Inoisy,Ireference);
bilatPSNR = psnr(Ibilat,Ireference);
approxPSNR = psnr(Iapprox,Ireference);
disp(['PSNR of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = ', ...
    num2str([noisyPSNR bilatPSNR approxPSNR])])
```

```
PSNR of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 20.2857      25.7
```

Measure the structural similarity index (SSIM) of each image. SSIM assesses the visual impact of three characteristics of an image: luminance, contrast and structure, against a reference image. The closer the SSIM value is to 1, the better the test image agrees with the reference image. See `ssim` (Image Processing Toolbox) for more information about this metric.

```
noisySSIM = ssim(Inoisy,Ireference);
bilatSSIM = ssim(Ibilat,Ireference);
approxSSIM = ssim(Iapprox,Ireference);
disp(['SSIM of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = ', ...
    num2str([noisySSIM bilatSSIM approxSSIM])])
```

```
SSIM of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 0.76251     0.915
```

Measure perceptual image quality using the Naturalness Image Quality Evaluator (NIQE). Smaller NIQE scores indicate better perceptual quality. See `niqe` (Image Processing Toolbox) for more information about this metric.

```
noisyNIQE = niqe(Inoisy);
bilatNIQE = niqe(Ibilat);
approxNIQE = niqe(Iapprox);
disp(['NIQE score of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = ',
    num2str([noisyNIQE bilatNIQE approxNIQE])])
```

```
NIQE score of: Noisy Image / Bilateral-Filtered Image / Operator Approximated Image = 12.1865
```

Compared to conventional bilateral filtering, the operator approximation produces better metric scores.

References

[1] Chen, Q. J. Xu, and V. Koltun. "Fast Image Processing with Fully-Convolutional Networks." In *Proceedings of the 2017 IEEE Conference on Computer Vision*. Venice, Italy, Oct. 2017, pp. 2516-2525.

[2] Grubinger, M., P. Clough, H. Müller, and T. Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

See Also

`randomPatchExtractionDatastore` | `trainNetwork` | `trainingOptions` | `layerGraph` | `activations` | `imbilatfilt` | `imageDatastore`

More About

- “Preprocess Images for Deep Learning” on page 19-16
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

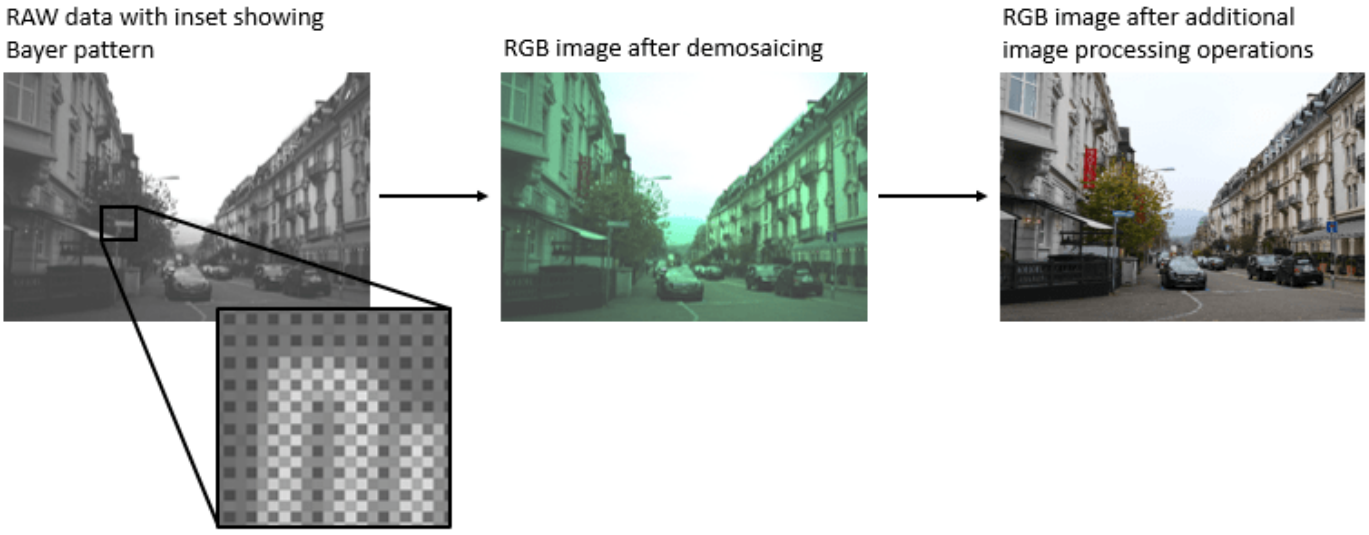
Develop RAW Camera Processing Pipeline Using Deep Learning

This example shows how to convert RAW camera data to an aesthetically pleasing color image using a U-Net.

DSLRs and many modern phone cameras offer the ability to save data collected directly from the camera sensor as a RAW file. Each pixel of RAW data corresponds directly to the amount of light captured by a corresponding camera photosensor. The data depends on fixed characteristics of the camera hardware, such as the sensitivity of each photosensor to a particular range of wavelengths of the electromagnetic spectrum. The data also depends on camera acquisition settings such as exposure time, and factors of the scene such as the light source.

Demosaicing is the only required operation to convert single-channel RAW data to a three-channel RGB image. However, without additional image processing operations, the resulting RGB image has subjectively poor visual quality.

A traditional image processing pipeline performs a combination of additional operations including denoising, linearization, white-balancing, color correction, brightness adjustment, and contrast adjustment [1 on page 9-0]. The challenge of designing a pipeline lies in refining algorithms to optimize the subjective appearance of the final RGB image regardless of variations in the scene and acquisition settings.



Deep learning techniques enable direct RAW to RGB conversion without the necessity of developing a traditional processing pipeline. For instance, one technique compensates for underexposure when converting RAW images to RGB [2 on page 9-0]. This example shows how to convert RAW images from a lower end phone camera to RGB images that approximate the quality of a higher end DSLR camera.

Download Zurich RAW to RGB Data Set

This example uses the Zurich RAW to RGB data set [3 on page 9-0]. The size of the data set is 22 GB. The data set contains 48,043 spatially registered pairs of RAW and RGB training image patches of size 448-by-448. The data set contains two separate test sets. One test set consists of 1,204 spatially registered pairs of RAW and RGB image patches of size 448-by-448. The other test set consists of unregistered full-resolution RAW and RGB images.

Create a directory to store the data set.

```
imageDir = fullfile(tempdir,'ZurichRAWToRGB');
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
```

To download the data set, request access using the Zurich RAW to RGB dataset form. Extract the data into the directory specified by the `imageDir` variable. When extracted successfully, `imageDir` contains three directories named `full_resolution`, `test`, and `train`.

Create Datastores for Training, Validation, and Testing

Create Datastore for RGB Image Patch Training Data

Create an `imageDatastore` that reads the target RGB training image patches acquired using a high-end Canon DSLR.

```
trainImageDir = fullfile(imageDir,'train');
dsTrainRGB = imageDatastore(fullfile(trainImageDir,'canon'),'ReadSize',16);
```

Preview an RGB training image patch.

```
groundTruthPatch = preview(dsTrainRGB);
imshow(groundTruthPatch)
```

Create Datastore for RAW Image Patch Training Data

Create an `imageDatastore` that reads the input RAW training image patches acquired using a Huawei phone camera. The RAW images are captured with 10-bit precision and are represented as both 8-bit and 16-bit PNG files. The 8-bit files provide a compact representation of patches with data in the range [0, 255]. No scaling has been done on any of the RAW data.

```
dsTrainRAW = imageDatastore(fullfile(trainImageDir,'huawei_raw'),'ReadSize',16);
```

Preview an input RAW training image patch. The datastore reads this patch as an 8-bit `uint8` image because the sensor counts are in the range [0, 255]. To simulate the 10-bit dynamic range of the training data, divide the image intensity values by 4. If you zoom in on the image, then you can see the RGGB Bayer pattern.

```
inputPatch = preview(dsTrainRAW);
inputPatchRAW = inputPatch/4;
imshow(inputPatchRAW)
```

To simulate the minimal traditional processing pipeline, demosaic the RGGB Bayer pattern of the RAW data using the `demosaic` (Image Processing Toolbox) function. Display the processed image and brighten the display. Compared to the target RGB image, the minimally-processed RGB image is dark and has imbalanced colors and noticeable artifacts. A trained RAW-to-RGB network performs preprocessing operations so that the output RGB image resembles the target image.

```
inputPatchRGB = demosaic(inputPatch,'rggb');
imshow(rescale(inputPatchRGB))
```

Partition Test Images into Validation and Test Sets

The test data contains RAW and RGB image patches and full-sized images. This example partitions the test image patches into a validation set and test set. The example uses the full-sized test images for qualitative testing only. See Evaluate Trained Image Processing Pipeline on Full-Sized Images on page 9-0 .

Create image datastores that read the RAW and RGB test image patches.

```
testImageDir = fullfile(imageDir,'test');
dsTestRAW = imageDatastore(fullfile(testImageDir,'huawei_raw'),'ReadSize',16);
dsTestRGB = imageDatastore(fullfile(testImageDir,'canon'),'ReadSize',16);
```

Randomly split the test data into two sets for validation and training. The validation data set contains 200 images. The test set contains the remaining images.

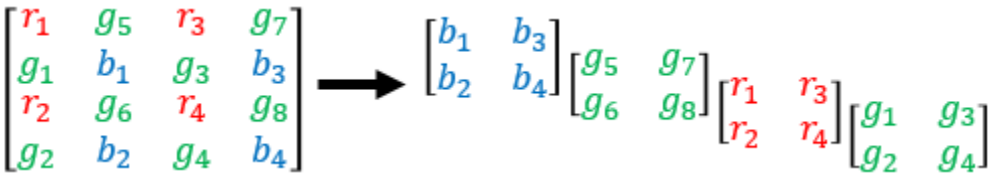
```
numTestImages = dsTestRAW.numpartitions;
numValImages = 200;
```

```
testIdx = randperm(numTestImages);
validationIdx = testIdx(1:numValImages);
testIdx = testIdx(numValImages+1:numTestImages);

dsValRAW = subset(dsTestRAW,validationIdx);
dsValRGB = subset(dsTestRGB,validationIdx);

dsTestRAW = subset(dsTestRAW,testIdx);
dsTestRGB = subset(dsTestRGB,testIdx);
```

Preprocess and Augment Data

The sensor acquires color data in a repeating Bayer pattern that includes one red, two green, and one blue photosensor. Preprocess the data into a four-channel image expected of the network using the `transform` function. The `transform` function processes the data using the operations specified in the `preprocessRAWDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `preprocessRAWDataForRAWToRGB` helper function converts an *H*-by-*W*-by-1 RAW image to an *H*/2-by-*W*/2-by-4 multichannel image consisting of one red, two green, and one blue channel.



The function also casts the data to data type `single` scaled to the range [0, 1].

```
dsTrainRAW = transform(dsTrainRAW,@preprocessRAWDataForRAWToRGB);
dsValRAW = transform(dsValRAW,@preprocessRAWDataForRAWToRGB);
dsTestRAW = transform(dsTestRAW,@preprocessRAWDataForRAWToRGB);
```

The target RGB images are stored on disk as unsigned 8-bit data. To make the computation of metrics and the network design more convenient, preprocess the target RGB training images using the `transform` function and the `preprocessRGBDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `preprocessRGBDataForRAWToRGB` helper function casts images to data type `single` scaled to the range [0, 1].

```
dsTrainRGB = transform(dsTrainRGB,@preprocessRGBDataForRAWToRGB);
dsValRGB = transform(dsValRGB,@preprocessRGBDataForRAWToRGB);
```

Combine the input RAW and target RGB data for the training, validation, and test image sets by using the `combine` function.

```
dsTrain = combine(dsTrainRAW,dsTrainRGB);
dsVal = combine(dsValRAW,dsValRGB);
dsTest = combine(dsTestRAW,dsTestRGB);
```

Randomly augment the training data using the `transform` function and the `augmentDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

The `augmentDataForRAWToRGB` helper function randomly applies 90 degree rotation and horizontal reflection to pairs of input RAW and target RGB training images.

```
dsTrainAug = transform(dsTrain,@augmentDataForRAWToRGB);
```

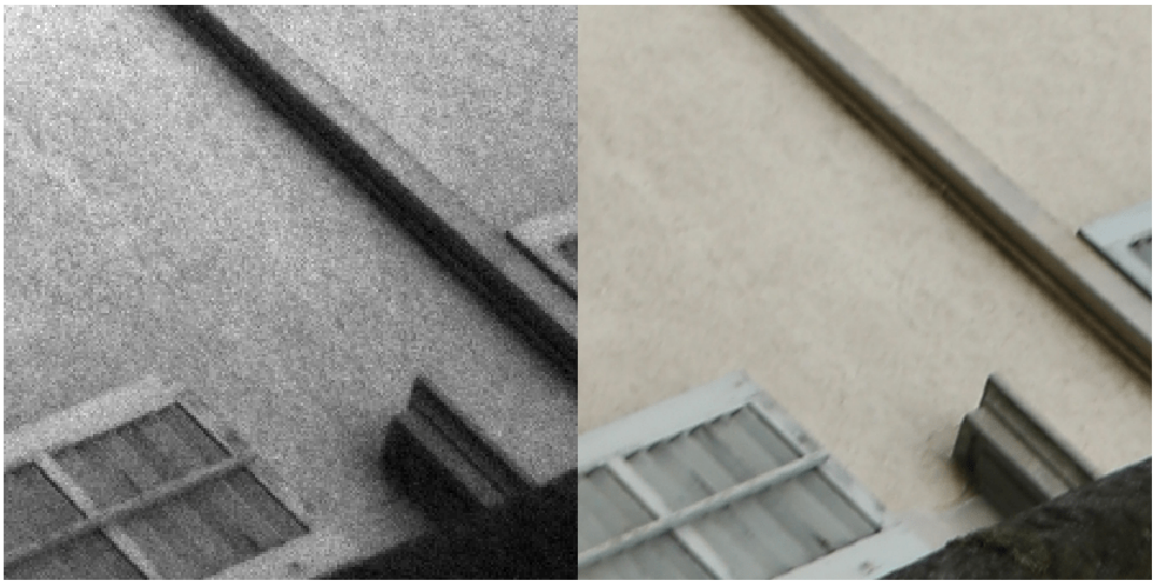
Preview the augmented training data.

```
exampleAug = preview(dsTrainAug)
```

```
exampleAug=8×2 cell array
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
    {224×224×4 single}    {448×448×3 single}
```

Display the network input and target image in a montage. The network input has four channels, so display the first channel rescaled to the range [0, 1]. The input RAW and target RGB images have identical augmentation.

```
exampleInput = exampleAug{1,1};
exampleOutput = exampleAug{1,2};
montage({rescale(exampleInput(:,:,1)),exampleOutput})
```



Batch Training and Validation Data During Training

This example uses a custom training loop. The `minibatchqueue` object is useful for managing the mini-batching of observations in custom training loops. The `minibatchqueue` object also casts data to a `dlarray` object that enables auto differentiation in deep learning applications.

```
miniBatchSize = 12;
valBatchSize = 10;
trainingQueue = minibatchqueue(dsTrainAug,'MiniBatchSize',miniBatchSize,'PartialMiniBatch','disca
validationQueue = minibatchqueue(dsVal,'MiniBatchSize',valBatchSize,'MiniBatchFormat','SSCB');
```

The `next` function of `minibatchqueue` yields the next mini-batch of data. Preview the outputs from one call to the `next` function. The outputs have data type `dlarray`. The data is already cast to `gpuArray`, on the GPU, and ready for training.

```
[inputRAW,targetRGB] = next(trainingQueue);
whos inputRAW
```

```
  Name            Size                    Bytes  Class      Attributes

  inputRAW      224x224x4x12            9633800  dlarray
```

```
whos targetRGB
```

```
  Name             Size                     Bytes  Class      Attributes

  targetRGB      448x448x3x12            28901384  dlarray
```

Set Up U-Net Network Layers

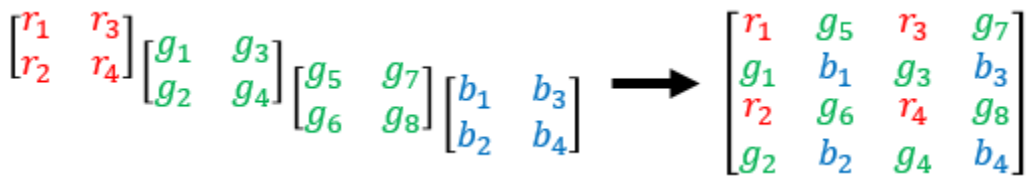
This example uses a variation of the U-Net network. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. The name U-Net comes from the fact that the network can be drawn with a symmetric shape like the letter U.

This example uses a simple U-Net architecture with two modifications. First, the network replaces the final transposed convolution operation with a custom pixel shuffle upsampling (also known as a depth-to-space) operation. Second, the network uses a custom hyperbolic tangent activation layer as the final layer in the network.

Pixel Shuffle Upsampling

Convolution followed by pixel shuffle upsampling can define subpixel convolution for super resolution applications. Subpixel convolution prevents the checkboard artifacts that can arise from transposed convolution [6 on page 9-0]. Because the model needs to map *H*/2-by-*W*/2-by-4 RAW inputs to *W*-by-*H*-by-3 RGB outputs, the final upsampling stage of the model can be thought of similarly to super resolution where the number of spatial samples grows from the input to the output.

The figure shows how pixel shuffle upsampling works for a 2-by-2-by-4 input. The first two dimensions are spatial dimensions and the third dimension is a channel dimension. In general, pixel shuffle upsampling by a factor of *S* takes an *H*-by-*W*-by-*C* input and yields an *S***H*-by-*S***W*-by-$\frac{C}{S^2}$ output.

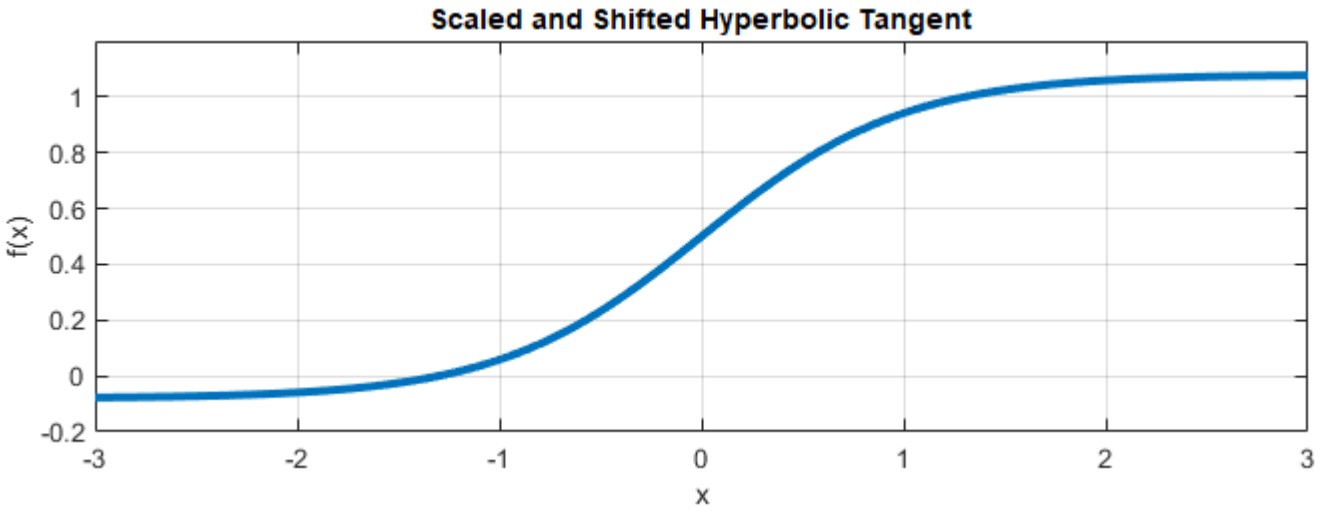


The pixel shuffle function grows the spatial dimensions of the output by mapping information from channel dimensions at a given spatial location into *S*-by-*S* spatial blocks in the output in which each channel contributes to a consistent spatial position relative to its neighbors during upsampling.

Scaled and Hyperbolic Tangent Activation

A hyperbolic tangent activation layer applies the `tanh` function on the layer inputs. This example uses a scaled and shifted version of the `tanh` function, which encourages but does not strictly enforce that the RGB network outputs are in the range [0, 1].

$$f(x) = 0.58 * \tanh(x) + 0.5$$



Calculate Training Set Statistics for Input Normalization

Use `tall` to compute per-channel mean reduction across the training data set. The input layer of the network performs mean centering of inputs during training and testing using the mean statistics.

```
dsIn = copy(dsTrainRAW);
dsIn.UnderlyingDatastore.ReadSize = 1;
t = tall(dsIn);
perChannelMean = gather(mean(t,[1 2]));
```

```
Evaluating tall expression using the Parallel Pool 'LocalProfile12':
- Pass 1 of 1: 0% complete
Evaluation 0% complete

- Pass 1 of 1: Completed in 1 min 1 sec
Evaluation completed in 1 min 2 sec
```

Create U-Net

Create layers of the initial subnetwork, specifying the per-channel mean.

```
inputSize = [256 256 4];
initialLayer = imageInputLayer(inputSize,"Normalization","zerocenter","Mean",perChannelMean, ...
    "Name","ImageInputLayer");
```

Add layers of the first encoding subnetwork. The first encoder has 32 convolutional filters.

```
numEncoderStages = 4;
numFiltersFirstEncoder = 32;
encoderNamePrefix = "Encoder-Stage-";

encoderLayers = [
    convolution2dLayer([3 3],numFiltersFirstEncoder,"Padding","same","WeightsInitializer","narrow
        "Name",strcat(encoderNamePrefix,"1-Conv-1"))
    leakyReluLayer(0.2,"Name",strcat(encoderNamePrefix,"1-ReLU-1"))
    convolution2dLayer([3 3],numFiltersFirstEncoder,"Padding","same","WeightsInitializer","narrow
        "Name",strcat(encoderNamePrefix,"1-Conv-2"))
    leakyReluLayer(0.2,"Name",strcat(encoderNamePrefix,"1-ReLU-2"))
    maxPooling2dLayer([2 2],"Stride",[2 2], ...
        "Name",strcat(encoderNamePrefix,"1-MaxPool"));
    ];
```

Add layers of additional encoding subnetworks. These subnetworks add channel-wise instance normalization after each convolutional layer using a `groupNormalizationLayer`. Each encoder subnetwork has twice the number of filters as the previous encoder subnetwork.

```
cnIdx = 1;
for stage = 2:numEncoderStages

    numFilters = numFiltersFirstEncoder*2^(stage-1);
    layerNamePrefix = strcat(encoderNamePrefix,num2str(stage));

    encoderLayers = [
        encoderLayers
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-1"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal"
            "Name",strcat(layerNamePrefix,"-Conv-2"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx+1)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))
        maxPooling2dLayer([2 2],"Stride",[2 2],"Name",strcat(layerNamePrefix,"-MaxPool"))
        ];

    cnIdx = cnIdx + 2;
end
```

Add bridge layers. The bridge subnetwork has twice the number of filters as the final encoder subnetwork and first decoder subnetwork.

```
numFilters = numFiltersFirstEncoder*2^numEncoderStages;
bridgeLayers = [
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name","Bridge-Conv-1")
    groupNormalizationLayer("channel-wise","Name","cn7")
    leakyReluLayer(0.2,"Name","Bridge-ReLU-1")
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name","Bridge-Conv-2")
    groupNormalizationLayer("channel-wise","Name","cn8")
    leakyReluLayer(0.2,"Name","Bridge-ReLU-2")];
```

Add layers of the first three decoder subnetworks.

```
numDecoderStages = 4;
cnIdx = 9;
```

```
decoderNamePrefix = "Decoder-Stage-";

decoderLayers = [];
for stage = 1:numDecoderStages-1

    numFilters = numFiltersFirstEncoder*2^(numDecoderStages-stage);
    layerNamePrefix = strcat(decoderNamePrefix,num2str(stage));

    decoderLayers = [
        decoderLayers
        transposedConv2dLayer([3 3],numFilters,"Stride",[2 2],"Cropping","same","WeightsInitiali
            "Name",strcat(layerNamePrefix,"-UpConv"))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-UpReLU"))
        depthConcatenationLayer(2,"Name",strcat(layerNamePrefix,"-DepthConcatenation"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal
            "Name",strcat(layerNamePrefix,"-Conv-1"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
        convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal
            "Name",strcat(layerNamePrefix,"-Conv-2"))
        groupNormalizationLayer("channel-wise","Name",strcat("cn",num2str(cnIdx+1)))
        leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))
        ];

    cnIdx = cnIdx + 2;
end
```

Add layers of the last decoder subnetwork. This subnetwork excludes the channel-wise instance normalization performed by the other decoder subnetworks. Each decoder subnetwork has half the number of filters as the previous subnetwork.

```
numFilters = numFiltersFirstEncoder;
layerNamePrefix = strcat(decoderNamePrefix,num2str(stage+1));

decoderLayers = [
    decoderLayers
    transposedConv2dLayer([3 3],numFilters,"Stride",[2 2],"Cropping","same","WeightsInitializer"
       "Name",strcat(layerNamePrefix,"-UpConv"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-UpReLU"))
    depthConcatenationLayer(2,"Name",strcat(layerNamePrefix,"-DepthConcatenation"))
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name",strcat(layerNamePrefix,"-Conv-1"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-1"))
    convolution2dLayer([3 3],numFilters,"Padding","same","WeightsInitializer","narrow-normal", .
        "Name",strcat(layerNamePrefix,"-Conv-2"))
    leakyReluLayer(0.2,"Name",strcat(layerNamePrefix,"-ReLU-2"))];
```

Add the final layers of the U-Net. The pixel shuffle layer moves from the *H*/2-by-*W*/2-by-12 channel size of the activations from the final convolution to *H*-by-*W*-by-3 channel activations using pixel shuffle upsampling. The final layer encourages outputs to the desired range [0, 1] using a hyperbolic tangent function.

```
finalLayers = [
    convolution2dLayer([3 3],12,"Padding","same","WeightsInitializer","narrow-normal", ...
       "Name","Decoder-Stage-4-Conv-3")
    pixelShuffleLayer("pixelShuffle",2)
    tanhScaledAndShiftedLayer("tanhActivation")];
```

```
layers = [initialLayer;encoderLayers;bridgeLayers;decoderLayers;finalLayers];
lgraph = layerGraph(layers);
```

Connect layers of the encoding and decoding subnetworks.

```
lgraph = connectLayers(lgraph,"Encoder-Stage-1-ReLU-2","Decoder-Stage-4-DepthConcatenation/in2");
lgraph = connectLayers(lgraph,"Encoder-Stage-2-ReLU-2","Decoder-Stage-3-DepthConcatenation/in2");
lgraph = connectLayers(lgraph,"Encoder-Stage-3-ReLU-2","Decoder-Stage-2-DepthConcatenation/in2");
lgraph = connectLayers(lgraph,"Encoder-Stage-4-ReLU-2","Decoder-Stage-1-DepthConcatenation/in2");
net = dlnetwork(lgraph);
```

Visualize the network architecture using the Deep Network Designer app.

```
%deepNetworkDesigner(lgraph)
```

Load the Feature Extraction Network

This function modifies a pretrained VGG-16 deep neural network to extract image features at various layers. These multilayer features are used to compute content loss.

To get a pretrained VGG-16 network, install `vgg16`. If you do not have the required support package installed, then the software provides a download link.

```
vggNet = load('vgg16');
vggNet = vggNet.net;
%vggNet = vgg16;
```

To make the VGG-16 network suitable for feature extraction, use the layers up to `'relu5_3'`.

```
vggNet = vggNet.Layers(1:31);
vggNet = dlnetwork(layerGraph(vggNet));
```

Define Model Gradients and Loss Functions

The helper function `modelGradients` calculates the gradients and overall loss for batches of training data. This function is defined in the Supporting Functions on page 9-0 section of this example.

The overall loss is a weighted sum of two losses: mean of absolute error (MAE) loss and content loss. The content loss is weighted such that the MAE loss and content loss contribute approximately equally to the overall loss:

$$lossOverall = lossMAE + weightFactor * lossContent$$

The MAE loss penalises the L^1 distance between samples of the network predictions and samples of the target image. L^1 is often a better choice than L^2 for image processing applications because it can help reduce blurring artifacts [4 on page 9-0]. This loss is implemented using the `maeLoss` helper function defined in the Supporting Functions on page 9-0 section of this example.

The content loss helps the network learn both high-level structural content and low-level edge and color information. The loss function calculates a weighted sum of the mean square error (MSE) between predictions and targets for each activation layer. This loss is implemented using the `contentLoss` helper function defined in the Supporting Functions on page 9-0 section of this example.

Calculate Content Loss Weight Factor

The `modelGradients` helper function requires the content loss weight factor as an input argument. Calculate the weight factor for a sample batch of training data such that the MAE loss is equal to the weighted content loss.

Preview a batch of training data, which consists of pairs of RAW network inputs and RGB target outputs.

```
trainingBatch = preview(dsTrainAug);
networkInput = dlarray((trainingBatch{1,1}),'SSC');
targetOutput = dlarray((trainingBatch{1,2}),'SSC');
```

Predict the response of the untrained U-Net network using the `forward` function.

```
predictedOutput = forward(net,networkInput);
```

Calculate the MAE and content losses between the predicted and target RGB images.

```
sampleMAELoss = maeLoss(predictedOutput,targetOutput);
sampleContentLoss = contentLoss(vggNet,predictedOutput,targetOutput);
```

Calculate the weight factor.

```
weightContent = sampleMAELoss/sampleContentLoss;
```

Specify Training Options

Define the training options that are used within the custom training loop to control aspects of Adam optimization. Train for 20 epochs.

```
learnRate = 5e-5;
numEpochs = 20;
```

Train Network

By default, the example downloads a pretrained version of the RAW-to-RGB network by using the helper function `downloadTrainedRAWToRGBNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradients` helper function.
- Update the network parameters using the `adamupdate` function and the gradient information.
- Update the training progress plot for every iteration and display various computed losses.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 88 hours on an NVIDIA™ Titan RTX and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
```

```
    % Create a directory to store checkpoints
    checkpointDir = fullfile(imageDir,'checkpoints',filesep);
    if ~exist(checkpointDir,'dir')
        mkdir(checkpointDir);
    end

    % Initialize training plot
    [hFig,batchLine,validationLine] = initializeTrainingPlotRAWPipeline;

    % Initialize Adam solver state
    [averageGrad,averageSqGrad] = deal([]);
    iteration = 0;

    start = tic;
    for epoch = 1:numEpochs
        reset(trainingQueue);
        shuffle(trainingQueue);
        while hasdata(trainingQueue)
            [inputRAW,targetRGB] = next(trainingQueue);

            [grad,loss] = dlfeval(@modelGradients,net,vggNet,inputRAW,targetRGB,weightContent);

            iteration = iteration + 1;

            [net,averageGrad,averageSqGrad] = adamupdate(net, ...
                grad,averageGrad,averageSqGrad,iteration,learnRate);

            updateTrainingPlotRAWPipeline(batchLine,validationLine,iteration,loss,start,epoch,..
                validationQueue,numValImages,valBatchSize,net,vggNet,weightContent);
        end
        % Save checkpoint of network state
        save(checkpointDir + "epoch" + epoch,'net');
    end
    % Save the final network state
    save(checkpointDir + "trainedRAWToRGBNet.mat",'net');

else
    trainedRAWToRGBNet_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedRAWToRGBI
    downloadTrainedRAWToRGBNet(trainedRAWToRGBNet_url,imageDir);
    load(fullfile(imageDir,'trainedRAWToRGBNet.mat'));
end
```

```
Pretrained RAW-to-RGB network already exists.
```

Calculate Image Quality Metrics

Reference-based quality metrics such as MSSIM or PSNR enable a quantitative measure of image quality. You can calculate the MSSIM and PSNR of the patched test images because they are spatially registered and the same size.

Iterate through the test set of patched images using a `minibatchqueue` object.

```
patchTestSet = combine(dsTestRAW,dsTestRGB);
testPatchQueue = minibatchqueue(patchTestSet,'MiniBatchSize',16,'MiniBatchFormat','SSCB');
```

Iterate through the test set and calculate the MSSIM and PSNR for each test image using the `multissim` (Image Processing Toolbox) and `psnr` (Image Processing Toolbox) functions. Compute

MSSIM for color images by using a mean of the metric for each color channel as an approximation since the metric isn't well defined for multi-channel inputs.

```
totalMSSIM = 0;
totalPSNR = 0;
while hasdata(testPatchQueue)
    [inputRAW,targetRGB] = next(testPatchQueue);
    outputRGB = forward(net,inputRAW);
    targetRGB = targetRGB ./ 255;
    mssimOut = sum(mean(multissim(outputRGB,targetRGB),3),4);
    psnrOut = sum(psnr(outputRGB,targetRGB),4);
    totalMSSIM = totalMSSIM + mssimOut;
    totalPSNR = totalPSNR + psnrOut;
end
```

Calculate the mean MSSIM and mean PSNR over the test set. This result is consistent with the similar U-Net approach from [3 on page 9-0] for mean MSSIM and competitive with the PyNet approach in [3 on page 9-0] in mean PSNR. The differences in loss functions and use of pixel shuffle upsampling compared to [3 on page 9-0] likely account for these differences.

```
numObservations = dsTestRGB.numpartitions;
meanMSSIM = totalMSSIM / numObservations
```

```
meanMSSIM =
  1(S) × 1(S) × 1(C) × 1(B) single dlarray

    0.8425
```

```
meanPSNR = totalPSNR / numObservations
```

```
meanPSNR =
  1(S) × 1(S) × 1(C) × 1(B) single dlarray

   21.1213
```

Evaluate Trained Image Processing Pipeline on Full-Sized Images

Because of sensor differences between the phone camera and DSLR used to acquire the full-resolution test images, the scenes are not registered and are not the same size. Reference-based comparison of the full-resolution images from the network and the DSLR ISP is difficult. However, a qualitative comparison of the images is useful because a goal of image processing is to create an aesthetically pleasing image.

Create an image datastore that contains full-sized RAW images acquired by a phone camera.

```
testImageDir = fullfile(imageDir,'test');
testImageDirRAW = "huawei_full_resolution";
dsTestFullRAW = imageDatastore(fullfile(testImageDir,testImageDirRAW));
```

Get the names of the image files in the full-sized RAW test set.

```
targetFilesToInclude = extractAfter(string(dsTestFullRAW.Files),fullfile(testImageDirRAW,filesep)
targetFilesToInclude = extractBefore(targetFilesToInclude,".png");
```

Preprocess the RAW data by converting the data to the form expected by the network using the `transform` function. The `transform` function processes the data using the operations specified in

the `preprocessRAWDataForRAWToRGB` helper function. The helper function is attached to the example as a supporting file.

```
dsTestFullRAW = transform(dsTestFullRAW,@preprocessRAWDataForRAWToRGB);
```

Create an image datastore that contains full-sized RGB test images captured from the high-end DSLR. The Zurich RAW-to-RGB data set contains more full-sized RGB images than RAW images, so include only the RGB images with a corresponding RAW image.

```
dsTestFullRGB = imageDatastore(fullfile(imageDir,'full_resolution','canon'));
dsTestFullRGB.Files = dsTestFullRGB.Files(contains(dsTestFullRGB.Files,targetFilesToInclude));
```

Read in the target RGB images. Get a sense of the overall output by looking at a montage view.

```
targetRGB = readall(dsTestFullRGB);
montage(targetRGB,"Size",[5 2],"Interpolation","bilinear")
```

Iterate through the test set of full-sized images using a `minibatchqueue` object. If you have a GPU device with sufficient memory to process full-resolution images, then you can run prediction on a GPU by specifying the output environment as `"gpu"`.

```
testQueue = minibatchqueue(dsTestFullRAW,"MiniBatchSize",1, ...
    "MiniBatchFormat","SSCB","OutputEnvironment","cpu");
```

For each full-sized RAW test image, predict the output RGB image by calling `forward` on the network.

```
outputSize = 2*size(preview(dsTestFullRAW),[1 2]);
outputImages = zeros([outputSize,3,dsTestFullRAW.numpartitions],'uint8');

idx = 1;
while hasdata(testQueue)
    inputRAW = next(testQueue);
    rgbOut = forward(net,inputRAW);
    rgbOut = gather(extractdata(rgbOut));
    outputImages(:,:,:,idx) = im2uint8(rgbOut);
    idx = idx+1;
end
```

Get a sense of the overall output by looking at a montage view. The network produces images that are aesthetically pleasing, with similar characteristics.

```
montage(outputImages,"Size",[5 2],"Interpolation","bilinear")
```

Compare one target RGB image with the corresponding image predicted by the network. The network produces colors which are more saturated than the target DSLR images. Although the colors from the simple U-Net architecture are not the same as the DSLR targets, the images are still qualitatively pleasing in many cases.

```
imgIdx = 1;
imTarget = targetRGB{imgIdx};
imPredicted = outputImages(:,:,:,imgIdx);
figure
montage({imTarget,imPredicted},"Interpolation","bilinear")
```

To improve the performance of the RAW-to-RGB network, a network architecture would learn detailed localized spatial features using multiple scales from global features that describe color and contrast [3 on page 9-0].

Supporting Functions

Model Gradients Function

The `modelGradients` helper function calculates the gradients and overall loss. The gradient information is returned as a table which includes the layer, parameter name and value for each learnable parameter in the model.

```
function [gradients,loss] = modelGradients(dlnet,vggNet,Xpatch,Target,weightContent)
    Y = forward(dlnet,Xpatch);
    lossMAE = maeLoss(Y,Target);
    lossContent = contentLoss(vggNet,Y,Target);
    loss = lossMAE + weightContent.*lossContent;
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

Mean Absolute Error Loss Function

The helper function `maeLoss` computes the mean absolute error between network predictions, `Y`, and target images, `T`.

```
function loss = maeLoss(Y,T)
    loss = mean(abs(Y-T),'all');
end
```

Content Loss Function

The helper function `contentLoss` calculates a weighted sum of the MSE between network predictions, `Y`, and target images, `T`, for each activation layer. The `contentLoss` helper function

calculates the MSE for each activation layer using the `mseLoss` helper function. Weights are selected such that the loss from each activation layers contributes roughly equally to the overall content loss.

```
function loss = contentLoss(net,Y,T)

    layers = ["relu1_1","relu1_2","relu2_1","relu2_2","relu3_1","relu3_2","relu3_3","relu4_1"];
    [T1,T2,T3,T4,T5,T6,T7,T8] = forward(net,T,'Outputs',layers);
    [X1,X2,X3,X4,X5,X6,X7,X8] = forward(net,Y,'Outputs',layers);

    l1 = mseLoss(X1,T1);
    l2 = mseLoss(X2,T2);
    l3 = mseLoss(X3,T3);
    l4 = mseLoss(X4,T4);
    l5 = mseLoss(X5,T5);
    l6 = mseLoss(X6,T6);
    l7 = mseLoss(X7,T7);
    l8 = mseLoss(X8,T8);

    layerLosses = [l1 l2 l3 l4 l5 l6 l7 l8];
    weights = [1 0.0449 0.0107 0.0023 6.9445e-04 2.0787e-04 2.0118e-04 6.4759e-04];
    loss = sum(layerLosses.*weights);
end
```

Mean Square Error Loss Function

The helper function `mseLoss` computes the MSE between network predictions, `Y`, and target images, `T`.

```
function loss = mseLoss(Y,T)
    loss = mean((Y-T).^2,'all');
end
```

References

1) Sumner, Rob. "Processing RAW Images in MATLAB". May 19, 2014. https://rcsumner.net/raw_guide/RAWguide.pdf.

2) Chen, Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. “Learning to See in the Dark.” *ArXiv:1805.01934 [Cs]*, May 4, 2018. http://arxiv.org/abs/1805.01934.

3) Ignatov, Andrey, Luc Van Gool, and Radu Timofte. “Replacing Mobile Camera ISP with a Single Deep Learning Model.” *ArXiv:2002.05509 [Cs, Eess]*, February 13, 2020. http://arxiv.org/abs/2002.05509. Project Website.

4) Zhao, Hang, Orazio Gallo, Iuri Frosio, and Jan Kautz. “Loss Functions for Neural Networks for Image Processing.” *ArXiv:1511.08861 [Cs]*, April 20, 2018. http://arxiv.org/abs/1511.08861.

5) Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. “Perceptual Losses for Real-Time Style Transfer and Super-Resolution.” *ArXiv:1603.08155 [Cs]*, March 26, 2016. http://arxiv.org/abs/1603.08155.

6) Shi, Wenzhe, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. “Real-Time Single Image and Video Super-Resolution Using an Efficient

Sub-Pixel Convolutional Neural Network.” *ArXiv:1609.05158 [Cs, Stat]*, September 23, 2016. http://arxiv.org/abs/1609.05158.

See Also

`trainingOptions` | `trainNetwork` | `imageDatastore` | `transform` | `combine`

Related Examples

- “Recover Images from Extreme Low-Light Conditions Using Deep Learning” on page 9-73

More About

- “Preprocess Images for Deep Learning” on page 19-16
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

Recover Images from Extreme Low-Light Conditions Using Deep Learning

This example shows how to recover brightened RGB images from RAW camera data collected in extreme low-light conditions using a U-Net.

Low-light image recovery in cameras is a challenging problem. A typical solution is to increase the exposure time, which allows more light in the scene to hit the sensor and increases the brightness of the image. However, longer exposure times can result in motion blur artifacts when objects in the scene move or when the camera is perturbed during acquisition.

Deep learning offers solutions that recover reasonable images for RAW data collected from DSLRs and many modern phone cameras despite low light conditions and short exposure times. These solutions take advantage of the full information present in RAW data to outperform brightening techniques performed in postprocessed RGB data [1].

Low Light Image (Left) and Recovered Image (Right)

This example shows how to train a network to implement a low-light camera pipeline using data from a particular camera sensor. This example shows how to recover well exposed RGB images from very low light, underexposed RAW data from the same type of camera sensor.

Download See-in-the-Dark Data Set

This example uses the Sony camera data from the See-in-the-Dark (SID) data set [1]. The SID data set provides registered pairs of RAW images of the same scene. In each pair, one image has a short exposure time and is underexposed, and the other image has a longer exposure time and is well exposed. The size of the Sony camera data from the SID data set is 25 GB.

Set `dataDir` as the desired location of the data set.

```
dataDir = fullfile(tempdir,"SID");
if ~exist(dataDir,"dir")
    mkdir(dataDir);
end
```

To download the data set, go to this link: https://storage.googleapis.com/isl-datasets/SID/Sony.zip. Extract the data into the directory specified by the `dataDir` variable. When extraction is successful,

`dataDir` contains the directory `Sony` with two subdirectories: `long` and `short`. The files in the `long` subdirectory have a long exposure and are well exposed. The files in the `short` subdirectory have a short exposure and are quite underexposed and dark.

The data set also provides text files that describe how to partition the files into training, validation, and test data sets. Move the files `Sony_train_list.txt`, `Sony_val_list.txt`, and `Sony_test_list.txt` to the directory specified by the `dataDir` variable.

Create Datastores for Training, Validation, and Testing

Import the list of files to include in the training, validation, and test data sets using the `importSonyFileInfo` helper function. This function is attached to the example as a supporting file.

```
trainInfo = importSonyFileInfo(fullfile(dataDir,"Sony_train_list.txt"));
valInfo = importSonyFileInfo(fullfile(dataDir,"Sony_val_list.txt"));
testInfo = importSonyFileInfo(fullfile(dataDir,"Sony_test_list.txt"));
```

Combine and Preprocess RAW and RGB Data Using Datastores

Create combined datastores that read and preprocess pairs of underexposed and well exposed RAW images using the `createCombinedDatastoreForLowLightRecovery` helper function. This function is attached to the example as a supporting file.

The `createCombinedDatastoreForLowLightRecovery` helper function performs these operations:

- Create an `imageDatastore` that reads the short exposure RAW images using a custom read function. The read function reads a RAW image using the `rawread` (Image Processing Toolbox) function, then separates the RAW Bayer pattern into separate channels for each of the four sensors using the `raw2planar` (Image Processing Toolbox) function. Normalize the data to the range [0, 1] by transforming the `imageDatastore` object.
- Create an `imageDatastore` object that reads long-exposure RAW images and converts the data to an RGB image in one step using the `raw2rgb` (Image Processing Toolbox) function. Normalize the data to the range [0, 1] by transforming the `imageDatastore` object.
- Combine the `imageDatastore` objects using the `combine` function.
- Apply a simple multiplicative gain to the pairs of images. The gain corrects for the exposure time difference between the shorter exposure time of the dark inputs and the longer exposure time of the output images. This gain is defined by taking the ratio of the long and short exposure times provided in the image file names.
- Associate the images with metadata such as exposure time, ISO, and aperture.

```
dsTrainFull = createCombinedDatastoreForLowLightRecovery(dataDir,trainInfo);
dsValFull = createCombinedDatastoreForLowLightRecovery(dataDir,valInfo);
dsTestFull = createCombinedDatastoreForLowLightRecovery(dataDir,testInfo);
```

Use a subset of the validation images to make computation of validation metrics quicker. Do not apply additional augmentation.

```
numVal = 30;
dsValFull = shuffle(dsValFull);
dsVal = subset(dsValFull,1:numVal);
```

Preprocess Training and Validation Data

Preprocess the training data set using the `transform` function and the `extractRandomPatch` helper function. The helper function is attached to the example as a supporting file. The

`extractRandomPatch` helper function crops multiple random patches of size 512-by-512-by-4 pixels from a planar RAW image and corresponding patches of size 1024-by-1024-by-3 pixels from an RGB image. The scene content in the patches matches. Extract 12 patches per training image.

```
inputSize = [512,512,4];
patchesPerImage = 12;
dsTrain = transform(dsTrainFull,@(data) extractRandomPatch(data,inputSize,patchesPerImage));
```

Preview an original full-sized image and a random training patch.

```
previewFull = preview(dsTrainFull);
previewPatch = preview(dsTrain);
montage({previewFull{1,2},previewPatch{1,2}},BackgroundColor="w");
```

Preprocess the validation data set using the `transform` function and the `extractCenterPatch` helper function. The helper function is attached to the example as a supporting file. The `extractCenterPatch` helper function crops a single patch of size 512-by-512-by-4 pixels from the center of a planar RAW image and corresponding patches of size 1024-by-1024-by-3 pixels from an RGB image. The scene content in the patches matches.

```
dsVal = transform(dsVal,@(data) extractCenterPatch(data,inputSize));
```

The testing data set does not require preprocessing. Test images are fed at full size into the network.

Augment Training Data

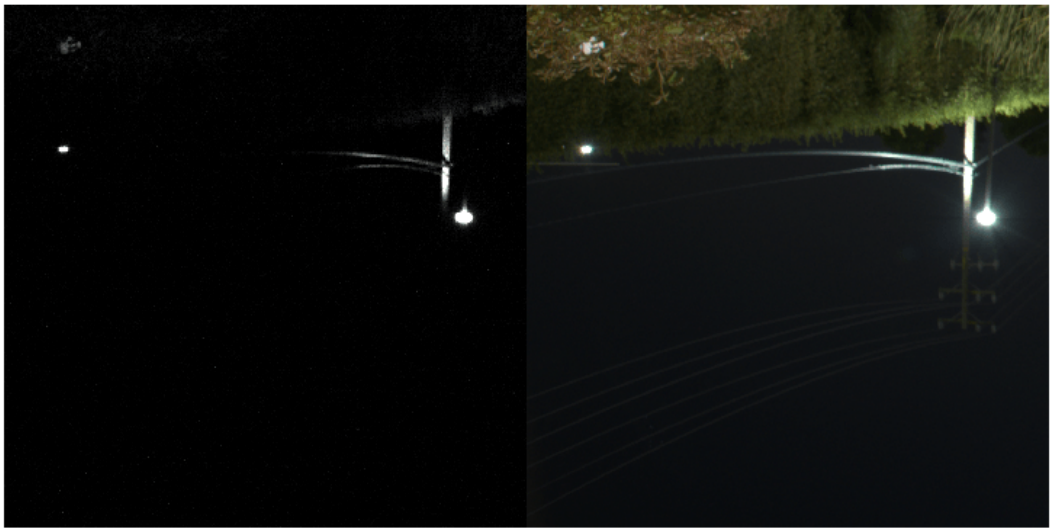
Augment the training data by adding random horizontal and vertical reflection and randomized 90-degree rotations.

```
dsTrain = transform(dsTrain,@(data) augmentPatchesForLowLightRecovery(data));
```

Verify that the preprocessing and augmentation operations work as expected by previewing one channel from the planar RAW image patch and the corresponding RGB decoded patch. The planar RAW data and the target RGB data depict patches of the same scene, randomly extracted from the original source image. Significant noise is visible in the RAW patch because of the short acquisition time of the RAW data, causing a low signal-to-noise ratio.

```
imagePairs = read(dsTrain);
rawImage = imagePairs{1,1};
```

```
rgbPatch = imagePairs{1,2};
montage({rawImage(:,:,1),rgbPatch});
```



Define Network

Use a network architecture similar to U-Net. The example creates the encoder and decoder subnetworks using the `blockedNetwork` (Image Processing Toolbox) function. This function creates the encoder and decoder subnetworks programmatically using the `buildEncoderBlock` and `buildDecoderBlock` helper functions, respectively. The helper functions are defined at the end of this example. The example uses instance normalization between convolution and activation layers in all network blocks except the first and last, and uses a leaky ReLU layer as the activation layer.

Create an encoder subnetwork that consists of four encoder modules. The first encoder module has 32 channels, or feature maps. Each subsequent module doubles the number of feature maps from the previous encoder module.

```
numModules = 4;
numChannelsEncoder = 2.^(5:8);
encoder = blockedNetwork(@(block) buildEncoderBlock(block,numChannelsEncoder), ...
    numModules,NamePrefix="encoder");
```

Create a decoder subnetwork that consists of four decoder modules. The first decoder module has 256 channels, or feature maps. Each subsequent decoder module halves the number of feature maps from the previous decoder module.

```
numChannelsDecoder = fliplr(numChannelsEncoder);
decoder = blockedNetwork(@(block) buildDecoderBlock(block,numChannelsDecoder), ...
    numModules,NamePrefix="decoder");
```

Specify the bridge layers that connect the encoder and decoder subnetworks.

```
bridgeLayers = [
    convolution2dLayer(3,512,Padding="same",PaddingValue="replicate")
    groupNormalizationLayer("channel-wise")
    leakyReluLayer(0.2)
    convolution2dLayer(3,512,Padding="same",PaddingValue="replicate")
    groupNormalizationLayer("channel-wise")
    leakyReluLayer(0.2)];
```

Specify the final layers of the network.

```
finalLayers = [
    convolution2dLayer(1,12)
    depthToSpace2dLayer(2)];
```

Combine the encoder subnetwork, bridge layers, decoder subnetwork, and final layers using the `encoderDecoderNetwork` (Image Processing Toolbox) function.

```
net = encoderDecoderNetwork(inputSize,encoder,decoder, ...
    LatentNetwork=bridgeLayers, ...
    SkipConnections="concatenate", ...
    FinalNetwork=finalLayers);
net = layerGraph(net);
```

Use mean centering normalization on the input as part of training.

```
net = replaceLayer(net,"encoderImageInputLayer",imageInputLayer(inputSize,Normalization="zerocent
```

Define the overall loss using the custom layer `ssimLossLayerGray`. This layer definition is attached to this example as a supporting file. The `ssimLossLayerGray` layer uses a loss of the form

$$lossOverall = \alpha \times lossSSIM + (1 - \alpha) \times lossL_1$$

The layer calculates a multiscale structural similarity (SSIM) loss for the grayscale representations of the predicted and target RGB images using the `multissim` (Image Processing Toolbox) function. The layer specifies the weighting factor α as 7/8 and uses five scales.

```
finalLayerName = net.Layers(end).Name;
lossLayer = ssimLossLayerGray;
net = addLayers(net,lossLayer);
net = connectLayers(net,finalLayerName,lossLayer.Name);
```

Specify Training Options

For training, use the Adam solver with an initial learning rate of 1e-3. Train for 30 epochs.

```
miniBatchSize = 12;
maxEpochs = 30;
options = trainingOptions("adam", ...
    Plots="training-progress", ...
    MiniBatchSize=miniBatchSize, ...
    InitialLearnRate=1e-3, ...
    MaxEpochs=maxEpochs, ...
    ValidationFrequency=400);
```

Train Network or Download Pretrained Network

By default, the example loads a pretrained version of the low-light recovery network. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
doTraining = false;

if doTraining
    checkpointsDir = fullfile(dataDir,"checkpoints");
    if ~exist(checkpointsDir,"dir")
        mkdir(checkpointsDir);
    end
    options.CheckpointPath=checkpointsDir;

    netTrained = trainNetwork(dsTrain,net,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(dataDir+"trainedLowLightCameraPipelineNet-"+modelDateTime+".mat",'netTrained');

else
    trainedNet_url = "https://ssd.mathworks.com/supportfiles/vision/data/trainedLowLightCameraPi
    trainedNet_filename = "trainedLowLightCameraPipelineNet.mat";
    downloadTrainedLowLightRecoveryNet(trainedNet_url,dataDir);
    load(fullfile(dataDir,trainedNet_filename));
end
```

Examine Results from Trained Network

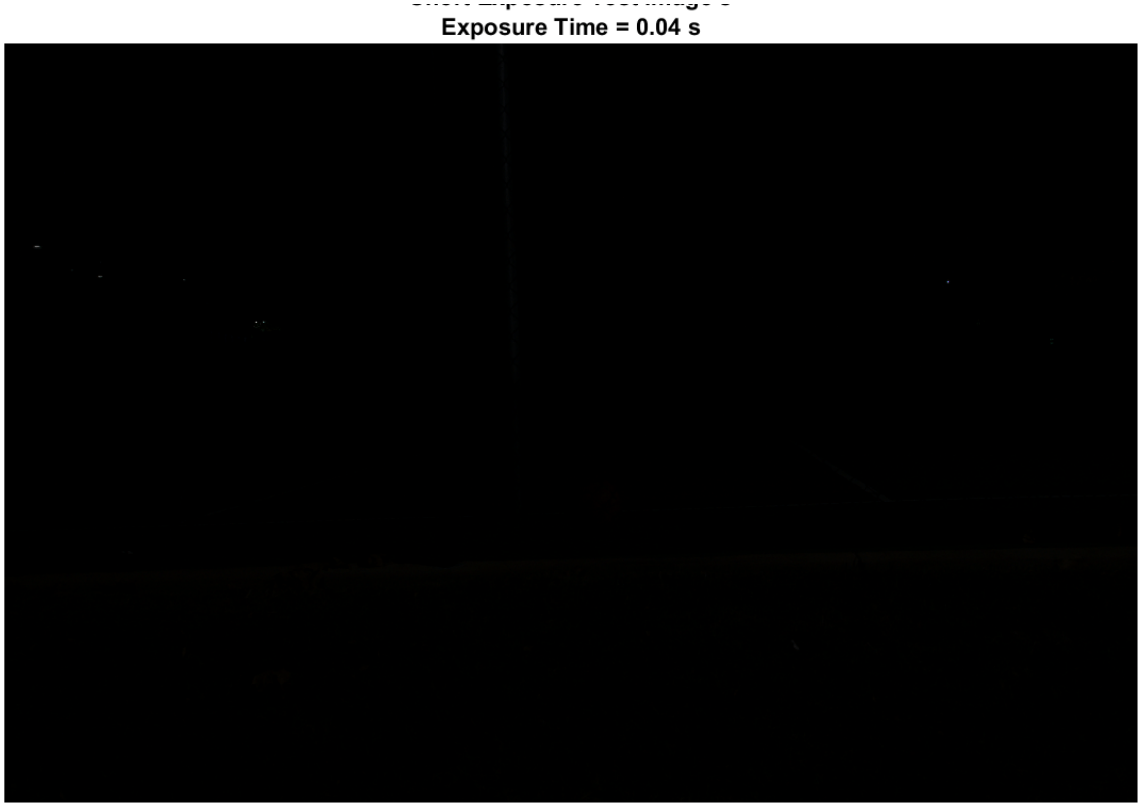
Visually examine the results of the trained low-light camera pipeline network.

Read a pair of images and accompanying metadata from the test set. Get the file names of the short and long exposure images from the metadata.

```
[testPair,info] = read(dsTestFull);
testShortFilename = info.ShortExposureFilename;
testLongFilename = info.LongExposureFilename;
```

Convert the original underexposed RAW image to an RGB image in one step using the `raw2rgb` (Image Processing Toolbox) function. Display the result, scaling the display range to the range of pixel values. The image looks almost completely black, with only a few bright pixels.

```
testShortImage = raw2rgb(testShortFilename);
testShortTime = info.ShortExposureTime;
imshow(testShortImage,[])
title(["Short Exposure Test Image";"Exposure Time = "+num2str(testShortTime)]+" s")
```



Convert the original well exposed RAW image to an RGB image in one step using the `raw2rgb` (Image Processing Toolbox) function. Display the result.

```
testLongImage = raw2rgb(testLongFilename);
testLongTime = info.LongExposureTime;
imshow(testLongImage)
title(["Long Exposure Target Image";"Exposure Time = "+num2str(testLongTime)]+" s")
```

Display the network prediction. The trained network recovers an impressive image under challenging acquisition conditions with very little noise or other visual artifacts. The colors of the network prediction are less saturated and vibrant than in the ground truth long-exposure image of the scene.

```
outputFromNetwork = im2uint8(activations(netTrained,testPair{1},'FinalNetworkLayer2'));
imshow(outputFromNetwork)
title("Low-Light Recovery Network Prediction")
```

Supporting Functions

The `extractRandomPatch` helper function crops multiple random patches from a planar RAW image and corresponding patches from an RGB image. The RAW data patch has size *m*-by-*n*-by-4 and the RGB image patch has size 2*m*-by-2*n*-by-3, where [*m n*] is the value of the `targetRAWSize` input argument. Both patches have the same scene content.

```
function dataOut = extractRandomPatch(data,targetRAWSize,patchesPerImage)
    dataOut = cell(patchesPerImage,2);
    raw = data{1};
    rgb = data{2};
    for idx = 1:patchesPerImage
        windowRAW = randomCropWindow3d(size(raw),targetRAWSize);
        windowRGB = images.spatialref.Rectangle(2*windowRAW.XLimits+[-1,0],2*windowRAW.YLimits+[
        dataOut(idx,:) = {imcrop3(raw,windowRAW),imcrop(rgb,windowRGB)};
    end
end
```

The `extractCenterPatch` helper function crops a single patch from the center of a planar RAW image and the corresponding patch from an RGB image. The RAW data patch has size *m*-by-*n*-by-4 and the RGB image patch has size 2*m*-by-2*n*-by-3, where [*m n*] is the value of the `targetRAWSize` input argument. Both patches have the same scene content.

```
function dataOut = extractCenterPatch(data,targetRAWSize)
    raw = data{1};
```

```
    rgb = data{2};
    windowRAW = centerCropWindow3d(size(raw),targetRAWSize);
    windowRGB = images.spatialref.Rectangle(2*windowRAW.XLimits+[-1,0],2*windowRAW.YLimits+[-1,0
    dataOut = {imcrop3(raw,windowRAW),imcrop(rgb,windowRGB)};
end
```

The `buildEncoderBlock` helper function defines the layers of a single encoder module within the encoder subnetwork.

```
function block = buildEncoderBlock(blockIdx,numChannelsEncoder)

    if blockIdx < 2
        instanceNorm = [];
    else
        instanceNorm = instanceNormalizationLayer;
    end

    filterSize = 3;
    numFilters = numChannelsEncoder(blockIdx);
    block = [
        convolution2dLayer(filterSize,numFilters,Padding="same",PaddingValue="replicate",Weights
        instanceNorm
        leakyReluLayer(0.2)
        convolution2dLayer(filterSize,numFilters,Padding="same",PaddingValue="replicate",Weights
        instanceNorm
        leakyReluLayer(0.2)
        maxPooling2dLayer(2,Stride=2,Padding="same")];
end
```

The `buildDecoderBlock` helper function defines the layers of a single encoder module within the decoder subnetwork.

```
function block = buildDecoderBlock(blockIdx,numChannelsDecoder)

    if blockIdx < 4
        instanceNorm = instanceNormalizationLayer;
    else
        instanceNorm = [];
    end

    filterSize = 3;
    numFilters = numChannelsDecoder(blockIdx);
    block = [
        transposedConv2dLayer(filterSize,numFilters,Stride=2,WeightsInitializer="he",Cropping="sa
        convolution2dLayer(filterSize,numFilters,Padding="same",PaddingValue="replicate",Weights
        instanceNorm
        leakyReluLayer(0.2)
        convolution2dLayer(filterSize,numFilters,Padding="same",PaddingValue="replicate",Weights
        instanceNorm
        leakyReluLayer(0.2)];
end
```

References

[1] Chen, Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. "Learning to See in the Dark." Preprint, submitted May 4, 2018. https://arxiv.org/abs/1805.01934.

See Also

`imageDatastore` | `trainingOptions` | `trainNetwork` | `transform` | `combine`

Related Examples

- “Develop RAW Camera Processing Pipeline Using Deep Learning” on page 9-51

More About

- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

Classify Tumors in Multiresolution Blocked Images

This example shows how to classify multiresolution whole slide images (WSIs) that might not fit in memory using an Inception-v3 deep neural network.

Deep learning methods for tumor classification rely on digital pathology, in which whole tissue slides are imaged and digitized. The resulting WSIs have high resolution, on the order of 200,000-by-100,000 pixels. WSIs are frequently stored in a multiresolution format to facilitate efficient display, navigation, and processing of images.

The example outlines an architecture to use block based processing to train large WSIs. The example trains an Inception-v3 based network using transfer learning techniques to classify individual blocks as `normal` or `tumor`.

If you do not want to download the training data and train the network, then continue to the Train or Download Network on page 9-0 section of this example.

Prepare Training Data

Prepare the training and validation data by following the instructions in “Preprocess Multiresolution Images for Training Classification Network” (Image Processing Toolbox). The preprocessing example saves the preprocessed training and validation datastores in the a file called `trainingAndValidationDatastores.mat`.

Set the value of the `dataDir` variable as the location where the `trainingAndValidationDatastores.mat` file is located. Load the training and validation datastores into variables called `dsTrainLabeled` and `dsValLabeled`.

```
dataDir = fullfile(tempdir,"Camelyon16");
load(fullfile(dataDir,"trainingAndValidationDatastores.mat"))
```

Set Up Inception-v3 Network Layers For Transfer Learning

This example uses an Inception-v3 network [2], a convolutional neural network that is trained on more than a million images from the ImageNet database [3]. The network is 48 layers deep and can classify images into 1,000 object categories, such as keyboard, mouse, pencil, and many animals.

The `inceptionv3` function returns a pretrained Inception-v3 network. Inception-v3 requires the Deep Learning Toolbox™ Model for Inception-v3 Network support package. If this support package is not installed, then the function provides a download link.

```
net = inceptionv3;
lgraph = layerGraph(net);
```

The convolutional layers of the network extract image features. The last learnable layer and the final classification layer classify an input image using the image features. These two layers contain information on how to combine the features into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set. For more information, see “Train Deep Learning Network to Classify New Images” on page 3-6.

Find the names of the two layers to replace using the helper function `findLayersToReplace`. This function is attached to the example as a supporting file. In Inception-v3, these two layers are named `'predictions'` and `'ClassificationLayer_predictions'`.

```
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
```

The goal of this example is to perform binary segmentation between two classes, `tumor` and `normal`. Create a new fully connected layer for two classes. Replace the final fully connected layer with the new layer.

```
numClasses = 2;
newLearnableLayer = fullyConnectedLayer(numClasses,Name="predictions");
lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
```

Create a new classification layer for two classes. Replace the final classification layer with the new layer.

```
newClassLayer = classificationLayer(Name="ClassificationLayer_predictions");
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);
```

Specify Training Options

Train the network using root mean squared propagation (RMSProp) optimization. Specify the hyperparameter settings for RMSProp by using the `trainingOptions` function.

Reduce `MaxEpochs` to a small number because the large amount of training data enables the network to reach convergence sooner. Specify a `MiniBatchSize` according to your available GPU memory. While larger mini-batch sizes can make the training faster, larger sizes can reduce the ability of the network to generalize. Set `ResetInputNormalization` to `false` to prevent a full read of the training data to compute normalization stats.

```
checkpointsDir = fullfile(dataDir,'checkpoints');
if ~exist(checkpointsDir,'dir')
    mkdir(checkpointsDir);
end

options = trainingOptions("rmsprop", ...
    MaxEpochs=1, ...
    MiniBatchSize=256, ...
    Shuffle="every-epoch", ...
    ValidationFrequency=250, ...
    InitialLearnRate=1e-4, ...
    SquaredGradientDecayFactor=0.99, ...
    ResetInputNormalization=false, ...
    ExecutionEnvironment="auto", ...
    Plots="training-progress", ...
    CheckpointPath=checkpointsDir);
```

Train Network or Download Pretrained Network

By default, this example downloads a pretrained version of the trained classification network using the helper function `downloadTrainedCamelyonNet`. The pretrained network can be used to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the network using the `trainNetwork` function.

Train on one or more GPUs, if available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
doTraining = false;
if doTraining
    options.ValidationData=dsValLabeled;
    trainedNet = trainNetwork(dsTrainLabeled,lgraph,options);
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(dataDir+"trainedCamelyonNet-"+modelDateTime+".mat","trainedNet");

else
    trainedCamelyonNet_url = "https://www.mathworks.com/supportfiles/vision/data/trainedCamelyonN
    dataDir = fullfile(tempdir,"Camelyon16");
    downloadTrainedCamelyonNet(trainedCamelyonNet_url,dataDir);
    load(fullfile(dataDir,"trainedCamelyonNet.mat"));
end
```

Download Test Data

The Camelyon16 test data set consists of 130 WSIs. These images have both normal and tumor tissue. The size of each file is approximately 2 GB.

To download the test data, go to the Camelyon17 website and click the first "CAMELYON16 data set" link. Open the "testing" directory, then follow these steps.

- Download the "lesion_annotations.zip" file. Extract all files to the directory specified by the `testAnnotationDir` variable.
- Open the "images" directory. Download the files to the directory specified by the `testImageDir` variable.

```
testDir = fullfile(dataDir,"testing");
testImageDir = fullfile(testDir,"images");
testAnnotationDir = fullfile(testDir,"lesion_annotations");
if ~exist(testDir,"dir")
    mkdir(testDir);
    mkdir(fullfile(testDir,"images"));
    mkdir(fullfile(testDir,"lesion_annotations"));
end
```

Preprocess Test Data

Create `blockedImage` Objects to Manage Test Images

Get the file names of the test images. Then, create an array of `blockedImage` (Image Processing Toolbox) objects that manage the test images. Each `blockedImage` object points to the corresponding image file on disk.

```
testFileSet = matlab.io.datastore.FileSet(testImageDir+filesep+"test*");
testImages = blockedImage(testFileSet);
```

Set the spatial referencing for all training data by using the `setSpatialReferencingForCamelyon16` helper function. This function is attached to the example as a supporting file. The `setSpatialReferencingForCamelyon16` function sets the `WorldStart` and `WorldEnd` properties of each `blockedImage` object using the spatial referencing information from the TIF file metadata.

```
testImages = setSpatialReferencingForCamelyon16(testImages);
```

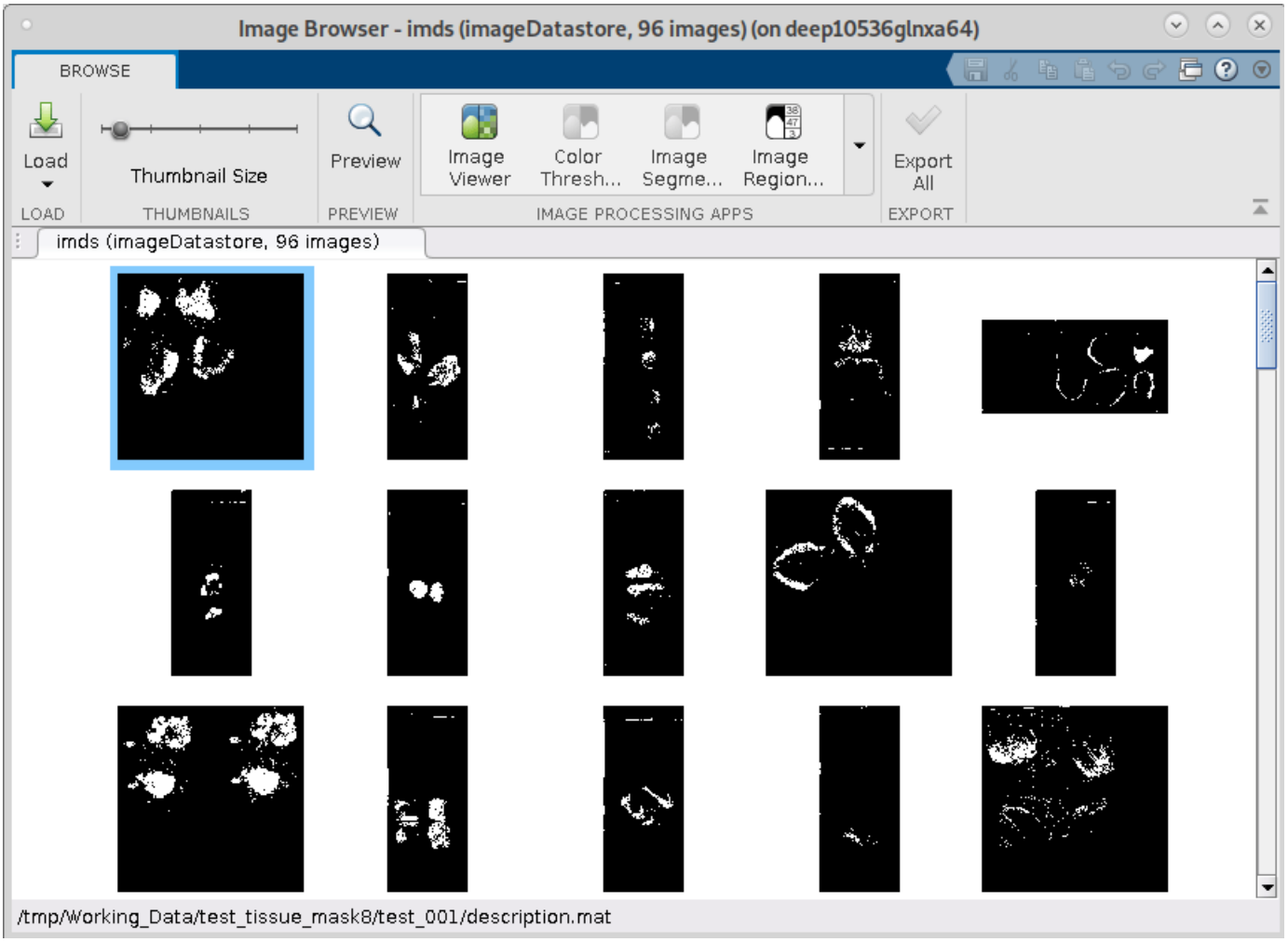
Create Tissue Masks

To process the WSI data efficiently, create a tissue mask for each test image. This process is the same as the one used for the preprocessing the normal training images. For more information, see "Preprocess Multiresolution Images for Training Classification Network" (Image Processing Toolbox).

```
normalMaskLevel = 8;

testTissueMaskDir = fullfile(testDir,"test_tissue_mask_level"+num2str(normalMaskLevel));

if ~isfolder(testTissueMaskDir)
    testTissueMasks = apply(testImages, @(bs)im2gray(bs.Data)<150, ...
        BlockSize=[512 512], ...
        Level=normalMaskLevel, ...
        UseParallel=canUseGPU, ...
        DisplayWaitbar=false, ...
        OutputLocation=testTissueMaskDir);
    save(fullfile(testTissueMaskDir,"testTissueMasks.mat"),"testTissueMasks")
else
    % Load previously saved data
    load(fullfile(testTissueMaskDir,"testTissueMasks.mat"),"testTissueMasks");
end
```

The tissue masks have only one level and are small enough to fit in memory. Display the tissue masks in the **Image Browser** app using the `browseBlockedImages` helper function. This helper function is attached to the example as a supporting file.

```
browseBlockedImages(testTissueMasks,1);
```



Preprocess Tumor Ground Truth Images

Specify the resolution level of the tumor masks.

```
tumorMaskLevel = 8;
```

Create a tumor mask for each ground truth tumor image using the `createMaskForCamelyon16TumorTissue` helper function. This helper function is attached to the example as a supporting file. The function performs these operations for each image:

- Read the (*x*, *y*) boundary coordinates for all ROIs in the annotated XML file.
- Separate the boundary coordinates for tumor and normal tissue ROIs into separate cell arrays.
- Convert the cell arrays of boundary coordinates to a binary blocked image using the `polyToBlockedImage` (Image Processing Toolbox) function. In the binary image, the ROI indicates tumor pixels and the background indicates normal tissue pixels. Pixels that are within both tumor and normal tissue ROIs are classified as background.

```
testTumorMaskDir = fullfile(testDir,['test_tumor_mask_level' num2str(tumorMaskLevel)]);
if ~isfolder(testTumorMaskDir)
    testTumorMasks = createMaskForCamelyon16TumorTissue(testImages,testAnnotationDir,testTumorMas
    save(fullfile(testTumorMaskDir,"testTumorMasks.mat"),"testTumorMasks")
```

```
else
    load(fullfile(testTumorMaskDir,"testTumorMasks.mat"),"testTumorMasks");
end
```

Predict Heatmaps of Tumor Probability

Use the trained classification network to predict a heatmap for each test image. The heatmap gives a probability score that each block is of the `tumor` class. The example performs these operations for each test image to create a heatmap:

- Select blocks using the `selectBlockLocations` (Image Processing Toolbox) function. Include all blocks that have at least one tissue pixel by specifying the `InclusionThreshold` name-value argument as `0`.
- Process batches of blocks using the `apply` (Image Processing Toolbox) function with the processing operations defined by the `predictBlock` helper function. The helper function is attached to the example as a supporting file. The `predictBlock` helper function calls the `predict` function on a block of data and returns the probability score that the block is `tumor`.
- Write the heatmap data to a TIF file using the `write` (Image Processing Toolbox) function. The final output after processing all blocks is a heatmap showing the probability of finding tumors over the entire WSI.

```
numTest = numel(testImages);
outputHeatmapsDir = fullfile(testDir,"heatmaps");
networkBlockSize = [299,299,3];
tic
for ind = 1:numTest
    % Check if TIF file already exists
    [~,id] = fileparts(testImages(ind).Source);
    outFile = fullfile(outputHeatmapsDir,id+".tif");
    if ~exist(outFile,"file")
        bls = selectBlockLocations(testImages(ind),Levels=1, ...
            BlockSize=networkBlockSize, ...
            Mask=testTissueMasks(ind),InclusionThreshold=0);

        % Resulting heat maps are in-memory blockedImage objects
        bhm = apply(testImages(ind),@(x)predictBlockForCamelyon16(x,trainedNet), ...
            Level=1,BlockLocationSet=bls,BatchSize=128, ...
            PadPartialBlocks=true,DisplayWaitBar=false);

        % Write results to a TIF file
        write(bhm,outFile,BlockSize=[512 512]);
    end
end
toc
```

Collect all of the written heatmaps as an array of `blockedImage` objects.

```
heatMapFileSet = matlab.io.datastore.FileSet(outputHeatmapsDir,FileExtensions=".tif");
bheatMapImages = blockedImage(heatMapFileSet);
```

Visualize Heatmap

Select a test image to display. On the left side of a figure, display the ground truth boundary coordinates as freehand ROIs using the `showCamelyon16TumorAnnotations` helper function. This helper function is attached to the example as a supporting file. Normal regions (shown with a green boundary) can occur inside tumor regions (shown with a red boundary).

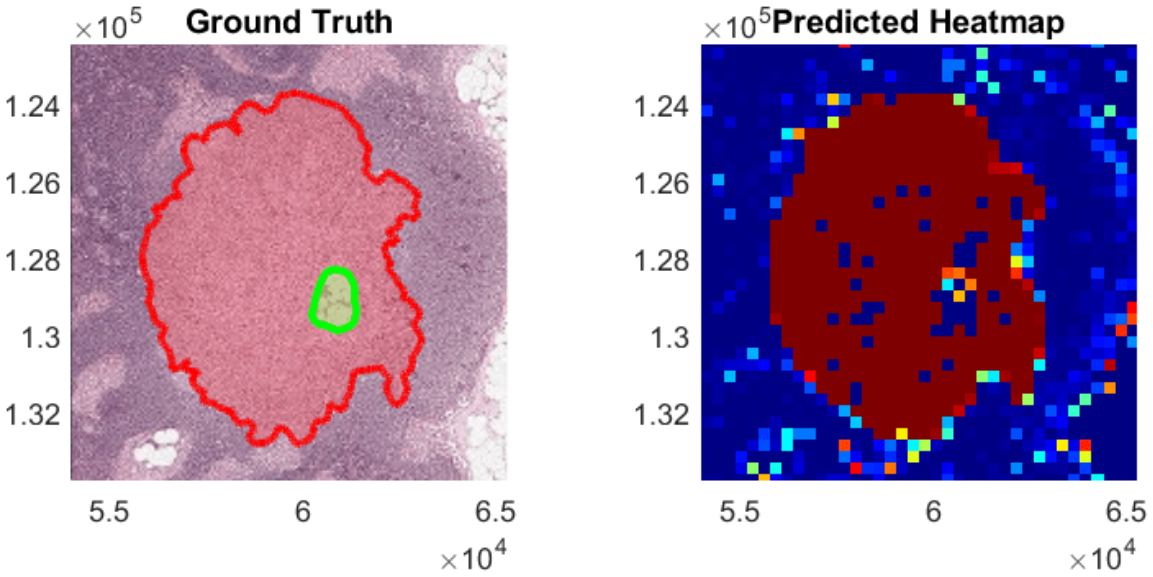
```
idx = 27;
figure
tiledlayout(1,2)
nexttile
hBim1 = showCamelyon16TumorAnnotations(testImages(idx),testAnnotationDir);
title("Ground Truth")
```

On the right side of the figure, display the heatmap for the test image.

```
nexttile
hBim2 = bigimageshow(bheatMapImages(idx),Interpolation="nearest");
colormap(jet)
```

Link the axes and zoom in to an area of interest.

```
linkaxes([hBim1.Parent,hBim2.Parent])
xlim([53982, 65269])
ylim([122475, 133762])
title("Predicted Heatmap")
```



Classify Test Images at Specific Threshold

To classify blocks as `tumor` or `normal`, apply a threshold to the heatmap probability values.

Pick a threshold probability above which blocks are classified as `tumor`. Ideally, you would calculate this threshold value using receiver operating characteristic (ROC) or precision-recall curves on the validation data set.

```
thresh = 0.8;
```

Classify the blocks in each test image and calculate the confusion matrix using the `apply` (Image Processing Toolbox) function with the processing operations defined by the `computeBlockConfusionMatrixForCamelyon16` helper function. The helper function is attached to the example as a supporting file.

The `computeBlockConfusionMatrixForCamelyon16` helper function performs these operations on each heatmap:

- Resize and refine the ground truth mask to match the size of the heatmap.
- Apply the threshold on the heatmap.
- Calculate a confusion matrix for all of the blocks at the finest resolution level. The confusion matrix gives the number of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) classification predictions.
- Save the total counts of TP, FP, TN, and FN blocks as a structure in a blocked image. The blocked image is returned as an element in the array of blocked images, `bcmatrix`.
- Save a numeric labeled image of the classification predictions in a blocked image. The values 0, 1, 2, and 3 correspond to TN, FP, FN, and TP results, respectively. The blocked image is returned as an element in the array of blocked images, `bcmatrixImage`.

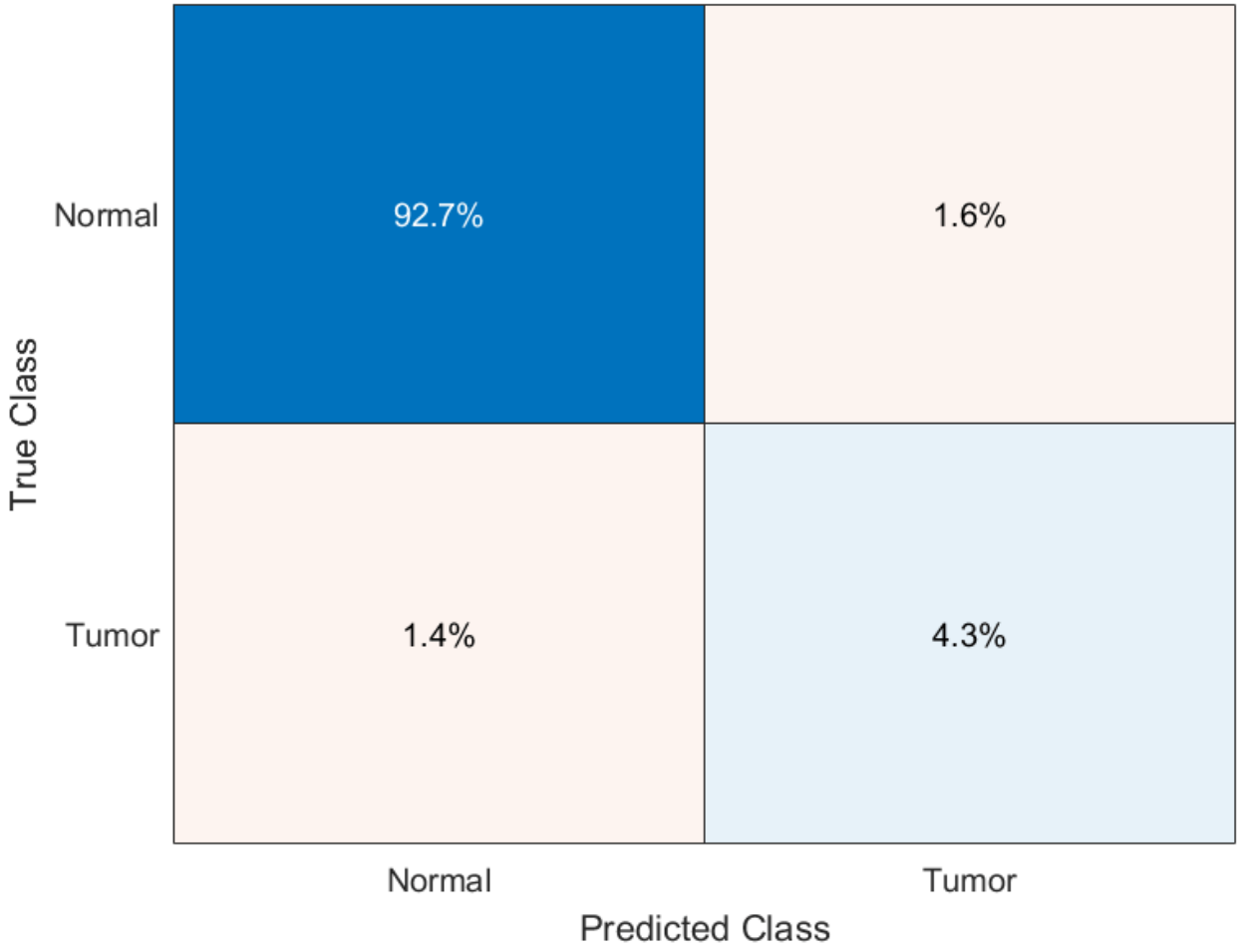
```
for ind = 1:numTest
    [bcmatrix(ind),bcmatrixImage{ind}] = apply(bheatMapImages(ind), ...
        @(bs,tumorMask,tissueMask)computeBlockConfusionMatrixForCamelyon16(bs,tumorMask,tissueMas
        ExtraImages=[testTumorMasks(ind),testTissueMasks(ind)]);
end
```

Calculate the global confusion matrix over all test images.

```
cmArray = arrayfun(@(c)gather(c),bcmatrix);
cm = [sum([cmArray.tp]),sum([cmArray.fp]);
    sum([cmArray.fn]),sum([cmArray.tn])];
```

Display the confusion chart of the normalized global confusion matrix. The majority of blocks in the WSI images are of normal tissue, resulting in a high percentage of true negative predictions.

```
figure
confusionchart(cm,["Tumor","Normal"],Normalization="total-normalized")
```



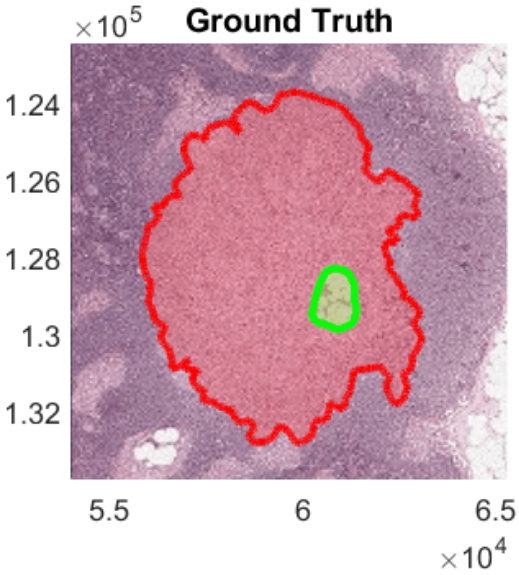
Visualize Classification Results

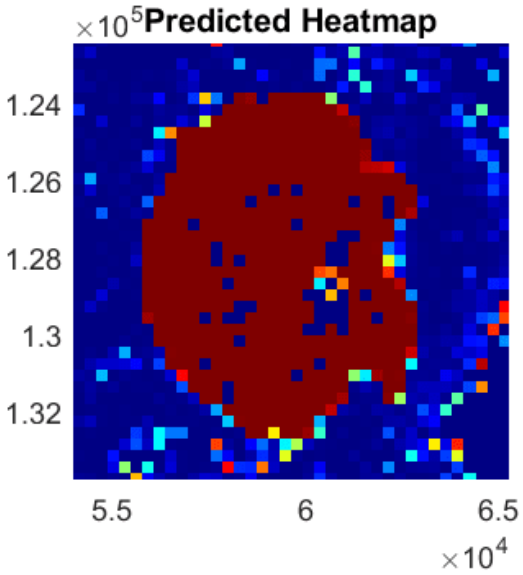
Compare the ground truth ROI boundary coordinates with the classification results. On the left side of a figure, display the ground truth boundary coordinates as freehand ROIs. On the right side of the figure, display the test image and overlay a color on each block based on the confusion matrix. Display true positives as red, false positives as cyan, false negatives as yellow, and true negatives with no color.

False negatives and false positives appear around the edges of the tumor region, which indicates that the network has difficulty classifying blocks with partial classes.

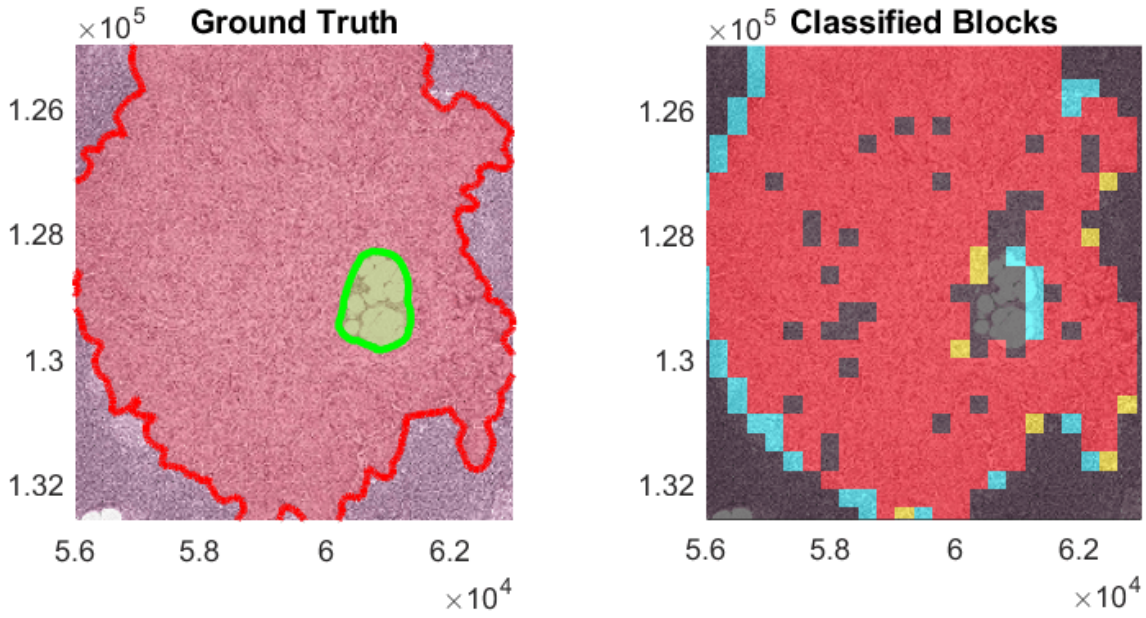
```
idx = 27;
figure
tiledlayout(1,2)
nexttile
hBim1 = showCamelyon16TumorAnnotations(testImages(idx),testAnnotationDir);
title("Ground Truth")
nexttile
hBim2 = bigimageshow(testImages(idx));
cmColormap = [0 0 0; 0 1 1; 1 1 0; 1 0 0];
```

```
showlabels(hBim2,bcmatrixImage{idx}, ...
    Colormap=cmColormap,AlphaData=bcmatrixImage{idx})
```






```
title("Classified Blocks")
linkaxes([hBim1.Parent,hBim2.Parent])
xlim([56000 63000])
ylim([125000 132600])
```



Note: To reduce the classification error around the perimeter of the tumor, you can retrain the network with less homogenous blocks. When preprocessing the `Tumor` blocks of the training data set, reduce the value of the `InclusionThreshold` name-value argument.

Quantify Network Prediction with AUC-ROC Curve

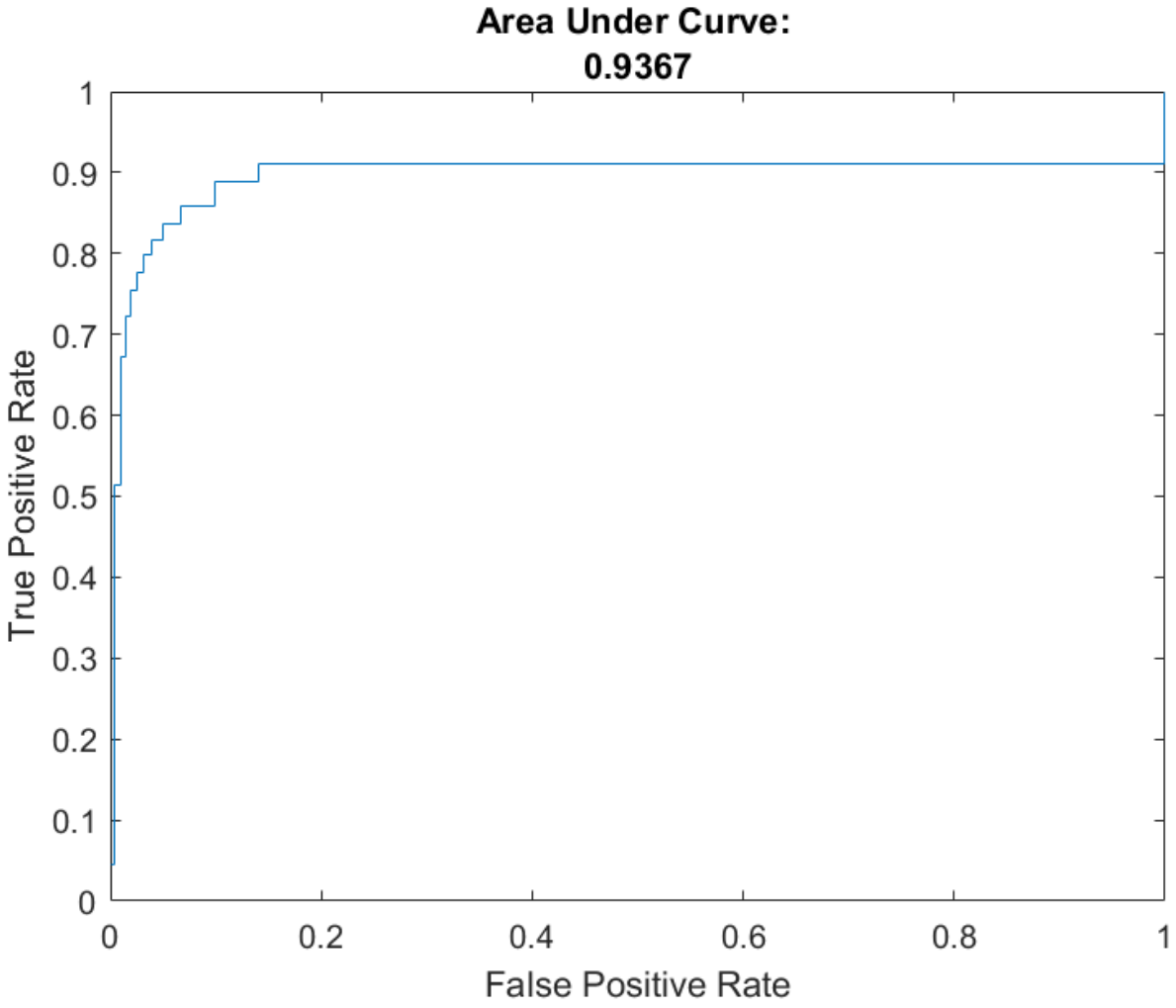
Calculate the ROC curve values at different thresholds by using the `computeROCCurvesForCamelyon16` helper function. This helper function is attached to the example as a supporting file.

```
threshs = [1 0.99 0.9:-.1:.1 0.05 0];
[tpr,fpr,ppv] = computeROCCurvesForCamelyon16(bheatMapImages,testTumorMasks,testTissueMasks,thres
```

Calculate the area under the curve (AUC) metric using the `trapz` function. The metric returns a value in the range [0, 1], where 1 indicates perfect model performance. The AUC for this data set is close to 1. You can use the AUC to fine-tune the training process.

```
figure
stairs(fpr,tpr,"-");
ROCAUC = trapz(fpr,tpr);
```

```
title(["Area Under Curve: " num2str(ROCAUC)]);
xlabel("False Positive Rate")
ylabel("True Positive Rate")
```



References

[1] Ehteshami Bejnordi, Babak, Mitko Veta, Paul Johannes van Diest, Bram van Ginneken, Nico Karssemeijer, Geert Litjens, Jeroen A. W. M. van der Laak, et al. “Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer.” *JAMA* 318, no. 22 (December 12, 2017): 2199–2210. https://doi.org/10.1001/jama.2017.14585.

[2] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. “Rethinking the Inception Architecture for Computer Vision.” Preprint, submitted December 2, 2015. https://arxiv.org/abs/1512.00567v3.

[3] ImageNet. https://www.image-net.org.

See Also

`blockedImageDatastore` | `blockedImage` | `blockLocationSet` | `selectBlockLocations` | `bigimageshow` | `trainingOptions` | `trainNetwork`

Related Examples

- “Preprocess Multiresolution Images for Training Classification Network” (Image Processing Toolbox)

More About

- “Set Up Spatial Referencing for Blocked Images” (Image Processing Toolbox)
- “Process Blocked Images Efficiently Using Partial Images or Lower Resolutions” (Image Processing Toolbox)
- “Process Blocked Images Efficiently Using Mask” (Image Processing Toolbox)
- “Create Labeled Blocked Image from ROIs and Masks” (Image Processing Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “List of Deep Learning Layers” on page 1-21

Unsupervised Day-to-Dusk Image Translation Using UNIT

This example shows how to translate images between daytime and dusk lighting conditions using an unsupervised image-to-image translation network (UNIT).

Domain translation is the task of transferring styles and characteristics from one image domain to another. This technique can be extended to other image-to-image learning operations, such as image enhancement, image colorization, defect generation, and medical image analysis.

UNIT [1] on page 9-0 is a type of generative adversarial network (GAN) that consists of one generator network and two discriminator networks that you train simultaneously to maximize the overall performance. For more information about UNIT, see “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox).

Download Data Set

This example uses the CamVid data set [2] on page 9-0 from the University of Cambridge for training. This data set is a collection of 701 images containing street-level views obtained while driving.

Download the CamVid data set. The download time depends on your internet connection.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.z

dataDir = fullfile(tempdir,'CamVid');
downloadCamVidImageData(dataDir,imageURL);
imgDir = fullfile(dataDir,"images","701_StillsRaw_full");
```

Load Day and Dusk Data

The CamVid image data set includes 497 images acquired in daytime and 124 images acquired at dusk. The performance of the trained UNIT network is limited because the number of CamVid training images is relatively small, which limits the performance of the trained network. Further, some images belong to an image sequence and therefore are correlated with other images in the data set. To minimize the impact of these limitations, this example manually partitions the data into training and test data sets in a way that maximizes the variability of the training data.

Get the file names of the day and dusk images for training and testing by loading the file `camvidDayDuskDatasetFileNames.mat`. The training data sets consist of 263 day images and 107 dusk images. The test data sets consist of 234 day images and 17 dusk images.

```
load('camvidDayDuskDatasetFileNames.mat');
```

Create `imageDatastore` objects that manage the day and dusk images for training and testing.

```
imdsDayTrain = imageDatastore(fullfile(imgDir,trainDayNames));
imdsDuskTrain = imageDatastore(fullfile(imgDir,trainDuskNames));
imdsDayTest = imageDatastore(fullfile(imgDir,testDayNames));
imdsDuskTest = imageDatastore(fullfile(imgDir,testDuskNames));
```

Preview a training image from the day and dusk training data sets.

```
day = preview(imdsDayTrain);
dusk = preview(imdsDuskTrain);
montage({day,dusk})
```

Preprocess and Augment Training Data

Specify the image input size for the source and target images.

```
inputSize = [256,256,3];
```

Augment and preprocess the training data by using the `transform` function with custom preprocessing operations specified by the helper function `augmentDataForDayToDusk`. This function is attached to the example as a supporting file.

The `augmentDataForDayToDusk` function performs these operations:

1. Resize the image to the specified input size using bicubic interpolation.
2. Randomly flip the image in the horizontal direction.
3. Scale the image to the range [-1, 1]. This range matches the range of the final `tanhLayer` used in the generator.

```
imdsDayTrain = transform(imdsDayTrain, @(x)augmentDataForDayToDusk(x,inputSize));
imdsDuskTrain = transform(imdsDuskTrain, @(x)augmentDataForDayToDusk(x,inputSize));
```

Create Generator Network

Create a UNIT generator network using the `unitGenerator` (Image Processing Toolbox) function. The source and target encoder sections of the generator each consist of two downsampling blocks and five residual blocks. The encoder sections share two of the five residual blocks. Similarly, the source and target decoder sections of the generator each consist of two downsampling blocks and five residual blocks, and the decoder sections share two of the five residual blocks.

```
gen = unitGenerator(inputSize,'NumResidualBlocks',5,'NumSharedBlocks',2);
```

Visualize the generator network.

```
analyzeNetwork(gen)
```

Create Discriminator Networks

Create two discriminator networks, one for each of the source and target domains, using the `patchGANDiscriminator` (Image Processing Toolbox) function. Day is the source domain and dusk is the target domain.

```
discDay = patchGANDiscriminator(inputSize,"NumDownsamplingBlocks",4,"FilterSize",3, ...
    "ConvolutionWeightsInitializer","narrow-normal","NormalizationLayer","none");
discDusk = patchGANDiscriminator(inputSize,"NumDownsamplingBlocks",4,"FilterSize",3, ...
    "ConvolutionWeightsInitializer","narrow-normal","NormalizationLayer","none");
```

Visualize the discriminator networks.

```
analyzeNetwork(discDay);
analyzeNetwork(discDusk);
```

Define Model Gradients and Loss Functions

The `modelGradientsDisc` and `modelGradientGen` helper functions calculate the gradients and losses for the discriminators and generator, respectively. These functions are defined in the Supporting Functions on page 9-0 section of this example.

The objective of each discriminator is to correctly distinguish between real images (1) and translated images (0) for images in its domain. Each discriminator has a single loss function.

The objective of the generator is to generate translated images that the discriminators classify as real. The generator loss is a weighted sum of five types of losses: self-reconstruction loss, cycle consistency loss, hidden KL loss, cycle hidden KL loss, and adversarial loss.

Specify the weight factors for the various losses.

```
lossWeights.selfReconLossWeight = 10;
lossWeights.hiddenKLLossWeight = 0.01;
lossWeights.cycleConsisLossWeight = 10;
lossWeights.cycleHiddenKLLossWeight = 0.01;
lossWeights.advLossWeight = 1;
lossWeights.discLossWeight = 0.5;
```

Specify Training Options

Specify the options for Adam optimization. Train the network for 35 epochs. Specify identical options for the generator and discriminator networks.

- Specify an equal learning rate of 0.0001.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.
- Use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.
- Use weight decay regularization with a factor of 0.0001.
- Use a mini-batch size of 1 for training.

```
learnRate = 0.0001;
gradDecay = 0.5;
sqGradDecay = 0.999;
weightDecay = 0.0001;

genAvgGradient = [];
genAvgGradientSq = [];
```

```
discDayAvgGradient = [];
discDayAvgGradientSq = [];

discDuskAvgGradient = [];
discDuskAvgGradientSq = [];

miniBatchSize = 1;
numEpochs = 35;
```

Batch Training Data

Create a `minibatchqueue` object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` object that enables automatic differentiation in deep learning applications.

Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch). Set the `"DispatchInBackground"` name-value argument as the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
mbqDayTrain = minibatchqueue(imdsDayTrain,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","DispatchInBackground",canUseGPU);
mbqDuskTrain = minibatchqueue(imdsDuskTrain,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","DispatchInBackground",canUseGPU);
```

Train Network

By default, the example downloads a pretrained version of the UNIT generator for the CamVid data set by using the helper function `downloadTrainedDayDuskGeneratorNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradientsDisc` and `modelGradientGen` helper functions.
- Update the network parameters using the `adamupdate` function.
- Display the input and translated images for both the source and target domains after each epoch.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 88 hours on an NVIDIA Titan RTX.

```
doTraining = false;
if doTraining
    % Create a figure to show the results
    figure("Units","Normalized");
    for iPlot = 1:4
        ax(iPlot) = subplot(2,2,iPlot);
    end

    iteration = 0;
```

```
% Loop over epochs
for epoch = 1:numEpochs

    % Shuffle data every epoch
    reset(mbqDayTrain);
    shuffle(mbqDayTrain);
    reset(mbqDuskTrain);
    shuffle(mbqDuskTrain);

    % Run the loop until all the images in the mini-batch queue mbqDayTrain are processed
    while hasdata(mbqDayTrain)
        iteration = iteration + 1;

        % Read data from the day domain
        imDay = next(mbqDayTrain);

        % Read data from the dusk domain
        if hasdata(mbqDuskTrain) == 0
            reset(mbqDuskTrain);
            shuffle(mbqDuskTrain);
        end
        imDusk = next(mbqDuskTrain);

        % Calculate discriminator gradients and losses
        [discDayGrads,discDuskGrads,discDayLoss,disDuskLoss] = dlfeval(@modelGradientDisc, .
            gen,discDay,discDusk,imDay,imDusk,lossWeights.discLossWeight);

        % Apply weight decay regularization on day discriminator gradients
        discDayGrads = dlupdate(@(g,w) g+weightDecay*w,discDayGrads,discDay.Learnables);

        % Update parameters of day discriminator
        [discDay,discDayAvgGradient,discDayAvgGradientSq] = adamupdate(discDay,discDayGrads,
            discDayAvgGradient,discDayAvgGradientSq,iteration,learnRate,gradDecay,sqGradDeca

        % Apply weight decay regularization on dusk discriminator gradients
        discDuskGrads = dlupdate(@(g,w) g+weightDecay*w,discDuskGrads,discDusk.Learnables);

        % Update parameters of dusk discriminator
        [discDusk,discDuskAvgGradient,discDuskAvgGradientSq] = adamupdate(discDusk,discDuskG
            discDuskAvgGradient,discDuskAvgGradientSq,iteration,learnRate,gradDecay,sqGradDe

        % Calculate generator gradient and loss
        [genGrad,genLoss,images] = dlfeval(@modelGradientGen,gen,discDay,discDusk,imDay,imDus

        % Apply weight decay regularization on generator gradients
        genGrad = dlupdate(@(g,w) g+weightDecay*w,genGrad,gen.Learnables);

        % Update parameters of generator
        [gen,genAvgGradient,genAvgGradientSq] = adamupdate(gen,genGrad,genAvgGradient, ...
            genAvgGradientSq,iteration,learnRate,gradDecay,sqGradDecay);
    end

    % Display the results
    updateTrainingPlotDayToDusk(ax,images{:});
end

% Save the trained network
modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
```

```
    save(strcat("trainedDayDuskUNITGeneratorNet-",modelDateTime,"-Epoch-",num2str(numEpochs),".ma

else
    net_url = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedDayDuskUNITGeneratorNet
    downloadTrainedDayDuskGeneratorNet(net_url,dataDir);
    load(fullfile(dataDir,'trainedDayDuskUNITGeneratorNet.mat'));
end
```

Evaluate Source-to-Target Translation

Source-to-target image translation uses the UNIT generator to generate an image in the target domain (dusk) from an image in the source domain (day).

Read an image from the datastore of day test images.

```
idxToTest = 1;
dayTestImage = readimage(imdsDayTest,idxToTest);
```

Convert the image to data type `single` and normalize the image to the range [-1, 1].

```
dayTestImage = im2single(dayTestImage);
dayTestImage = (dayTestImage-0.5)/0.5;
```

Create a `dlarray` object that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a `gpuArray` object.

```
dlDayImage = dlarray(dayTestImage,'SSCB');
if canUseGPU
    dlDayImage = gpuArray(dlDayImage);
end
```

Translate the input day image to the dusk domain using the `unitPredict` (Image Processing Toolbox) function.

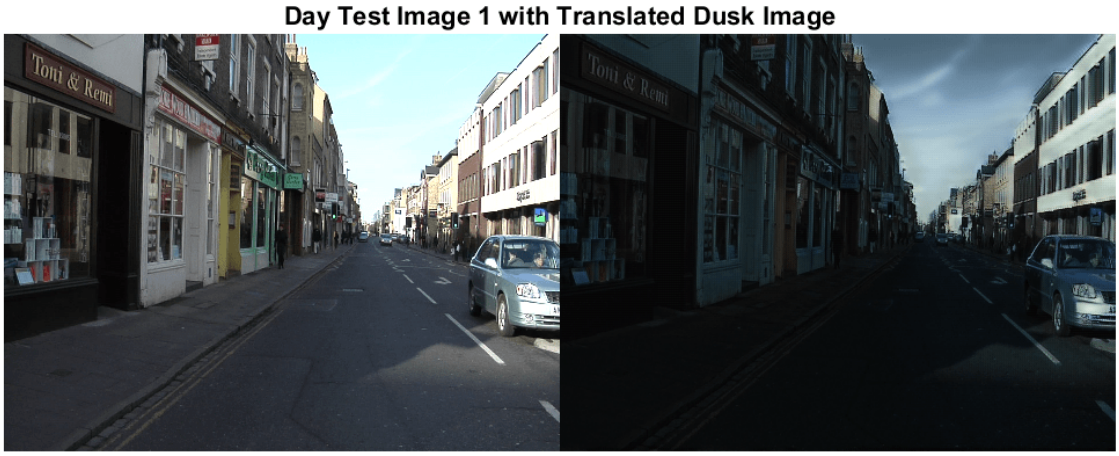
```
dlDayToDuskImage = unitPredict(gen,dlDayImage);
dayToDuskImage = extractdata(gather(dlDayToDuskImage));
```

The final layer of the generator network produces activations in the range [-1, 1]. For display, rescale the activations to the range [0, 1]. Also, rescale the input day image before display.

```
dayToDuskImage = rescale(dayToDuskImage);
dayTestImage = rescale(dayTestImage);
```

Display the input day image and its translated dusk version in a montage.

```
figure
montage({dayTestImage dayToDuskImage})
title(['Day Test Image ',num2str(idxToTest),' with Translated Dusk Image'])
```



Evaluate Target-to-Source Translation

Target-to-source image translation uses the UNIT generator to generate an image in the source domain (day) from an image in the target domain (dusk).

Read an image from the datastore of dusk test images.

```
idxToTest = 1;
duskTestImage = readimage(imdsDuskTest,idxToTest);
```

Convert the image to data type `single` and normalize the image to the range [-1, 1].

```
duskTestImage = im2single(duskTestImage);
duskTestImage = (duskTestImage-0.5)/0.5;
```

Create a `dlarray` object that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a `gpuArray` object.

```
dlDuskImage = dlarray(duskTestImage,'SSCB');
if canUseGPU
    dlDuskImage = gpuArray(dlDuskImage);
end
```

Translate the input dusk image to the day domain using the `unitPredict` (Image Processing Toolbox) function.

```
dlDuskToDayImage = unitPredict(gen,dlDuskImage,"OutputType","TargetToSource");
duskToDayImage = extractdata(gather(dlDuskToDayImage));
```

For display, rescale the activations to the range [0, 1]. Also, rescale the input dusk image before display.

```
duskToDayImage = rescale(duskToDayImage);
duskTestImage = rescale(duskTestImage);
```

Display the input dusk image and its translated day version in a montage.

```
montage({duskTestImage duskToDayImage})
title(['Test Dusk Image ',num2str(idxToTest),' with Translated Day Image'])
```

Supporting Functions

Model Gradients Functions

The `modelGradientDisc` helper function calculates the gradients and loss for the two discriminators.

```
function [discAGrads,discBGrads,discALoss,discBLoss] = modelGradientDisc(gen, ...
    discA,discB,ImageA,ImageB,discLossWeight)

    [~,fakeA,fakeB,~] = forward(gen,ImageA,ImageB);

    % Calculate loss of the discriminator for X_A
    outA = forward(discA,ImageA);
    outfA = forward(discA,fakeA);
    discALoss = discLossWeight*computeDiscLoss(outA,outfA);

    % Update parameters of the discriminator for X
    discAGrads = dlgradient(discALoss,discA.Learnables);

    % Calculate loss of the discriminator for X_B
    outB = forward(discB,ImageB);
    outfB = forward(discB,fakeB);
    discBLoss = discLossWeight*computeDiscLoss(outB,outfB);

    % Update parameters of the discriminator for Y
    discBGrads = dlgradient(discBLoss,discB.Learnables);

    % Convert the data type from dlarray to single
    discALoss = extractdata(discALoss);
    discBLoss = extractdata(discBLoss);
end
```

The `modelGradientGen` helper function calculates the gradients and loss for the generator.

```
function [genGrad,genLoss,images] = modelGradientGen(gen,discA,discB,ImageA,ImageB,lossWeights)

    [ImageAA,ImageBA,ImageAB,ImageBB] = forward(gen,ImageA,ImageB);
    hidden = forward(gen,ImageA,ImageB,'Outputs','encoderSharedBlock');

    [~,ImageABA,ImageBAB,~] = forward(gen,ImageBA,ImageAB);
    cycle_hidden = forward(gen,ImageBA,ImageAB,'Outputs','encoderSharedBlock');

    % Calculate different losses
    selfReconLoss = computeReconLoss(ImageA,ImageAA) + computeReconLoss(ImageB,ImageBB);
    hiddenKLLoss = computeKLLoss(hidden);
    cycleReconLoss = computeReconLoss(ImageA,ImageABA) + computeReconLoss(ImageB,ImageBAB);
    cycleHiddenKLLoss = computeKLLoss(cycle_hidden);

    outA = forward(discA,ImageBA);
    outB = forward(discB,ImageAB);
    advLoss = computeAdvLoss(outA) + computeAdvLoss(outB);

    % Calculate the total loss of generator as a weighted sum of five
    % losses
    genTotalLoss = ...
        selfReconLoss*lossWeights.selfReconLossWeight + ...
        hiddenKLLoss*lossWeights.hiddenKLLossWeight + ...
        cycleReconLoss*lossWeights.cycleConsisLossWeight + ...
        cycleHiddenKLLoss*lossWeights.cycleHiddenKLLossWeight + ...
        advLoss*lossWeights.advLossWeight;

    % Update the parameters of generator
    genGrad = dlgradient(genTotalLoss,gen.Learnables);

    % Convert the data type from dlarray to single
    genLoss = extractdata(genTotalLoss);
    images = {ImageA,ImageAB,ImageB,ImageBA};
end
```

Loss Functions

The `computeDiscLoss` helper function calculates the discriminator loss. Each discriminator loss is a sum of two components:

- The squared difference between a vector of ones and the predictions of the discriminator on real images, Y_{real}
- The squared difference between a vector of zeros and the predictions of the discriminator on generated images, $\widehat{Y}_{translated}$

$$discriminatorLoss = (1 - Y_{real})^2 + \left(0 - \widehat{Y}_{translated}\right)^2$$

```
function discLoss = computeDiscLoss(Yreal,Ytranslated)
    discLoss = mean(((1-Yreal).^2),"all") + ...
               mean(((0-Ytranslated).^2),"all");
end
```

The `computeAdvLoss` helper function calculates the adversarial loss for the generator. Adversarial loss is the squared difference between a vector of ones and the discriminator predictions on the translated image.

$$adversarialLoss = \left(1 - \widehat{Y}_{translated}\right)^2$$

```
function advLoss = computeAdvLoss(Ytranslated)
    advLoss = mean(((Ytranslated-1).^2),"all");
end
```

The `computeReconLoss` helper function calculates the self-reconstruction loss and cycle-consistency loss for the generator. Self-reconstruction loss is the L^1 distance between the input images and their self-reconstructed versions. Cycle-consistency loss is the L^1 distance between the input images and their cycle-reconstructed versions.

$$selfReconstructionLoss = \left\|\left(Y_{real} - Y_{self-reconstructed}\right)\right\|_1$$

$$cycleConsistencyLoss = \left\|\left(Y_{real} - Y_{cycle-reconstructed}\right)\right\|_1$$

```
function reconLoss = computeReconLoss(Yreal,Yrecon)
    reconLoss = mean(abs(Yreal-Yrecon),"all");
end
```

The `computeKLLoss` helper function calculates the hidden KL loss and cycle-hidden KL loss for the generator. Hidden KL loss is the squared difference between a vector of zeros and the `encoderSharedBlock` activation for the self-reconstruction stream. Cycle-hidden KL loss is the squared difference between a vector of zeros and the `encoderSharedBlock` activation for the cycle-reconstruction stream.

$$hiddenKLLoss = \left(0 - Y_{encoderSharedBlockActivation}\right)^2$$

$$cycleHiddenKLLoss = \left(0 - Y_{encoderSharedBlockActivation}\right)^2$$

```
function klLoss = computeKLLoss(hidden)
    klLoss = mean(abs(hidden.^2),"all");
end
```

References

[1] Liu, Ming-Yu, Thomas Breuel, and Jan Kautz, "Unsupervised image-to-image translation networks". In *Advances in Neural Information Processing Systems,* 2017. https://arxiv.org/abs/1703.00848.

[2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A High-Definition Ground Truth Database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

See Also

`transform` | `unitGenerator` | `unitPredict` | `dlarray` | `dlfeval` | `adamupdate` | `minibatchqueue` | `patchGANDiscriminator`

More About

- “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209

- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

Quantify Image Quality Using Neural Image Assessment

This example shows how to analyze the aesthetic quality of images using a Neural Image Assessment (NIMA) convolutional neural network (CNN).

Image quality metrics provide an objective measure of image quality. An effective metric provides quantitative scores that correlate well with a subjective perception of quality by a human observer. Quality metrics enable the comparison of image processing algorithms.

NIMA [1] on page 9-0 is a no-reference technique that predicts the quality of an image without relying on a pristine reference image, which is frequently unavailable. NIMA uses a CNN to predict a distribution of quality scores for each image.

Evaluate Image Quality Using Trained NIMA Model

Download a pretrained NIMA neural network by using the helper function `downloadTrainedNIMANet`. The helper function is attached to the example as a supporting file. This model predicts a distribution of quality scores for each image in the range [1, 10], where 1 and 10 are the lowest and the highest possible values for the score, respectively. A high score indicates good image quality.

```
imageDir = fullfile(tempdir,"LIVEInTheWild");
if ~exist(imageDir,'dir')
    mkdir(imageDir);
end
trainedNIMA_url = 'https://ssd.mathworks.com/supportfiles/image/data/trainedNIMA.zip';
downloadTrainedNIMANet(trainedNIMA_url,imageDir);
load(fullfile(imageDir,'trainedNIMA.mat'));
```

You can evaluate the effectiveness of the NIMA model by comparing the predicted scores for a high-quality and lower quality image.

Read a high-quality image into the workspace.

```
imOriginal = imread('kobi.png');
```

Reduce the aesthetic quality of the image by applying a Gaussian blur. Display the original image and the blurred image in a montage. Subjectively, the aesthetic quality of the blurred image is worse than the quality of the original image.

```
imBlur = imgaussfilt(imOriginal,5);
montage({imOriginal,imBlur})
```

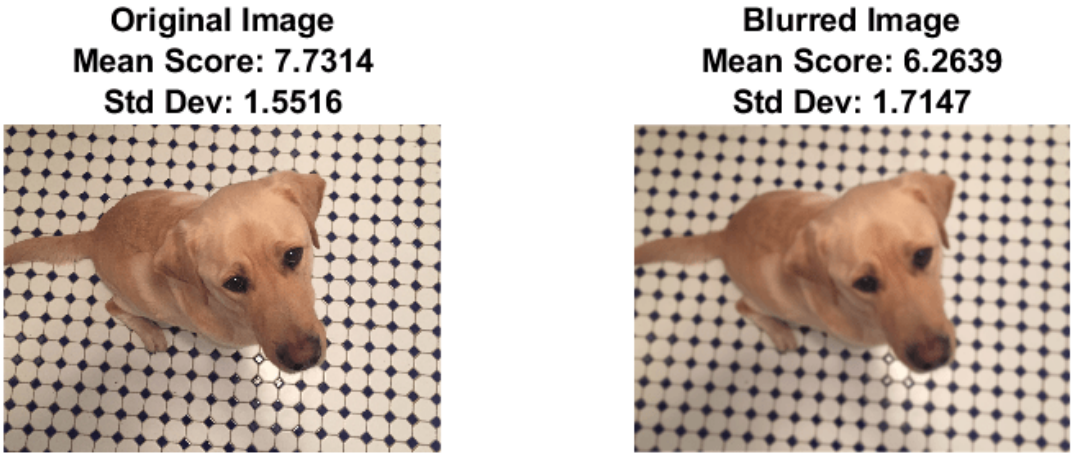
Predict the NIMA quality score distribution for the two images using the `predictNIMAScore` helper function. This function is attached to the example as a supporting file.

The `predictNIMAScore` function returns the mean and standard deviation of the NIMA score distribution for an image. The predicted mean score is a measure of the quality of the image. The standard deviation of scores can be considered a measure of the confidence level of the predicted mean score.

```
[meanOriginal,stdOriginal] = predictNIMAScore(dlnet,imOriginal);
[meanBlur,stdBlur] = predictNIMAScore(dlnet,imBlur);
```

Display the images along with the mean and standard deviation of the score distributions predicted by the NIMA model. The NIMA model correctly predicts scores for these images that agree with the subjective visual assessment.

```
figure
t = tiledlayout(1,2);
displayImageAndScoresForNIMA(t,imOriginal,meanOriginal,stdOriginal,"Original Image")
displayImageAndScoresForNIMA(t,imBlur,meanBlur,stdBlur,"Blurred Image")
```



The rest of this example shows how to train and evaluate a NIMA model.

Download LIVE In the Wild Data Set

This example uses the LIVE In the Wild data set [2] on page 9-0 , which is a public-domain subjective image quality challenge database. The data set contains 1162 photos captured by mobile devices, with 7 additional images provided to train the human scorers. Each image is rated by an average of 175 individuals on a scale of [1, 100]. The data set provides the mean and standard deviation of the subjective scores for each image.

Download the data set by following the instructions outlined in LIVE In the Wild Image Quality Challenge Database. Extract the data into the directory specified by the `imageDir` variable. When extraction is successful, `imageDir` contains two directories: `Data` and `Images`.

Load LIVE In the Wild Data

Get the file paths to the images.

```
imageData = load(fullfile(imageDir,'Data','AllImages_release.mat'));
imageData = imageData.AllImages_release;
```

```
nImg = length(imageData);
imageList(1:7) = fullfile(imageDir,'Images','trainingImages',imageData(1:7));
imageList(8:nImg) = fullfile(imageDir,'Images',imageData(8:end));
```

Create an image datastore that manages the image data.

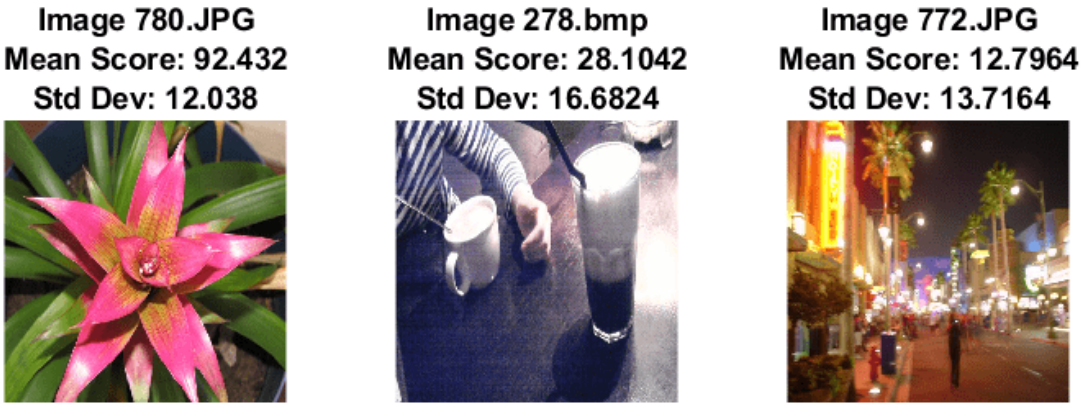
```
imds = imageDatastore(imageList);
```

Load the mean and standard deviation data corresponding to the images.

```
meanData = load(fullfile(imageDir,'Data','AllMOS_release.mat'));
meanData = meanData.AllMOS_release;
stdData = load(fullfile(imageDir,'Data','AllStdDev_release.mat'));
stdData = stdData.AllStdDev_release;
```

Optionally, display a few sample images from the data set with the corresponding mean and standard deviation values.

```
figure
t = tiledlayout(1,3);
idx1 = 785;
displayImageAndScoresForNIMA(t,readimage(imds,idx1), ...
    meanData(idx1),stdData(idx1),"Image "+imageData(idx1))
idx2 = 203;
displayImageAndScoresForNIMA(t,readimage(imds,idx2), ...
    meanData(idx2),stdData(idx2),"Image "+imageData(idx2))
idx3 = 777;
displayImageAndScoresForNIMA(t,readimage(imds,idx3), ...
    meanData(idx3),stdData(idx3),"Image "+imageData(idx3))
```



Preprocess and Augment Data

Preprocess the images by resizing them to 256-by-256 pixels.

```
rescaleSize = [256 256];
imds = transform(imds,@(x)imresize(x,rescaleSize));
```

The NIMA model requires a distribution of human scores, but the LIVE data set provides only the mean and standard deviation of the distribution. Approximate an underlying distribution for each image in the LIVE data set using the `createNIMAScoreDistribution` helper function. This function is attached to the example as a supporting file.

The `createNIMAScoreDistribution` rescales the scores to the range [1, 10], then generates maximum entropy distribution of scores from the mean and standard deviation values.

```
newMaxScore = 10;
prob = createNIMAScoreDistribution(meanData,stdData);
cumProb = cumsum(prob,2);
```

Create an `arrayDatastore` that manages the score distributions.

```
probDS = arrayDatastore(cumProb','IterationDimension',2);
```

Combine the datastores containing the image data and score distribution data.

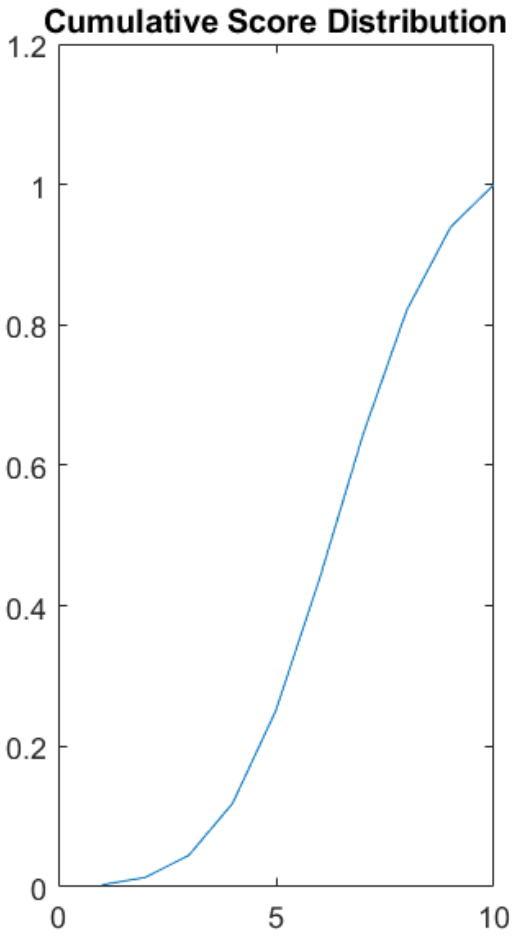
```
dsCombined = combine(imds,probDS);
```

Preview the output of reading from the combined datastore.

```
sampleRead = preview(dsCombined)
```

```
sampleRead=1×2 cell array
    {256×256×3 uint8}    {10×1 double}
```

```
figure
tiledlayout(1,2)
nexttile
imshow(sampleRead{1})
title("Sample Image from Data Set")
nexttile
plot(sampleRead{2})
title("Cumulative Score Distribution")
```



Split Data for Training, Validation, and Testing

Partition the data into training, validation, and test sets. Allocate 70% of the data for training, 15% for validation, and the remainder for testing.

```
numTrain = floor(0.70 * nImg);
numVal = floor(0.15 * nImg);

Idx = randperm(nImg);
idxTrain = Idx(1:numTrain);
idxVal = Idx(numTrain+1:numTrain+numVal);
idxTest = Idx(numTrain+numVal+1:nImg);

dsTrain = subset(dsCombined,idxTrain);
dsVal = subset(dsCombined,idxVal);
dsTest = subset(dsCombined,idxTest);
```

Augment Training Data

Augment the training data using the `augmentImageTest` helper function. This function is attached to the example as a supporting file. The `augmentDataForNIMA` function performs these augmentation operations on each training image:

- Crop the image to 224-by-244 pixels to reduce overfitting.
- Flip the image horizontally with 50% probability.

```
inputSize = [224 224];
dsTrain = transform(dsTrain,@(x)augmentDataForNIMA(x,inputSize));
```

Calculate Training Set Statistics for Input Normalization

The input layer of the network performs z-score normalization of the training images. Calculate the mean and standard deviation of the training images for use in z-score normalization.

```
meanImage = zeros([inputSize 3]);
meanImageSq = zeros([inputSize 3]);
while hasdata(dsTrain)
    dat = read(dsTrain);
    img = double(dat{1});
    meanImage = meanImage + img;
    meanImageSq = meanImageSq + img.^2;
end
meanImage = meanImage/numTrain;
meanImageSq = meanImageSq/numTrain;
varImage = meanImageSq - meanImage.^2;
stdImage = sqrt(varImage);
```

Reset the datastore to its initial state.

```
reset(dsTrain);
```

Batch Training Data

Create a `minibatchqueue` object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` object that enables automatic differentiation in deep learning applications.

Specify the mini-batch data extraction format as `'SSCB'` (spatial, spatial, channel, batch). Set the `'DispatchInBackground'` name-value argument to the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
miniBatchSize = 128;
mbqTrain = minibatchqueue(dsTrain,'MiniBatchSize',miniBatchSize, ...
    'PartialMiniBatch','discard','MiniBatchFormat',{'SSCB',''}, ...
    'DispatchInBackground',canUseGPU);
mbqVal = minibatchqueue(dsVal,'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFormat',{'SSCB',''},'DispatchInBackground',canUseGPU);
```

Load and Modify MobileNet-v2 Network

This example starts with a MobileNet-v2 [3] on page 9-0 CNN trained on ImageNet [4] on page 9-0 . The example modifies the network by replacing the last layer of the MobileNet-v2 network with a fully connected layer with 10 neurons, each representing a discrete score from 1 through 10. The network predicts the probability of each score for each image. The example normalizes the outputs of the fully connected layer using a softmax activation layer.

The `mobilenetv2` function returns a pretrained MobileNet-v2 network. This function requires the Deep Learning Toolbox™ Model *for MobileNet-v2 Network* support package. If this support package is not installed, then the function provides a download link.

```
net = mobilenetv2;
```

Convert the network into a `layerGraph` object.

```
lgraph = layerGraph(net);
```

The network has an image input size of 224-by-224 pixels. Replace the input layer with an image input layer that performs z-score normalization on the image data using the mean and standard deviation of the training images.

```
inLayer = imageInputLayer([inputSize 3],'Name','input','Normalization','zscore','Mean',meanImage
lgraph = replaceLayer(lgraph,'input_1',inLayer);
```

Replace the original final classification layer with a fully connected layer with 10 neurons. Add a softmax layer to normalize the outputs. Set the learning rate of the fully connected layer to 10 times the learning rate of the baseline CNN layers. Apply a dropout of 75%.

```
lgraph = removeLayers(lgraph,{'ClassificationLayer_Logits','Logits_softmax','Logits'});
newFinalLayers = [
    dropoutLayer(0.75,'Name','drop')
    fullyConnectedLayer(newMaxScore,'Name','fc','WeightLearnRateFactor',10,'BiasLearnRateFactor'
    softmaxLayer('Name','prob')];
lgraph = addLayers(lgraph,newFinalLayers);
lgraph = connectLayers(lgraph,'global_average_pooling2d_1','drop');
dlnet = dlnetwork(lgraph);
```

Visualize the network using the Deep Network Designer app.

```
deepNetworkDesigner(lgraph)
```

Define Model Gradients and Loss Functions

The `modelGradients` helper function calculates the gradients and losses for each iteration of training the network. This function is defined in the Supporting Functions on page 9-0 section of this example.

The objective of the NIMA network is to minimize the earth mover's distance (EMD) between the ground truth and predicted score distributions. EMD loss considers the distance between classes when penalizing misclassification. Therefore, EMD loss performs better than a typical softmax cross-entropy loss used in classification tasks [5] on page 9-0 . This example calculates the EMD loss using the `earthMoverDistance` helper function, which is defined in the Supporting Functions on page 9-0 section of this example.

For the EMD loss function, use an r-norm distance with $r = 2$. This distance allows for easy optimization when you work with gradient descent.

Specify Training Options

Specify the options for SGDM optimization. Train the network for 150 epochs.

```
numEpochs = 150;
momentum = 0.9;
initialLearnRate = 3e-3;
decay = 0.95;
```

Train Network

By default, the example loads a pretrained version of the NIMA network. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradients` helper function.
- Update the network parameters using the `sgdmupdate` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

```
doTraining = false;
if doTraining
    iteration = 0;
    velocity = [];
    start = tic;

    [hFig,lineLossTrain,lineLossVal] = initializeTrainingPlotNIMA;

    for epoch = 1:numEpochs

        shuffle (mbqTrain);
        learnRate = initialLearnRate/(1+decay*floor(epoch/10));

        while hasdata(mbqTrain)
```

```
            iteration = iteration + 1;
            [dlX,cdfY] = next(mbqTrain);
            [grad,loss] = dlfeval(@modelGradients,dlnet,dlX,cdfY);
            [dlnet,velocity] = sgdmupdate(dlnet,grad,velocity,learnRate,momentum);

            updateTrainingPlotNIMA(lineLossTrain,loss,epoch,iteration,start)

        end

        % Add validation data to plot
        [~,lossVal,~] = modelPredictions(dlnet,mbqVal);
        updateTrainingPlotNIMA(lineLossVal,lossVal,epoch,iteration,start)

    end

    % Save the trained network
    modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
    save(strcat("trainedNIMA-",modelDateTime,"-Epoch-",num2str(numEpochs),".mat"),'dlnet');

else
    load(fullfile(imageDir,'trainedNIMA.mat'));
end
```

Evaluate NIMA Model

Evaluate the performance of the model on the test data set using three metrics: EMD, binary classification accuracy, and correlation coefficients. The performance of the NIMA network on the test data set is in agreement with the performance of the reference NIMA model reported by Talebi and Milanfar [1] on page 9-0 .

Create a `minibatchqueue` object that manages the mini-batching of test data.

```
mbqTest = minibatchqueue(dsTest,'MiniBatchSize',miniBatchSize,'MiniBatchFormat',{'SSCB',''});
```

Calculate the predicted probabilities and ground truth cumulative probabilities of mini-batches of test data using the `modelPredictions` function. This function is defined in the Supporting Functions on page 9-0 section of this example.

```
[YPredTest,~,cdfYTest] = modelPredictions(dlnet,mbqTest);
```

Calculate the mean and standard deviation values of the ground truth and predicted distributions.

```
meanPred = extractdata(YPredTest)' * (1:10)';
stdPred = sqrt(extractdata(YPredTest)'*((1:10).^2)' - meanPred.^2);
origCdf = extractdata(cdfYTest);
origPdf = [origCdf(1,:); diff(origCdf)];
meanOrig = origPdf' * (1:10)';
stdOrig = sqrt(origPdf'*((1:10).^2)' - meanOrig.^2);
```

Calculate EMD

Calculate the EMD of the ground truth and predicted score distributions. For prediction, use an r-norm distance with $r = 1$. The EMD value indicates the closeness of the predicted and ground truth rating distributions.

```
EMDTest = earthMoverDistance(YPredTest,cdfYTest,1)
```

```
EMDTest =
  1×1 single gpuArray dlarray
```

```
    0.1158
```

Calculate Binary Classification Accuracy

For binary classification accuracy, convert the distributions to two classifications: high-quality and low-quality. Classify images with a mean score greater than a threshold as high-quality.

```
qualityThreshold = 5;
binaryPred = meanPred > qualityThreshold;
binaryOrig = meanOrig > qualityThreshold;
```

Calculate the binary classification accuracy.

```
binaryAccuracy = 100 * sum(binaryPred==binaryOrig)/length(binaryPred)
```

```
binaryAccuracy =

   84.6591
```

Calculate Correlation Coefficients

Large correlation values indicate a large positive correlation between the ground truth and predicted scores. Calculate the linear correlation coefficient (LCC) and Spearman's rank correlation coefficient (SRCC) for the mean scores.

```
meanLCC = corr(meanOrig,meanPred)
```

```
meanLCC =

  gpuArray single

    0.7265
```

```
meanSRCC = corr(meanOrig,meanPred,'type','Spearman')
```

```
meanSRCC =

  gpuArray single

    0.6451
```

Supporting Functions

Model Gradients Function

The `modelGradients` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding target cumulative probabilities `cdfY`. The function returns the gradients of the loss with respect to the learnable parameters in `dlnet` as well as the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,loss] = modelGradients(dlnet,dlX,cdfY)
    dlYPred = forward(dlnet,dlX);
    loss = earthMoverDistance(dlYPred,cdfY,2);
    gradients = dlgradient(loss,dlnet.Learnables);
end
```

Loss Function

The `earthMoverDistance` function calculates the EMD between the ground truth and predicted distributions for a specified *r*-norm value. The `earthMoverDistance` uses the `computeCDF` helper function to calculate the cumulative probabilities of the predicted distribution.

```
function loss = earthMoverDistance(YPred,cdfY,r)
    N = size(cdfY,1);
    cdfYPred = computeCDF(YPred);
    cdfDiff = (1/N) * (abs(cdfY - cdfYPred).^r);
    lossArray = sum(cdfDiff,1).^(1/r);
    loss = mean(lossArray);

end
function cdfY = computeCDF(Y)
% Given a probability mass function Y, compute the cumulative probabilities
    [N,miniBatchSize] = size(Y);
    L = repmat(triu(ones(N)),1,1,miniBatchSize);
    L3d = permute(L,[1 3 2]);
    prod = Y.*L3d;
    prodSum = sum(prod,1);
    cdfY = reshape(prodSum(:)',miniBatchSize,N)';
end
```

Model Predictions Function

The `modelPredictions` function calculates the estimated probabilities, loss, and ground truth cumulative probabilities of mini-batches of data.

```
function [dlYPred,loss,cdfYOrig] = modelPredictions(dlnet,mbq)
    reset(mbq);
    loss = 0;
    numObservations = 0;
    dlYPred = [];
    cdfYOrig = [];

    while hasdata(mbq)
        [dlX,cdfY] = next(mbq);
        miniBatchSize = size(dlX,4);

        dlY = predict(dlnet,dlX);
        loss = loss + earthMoverDistance(dlY,cdfY,2)*miniBatchSize;
        dlYPred = [dlYPred dlY];
        cdfYOrig = [cdfYOrig cdfY];

        numObservations = numObservations + miniBatchSize;

    end
    loss = loss / numObservations;
end
```

References

[1] Talebi, Hossein, and Peyman Milanfar. “NIMA: Neural Image Assessment.” IEEE Transactions on Image Processing 27, no. 8 (August 2018): 3998–4011. https://doi.org/10.1109/TIP.2018.2831899.

[2] LIVE: Laboratory for Image and Video Engineering. "LIVE In the Wild Image Quality Challenge Database." https://live.ece.utexas.edu/research/ChallengeDB/index.html.

[3] Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 4510–20. Salt Lake City, UT: IEEE, 2018. https://doi.org/10.1109/CVPR.2018.00474.

[4] ImageNet. https://www.image-net.org.

[5] Hou, Le, Chen-Ping Yu, and Dimitris Samaras. “Squared Earth Mover’s Distance-Based Loss for Training Deep Neural Networks.” Preprint, submitted November 30, 2016. https://arxiv.org/abs/1611.05916.

See Also

`mobilenetv2` | `transform` | `layerGraph` | `dlnetwork` | `minibatchqueue` | `predict` | `dlfeval` | `sgdmupdate`

More About

- “Image Quality Metrics” (Image Processing Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

Neural Style Transfer Using Deep Learning

This example shows how to apply the stylistic appearance of one image to the scene content of a second image using a pretrained VGG-19 network [1] on page 9-0 .

Load Data

Load the style image and content image. This example uses the distinctive Van Gogh painting "Starry Night" as the style image and a photograph of a lighthouse as the content image.

```
styleImage = im2double(imread('starryNight.jpg'));
contentImage = imread('lighthouse.png');
```

Display the style image and content image as a montage.

```
imshow(imtile({styleImage,contentImage},'BackgroundColor','w'));
```

Load Feature Extraction Network

In this example, you use a modified pretrained VGG-19 deep neural network to extract the features of the content and style image at various layers. These multilayer features are used to compute respective content and style losses. The network generates the stylized transfer image using the combined loss.

To get a pretrained VGG-19 network, install `vgg19`. If you do not have the required support packages installed, then the software provides a download link.

```
net = vgg19;
```

To make the VGG-19 network suitable for feature extraction, remove all of the fully connected layers from the network.

```
lastFeatureLayerIdx = 38;
layers = net.Layers;
layers = layers(1:lastFeatureLayerIdx);
```

The max pooling layers of the VGG-19 network cause a fading effect. To decrease the fading effect and increase the gradient flow, replace all max pooling layers with average pooling layers [1] on page 9-0 .

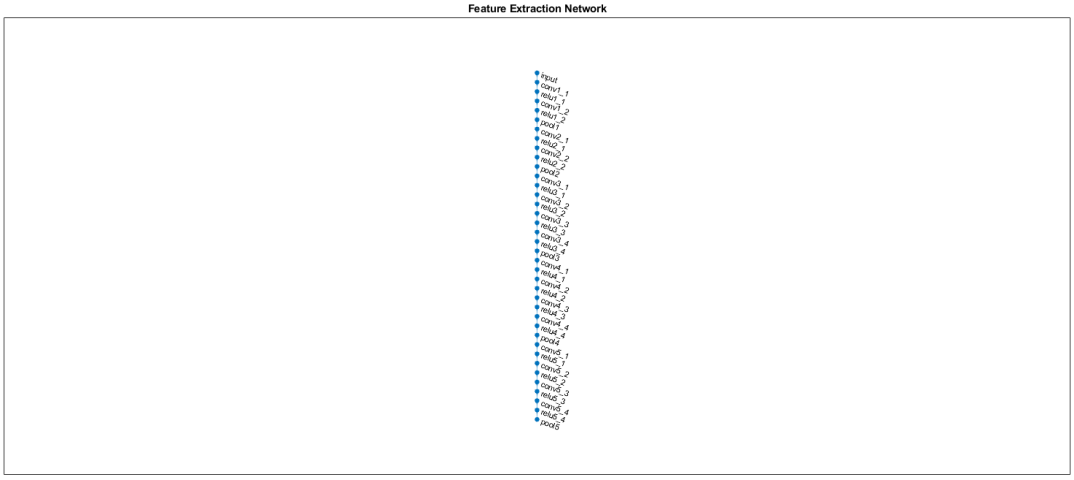
```
for l = 1:lastFeatureLayerIdx
    layer = layers(l);
    if isa(layer,'nnet.cnn.layer.MaxPooling2DLayer')
        layers(l) = averagePooling2dLayer(layer.PoolSize,'Stride',layer.Stride,'Name',layer.Name
    end
end
```

Create a layer graph with the modified layers.

```
lgraph = layerGraph(layers);
```

Visualize the feature extraction network in a plot.

```
plot(lgraph)
title('Feature Extraction Network')
```



To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnet = dlnetwork(lgraph);
```

Preprocess Data

Resize the style image and content image to a smaller size for faster processing.

```
imageSize = [384,512];
styleImg = imresize(styleImage,imageSize);
contentImg = imresize(contentImage,imageSize);
```

The pretrained VGG-19 network performs classification on a channel-wise mean subtracted image. Get the channel-wise mean from the image input layer, which is the first layer in the network.

```
imgInputLayer = lgraph.Layers(1);
meanVggNet = imgInputLayer.Mean(1,1,:);
```

The values of the channel-wise mean are appropriate for images of floating point data type with pixel values in the range [0, 255]. Convert the style image and content image to data type `single` with range [0, 255]. Then, subtract the channel-wise mean from the style image and content image.

```
styleImg = rescale(single(styleImg),0,255) - meanVggNet;
contentImg = rescale(single(contentImg),0,255) - meanVggNet;
```

Initialize Transfer Image

The transfer image is the output image as a result of style transfer. You can initialize the transfer image with a style image, content image, or any random image. Initialization with a style image or content image biases the style transfer process and produces a transfer image more similar to the input image. In contrast, initialization with white noise removes the bias but takes longer to converge on the stylized image. For better stylization and faster convergence, this example initializes the output transfer image as a weighted combination of the content image and a white noise image.

```
noiseRatio = 0.7;
randImage = randi([-20,20],[imageSize 3]);
transferImage = noiseRatio.*randImage + (1-noiseRatio).*contentImg;
```

Define Loss Functions and Style Transfer Parameters

Content Loss

The objective of content loss is to make the features of the transfer image match the features of the content image. The content loss is computed as the mean squared difference between content image features and transfer image features for each content feature layer [1] on page 9-0 . $\widehat{Y}$ is the predicted feature map for the transfer image and Y is the predicted feature map for the content image. W_c^l is the content layer weight for the l^{th} layer. $H, W, C$are the height, width, and channels of the feature maps, respectively.

$$L_{content} = \sum_l W_c^l \times \frac{1}{HWC} \sum_{i,j} (\widehat{Y}_{i,j}^l - Y_{i,j}^l)^2$$

Specify the content feature extraction layer names. The features extracted from these layers are used to compute the content loss. In the VGG-19 network, training is more effective using features from deeper layers rather than features from shallow layers. Therefore, specify the content feature extraction layer as the fourth convolutional layer.

```
styleTransferOptions.contentFeatureLayerNames = {'conv4_2'};
```

Specify the weights of the content feature extraction layers.

```
styleTransferOptions.contentFeatureLayerWeights = 1;
```

Style Loss

The objective of style loss is to make the texture of the transfer image match the texture of the style image. The style representation of an image is represented as a Gram matrix. Therefore, the style loss is computed as the mean squared difference between the Gram matrix of the style image and the Gram matrix of the transfer image [1] on page 9-0 . Z and $\widehat{Z}$ are the predicted feature maps for the style and transfer image, respectively. G_Z and $G_{\widehat{Z}}$ are Gram matrices for style features and transfer features, respectively. W_s^l is the style layer weight for the l^{th} style layer.

$$G_{\hat{Z}} = \sum_{i,j} \hat{Z}_{i,j} \times \hat{Z}_{j,i}$$

$$G_Z = \sum_{i,j} Z_{i,j} \times Z_{j,i}$$

$$L_{style} = \sum_l W_s^l \times \frac{1}{(2HWC)^2} \sum (G_{\hat{Z}}^l - G_Z^l)^2$$

Specify the names of the style feature extraction layers. The features extracted from these layers are used to compute style loss.

```
styleTransferOptions.styleFeatureLayerNames = {'conv1_1','conv2_1','conv3_1','conv4_1','conv5_1'}
```

Specify the weights of the style feature extraction layers. Specify small weights for simple style images and increase the weights for complex style images.

```
styleTransferOptions.styleFeatureLayerWeights = [0.5,1.0,1.5,3.0,4.0];
```

Total Loss

The total loss is a weighted combination of content loss and style loss. α and β are weight factors for content loss and style loss, respectively.

$$L_{total} = \alpha \times L_{content} + \beta \times L_{style}$$

Specify the weight factors `alpha` and `beta` for content loss and style loss. The ratio of `alpha` to `beta` should be around 1e-3 or 1e-4 [1] on page 9-0 .

```
styleTransferOptions.alpha = 1;
styleTransferOptions.beta = 1e3;
```

Specify Training Options

Train for 2500 iterations.

```
numIterations = 2500;
```

Specify options for Adam optimization. Set the learning rate to 2 for faster convergence. You can experiment with the learning rate by observing your output image and losses. Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.

```
learningRate = 2;
trailingAvg = [];
trailingAvgSq = [];
```

Train the Network

Convert the style image, content image, and transfer image to `dlarray` objects with underlying type `single` and dimension labels `'SSC'`.

```
dlStyle = dlarray(styleImg,'SSC');
dlContent = dlarray(contentImg,'SSC');
dlTransfer = dlarray(transferImage,'SSC');
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). For GPU training, convert the data into a `gpuArray`.

```
if canUseGPU
    dlContent = gpuArray(dlContent);
    dlStyle = gpuArray(dlStyle);
    dlTransfer = gpuArray(dlTransfer);
end
```

Extract the content features from the content image.

```
numContentFeatureLayers = numel(styleTransferOptions.contentFeatureLayerNames);
contentFeatures = cell(1,numContentFeatureLayers);
[contentFeatures{:}] = forward(dlnet,dlContent,'Outputs',styleTransferOptions.contentFeatureLaye
```

Extract the style features from the style image.

```
numStyleFeatureLayers = numel(styleTransferOptions.styleFeatureLayerNames);
styleFeatures = cell(1,numStyleFeatureLayers);
[styleFeatures{:}] = forward(dlnet,dlStyle,'Outputs',styleTransferOptions.styleFeatureLayerNames
```

Train the model using a custom training loop. For each iteration:

- Calculate the content loss and style loss using the features of the content image, style image, and transfer image. To calculate the loss and gradients, use the helper function `imageGradients` (defined in the Supporting Functions on page 9-0 section of this example).
- Update the transfer image using the `adamupdate` function.
- Select the best style transfer image as the final output image.

```
figure

minimumLoss = inf;

for iteration = 1:numIterations
    % Evaluate the transfer image gradients and state using dlfeval and the
    % imageGradients function listed at the end of the example.
    [grad,losses] = dlfeval(@imageGradients,dlnet,dlTransfer,contentFeatures,styleFeatures,styleT
    [dlTransfer,trailingAvg,trailingAvgSq] = adamupdate(dlTransfer,grad,trailingAvg,trailingAvgSq

    if losses.totalLoss < minimumLoss
        minimumLoss = losses.totalLoss;
        dlOutput = dlTransfer;
    end

    % Display the transfer image on the first iteration and after every 50
    % iterations. The postprocessing steps are described in the "Postprocess
    % Transfer Image for Display" section of this example.
    if mod(iteration,50) == 0 || (iteration == 1)

        transferImage = gather(extractdata(dlTransfer));
        transferImage = transferImage + meanVggNet;
        transferImage = uint8(transferImage);
        transferImage = imresize(transferImage,size(contentImage,[1 2]));

        image(transferImage)
        title(['Transfer Image After Iteration ',num2str(iteration)])
```

```
        axis off image
        drawnow
    end

end
```

Postprocess Transfer Image for Display

Get the updated transfer image.

```
transferImage = gather(extractdata(dlOutput));
```

Add the network-trained mean to the transfer image.

```
transferImage = transferImage + meanVggNet;
```

Some pixel values can exceed the original range [0, 255] of the content and style image. You can clip the values to the range [0, 255] by converting the data type to `uint8`.

```
transferImage = uint8(transferImage);
```

Resize the transfer image to the original size of the content image.

```
transferImage = imresize(transferImage,size(contentImage,[1 2]));
```

Display the content image, transfer image, and style image in a montage.

```
imshow(imtile({contentImage,transferImage,styleImage}, ...
    'GridSize',[1 3],'BackgroundColor','w'));
```

Supporting Functions

Compute Image Loss and Gradients

The `imageGradients` helper function returns the loss and gradients using features of the content image, style image, and transfer image.

```
function [gradients,losses] = imageGradients(dlnet,dlTransfer,contentFeatures,styleFeatures,paran

    % Initialize transfer image feature containers.
    numContentFeatureLayers = numel(params.contentFeatureLayerNames);
    numStyleFeatureLayers = numel(params.styleFeatureLayerNames);

    transferContentFeatures = cell(1,numContentFeatureLayers);
    transferStyleFeatures = cell(1,numStyleFeatureLayers);

    % Extract content features of transfer image.
    [transferContentFeatures{:}] = forward(dlnet,dlTransfer,'Outputs',params.contentFeatureLayerN

    % Extract style features of transfer image.
    [transferStyleFeatures{:}] = forward(dlnet,dlTransfer,'Outputs',params.styleFeatureLayerNames

    % Compute content loss.
    cLoss = contentLoss(transferContentFeatures,contentFeatures,params.contentFeatureLayerWeights

    % Compute style loss.
    sLoss = styleLoss(transferStyleFeatures,styleFeatures,params.styleFeatureLayerWeights);

    % Compute final loss as weighted combination of content and style loss.
    loss = (params.alpha * cLoss) + (params.beta * sLoss);

    % Calculate gradient with respect to transfer image.
    gradients = dlgradient(loss,dlTransfer);

    % Extract various losses.
```

```
    losses.totalLoss = gather(extractdata(loss));
    losses.contentLoss = gather(extractdata(cLoss));
    losses.styleLoss = gather(extractdata(sLoss));

end
```

Compute Content Loss

The `contentLoss` helper function computes the weighted mean squared difference between the content image features and the transfer image features.

```
function loss = contentLoss(transferContentFeatures,contentFeatures,contentWeights)

    loss = 0;
    for i=1:numel(contentFeatures)
        temp = 0.5 .* mean((transferContentFeatures{1,i} - contentFeatures{1,i}).^2,'all');
        loss = loss + (contentWeights(i)*temp);
    end
end
```

Compute Style Loss

The `styleLoss` helper function computes the weighted mean squared difference between the Gram matrix of the style image features and the Gram matrix of the transfer image features.

```
function loss = styleLoss(transferStyleFeatures,styleFeatures,styleWeights)

    loss = 0;
    for i=1:numel(styleFeatures)

        tsf = transferStyleFeatures{1,i};
        sf = styleFeatures{1,i};
        [h,w,c] = size(sf);

        gramStyle = computeGramMatrix(sf);
        gramTransfer = computeGramMatrix(tsf);
        sLoss = mean((gramTransfer - gramStyle).^2,'all') / ((h*w*c)^2);

        loss = loss + (styleWeights(i)*sLoss);
    end
end
```

Compute Gram Matrix

The `computeGramMatrix` helper function is used by the `styleLoss` helper function to compute the Gram matrix of a feature map.

```
function gramMatrix = computeGramMatrix(featureMap)
    [H,W,C] = size(featureMap);
    reshapedFeatures = reshape(featureMap,H*W,C);
    gramMatrix = reshapedFeatures' * reshapedFeatures;
end
```

References

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style." Preprint, submitted September 2, 2015. https://arxiv.org/abs/1508.06576

See Also

`vgg19` | `trainNetwork` | `trainingOptions` | `dlarray`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “List of Functions with dlarray Support” on page 18-423
- “List of Deep Learning Layers” on page 1-21

Unsupervised Medical Image Denoising Using CycleGAN

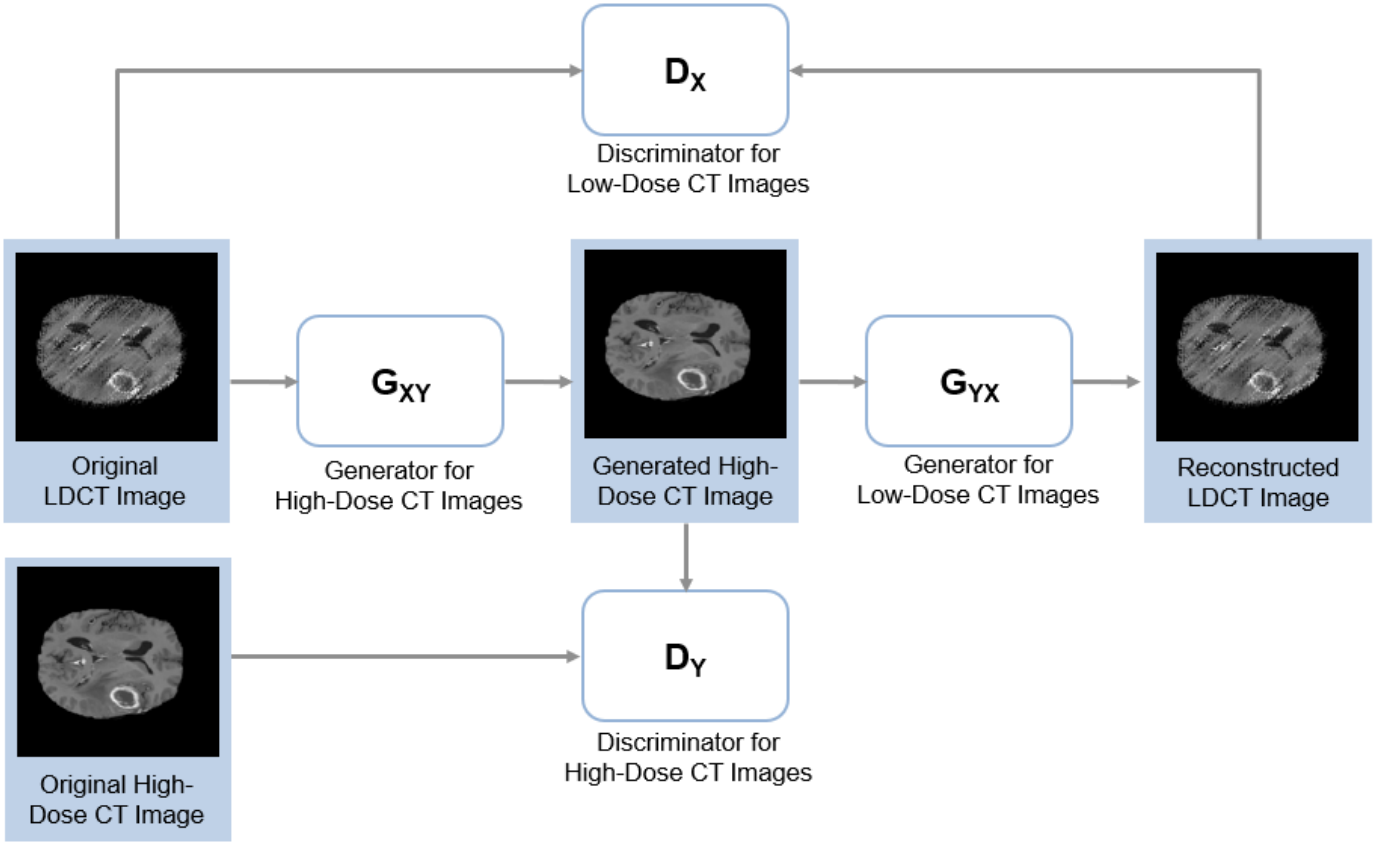
This example shows how to generate high-quality high-dose computed tomography (CT) images from noisy low-dose CT images using a CycleGAN neural network.

This example uses a cycle-consistent generative adversarial network (CycleGAN) trained on patches of image data from a large sample of data. For a similar approach using a UNIT neural network trained on full images from a limited sample of data, see “Unsupervised Medical Image Denoising Using CycleGAN” (Image Processing Toolbox).

X-ray CT is a popular imaging modality used in clinical and industrial applications because it produces high-quality images and offers superior diagnostic capabilities. To protect the safety of patients, clinicians recommend a low radiation dose. However, a low radiation dose results in a lower signal-to-noise ratio (SNR) in the images, and therefore reduces the diagnostic accuracy.

Deep learning techniques offer solutions to improve the image quality for low-dose CT (LDCT) images. Using a generative adversarial network (GAN) for image-to-image translation, you can convert noisy LDCT images to images of the same quality as regular-dose CT images. For this application, the source domain consists of LDCT images and the target domain consists of regular-dose images.

CT image denoising requires a GAN that performs unsupervised training because clinicians do not typically acquire matching pairs of low-dose and regular-dose CT images of the same patient in the same session. This example uses a CycleGAN architecture that supports unsupervised training. For more information, see “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox).



Download LDCT Data Set

This example uses data from the Low Dose CT Grand Challenge [2, 3, 4]. The data includes pairs of regular-dose CT images and simulated low-dose CT images for 99 head scans (labeled N for neuro), 100 chest scans (labeled C for chest), and 100 abdomen scans (labeled L for liver). The size of the data set is 1.2 TB.

Set `dataDir` as the desired location of the data set.

```
dataDir = fullfile(tempdir,"LDCT","LDCT-and-Projection-data");
```

To download the data, go to the Cancer Imaging Archive website. This example uses only images from the chest. Download the chest files from the "Images (DICOM, 952 GB)" data set into the directory specified by `dataDir` using the NBIA Data Retriever. When the download is successful, `dataDir` contains 50 subfolders with names such as "C002" and "C004", ending with "C296".

Create Datastores for Training, Validation, and Testing

The LDCT data set provides pairs of low-dose and high-dose CT images. However, the CycleGAN architecture requires unpaired data for unsupervised learning. This example simulates unpaired training and validation data by partitioning images such that the patients used to obtain low-dose CT and high-dose CT images do not overlap. The example retains pairs of low-dose and regular-dose images for testing.

Split the data into training, validation, and test data sets using the `createLDCTFolderList` helper function. This function is attached to the example as a supporting file. The helper function splits the

data such that there is roughly good representation of the two types of images in each group. Approximately 80% of the data is used for training, 15% is used for testing, and 5% is used for validation.

```
maxDirsForABodyPart = 25;
[filesTrainLD,filesTrainHD,filesTestLD,filesTestHD,filesValLD,filesValHD] = ...
    createLDCTFolderList(dataDir,maxDirsForABodyPart);
```

Create image datastores that contain training and validation images for both domains, namely low-dose CT images and high-dose CT images. The data set consists of DICOM images, so use the custom `ReadFcn` name-value argument in `imageDatastore` to enable reading the data.

```
exts = {'.dcm'};
readFcn = @(x)dicomread(x);
imdsTrainLD = imageDatastore(filesTrainLD,FileExtensions=exts,ReadFcn=readFcn);
imdsTrainHD = imageDatastore(filesTrainHD,FileExtensions=exts,ReadFcn=readFcn);
imdsValLD = imageDatastore(filesValLD,FileExtensions=exts,ReadFcn=readFcn);
imdsValHD = imageDatastore(filesValHD,FileExtensions=exts,ReadFcn=readFcn);
imdsTestLD = imageDatastore(filesTestLD,FileExtensions=exts,ReadFcn=readFcn);
imdsTestHD = imageDatastore(filesTestHD,FileExtensions=exts,ReadFcn=readFcn);
```

The number of low-dose and high-dose images can differ. Select a subset of the files such that the number of images is equal.

```
numTrain = min(numel(imdsTrainLD.Files),numel(imdsTrainHD.Files));
imdsTrainLD = subset(imdsTrainLD,1:numTrain);
imdsTrainHD = subset(imdsTrainHD,1:numTrain);

numVal = min(numel(imdsValLD.Files),numel(imdsValHD.Files));
imdsValLD = subset(imdsValLD,1:numVal);
imdsValHD = subset(imdsValHD,1:numVal);

numTest = min(numel(imdsTestLD.Files),numel(imdsTestHD.Files));
imdsTestLD = subset(imdsTestLD,1:numTest);
imdsTestHD = subset(imdsTestHD,1:numTest);
```

Preprocess and Augment Data

Preprocess the data by using the `transform` function with custom preprocessing operations specified by the `normalizeCTImages` helper function. This function is attached to the example as a supporting file. The `normalizeCTImages` function rescales the data to the range [-1, 1].

```
timdsTrainLD = transform(imdsTrainLD,@(x){normalizeCTImages(x)});
timdsTrainHD = transform(imdsTrainHD,@(x){normalizeCTImages(x)});
timdsValLD = transform(imdsValLD,@(x){normalizeCTImages(x)});
timdsValHD  = transform(imdsValHD,@(x){normalizeCTImages(x)});
timdsTestLD = transform(imdsTestLD,@(x){normalizeCTImages(x)});
timdsTestHD  = transform(imdsTestHD,@(x){normalizeCTImages(x)});
```

Combine the low-dose and high-dose training data by using a `randomPatchExtractionDatastore` (Image Processing Toolbox). When reading from this datastore, augment the data using random rotation and horizontal reflection.

```
inputSize = [128,128,1];
augmenter = imageDataAugmenter(RandRotation=@()90*(randi([0,1],1)),RandXReflection=true);
dsTrain = randomPatchExtractionDatastore(timdsTrainLD,timdsTrainHD, ...
    inputSize(1:2),PatchesPerImage=16,DataAugmentation=augmenter);
```

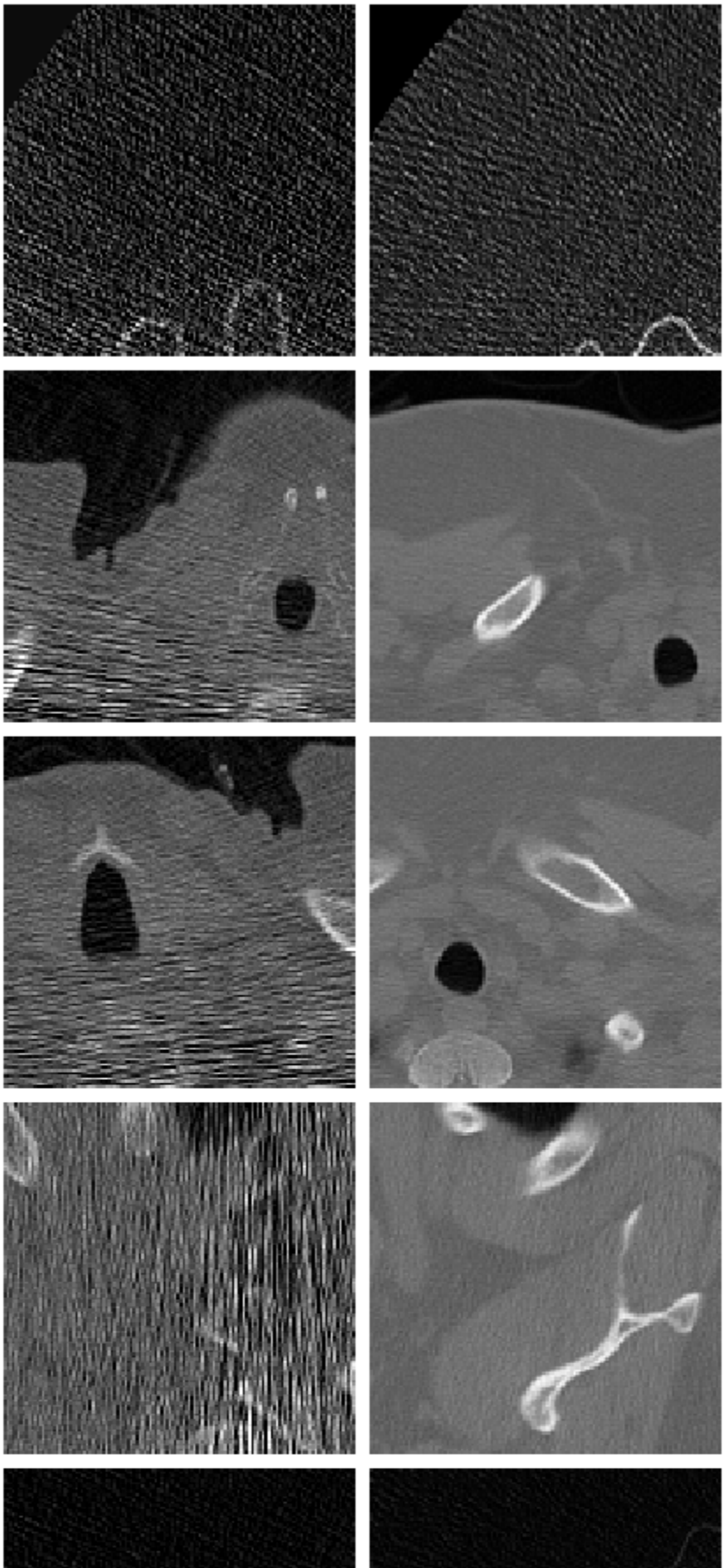
Combine the validation data by using a `randomPatchExtractionDatastore`. You do not need to perform augmentation when reading validation data.

```
dsVal = randomPatchExtractionDatastore(timdsValLD,timdsValHD,inputSize(1:2));
```

Visualize Data Set

Look at a few low-dose and high-dose image patch pairs from the training set. Notice that the image pairs of low-dose (left) and high-dose (right) images are unpaired, as they are from different patients.

```
numImagePairs = 6;
imagePairsTrain = [];
for i = 1:numImagePairs
    imLowAndHighDose = read(dsTrain);
    inputImage = imLowAndHighDose.InputImage{1};
    inputImage = rescale(im2single(inputImage));
    responseImage = imLowAndHighDose.ResponseImage{1};
    responseImage = rescale(im2single(responseImage));
    imagePairsTrain = cat(4,imagePairsTrain,inputImage,responseImage);
end
montage(imagePairsTrain,Size=[numImagePairs 2],BorderSize=4,BackgroundColor="w")
```



Batch Training and Validation Data During Training

This example uses a custom training loop. The `minibatchqueue` object is useful for managing the mini-batching of observations in custom training loops. The `minibatchqueue` object also casts data to a `dlarray` object that enables auto differentiation in deep learning applications.

Process the mini-batches by concatenating image patches along the batch dimension using the helper function `concatenateMiniBatchLD2HDCT`. This function is attached to the example as a supporting file. Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch). Discard any partial mini-batches with less than `miniBatchSize` observations.

```
miniBatchSize = 32;

mbqTrain = minibatchqueue(dsTrain, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@concatenateMiniBatchLD2HDCT, ...
    PartialMiniBatch="discard", ...
    MiniBatchFormat="SSCB");
mbqVal = minibatchqueue(dsVal, ...
    MiniBatchSize=miniBatchSize, ...
    MiniBatchFcn=@concatenateMiniBatchLD2HDCT, ...
    PartialMiniBatch="discard", ...
    MiniBatchFormat="SSCB");
```

Create Generator and Discriminator Networks

The CycleGAN consists of two generators and two discriminators. The generators perform image-to-image translation from low-dose to high-dose and vice versa. The discriminators are PatchGAN networks that return the patch-wise probability that the input data is real or generated. One discriminator distinguishes between the real and generated low-dose images and the other discriminator distinguishes between real and generated high-dose images.

Create each generator network using the `cycleGANGenerator` (Image Processing Toolbox) function. For an input size of 256-by-256 pixels, specify the `NumResidualBlocks` argument as `9`. By default, the function has 3 encoder modules and uses 64 filters in the first convolutional layer.

```
numResiduals = 6;
genHD2LD = cycleGANGenerator(inputSize,NumResidualBlocks=numResiduals,NumOutputChannels=1);
genLD2HD = cycleGANGenerator(inputSize,NumResidualBlocks=numResiduals,NumOutputChannels=1);
```

Create each discriminator network using the `patchGANDiscriminator` (Image Processing Toolbox) function. Use the default settings for the number of downsampling blocks and number of filters in the first convolutional layer in the discriminators.

```
discLD = patchGANDiscriminator(inputSize);
discHD = patchGANDiscriminator(inputSize);
```

Define Loss Functions and Scores

The `modelGradients` helper function calculates the gradients and losses for the discriminators and generators. This function is defined in the Supporting Functions on page 9-0 section of this example.

The objective of the generator is to generate translated images that the discriminators classify as real. The generator loss is a weighted sum of three types of losses: adversarial loss, cycle consistency loss, and fidelity loss. Fidelity loss is based on structural similarity (SSIM) loss.

$$L_{\text{Total}} = L_{\text{Adversarial}} + \lambda * L_{\text{Cycle consistency}} + L_{\text{Fidelity}}$$

Specify the weighting factor λ that controls the relative significance of the cycle consistency loss with the adversarial and fidelity losses.

```
lambda = 10;
```

The objective of each discriminator is to correctly distinguish between real images (1) and translated images (0) for images in its domain. Each discriminator has a single loss function that relies on the mean squared error (MSE) between the expected and predicted output.

Specify Training Options

Train with a mini-batch size of 32 for 3 epochs.

```
numEpochs = 3;
miniBatchSize = 32;
```

Specify the options for Adam optimization. For both generator and discriminator networks, use:

- A learning rate of 0.0002
- A gradient decay factor of 0.5
- A squared gradient decay factor of 0.999

```
learnRate = 0.0002;
gradientDecay = 0.5;
squaredGradientDecayFactor = 0.999;
```

Initialize Adam parameters for the generators and discriminators.

```
avgGradGenLD2HD = [];
avgSqGradGenLD2HD = [];
avgGradGenHD2LD = [];
avgSqGradGenHD2LD = [];
avgGradDiscLD = [];
avgSqGradDiscLD = [];
avgGradDiscHD = [];
avgSqGradDiscHD = [];
```

Display the generated validation images every 100 iterations.

```
validationFrequency = 100;
```

Train or Download Model

By default, the example downloads a pretrained version of the CycleGAN generator for low-dose to high-dose CT by using the helper function `downloadTrainedLD2HDCTCycleGANNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradients` helper function.

- Update the network parameters using the `adamupdate` function.
- Display the input and translated images for both the source and target domains after each epoch.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see "GPU Support by Release" (Parallel Computing Toolbox). Training takes about 30 hours on an NVIDIA™ Titan X with 24 GB of GPU memory.

```
doTraining = false;
if doTraining

    iteration = 0;
    start = tic;

    % Create a directory to store checkpoints
    checkpointDir = fullfile(dataDir,"checkpoints");
    if ~exist(checkpointDir,"dir")
        mkdir(checkpointDir);
    end

    % Initialize plots for training progress
    [figureHandle,tileHandle,imageAxes,scoreAxesX,scoreAxesY, ...
        lineScoreGenLD2HD,lineScoreGenD2LD, ...
        lineScoreDiscHD,lineScoreDiscLD] = initializeTrainingPlotLD2HDCT;

    for epoch = 1:numEpochs

        shuffle(mbqTrain);

        % Loop over mini-batches
        while hasdata(mbqTrain)
            iteration = iteration + 1;

            % Read mini-batch of data
            [imageLD,imageHD] = next(mbqTrain);

            % Convert mini-batch of data to dlarray and specify the dimension labels
            % "SSCB" (spatial, spatial, channel, batch)
            imageLD = dlarray(imageLD,"SSCB");
            imageHD = dlarray(imageHD,"SSCB");

            % If training on a GPU, then convert data to gpuArray
            if canUseGPU
                imageLD = gpuArray(imageLD);
                imageHD = gpuArray(imageHD);
            end

            % Calculate the loss and gradients
            [genHD2LDGrad,genLD2HDGrad,discrXGrad,discYGrad, ...
                genHD2LDState,genLD2HDState,scores,imagesOutLD2HD,imagesOutHD2LD] = ...
                dlfeval(@modelGradients,genLD2HD,genHD2LD, ...
                discLD,discHD,imageHD,imageLD,lambda);
            genHD2LD.State = genHD2LDState;
            genLD2HD.State = genLD2HDState;

            % Update parameters of discLD, which distinguishes
            % the generated low-dose CT images from real low-dose CT images
            [discLD.Learnables,avgGradDiscLD,avgSqGradDiscLD] = ...
```

```
                    adamupdate(discLD.Learnables,discrXGrad,avgGradDiscLD, ...
                    avgSqGradDiscLD,iteration,learnRate,gradientDecay,squaredGradientDecayFactor);

                % Update parameters of discHD, which distinguishes
                % the generated high-dose CT images from real high-dose CT images
                [discHD.Learnables,avgGradDiscHD,avgSqGradDiscHD] = ...
                    adamupdate(discHD.Learnables,discYGrad,avgGradDiscHD, ...
                    avgSqGradDiscHD,iteration,learnRate,gradientDecay,squaredGradientDecayFactor);

                % Update parameters of genHD2LD, which
                % generates low-dose CT images from high-dose CT images
                [genHD2LD.Learnables,avgGradGenHD2LD,avgSqGradGenHD2LD] = ...
                    adamupdate(genHD2LD.Learnables,genHD2LDGrad,avgGradGenHD2LD, ...
                    avgSqGradGenHD2LD,iteration,learnRate,gradientDecay,squaredGradientDecayFactor);

                % Update parameters of genLD2HD, which
                % generates high-dose CT images from low-dose CT images
                [genLD2HD.Learnables,avgGradGenLD2HD,avgSqGradGenLD2HD] = ...
                    adamupdate(genLD2HD.Learnables,genLD2HDGrad,avgGradGenLD2HD, ...
                    avgSqGradGenLD2HD,iteration,learnRate,gradientDecay,squaredGradientDecayFactor);

                % Update the plots of network scores
                updateTrainingPlotLD2HDCT(scores,iteration,epoch,start,scoreAxesX,scoreAxesY,...
                    lineScoreGenLD2HD,lineScoreGenD2LD, ...
                    lineScoreDiscHD,lineScoreDiscLD)

                %  Every validationFrequency iterations, display a batch of
                %  generated images using the held-out generator input
                if mod(iteration,validationFrequency) == 0 || iteration == 1
                     displayGeneratedLD2HDCTImages(mbqVal,imageAxes,genLD2HD,genHD2LD);
                end
            end

            % Save the model after each epoch
            if canUseGPU
                [genLD2HD,genHD2LD,discLD,discHD] = ...
                    gather(genLD2HD,genHD2LD,discLD,discHD);
            end
            generatorHighDoseToLowDose = genHD2LD;
            generatorLowDoseToHighDose = genLD2HD;
            discriminatorLowDose = discLD;
            discriminatorHighDose = discHD;
            modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
            save(checkpointDir+filesep+"LD2HDCTCycleGAN-"+modelDateTime+"-Epoch-"+epoch+".mat", ...
                'generatorLowDoseToHighDose','generatorHighDoseToLowDose', ...
                'discriminatorLowDose','discriminatorHighDose');
        end

        % Save the final model
        modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
        save(checkpointDir+filesep+"trainedLD2HDCTCycleGANNet-"+modelDateTime+".mat", ...
            'generatorLowDoseToHighDose','generatorHighDoseToLowDose', ...
            'discriminatorLowDose','discriminatorHighDose');

    else
        trainedCycleGANNetURL = "https://www.mathworks.com/supportfiles/vision/data/trainedLD2HDCTCy
        netDir = fullfile(tempdir,"LD2HDCT");
        downloadTrainedLD2HDCTCycleGANNet(trainedCycleGANNetURL,netDir);
```

```
    load(fullfile(netDir,"trainedLD2HDCTCycleGANNet.mat"));
end
```

Generate New Images Using Test Data

Define the number of test images to use for calculating quality metrics. Randomly select two test images to display.

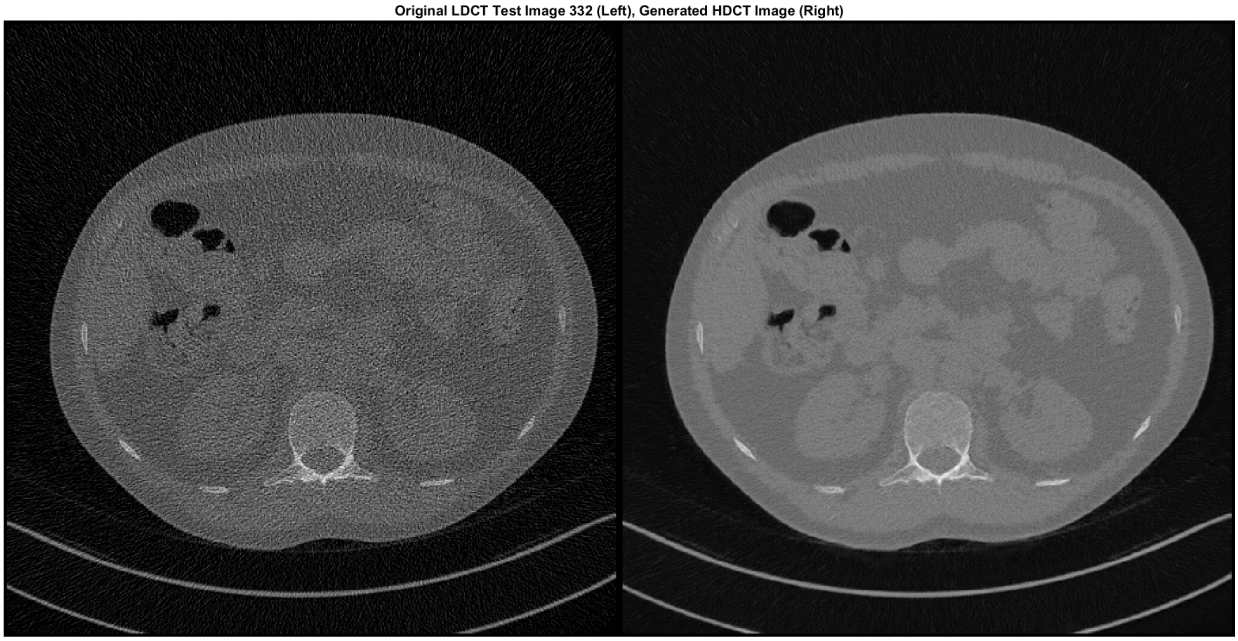
```
numTest = timdsTestLD.numpartitions;
numImagesToDisplay = 2;
idxImagesToDisplay = randi(numTest,1,numImagesToDisplay);
```

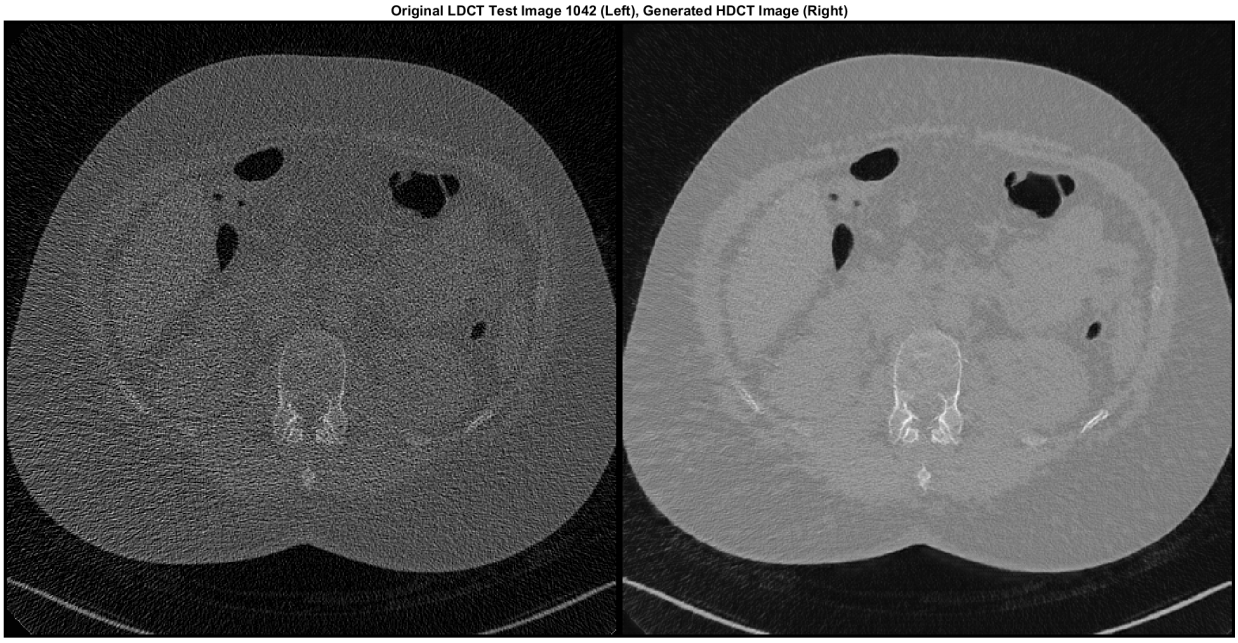
Initialize variables to calculate PSNR and SSIM.

```
origPSNR = zeros(numTest,1);
generatedPSNR = zeros(numTest,1);
origSSIM = zeros(numTest,1);
generatedSSIM = zeros(numTest,1);
```

To generate new translated images, use the `predict` function. Read images from the test data set and use the trained generators to generate new images.

```
for idx = 1:numTest
    imageTestLD = read(timdsTestLD);
    imageTestHD = read(timdsTestHD);

    imageTestLD = cat(4,imageTestLD{1});
    imageTestHD = cat(4,imageTestHD{1});

    % Convert mini-batch of data to dlarray and specify the dimension labels
    % 'SSCB' (spatial, spatial, channel, batch)
    imageTestLD = dlarray(imageTestLD,'SSCB');
    imageTestHD = dlarray(imageTestHD,'SSCB');

    % If running on a GPU, then convert data to gpuArray
    if canUseGPU
        imageTestLD = gpuArray(imageTestLD);
        imageTestHD = gpuArray(imageTestHD);
    end

    % Generate translated images
    generatedImageHD = predict(generatorLowDoseToHighDose,imageTestLD);
    generatedImageLD = predict(generatorHighDoseToLowDose,imageTestHD);

    % Display a few images to visualize the network responses
     if ismember(idx,idxImagesToDisplay)
         figure
         origImLD = rescale(extractdata(imageTestLD));
         genImHD = rescale(extractdata(generatedImageHD));
         montage({origImLD,genImHD},Size=[1 2],BorderSize=5)
         title("Original LDCT Test Image "+idx+" (Left), Generated HDCT Image (Right)")
    end

    origPSNR(idx) = psnr(imageTestLD,imageTestHD);
    generatedPSNR(idx) = psnr(generatedImageHD,imageTestHD);

    origSSIM(idx) = multissim(imageTestLD,imageTestHD);
    generatedSSIM(idx) = multissim(generatedImageHD,imageTestHD);
end
```





Calculate the average PSNR of the original and generated images. A larger PSNR value indicates better image quality.

```
disp("Average PSNR of original images: "+mean(origPSNR,"all"));
```

```
Average PSNR of original images: 20.4045
```

```
disp("Average PSNR of generated images: "+mean(generatedPSNR,"all"));
```

```
Average PSNR of generated images: 27.9155
```

Calculate the average SSIM of the original and generated images. An SSIM value closer to 1 indicates better image quality.

```
disp("Average SSIM of original images: "+mean(origSSIM,"all"));
```

```
Average SSIM of original images: 0.76651
```

```
disp("Average SSIM of generated images: "+mean(generatedSSIM,"all"));
```

```
Average SSIM of generated images: 0.90194
```

Supporting Functions

Model Gradients Function

The function `modelGradients` takes as input the two generator and discriminator `dlnetwork` objects and a mini-batch of input data. The function returns the gradients of the loss with respect to the learnable parameters in the networks and the scores of the four networks. Because the discriminator outputs are not in the range [0, 1], the `modelGradients` function applies the sigmoid function to convert discriminator outputs into probability scores.

```
function [genHD2LDGrad,genLD2HDGrad,discLDGrad,discHDGrad, ...
    genHD2LDState,genLD2HDState,scores,imagesOutLDAndHDGenerated,imagesOutHDAndLDGenerated] = ..
    modelGradients(genLD2HD,genHD2LD,discLD,discHD,imageHD,imageLD,lambda)

% Translate images from one domain to another: low-dose to high-dose and
% vice versa
[imageLDGenerated,genHD2LDState] = forward(genHD2LD,imageHD);
[imageHDGenerated,genLD2HDState] = forward(genLD2HD,imageLD);

% Calculate predictions for real images in each domain by the corresponding
% discriminator networks
predRealLD = forward(discLD,imageLD);
predRealHD = forward(discHD,imageHD);

% Calculate predictions for generated images in each domain by the
% corresponding discriminator networks
predGeneratedLD = forward(discLD,imageLDGenerated);
predGeneratedHD = forward(discHD,imageHDGenerated);

% Calculate discriminator losses for real images
discLDLossReal = lossReal(predRealLD);
discHDLossReal = lossReal(predRealHD);

% Calculate discriminator losses for generated images
discLDLossGenerated = lossGenerated(predGeneratedLD);
discHDLossGenerated = lossGenerated(predGeneratedHD);

% Calculate total discriminator loss for each discriminator network
discLDLossTotal = 0.5*(discLDLossReal + discLDLossGenerated);
discHDLossTotal = 0.5*(discHDLossReal + discHDLossGenerated);

% Calculate generator loss for generated images
genLossHD2LD = lossReal(predGeneratedLD);
```

```
genLossLD2HD = lossReal(predGeneratedHD);

% Complete the round-trip (cycle consistency) outputs by applying the
% generator to each generated image to get the images in the corresponding
% original domains
cycleImageLD2HD2LD = forward(genHD2LD,imageHDGenerated);
cycleImageHD2LD2HD = forward(genLD2HD,imageLDGenerated);

% Calculate cycle consistency loss between real and generated images
cycleLossLD2HD2LD = cycleConsistencyLoss(imageLD,cycleImageLD2HD2LD,lambda);
cycleLossHD2LD2HD = cycleConsistencyLoss(imageHD,cycleImageHD2LD2HD,lambda);

% Calculate identity outputs
identityImageLD = forward(genHD2LD,imageLD);
identityImageHD = forward(genLD2HD,imageHD);

% Calculate fidelity loss (SSIM) between the identity outputs
fidelityLossLD = mean(1-multissim(identityImageLD,imageLD),"all");
fidelityLossHD = mean(1-multissim(identityImageHD,imageHD),"all");

% Calculate total generator loss
genLossTotal = genLossHD2LD + cycleLossHD2LD2HD + ...
    genLossLD2HD + cycleLossLD2HD2LD + fidelityLossLD + fidelityLossHD;

% Calculate scores of generators
genHD2LDScore = mean(sigmoid(predGeneratedLD),"all");
genLD2HDScore = mean(sigmoid(predGeneratedHD),"all");

% Calculate scores of discriminators
discLDScore = 0.5*mean(sigmoid(predRealLD),"all") + ...
    0.5*mean(1-sigmoid(predGeneratedLD),"all");
discHDScore = 0.5*mean(sigmoid(predRealHD),"all") + ...
    0.5*mean(1-sigmoid(predGeneratedHD),"all");

% Combine scores into cell array
scores = {genHD2LDScore,genLD2HDScore,discLDScore,discHDScore};

% Calculate gradients of generators
genLD2HDGrad = dlgradient(genLossTotal,genLD2HD.Learnables,'RetainData',true);
genHD2LDGrad = dlgradient(genLossTotal,genHD2LD.Learnables,'RetainData',true);

% Calculate gradients of discriminators
discLDGrad = dlgradient(discLDLossTotal,discLD.Learnables,'RetainData',true);
discHDGrad = dlgradient(discHDLossTotal,discHD.Learnables);

% Return mini-batch of images transforming low-dose CT into high-dose CT
imagesOutLDAndHDGenerated = {imageLD,imageHDGenerated};

% Return mini-batch of images transforming high-dose CT into low-dose CT
imagesOutHDAndLDGenerated = {imageHD,imageLDGenerated};
end
```

Loss Functions

Specify MSE loss functions for real and generated images.

```
function loss = lossReal(predictions)
    loss = mean((1-predictions).^2,"all");
```

```
end

function loss = lossGenerated(predictions)
    loss = mean((predictions).^2,"all");
end
```

Specify cycle consistency loss functions for real and generated images.

```
function loss = cycleConsistencyLoss(imageReal,imageGenerated,lambda)
    loss = mean(abs(imageReal-imageGenerated),"all") * lambda;
end
```

References

[1] Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A. Efros. “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks.” In *2017 IEEE International Conference on Computer Vision (ICCV)*, 2242–51. Venice: IEEE, 2017. https://doi.org/10.1109/ICCV.2017.244.

[2] McCollough, Cynthia, Baiyu Chen, David R Holmes III, Xinhui Duan, Zhicong Yu, Lifeng Yu, Shuai Leng, and Joel Fletcher. “Low Dose CT Image and Projection Data (LDCT-and-Projection-Data).” The Cancer Imaging Archive, 2020. https://doi.org/10.7937/9NPB-2637.

[3] Grants EB017095 and EB017185 (Cynthia McCollough, PI) from the National Institute of Biomedical Imaging and Bioengineering.

[4] Clark, Kenneth, Bruce Vendt, Kirk Smith, John Freymann, Justin Kirby, Paul Koppel, Stephen Moore, et al. “The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository.” *Journal of Digital Imaging* 26, no. 6 (December 2013): 1045–57. https://doi.org/10.1007/s10278-013-9622-7.

[5] You, Chenyu, Qingsong Yang, Hongming Shan, Lars Gjesteby, Guang Li, Shenghong Ju, Zhuiyang Zhang, et al. “Structurally-Sensitive Multi-Scale Deep Neural Network for Low-Dose CT Denoising.” *IEEE Access* 6 (2018): 41839–55. https://doi.org/10.1109/ACCESS.2018.2858196.

See Also

`cycleGANGenerator` | `patchGANDiscriminator` | `transform` | `combine` | `minibatchqueue` | `dlarray` | `dlfeval` | `adamupdate`

Related Examples

- “Unsupervised Medical Image Denoising Using UNIT” on page 9-144

More About

- “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

Unsupervised Medical Image Denoising Using UNIT

This example shows how to generate high-quality computed tomography (CT) images from noisy low-dose CT images using a UNIT neural network.

This example uses an unsupervised image-to-image translation (UNIT) neural network trained on full images from a limited sample of data. For a similar approach using a CycleGAN neural network trained on patches of image data from a large sample of data, see “Unsupervised Medical Image Denoising Using CycleGAN” (Image Processing Toolbox).

X-ray CT is a popular imaging modality used in clinical and industrial applications because it produces high-quality images and offers superior diagnostic capabilities. To protect the safety of patients, clinicians recommend a low radiation dose. However, a low radiation dose results in a lower signal-to-noise ratio (SNR) in the images, and therefore reduces the diagnostic accuracy.

Deep learning techniques offer solutions to improve the image quality for low-dose CT (LDCT) images. Using a generative adversarial network (GAN) for image-to-image translation, you can convert noisy LDCT images to images of the same quality as regular-dose CT images. For this application, the source domain consists of LDCT images and the target domain consists of regular-dose images.

CT image denoising requires a GAN that performs unsupervised training because clinicians do not typically acquire matching pairs of low-dose and regular-dose CT images of the same patient in the same session. This example uses a UNIT architecture that supports unsupervised training. For more information, see “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox).

Download LDCT Data Set

This example uses data from the Low Dose CT Grand Challenge [2, 3, 4]. The data includes pairs of regular-dose CT images and simulated low-dose CT images for 99 head scans (labeled N for neuro), 100 chest scans (labeled C for chest), and 100 abdomen scans (labeled L for liver).

Set `dataDir` as the desired location of the data set. The data for this example requires 52 GB of memory.

```
dataDir = fullfile(tempdir,"LDCT","LDCT-and-Projection-data");
```

To download the data, go to the Cancer Imaging Archive website. This example uses ony two patient scans from the chest. Download the chest files "C081" and "C120" from the "Images (DICOM, 952 GB)" data set using the NBIA Data Retriever. Specify the `dataFolder` variable as the location of the downloaded data. When the download is successful, `dataFolder` contains two subfolders named "C081" and "C120".

Create Datastores for Training, Validation, and Testing

Specify the patient scans that are the source of each data set.

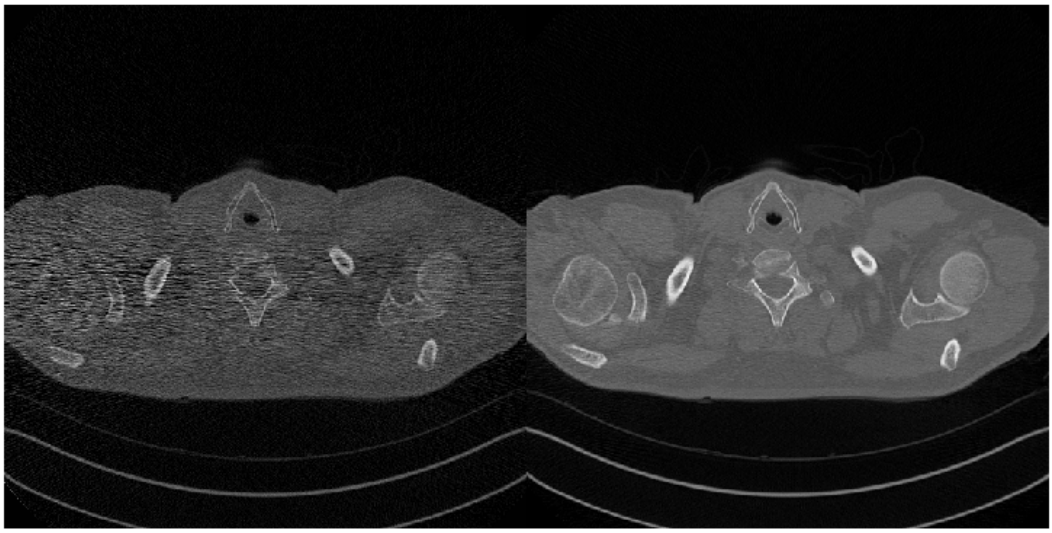
```
scanDirTrain = fullfile(dataDir,"C120","08-30-2018-97899");
scanDirTest = fullfile(dataDir,"C081","08-29-2018-10762");
```

Create `imageDatastore` objects that manage the low-dose and high-dose CT images for training and testing. The data set consists of DICOM images, so use the custom `ReadFcn` name-value argument in `imageDatastore` to enable reading the data.

```
exts = {'.dcm'};
readFcn = @(x)rescale(dicomread(x));
imdsLDTrain = imageDatastore(fullfile(scanDirTrain,"1.000000-Low Dose Images-71581"), ...
    FileExtensions=exts,ReadFcn=readFcn);
imdsHDTrain = imageDatastore(fullfile(scanDirTrain,"1.000000-Full dose images-34601"), ...
    FileExtensions=exts,ReadFcn=readFcn);
imdsLDTest = imageDatastore(fullfile(scanDirTest,"1.000000-Low Dose Images-32837"), ...
    FileExtensions=exts,ReadFcn=readFcn);
imdsHDTest = imageDatastore(fullfile(scanDirTest,"1.000000-Full dose images-95670"), ...
    FileExtensions=exts,ReadFcn=readFcn);
```

Preview a training image from the low-dose and high-dose CT training data sets.

```
lowDose = preview(imdsLDTrain);
highDose = preview(imdsHDTrain);
montage({lowDose,highDose})
```



Preprocess and Augment Training Data

Specify the image input size for the source and target images.

```
inputSize = [256,256,1];
```

Augment and preprocess the training data by using the `transform` function with custom preprocessing operations specified by the `augmentDataForLD2HDCT` helper function. This function is attached to the example as a supporting file.

The `augmentDataForLD2HDCT` function performs these operations:

1. Resize the image to the specified input size using bicubic interpolation.
2. Randomly flip the image in the horizontal direction.

3 Scale the image to the range [-1, 1]. This range matches the range of the final `tanhLayer` used in the generator.

```
imdsLDTrain = transform(imdsLDTrain, @(x)augmentDataForLD2HDCT(x,inputSize));
imdsHDTrain = transform(imdsHDTrain, @(x)augmentDataForLD2HDCT(x,inputSize));
```

The LDCT data set provides pairs of low-dose and high-dose CT images. However, the UNIT architecture requires unpaired data for unsupervised learning. This example simulates unpaired training and validation data by shuffling the data in each iteration of the training loop.

Batch Training and Validation Data During Training

This example uses a custom training loop. The `minibatchqueue` object is useful for managing the mini-batching of observations in custom training loops. The `minibatchqueue` object also casts data to a `dlarray` object that enables auto differentiation in deep learning applications.

Specify the mini-batch data extraction format as `SSCB` (spatial, spatial, channel, batch). Set the `DispatchInBackground` name-value argument as the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
miniBatchSize = 1;
mbqLDTrain = minibatchqueue(imdsLDTrain,MiniBatchSize=miniBatchSize, ...
    MiniBatchFormat="SSCB",DispatchInBackground=canUseGPU);
mbqHDTrain = minibatchqueue(imdsHDTrain,MiniBatchSize=miniBatchSize, ...
    MiniBatchFormat="SSCB",DispatchInBackground=canUseGPU);
```

Create Generator Network

The UNIT consists of one generator and two discriminators. The generator performs image-to-image translation from low dose to high dose. The discriminators are PatchGAN networks that return the patch-wise probability that the input data is real or generated. One discriminator distinguishes between the real and generated low-dose images and the other discriminator distinguishes between real and generated high-dose images.

Create a UNIT generator network using the `unitGenerator` (Image Processing Toolbox) function. The source and target encoder sections of the generator each consist of two downsampling blocks and five residual blocks. The encoder sections share two of the five residual blocks. Likewise, the source and target decoder sections of the generator each consist of two downsampling blocks and five residual blocks, and the decoder sections share two of the five residual blocks.

```
gen = unitGenerator(inputSize);
```

Visualize the generator network.

```
analyzeNetwork(gen)
```

Create Discriminator Networks

There are two discriminator networks, one for each of the image domains (low-dose CT and high-dose CT). Create the discriminators for the source and target domains using the `patchGANDiscriminator` (Image Processing Toolbox) function.

```
discLD = patchGANDiscriminator(inputSize,NumDownsamplingBlocks=4,FilterSize=3, ...
    ConvolutionWeightsInitializer="narrow-normal",NormalizationLayer="none");
discHD = patchGANDiscriminator(inputSize,"NumDownsamplingBlocks",4,FilterSize=3, ...
    ConvolutionWeightsInitializer="narrow-normal",NormalizationLayer="none");
```

Visualize the discriminator networks.

```
analyzeNetwork(discLD);
analyzeNetwork(discHD);
```

Define Model Gradients and Loss Functions

The `modelGradientDisc` and `modelGradientGen` helper functions calculate the gradients and losses for the discriminators and generator, respectively. These functions are defined in the Supporting Functions on page 9-0 section of this example.

The objective of each discriminator is to correctly distinguish between real images (1) and translated images (0) for images in its domain. Each discriminator has a single loss function.

The objective of the generator is to generate translated images that the discriminators classify as real. The generator loss is a weighted sum of five types of losses: self-reconstruction loss, cycle consistency loss, hidden KL loss, cycle hidden KL loss, and adversarial loss.

Specify the weight factors for the various losses.

```
lossWeights.selfReconLossWeight = 10;
lossWeights.hiddenKLLossWeight = 0.01;
lossWeights.cycleConsisLossWeight = 10;
lossWeights.cycleHiddenKLLossWeight = 0.01;
lossWeights.advLossWeight = 1;
lossWeights.discLossWeight = 0.5;
```

Specify Training Options

Specify the options for Adam optimization. Train the network for 26 epochs.

```
numEpochs = 26;
```

Specify identical options for the generator and discriminator networks.

- Specify a learning rate of 0.0001.
- Initialize the trailing average gradient and trailing average gradient-square decay rates with `[]`.
- Use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.
- Use weight decay regularization with a factor of 0.0001.
- Use a mini-batch size of 1 for training.

```
learnRate = 0.0001;
gradDecay = 0.5;
sqGradDecay = 0.999;
weightDecay = 0.0001;

genAvgGradient = [];
genAvgGradientSq = [];
discLDAvgGradient = [];
discLDAvgGradientSq = [];
discHDAvgGradient = [];
discHDAvgGradientSq = [];
```

Train Model or Download Pretrained UNIT Network

By default, the example downloads a pretrained version of the UNIT generator for the NIH-AAPM-Mayo Clinic Low-Dose CT data set by using the helper function

`downloadTrainedLD2HDCTUNITNet`. The helper function is attached to the example as a supporting file. The pretrained network enables you to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the discriminator model gradients using the `dlfeval` function and the `modelGradientDisc` helper function.
- Update the parameters of the discriminator networks using the `adamupdate` function.
- Evaluate the generator model gradients using the `dlfeval` function and the `modelGradientGen` helper function.
- Update the parameters of the generator network using the `adamupdate` function.
- Display the input and translated images for both the source and target domains after each epoch.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 58 hours on an NVIDIA Titan RTX.

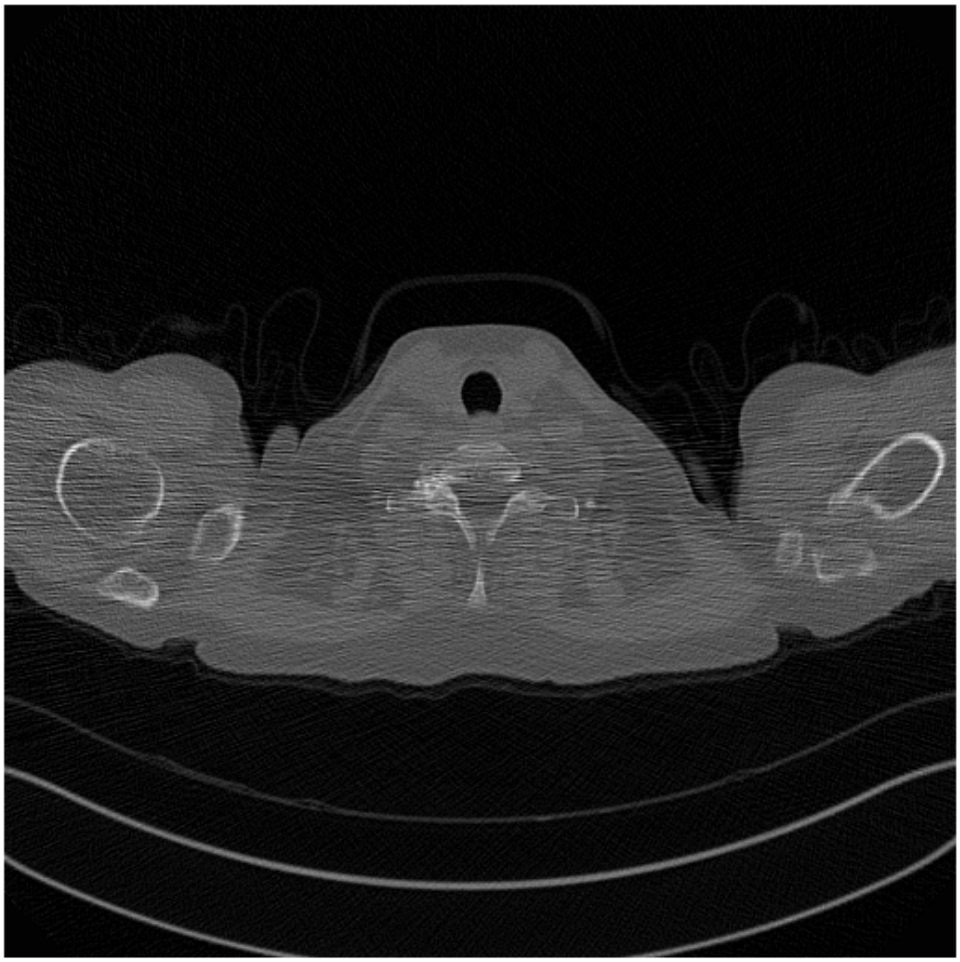
```
doTraining = false;
if doTraining

    % Create a figure to show the results
    figure("Units","Normalized");
    for iPlot = 1:4
        ax(iPlot) = subplot(2,2,iPlot);
    end

    iteration = 0;

    % Loop over epochs
    for epoch = 1:numEpochs

        % Shuffle data every epoch
        reset(mbqLDTrain);
        shuffle(mbqLDTrain);
        reset(mbqHDTrain);
        shuffle(mbqHDTrain);

        % Run the loop until all the images in the mini-batch queue
        % mbqLDTrain are processed
        while hasdata(mbqLDTrain)
            iteration = iteration + 1;

            % Read data from the low-dose domain
            imLowDose = next(mbqLDTrain);

            % Read data from the high-dose domain
            if hasdata(mbqHDTrain) == 0
                reset(mbqHDTrain);
                shuffle(mbqHDTrain);
            end
            imHighDose = next(mbqHDTrain);
```

```
            % Calculate discriminator gradients and losses
            [discLDGrads,discHDGrads,discLDLoss,discHDLoss] = dlfeval(@modelGradientDisc, ...
                gen,discLD,discHD,imLowDose,imHighDose,lossWeights.discLossWeight);

            % Apply weight decay regularization on low-dose discriminator gradients
            discLDGrads = dlupdate(@(g,w) g+weightDecay*w,discLDGrads,discLD.Learnables);

            % Update parameters of low-dose discriminator
            [discLD,discLDAvgGradient,discLDAvgGradientSq] = adamupdate(discLD,discLDGrads, ...
                discLDAvgGradient,discLDAvgGradientSq,iteration,learnRate,gradDecay,sqGradDecay)

            % Apply weight decay regularization on high-dose discriminator gradients
            discHDGrads = dlupdate(@(g,w) g+weightDecay*w,discHDGrads,discHD.Learnables);

            % Update parameters of high-dose discriminator
            [discHD,discHDAvgGradient,discHDAvgGradientSq] = adamupdate(discHD,discHDGrads, ...
                discHDAvgGradient,discHDAvgGradientSq,iteration,learnRate,gradDecay,sqGradDecay)

            % Calculate generator gradient and loss
            [genGrad,genLoss,images] = dlfeval(@modelGradientGen,gen,discLD,discHD,imLowDose,imH

            % Apply weight decay regularization on generator gradients
            genGrad = dlupdate(@(g,w) g+weightDecay*w,genGrad,gen.Learnables);

            % Update parameters of generator
            [gen,genAvgGradient,genAvgGradientSq] = adamupdate(gen,genGrad,genAvgGradient, ...
                genAvgGradientSq,iteration,learnRate,gradDecay,sqGradDecay);
        end

        % Display the results
        updateTrainingPlotLowDoseToHighDose(ax,images{:});
    end

    % Save the trained network
    modelDateTime = string(datetime("now",Format="yyyy-MM-dd-HH-mm-ss"));
    save(strcat("trainedLowDoseHighDoseUNITGeneratorNet-",modelDateTime,"-Epoch-",num2str(numEpo

else
    net_url = "https://ssd.mathworks.com/supportfiles/vision/data/trainedLowDoseHighDoseUNITGene
    downloadTrainedLD2HDCTUNITNet(net_url,dataDir);
    load(fullfile(dataDir,"trainedLowDoseHighDoseUNITGeneratorNet.mat"));
end
```

Generate High-Dose Image Using Trained Network

Read and display an image from the datastore of low-dose test images.

```
idxToTest = 1;
imLowDoseTest = readimage(imdsLDTest,idxToTest);
figure
imshow(imLowDoseTest)
```



Convert the image to data type `single`. Rescale the image data to the range [-1, 1] as expected by the final layer of the generator network.

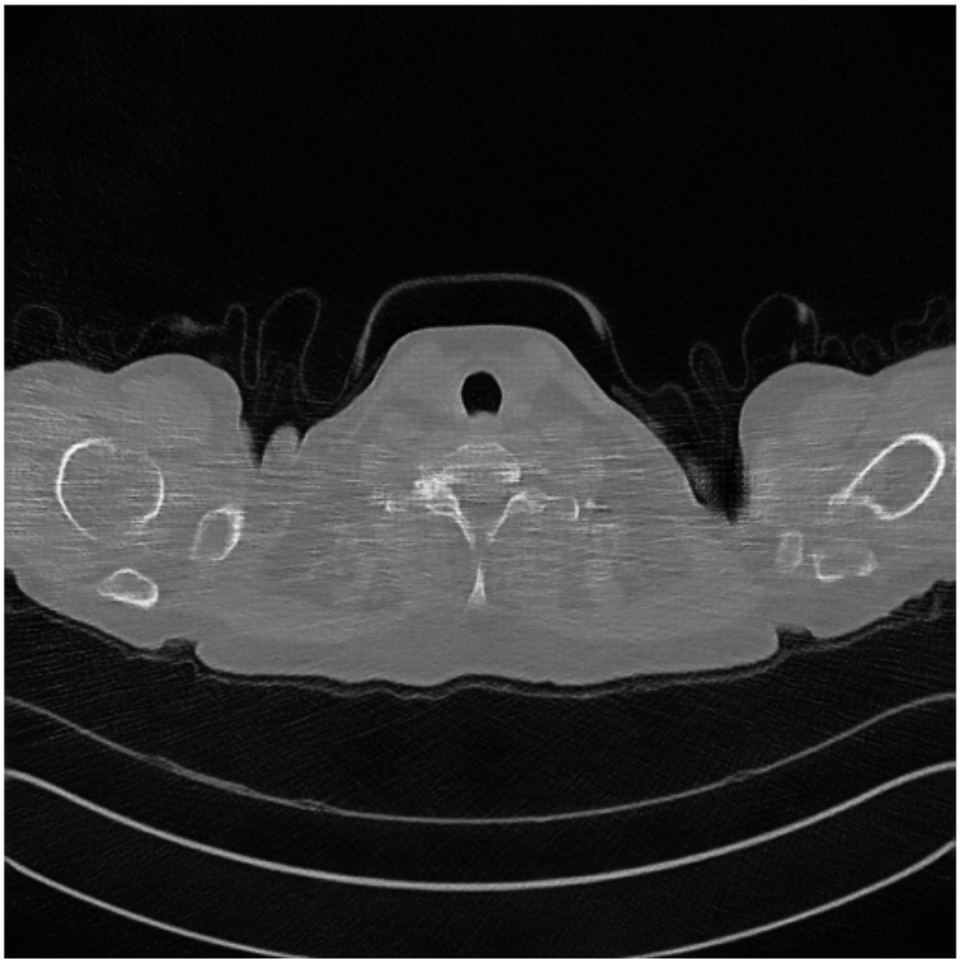
```
imLowDoseTest = im2single(imLowDoseTest);
imLowDoseTestRescaled = (imLowDoseTest-0.5)/0.5;
```

Create a `dlarray` object that inputs data to the generator. If a supported GPU is available for computation, then perform inference on a GPU by converting the data to a `gpuArray` object.

```
dlLowDoseImage = dlarray(imLowDoseTestRescaled,'SSCB');
if canUseGPU
    dlLowDoseImage = gpuArray(dlLowDoseImage);
end
```

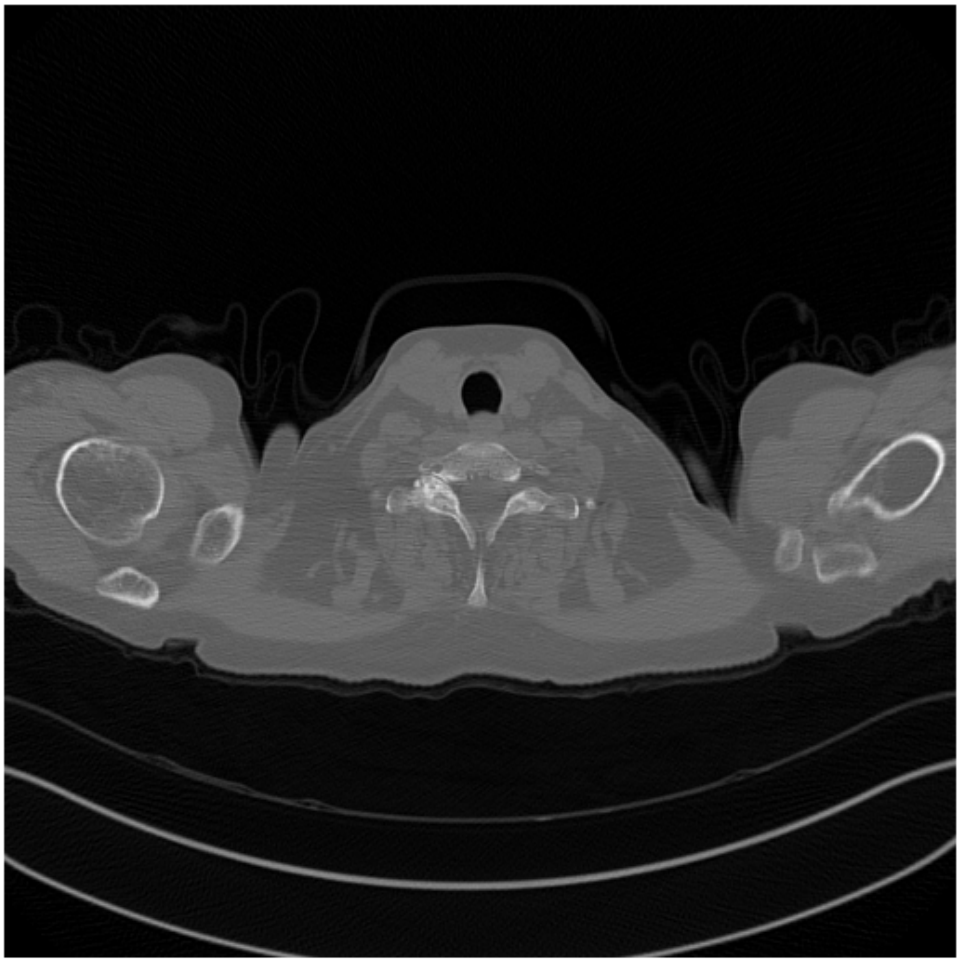
Translate the input low-dose image to the high-dose domain using the `unitPredict` (Image Processing Toolbox) function. The generated image has pixel values in the range [-1, 1]. For display, rescale the activations to the range [0, 1].

```
dlImLowDoseToHighDose = unitPredict(gen,dlLowDoseImage);
imHighDoseGenerated = extractdata(gather(dlImLowDoseToHighDose));
imHighDoseGenerated = rescale(imHighDoseGenerated);
imshow(imHighDoseGenerated)
```



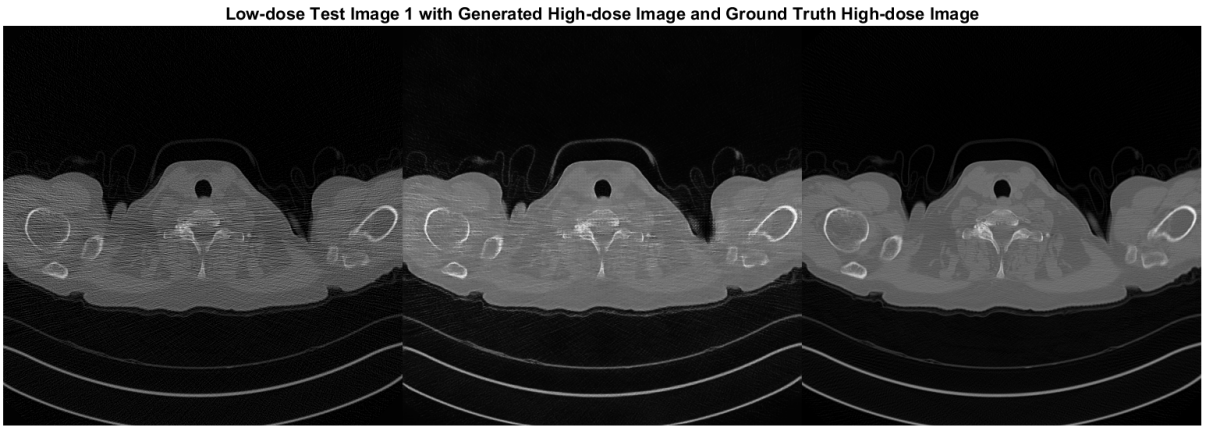
Read and display the ground truth high-dose image. The high-dose and low-dose test datastores are not shuffled, so the ground truth high-dose image corresponds directly to the low-dose test image.

```
imHighDoseGroundTruth = readimage(imdsHDTest,idxToTest);
imshow(imHighDoseGroundTruth)
```



Display the input low-dose CT, the generated high-dose version, and the ground truth high-dose image in a montage. Although the network is trained on data from a single patient scan, the network generalizes well to test images from other patient scans.

```
imshow([imLowDoseTest imHighDoseGenerated imHighDoseGroundTruth])
title(['Low-dose Test Image ',num2str(idxToTest),' with Generated High-dose Image and Ground Tru
```



Supporting Functions

Model Gradients Function

The `modelGradientGen` helper function calculates the gradients and loss for the generator.

```
function [genGrad,genLoss,images] = modelGradientGen(gen,discLD,discHD,imLD,imHD,lossWeights)

    [imLD2LD,imHD2LD,imLD2HD,imHD2HD] = forward(gen,imLD,imHD);
    hidden = forward(gen,imLD,imHD,Outputs="encoderSharedBlock");

    [~,imLD2HD2LD,imHD2LD2HD,~] = forward(gen,imHD2LD,imLD2HD);
    cycle_hidden = forward(gen,imHD2LD,imLD2HD,Outputs="encoderSharedBlock");

    % Calculate different losses
    selfReconLoss = computeReconLoss(imLD,imLD2LD) + computeReconLoss(imHD,imHD2HD);
    hiddenKLLoss = computeKLLoss(hidden);
    cycleReconLoss = computeReconLoss(imLD,imLD2HD2LD) + computeReconLoss(imHD,imHD2LD2HD);
    cycleHiddenKLLoss = computeKLLoss(cycle_hidden);

    outA = forward(discLD,imHD2LD);
    outB = forward(discHD,imLD2HD);
    advLoss = computeAdvLoss(outA) + computeAdvLoss(outB);

    % Calculate the total loss of generator as a weighted sum of five losses
    genTotalLoss = ...
        selfReconLoss*lossWeights.selfReconLossWeight + ...
        hiddenKLLoss*lossWeights.hiddenKLLossWeight + ...
        cycleReconLoss*lossWeights.cycleConsisLossWeight + ...
        cycleHiddenKLLoss*lossWeights.cycleHiddenKLLossWeight + ...
        advLoss*lossWeights.advLossWeight;

    % Update the parameters of generator
    genGrad = dlgradient(genTotalLoss,gen.Learnables);

    % Convert the data type from dlarray to single
    genLoss = extractdata(genTotalLoss);
    images = {imLD,imLD2HD,imHD,imHD2LD};
end
```

The `modelGradientDisc` helper function calculates the gradients and loss for the two discriminators.

```
function [discLDGrads,discHDGrads,discLDLoss,discHDLoss] = modelGradientDisc(gen, ...
    discLD,discHD,imRealLD,imRealHD,discLossWeight)

    [~,imFakeLD,imFakeHD,~] = forward(gen,imRealLD,imRealHD);

    % Calculate loss of the discriminator for low-dose images
    outRealLD = forward(discLD,imRealLD);
    outFakeLD = forward(discLD,imFakeLD);
    discLDLoss = discLossWeight*computeDiscLoss(outRealLD,outFakeLD);

    % Update parameters of the discriminator for low-dose images
    discLDGrads = dlgradient(discLDLoss,discLD.Learnables);

    % Calculate loss of the discriminator for high-dose images
    outRealHD = forward(discHD,imRealHD);
    outFakeHD = forward(discHD,imFakeHD);
    discHDLoss = discLossWeight*computeDiscLoss(outRealHD,outFakeHD);

    % Update parameters of the discriminator for high-dose images
    discHDGrads = dlgradient(discHDLoss,discHD.Learnables);

    % Convert the data type from dlarray to single
    discLDLoss = extractdata(discLDLoss);
    discHDLoss = extractdata(discHDLoss);
end
```

Loss Functions

The `computeDiscLoss` helper function calculates discriminator loss. Each discriminator loss is a sum of two components:

- The squared difference between a vector of ones and the predictions of the discriminator on real images, Y_{real}
- The squared difference between a vector of zeros and the predictions of the discriminator on generated images, $\widehat{Y}_{translated}$

$$discriminatorLoss = (1 - Y_{real})^2 + \left(0 - \widehat{Y}_{translated}\right)^2$$

```
function discLoss = computeDiscLoss(Yreal,Ytranslated)
    discLoss = mean(((1-Yreal).^2),"all") + ...
               mean(((0-Ytranslated).^2),"all");
end
```

The `computeAdvLoss` helper function calculates adversarial loss for the generator. Adversarial loss is the squared difference between a vector of ones and the discriminator predictions on the translated image.

$$adversarialLoss = \left(1 - \widehat{Y}_{translated}\right)^2$$

```
function advLoss = computeAdvLoss(Ytranslated)
    advLoss = mean(((Ytranslated-1).^2),"all");
end
```

The `computeReconLoss` helper function calculates self-reconstruction loss and cycle consistency loss for the generator. Self reconstruction loss is the L^1 distance between the input images and their self-reconstructed versions. Cycle consistency loss is the L^1 distance between the input images and their cycle-reconstructed versions.

$$selfReconstructionLoss = \left\|\left(Y_{real} - Y_{self-reconstructed}\right)\right\|_1$$

$$cycleConsistencyLoss = \left\|\left(Y_{real} - Y_{cycle-reconstructed}\right)\right\|_1$$

```
function reconLoss = computeReconLoss(Yreal,Yrecon)
    reconLoss = mean(abs(Yreal-Yrecon),"all");
end
```

The `computeKLLoss` helper function calculates hidden KL loss and cycle-hidden KL loss for the generator. Hidden KL loss is the squared difference between a vector of zeros and the '`encoderSharedBlock`' activation for the self-reconstruction stream. Cycle-hidden KL loss is the squared difference between a vector of zeros and the '`encoderSharedBlock`' activation for the cycle-reconstruction stream.

$$hiddenKLLoss = (0 - Y_{encoderSharedBlockActivation})^2$$

$$cycleHiddenKLLoss = (0 - Y_{encoderSharedBlockActivation})^2$$

```
function klLoss = computeKLLoss(hidden)
    klLoss = mean(abs(hidden.^2),"all");
end
```

References

[1] Liu, Ming-Yu, Thomas Breuel, and Jan Kautz, "Unsupervised image-to-image translation networks". In *Advances in Neural Information Processing Systems,* 2017. https://arxiv.org/pdf/1703.00848.pdf.

[2] McCollough, C.H., Chen, B., Holmes, D., III, Duan, X., Yu, Z., Yu, L., Leng, S., Fletcher, J. (2020). Data from Low Dose CT Image and Projection Data [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/9npb-2637.

[3] Grants EB017095 and EB017185 (Cynthia McCollough, PI) from the National Institute of Biomedical Imaging and Bioengineering.

[4] Clark, Kenneth, Bruce Vendt, Kirk Smith, John Freymann, Justin Kirby, Paul Koppel, Stephen Moore, et al. “The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository.” *Journal of Digital Imaging* 26, no. 6 (December 2013): 1045–57. https://doi.org/10.1007/s10278-013-9622-7.

See Also

`unitGenerator` | `unitPredict` | `patchGANDiscriminator` | `minibatchqueue` | `dlarray` | `dlfeval` | `adamupdate`

Related Examples

- “Unsupervised Medical Image Denoising Using CycleGAN” on page 9-130

More About

- “Get Started with GANs for Image-to-Image Translation” (Image Processing Toolbox)
- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

Detect Image Anomalies Using Explainable One-Class Classification Neural Network

This example shows how to detect and localize anomalies such as cracks in concrete using explainable single-class classification.

In one-class approaches to anomaly detection, training is semi-supervised, meaning that the network trains on data consisting only of examples of images without anomalies [1 on page 9-0]. Despite training on samples only of normal scenes, the model learns how to distinguish between normal and anamalous scenes. One-class learning offers many advantages for anomaly detection problems:

1. Representations of anomalies can be scarce.
2. Anomalies can represent expensive or catastophic outcomes.
3. There can be many kinds of anomalies, and the kinds of anomalies can change over the lifetime of the model. Describing what "good" looks like than is often more feasable than providing data that represents all posssible anomalies in real world settings.

A crucial part of anomaly detection is for a human observer to be able to understand why a trained network classifies images as anomalies. Explainable classification supplements the class prediction with information that justifies how the neural network reached its classification decision.

This example explores how one-class deep learning can be used to create accurate anomaly detection classifiers. The example also implements explainable classification using a network that returns a heatmap with the probability that each pixel is anomalous.

Download Concrete Crack Images for Classification Data Set

This example works with the Concrete Crack Images for Classification data set. The data set contains images of two classes: `Negative` images without cracks present in the road and `Positive` images with cracks. The data set provides 20,000 images of each class. The size of the data set is 235 MB.

Set `dataDir` as the desired location of the data set.

```
dataDir = fullfile(tempdir,"ConcreteCracks");
if ~exist(dataDir,"dir")
    mkdir(dataDir);
end
```

To download the data set, go to this link: https://md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws.com/5y9wdsg2zt-2.zip. Extract the ZIP file to obtain a RAR file, then extract the contents of the RAR file into the directory specified by the `dataDir` variable. When extracted successfully, `dataDir` contains two subdirectories: `Negative` and `Positive`.

Load and Preprocess Data

Create an `imageDatastore` that reads and manages the image data. Label each image as `Positive` or `Negative` according to the name of its directory.

```
imdsP = imageDatastore(fullfile(dataDir,"Positive"),LabelSource="foldernames");
imdsN = imageDatastore(fullfile(dataDir,"Negative"),LabelSource="foldernames");
```

Display an example of each class. Display a negative, or good, image without crack anomalies on the left. In the good image, imperfections and deviations in texture are small. Display a positive, or anomalous, image on the right. The anomalous image shows a large black crack oriented vertically.

```
samplePositive = preview(imdsP);
sampleNegative = preview(imdsN);
montage({sampleNegative,samplePositive})
title("Road Images Without (Left) and With (Right) Crack Anomalies")
```

Partition Data into Training, Validation, and Test Sets

To simulate a more typical semi-supervised workflow, create a training set of only 250 good images. Include a small collection of anomalous training images to provide better classification results. Create a balanced test set using 1000 images from each class. Create a balanced validation set using 100 images of each class.

```
numTrainNormal = 250;
numTrainAnomaly = 10;
numTest = 1000;
numVal = 100;

[dsTrainPos,dsTestPos,dsValPos,~] = splitEachLabel(imdsP,numTrainAnomaly,numTest,numVal);
[dsTrainNeg,dsTestNeg,dsValNeg,~] = splitEachLabel(imdsN,numTrainNormal,numTest,numVal);

dsTrain = imageDatastore(cat(1,dsTrainPos.Files,dsTrainNeg.Files),LabelSource="foldernames");
dsTest = imageDatastore(cat(1,dsTestPos.Files,dsTestNeg.Files),LabelSource="foldernames");
dsVal = imageDatastore(cat(1,dsValPos.Files,dsValNeg.Files),LabelSource="foldernames");
```

Define an anonymous function, `addLabelFcn`, that creates a one-hot encoded representation of label information from an input image. Then, transform the datastores using the `transform` function such that the datastores return a cell array of image data and corresponding one-hot encoded array. The `transform` function applies the operations specified by the anonymous `addLabelFcn` function.

```
addLabelFcn = @(x,info) deal({x,onehotencode(info.Label,1)},info);
dsTrain = transform(dsTrain,addLabelFcn,IncludeInfo=true);
```

```
dsVal = transform(dsVal,addLabelFcn,IncludeInfo=true);
dsTest = transform(dsTest,addLabelFcn,IncludeInfo=true);
```

Augment Training Data

Augment the training data by using the `transform` function with custom preprocessing operations specified by the helper function `augmentDataForConcreteClassification`. The helper function is attached to the example as a supporting file.

The `augmentDataForConcreteClassification` function randomly applies 90 degree rotation and horizontal and vertical reflection to each input image.

```
dsTrain = transform(dsTrain,@augmentDataForConcreteClassification);
```

Batch Training Data

Create a `minibatchqueue` object that manages the mini-batching of observations in a custom training loop. The `minibatchqueue` object also casts data to a `dlarray` object that enables automatic differentiation in deep learning applications.

Specify the mini-batch data extraction format as `"SSCB"` (spatial, spatial, channel, batch). Set the `DispatchInBackground` name-value argument as the boolean returned by `canUseGPU`. If a supported GPU is available for computation, then the `minibatchqueue` object preprocesses mini-batches in the background in a parallel pool during training.

```
mbSize = 128;
mbqTrain = minibatchqueue(dsTrain,PartialMiniBatch="discard", ...
    MiniBatchFormat=["SSCB","CB"],MiniBatchSize=mbSize);

mbqVal = minibatchqueue(dsVal,MiniBatchFormat=["SSCB","CB"],MiniBatchSize=mbSize);
```

Calculate Training Set Statistics for Input Normalization

Calculate the per-channel mean of the training images for use in zero-mean normalization.

```
queue = copy(mbqTrain);
queue.PartialMiniBatch = "return";

X = next(queue);
sumImg = sum(X,4);

while hasdata(queue)
    X = next(queue);
    sumImg = sumImg + sum(X,4);
end
trainSetMean = sumImg ./ dsTrain.numpartitions;
trainSetMean = mean(trainSetMean,[1 2]);
```

Create FCDD Model

This example uses a fully convolutional data description (FCDD) model [1 on page 9-0]. The basic idea of FCDD is to train a network to produce a heatmap from an input image.

This example uses a VGG-16 network [3 on page 9-0] trained on ImageNet [4 on page 9-0] as the base fully convolutional network architecture. The example freezes the majority of the model and randomly initializes and trains the final convolutional stages. This approach enables quick training with small amounts of input training data.

The `vgg16` function returns a pretrained VGG-16 network. This function requires the Deep Learning Toolbox™ Model *for VGG-16 Network* support package. If this support package is not installed, then the function provides a download link.

```
pretrainedVGG = vgg16;
```

Freeze the first 24 layers of the network by setting the weights and bias to `0`.

```
numLayersToFreeze = 24;
net = pretrainedVGG.Layers(1:numLayersToFreeze);
for idx = 1:numLayersToFreeze
    if isprop(net(idx),"Weights")
        net(idx) = setLearnRateFactor(net(idx),Weights=0);
        net(idx) = setLearnRateFactor(net(idx),Bias=0);
    end
end
```

Add a final convolutional stage. This stage is similar to the next convolutional stage of VGG-16 but with randomly initialized and trainable convolutional layers and with batch normalization. A final 1-by-1 convolution compresses the network output into a one-channel heatmap. The final layer is a Pseudo-Huber loss function used to stabilize training with the FCDD loss [1 on page 9-0] [2 on page 9-0].

```
finalLayers = [
    convolution2dLayer(3,512,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,512,Padding="same")
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(1,1)
    functionLayer(@(x)sqrt(x.^2+1)-1,Name="pseudoHuber")];
```

Assemble the complete network.

```
net = dlnetwork([net;finalLayers]);
lgraph = layerGraph(net);
```

Replace the image input layer in the encoder with a new input layer that performs zero-center normalization using the computed mean. Set the input size of the network equal to the size of the images in the data set.

```
inputSize = size(sampleNegative);
newInputLayer = imageInputLayer(inputSize,Normalization="zerocenter", ...
    Mean=extractdata(trainSetMean),Name="inputLayer");
lgraph = replaceLayer(lgraph,lgraph.Layers(1).Name,newInputLayer);
```

Specify Training Options

Specify the training options for Adam optimization. Train the network for 70 epochs.

```
learnRate = 1e-4;
averageGrad = [];
averageSqGrad = [];
numEpochs = 70;
```

Train Network or Download Pretrained Network

By default, this example downloads a pretrained version of the VGG-16 network using the helper function `downloadTrainedConcreteClassificationNet`. The pretrained network can be used to run the entire example without waiting for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. Train the model in a custom training loop. For each iteration:

- Read the data for the current mini-batch using the `next` function.
- Evaluate the model gradients using the `dlfeval` function and the `modelGradients` helper function. This function is defined at the end of this example.
- Update the network parameters using the `adamupdate` function.
- Update a plot of the loss.

Train on one or more GPUs, if available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 7 minutes on an NVIDIA™ Titan X.

```
doTraining = false;
if doTraining

    [hFig,batchLine] = initializeTrainingPlotConcreteClassification;
    start = tic;

    iteration = 0;
    for epoch = 1:numEpochs
        reset(mbqTrain);
        shuffle(mbqTrain);

        while hasdata(mbqTrain)
            [X,T] = next(mbqTrain);
            [grad,loss,state] = dlfeval(@modelGradients,net,X,T);

            iteration = iteration + 1;

            [net,averageGrad,averageSqGrad] = adamupdate(net, ...
                grad,averageGrad,averageSqGrad,iteration,learnRate);

            net.State = state;

            % Update the plot
            updateTrainingPlotConcreteClassification(batchLine,iteration,loss,start,epoch);
        end
    end
else
    trainedConcreteClassificationNet_url = 'https://www.mathworks.com/supportfiles/vision/data/t
    downloadTrainedConcreteClassificationNet(trainedConcreteClassificationNet_url,dataDir);
    load(fullfile(dataDir,"trainedAnomalyDetectionNet.mat"));
end
```

Create Classification Model

Classify an image as good or anomalous using the anomaly heatmap predicted by the network in two steps. First, create a classifier that returns the probability that an image has an anomaly based on the

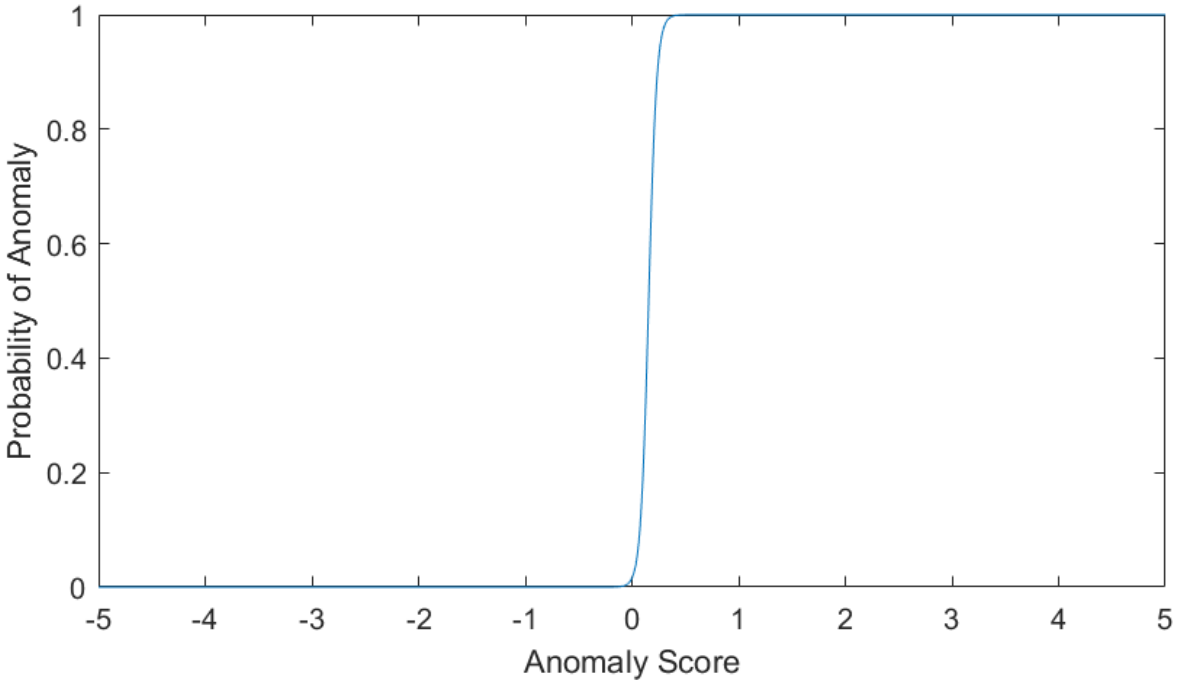
heatmap predicted by the trained network. Then, you can assign a class label to test images based on the probability returned by the classifier.

Calculate the mean anomaly score and the known ground truth label (`Positive` or `Negative`) for each image in the validation set. This example chooses the anomaly score threshold using the validation set to avoid leaking information from the test set into the design of the classifier.

```
reset(mbqVal);

scores = zeros(dsVal.numpartitions,1);
labels = zeros(dsVal.numpartitions,1);
idx = 1;
while hasdata(mbqVal)
    [X,T] = next(mbqVal);
    batchSize = size(X,4);
    Y = predict(net,X);
    Y = mean(Y,[1 2]);
    T = onehotdecode(T,[0 1],1,"double");
    scores(idx:idx+batchSize-1) = Y;
    labels(idx:idx+batchSize-1) = T;
    idx = idx+batchSize;
end
```

Fit a sigmoid to the anomaly scores and associated class labels. The resulting anonymous function, `anomalyProbability`, takes an input anomaly score predicted by the network and returns a probability that the score describes an anomaly. Outputs from `anomalyProbability` greater than 0.5 are considered anomalies.

```
coeff = glmfit(scores,labels,"binomial","link","logit");
sigmoid = @(x) 1./(1+exp(-x));
anomalyProbability = @(x) sigmoid(coeff(2).*x + coeff(1));
fplot(anomalyProbability)
xlabel("Anomaly Score")
ylabel("Probability of Anomaly")
```



The default sigmoid has a value of 0.5 at a score of 0. Calculate the anomaly score at which the fitted sigmoid defined by the `anomalyProbability` function has a value of 0.5.

```
threshold = -coeff(1)/coeff(2)
```

```
threshold = 0.1567
```

Evaluate Classification Model

Predict the anomaly heatmap and calculate the mean anomaly score for each image in the test set. Also get the ground truth labels of each test image.

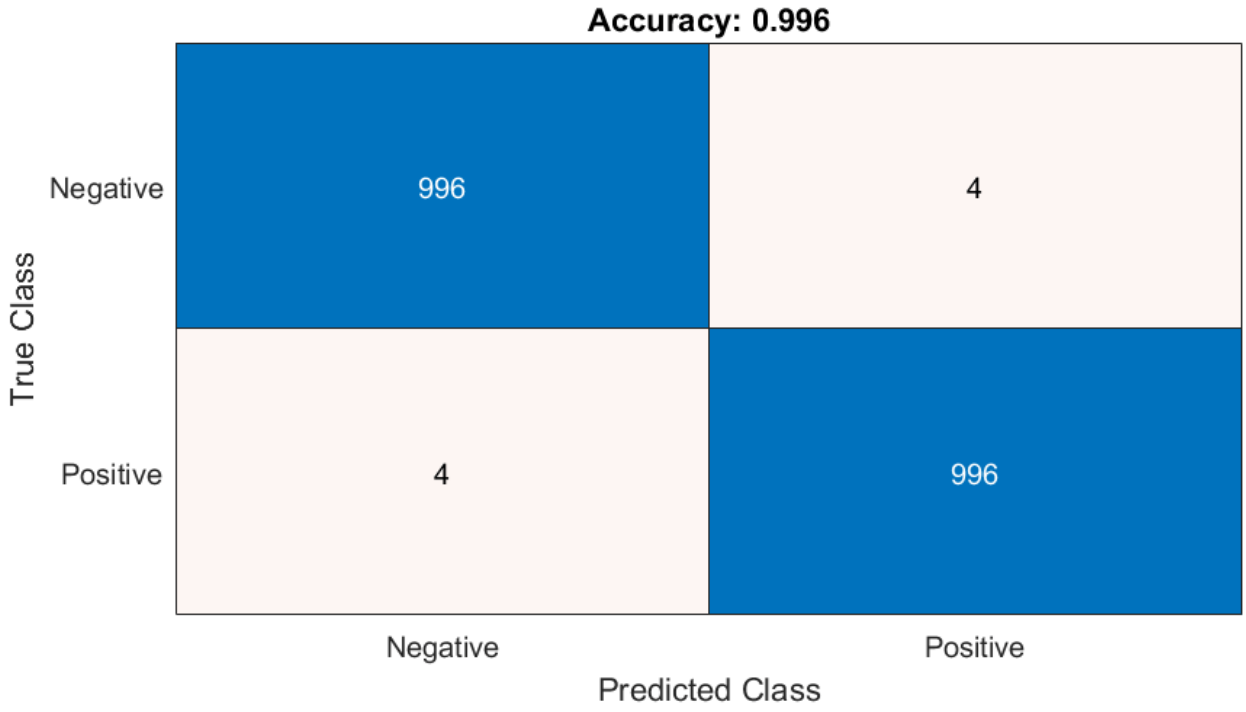
```
q = minibatchqueue(dsTest,MiniBatchSize=64,MiniBatchFormat=["SSCB","CB"]);
scores = zeros(dsTest.numpartitions,1);
labels = zeros(dsTest.numpartitions,1);
idx = 1;
while hasdata(q)
    [X,T] = next(q);
    batchSize = size(X,4);
    Y = predict(net,X);
    Y = mean(Y,[1 2]);
    T = onehotdecode(T,[0 1],1,"double");
    scores(idx:idx+batchSize-1) = Y;
    labels(idx:idx+batchSize-1) = T;
    idx = idx+batchSize;
end
```

Predict class labels for the test set using the classification model defined by the `anomalyProbability` function.

```
predictedLabels = anomalyProbability(scores) > 0.5;
```

Calculate the confusion matrix and the classification accuracy for the test set. The classification model in this example is very accurate and predicts a small percentage of false positives and false negatives.

```
targetLabels = logical(labels);
M = confusionmat(targetLabels,predictedLabels);
confusionchart(M,["Negative","Positive"])
acc = sum(diag(M)) / sum(M,'all');
title(sprintf("Accuracy: %g",acc));
```



Explain Classification Decisions

View Heatmap of Anomaly

Select and display an image of a correctly classified anomaly. This result is a true positive classification.

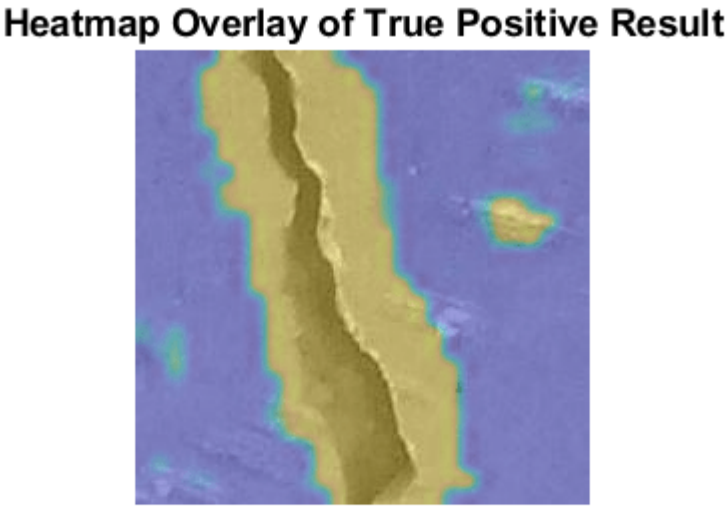
```
idxTruePositive = find(targetLabels & predictedLabels,1);
dsExample = subset(dsTest,idxTruePositive);
dataTP = preview(dsExample);
imgTP = dataTP{1};
imshow(imgTP)
title("True Positive Test Image")
```

Obtain a heatmap of an anomaly image by calling the `predict` function on the trained network. The network returns a heatmap with a smaller size than the input image, so resize the heatmap to the size of the input image.

```
hmapTP = predict(net,gpuArray(dlarray(single(imgTP),"SSCB")));
hmapTP = gather(extractdata(hmapTP));
hmapTP = anomalyProbability(hmapTP);
hmapTP = imresize(hmapTP,OutputSize=size(imgTP,[1 2]));
```

Display an overlay of a colored heatmap on a grayscale representation of the color image using the `heatmapOverlay` helper function. This function is defined at the end of the example. Adjust the opacity of the overlaid heatmap by setting `alpha` to a number in the range [0, 1]. For a fully opaque heatmap, specify `alpha` as `1`. For a fully transparent heatmap, specify `alpha` as `0`.

```
alpha = 0.4;
imshow(heatmapOverlay(imgTP,hmapTP,alpha))
title("Heatmap Overlay of True Positive Result")
```



To quantitatively confirm the result, display the mean heatmap anomaly score of the true positive test image as predicted by the network.

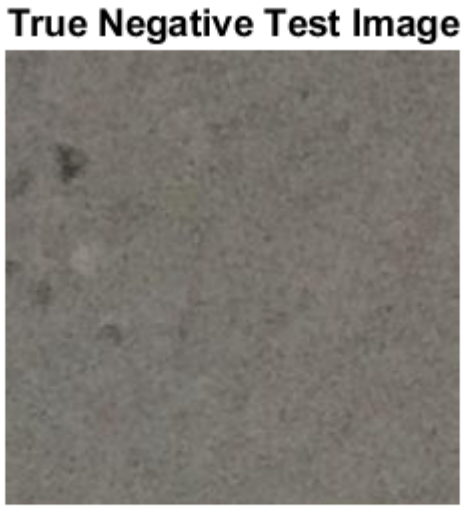
```
disp("Mean heatmap anomaly score of test image: "+scores(idxTruePositive(1)))
```

```
Mean heatmap anomaly score of test image: 1.2185
```

View Heatmap of Normal Image

Select and display an image of a correctly classified normal image. This result is a true negative classification.

```
idxTrueNegative = find(~targetLabels & ~predictedLabels);
dsTestTN = subset(dsTest,idxTrueNegative);
dataTN = read(dsTestTN);
imgTN = dataTN{1};
imshow(imgTN)
title("True Negative Test Image")
```

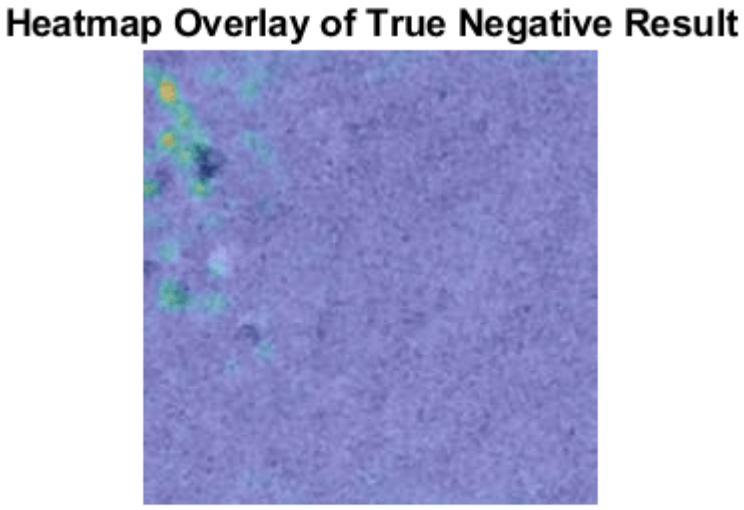


Obtain a heatmap of the correctly predicted negative image by calling the `predict` function on the trained network. Resize the heatmap to the size of the input image.

```
hmapTN = predict(net,gpuArray(dlarray(single(imgTN),"SSCB")));
hmapTN = gather(extractdata(hmapTN));
hmapTN = anomalyProbability(hmapTN);
hmapTN = imresize(hmapTN,OutputSize=size(imgTN,[1 2]));
```

Display an overlay of a colored heatmap on a grayscale representation of the color image using the `heatmapOverlay` helper function. This function is defined at the end of the example. Many true negative test images, such as this test image, either have small anomaly scores across the entire image or large anomaly scores in a localized portion of the image.

```
imshow(heatmapOverlay(imgTN,hmapTN,alpha))
title("Heatmap Overlay of True Negative Result")
```



Display the mean heatmap anomaly score of the true negative test image as predicted by the network.

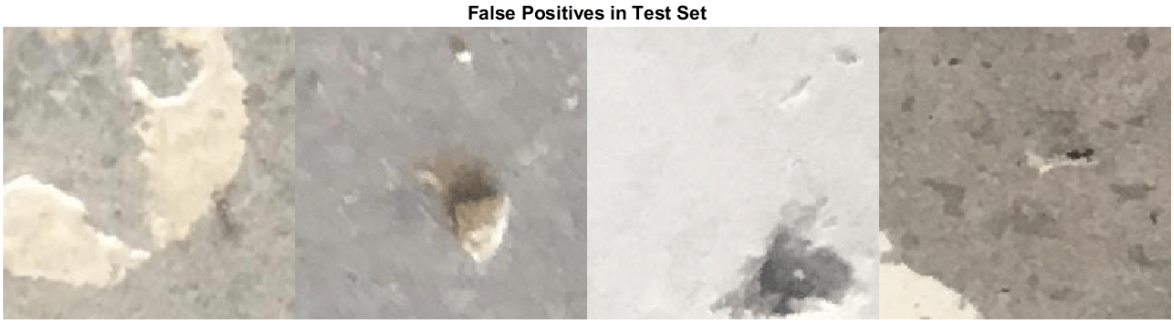
```
disp("Mean heatmap anomaly score of test image: "+scores(idxTrueNegative(1)))
```

```
Mean heatmap anomaly score of test image: 0.0022878
```

Visualize False Positives

False positives are images without crack anomalies that the network classifies as anomalous. Display any false positive images from the test set in a montage.

```
falsePositiveIdx = find(predictedLabels & ~targetLabels);
dataFP = readall(subset(dsTest,falsePositiveIdx));
numFP = length(falsePositiveIdx);
if numFP>0
    montage(dataFP(:,1),Size=[1,numFP]);
    title("False Positives in Test Set")
end
```



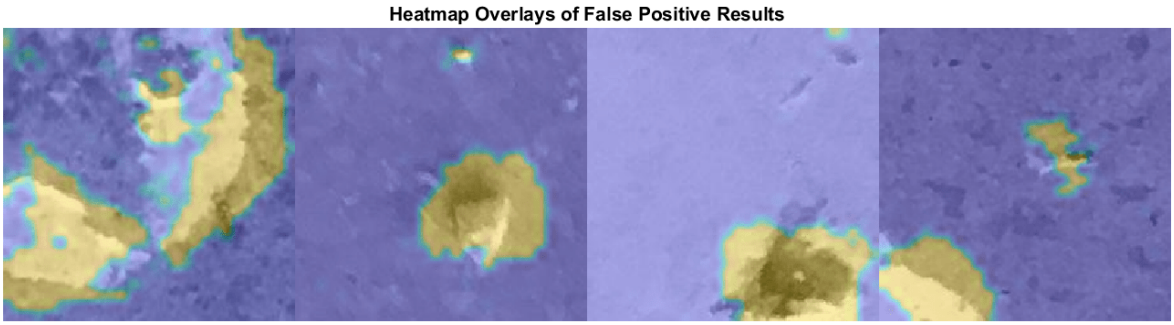
Use the explanation from the network to gain insights into the misclassifications.

Calculate the heatmap overlays.

```
hmapOverlay = cell(1,numFP);
for idx = 1:numFP
    img = dataFP{idx,1};
    map = predict(net,gpuArray(dlarray(single(img),"SSCB")));
    map = gather(extractdata(map));
    mapProb = anomalyProbability(map);
    hmap = imresize(mapProb,OutputSize=size(img,[1 2]));
    hmapOverlay{idx} = heatmapOverlay(img,hmap,alpha);
end
```

The false positive images show features such as rocks that have similar visual characteristics to cracks. In all cases, these are hard classification problems in which the explainable nature of the network architecture is useful for gaining insights.

```
if numFP>0
    montage(hmapOverlay,Size=[1,numFP])
    title("Heatmap Overlays of False Positive Results")
end
```



Display the mean anomaly heatmap scores of the false positive test images as predicted by the network. The mean scores are larger than the threshold used by the classification model, resulting in misclassifications.

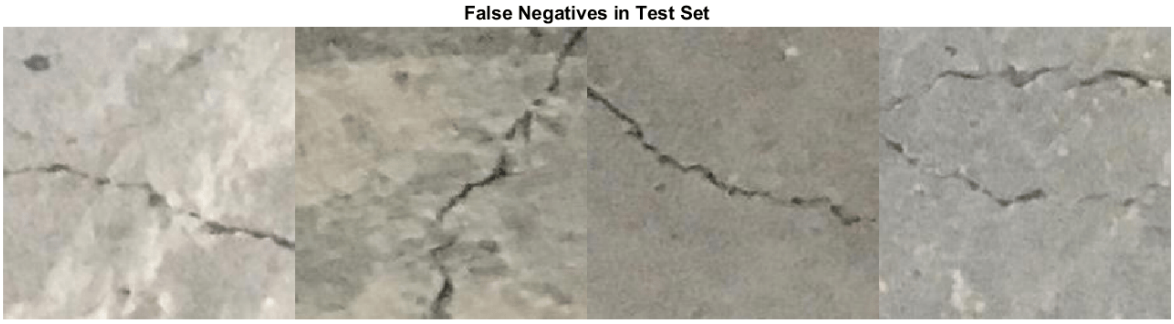
```
disp("Mean heatmap anomaly scores:"); scores(falsePositiveIdx)
```

```
Mean heatmap anomaly scores:

ans = 4×1

    0.3218
    0.1700
    0.2516
    0.1862
```

Visualize False Negatives

False negatives are images with anomalies that the network classifies as good. Display any false negative images from the test set in a montage. The false negative images show relatively thin visible cracks.

```
falseNegativeIdx = find(~predictedLabels & targetLabels);
dataFN = readall(subset(dsTest,falseNegativeIdx));
numFN = length(falseNegativeIdx);
if numFN>0
    montage(dataFN(:,1),Size=[1,numFN])
    title("False Negatives in Test Set")
end
```



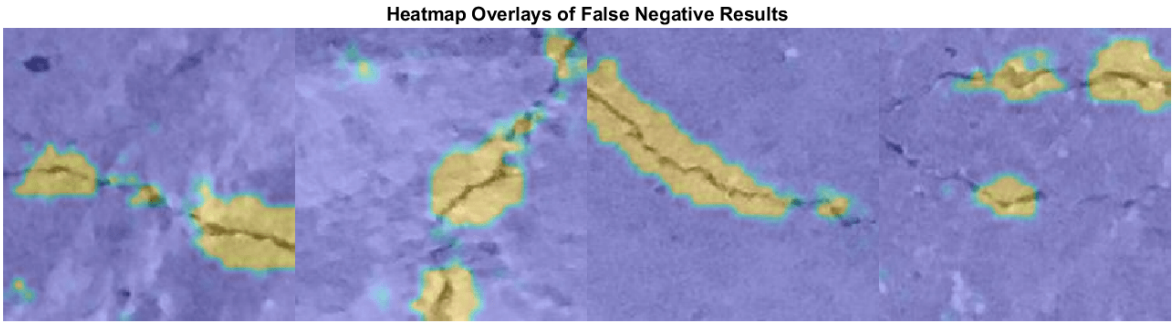
Calculate the heatmap overlays.

```
hmapOverlay = cell(1,numFN);
for idx = 1:numFN
    img = dataFN{idx,1};
    map = predict(net,gpuArray(dlarray(single(img),"SSCB")));
    map = gather(extractdata(map));
    mapProb = anomalyProbability(map);
    hmap = imresize(mapProb,OutputSize=size(img,[1 2]));
    hmapOverlay{idx} = heatmapOverlay(img,hmap,alpha);
end
```

Display the heatmap overlays. The network assigns a large anomaly score along cracks, as expected.

```
if numFN>0
    montage(hmapOverlay,Size=[1,numFN])
```

```
    title("Heatmap Overlays of False Negative Results")
end
```



Display the mean heatmap anomaly scores of the negative positive test images as predicted by the network. The mean scores are smaller than the threshold used by the sigmoid classification model, resulting in misclassifications.

```
disp("Mean heatmap anomaly scores:"); scores(falseNegativeIdx)
```

```
Mean heatmap anomaly scores:

ans = 4×1

    0.1477
    0.1186
    0.1241
    0.0920
```

Supporting Functions

The `modelGradients` helper function calculates the gradients and loss for the network.

```
function [grad,loss,state] = modelGradients(net,X,T)
    [Y,state] = forward(net,X);
    loss = fcddLoss(Y,T);
    grad = dlgradient(loss,net.Learnables);
end
```

The `fcddLoss` helper function calculates the FDCC loss.

```
function loss = fcddLoss(Y,T)
    normalTerm = mean(Y,[1 2 3]);
    anomalyTerm = log(1-exp(-normalTerm));

    isNegative = T(1,:) == 1;
    loss = mean(single(isNegative) .* normalTerm-single(~isNegative) .* anomalyTerm);
end
```

The `heatmapOverlay` helper function overlays a colored heatmap `hmap` on a grayscale representation of the image `img`. Adjust the transparency of the heatmap overlay by specifying the `alpha` argument as a number in the range [0, 1].

```
function out = heatmapOverlay(img,hmap,alpha)

    % Normalize to the range [0, 1]
    img = mat2gray(img);
    hmap = mat2gray(hmap);

    % Convert image to a 3-channel grayscale representation
    imgGray = im2gray(img);
    imgGray = repmat(imgGray,[1 1 3]);

    % Convert heatmap to an RGB image using a colormap
    cmap = parula(256);
    hmapRGB = ind2rgb(gray2ind(hmap,size(cmap,1)),cmap);

    % Blend results
    hmapWeight = alpha;
    grayWeight = 1-hmapWeight;
    out = im2uint8(grayWeight.*imgGray + hmapWeight.*hmapRGB);
end
```

References

[1] Liznerski, Philipp, Lukas Ruff, Robert A. Vandermeulen, Billy Joe Franks, Marius Kloft, and Klaus-Robert Müller. "Explainable Deep One-Class Classification." Preprint, submitted March 18, 2021. https://arxiv.org/abs/2007.01760.

[2] Ruff, Lukas, Robert A. Vandermeulen, Billy Joe Franks, Klaus-Robert Müller, and Marius Kloft. "Rethinking Assumptions in Deep Anomaly Detection." Preprint, submitte, May 30, 2020. https://arxiv.org/abs/2006.00339.

[3] Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." Preprint, submitted April 10, 2015. https://arxiv.org/abs/1409.1556.

[4] *ImageNet*. https://www.image-net.org.

See Also

`transform` | `dlarray` | `dlfeval` | `adamupdate` | `minibatchqueue`

More About

- “Datastores for Deep Learning” on page 19-2
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225

Automated Driving Examples

Train a Deep Learning Vehicle Detector

This example shows how to train a vision-based vehicle detector using deep learning.

Overview

Vehicle detection using computer vision is an important component for tracking vehicles around the ego vehicle. The ability to detect and track vehicles is required for many autonomous driving applications, such as for forward collision warning, adaptive cruise control, and automated lane keeping. Automated Driving Toolbox™ provides pretrained vehicle detectors (`vehicleDetectorFasterRCNN` (Automated Driving Toolbox) and `vehicleDetectorACF` (Automated Driving Toolbox)) to enable quick prototyping. However, the pretrained models might not suit every application, requiring you to train from scratch. This example shows how to train a vehicle detector from scratch using deep learning.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several deep learning techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a Faster R-CNN vehicle detector using the `trainFasterRCNNObjectDetector` function. For more information, see “Object Detection” (Computer Vision Toolbox).

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTrainingAndEval` variable to true.

```
doTrainingAndEval = false;
if ~doTrainingAndEval && ~exist('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToE
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small labeled dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the Faster R-CNN training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

Split the data set into a training set for training the detector and a test set for evaluating the detector. Select 60% of the data for training. Use the rest for evaluation.

```
rng(0)
shuffledIdx = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));
```

```
trainingDataTbl = vehicleDataset(shuffledIdx(1:idx),:);
testDataTbl = vehicleDataset(shuffledIdx(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

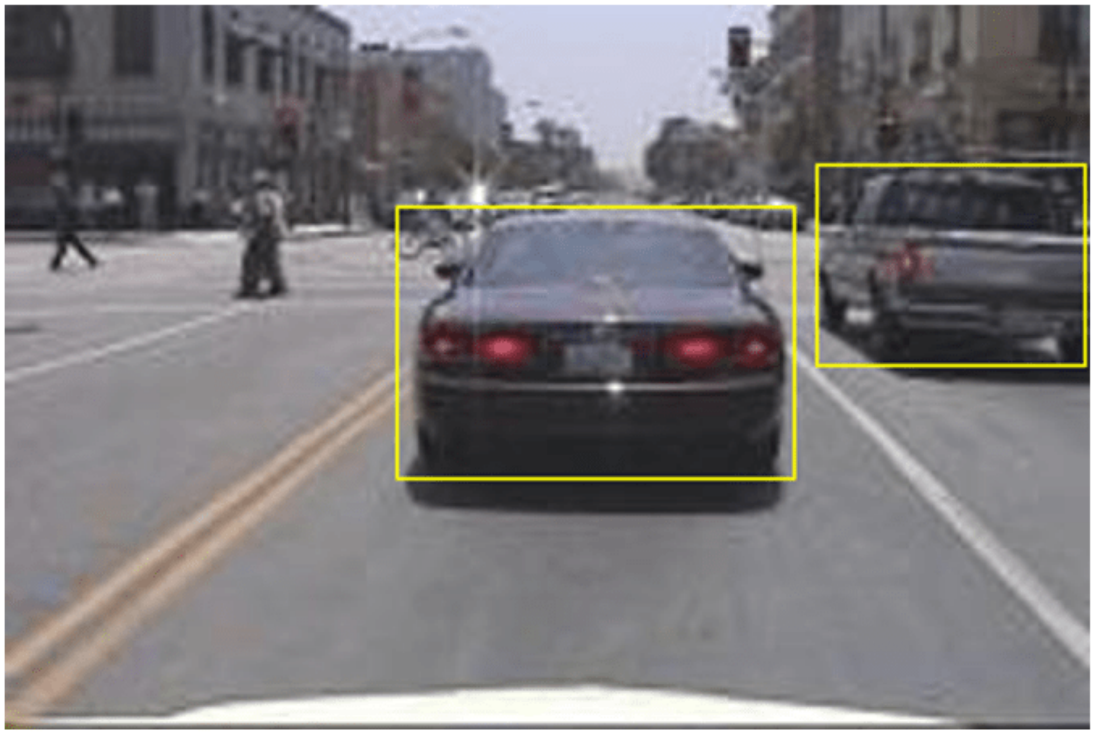
```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest,bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Create Faster R-CNN Detection Network

A Faster R-CNN object detection network is composed of a feature extraction network followed by two subnetworks. The feature extraction network is typically a pretrained CNN, such as ResNet-50 or Inception v3. The first subnetwork following the feature extraction network is a region proposal network (RPN) trained to generate object proposals - areas in the image where objects are likely to exist. The second subnetwork is trained to predict the actual class of each object proposal.

The feature extraction network is typically a pretrained CNN (for details, see “Pretrained Deep Neural Networks” on page 1-8). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18, depending on your application requirements.

Use `fasterRCNNLayers` to create a Faster R-CNN network automatically given a pretrained feature extraction network. `fasterRCNNLayers` requires you to specify several inputs that parameterize a Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes.

```
preprocessedTrainingData = transform(trainingData, @(data)preprocessData(data,inputSize));
numAnchors = 4;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

```
anchorBoxes = 4×2

    96    91
    68    65
   150   125
    38    29
```

For more information on choosing anchor boxes, see “Estimate Anchor Boxes From Training Data” (Computer Vision Toolbox) (Computer Vision Toolbox™) and “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis. You can use `analyzeNetwork` to find the names of other potential feature extraction layers within a network.

```
featureLayer = 'activation_40_relu';
```

Define the number of classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the Faster R-CNN object detection network.

```
lgraph = fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featureLayer
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the Faster R-CNN network architecture, use Deep Network Designer to design the Faster R-CNN detection network manually. For more information, see “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” (Computer Vision Toolbox).

Data Augmentation

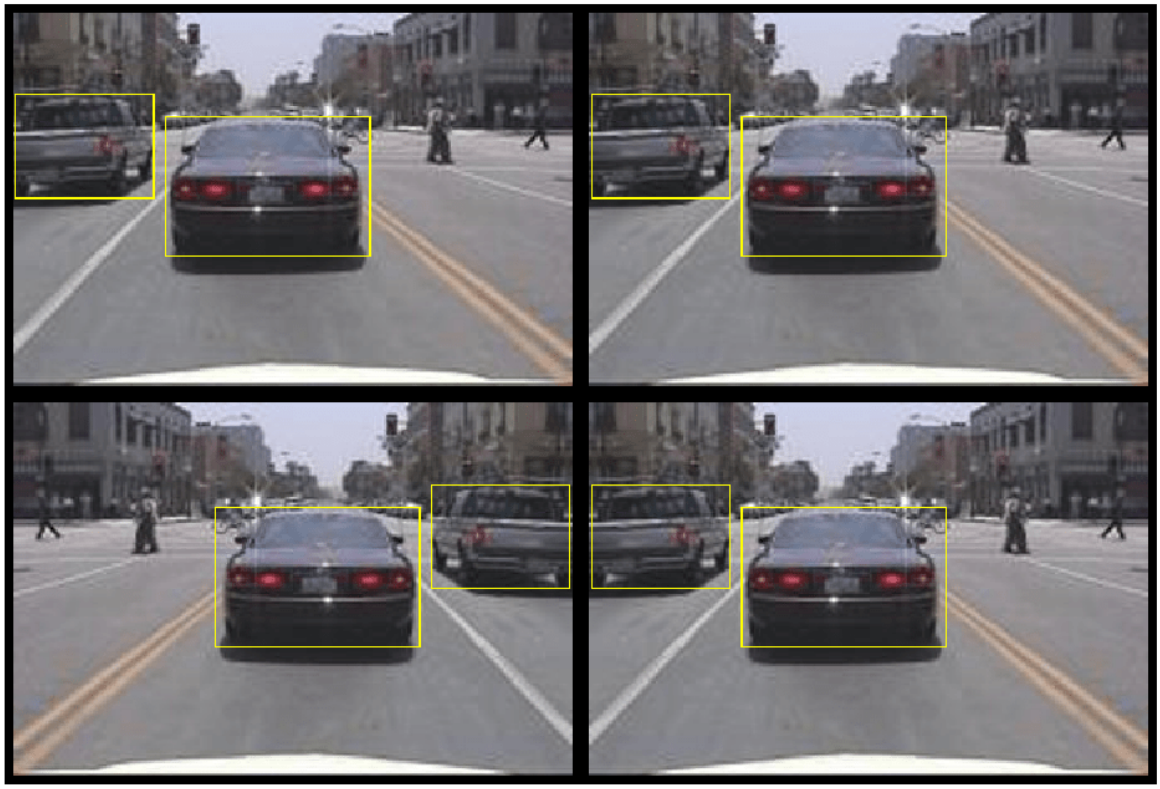
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to test data. Ideally, test data is representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data to prepare for training.

```
trainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed data.

```
data = read(trainingData);
```

Display the image and box bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Train Faster R-CNN

Use `trainingOptions` to specify network training options. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',7,...
    'MiniBatchSize',1,...
    'InitialLearnRate',1e-3,...
    'CheckpointPath',tempdir);
```

Use `trainFasterRCNNObjectDetector` to train Faster R-CNN object detector if `doTrainingAndEval` is true. Otherwise, load the pretrained network.

```
if doTrainingAndEval
    % Train the Faster R-CNN detector.
    % * Adjust NegativeOverlapRange and PositiveOverlapRange to ensure
```

```
    %   that training samples tightly overlap with ground truth.
    [detector, info] = trainFasterRCNNObjectDetector(trainingData,lgraph,options, ...
        'NegativeOverlapRange',[0 0.3], ...
        'PositiveOverlapRange',[0.6 1]);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detector = pretrained.detector;
end
```

This example was verified on an Nvidia(TM) Titan X GPU with 12 GB of memory. Training the network took approximately 20 minutes. The training time varies depending on the hardware you use.

As a quick check, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{1});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data.

```
testData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

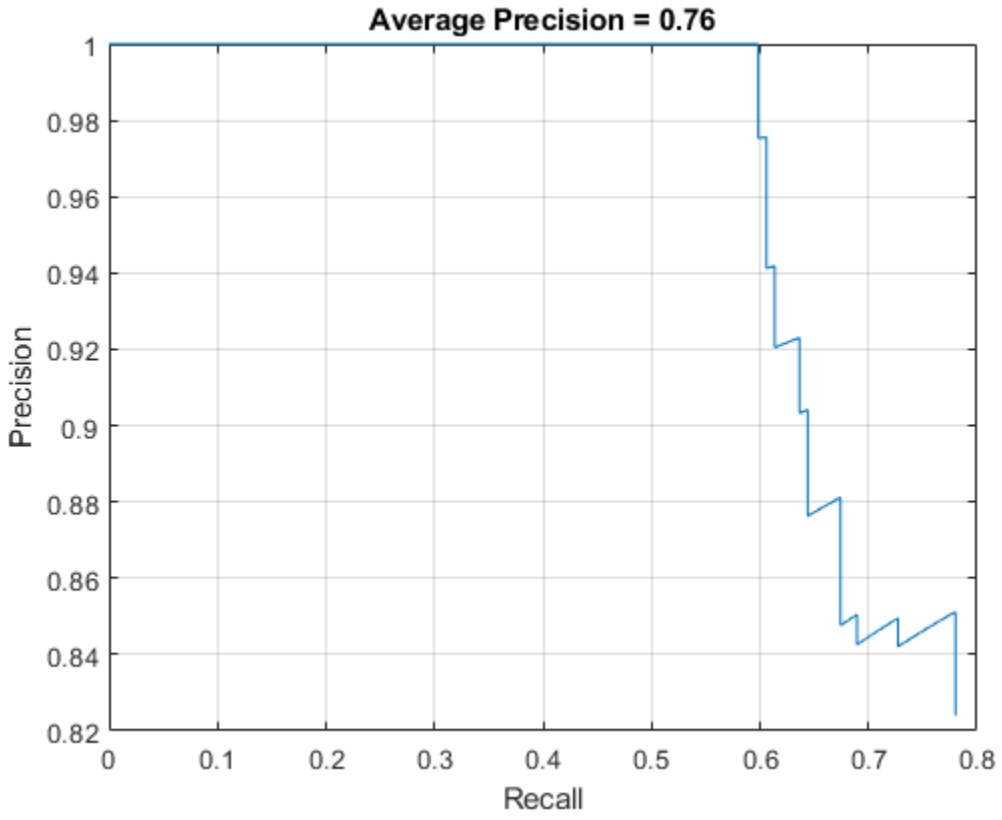
```
if doTrainingAndEval
    detectionResults = detect(detector,testData,'MinibatchSize',4);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detectionResults = pretrained.detectionResults;
end
```

Evaluate the object detector using the average precision metric.

```
[ap, recall, precision] = evaluateDetectionPrecision(detectionResults,testData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```



Supporting Functions

```
function data = augmentData(data)
% Randomly flip images and bounding boxes horizontally.
tform = randomAffine2d('XReflection',true);
sz = size(data{1},[1 2]);
rout = affineOutputView(sz, tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Warp boxes.
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Ren, S., K. He, R. Gershick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions of Pattern Analysis and Machine Intelligence*. Vol. 39, Issue 6, June 2017, pp. 1137-1149.

[2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[3] Girshick, R. "Fast R-CNN." *Proceedings of the 2015 IEEE International Conference on Computer Vision*. Santiago, Chile, Dec. 2015, pp. 1440-1448.

[4] Zitnick, C. L., and P. Dollar. "Edge Boxes: Locating Object Proposals from Edges." *European Conference on Computer Vision*. Zurich, Switzerland, Sept. 2014, pp. 391-405.

[5] Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. "Selective Search for Object Recognition." *International Journal of Computer Vision*. Vol. 104, Number 2, Sept. 2013, pp. 154-171.

See Also

Functions

`trainRCNNObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector`

More About

- “Getting Started with R-CNN, Fast R-CNN, and Faster R-CNN” (Computer Vision Toolbox)
- “Object Detection Using Faster R-CNN Deep Learning” on page 8-201
- “Train Object Detector Using R-CNN Deep Learning” on page 8-188

Create Occupancy Grid Using Monocular Camera and Semantic Segmentation

This example shows how to estimate free space around a vehicle and create an occupancy grid using semantic segmentation and deep learning. You then use this occupancy grid to create a vehicle costmap, which can be used to plan a path.

About Free Space Estimation

Free space estimation identifies areas in the environment where the ego vehicle can drive without hitting any obstacles such as pedestrians, curbs, or other vehicles. A vehicle can use a variety of sensors to estimate free space such as radar, lidar, or cameras. This example focuses on estimating free space from an image sensor using semantic segmentation.

In this example, you learn how to:

- Use semantic image segmentation to estimate free space.
- Create an occupancy grid using the free space estimate.
- Visualize the occupancy grid on a bird's-eye plot.
- Create a vehicle costmap using the occupancy grid.
- Check whether locations in the world are occupied or free.

Download Pretrained Network

This example uses a pretrained semantic segmentation network, which can classify pixels into 11 different classes, including `Road`, `Pedestrian`, `Car`, and `Sky`. The free space in an image can be estimated by defining image pixels classified as `Road` as free space. All other classes are defined as non-free space or obstacles.

The complete procedure for training this network is shown in the “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox) example. Download the pretrained network.

```
% Download the pretrained network.
pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/segnetVGG16CamVid.mat';
pretrainedFolder = fullfile(tempdir,'pretrainedSegNet');
pretrainedSegNet = fullfile(pretrainedFolder,'segnetVGG16CamVid.mat');
if ~exist(pretrainedFolder,'dir')
    mkdir(pretrainedFolder);
    disp('Downloading pretrained SegNet (107 MB)...');
    websave(pretrainedSegNet,pretrainedURL);
    disp('Download complete.');
end

% % Load the network.
data = load(pretrainedSegNet);
net = data.net;
```

Note: Download time of the data depends on your Internet connection. The commands used above block MATLAB® until the download is complete. Alternatively, you can use your web browser to first download the data set to your local disk. In this case, to use the file you downloaded from the web, change the `pretrainedFolder` variable above to the location of the downloaded file.

Estimate Free Space

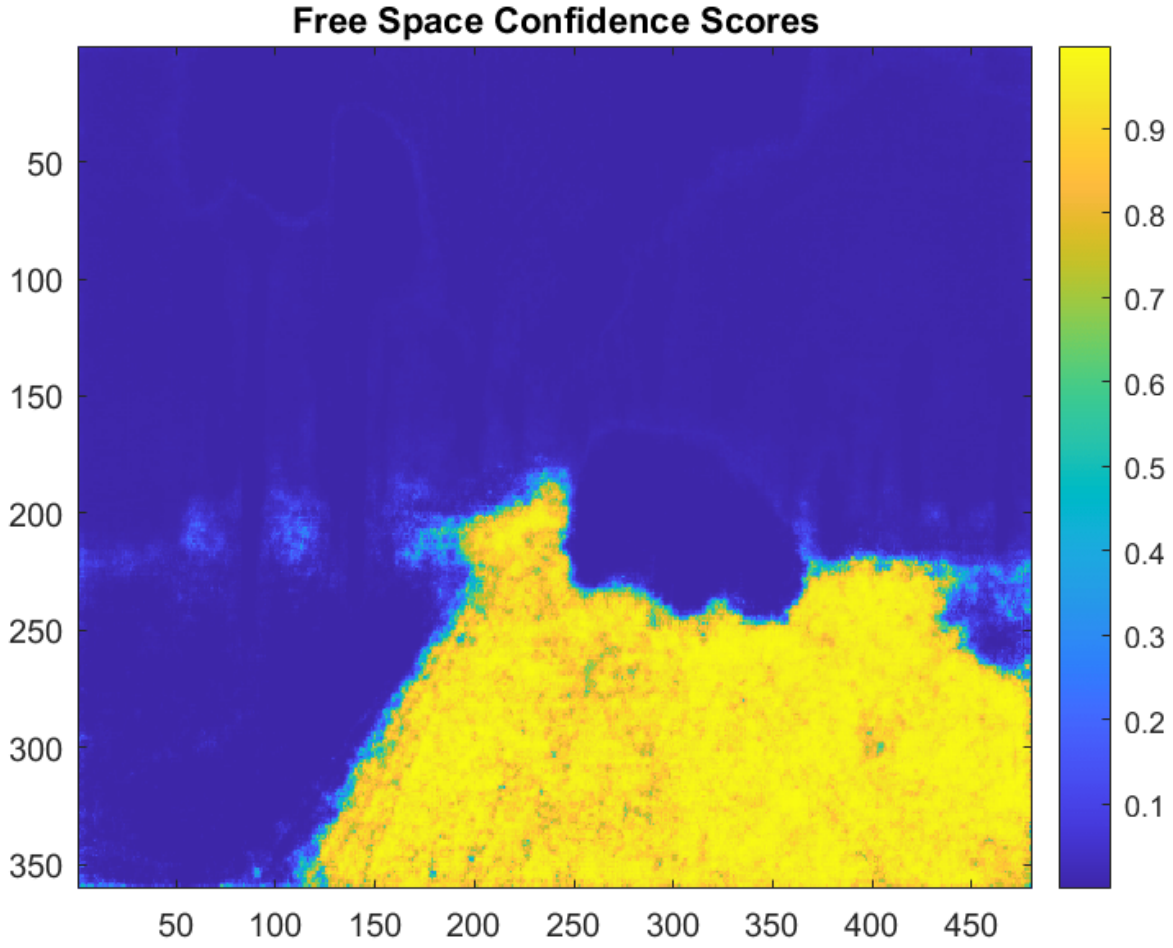
Estimate free space by processing the image using downloaded semantic segmentation network. The network returns classifications for each image pixel in the image. The free space is identified as image pixels that have been classified as `Road`.

The image used in this example is a single frame from an image sequence in the CamVid data set[1]. The procedure shown in this example can be applied to a sequence of frames to estimate free space as a vehicle drives along. However, because a very deep convolutional neural network architecture is used in this example (SegNet with a VGG-16 encoder), it takes about 1 second to process each frame. Therefore, for expediency, process a single frame.

```
% Read the image.
I = imread('seq05vd_snap_shot.jpg');

% Segment the image.
[C,scores,allScores] = semanticseg(I,net);

% Overlay free space onto the image.
B = labeloverlay(I,C,'IncludedLabels',"Road");

% Display free space and image.
figure
imshow(B)
```

To understand the confidence in the free space estimate, display the output score for the Road class for every pixel. The confidence values can be used to inform downstream algorithms of the estimate's validity. For example, even if the network classifies a pixel as Road, the confidence score may be low enough to disregard that classification for safety reasons.

```
% Use the network's output score for Road as the free space confidence.
roadClassIdx = 4;
freeSpaceConfidence = allScores(:,:,roadClassIdx);

% Display the free space confidence.
figure
imagesc(freeSpaceConfidence)
title('Free Space Confidence Scores')
colorbar
```



Although the initial segmentation result for `Road` pixels showed most pixels on the road were classified correctly, visualizing the scores provides richer detail on the classifier's confidence in those classifications. For example, the confidence decreases as you get closer to the boundary of the car.

Create Bird's-Eye-View Image

The free space estimate is generated in the image space. To facilitate generation of an occupancy grid that is useful for navigation, the free space estimate needs to be transformed into the vehicle coordinate system. This can be done by transforming the free space estimate to a bird's-eye-view image.

To create the bird's-eye-view image, first define the camera sensor configuration. The supporting function listed at the end of this example, `camvidMonoCameraSensor`, returns a `monoCamera` (Automated Driving Toolbox) object representing the monocular camera used to collect the CamVid[1] data. Configuring the `monoCamera` (Automated Driving Toolbox) requires the camera intrinsics and extrinsics, which were estimated using data provided in the CamVid data set. To estimate the camera intrinsics, the function used CamVid checkerboard calibration images and the Camera Calibrator (Computer Vision Toolbox) app. Estimates of the camera extrinsics, such as height and pitch, were derived from the extrinsic data estimated by the authors of the CamVid data set.

```
% Create monoCamera for CamVid data.
sensor = camvidMonoCameraSensor();
```

Given the camera setup, the `birdsEyeView` (Automated Driving Toolbox) object transforms the original image to the bird's-eye view. This object lets you specify the area that you want transformed using vehicle coordinates. Note that the vehicle coordinate units were established by the `monoCamera` (Automated Driving Toolbox) object, when the camera mounting height was specified in meters. For example, if the height was specified in millimeters, the rest of the simulation would use millimeters.

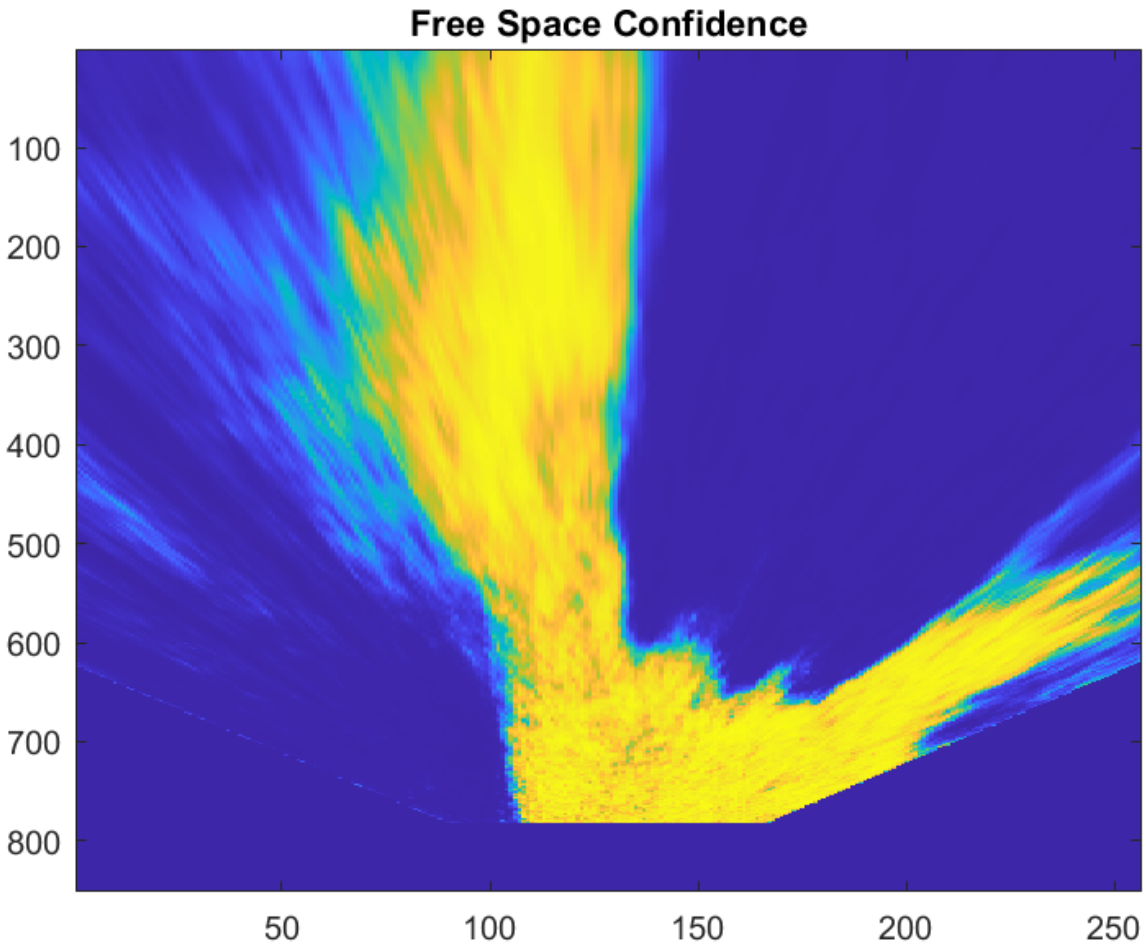
```
% Define bird's-eye-view transformation parameters.
distAheadOfSensor = 20; % in meters, as previously specified in monoCamera height input
spaceToOneSide    = 3;  % look 3 meters to the right and left
bottomOffset      = 0;
outView = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide];

outImageSize = [NaN, 256]; % output image width in pixels; height is chosen automatically to pres

birdsEyeConfig = birdsEyeView(sensor,outView,outImageSize);
```

Generate bird's-eye-view image for the image and free space confidence.

```
% Resize image and free space estimate to size of CamVid sensor.
imageSize = sensor.Intrinsics.ImageSize;
I = imresize(I,imageSize);
freeSpaceConfidence = imresize(freeSpaceConfidence,imageSize);

% Transform image and free space confidence scores into bird's-eye view.
imageBEV = transformImage(birdsEyeConfig,I);
freeSpaceBEV = transformImage(birdsEyeConfig,freeSpaceConfidence);

% Display image frame in bird's-eye view.
figure
imshow(imageBEV)
```

Transform the image into a bird's-eye view and generate the free space confidence.

```
figure
imagesc(freeSpaceBEV)
title('Free Space Confidence')
```



The areas farther away from the sensor are more blurry, due to having fewer pixels and thus requiring greater amount of interpolation.

Create Occupancy Grid Based on Free Space Estimation

Occupancy grids are used to represent a vehicle's surroundings as a discrete grid in vehicle coordinates and are used for path planning. Each cell in the occupancy grid has a value representing the probability of the occupancy of that cell. The estimated free space can be used to fill in values of the occupancy grid.

The procedure to fill the occupancy grid using the free space estimate is as follows:

1. Define the dimensions of the occupancy grid in vehicle coordinates.
2. Generate a set of (X,Y) points for each grid cell. These points are in the vehicle's coordinate system.

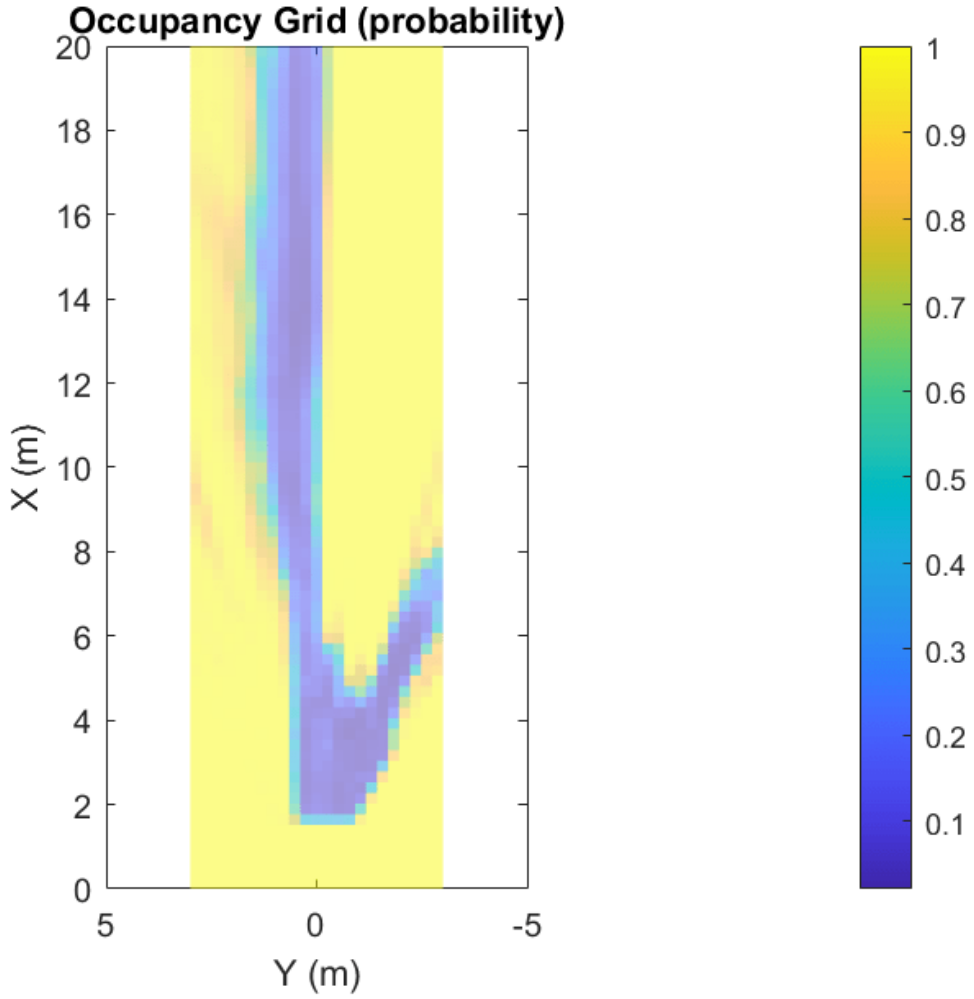
3. Transform the points from the vehicle coordinate space (X,Y) into the bird's-eye-view image coordinate space (x,y) using the `vehicleToImage` (Automated Driving Toolbox) transform.
4. Sample the free space confidence values at (x,y) locations using `griddedInterpolant` to interpolate free space confidence values that are not exactly at pixel centers in the image.
5. Fill the occupancy grid cell with the average free space confidence value for all (x,y) points that correspond to that grid cell.

For brevity, the procedure shown above is implemented in the supporting function, `createOccupancyGridFromFreeSpaceEstimate`, which is listed at the end of this example. Define the dimensions of the occupancy grid in terms of the bird's-eye-view configuration and create the occupancy grid by calling `createOccupancyGridFromFreeSpaceEstimate`.

```
% Define dimensions and resolution of the occupancy grid.
gridX = distAheadOfSensor;
gridY = 2 * spaceToOneSide;
cellSize = 0.25; % in meters to match units used by CamVid sensor

% Create the occupancy grid from the free space estimate.
occupancyGrid = createOccupancyGridFromFreeSpaceEstimate(...
    freeSpaceBEV, birdsEyeConfig, gridX, gridY, cellSize);
```

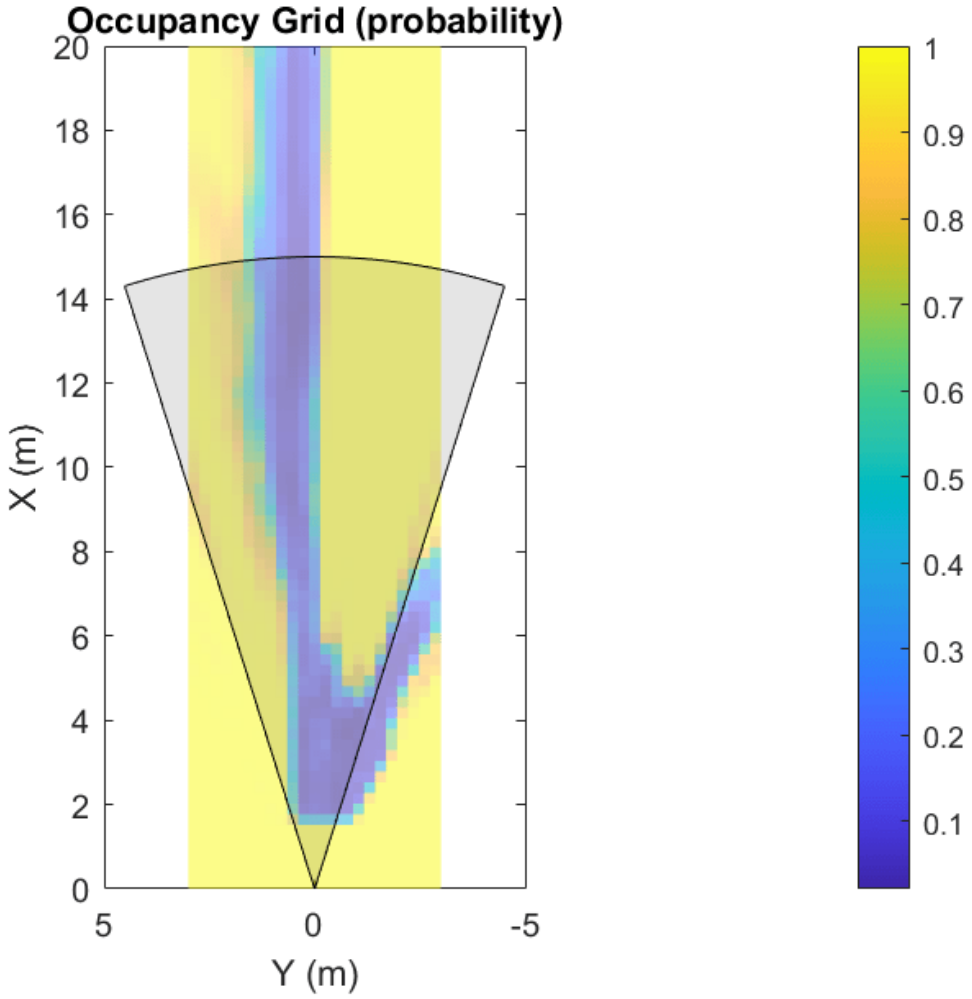
Visualize the occupancy grid using `birdsEyePlot` (Automated Driving Toolbox). Create a `birdsEyePlot` (Automated Driving Toolbox) and add the occupancy grid on top using `pcolor`.

```
% Create bird's-eye plot.
bep = birdsEyePlot('XLimits',[0 distAheadOfSensor],'YLimits', [-5 5]);

% Add occupancy grid to bird's-eye plot.
hold on
[numCellsY,numCellsX] = size(occupancyGrid);
X = linspace(0, gridX, numCellsX);
Y = linspace(-gridY/2, gridY/2, numCellsY);
h = pcolor(X,Y,occupancyGrid);
title('Occupancy Grid (probability)')
colorbar
delete(legend)

% Make the occupancy grid visualization transparent and remove grid lines.
h.FaceAlpha = 0.5;
h.LineStyle = 'none';
```



The bird's-eye plot can also display data from multiple sensors. For example, add the radar coverage area using `coverageAreaPlotter` (Automated Driving Toolbox).

```
% Add coverage area to plot.
caPlotter = coverageAreaPlotter(bep, 'DisplayName', 'Coverage Area');

% Update it with a field of view of 35 degrees and a range of 60 meters
mountPosition = [0 0];
range = 15;
orientation = 0;
fieldOfView = 35;
plotCoverageArea(caPlotter, mountPosition, range, orientation, fieldOfView);
hold off
```



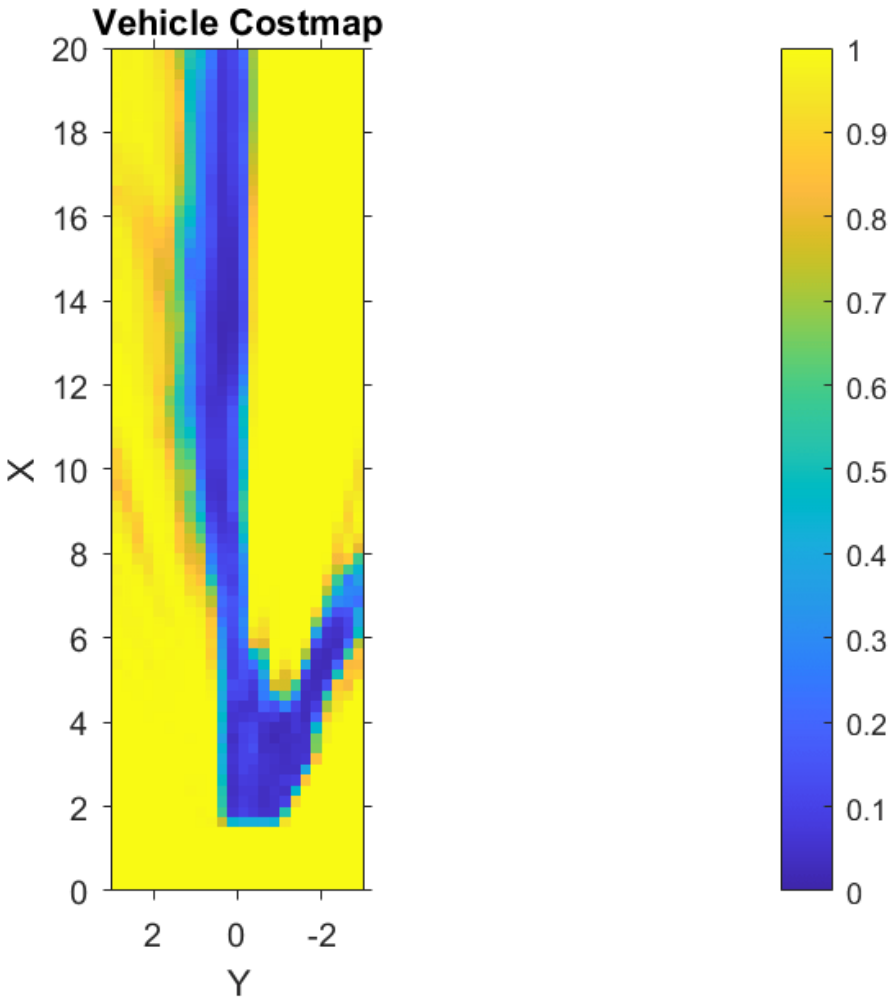
Displaying data from multiple sensors is useful for diagnosing and debugging decisions made by autonomous vehicles.

Create Vehicle Costmap Using the Occupancy Grid

The `vehicleCostmap` (Automated Driving Toolbox) provides functionality to check if locations, in vehicle or world coordinates, are occupied or free. This check is required for any path-planning or decision-making algorithm. Create the `vehicleCostmap` (Automated Driving Toolbox) using the generated `occupancyGrid`.

```
% Create the costmap.
costmap = vehicleCostmap(flipud(occupancyGrid), ...
    'CellSize',cellSize, ...
    'MapLocation',[0,-spaceToOneSide]);
costmap.CollisionChecker.InflationRadius = 0;

% Display the costmap.
figure
plot(costmap,'Inflation','off')
colormap(parula)
```

```
colorbar
title('Vehicle Costmap')

% Orient the costmap so that it lines up with the vehicle coordinate
% system, where the X-axis points in front of the ego vehicle and the
% Y-axis points to the left.
view(gca,-90,90)
```

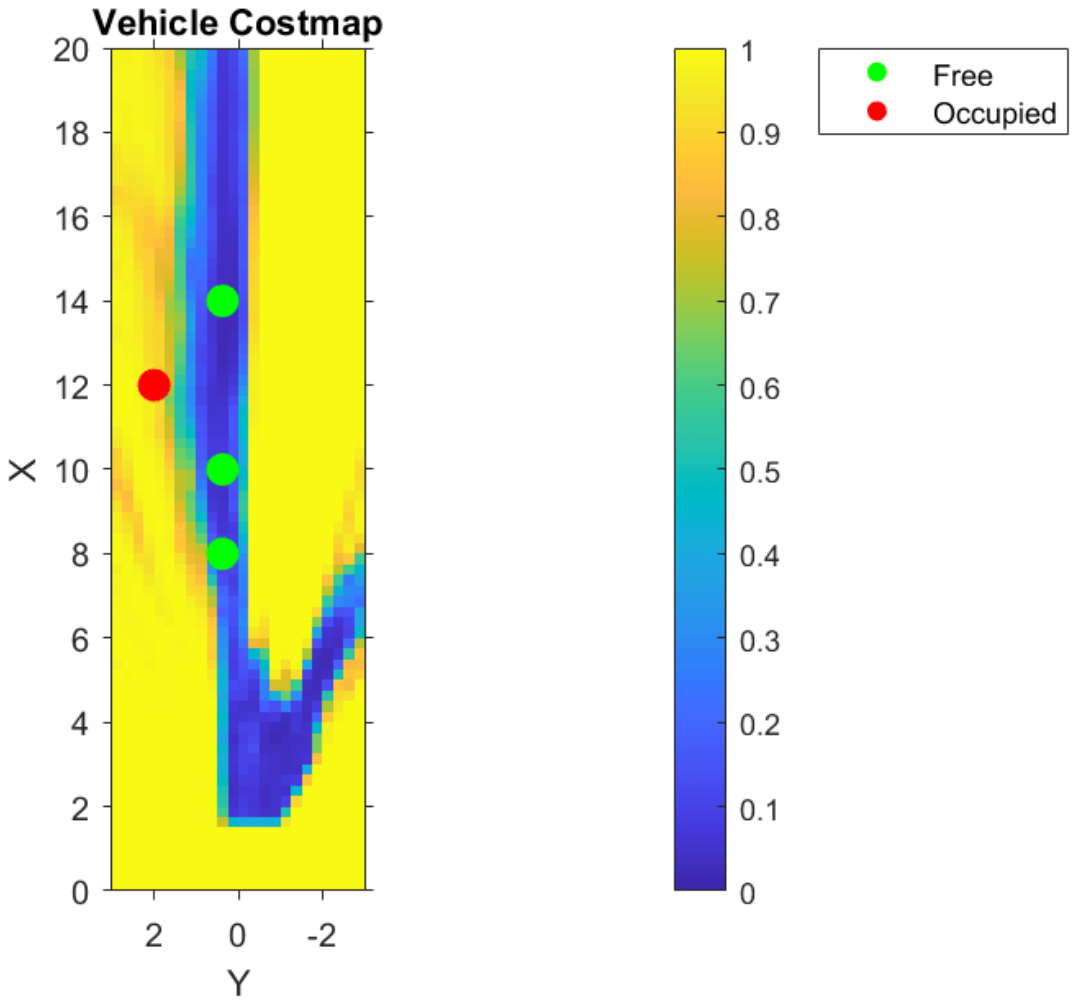


To illustrate how to use the `vehicleCostmap` (Automated Driving Toolbox), create a set of locations in world coordinates. These locations represent a path the vehicle could traverse.

```
% Create a set of locations in vehicle coordinates.
candidateLocations = [
    8 0.375
    10 0.375
    12 2
    14 0.375
    ];
```

Use `checkOccupied` (Automated Driving Toolbox) to check whether each location is occupied or free. Based on the results, a potential path might be impossible to follow because it collides with obstacles defined in the `costmap`.

```
% Check if locations are occupied.
isOccupied = checkOccupied(costmap,candidateLocations);

% Partition locations into free and occupied for visualization purposes.
occupiedLocations = candidateLocations(isOccupied,:);
freeLocations = candidateLocations(~isOccupied,:);

% Display free and occupied points on top of costmap.
hold on
markerSize = 100;
scatter(freeLocations(:,1),freeLocations(:,2),markerSize,'g','filled')
scatter(occupiedLocations(:,1),occupiedLocations(:,2),markerSize,'r','filled');
legend(["Free" "Occupied"])
hold off
```



The use of `occupancyGrid`, `vehicleCostmap` (Automated Driving Toolbox), and `checkOccupied` (Automated Driving Toolbox) shown above illustrate the basic operations used by path planners such

as `pathPlannerRRT` (Automated Driving Toolbox). Learn more about path planning in the "Automated Parking Valet" (Automated Driving Toolbox) example.

References

[1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object Classes in Video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp. 88-97.

Supporting Functions

```
function sensor = camvidMonoCameraSensor()
% Return a monoCamera camera configuration based on data from the CamVid
% data set[1].
%
% The cameraCalibrator app was used to calibrate the camera using the
% calibration images provided in CamVid:
%
% http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/CalibrationSeq_and_Files_0010YU
%
% Calibration pattern grid size is 28 mm.
%
% Camera pitch is computed from camera pose matrices [R t] stored here:
%
% http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/EgoBoost_trax_matFiles.zip

% References
% ----------
% [1] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic Object
% Classes in Video: A high-definition ground truth database." _Pattern Recognition
% Letters_. Vol. 30, Issue 2, 2009, pp. 88-97.

calibrationData = load('camera_params_camvid.mat');

% Describe camera configuration.
focalLength    = calibrationData.cameraParams.FocalLength;
principalPoint = calibrationData.cameraParams.PrincipalPoint;
imageSize      = calibrationData.cameraParams.ImageSize;

% Camera height estimated based on camera setup pictured in [1].
height = 0.5;  % height in meters from the ground

% Camera pitch was computed using camera extrinsics provided in data set.
pitch = 0;  % pitch of the camera, towards the ground, in degrees

camIntrinsics = cameraIntrinsics(focalLength,principalPoint,imageSize);
sensor = monoCamera(camIntrinsics,height,'Pitch',pitch);
end

function occupancyGrid = createOccupancyGridFromFreeSpaceEstimate(...
    freeSpaceBEV,birdsEyeConfig,gridX,gridY,cellSize)
% Return an occupancy grid that contains the occupancy probability over
% a uniform 2-D grid.

% Number of cells in occupancy grid.
numCellsX = ceil(gridX / cellSize);
numCellsY = ceil(gridY / cellSize);

% Generate a set of (X,Y) points for each grid cell. These points are in
```

```
% the vehicle's coordinate system. Start by defining the edges of each grid
% cell.

% Define the edges of each grid cell in vehicle coordinates.
XEdges = linspace(0,gridX,numCellsX);
YEdges = linspace(-gridY/2,gridY/2,numCellsY);

% Next, specify the number of sample points to generate along each
% dimension within a grid cell. Use these to compute the step size in the
% X and Y direction. The step size will be used to shift the edge values of
% each grid to produce points that cover the entire area of a grid cell at
% the desired resolution.

% Sample 20 points from each grid cell. Sampling more points may produce
% smoother estimates at the cost of additional computation.
numSamplePoints = 20;

% Step size needed to sample number of desired points.
XStep = (XEdges(2)-XEdges(1)) / (numSamplePoints-1);
YStep = (YEdges(2)-YEdges(1)) / (numSamplePoints-1);

% Finally, slide the set of points across both dimensions of the grid
% cells. Sample the occupancy probability along the way using
% griddedInterpolant.

% Create griddedInterpolant for sampling occupancy probability. Use 1
% minus the free space confidence to represent the probability of occupancy.
occupancyProb = 1 - freeSpaceBEV;
sz = size(occupancyProb);
[y,x] = ndgrid(1:sz(1),1:sz(2));
F = griddedInterpolant(y,x,occupancyProb);

% Initialize the occupancy grid to zero.
occupancyGrid = zeros(numCellsY*numCellsX,1);

% Slide the set of points XEdges and YEdges across both dimensions of the
% grid cell.
for j = 1:numSamplePoints

    % Increment sample points in the X-direction
    X = XEdges + (j-1)*XStep;

    for i = 1:numSamplePoints

        % Increment sample points in the Y-direction
        Y = YEdges + (i-1)*YStep;

        % Generate a grid of sample points in bird's-eye-view vehicle coordinates
        [XGrid,YGrid] = meshgrid(X,Y);

        % Transform grid of sample points to image coordinates
        xy = vehicleToImage(birdsEyeConfig,[XGrid(:) YGrid(:)]);

        % Clip sample points to lie within image boundaries
        xy = max(xy,1);
        xq = min(xy(:,1),sz(2));
        yq = min(xy(:,2),sz(1));
```

```
            % Sample occupancy probabilities using griddedInterpolant and keep
            % a running sum.
            occupancyGrid = occupancyGrid + F(yq,xq);
        end

    end

    % Determine mean occupancy probability.
    occupancyGrid = occupancyGrid / numSamplePoints^2;
    occupancyGrid = reshape(occupancyGrid,numCellsY,numCellsX);
    end
```

Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

This example shows how to use 3-D simulation data to train a semantic segmentation network and fine-tune it to real-world data using generative adversarial networks (GANs).

This example uses 3-D simulation data generated by Driving Scenario Designer and the Unreal Engine®. For an example showing how to generate such simulation data, see “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” (Automated Driving Toolbox). The 3-D simulation environment generates the images and the corresponding ground truth pixel labels. Using the simulation data avoids the annotation process, which is both tedious and requires a large amount of human effort. However, domain shift models trained on only simulation data do not perform well on real-world data sets. To address this, you can use domain adaptation to fine-tune the trained model to work on a real-world data set.

This example uses AdaptSegNet [1 on page 10-0], a network that adapts the structure of the output segmentation predictions, which look alike irrespective of the input domain. The AdaptSegNet network is based on the GAN model and consists of two networks that are trained simultaneously to maximize the performance of both:

1. Generator — Network trained to generate high-quality segmentation results from real or simulated input images
2. Discriminator — Network that compares and attempts to distinguish whether the segmentation predictions of the generator are from real or simulated data

To fine-tune the AdaptSegNet model for real-world data, this example uses a subset of the CamVid data [2 on page 10-0] and adapts the model to generate high-quality segmentation predictions on the CamVid data.

Download Pretrained Network

Download the pretrained network. The pretrained model allows you to run the entire example without having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
doTraining = false;
if ~doTraining
    pretrainedURL = 'https://ssd.mathworks.com/supportfiles/vision/data/trainedAdaptSegGANNet.ma
    pretrainedFolder = fullfile(tempdir,'pretrainedNetwork');
    pretrainedNetwork = fullfile(pretrainedFolder,'trainedAdaptSegGANNet.mat');
    if ~exist(pretrainedNetwork,'file')
        mkdir(pretrainedFolder);
        disp('Downloading pretrained network (57 MB)...');
        websave(pretrainedNetwork,pretrainedURL);
    end
    pretrained = load(pretrainedNetwork);
    dlnetGenerator = pretrained.dlnetGenerator;
end
```

Download Data Sets

Download the simulation and real data sets by using the `downloadDataset` function, defined in the Supporting Functions section of this example. The `downloadDataset` function downloads the entire CamVid data set and partition the data into training and test sets.

The simulation data set was generated by Driving Scenario Designer. The generated scenarios, which consist of 553 photorealistic images with labels, were rendered by the Unreal Engine. You use this data set to train the model.

The real data set is a subset of the CamVid data set from the University of Cambridge. To adapt the model to real-world data, 69 CamVid images. To evaluate the trained model, you use 368 CamVid images.

The download time depends on your internet connection.

```
simulationDataURL = 'https://ssd.mathworks.com/supportfiles/vision/data/SimulationDrivingDataset
realImageDataURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRa
realLabelDataURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApprove

simulationDataLocation = fullfile(tempdir,'SimulationData');
realDataLocation = fullfile(tempdir,'RealData');
[simulationImagesFolder, simulationLabelsFolder, realImagesFolder, realLabelsFolder, ...
    realTestImagesFolder, realTestLabelsFolder] = ...
    downloadDataset(simulationDataLocation,simulationDataURL,realDataLocation,realImageDataURL,re
```

The downloaded files include the pixel labels for the real domain, but note that you do not use these pixel labels in the training process. This example uses the real domain pixel labels only to calculate the mean intersection over union (IoU) value to evaluate the efficacy of the trained model.

Load Simulation and Real Data

Use `imageDatastore` to load the simulation and real data sets for training. By using an image datastore, you can efficiently load a large collection of images on disk.

```
simData = imageDatastore(simulationImagesFolder);
realData = imageDatastore(realImagesFolder);
```

Preview images from the simulation data set and real data set.

```
simImage = preview(simData);
realImage = preview(realData);
montage({simImage,realImage})
```

The real and simulated images look very different. Consequently, models trained on simulated data and evaluated on real data perform poorly due to domain shift.

Load Pixel-Labeled Images for Simulation Data and Real Data

Load the simulation pixel label image data by using `pixelLabelDatastore` (Computer Vision Toolbox). A pixel label datastore encapsulates the pixel label data and the label ID to a class name mapping.

For this example, specify five classes useful for an automated driving application: road, background, pavement, sky, and car.

```
classes = [
    "Road"
    "Background"
    "Pavement"
    "Sky"
    "Car"
    ];
numClasses = numel(classes);
```

The simulation data set has eight classes. Reduce the number of classes from eight to five by grouping the building, tree, traffic signal, and light classes from the original data set into a single background class. Return the grouped label IDs by using the helper function `simulationPixelLabelIDs`. This helper function is attached to the example as a supporting file.

```
labelIDs = simulationPixelLabelIDs;
```

Use the classes and label IDs to create a pixel label datastore of the simulation data.

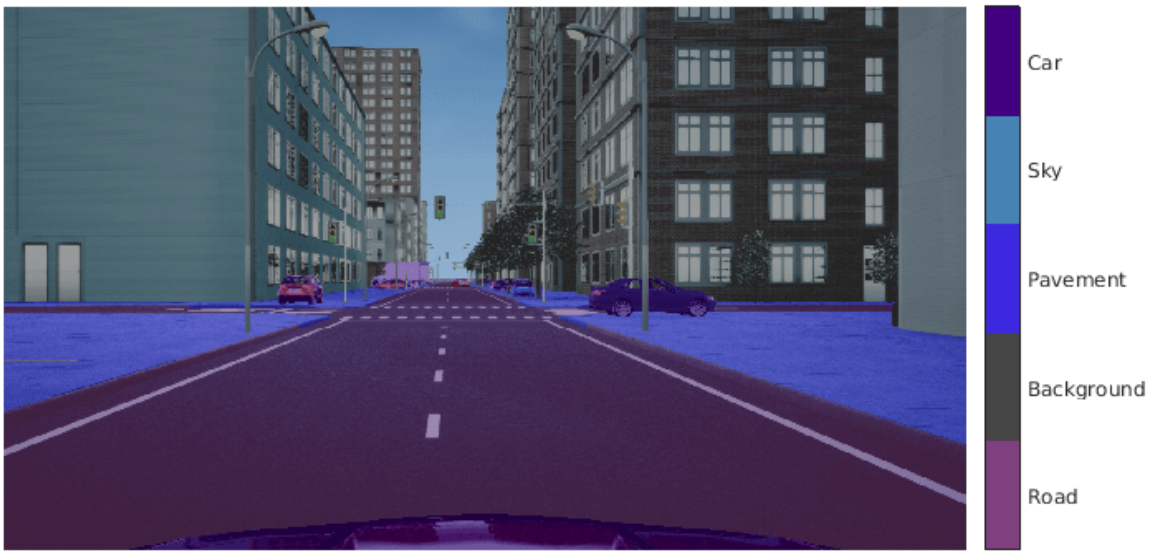
```
simLabels = pixelLabelDatastore(simulationLabelsFolder,classes,labelIDs);
```

Initialize the colormap for the segmented images using the helper function `domainAdaptationColorMap`, defined in the Supporting Functions section.

```
dmap = domainAdaptationColorMap;
```

Preview a pixel-labeled image by overlaying the label on top of the image using the `labeloverlay` (Image Processing Toolbox) function.

```
simImageLabel = preview(simLabels);
overlayImageSimulation = labeloverlay(simImage,simImageLabel,'ColorMap',dmap);
figure
imshow(overlayImageSimulation)
labelColorbar(dmap,classes);
```



Shift the simulation and real data used for training to zero center, to center the data around the origin, by using the `transform` function and the `preprocessData` helper function, defined in the Supporting Functions section.

```
preprocessedSimData = transform(simData, @(simdata)preprocessData(simdata));
preprocessedRealData = transform(realData, @(realdata)preprocessData(realdata));
```

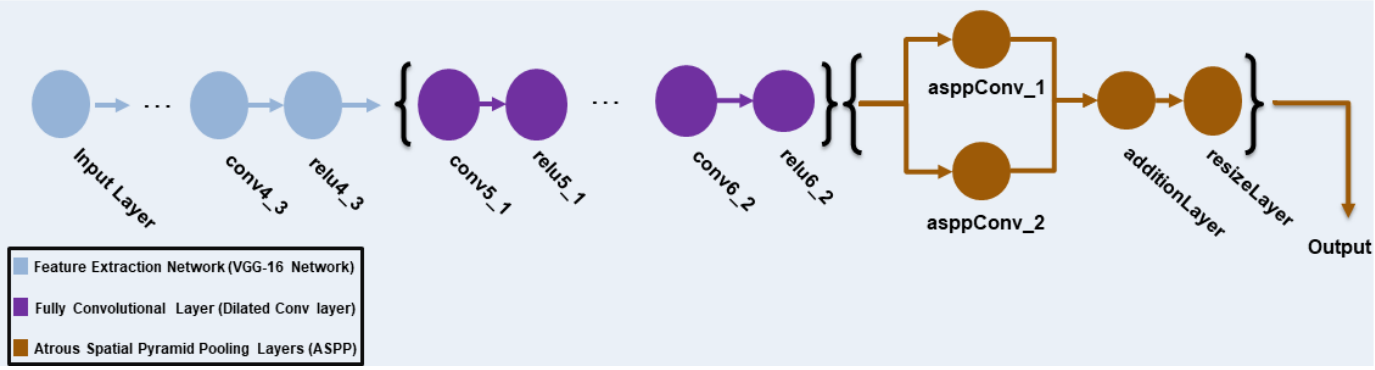
Use the `combine` function to combine the transformed image datastore and pixel label datastores of the simulation domain. The training process does not use the pixel labels of real data.

```
combinedSimData = combine(preprocessedSimData,simLabels);
```

Define AdaptSegNet Generator

This example modifies the VGG-16 network pretrained on ImageNet to a fully convolutional network. To enlarge the receptive fields, dilated convolutional layers with strides of 2 and 4 are added. This makes the output feature map resolution one-eighth of the input size. Atrous spatial pyramid pooling (ASPP) is used to provide multiscale information and is followed by a `resize2dlayer` with an upsampling factor of 8 to resize the output to the size of the input.

The AdaptSegNet generator network used in this example is illustrated in the following diagram.



To get a pretrained VGG-16 network, install the `vgg16`. If the support package is not installed, then the software provides a download link.

```
net = vgg16;
```

To make the VGG-16 network suitable for semantic segmentation, remove all VGG layers after `'relu4_3'`.

```
vggLayers = net.Layers(2:24);
```

Create an image input layer of size 1280-by-720-by-3 for the generator.

```
inputSizeGenerator = [1280 720 3];
inputLayer = imageInputLayer(inputSizeGenerator,'Normalization','None','Name','inputLayer');
```

Create fully convolutional network layers. Use dilation factors of 2 and 4 to enlarge the respective fields.

```
fcnlayers = [
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2],'Name','conv5_1','We
    reluLayer('Name','relu5_1')
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2] ,'Name','conv5_2','W
    reluLayer('Name','relu5_2')
    convolution2dLayer([3 3], 360,'DilationFactor',[2 2],'Padding',[2 2 2 2],'Name','conv5_3','We
    reluLayer('Name','relu5_3')
    convolution2dLayer([3 3], 480,'DilationFactor',[4 4],'Padding',[4 4 4 4],'Name','conv6_1','We
    reluLayer('Name','relu6_1')
    convolution2dLayer([3 3], 480,'DilationFactor',[4 4],'Padding',[4 4 4 4] ,'Name','conv6_2','W
    reluLayer('Name','relu6_2')
    ];
```

Combine the layers and create the layer graph.

```
layers = [
    inputLayer
    vggLayers
    fcnlayers
    ];
lgraph = layerGraph(layers);
```

ASPP is used to provide multiscale information. Add the ASPP module to the layer graph with a filter size equal to the number of channels by using the `addASPPToNetwork` helper function, defined in the Supporting Functions section.

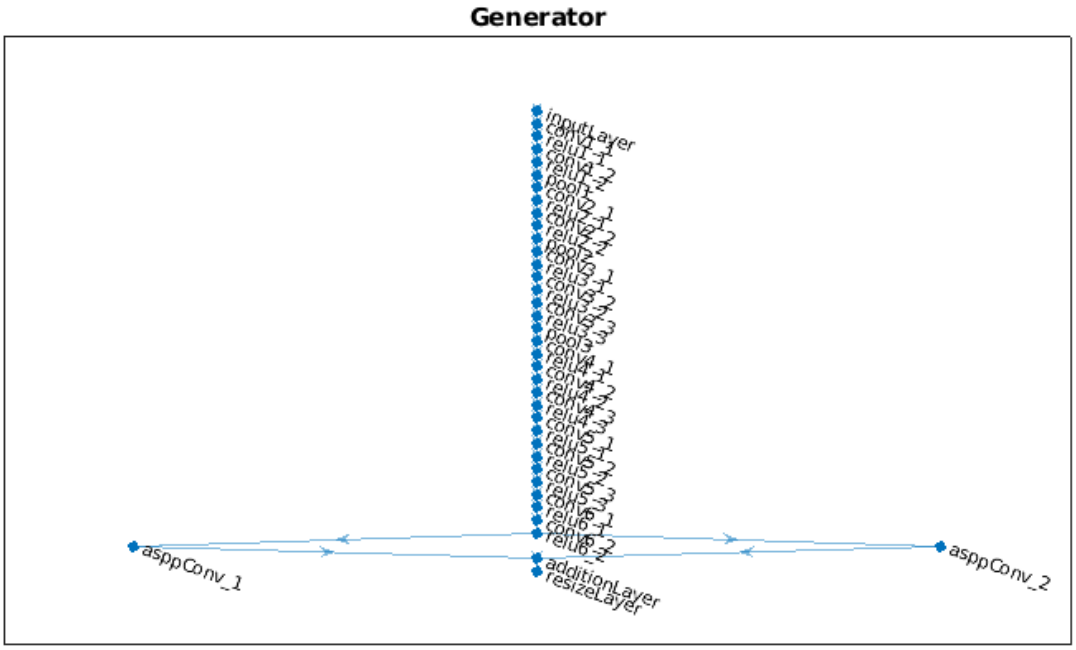
```
lgraph  = addASPPToNetwork(lgraph, numClasses);
```

Apply `resize2dLayer` with an upsampling factor of 8 to make the output match the size of the input.

```
upSampleLayer = resize2dLayer('Scale',8,'Method','bilinear','Name','resizeLayer');
lgraphGenerator = addLayers(lgraph,upSampleLayer);
lgraphGenerator = connectLayers(lgraphGenerator,'additionLayer','resizeLayer');
```

Visualize the generator network in a plot.

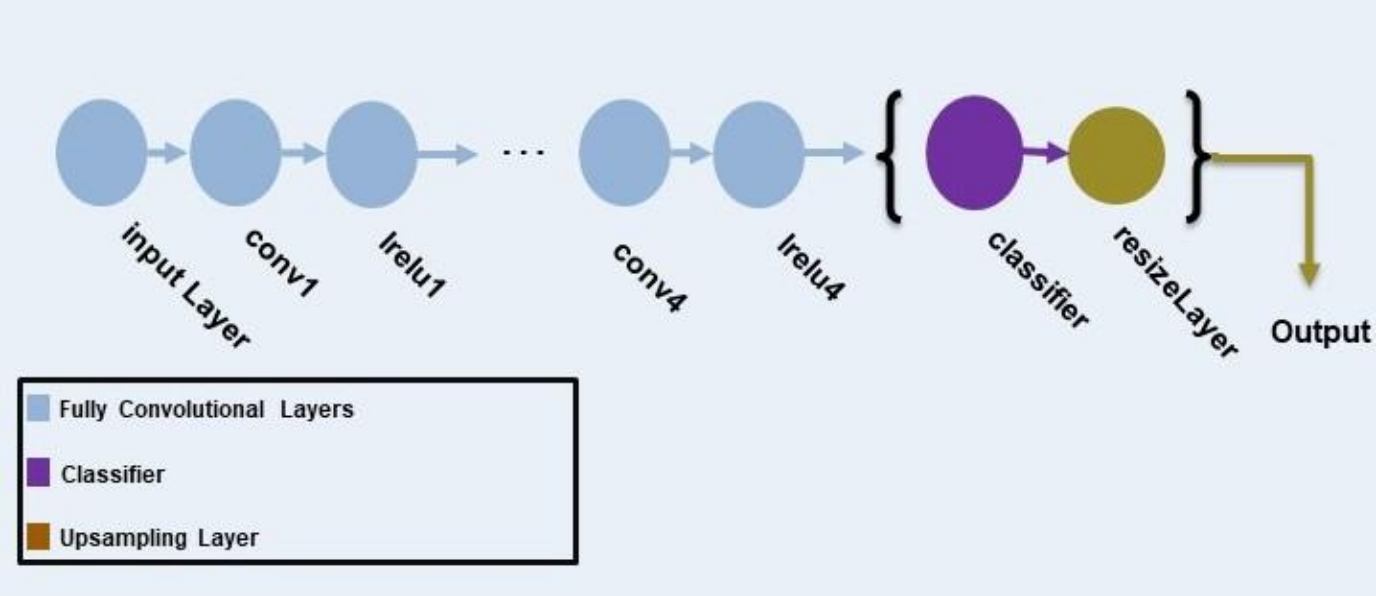
```
plot(lgraphGenerator)
title("Generator")
```



Define AdaptSeg Discriminator

The discriminator network consists of five convolutional layers with a kernel size of 3 and a stride of 2, where the number of channels is {64, 128, 256, 512, 1}. Each layer is followed by a leaky ReLU layer parameterized by a scale of 0.2, except for the last layer. `resize2dLayer` is used to resize the output of the discriminator. Note that this example does not use batch normalization, as the discriminator is jointly trained with the segmentation network using a small batch size.

The AdaptSegNet discriminator network in this example is illustrated in the following diagram.



Create an image input layer of size 1280-by-720-by-`numClasses` that takes in the segmentation predictions of the simulation and real domains.

```
inputSizeDiscriminator = [1280 720 numClasses];
```

Create fully convolutional layers and generate the discriminator layer graph.

```
% Factor for number of channels in convolution layer.
numChannelsFactor = 64;

% Scale factor to resize the output of the discriminator.
resizeScale = 64;

% Scalar multiplier for leaky ReLU layers.
leakyReLUScale = 0.2;

% Create the layers of the discriminator.
layers = [
    imageInputLayer(inputSizeDiscriminator,'Normalization','none','Name','inputLayer')
    convolution2dLayer(3,numChannelsFactor,'Stride',2,'Padding',1,'Name','conv1','WeightsInitial
    leakyReluLayer(leakyReLUScale,'Name','lrelu1')
    convolution2dLayer(3,numChannelsFactor*2,'Stride',2,'Padding',1,'Name','conv2','WeightsInitia
    leakyReluLayer(leakyReLUScale,'Name','lrelu2')
    convolution2dLayer(3,numChannelsFactor*4,'Stride',2,'Padding',1,'Name','conv3','WeightsInitia
    leakyReluLayer(leakyReLUScale,'Name','lrelu3')
    convolution2dLayer(3,numChannelsFactor*8,'Stride',2,'Padding',1,'Name','conv4','WeightsInitia
    leakyReluLayer(leakyReLUScale,'Name','lrelu4')
    convolution2dLayer(3,1,'Stride',2,'Padding',1,'Name','classifer','WeightsInitializer','narrow
    resize2dLayer('Scale', resizeScale,'Method','bilinear','Name','resizeLayer');
    ];

% Create the layer graph of the discriminator.
lgraphDiscriminator  = layerGraph(layers);
```

Visualize the discriminator network in a plot.

```
plot(lgraphDiscriminator)
title("Discriminator")
```

Specify Training Options

Specify these training options.

- Set the total number of iterations to `5000`. By doing so, you train the network for around 10 epochs.
- Set the learning rate for the generator to `2.5e-4`.
- Set the learning rate for the discriminator to `1e-4`.
- Set the L2 regularization factor to `0.0005`.
- The learning rate exponentially decreases based on the formula *learningrate* × $\left[\frac{iteration}{total\ \ iterations}\right]^{power}$. This decrease helps to stabilize the gradients at higher iterations. Set the power to `0.9`.
- Set the weight of the adversarial loss to `0.001`.
- Initialize the velocity of the gradient as `[ ]`. This value is used by SGDM to store the velocity of the gradients.
- Initialize the moving average of the parameter gradients as `[ ]`. This value is used by Adam initializer to store the average of parameter gradients.
- Initialize the moving average of squared parameter gradients as `[ ]`. This value is used by Adam initializer to store the average of the squared parameter gradients.
- Set the mini-batch size to `1`.

```
numIterations = 5000;
learnRateGenBase = 2.5e-4;
```

```
learnRateDisBase = 1e-4;
l2Regularization = 0.0005;
power = 0.9;
lamdaAdv = 0.001;
vel= [];
averageGrad = [];
averageSqGrad = [];
miniBatchSize = 1;
```

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. To automatically detect if you have a GPU available, set `executionEnvironment` to `"auto"`. If you do not have a GPU, or do not want to use one for training, set `executionEnvironment` to `"cpu"`. To ensure the use of a GPU for training, set `executionEnvironment` to `"gpu"`. For information about the supported compute capabilities, see "GPU Support by Release" (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Create the `minibatchqueue` object from the combined datastore of the simulation domain.

```
mbqTrainingDataSimulation =  minibatchqueue(combinedSimData,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

Create the `minibatchqueue` object from the input image datastore of the real domain.

```
mbqTrainingDataReal = minibatchqueue(preprocessedRealData,"MiniBatchSize",miniBatchSize, ...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

Train Model

Train the model using a custom training loop. The helper function `modelGradients`, defined in the Supporting Functions section of this example, calculate the gradients and losses for the generator and discriminator. Create the training progress plot using `configureTrainingLossPlotter`, attached to this example as a supporting file, and update the training progress using `updateTrainingPlots`. Loop over the training data and update the network parameters at each iteration.

For each iteration:

- Read the image and label information from the `minibatchqueue` object of the simulation data using the `next` function.
- Read the image information from the `minibatchqueue` object of the real data using the `next` function.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` helper function, defined in the Supporting Functions section. `modelGradients` returns the gradients of the loss with respect to the learnable parameters.
- Update the generator network parameters using the `sgdmupdate` function.
- Update the discriminator network parameters using the `adamupdate` function.
- Update the training progress plot for every iteration and display various computed losses.

```
if doTraining

    % Create the dlnetwork object of the generator.
    dlnetGenerator = dlnetwork(lgraphGenerator);
```

```
        % Create the dlnetwork object of the discriminator.
        dlnetDiscriminator = dlnetwork(lgraphDiscriminator);

        % Create the subplots for the generator and discriminator loss.
        fig = figure;
        [generatorLossPlotter, discriminatorLossPlotter] = configureTrainingLossPlotter(fig);

        % Loop through the data for the specified number of iterations.
        for iter = 1:numIterations

            % Reset the minibatchqueue of simulation data.
            if ~hasdata(mbqTrainingDataSimulation)
                reset(mbqTrainingDataSimulation);
            end

            % Retrieve the next mini-batch of simulation data and labels.
            [dlX,label] = next(mbqTrainingDataSimulation);

            % Reset the minibatchqueue of real data.
            if ~hasdata(mbqTrainingDataReal)
                reset(mbqTrainingDataReal);
            end

            % Retrieve the next mini-batch of real data.
            dlZ = next(mbqTrainingDataReal);

            % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
            [gradientGenerator,gradientDiscriminator, lossSegValue, lossAdvValue, lossDisValue] = ..
                dlfeval(@modelGradients,dlnetGenerator,dlnetDiscriminator,dlX,dlZ,label,lamdaAdv);

            % Apply L2 regularization.
            gradientGenerator  = dlupdate(@(g,w) g + l2Regularization*w, gradientGenerator, dlnetGene

            % Adjust the learning rate.
            learnRateGen = piecewiseLearningRate(iter,learnRateGenBase,numIterations,power);
            learnRateDis = piecewiseLearningRate(iter,learnRateDisBase,numIterations,power);

             % Update the generator network learnable parameters using the SGDM optimizer.
            [dlnetGenerator.Learnables, vel] = ...
                sgdmupdate(dlnetGenerator.Learnables,gradientGenerator,vel,learnRateGen);

             % Update the discriminator network learnable parameters using the Adam optimizer.
            [dlnetDiscriminator.Learnables, averageGrad, averageSqGrad] = ...
                adamupdate(dlnetDiscriminator.Learnables,gradientDiscriminator,averageGrad,averageSq

            % Update the training plot with loss values.
            updateTrainingPlots(generatorLossPlotter,discriminatorLossPlotter,iter, ...
                double(gather(extractdata(lossSegValue + lamdaAdv * lossAdvValue))),double(gather(ex

        end

        % Save the trained model.
        save('trainedAdaptSegGANNet.mat','dlnetGenerator');
    end
```

The discriminator can now identify whether the input is from the simulation or real domain. In turn, the generator can now generate segmentation predictions that are similar across the simulation and real domains.

Evaluate Model on Real Test Data

Evaluate the performance of the trained AdaptSegNet network by computing the mean IoU for the test data predictions.

Load the test data using `imageDatastore`.

```
realTestData = imageDatastore(realTestImagesFolder);
```

The CamVid data set has 32 classes. Use the `realpixelLabelIDs` helper function to reduce the number of classes to five, as for the simulation data set. The `realpixelLabelIDs` helper function is attached to this example as a supporting file.

```
labelIDs = realPixelLabelIDs;
```

Use `pixelLabelDatastore` (Computer Vision Toolbox) to load the ground truth label images for the test data.

```
realTestLabels = pixelLabelDatastore(realTestLabelsFolder,classes,labelIDs);
```

Shift the data to zero center to center the data around the origin, as for the training data, by using the `transform` function and the `preprocessData` helper function, defined in the Supporting Functions section.

```
preprocessedRealTestData = transform(realTestData, @(realtestdata)preprocessData(realtestdata));
```

Use `combine` to combine the transformed image datastore and pixel label datastores of the real test data.

```
combinedRealTestData = combine(preprocessedRealTestData,realTestLabels);
```

Create the `minibatchqueue` object from the combined datastore of the test data. Set `"MiniBatchSize"` to `1` for ease of evaluating the metrics.

```
mbqimdsTest = minibatchqueue(combinedRealTestData,"MiniBatchSize",1,...
    "MiniBatchFormat","SSCB","OutputEnvironment",executionEnvironment);
```

To generate the confusion matrix cell array, use the helper function `predictSegmentationLabelsOnTestSet` on `minibatchqueue` object of test data. The helper function `predictSegmentationLabelsOnTestSet` is listed below in Supporting Functions section.

```
imageSetConfusionMat = predictSegmentationLabelsOnTestSet(dlnetGenerator,mbqimdsTest);
```

Use `evaluateSemanticSegmentation` (Computer Vision Toolbox) to measure semantic segmentation metrics on the test set confusion matrix.

```
metrics = evaluateSemanticSegmentation(imageSetConfusionMat,classes,'Verbose',false);
```

To see the data set level metrics, inspect `metrics.DataSetMetrics`.

```
metrics.DataSetMetrics
```

```
ans=1×4 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU
```

```
    _____________    _____________    _______    ___________

      0.86883            0.769        0.64487      0.78026
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the per-class metrics using `metrics.ClassMetrics`.

```
metrics.ClassMetrics
```

```
ans=5×2 table
                 Accuracy      IoU
                 ________    _______

    Road          0.9147     0.81301
    Background    0.93418    0.85518
    Pavement      0.33373    0.27105
    Sky           0.82652    0.81109
    Car           0.83586    0.47399
```

The data set performance is good, but the class metrics show that the car and pavement classes are not segmented well. Training the network using additional data can yield improved results.

Segment Image

Run the trained network on one test image to check the segmented output prediction.

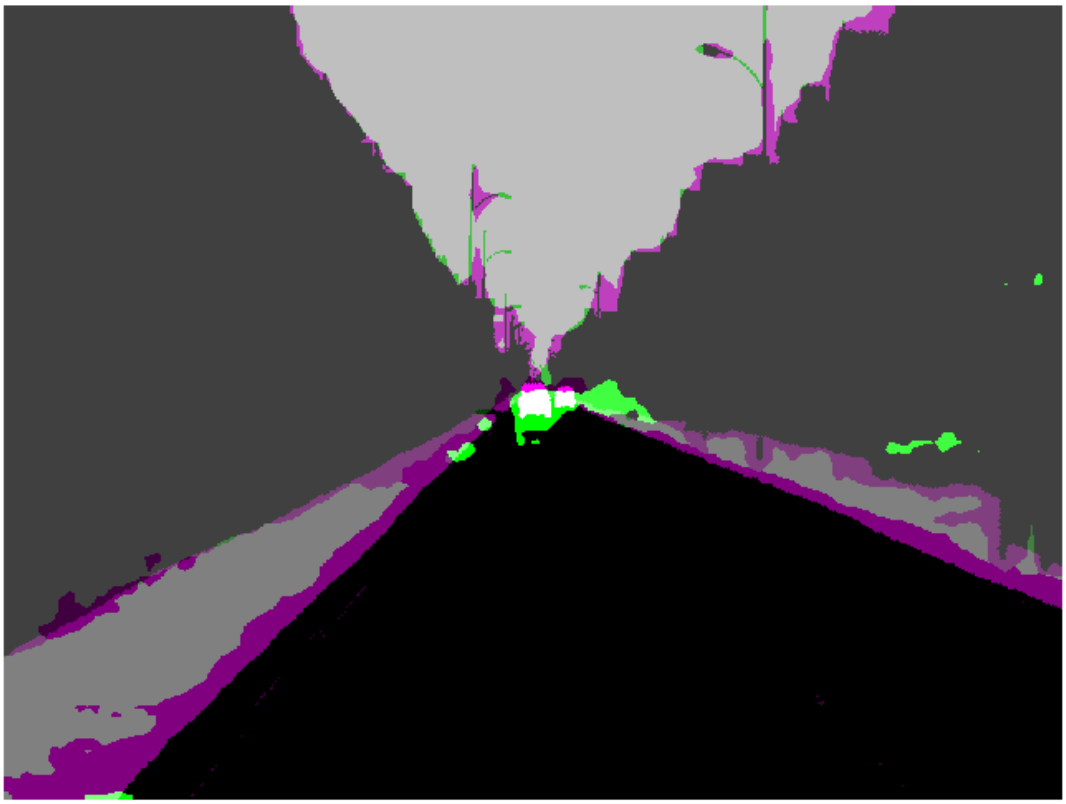
```
% Read the image from the test data.
data = readimage(realTestData,350);

% Perform the preprocessing step of zero shift on the image.
processeddata = preprocessData(data);

% Convert the data to dlarray.
processeddata = dlarray(processeddata,'SSCB');

% Predict the output of the network.
[genPrediction, ~] = forward(dlnetGenerator,processeddata);

% Get the label, which is the index with the maximum value in the channel dimension.
[~, labels] = max(genPrediction,[],3);

% Overlay the predicted labels on the image.
segmentedImage = labeloverlay(data,uint8(gather(extractdata(labels))),'Colormap',dmap);
```

Display the results.

```
figure
imshow(segmentedImage);
labelColorbar(dmap,classes);
```

Compare the label results with the expected ground truth stored in `realTestLabels`. The green and magenta regions highlight areas where the segmentation results differ from the expected ground truth.

```
expectedResult = readimage(realTestLabels,350);
actual = uint8(gather(extractdata(labels)));
expected = uint8(expectedResult);
figure
imshowpair(actual,expected)
```



Visually, the semantic segmentation results overlap well for the road, sky, and building classes. However, the results do not overlap well for the car and pavement classes.

Supporting Functions

Model Gradients Function

The helper function `modelGradients` calculates the gradients and adversarial loss for the generator and discriminator. The function also calculates the segmentation loss for the generator and the cross-entropy loss for the discriminator. As no state information is required to be remembered between the iterations for both generator and discriminator networks, the states are not updated.

```
function [gradientGenerator, gradientDiscriminator, lossSegValue, lossAdvValue, lossDisValue] = m

% Labels for adversarial training.
simulationLabel = 0;
realLabel = 1;

% Extract the predictions of the simulation from the generator.
[genPredictionSimulation, ~] = forward(dlnetGenerator,dlX);

% Compute the generator loss.
lossSegValue = segmentationLoss(genPredictionSimulation,label);
```

```
% Extract the predictions of the real data from the generator.
[genPredictionReal, ~] = forward(dlnetGenerator,dlZ);

% Extract the softmax predictions of the real data from the discriminator.
disPredictionReal = forward(dlnetDiscriminator,softmax(genPredictionReal));

% Create a matrix of simulation labels of real prediction size.
Y = simulationLabel * ones(size(disPredictionReal));

% Compute the adversarial loss to make the real distribution close to the simulation label.
lossAdvValue = mse(disPredictionReal,Y)/numel(Y(:));

% Compute the gradients of the generator with regard to loss.
gradientGenerator = dlgradient(lossSegValue + lamdaAdv*lossAdvValue,dlnetGenerator.Learnables);

% Extract the softmax predictions of the simulation from the discriminator.
disPredictionSimulation = forward(dlnetDiscriminator,softmax(genPredictionSimulation));

% Create a matrix of simulation labels of simulation prediction size.
Y = simulationLabel * ones(size(disPredictionSimulation));

% Compute the discriminator loss with regard to simulation class.
lossDisValueSimulation = mse(disPredictionSimulation,Y)/numel(Y(:));

% Extract the softmax predictions of the real data from the discriminator.
disPredictionReal = forward(dlnetDiscriminator,softmax(genPredictionReal));

% Create a matrix of real labels of real prediction size.
Y = realLabel * ones(size(disPredictionReal));

% Compute the discriminator loss with regard to real class.
lossDisValueReal = mse(disPredictionReal,Y)/numel(Y(:));

% Compute the total discriminator loss.
lossDisValue = lossDisValueSimulation + lossDisValueReal;

% Compute the gradients of the discriminator with regard to loss.
gradientDiscriminator = dlgradient(lossDisValue,dlnetDiscriminator.Learnables);

end
```

Segmentation Loss Function

The helper function `segmentationLoss` computes the feature segmentation loss, which is defined as the cross-entropy loss for the generator using the simulation data and its respective ground truth. The helper function computes the loss by using the `crossentropy` function.

```
function loss = segmentationLoss(predict, target)

% Generate the one-hot encodings of the ground truth.
oneHotTarget = onehotencode(categorical(extractdata(target)),4);

% Convert the one-hot encoded data to dlarray.
oneHotTarget = dlarray(oneHotTarget,'SSBC');

% Compute the softmax output of the predictions.
predictSoftmax = softmax(predict);
```

```
% Compute the cross-entropy loss.
loss =  crossentropy(predictSoftmax,oneHotTarget,'TargetCategories','exclusive')/(numel(oneHotTa
end
```

The helper function `downloadDataset` downloads both the simulation and real data sets from the specified URLs to the specified folder locations if they do not exist. The function returns the paths of the simulation, real training data, and real testing data. The function downloads the entire CamVid data set and partition the data into training and test sets using the `subsetCamVidDatasetFileNames` mat file, attached to the example as a supporting file.

```
function [simulationImagesFolder, simulationLabelsFolder, realImagesFolder, realLabelsFolder,...
    realTestImagesFolder, realTestLabelsFolder] = ...
    downloadDataset(simulationDataLocation, simulationDataURL, realDataLocation, realImageDataURI

% Build the training image and label folder location for simulation data.
simulationDataZip = fullfile(simulationDataLocation,'SimulationDrivingDataset.zip');

% Get the simulation data if it does not exist.
if ~exist(simulationDataZip,'file')
    mkdir(simulationDataLocation)

    disp('Downloading the simulation data');
    websave(simulationDataZip,simulationDataURL);
    unzip(simulationDataZip,simulationDataLocation);
end

simulationImagesFolder = fullfile(simulationDataLocation,'SimulationDrivingDataset','images');
simulationLabelsFolder = fullfile(simulationDataLocation,'SimulationDrivingDataset','labels');

camVidLabelsZip = fullfile(realDataLocation,'CamVidLabels.zip');
camVidImagesZip = fullfile(realDataLocation,'CamVidImages.zip');

if ~exist(camVidLabelsZip,'file') || ~exist(camVidImagesZip,'file')
    mkdir(realDataLocation)

    disp('Downloading 16 MB CamVid dataset labels...');
    websave(camVidLabelsZip, realLabelDataURL);
    unzip(camVidLabelsZip, fullfile(realDataLocation,'CamVidLabels'));

    disp('Downloading 587 MB CamVid dataset images...');
    websave(camVidImagesZip, realImageDataURL);
    unzip(camVidImagesZip, fullfile(realDataLocation,'CamVidImages'));
end

% Build the training image and label folder location for real data.
realImagesFolder = fullfile(realDataLocation,'train','images');
realLabelsFolder = fullfile(realDataLocation,'train','labels');

% Build the testing image and label folder location for real data.
realTestImagesFolder = fullfile(realDataLocation,'test','images');
realTestLabelsFolder = fullfile(realDataLocation,'test','labels');

% Partition the data into training and test sets if they do not exist.
if ~exist(realImagesFolder,'file') || ~exist(realLabelsFolder,'file') || ...
        ~exist(realTestImagesFolder,'file') || ~exist(realTestLabelsFolder,'file')


    mkdir(realImagesFolder);
```

```
    mkdir(realLabelsFolder);
    mkdir(realTestImagesFolder);
    mkdir(realTestLabelsFolder);

    % Load the mat file that has the names for testing and training.
    partitionNames = load('subsetCamVidDatasetFileNames.mat');

    % Extract the test images names.
    imageTestNames = partitionNames.imageTestNames;

    % Remove the empty cells.
    imageTestNames = imageTestNames(~cellfun('isempty',imageTestNames));

    % Extract the test labels names.
    labelTestNames = partitionNames.labelTestNames;

    % Remove the empty cells.
    labelTestNames = labelTestNames(~cellfun('isempty',labelTestNames));

    % Copy the test images to the respective folder.
    for i = 1:size(imageTestNames,1)
        labelSource = fullfile(realDataLocation,'CamVidLabels',labelTestNames(i));
        imageSource = fullfile(realDataLocation,'CamVidImages','701_StillsRaw_full',imageTestName
        copyfile(imageSource{1}, realTestImagesFolder);
        copyfile(labelSource{1}, realTestLabelsFolder);
    end

    % Extract the train images names.
    imageTrainNames = partitionNames.imageTrainNames;

    % Remove the empty cells.
    imageTrainNames = imageTrainNames(~cellfun('isempty',imageTrainNames));

    % Extract the train labels names.
    labelTrainNames = partitionNames.labelTrainNames;

    % Remove the empty cells.
    labelTrainNames = labelTrainNames(~cellfun('isempty',labelTrainNames));

    % Copy the train images to the respective folder.
    for i = 1:size(imageTrainNames,1)
        labelSource = fullfile(realDataLocation,'CamVidLabels',labelTrainNames(i));
        imageSource = fullfile(realDataLocation,'CamVidImages','701_StillsRaw_full',imageTrainNa
        copyfile(imageSource{1},realImagesFolder);
        copyfile(labelSource{1},realLabelsFolder);
    end
end
end
```

The helper function `addASPPToNetwork` creates the atrous spatial pyramid pooling (ASPP) layers and adds them to the input layer graph. The function returns the layer graph with ASPP layers connected to it.

```
function lgraph  = addASPPToNetwork(lgraph, numClasses)

% Define the ASPP dilation factors.
asppDilationFactors = [6,12];
```

```
% Define the ASPP filter sizes.
asppFilterSizes = [3,3];

% Extract the last layer of the layer graph.
lastLayerName = lgraph.Layers(end).Name;

% Define the addition layer.
addLayer = additionLayer(numel(asppDilationFactors),'Name','additionLayer');

% Add the addition layer to the layer graph.
lgraph = addLayers(lgraph,addLayer);

% Create the ASPP layers connected to the addition layer
% and connect the layer graph.
for i = 1: numel(asppDilationFactors)
    asppConvName = "asppConv_" + string(i);
    branchFilterSize = asppFilterSizes(i);
    branchDilationFactor = asppDilationFactors(i);
    asspLayer  = convolution2dLayer(branchFilterSize, numClasses,'DilationFactor', branchDilatio
        'Padding','same','Name',asppConvName,'WeightsInitializer','narrow-normal','BiasInitializ
    lgraph = addLayers(lgraph,asspLayer);
    lgraph = connectLayers(lgraph,lastLayerName,asppConvName);
    lgraph = connectLayers(lgraph,asppConvName,strcat(addLayer.Name,'/',addLayer.InputNames{i}))
end
end
```

The helper function `predictSegmentationLabelsOnTestSet` calculates the confusion matrix of the predicted and ground truth labels using the `segmentationConfusionMatrix` (Computer Vision Toolbox) function.

```
function confusionMatrix =  predictSegmentationLabelsOnTestSet(net, minbatchTestData)

confusionMatrix = {};
i = 1;
while hasdata(minbatchTestData)

    % Use next to retrieve a mini-batch from the datastore.
    [dlX, gtlabels] = next(minbatchTestData);

    % Predict the output of the network.
    [genPrediction, ~] = forward(net,dlX);

    % Get the label, which is the index with maximum value in the channel dimension.
    [~, labels] = max(genPrediction,[],3);

    % Get the confusion matrix of each image.
    confusionMatrix{i}  = segmentationConfusionMatrix(double(gather(extractdata(labels))),double

    i = i+1;
end

confusionMatrix = confusionMatrix';

end
```

The helper function `piecewiseLearningRate` computes the current learning rate based on the iteration number.

```
function lr = piecewiseLearningRate(i, baseLR, numIterations, power)

fraction = i/numIterations;
factor = (1 - fraction)^power * 1e1;
lr = baseLR * factor;

end
```

The helper function `preprocessData` performs a zero center shift by subtracting the number of the image channels by the respective mean.

```
function data = preprocessData(data)

% Extract respective channels.
rc = data(:,:,1);
gc = data(:,:,2);
bc = data(:,:,3);

% Compute the respective channel means.
r = mean(rc(:));
g = mean(gc(:));
b = mean(bc(:));

% Shift the data by the mean of respective channel.
data = single(data) - single(shiftdim([r g b],-1));
end
```

References

[1] Tsai, Yi-Hsuan, Wei-Chih Hung, Samuel Schulter, Kihyuk Sohn, Ming-Hsuan Yang, and Manmohan Chandraker. “Learning to Adapt Structured Output Space for Semantic Segmentation.” In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 7472–81. Salt Lake City, UT: IEEE, 2018. https://doi.org/10.1109/CVPR.2018.00780.

[2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. “Semantic Object Classes in Video: A High-Definition Ground Truth Database.” *Pattern Recognition Letters* 30, no. 2 (January 2009): 88–97. https://doi.org/10.1016/j.patrec.2008.04.005.

Lidar Examples

Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning

This example shows how to train a PointNet++ deep learning network to perform semantic segmentation on aerial lidar data.

Lidar data acquired from airborne laser scanning systems is used in applications such as topographic mapping, city modeling, biomass measurement, and disaster management. Extracting meaningful information from this data requires semantic segmentation, a process where each point in the point cloud is assigned a unique class label.

In this example, you train a PointNet++ network to perform semantic segmentation by using the Dayton Annotated Lidar Earth Scan (DALES) dataset [1 on page 11-0]. The dataset contains scenes of dense, labeled aerial lidar data from urban, suburban, rural, and commercial settings. The dataset provides semantic segmentation labels for 8 classes such as buildings, cars, trucks, poles, power lines, fences, ground, and vegetation.

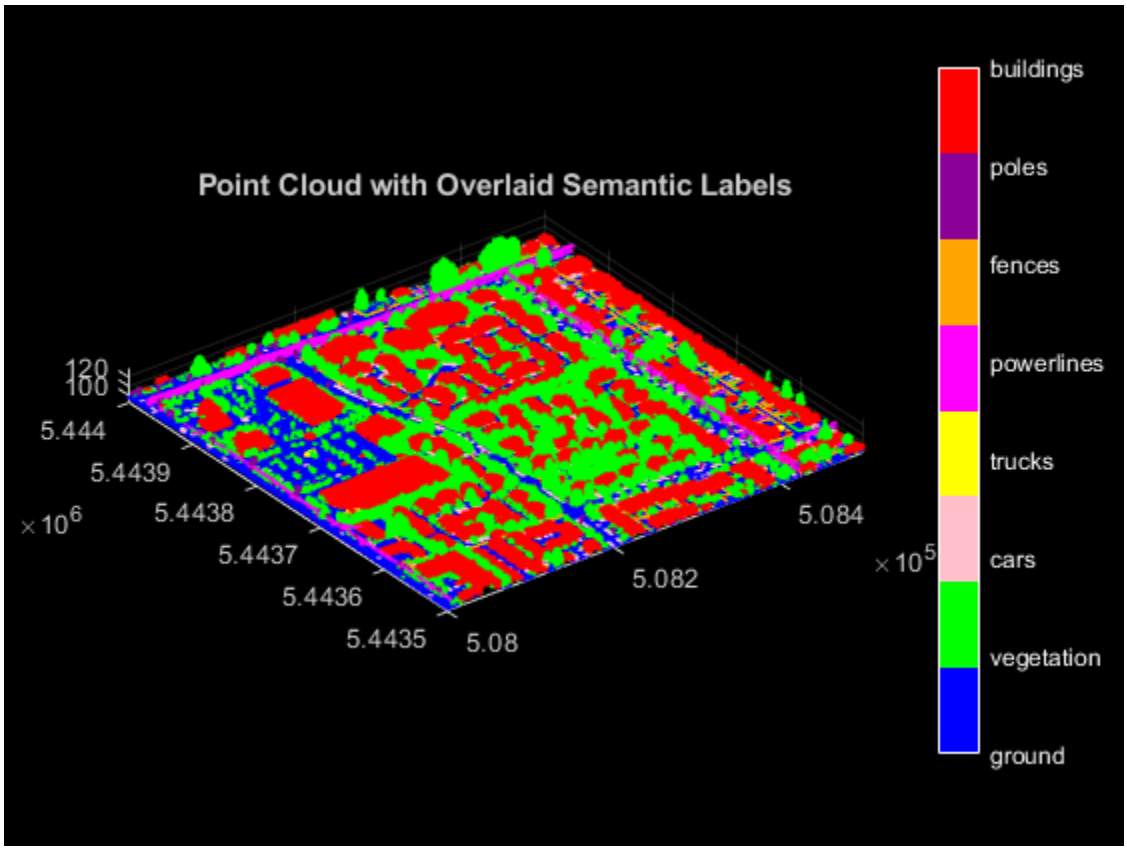
Load DALES Data

The DALES dataset contains 40 scenes of aerial lidar data. Out of the 40 scenes, 29 scenes are used for training and the remaining 11 scenes are used for testing. Each pixel in the data has a class label. Follow the instructions on the DALES website to download the dataset to the folder specified by the `dataFolder` variable. Create folders to store training and test data.

```
dataFolder = fullfile(tempdir,'DALES');
trainDataFolder = fullfile(dataFolder,'dales_las','train');
testDataFolder = fullfile(dataFolder,'dales_las','test');
```

Preview a point cloud from the training data.

```
lasReader = lasFileReader(fullfile(trainDataFolder,'5080_54435.las'));
[pc,attr] = readPointCloud(lasReader,'Attributes','Classification');
labels = attr.Classification;

% Select only labeled data.
pc = select(pc,labels~=0);
labels = labels(labels~=0);
classNames = [
    "ground"
    "vegetation"
    "cars"
    "trucks"
    "powerlines"
    "fences"
    "poles"
    "buildings"
    ];
figure;
ax = pcshow(pc.Location,labels);
helperLabelColorbar(ax,classNames);
title('Point Cloud with Overlaid Semantic Labels');
```



Preprocess Data

Each point cloud in the DALES dataset covers an area of 500-by-500 meters, which is much larger than the typical area covered by terrestrial rotating lidar point clouds. For efficient memory processing, divide the point cloud into small, non-overlapping grids.

Use the `helperCropPointCloudsAndMergeLabels` function, attached to this example as a supporting file, to:

- Crop the point clouds into non-overlapping grids of size 50-by-50 meters.
- Downsample the point cloud to a fixed size.
- Normalize the point clouds to range [0 1].
- Save the cropped grids and semantic labels as PCD and PNG files, respectively.

Define the grid dimensions and set a fixed number of points per grid to enable faster training.

```
gridSize = [50,50];
numPoints = 8192;
```

If the training data is already divided into grids, set `writeFiles` to `false`. Please note that the training data must be in a format supported by the `pcread` (Computer Vision Toolbox) function.

```
writeFiles = true;
numClasses = numel(classNames);
[pcCropTrainPath,labelsCropTrainPath,weights] = helperCropPointCloudsAndMergeLabels( ...
    gridSize,trainDataFolder,numPoints,writeFiles,numClasses);
```

Note: Processing can take some time. The code suspends MATLAB® execution until processing is complete.

The point distribution in the training dataset across all the classes is captured in `weights`. Normalize the `weights` using the `maxWeight`.

```
[maxWeight,maxLabel] = max(weights);
weights = sqrt(maxWeight./weights);
```

Create Datastore Objects for Training

Create a `fileDatastore` object to load PCD files using the `pcread` (Computer Vision Toolbox) function.

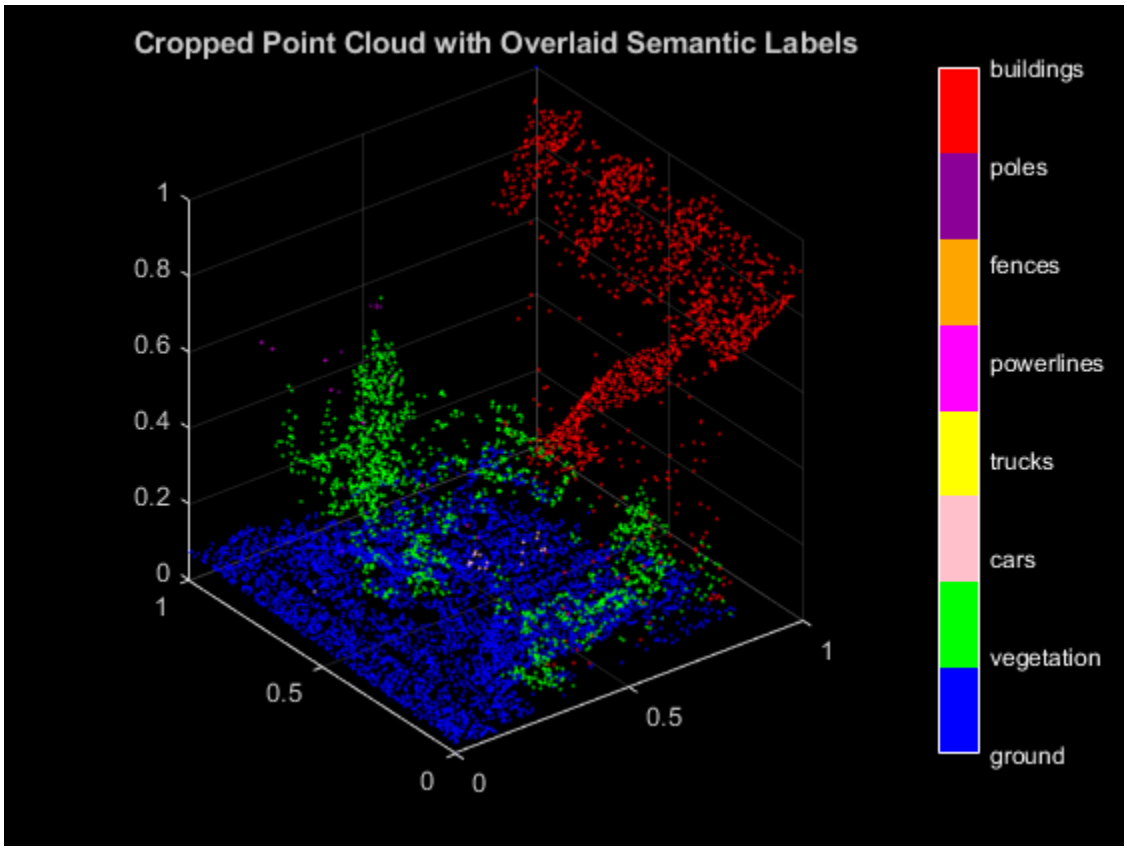
```
ldsTrain = fileDatastore(pcCropTrainPath,'ReadFcn',@(x) pcread(x));
```

Use a `pixelLabelDatastore` (Computer Vision Toolbox) object to store pixel-wise labels from the pixel label images. The object maps each pixel label to a class name and assigns a unique label ID to each class.

```
% Specify label IDs from 1 to the number of classes.
labelIDs = 1 : numClasses;
pxdsTrain = pixelLabelDatastore(labelsCropTrainPath,classNames,labelIDs);
```

Load and display the point cloud.

```
ptcld = preview(ldsTrain);
labels = preview(pxdsTrain);
figure;
ax = pcshow(ptcld.Location,uint8(labels));
helperLabelColorbar(ax,classNames);
title('Cropped Point Cloud with Overlaid Semantic Labels');
```



Use the `helperConvertPointCloud` function, defined at the end of this example, to convert the point cloud to cell array. This function also permutes the dimensions of the point cloud to make it compatible with the input layer of the network.

```
ldsTransformed = transform(ldsTrain,@(x) helperConvertPointCloud(x));
```

Use the `combine` function to combine the point clouds and pixel labels into a single datastore for training.

```
dsTrain = combine(ldsTransformed,pxdsTrain);
```

Define PointNet++ Model

The PointNet++ [2 on page 11-0] segmentation model consists of two main components:

- Set abstraction modules
- Feature propagation modules

The series of set abstraction modules progressively subsamples points of interest by hierarchically grouping points, and uses a custom PointNet architecture to encode points into feature vectors. Because semantic segmentation tasks require point features for all the original points, a series of feature propagation modules are used to hierarchically interpolate features to original points using an inverse-distance based interpolation scheme.

Define the PointNet++ architecture using the `pointnetplusLayers` (Lidar Toolbox) function.

```
lgraph = pointnetplusLayers(numPoints,3,numClasses);
```

To handle the class-imbalance on the DALES dataset, the weighted cross-entropy loss from the `pixelClassificationLayer` (Computer Vision Toolbox) function is used. This will penalize the network more if a point belonging to a class with lower weight is misclassified.

```
% Replace the FocalLoss layer with pixelClassificationLayer.
larray = pixelClassificationLayer('Name','SegmentationLayer','ClassWeights', ...
    weights,'Classes',classNames);
lgraph = replaceLayer(lgraph,'FocalLoss',larray);
```

Specify Training Options

Use the `Adam` optimization algorithm to train the network. Use the `trainingOptions` function to specify the hyperparameters.

```
learningRate = 0.0005;
l2Regularization = 0.01;
numEpochs = 20;
miniBatchSize = 6;
learnRateDropFactor = 0.1;
learnRateDropPeriod = 10;
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;

options = trainingOptions('adam', ...
    'InitialLearnRate',learningRate, ...
    'L2Regularization',l2Regularization, ...
    'MaxEpochs',numEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',learnRateDropFactor, ...
    'LearnRateDropPeriod',learnRateDropPeriod, ...
    'GradientDecayFactor',gradientDecayFactor, ...
    'SquaredGradientDecayFactor',squaredGradientDecayFactor, ...
    'Plots','training-progress');
```

Note: Reduce the `miniBatchSize` value to control memory usage when training.

Train Model

You can train the network yourself by setting the `doTraining` argument to `true`. If you train the network, you can use a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Otherwise, load a pretrained network.

```
doTraining = false;
if doTraining
    % Train the network on the dsTrain datastore using the trainNetwork function.
    [net, info] = trainNetwork(dsTrain,lgraph,options);
else
    % Load the pretrained network.
    load('pointnetplusTrained.mat','net');
end
```

Segment Aerial Point Cloud

The network is trained on downsampled point clouds. To perform segmentation on a test point cloud, first downsample the test point cloud, similar to how training data is downsampled. Perform inference

on this downsampled test point cloud to compute prediction labels. Interpolate the prediction labels, to obtain prediction labels on the dense point cloud.

Define `numNearestNeighbors` and `radius` to find the nearest points in the downsampled point cloud for each point in the dense point cloud and to perform interpolation effectively.

```
numNearestNeighbors = 20;
radius = 0.05;
```

Read the full test point cloud.

```
lasReader = lasFileReader(fullfile(testDataFolder,'5080_54470.las'));
[pc,attr] = readPointCloud(lasReader,'Attributes','Classification');
labelsDenseTarget = attr.Classification;

% Select only labeled data.
pc = select(pc,labelsDenseTarget~=0);
labelsDenseTarget = labelsDenseTarget(labelsDenseTarget~=0);

% Initialize prediction labels
labelsDensePred = zeros(size(labelsDenseTarget));
```

Calculate the number of non-overlapping grids based on `gridSize`, `XLimits`, and `YLimits` of the point cloud.

```
numGridsX = round(diff(pc.XLimits)/gridSize(1));
numGridsY = round(diff(pc.YLimits)/gridSize(2));
[~,edgesX,edgesY,indx,indy] = histcounts2(pc.Location(:,1),pc.Location(:,2), ...
    [numGridsX,numGridsY],'XBinLimits',pc.XLimits,'YBinLimits',pc.YLimits);
ind = sub2ind([numGridsX,numGridsY],indx,indy);
```

Iterate over all the non-overlapping grids and predict the labels using the `semanticseg` (Computer Vision Toolbox) function.

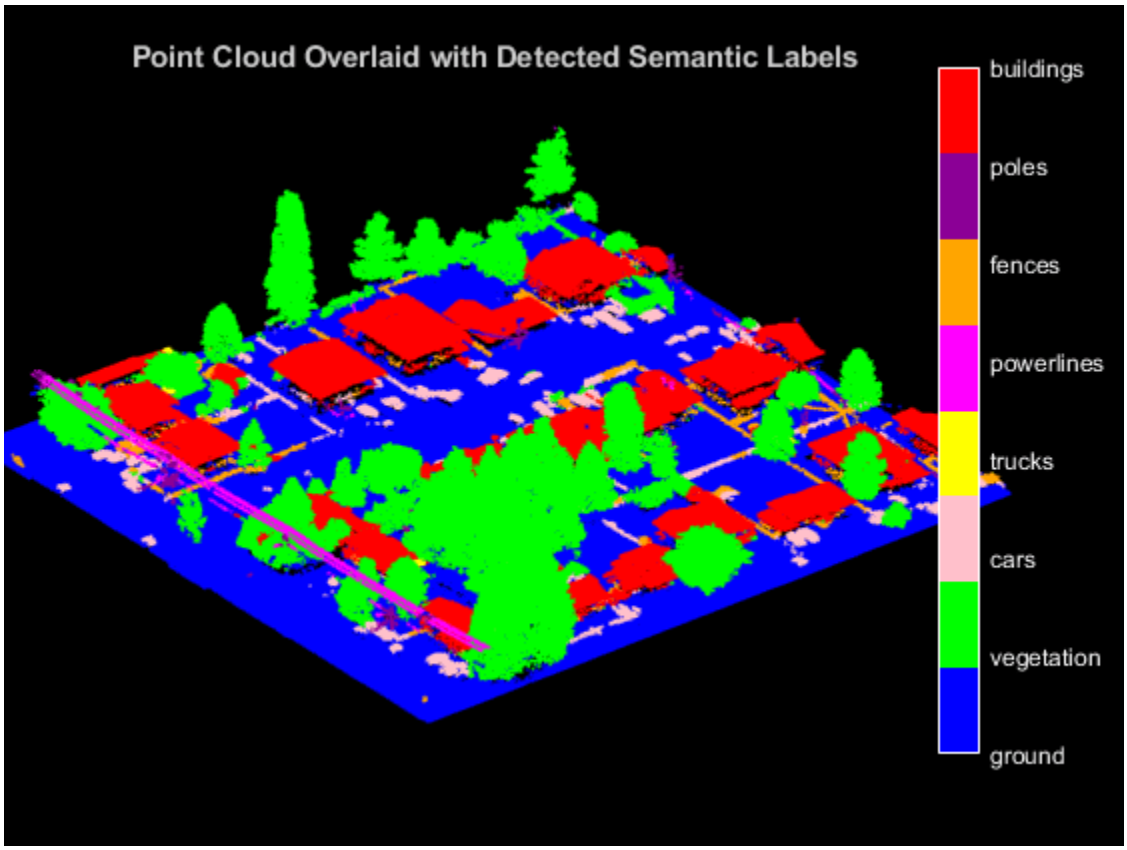
```
for num=1:numGridsX*numGridsY
    idx = ind==num;
    ptCloudDense = select(pc,idx);
    labelsDense = labelsDenseTarget(idx);

    % Use the helperDownsamplePoints function, attached to this example as a
    % supporting file, to extract a downsampled point cloud from the
    % dense point cloud.
    ptCloudSparse = helperDownsamplePoints(ptCloudDense, ...
        labelsDense,numPoints);

    % Make the spatial extent of the dense point cloud and the sparse point
    % cloud same.
    limits = [ptCloudDense.XLimits;ptCloudDense.YLimits;ptCloudDense.ZLimits];
    ptCloudSparseLocation = ptCloudSparse.Location;
    ptCloudSparseLocation(1:2,:) = limits(:,1:2)';
    ptCloudSparse = pointCloud(ptCloudSparseLocation,'Color',ptCloudSparse.Color, ...
        'Intensity',ptCloudSparse.Intensity, ...
        'Normal',ptCloudSparse.Normal);

    % Use the helperNormalizePointCloud function, attached to this example as
    % a supporting file, to normalize the point cloud between 0 and 1.
    ptCloudSparseNormalized = helperNormalizePointCloud(ptCloudSparse);
    ptCloudDenseNormalized = helperNormalizePointCloud(ptCloudDense);
```

```
    % Use the helperConvertPointCloud function, defined at the end of this
    % example, to convert the point cloud to a cell array and to permute the
    % dimensions of the point cloud to make it compatible with the input layer
    % of the network.
    ptCloudSparseForPrediction = helperConvertPointCloud(ptCloudSparseNormalized);

    % Get the output predictions.
    labelsSparsePred = semanticseg(ptCloudSparseForPrediction{1,1}, ...
        net,'OutputType','uint8');

    % Use the helperInterpolate function, attached to this example as a
    % supporting file, to calculate labels for the dense point cloud,
    % using the sparse point cloud and labels predicted on the sparse point cloud.
    interpolatedLabels = helperInterpolate(ptCloudDenseNormalized, ...
        ptCloudSparseNormalized,labelsSparsePred,numNearestNeighbors, ...
        radius,maxLabel,numClasses);

    labelsDensePred(idx) = interpolatedLabels;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

For better visualisation, select a region of interest from the point cloud data. Modify the limits in the `roi` variable according to the point cloud data.

```
roi = [edgesX(5) edgesX(8) edgesY(8) edgesY(11) pc.ZLimits];
indices = findPointsInROI(pc,roi);
figure;
ax = pcshow(select(pc,indices).Location, labelsDensePred(indices));
axis off;
zoom(ax,1.5);
helperLabelColorbar(ax,classNames);
title('Point Cloud Overlaid with Detected Semantic Labels');
```



Evaluate Network

Evaluate the network performance on the test data. Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute the semantic segmentation metrics from the test set results. The target and predicted labels are computed previously and are stored in the `labelsDensePred` and the `labelsDenseTarget` variables respectively.

```
confusionMatrix = segmentationConfusionMatrix(labelsDensePred, ...
    double(labelsDenseTarget),'Classes',1:numClasses);
metrics = evaluateSemanticSegmentation({confusionMatrix},classNames,'Verbose',false);
```

You can measure the amount of overlap per class using the intersection-over-union (IoU) metric.

The `evaluateSemanticSegmentation` (Computer Vision Toolbox) function returns metrics for the entire data set, for individual classes, and for each test image. To see the metrics at the data set level, use the `metrics.DataSetMetrics` property.

```
metrics.DataSetMetrics
```

```
ans=1×4 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU
    ______________    ____________    _______    ___________

       0.93191           0.64238      0.52709      0.88198
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the metrics for each class using the `metrics.ClassMetrics` property.

```
metrics.ClassMetrics
```

```
ans=8×2 table
                 Accuracy       IoU
                 ________    _________

    ground         0.98874      0.93499
    vegetation     0.85948      0.81865
    cars           0.61847      0.36659
    trucks        0.018676    0.0070006
    powerlines      0.7758       0.6904
    fences          0.3753      0.21718
    poles           0.5741      0.28528
    buildings      0.92843      0.89662
```

Although the overall network performance is good, the class metrics for some classes like `Trucks` indicate that more training data is required for better performance.

Supporting Functions

The `helperLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperLabelColorbar(ax,classNames)
% Colormap for the original classes.
cmap = [[0,0,255];
    [0,255,0];
    [255,192,203];
    [255,255,0];
    [255,0,255];
    [255,165,0];
    [139,0,150];
    [255,0,0]];
cmap = cmap./255;
cmap = cmap(1:numel(classNames),:);
colormap(ax,cmap);

% Add colorbar to current figure.
c = colorbar(ax);
c.Color = 'w';

% Center tick labels and use class names for tick marks.
numClasses = size(classNames, 1);
c.Ticks = 1:1:numClasses;
c.TickLabels = classNames;

% Remove tick mark.
c.TickLength = 0;
end
```

The `helperConvertPointCloud` function converts the point cloud to a cell array and permutes the dimensions of the point cloud to make it compatible with the input layer of the network.

```
function data = helperConvertPointCloud(data)
if ~iscell(data)
    data = {data};
end
numObservations = size(data,1);
for i = 1:numObservations
    tmp = data{i,1}.Location;
    data{i,1} = permute(tmp,[1,3,2]);
end
end
```

References

[1] Varney, Nina, Vijayan K. Asari, and Quinn Graehling. "DALES: A Large-Scale Aerial LiDAR dataset for Semantic Segmentation." *ArXiv:2004.11985 [Cs, Stat]*, April 14, 2020. https://arxiv.org/abs/2004.11985.

[2] Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space." *ArXiv:1706.02413 [Cs]*, June 7, 2017. https://arxiv.org/abs/1706.02413.

Code Generation For Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning

This example shows how to generate CUDA® MEX code for a PointNet++ [1 on page 11-0] network for lidar semantic segmentation. This example uses a pretrained PointNet++ network that can segment unorganized lidar point clouds belonging to eight classes (buildings, cars, trucks, poles, power lines, fences, ground, and vegetation). For more information on PointNet++ network, see "Getting started with PointNet++" (Lidar Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static libraries, dynamic libraries, or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see "Third-Party Hardware" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` (GPU Coder) function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Load PointNet++ Network

Use the getPointnetplusNet function, attached as a supporting file to this example, to load the pretrained PointNet++ network. For more information on how to train this network, see "Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning" (Lidar Toolbox).

```
net = getPointnetplusNet;
```

The pretrained network is a DAG network. To display an interactive visualization of the network architecture, use the `analyzeNetwork` function.

The sampling and grouping layer, and the interpolation layer are implemented using the `functionLayer` function, that does not support code generation. So, replace the function layers in the network with custom layers that support code generation using the `helperReplaceFunctionLayers` helper function and save the network as a MAT file with the name `pointnetplusCodegenNet.mat`.

```
net = helperReplaceFunctionLayers(net);
```

`pointnetplusPredict` Entry-Point Function

The `pointnetplusPredict` entry-point function takes a point cloud data matrix as input and performs prediction on it by using the deep learning network saved in the `pointnetplusCodegenNet.mat` file. The function loads the network object from the `pointnetplusCodegenNet.mat` file into a persistent variable `mynet` and reuses the persistent variable in subsequent prediction calls.

```
type('pointnetplusPredict.m');
```

```
function out = pointnetplusPredict(in)
%#codegen

% A persistent object mynet is used to load the DAG network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.

% Copyright 2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('pointnetplusCodegenNet.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX Code

To generate CUDA® code for the `pointnetplusPredict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command with the size of point cloud data in the input layer of the network, which in this case is [8192 1 3].

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig(TargetLibrary='cudnn');
codegen -config cfg pointnetplusPredict -args {randn(8192,1,3,'single')} -report
```

```
Code generation successful: View report
```

To generate CUDA® code for the TensorRT target, create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object.

Segment Aerial Point Cloud Using Generated MEX Code

The network in this example is trained on the DALES data set [2 on page 11-0]. Follow the instructions on the DALES website to download the data set to the folder specified by the `dataFolder` variable. Create folders to store training and test data.

```
dataFolder = fullfile(tempdir,'DALES');
testDataFolder = fullfile(dataFolder,'dales_las','test');
```

Since the network is trained on downsampled point clouds, to perform segmentation on a test point cloud, first downsample the test point cloud, similar to how training data is downsampled. Perform inference on this downsampled test point cloud to compute prediction labels. Interpolate the prediction labels to obtain prediction labels on the dense point cloud.

Define `numNearestNeighbors` and `radius` to find the nearest points in the downsampled point cloud for each point in the dense point cloud and to perform interpolation effectively. Define the parameters `gridSize`, `numPoints`, and `maxLabel`, used to train the network.

```
numNearestNeighbors = 20;
radius = 0.05;
gridSize = [50,50];
numPoints = 8192;
maxLabel = 1;
```

Store the interpolated labels and the target labels in the `DensePredictedLabels`, and `DenseTargetLabels` directories, respectively.

```
densePcTargetLabelsPath = fullfile(testDataFolder,'DenseTargetLabels');
densePcPredLabelsPath =  fullfile(testDataFolder,'DensePredictedLabels');
```

Read the full test point cloud.

```
lasReader = lasFileReader(fullfile(testDataFolder,'5080_54470.las'));
[pc,attr] = readPointCloud(lasReader,'Attributes','Classification');
labelsDenseTarget = attr.Classification;

% Select only labeled data.
pc = select(pc,labelsDenseTarget~=0);
labelsDenseTarget = labelsDenseTarget(labelsDenseTarget~=0);

% Initialize prediction labels.
labelsDensePred = zeros(size(labelsDenseTarget));
classNames = [
    "ground"
    "vegetation"
    "cars"
    "trucks"
    "powerlines"
    "fences"
    "poles"
    "buildings"
    ];
numClasses = numel(classNames);
```

Calculate the number of non-overlapping grids based on the `gridSize`, `XLimits`, and `YLimits` values of the point cloud.

```
numGridsX = round(diff(pc.XLimits)/gridSize(1));
numGridsY = round(diff(pc.YLimits)/gridSize(2));
[~,edgesX,edgesY,indx,indy] = histcounts2(pc.Location(:,1),pc.Location(:,2), ...
    [numGridsX,numGridsY],'XBinLimits',pc.XLimits,'YBinLimits',pc.YLimits);
ind = sub2ind([numGridsX,numGridsY],indx,indy);
```

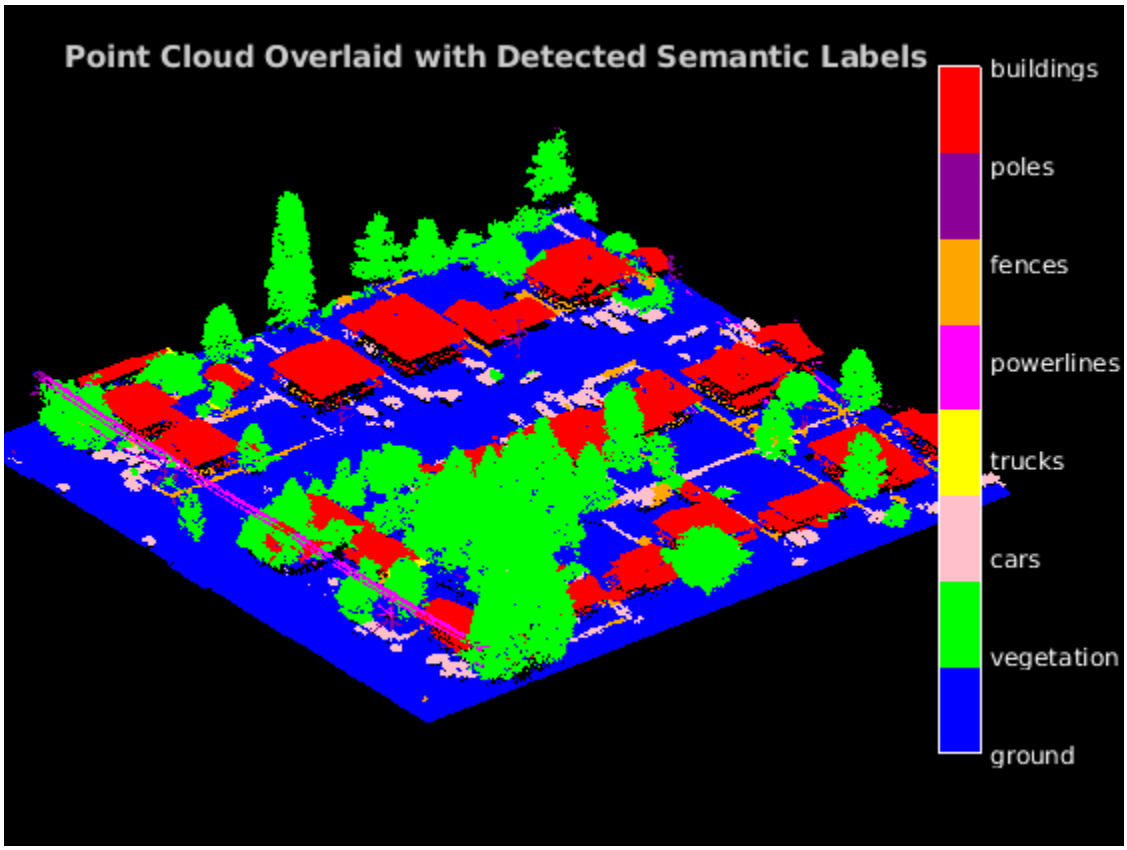
Iterate over all the non-overlapping grids and predict the labels using the `pointnetplusPredict_mex` function.

```
for num=1:numGridsX*numGridsY
    idx = ind==num;
```

```
    ptCloudDense = select(pc,idx);
    labelsDense = labelsDenseTarget(idx);

    % Use the helperDownsamplePoints function, attached to this example as a
    % supporting file, to extract a downsampled point cloud from the
    % dense point cloud.
    ptCloudSparse = helperDownsamplePoints(ptCloudDense, ...
        labelsDense,numPoints);

    % Make the spatial extent of the dense point cloud and the sparse point
    % cloud the same.
    limits = [ptCloudDense.XLimits;ptCloudDense.YLimits;ptCloudDense.ZLimits];
    ptCloudSparseLocation = ptCloudSparse.Location;
    ptCloudSparseLocation(1:2,:) = limits(:,1:2)';
    ptCloudSparse = pointCloud(ptCloudSparseLocation,'Color',ptCloudSparse.Color, ...
        'Intensity',ptCloudSparse.Intensity, ...
        'Normal',ptCloudSparse.Normal);

    % Use the helperNormalizePointCloud function, attached to this example as
    % a supporting file, to normalize the point cloud between 0 and 1.
    ptCloudSparseNormalized = helperNormalizePointCloud(ptCloudSparse);
    ptCloudDenseNormalized = helperNormalizePointCloud(ptCloudDense);

    % Use the helperConvertPointCloud function, defined at the end of this
    % example, to convert the point cloud to a cell array and to permute the
    % dimensions of the point cloud to make it compatible with the input layer
    % of the network.
    ptCloudSparseForPrediction = helperConvertPointCloud(ptCloudSparseNormalized);

    % Get the output predictions.
    scoresPred = pointnetplusPredict_mex(ptCloudSparseForPrediction{1,1});
    [~,labelsSparsePred] = max(scoresPred,[],3);
    labelsSparsePred = uint8(labelsSparsePred);

    % Use the helperInterpolate function, attached to this example as a
    % supporting file, to calculate labels for the dense point cloud,
    % using the sparse point cloud and labels predicted on the sparse point cloud.
    interpolatedLabels = helperInterpolate(ptCloudDenseNormalized, ...
        ptCloudSparseNormalized,labelsSparsePred,numNearestNeighbors, ...
        radius,maxLabel,numClasses);

    labelsDensePred(idx) = interpolatedLabels;
end

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

For better visualization, select a region of interest from the point cloud data. Modify the limits in the `roi` variable according to the point cloud data.

```
roi = [edgesX(5) edgesX(8) edgesY(8) edgesY(11) pc.ZLimits];
indices = findPointsInROI(pc,roi);
figure;
ax = pcshow(select(pc,indices).Location, labelsDensePred(indices));
axis off;
zoom(ax,1.5);
helperLabelColorbar(ax,classNames);
title('Point Cloud Overlaid with Detected Semantic Labels');
```



Supporting Functions

The `helperLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperLabelColorbar(ax,classNames)
% Colormap for the original classes.
cmap = [[0,0,255];
    [0,255,0];
    [255,192,203];
    [255,255,0];
    [255,0,255];
    [255,165,0];
    [139,0,150];
    [255,0,0]];
cmap = cmap./255;
cmap = cmap(1:numel(classNames),:);
colormap(ax,cmap);

% Add colorbar to current figure.
c = colorbar(ax);
c.Color = 'w';

% Center tick labels and use class names for tick marks.
numClasses = size(classNames, 1);
c.Ticks = 1:1:numClasses;
c.TickLabels = classNames;
```

```
% Remove tick mark.
c.TickLength = 0;
end
```

The `helperConvertPointCloud` function converts the point cloud to a cell array and permutes the dimensions of the point cloud to make it compatible with the input layer of the network.

```
function data = helperConvertPointCloud(data)
if ~iscell(data)
    data = {data};
end
numObservations = size(data,1);
for i = 1:numObservations
    tmp = data{i,1}.Location;
    data{i,1} = permute(tmp,[1,3,2]);
end
end
```

References

[1] Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space." *ArXiv:1706.02413 [Cs]*, June 7, 2017. https://arxiv.org/abs/1706.02413.

[2] Varney, Nina, Vijayan K. Asari, and Quinn Graehling. "DALES: A Large-Scale Aerial LiDAR Data Set for Semantic Segmentation." *ArXiv:2004.11985 [Cs, Stat]*, April 14, 2020. https://arxiv.org/abs/2004.11985.

Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network

This example shows how to train a PointSeg semantic segmentation network on 3-D organized lidar point cloud data.

PointSeg [1 on page 11-0] is a convolutional neural network (CNN) for performing end-to-end semantic segmentation of road objects based on an organized lidar point cloud. By using methods such as atrous spatial pyramid pooling (ASPP) and squeeze-and-excitation blocks, the network provides improved segmentation results. The training procedure shown in this example requires 2-D spherical projected images as inputs to the deep learning network.

This example uses a highway scene data set collected using an Ouster OS1 sensor. It contains organized lidar point cloud scans of highway scenes and corresponding ground truth labels for car and truck objects. The size of the data file is approximately 760 MB.

Download Lidar Data Set

Execute this code to download the highway scene data set. The data set contains 1617 point clouds stored as `pointCloud` objects in a cell array. Corresponding ground truth data, which is attached to the example, contains bounding box information of cars and trucks in each point cloud.

```
url = 'https://www.mathworks.com/supportfiles/lidar/data/WPI_LidarData.tar.gz';

outputFolder = fullfile(tempdir,'WPI');
lidarDataTarFile = fullfile(outputFolder,'WPI_LidarData.tar.gz');

if ~exist(lidarDataTarFile, 'file')
    mkdir(outputFolder);

    disp('Downloading WPI Lidar driving data (760 MB)...');
    websave(lidarDataTarFile, url);
    untar(lidarDataTarFile,outputFolder);
end

% Check if tar.gz file is downloaded, but not uncompressed.
if ~exist(fullfile(outputFolder, 'WPI_LidarData.mat'), 'file')
    untar(lidarDataTarFile,outputFolder);
end
lidarData = load(fullfile(outputFolder, 'WPI_LidarData.mat'));

groundTruthData = load('WPI_LidarGroundTruth.mat');
```

Note: Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser, and then extract `WPI_LidarData`. To use the file you downloaded from the web, change the `outputFolder` variable in the code to the location of the downloaded file.

Download Pretrained Network

Download the pretrained network to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('trainedPointSegNet.mat','file')
```

```
    disp('Downloading pretrained network (14 MB)...');
    pretrainedURL = 'https://www.mathworks.com/supportfiles/lidar/data/trainedPointSegNet.mat';
    websave('trainedPointSegNet.mat', pretrainedURL);
end
```

```
Downloading pretrained network (14 MB)...
```

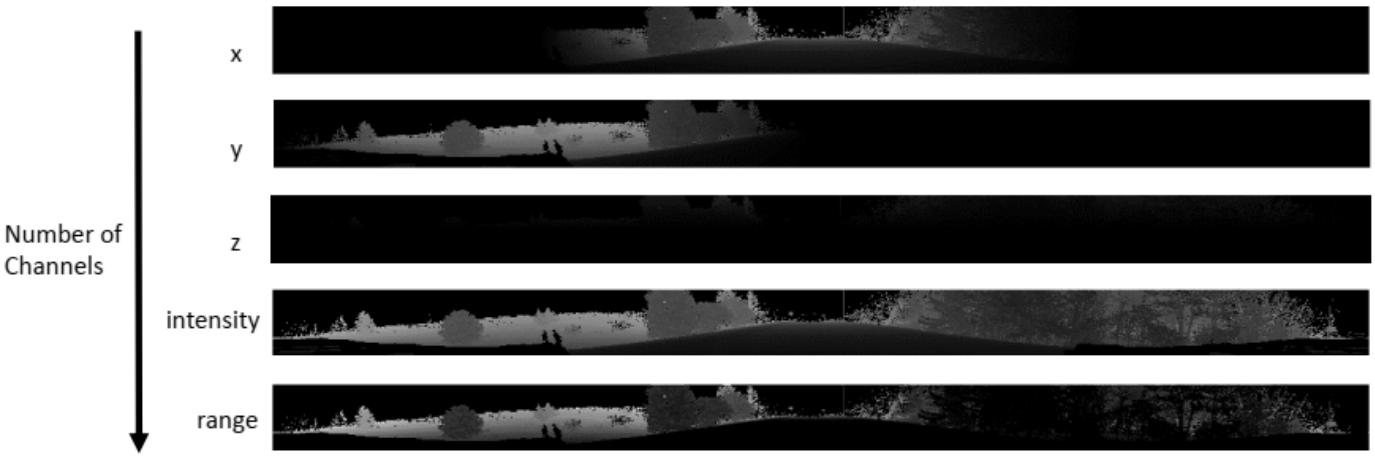
Prepare Data for Training

Load Lidar Point Clouds and Class Labels

Use the `helperGenerateTrainingData` supporting function, attached to this example, to generate training data from the lidar point clouds. The function uses point cloud and bounding box data to create five-channel input images and pixel label images. To create the pixel label images, the function selects points inside the bounding box and labels them with the bounding box class ID. Each training image is specified as a 64-by-1024-by-5 array:

- The height of each image is 64 pixels.
- The width of each image is 1024 pixels.
- The width of each image is 1024 pixels.
- Each image has 5 channels. The five channels specify the 3-D coordinates of the point cloud, intensity, and range: $r = \sqrt{x^2 + y^2 + z^2}$.

A visual representation of the training data follows.



Generate the five-channel training images and pixel label images.

```
imagesFolder = fullfile(outputFolder, 'images');
labelsFolder = fullfile(outputFolder, 'labels');

helperGenerateTrainingData(lidarData, groundTruthData, imagesFolder, labelsFolder);
```

```
Preprocessing data 100.00% complete
```

The five-channel images are saved as MAT files. Pixel labels are saved as PNG files.

Note: Processing can take some time. The code suspends MATLAB® execution until processing is complete.

Create ImageDatastore and PixelLabelDatastore

Use the `imageDatastore` object to extract and store the five channels of the 2-D spherical images using the `helperImageMatReader` supporting function, which is a custom MAT file reader. This function is attached to this example as a supporting file.

```
imds = imageDatastore(imagesFolder, ...
          'FileExtensions', '.mat', ...
          'ReadFcn', @helperImageMatReader);
```

Use the `pixelLabelDatastore` (Computer Vision Toolbox) object to store pixel-wise labels from the label images. The object maps each pixel label to a class name. In this example, cars and trucks are the only objects of interest; all other pixels are the background. Specify these classes (car, truck, and background) and assign a unique label ID to each class.

```
classNames = [
    "background"
    "car"
    "truck"
];

numClasses = numel(classNames);

% Specify label IDs from 1 to the number of classes.
labelIDs = 1 : numClasses;

pxds = pixelLabelDatastore(labelsFolder, classNames, labelIDs);
```

Load and display one of the labeled images by overlaying it on the corresponding intensity image using the `helperDisplayLidarOverlayImage` function, defined in the Supporting Functions on page 11-0 section of this example.

```
imageNumber = 225;

% Point cloud (channels 1, 2, and 3 are for location, channel 4 is for intensity).
I = readimage(imds, imageNumber);

labelMap = readimage(pxds, imageNumber);
figure;
helperDisplayLidarOverlayImage(I, labelMap, classNames);
title('Ground Truth');
```

Prepare Training, Validation, and Test Sets

Use the `helperPartitionLidarData` supporting function, attached to this example, to split the data into training, validation, and test sets that contain 970, 216, and 431 images, respectively.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = ...
    helperPartitionLidarData(imds, pxds);
```

Use the `combine` function to combine the pixel and image datastores for the training and validation data sets.

```
trainingData = combine(imdsTrain, pxdsTrain);
validationData = combine(imdsVal, pxdsVal);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Augment the training data using the `transform` function with custom preprocessing operations specified by the `augmentData` function, defined in the Supporting Functions on page 11-0 section of this example. This function randomly flips the spherical 2-D image and associated labels in the horizontal direction. Apply data augmentation to only the training data set.

```
augmentedTrainingData = transform(trainingData, @(x) augmentData(x));
```

Balance Classes Using Class Weighting

To see the distribution of class labels in the data set, use the `countEachLabel` (Computer Vision Toolbox) function.

```
tbl = countEachLabel(pxds);
tbl(:,{'Name','PixelCount','ImagePixelCount'})
```

```
ans=3×3 table
         Name         PixelCount    ImagePixelCount
    ______________    __________    _______________

    {'background'}    1.0473e+08      1.0597e+08
    {'car'       }    9.7839e+05      8.4738e+07
    {'truck'     }    2.6017e+05      1.9726e+07
```

The classes in this data set are imbalanced, which is a common issue in automotive data sets containing street scenes. The background class covers more area than the car and truck classes. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes.

Use these weights to correct the class imbalance. Use the pixel label counts from the `tbl.PixelCount` property and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq
```

```
classWeights = 3×1

    0.0133
    1.1423
    1.0000
```

Define Network Architecture

Create a PointSeg network using the `createPointSeg` supporting function, which is attached to the example. The code returns the layer graph that you use to train the network.

```
inputSize = [64 1024 5];

lgraph = createPointSeg(inputSize, classNames, classWeights);
```

Use the `analyzeNetwork` function to display an interactive visualization of the network architecture.

```
analyzeNetwork(lgraph)
```

Specify Training Options

Use the `rmsprop` optimization algorithm to train the network. Specify the hyperparameters for the algorithm by using the `trainingOptions` function.

```
maxEpochs = 30;
initialLearningRate= 5e-4;
miniBatchSize = 8;
l2reg = 2e-4;

options = trainingOptions('rmsprop', ...
    'InitialLearnRate', initialLearningRate, ...
    'L2Regularization', l2reg, ...
    'MaxEpochs', maxEpochs, ...
    'MiniBatchSize', miniBatchSize, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 10, ...
    'ValidationData', validationData, ...
    'Plots', 'training-progress', ...
    'VerboseFrequency', 20);
```

Note: Reduce `miniBatchSize` to control memory usage when training.

Train Network

Use the `trainNetwork` function to train a PointSeg network if `doTraining` is `true`. Otherwise, load the pretrained network.

If you train the network, you can use a CPU or a GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox).

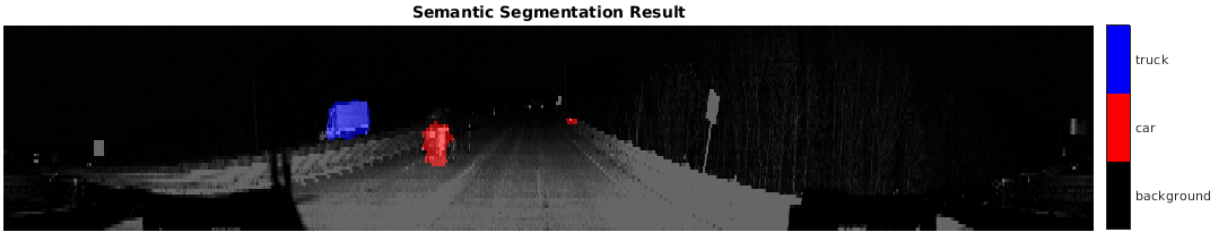
```
if doTraining
    [net, info] = trainNetwork(trainingData, lgraph, options);
else
    pretrainedNetwork = load('trainedPointSegNet.mat');
    net = pretrainedNetwork.net;
end
```

Predict Results on Test Point Cloud

Use the trained network to predict results on a test point cloud and display the segmentation result.

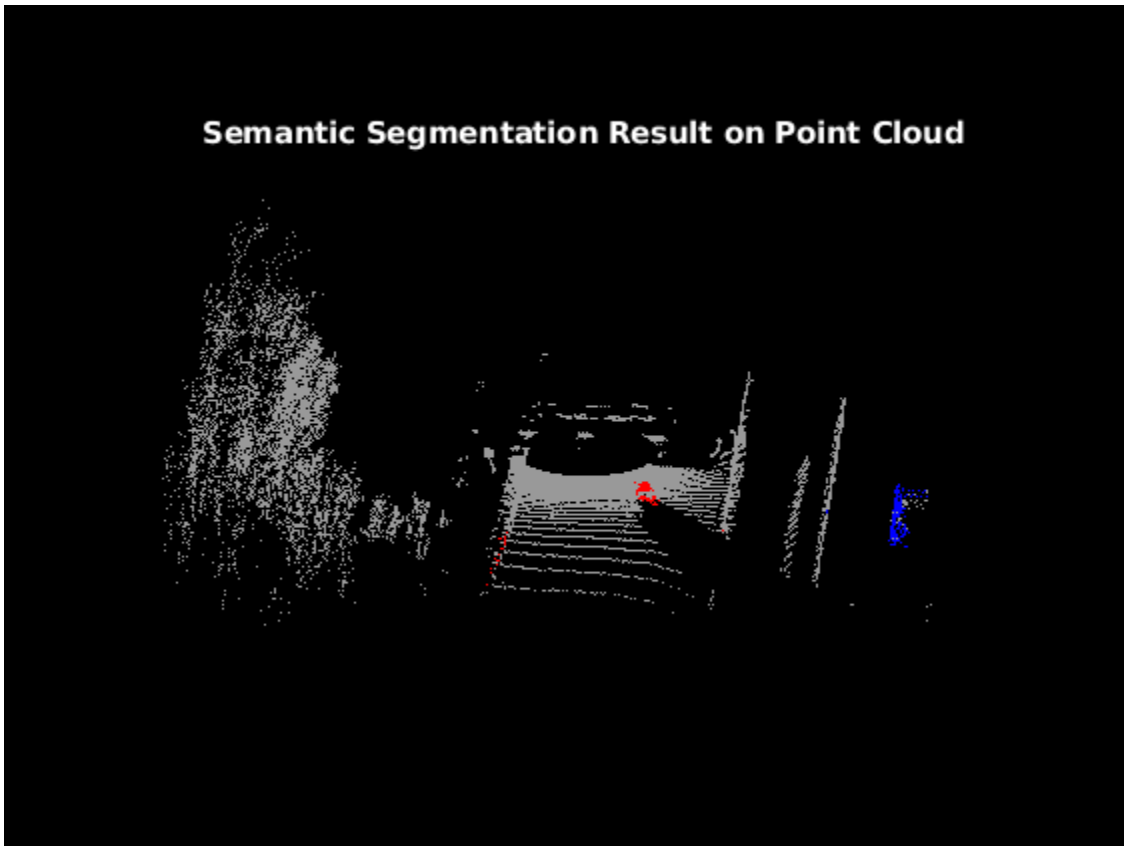
First, read a PCD file and convert the point cloud to a five-channel input image. Predict the labels using the trained network. Display the figure with the segmentation as an overlay.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = helperPointCloudToImage(ptCloud);
predictedResult = semanticseg(I, net);

figure;
helperDisplayLidarOverlayImage(I, predictedResult, classNames);
title('Semantic Segmentation Result');
```



Use the `helperDisplayLidarOverlayPointCloud` helper function, defined in the Supporting Functions on page 11-0 section of this example, to display the segmentation result over the 3-D point cloud object `ptCloud` .

```
figure;
helperDisplayLidarOverlayPointCloud(ptCloud, predictedResult, numClasses);
view([95.71 24.14])
title('Semantic Segmentation Result on Point Cloud');
```



Evaluate Network

Run the `semanticseg` function on the entire test set to measure the accuracy of the network. Set `MiniBatchSize` to a value of 8 to reduce memory usage when segmenting images. You can increase or decrease this value depending on the amount of GPU memory you have on your system.

```
outputLocation = fullfile(tempdir, 'output');
if ~exist(outputLocation,'dir')
    mkdir(outputLocation);
end
pxdsResults = semanticseg(imdsTest, net, ...
                'MiniBatchSize', 8, ...
                'WriteLocation', outputLocation, ...
                'Verbose', false);
```

The `semanticseg` function returns the segmentation results on the test data set as a `PixelLabelDatastore` object. The function writes the actual pixel label data for each test image in the `imdsTest` object to the disk in the location specified by the `'WriteLocation'` argument.

Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute the semantic segmentation metrics from the test set results.

```
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTest, 'Verbose', false);
```

You can measure the amount of overlap per class using the intersection-over-union (IoU) metric.

The `evaluateSemanticSegmentation` function returns metrics for the entire data set, for individual classes, and for each test image. To see the metrics at the data set level, use the `metrics.DataSetMetrics` property.

```
metrics.DataSetMetrics
```

```
ans=1×5 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.99209           0.83752      0.67895      0.98685        0.91654
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the metrics for each class using the `metrics.ClassMetrics` property.

```
metrics.ClassMetrics
```

```
ans=3×3 table
                  Accuracy      IoU      MeanBFScore
                  ________    _______    ___________

    background    0.99466     0.99212      0.98529
    car           0.75977     0.50096      0.82682
    truck         0.75814     0.54378      0.77119
```

Although the network overall performance is good, the class metrics show that biased classes (car and truck) are not segmented as well as the classes with abundant data (background). You can improve the network performance by training the network on more labeled data containing the car and truck classes.

Supporting Functions

Function to Augment Data

The `augmentData` function randomly flips the 2-D spherical image and associated labels in the horizontal direction.

```
function out = augmentData(inp)
%augmentData Apply random horizontal flipping.

out = cell(size(inp));

% Randomly flip the five-channel image and pixel labels horizontally.
I = inp{1};
sz = size(I);
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');

out{1} = imwarp(I,tform,'OutputView',rout);
out{2} = imwarp(inp{2},tform,'OutputView',rout);
end
```

Function to Display Lidar Segmentation Map Overlaid on 2-D Spherical Image

The `helperDisplayLidarOverlayImage` function overlays the semantic segmentation map over the intensity channel of the 2-D spherical image. The function also resizes the overlaid image for better visualization.

```
function helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%helperDisplayLidarOverlayImage Overlay labels over the intensity image.
%
%  helperDisplayLidarOverlayImage(lidarImage, labelMap, classNames)
%  displays the overlaid image. lidarImage is a five-channel lidar input.
%  labelMap contains pixel labels and classNames is an array of label
%  names.

% Read the intensity channel from the lidar image.
intensityChannel = uint8(lidarImage(:,:,4));

% Load the lidar color map.
cmap = helperLidarColorMap();

% Overlay the labels over the intensity image.
B = labeloverlay(intensityChannel,labelMap,'Colormap',cmap,'Transparency',0.4);

% Resize for better visualization.
B = imresize(B, 'Scale', [3 1], 'method', 'nearest');
imshow(B);

% Display the color bar.
helperPixelLabelColorbar(cmap, classNames);
end
```

Function To Display Lidar Segmentation Map Overlaid on 3-D Point Cloud

The `helperDisplayLidarOverPointCloud` function overlays the segmentation result over a 3-D organized point cloud.

```
function helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%helperDisplayLidarOverlayPointCloud Overlay labels over a point cloud object.
%
%  helperDisplayLidarOverlayPointCloud(ptCloud, labelMap, numClasses)
%  displays the overlaid pointCloud object. ptCloud is the organized
%  3-D point cloud input. labelMap contains pixel labels and numClasses
%  is the number of predicted classes.

sz = size(labelMap);

% Apply the color red to cars.
carClassCar = zeros(sz(1), sz(2), numClasses, 'uint8');
carClassCar(:,:,1) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color blue to trucks.
truckClassColor = zeros(sz(1), sz(2), numClasses, 'uint8');
truckClassColor(:,:,3) = 255*ones(sz(1), sz(2), 'uint8');

% Apply the color gray to the background.
backgroundClassColor = 153*ones(sz(1), sz(2), numClasses, 'uint8');

% Extract indices from the labels.
```

```
carIndices = labelMap == 'car';
truckIndices = labelMap == 'truck';
backgroundIndices = labelMap == 'background';

% Extract a point cloud for each class.
carPointCloud = select(ptCloud, carIndices, 'OutputSize','full');
truckPointCloud = select(ptCloud, truckIndices, 'OutputSize','full');
backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize','full');

% Apply colors to different classes.
carPointCloud.Color = carClassCar;
truckPointCloud.Color = truckClassColor;
backgroundPointCloud.Color = backgroundClassColor;

% Merge and add all the processed point clouds with class information.
coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

% Plot the colored point cloud. Set an ROI for better visualization.
ax = pcshow(coloredCloud);
set(ax,'XLim',[-35.0 35.0],'YLim',[-32.0 32.0],'ZLim',[-3 8], ...
    'XColor','none','YColor','none','ZColor','none');
set(get(ax,'parent'), 'units','normalized');
end
```

Function to Define Lidar Colormap

The `helperLidarColorMap` function defines the colormap used by the lidar data set.

```
function cmap = helperLidarColorMap()

cmap = [
   0.00  0.00   0.00  % background
   0.98  0.00   0.00  % car
   0.00  0.00   0.98  % truck
   ];
end
```

Function to Display Pixel Label Colorbar

The `helperPixelLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperPixelLabelColorbar(cmap, classNames)

colormap(gca, cmap);

% Add a colorbar to the current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(classNames, 1);

% Center tick labels.
c.Ticks = 1/(numClasses * 2):1/numClasses:1;

% Remove tick marks.
```

```
c.TickLength = 0;
end
```

References

[1] Wang, Yuan, Tianyue Shi, Peng Yun, Lei Tai, and Ming Liu. “PointSeg: Real-Time Semantic Segmentation Based on 3D LiDAR Point Cloud.” *ArXiv:1807.06288 [Cs]*, September 25, 2018. http://arxiv.org/abs/1807.06288.

Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network

This example shows how to train a SqueezeSegV2 semantic segmentation network on 3-D organized lidar point cloud data.

SqueezeSegV2 [1 on page 11-0] is a convolutional neural network (CNN) for performing end-to-end semantic segmentation of an organized lidar point cloud. The training procedure shown in this example requires 2-D spherical projected images as inputs to the deep learning network.

This example uses PandaSet data set from Hesai and Scale [2] on page 11-0 . The PandaSet contains 4800 unorganized lidar point cloud scans of the various city scenes captured using the Pandar 64 sensor. The data set provides semantic segmentation labels for 42 different classes including car, road, and pedestrian.

Download Lidar Data Set

This example uses a subset of PandaSet, that contains 2560 preprocessed organized point clouds. Each point cloud is specified as a 64-by-1856 matrix. The corresponding ground truth contains the semantic segmentation labels for 12 classes. The point clouds are stored in PCD format, and the ground truth data is stored in PNG format. The size of the data set is 5.2 GB. Execute this code to download the data set.

```
url = 'https://ssd.mathworks.com/supportfiles/lidar/data/Pandaset_LidarData.tar.gz';
outputFolder = fullfile(tempdir,'Pandaset');
lidarDataTarFile = fullfile(outputFolder,'Pandaset_LidarData.tar.gz');
if ~exist(lidarDataTarFile, 'file')
    mkdir(outputFolder);
    disp('Downloading Pandaset Lidar driving data (5.2 GB)...');
    websave(lidarDataTarFile, url);
    untar(lidarDataTarFile,outputFolder);
end
% Check if tar.gz file is downloaded, but not uncompressed.
if (~exist(fullfile(outputFolder,'Lidar'), 'file'))...
        &&(~exist(fullfile(outputFolder,'semanticLabels'), 'file'))
    untar(lidarDataTarFile,outputFolder);
end
lidarData =  fullfile(outputFolder,'Lidar');
labelsFolder = fullfile(outputFolder,'semanticLabels');
```

Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser, and then extract `Pandaset_LidarData` folder. To use the file you downloaded from the web, change the `outputFolder` variable in the code to the location of the downloaded file.

The training procedure for this example is for organized point clouds. For an example showing how to convert unorganized to organized point clouds, see “Unorganized to Organized Conversion of Point Clouds Using Spherical Projection” (Lidar Toolbox).

Download Pretrained Network

Download the pretrained network to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

```
doTraining = false;
pretrainedNetURL = ...
'https://ssd.mathworks.com/supportfiles/lidar/data/trainedSqueezeSegV2PandasetNet.zip';
if ~doTraining
    downloadPretrainedSqueezeSegV2Net(outputFolder, pretrainedNetURL);
end
```

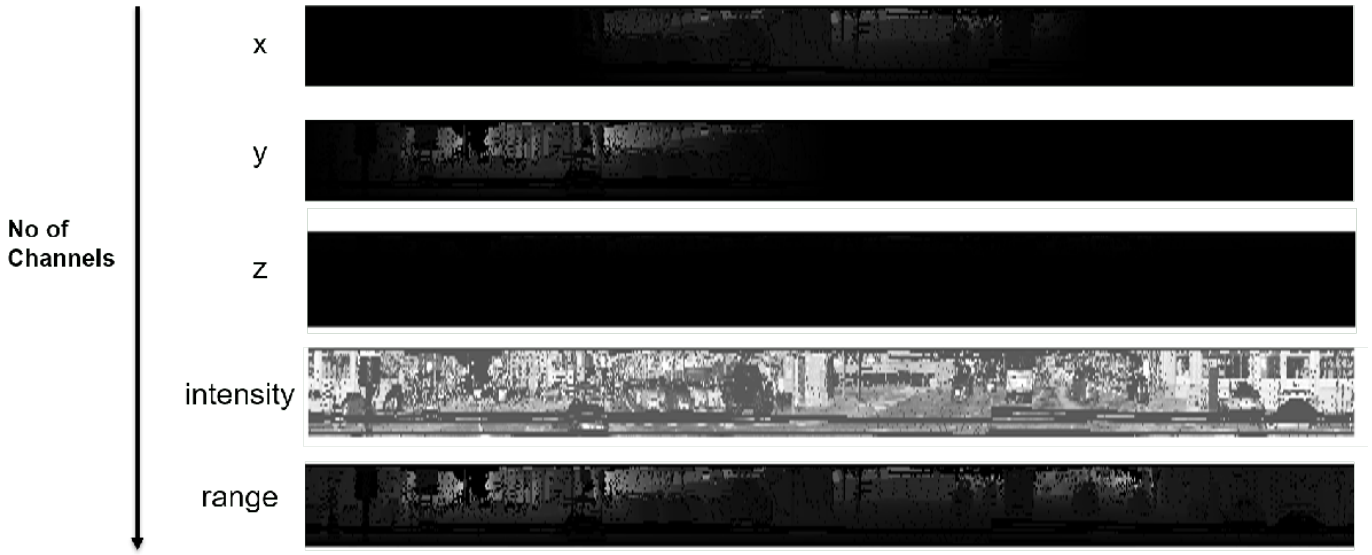
Prepare Data for Training

Load Lidar Point Clouds and Class Labels

Use the `helperTransformOrganizedPointCloudToTrainingData` supporting function, attached to this example, to generate training data from the lidar point clouds. The function uses point cloud data to create five-channel input images. Each training image is specified as a 64-by-1856-by-5 array:

- The height of each image is 64 pixels.
- The width of each image is 1856 pixels.
- Each image has five channels. The five channels specify the 3-D coordinates of the point cloud, intensity, and range: $r = \sqrt{x^2 + y^2 + z^2}$.

A visual representation of the training data follows.



Generate the five-channel training images.

```
imagesFolder = fullfile(outputFolder,'images');
helperTransformOrganizedPointCloudToTrainingData(lidarData,imagesFolder);
```

```
Preprocessing data 100% complete
```

The five-channel images are saved as MAT files.

Processing can take some time. The code suspends MATLAB® execution until processing is complete.

Create imageDatastore and pixelLabelDatastore

Create an `imageDatastore` to extract and store the five channels of the 2-D spherical images using imageDatastore and the helperImageMatReader supporting function, which is a custom MAT file reader. This function is attached to this example as a supporting file.

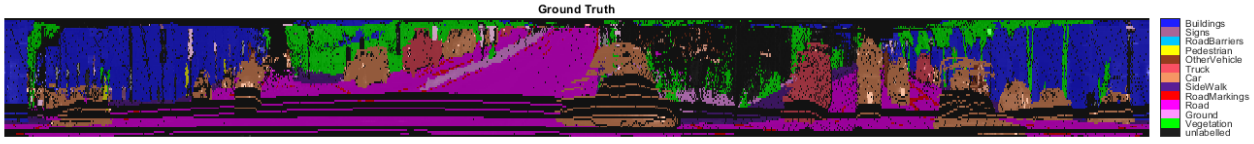
```
imds = imageDatastore(imagesFolder, ...
    'FileExtensions', '.mat', ...
    'ReadFcn', @helperImageMatReader);
```

Create a pixel label datastore using `pixelLabelDatastore` (Computer Vision Toolbox) to store pixel-wise labels from the pixel label images. The object maps each pixel label to a class name. In this example, the vegetation, ground, road, road markings, sidewalk, cars, trucks, other vehicles, pedestrian, road barrier, signs, and buildings are the objects of interest; all other pixels are the background. Specify these classes and assign a unique label ID to each class.

```
classNames = ["unlabelled"
              "Vegetation"
              "Ground"
              "Road"
              "RoadMarkings"
              "SideWalk"
              "Car"
              "Truck"
              "OtherVehicle"
              "Pedestrian"
              "RoadBarriers"
              "Signs"
              "Buildings"];
numClasses = numel(classNames);
% Specify label IDs from 1 to the number of classes.
labelIDs = 1 : numClasses;
pxds = pixelLabelDatastore(labelsFolder, classNames, labelIDs);
```

Load and display one of the labeled images by overlaying it on the corresponding intensity image using the `helperDisplayLidarOverlaidImage` function, defined in the Supporting Functions on page 11-0 section of this example.

```
% Point cloud (channels 1, 2, and 3 are for location, channel 4 is for intensity, and channel 5 i
I = read(imds);
labelMap = read(pxds);
figure;
helperDisplayLidarOverlaidImage(I, labelMap{1,1}, classNames);
title('Ground Truth');
```



Prepare Training, Validation, and Test Sets

Use the `helperPartitionLidarSegmentationDataset` supporting function, attached to this example, to split the data into training, validation, and test sets. You can split the training data

according to the percentage specified by the `trainingDataPercentage`. Divide the rest of the data in a 2:1 ratio into validation and testing data. Default value of `trainingDataPercentage` is `0.7`.

```
[imdsTrain, imdsVal, imdsTest, pxdsTrain, pxdsVal, pxdsTest] = ...
helperPartitionLidarSegmentationDataset(imds, pxds, 'trainingDataPercentage', 0.75);
```

Use the `combine` function to combine the pixel label and image datastores for the training and validation data.

```
trainingData = combine(imdsTrain, pxdsTrain);
validationData = combine(imdsVal, pxdsVal);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Augment the training data by using the `transform` function with custom preprocessing operations specified by the `helperAugmentData` function, defined in the Supporting Functions on page 11-0 section of this example. This function randomly flips the multichannel 2-D image and associated labels in the horizontal direction. Apply data augmentation to only the training data set.

```
augmentedTrainingData = transform(trainingData, @(x) helperAugmentData(x));
```

Define Network Architecture

Create a standard SqueezeSegV2 [1 on page 11-0] network by using the `squeezesegv2Layers` (Lidar Toolbox) function. In the SqueezeSegV2 network, the encoder subnetwork consists of FireModules interspersed with max-pooling layers. This arrangement successively decreases the resolution of the input image. In addition, the SqueezeSegV2 network uses the *focal loss* function to mitigate the effect of the imbalanced class distribution on network accuracy. For more details on how to use the focal loss function in semantic segmentation, see `focalLossLayer` (Computer Vision Toolbox).

Execute this code to create a layer graph that can be used to train the network.

```
inputSize = [64 1856 5];
lgraph = squeezesegv2Layers(inputSize, ...
numClasses,'NumEncoderModules',4,'NumContextAggregationModules',2);
```

Use the `analyzeNetwork` function to display an interactive visualization of the network architecture.

```
analyzeNetwork(lgraph);
```

Specify Training Options

Use the Adam optimization algorithm to train the network. Use the `trainingOptions` function to specify the hyperparameters.

```
maxEpochs = 30;
initialLearningRate = 1e-3;
miniBatchSize = 8;
l2reg = 2e-4;
options = trainingOptions('adam', ...
    'InitialLearnRate', initialLearningRate, ...
    'L2Regularization', l2reg, ...
```

```
    'MaxEpochs', maxEpochs, ...
    'MiniBatchSize', miniBatchSize, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 10, ...
    'ValidationData', validationData, ...
    'Plots', 'training-progress', ...
    'VerboseFrequency', 20);
```

Note: Reduce the `miniBatchSize` value to control memory usage when training.

Train Network

You can train the network yourself by setting the `doTraining` argument to `true`. If you train the network, you can use a CPU or a GPU. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox). Otherwise, load a pretrained network.

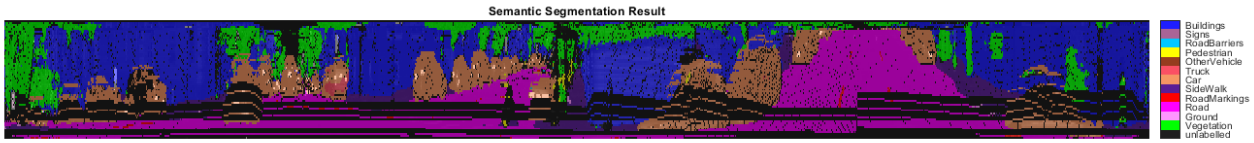
```
if doTraining
    [net, info] = trainNetwork(trainingData, lgraph, options);
else
    load(fullfile(outputFolder,'trainedSqueezeSegV2PandasetNet.mat'),'net');
end
```

Predict Results on Test Point Cloud

Use the trained network to predict results on a test point cloud and display the segmentation result. First, read a five-channel input image and predict the labels using the trained network.

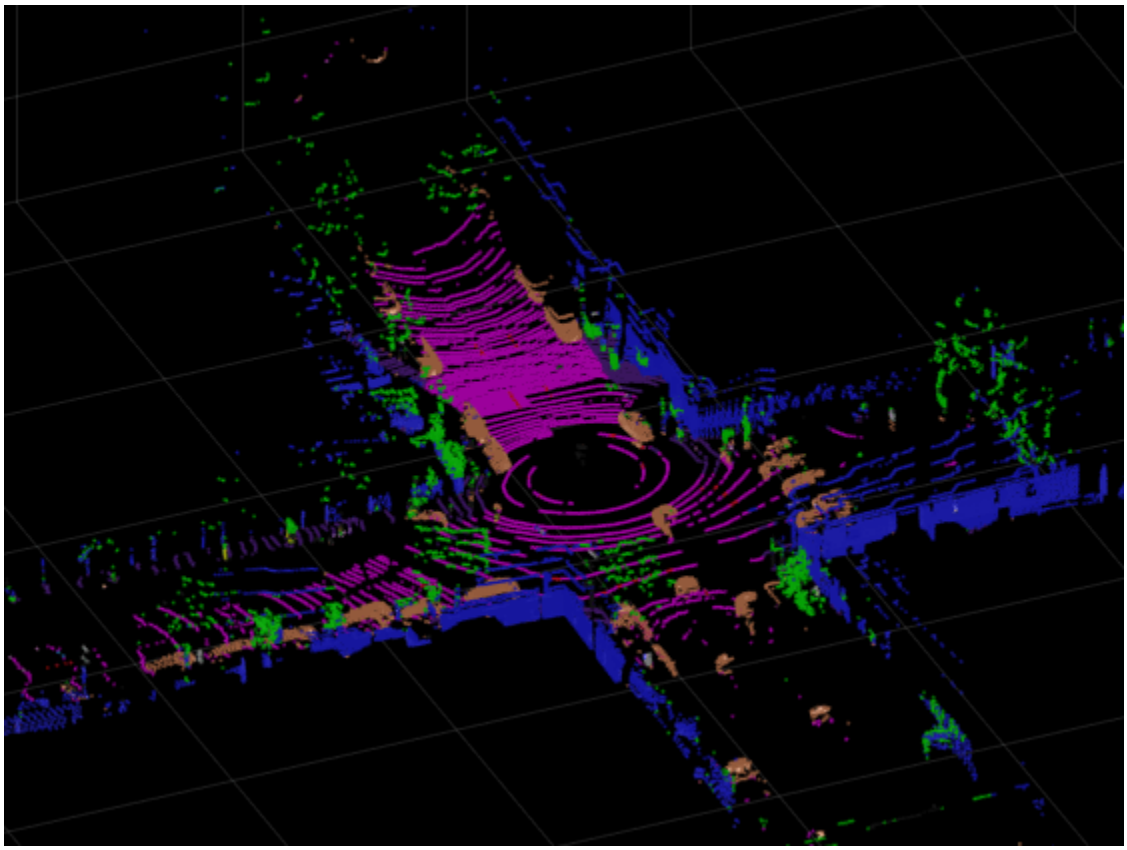
Display the figure with the segmentation as an overlay.

```
I = read(imdsTest);
predictedResult = semanticseg(I, net);
figure;
helperDisplayLidarOverlaidImage(I, predictedResult, classNames);
title('Semantic Segmentation Result');
```



Use the `helperDisplayLabelOverlaidPointCloud` function, defined in the Supporting Functions on page 11-0 section of this example, to display the segmentation result on the point cloud.

```
figure;
helperDisplayLabelOverlaidPointCloud(I, predictedResult);
view([39.2 90.0 60]);
title('Semantic Segmentation Result on Point Cloud');
```



Evaluate Network

Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute the semantic segmentation metrics from the test set results.

```
outputLocation = fullfile(tempdir, 'output');
if ~exist(outputLocation,'dir')
    mkdir(outputLocation);
end
pxdsResults = semanticseg(imdsTest, net, ...
    'MiniBatchSize', 4, ...
    'WriteLocation', outputLocation, ...
    'Verbose', false);
metrics = evaluateSemanticSegmentation(pxdsResults, pxdsTest, 'Verbose', false);
```

You can measure the amount of overlap per class using the intersection-over-union (IoU) metric.

The `evaluateSemanticSegmentation` function returns metrics for the entire data set, for individual classes, and for each test image. To see the metrics at the data set level, use the `metrics.DataSetMetrics` property.

```
metrics.DataSetMetrics
```

```
ans=1×5 table
    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________
```

```
       0.89724          0.61685       0.54431      0.81806        0.74537
```

The data set metrics provide a high-level overview of network performance. To see the impact each class has on the overall performance, inspect the metrics for each class using the `metrics.ClassMetrics` property.

```
metrics.ClassMetrics
```

```
ans=13×3 table
                    Accuracy      IoU      MeanBFScore
                    ________    _______    ___________

    unlabelled         0.94       0.9005      0.99911
    Vegetation      0.77873      0.64819      0.95466
    Ground          0.69019      0.59089      0.60657
    Road            0.94045      0.83663      0.99084
    RoadMarkings    0.37802      0.34149      0.77073
    SideWalk         0.7874      0.65668      0.93687
    Car              0.9334      0.81065      0.95448
    Truck           0.30352      0.27401      0.37273
    OtherVehicle    0.64397      0.58108      0.47253
    Pedestrian      0.26214      0.20896      0.45918
    RoadBarriers    0.23955      0.21971      0.19433
    Signs           0.17276      0.15613      0.44275
    Buildings       0.94891      0.85117      0.96929
```

Although the overall network performance is good, the class metrics for some classes like `RoadMarkings` and `Truck` indicate that more training data is required for better performance.

Supporting Functions

Function to Augment Data

The `helperAugmentData` function randomly flips the spherical image and associated labels in the horizontal direction.

```
function out = helperAugmentData(inp)
% Apply random horizontal flipping.
out = cell(size(inp));
% Randomly flip the five-channel image and pixel labels horizontally.
I = inp{1};
sz = size(I);
tform = randomAffine2d('XReflection',true);
rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');
out{1} = imwarp(I,tform,'OutputView',rout);
out{2} = imwarp(inp{2},tform,'OutputView',rout);
end
```

Function to Display Lidar Segmentation Map Overlaid on 2-D Spherical Image

The `helperDisplayLidarOverlaidImage` function overlays the semantic segmentation map over the intensity channel of the 2-D spherical image. The function also resizes the overlaid image for better visualization.

```
function helperDisplayLidarOverlaidImage(lidarImage, labelMap, classNames)
% helperDisplayLidarOverlaidImage Overlay labels over the intensity image.
```

```
%
%  helperDisplayLidarOverlaidImage(lidarImage, labelMap, classNames)
%  displays the overlaid image. lidarImage is a five-channel lidar input.
%  labelMap contains pixel labels and classNames is an array of label
%  names.
% Read the intensity channel from the lidar image.
intensityChannel = uint8(lidarImage(:,:,4));
% Load the lidar color map.
cmap = helperPandasetColorMap;
% Overlay the labels over the intensity image.
B = labeloverlay(intensityChannel,labelMap,'Colormap',cmap,'Transparency',0.4);
% Resize for better visualization.
B = imresize(B,'Scale',[3 1],'method','nearest');
imshow(B);
helperPixelLabelColorbar(cmap, classNames);
end
```

Function to Display Lidar Segmentation Map Overlaid on 3-D Point Cloud

The `helperDisplayLabelOverlaidPointCloud` function overlays the segmentation result over a 3-D organized point cloud.

```
function helperDisplayLabelOverlaidPointCloud(I,predictedResult)
% helperDisplayLabelOverlaidPointCloud Overlay labels over point cloud object.
%  helperDisplayLabelOverlaidPointCloud(I, predictedResult)
%  displays the overlaid pointCloud object. I is the 5 channels organized
%  input image. predictedResult contains pixel labels.
ptCloud = pointCloud(I(:,:,1:3),'Intensity',I(:,:,4));
cmap = helperPandasetColorMap;
B = ...
labeloverlay(uint8(ptCloud.Intensity),predictedResult,'Colormap',cmap,'Transparency',0.4);
pc = pointCloud(ptCloud.Location,'Color',B);
figure;
ax = pcshow(pc);
set(ax,'XLim',[-70 70],'YLim',[-70 70]);
zoom(ax,3.5);
end
```

Function to Define Lidar Colormap

The `helperPandasetColorMap` function defines the colormap used by the lidar data set.

```
function cmap = helperPandasetColorMap
cmap = [[30,30,30];      % Unlabeled
        [0,255,0];       % Vegetation
        [255, 150, 255]; % Ground
        [255,0,255];     % Road
        [255,0,0];       % Road Markings
        [90, 30, 150];   % Sidewalk
        [245,150,100];   % Car
        [250, 80, 100];  % Truck
        [150, 60, 30];   % Other Vehicle
        [255, 255, 0];   % Pedestrian
        [0, 200, 255];   % Road Barriers
        [170,100,150];   % Signs
        [30, 30, 255]];  % Building
cmap = cmap./255;
end
```

Function to Display Pixel Label Colorbar

The `helperPixelLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperPixelLabelColorbar(cmap, classNames)
colormap(gca, cmap);
% Add a colorbar to the current figure.
c = colorbar('peer', gca);
% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(classNames, 1);
% Center tick labels.
c.Ticks = 1/(numClasses * 2):1/numClasses:1;
% Remove tick marks.
c.TickLength = 0;
end
```

Function to Download Pretrained Model

The `downloadPretrainedSqueezeSegV2Net` function downloads the pretrained model.

```
function downloadPretrainedSqueezeSegV2Net(outputFolder, pretrainedNetURL)
    preTrainedMATFile = fullfile(outputFolder,'trainedSqueezeSegV2PandasetNet.mat');
    preTrainedZipFile = fullfile(outputFolder,'trainedSqueezeSegV2PandasetNet.zip');

    if ~exist(preTrainedMATFile,'file')
        if ~exist(preTrainedZipFile,'file')
            disp('Downloading pretrained model (5 MB)...');
            websave(preTrainedZipFile, pretrainedNetURL);
        end
        unzip(preTrainedZipFile, outputFolder);
    end
end
```

References

[1] Wu, Bichen, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. “SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud.” In *2019 International Conference on Robotics and Automation (ICRA)*, 4376–82. Montreal, QC, Canada: IEEE, 2019.https://doi.org/10.1109/ICRA.2019.8793495.

[2] Hesai and Scale. PandaSet. https://scale.com/open-datasets/pandaset

Code Generation for Lidar Point Cloud Segmentation Network

This example shows how to generate CUDA® MEX code for a deep learning network for lidar semantic segmentation. This example uses a pretrained SqueezeSegV2 [1] network that can segment organized lidar point clouds belonging to three classes (*background*, *car*, and *truck*). For information on the training procedure for the network, see “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” (Lidar Toolbox). The generated MEX code takes a point cloud as input and performs prediction on the point cloud by using the `DAGNetwork` object for the SqueezeSegV2 network.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- NVIDIA TensorRT library.
- Environment variables for the compilers and libraries. For details, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` (GPU Coder) function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Segmentation Network

SqueezeSegV2 is a convolutional neural network (CNN) designed for the semantic segmentation of organized lidar point clouds. It is a deep encoder-decoder segmentation network trained on a lidar data set and imported into MATLAB® for inference. In SqueezeSegV2, the encoder subnetwork consists of convolution layers that are interspersed with max-pooling layers. This arrangement successively decreases the resolution of the input image. The decoder subnetwork consists of a series of transposed convolution layers, which successively increase the resolution of the input image. In addition, the SqueezeSegV2 network mitigates the impact of missing data by including context aggregation modules (CAMs). A CAM is a convolutional subnetwork with filterSize of value [7, 7] that aggregates contextual information from a larger receptive field, which improves the robustness of the network to missing data. The SqueezeSegV2 network in this example is trained to segment points belonging to three classes (background, car, and truck).

For more information on training a semantic segmentation network in MATLAB® by using the Mathworks lidar dataset, see “Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network” (Lidar Toolbox).

Download the pretrained SqueezeSegV2 Network.

```
net = getSqueezeSegV2Net();
```

```
Downloading pretrained SqueezeSegV2 (2 MB)...
```

The DAG network contains 238 layers, including convolution, ReLU, and batch normalization layers, and a focal loss output layer. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(net);
```

`squeezesegv2_predict` Entry-Point Function

The `squeezesegv2_predict.m` entry-point function, which is attached to this example, takes a point cloud as input and performs prediction on it by using the deep learning network saved in the `SqueezeSegV2Net.mat` file. The function loads the network object from the `SqueezeSegV2Net.mat` file into a persistent variable `mynet` and reuses the persistent variable in subsequent prediction calls.

```
type('squeezesegv2_predict.m');
```

```
function out = squeezesegv2_predict(in)
%#codegen

% A persistent object mynet is used to load the DAG network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('SqueezeSegV2Net.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX Code

To generate CUDA MEX code for the `squeezesegv2_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command, specifying an input size of [64, 1024, 5]. This value corresponds to the size of the input layer of the SqueezeSegV2 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
```

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg squeezesegv2_predict -args {ones(64,1024,5,'uint8')} -report

Code generation successful: View report
```

To generate CUDA C++ code that takes advantage of NVIDIA TensorRT libraries, in the code, specify `coder.DeepLearningConfig('tensorrt')` instead of `coder.DeepLearningConfig('cudnn')`.

For information on how to generate MEX code for deep learning networks on Intel® processors, see "Code Generation for Deep Learning Networks with MKL-DNN" (MATLAB Coder).

Prepare Data

Load an organized test point cloud in MATLAB®. Convert the point cloud to a five-channel image for prediction.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = pointCloudToImage(ptCloud);

% Examine converted data
whos I
```

```
  Name       Size                 Bytes  Class    Attributes

  I         64x1024x5            327680  uint8
```

The image has five channels. The *(x,y,z)* point coordinates comprise the first three channels. The fourth channel contains the lidar intensity measurement. The fifth channel contains the range information, which is computed as $r = \sqrt{x^2 + y^2 + z^2}$.

Visualize intensity channel of the image.

```
intensityChannel = I(:,:,4);

figure;
imshow(intensityChannel);
title('Intensity Image');
```

Run Generated MEX on Data

Call `squeezesegv2_predict_mex` on the five-channel image.

```
predict_scores = squeezesegv2_predict_mex(I);
```

The `predict_scores` variable is a three-dimensional matrix that has three channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get the pixel-wise labels

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the intensity channel image and display the segmented region. Resize the segmented output and add a colorbar for better visualization.

```
classes = [
    "background"
    "car"
    "truck"
    ];

cmap = lidarColorMap();
SegmentedImage = labeloverlay(intensityChannel,argmax,'ColorMap',cmap);
SegmentedImage = imresize(SegmentedImage, 'Scale', [2 1], 'method', 'nearest');
figure;
imshow(SegmentedImage);

N = numel(classes);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classes),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','none
colormap(cmap)
title('Semantic Segmentation Result');
```

Run Generated MEX Code on Point Cloud Sequence

Read an input point cloud sequence. The sequence contains 10 organized `pointCloud` frames collected using an Ouster OS1 lidar sensor. The input data has a height of 64 and a width of 1024, so each pointCloud object is of size 64-by-1024.

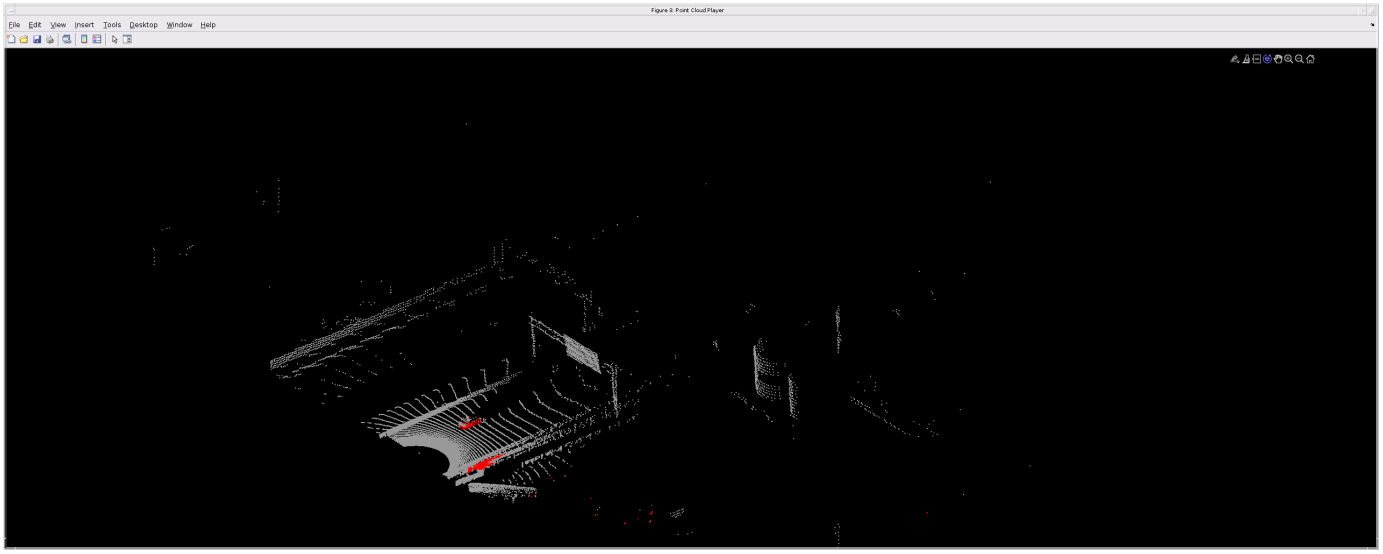
```
dataFile = 'highwaySceneData.mat';

% Load data in workspace.
load(dataFile);
```

Setup different colors to visualize point-wise labels for different classes of interest.

```
% Apply the color red to cars.
carClassCar = zeros(64, 1024, 3, 'uint8');
carClassCar(:,:,1) = 255*ones(64, 1024, 'uint8');

% Apply the color blue to trucks.
truckClassColor = zeros(64, 1024, 3, 'uint8');
truckClassColor(:,:,3) = 255*ones(64, 1024, 'uint8');

% Apply the color gray to background.
backgroundClassColor = 153*ones(64, 1024, 3, 'uint8');
```

Set the `pcplayer` function properties to display the sequence and the output predictions. Read the input sequence frame by frame and detect classes of interest using the model.

```
xlimits = [0 120.0];
ylimits = [-80.7 80.7];
zlimits = [-8.4 27];

player = pcplayer(xlimits, ylimits, zlimits);
set(get(player.Axes,'parent'), 'units','normalized','outerposition',[0 0 1 1]);
zoom(get(player.Axes,'parent'),2);
set(player.Axes,'XColor','none','YColor','none','ZColor','none');

for i = 1 : numel(inputData)
    ptCloud = inputData{i};

    % Convert point cloud to five-channel image for prediction.
    I = pointCloudToImage(ptCloud);

    % Call squeezesegv2_predict_mex on the 5-channel image.
    predict_scores = squeezesegv2_predict_mex(I);

    % Convert the numeric output values to categorical labels.
    [~,predictedOutput] = max(predict_scores,[],3);
    predictedOutput = categorical(predictedOutput, 1:3, classes);

    % Extract the indices from labels.
    carIndices = predictedOutput == 'car';
    truckIndices = predictedOutput == 'truck';
    backgroundIndices = predictedOutput == 'background';

    % Extract a point cloud for each class.
    carPointCloud = select(ptCloud, carIndices, 'OutputSize','full');
    truckPointCloud = select(ptCloud, truckIndices, 'OutputSize','full');
    backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize','full');

    % Fill the colors to different classes.
    carPointCloud.Color = carClassCar;
    truckPointCloud.Color = truckClassColor;
    backgroundPointCloud.Color = backgroundClassColor;

    % Merge and add all the processed point clouds with class information.
    coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
    coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

    % View the output.
    view(player, coloredCloud);
    drawnow;
end
```



Helper Functions

The helper functions used in this example follow.

```
type pointCloudToImage.m
```

```
function image = pointCloudToImage(ptcloud)
%pointCloudToImage Converts organized 3-D point cloud to 5-channel
%   2-D image.

image = ptcloud.Location;
image(:,:,4) = ptcloud.Intensity;
rangeData = iComputeRangeData(image(:,:,1),image(:,:,2),image(:,:,3));
image(:,:,5) = rangeData;

% Cast to uint8.
image = uint8(image);
end

%--------------------------------------------------------------------------
function rangeData = iComputeRangeData(xChannel,yChannel,zChannel)
rangeData = sqrt(xChannel.*xChannel+yChannel.*yChannel+zChannel.*zChannel);
end
```

```
type lidarColorMap.m
```

```
function cmap = lidarColorMap()

cmap = [
   0.00  0.00   0.00  % background
   0.98  0.00   0.00  % car
   0.00  0.00   0.98  % truck
   ];
end
```

References

[1] Wu, Bichen, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. “SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud.” Preprint, submitted September 22, 2018. http://arxiv.org/abs/1809.08495.

Lidar 3-D Object Detection Using PointPillars Deep Learning

This example shows how to train a PointPillars network for object detection in point clouds.

Lidar point cloud data can be acquired by a variety of lidar sensors, including Velodyne®, Pandar, and Ouster sensors. These sensors capture 3-D position information about objects in a scene, which is useful for many applications in autonomous driving and augmented reality. However, training robust detectors with point cloud data is challenging because of the sparsity of data per object, object occlusions, and sensor noise. Deep learning techniques have been shown to address many of these challenges by learning robust feature representations directly from point cloud data. One deep learning technique for 3-D object detection is PointPillars [1 on page 11-0]. Using a similar architecture to PointNet, the PointPillars network extracts dense, robust features from sparse point clouds called pillars, then uses a 2-D deep learning network with a modified SSD object detection network to estimate joint 3-D bounding boxes, orientations, and class predictions.

This example uses the PandaSet [2 on page 11-0] data set from Hesai and Scale. PandaSet contains 8240 unorganized lidar point cloud scans of various city scenes captured using a Pandar64 sensor. The data set provides 3-D bounding box labels for 18 different object classes, including car, truck, and pedestrian.

Download Lidar Data Set

This example uses a subset of PandaSet that contains 2560 preprocessed organized point clouds. Each point cloud covers 360^o of view, and is specified as a 64-by-1856 matrix. The point clouds are stored in PCD format and their corresponding ground truth data is stored in the `PandaSetLidarGroundTruth.mat` file. The file contains 3-D bounding box information for three classes, which are car, truck, and pedestrian. The size of the data set is 5.2 GB.

Download the Pandaset dataset from the given URL using the `helperDownloadPandasetData` helper function, defined at the end of this example.

```
doTraining = false;

outputFolder = fullfile(tempdir,'Pandaset');

lidarURL = ['https://ssd.mathworks.com/supportfiles/lidar/data/' ...
    'Pandaset_LidarData.tar.gz'];
helperDownloadPandasetData(outputFolder,lidarURL);
```

Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the `outputFolder` variable in the code to the location of the downloaded file.

Load Data

Create a file datastore to load the PCD files from the specified path using the `pcread` (Computer Vision Toolbox) function.

```
path = fullfile(outputFolder,'Lidar');
lidarData = fileDatastore(path,'ReadFcn',@(x) pcread(x));
```

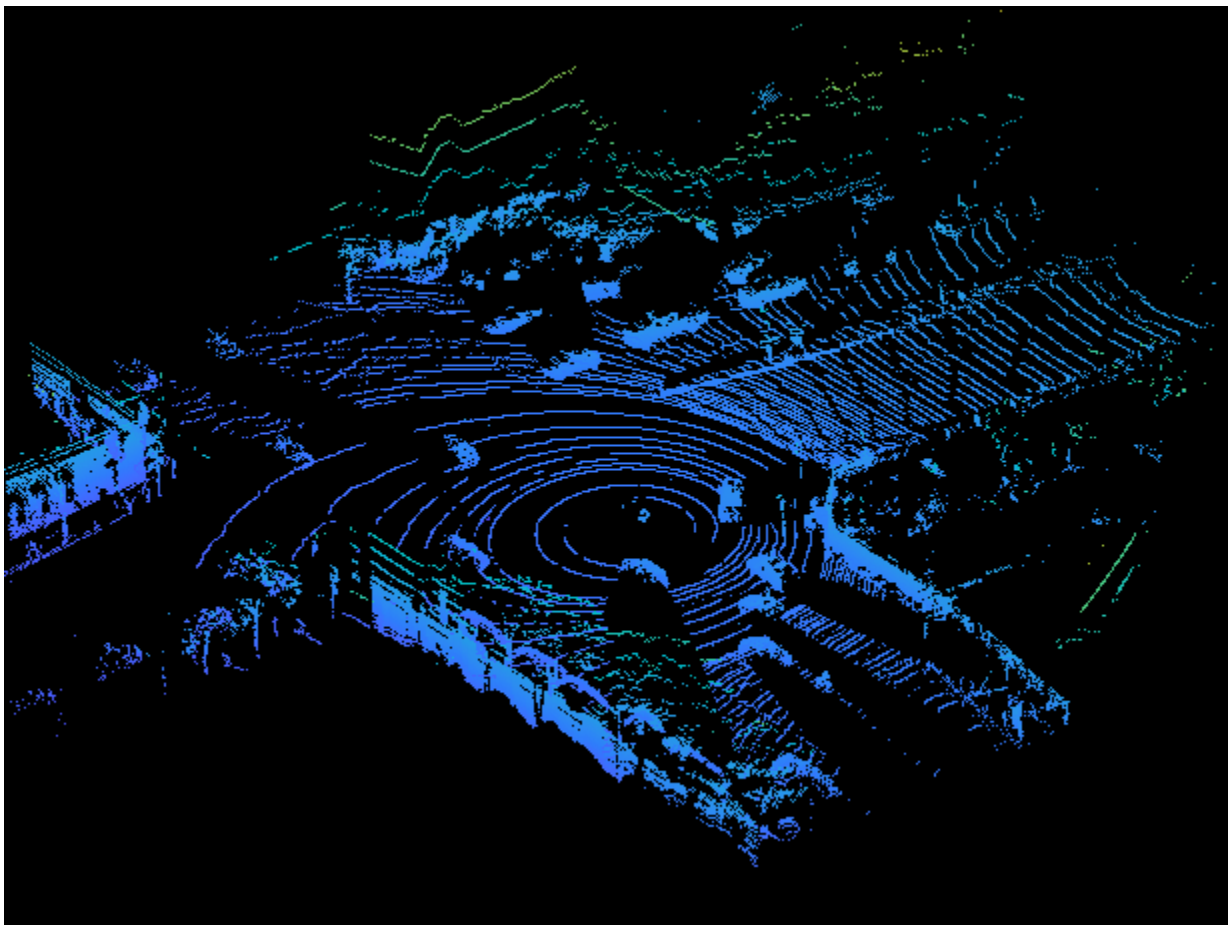
Load the 3-D bounding box labels of the car and truck objects.

```
gtPath = fullfile(outputFolder,'Cuboids','PandaSetLidarGroundTruth.mat');
data = load(gtPath,'lidarGtLabels');
```

```
Labels = timetable2table(data.lidarGtLabels);
boxLabels = Labels(:,2:3);
```

Display the full-view point cloud.

```
figure
ptCld = read(lidarData);
ax = pcshow(ptCld.Location);
set(ax,'XLim',[-50 50],'YLim',[-40 40]);
zoom(ax,2.5);
axis off;
```



```
reset(lidarData);
```

Preprocess Data

The PandaSet data consists of full-view point clouds. For this example, crop the full-view point clouds to front-view point clouds using the standard parameters [1 on page 11-0]. These parameters determine the size of the input passed to the network. Selecting a smaller range of point clouds along the x, y, and z-axis helps detect objects that are closer to the origin and also decreases the overall training time of the network.

```
xMin = 0.0;     % Minimum value along X-axis.
yMin = -39.68;  % Minimum value along Y-axis.
zMin = -5.0;    % Minimum value along Z-axis.
```

```
xMax = 69.12;   % Maximum value along X-axis.
yMax = 39.68;   % Maximum value along Y-axis.
zMax = 5.0;     % Maximum value along Z-axis.
xStep = 0.16;   % Resolution along X-axis.
yStep = 0.16;   % Resolution along Y-axis.
dsFactor = 2.0; % Downsampling factor.

% Calculate the dimensions for the pseudo-image.
Xn = round(((xMax - xMin) / xStep));
Yn = round(((yMax - yMin) / yStep));

% Define point cloud parameters.
pointCloudRange = [xMin,xMax,yMin,yMax,zMin,zMax];
voxelSize = [xStep,yStep];
```

Use the `cropFrontViewFromLidarData` helper function, attached to this example as a supporting file, to:

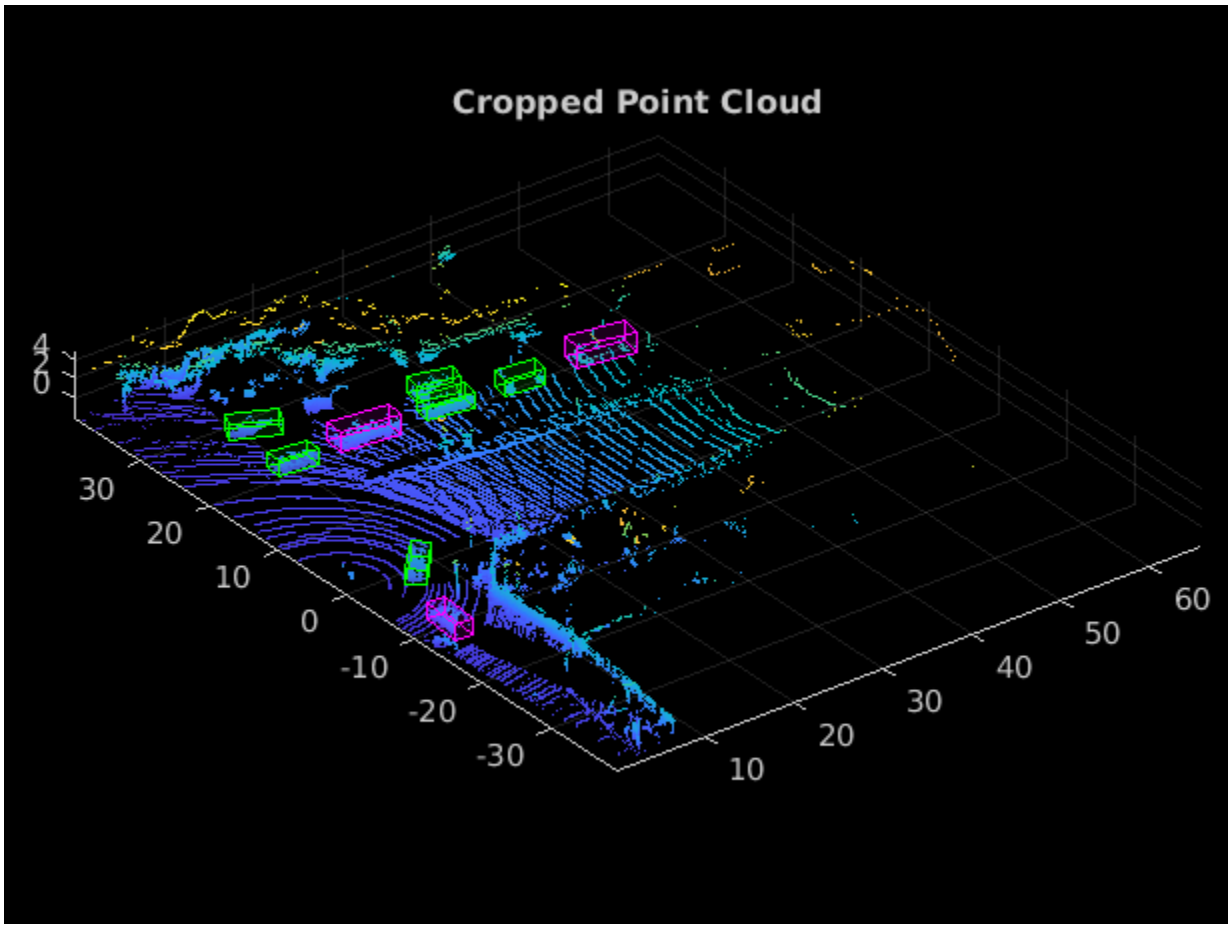
- Crop the front view from the input full-view point cloud.
- Select the box labels that are inside the ROI specified by `gridParams`.

```
[croppedPointCloudObj,processedLabels] = cropFrontViewFromLidarData(...
    lidarData,boxLabels,pointCloudRange);
```

```
Processing data 100% complete
```

Display the cropped point cloud and the ground truth box labels using the `helperDisplay3DBoxesOverlaidPointCloud` helper function defined at the end of the example.

```
pc = croppedPointCloudObj{1,1};
gtLabelsCar = processedLabels.Car{1};
gtLabelsTruck = processedLabels.Truck{1};

helperDisplay3DBoxesOverlaidPointCloud(pc.Location,gtLabelsCar,...
   'green',gtLabelsTruck,'magenta','Cropped Point Cloud');
```




```
reset(lidarData);
```

Create Datastore Objects for Training

Split the data set into training and test sets. Select 70% of the data for training the network and the rest for evaluation.

```
rng(1);
shuffledIndices = randperm(size(processedLabels,1));
idx = floor(0.7 * length(shuffledIndices));

trainData = croppedPointCloudObj(shuffledIndices(1:idx),:);
testData = croppedPointCloudObj(shuffledIndices(idx+1:end),:);

trainLabels = processedLabels(shuffledIndices(1:idx),:);
testLabels = processedLabels(shuffledIndices(idx+1:end),:);
```

So that you can easily access the datastores, save the training data as PCD files by using the `saveptCldToPCD` helper function, attached to this example as a supporting file. You can set `writeFiles` to `"false"` if your training data is saved in a folder and is supported by the `pcread` function.

```
writeFiles = true;
dataLocation = fullfile(outputFolder,'InputData');
```

```
[trainData,trainLabels] = saveptCldToPCD(trainData,trainLabels,...
    dataLocation,writeFiles);
```

```
Processing data 100% complete
```

Create a file datastore using `fileDatastore` to load PCD files using the `pcread` (Computer Vision Toolbox) function.

```
lds = fileDatastore(dataLocation,'ReadFcn',@(x) pcread(x));
```

Createa box label datastore using `boxLabelDatastore` (Computer Vision Toolbox) for loading the 3-D bounding box labels.

```
bds = boxLabelDatastore(trainLabels);
```

Use the `combine` function to combine the point clouds and 3-D bounding box labels into a single datastore for training.

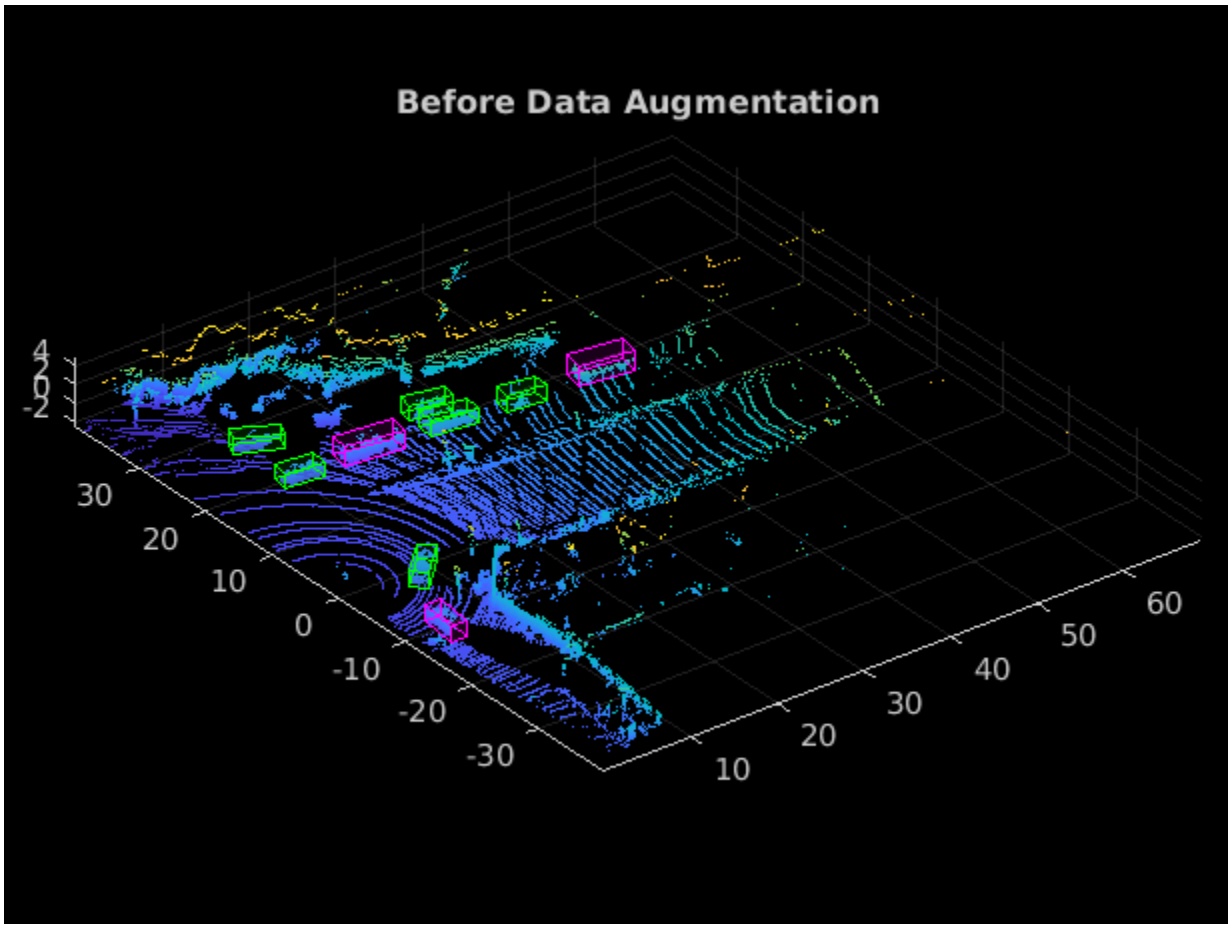
```
cds = combine(lds,bds);
```

Data Augmentation

This example uses ground truth data augmentation and several other global data augmentation techniques to add more variety to the training data and corresponding boxes. For more information on typical data augmentation techniques used in 3-D object detection workflows with lidar data, see “Data Augmentations for Lidar Object Detection Using Deep Learning” (Lidar Toolbox).

Read and display a point cloud before augmentation using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example..

```
augData = read(cds);
augptCld = augData{1,1};
augLabels = augData{1,2};
augClass = augData{1,3};

labelsCar = augLabels(augClass=='Car',:);
labelsTruck = augLabels(augClass=='Truck',:);

helperDisplay3DBoxesOverlaidPointCloud(augptCld.Location,labelsCar,'green',...
    labelsTruck,'magenta','Before Data Augmentation');
```




```
reset(cds);
```

Use the `sampleGroundTruthObjectsFromLidarData` helper function, attached to this example as a supporting file, to extract all the ground truth bounding boxes from the training data.

```
classNames = {'Car','Truck'};
sampleLocation = fullfile(tempdir,'GTsamples');
[sampledGTData,indices] = sampleGroundTruthObjectsFromLidarData(cds,classNames,...
    'MinPoints',20,'sampleLocation',sampleLocation);
```

Use the `augmentGroundTruthObjectsToLidarData` helper function, attached to this example as a supporting file, to randomly add a fixed number of car and truck class objects to every point cloud. Use the `transform` function to apply the ground truth and custom data augmentations to the training data.

```
numObjects = [10,10];
cdsAugmented = transform(cds,@(x) augmentGroundTruthObjectsToLidarData(x,...
    sampledGTData,indices,classNames,numObjects));
```

In addition, apply the following data augmentations to every point cloud.

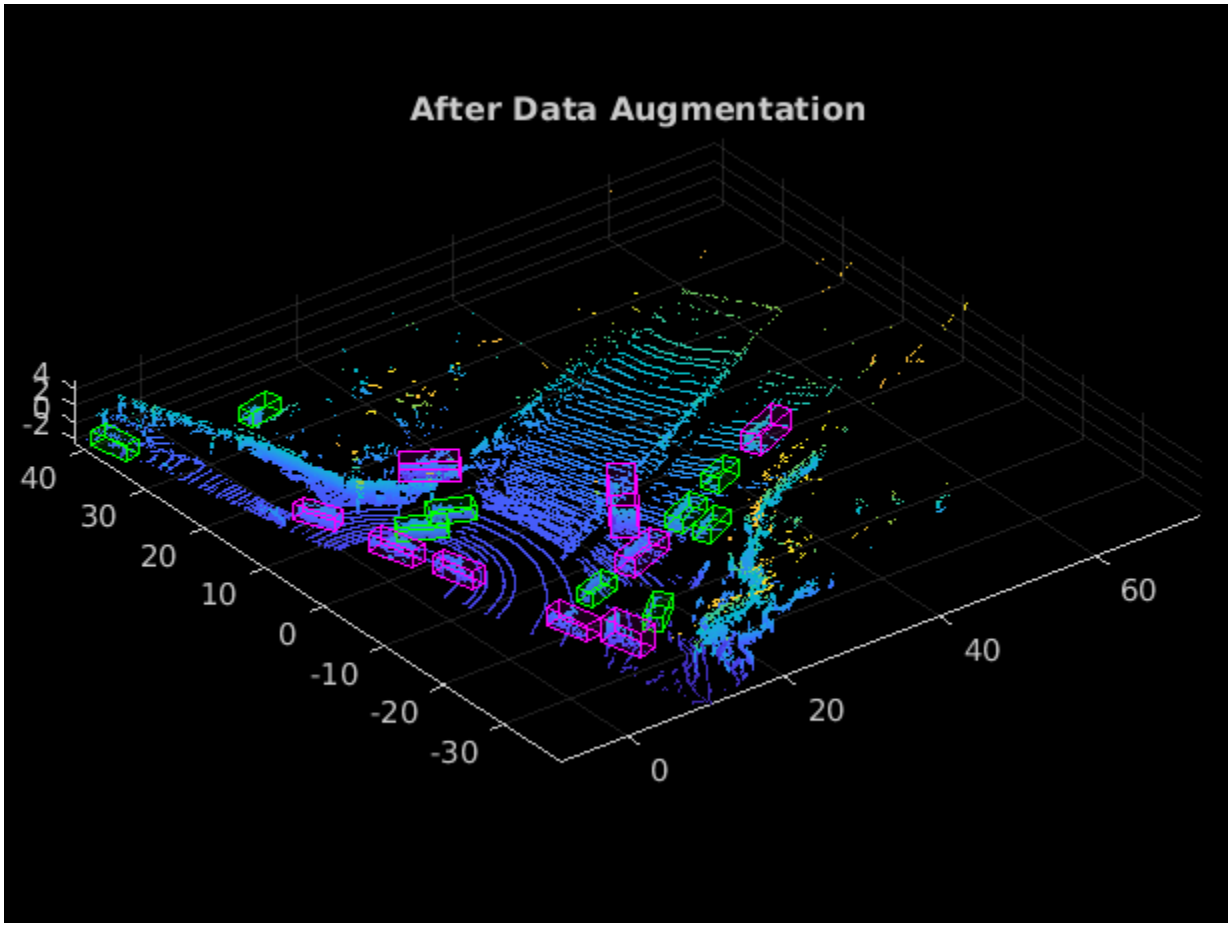
- Random flipping along the x-axis
- Random scaling by 5 percent

- Random rotation along the z-axis from [-pi/4, pi/4]
- Random translation by [0.2, 0.2, 0.1] meters along the x-, y-, and z-axis respectively

```
cdsAugmented = transform(cdsAugmented,@(x) augmentData(x));
```

Display an augmented point cloud along with ground truth augmented boxes using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example.

```
augData = read(cdsAugmented);
augptCld = augData{1,1};
augLabels = augData{1,2};
augClass = augData{1,3};

labelsCar = augLabels(augClass=='Car',:);
labelsTruck = augLabels(augClass=='Truck',:);

helperDisplay3DBoxesOverlaidPointCloud(augptCld.Location,labelsCar,'green',...
    labelsTruck,'magenta','After Data Augmentation');
```




```
reset(cdsAugmented);
```

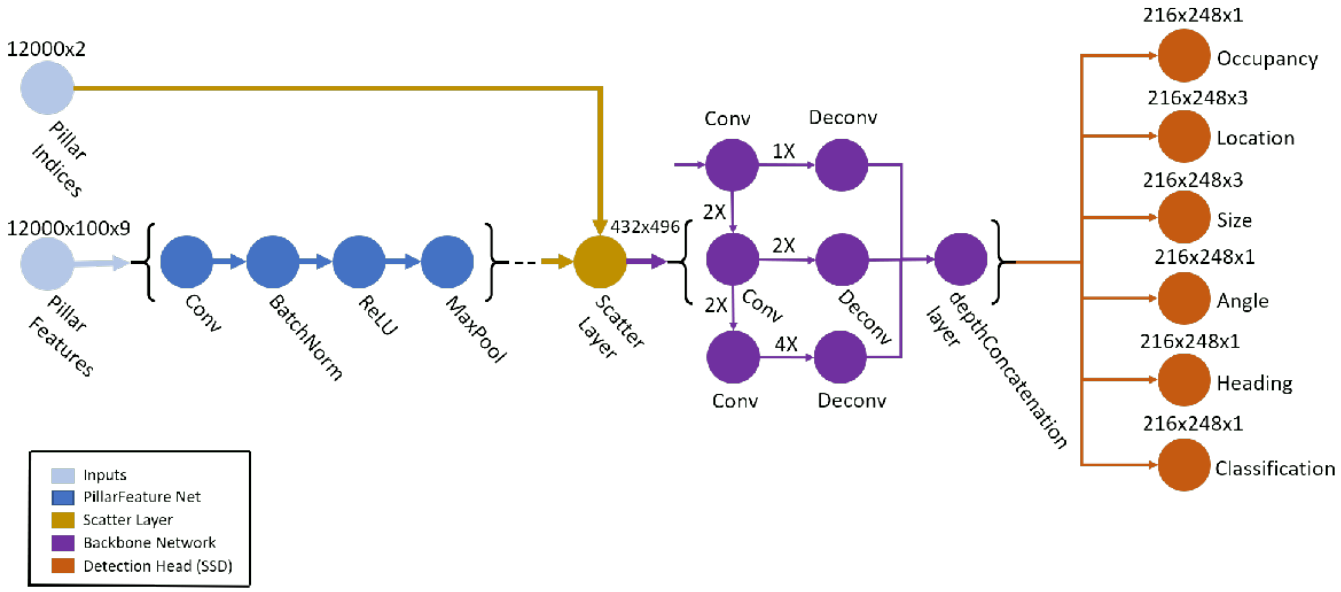
Create PointPillars Object Detector

Use the `pointPillarsObjectDetector` (Lidar Toolbox) function to create a PointPillars object detection network automatically. The PointPillars network uses a simplified version of the PointNet

network that takes pillar features as input. For each pillar feature, the network applies a linear layer, followed by batch normalization and ReLU layers. Finally, the network applies a max-pooling operation over the channels to get high-level encoded features. These encoded features are scattered back to the original pillar locations to create a pseudo-image. The network then processes the pseudo-image with a 2-D convolutional backbone followed by various SSD detection heads to predict the 3-D bounding boxes along with its classes.

The PointPillars network present in the PointPillars detector is illustrated in the following diagram.

You can use Deep Network Designer to create the network shown in the diagram.



The `pointPillarsObjectDetector` function requires you to specify several inputs that parameterize the PointPillars network:

- Class names
- Anchor boxes
- Point cloud range
- Voxel size
- Number of prominent pillars
- Number of points per pillar

```
% Define number of prominent pillars.
P = 12000;

% Define number of points per pillar.
N = 100;

% Estimate anchor boxes from training data.
anchorBoxes = calculateAnchorsPointPillars(trainLabels);
classNames = trainLabels.Properties.VariableNames;

% Define the PointPillars detector.
detector = pointPillarsObjectDetector(pointCloudRange,classNames,anchorBoxes,...
    'VoxelSize',voxelSize,'NumPillars',P,'NumPointsPerPillar',N);
```

If more control is required over the PointPillars network architecture, you can design the network manually. For more information, see Design a PointPillars Network.

Train Pointpillars Object Detector

Specify the network training parameters using `trainingOptions`. Set `'CheckpointPath'` to a temporary location to enable saving of partially trained detectors during the training process. If training is interrupted, you can resume training from the saved checkpoint.

Train the detector using a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). To automatically detect if you have a GPU available, set `executionEnvironment` to `"auto"`. If you do not have a GPU, or do not want to use one for training, set `executionEnvironment` to `"cpu"`. To ensure the use of a GPU for training, set `executionEnvironment` to `"gpu"`.

```
executionEnvironment = "auto";
if canUseParallelPool
    dispatchInBackground = true;
else
    dispatchInBackground = false;
end

options = trainingOptions('adam',...
    'Plots',"none",...
    'MaxEpochs',60,...
    'MiniBatchSize',3,...
    'GradientDecayFactor',0.9,...
    'SquaredGradientDecayFactor',0.999,...
    'LearnRateSchedule',"piecewise",...
    'InitialLearnRate',0.0002,...
    'LearnRateDropPeriod',15,...
    'LearnRateDropFactor',0.8,...
    'ExecutionEnvironment',executionEnvironment,...
    'DispatchInBackground',dispatchInBackground,...
    'BatchNormalizationStatistics','moving',...
    'ResetInputNormalization',false,...
    'CheckpointPath',tempdir);
```

Use `trainPointPillarsObjectDetector` (Lidar Toolbox) function to train the PointPillars object detector if `doTraining` is true. Otherwise, load the pretrained detector.

```
if doTraining
    [detector,info] = trainPointPillarsObjectDetector(cdsAugmented,detector,options);
else
    pretrainedDetector = load('pretrainedPointPillarsDetector.mat','detector');
    detector = pretrainedDetector.detector;
end
```

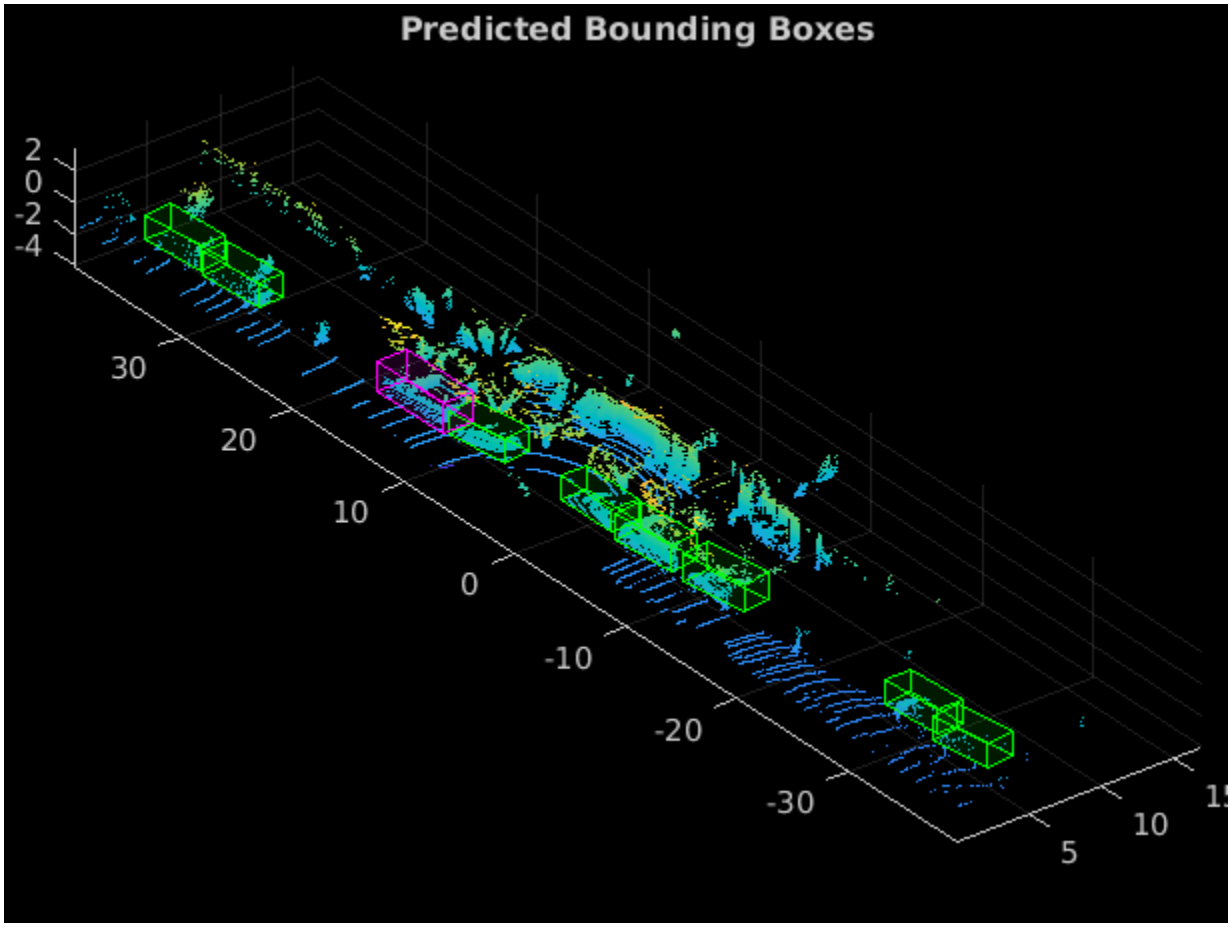
Generate Detections

Use the trained network to detect objects in the test data:

- Read the point cloud from the test data.
- Run the detector on the test point cloud to get the predicted bounding boxes and confidence scores.

- Display the point cloud with bounding boxes using the `helperDisplay3DBoxesOverlaidPointCloud` helper function, defined at the end of the example.

```
ptCloud = testData{45,1};
gtLabels = testLabels(45,:);

% Specify the confidence threshold to use only detections with
% confidence scores above this value.
confidenceThreshold = 0.5;
[box,score,labels] = detect(detector,ptCloud,'Threshold',confidenceThreshold);

boxlabelsCar = box(labels'=='Car',:);
boxlabelsTruck = box(labels'=='Truck',:);

% Display the predictions on the point cloud.
helperDisplay3DBoxesOverlaidPointCloud(ptCloud.Location,boxlabelsCar,'green',...
    boxlabelsTruck,'magenta','Predicted Bounding Boxes');
```



Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of point cloud data to measure the performance.

```
numInputs = 50;
```

```
% Generate rotated rectangles from the cuboid labels.
bds = boxLabelDatastore(testLabels(1:numInputs,:));
groundTruthData = transform(bds,@(x) createRotRect(x));

% Set the threshold values.
nmsPositiveIoUThreshold = 0.5;
confidenceThreshold = 0.25;

detectionResults = detect(detector,testData(1:numInputs,:),...
    'Threshold',confidenceThreshold);

% Convert to rotated rectangles format for calculating metrics
for i = 1:height(detectionResults)
    box = detectionResults.Boxes{i};
    detectionResults.Boxes{i} = box(:,[1,2,4,5,7]);
end

metrics = evaluateDetectionAOS(detectionResults,groundTruthData,...
    nmsPositiveIoUThreshold);
disp(metrics(:,1:2))
```

```
              AOS        AP
            _______    _______

    Car     0.89735    0.89735
    Truck     0.758      0.758
```

Helper Functions

```
function helperDownloadPandasetData(outputFolder,lidarURL)
% Download the data set from the given URL to the output folder.

    lidarDataTarFile = fullfile(outputFolder,'Pandaset_LidarData.tar.gz');

    if ~exist(lidarDataTarFile,'file')
        mkdir(outputFolder);

        disp('Downloading PandaSet Lidar driving data (5.2 GB)...');
        websave(lidarDataTarFile,lidarURL);
        untar(lidarDataTarFile,outputFolder);
    end

    % Extract the file.
    if (~exist(fullfile(outputFolder,'Lidar'),'dir'))...
            &&(~exist(fullfile(outputFolder,'Cuboids'),'dir'))
        untar(lidarDataTarFile,outputFolder);
    end

end

function helperDisplay3DBoxesOverlaidPointCloud(ptCld,labelsCar,carColor,...
    labelsTruck,truckColor,titleForFigure)
% Display the point cloud with different colored bounding boxes for different
% classes.
    figure;
    ax = pcshow(ptCld);
    showShape('cuboid',labelsCar,'Parent',ax,'Opacity',0.1,...
        'Color',carColor,'LineWidth',0.5);
```

```
    hold on;
    showShape('cuboid',labelsTruck,'Parent',ax,'Opacity',0.1,...
        'Color',truckColor,'LineWidth',0.5);
    title(titleForFigure);
    zoom(ax,1.5);
end
```

References

[1] Lang, Alex H., Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. "PointPillars: Fast Encoders for Object Detection From Point Clouds." In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (CVPR), 12689-12697. Long Beach, CA, USA: IEEE, 2019. https://doi.org/10.1109/CVPR.2019.01298.

[2] Hesai and Scale. PandaSet. https://scale.com/open-datasets/pandaset.

Signal Processing Examples

Learn Pre-Emphasis Filter Using Deep Learning

This example shows how to use a convolutional deep network to learn a pre-emphasis filter for speech recognition. The example uses a learnable short-time Fourier transform (STFT) layer to obtain a time-frequency representation suitable for use with 2-D convolutional layers. The use of a learnable STFT enables a gradient-based optimization of the pre-emphasis filter weights.

Data

Clone or download the Free Spoken Digit Dataset (FSDD), available at https://github.com/Jakobovski/free-spoken-digit-dataset. FSDD is an open data set, which means that it can grow over time. This example uses the version committed on 08/20/2020 which consists of 3000 recordings of the English digits 0 through 9 obtained from six speakers. The data is sampled at 8000 Hz.

This example assumes that you have downloaded the data into the folder corresponding to the value of `tempdir` in MATLAB. If you use a different folder, substitute that folder name for `tempdir` in the following code. Use audioDatastore to manage data access and ensure random division of data into training and test sets.

```
tempdir = '/mathworks/devel/sandbox/wking';
pathToRecordingsFolder = fullfile(tempdir,'free-spoken-digit-dataset','recordings');
ads = audioDatastore(pathToRecordingsFolder);
```

The helper function `helpergenLabels` creates a categorical array of labels from the FSDD files. The source code for `helpergenLabels` is listed in the appendix. List the classes and the number of examples in each class. It may take a few minutes to generate all the labels for this dataset.

```
ads.Labels = helpergenLabels(ads);
summary(ads.Labels)
```

```
     0      300
     1      300
     2      300
     3      300
     4      300
     5      300
     6      300
     7      300
     8      300
     9      300
```

Split the FSDD into training and test sets maintaining equal class proportions in each subset. For reproducible results, set the random number generator to its default value. Eighty percent, or 2400 recordings, are used for training. The remaining 600 recordings, 20% of the total, are held out for testing. Shuffle the files in the datastore once before creating the training and test sets.

```
rng default;
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8,0.2);
```

The recordings in FSDD are not equal in length. Use a transform so that each read from the datastore is padded or truncated to 8192 samples. The data are additionally cast to single-precision and a z-score normalization is applied.

```
transTrain = transform(adsTrain,@(x,info)helperReadData(x,info),'IncludeInfo',true);
transTest = transform(adsTest,@(x,info)helperReadData(x,info),'IncludeInfo',true);
```

Deep Convolutional Neural Network (DCNN) Architecture

This example uses a custom training loop with the following deep convolutional network.

```
numF = 12;
dropoutProb = 0.2;
layers = [
    sequenceInputLayer(1,'Name','input','MinLength',8192,...
         'Normalization',"none")

    convolution1dLayer(5,1,"name","pre-emphasis-filter",...
    "WeightsInitializer",@(sz)kronDelta(sz),"BiasLearnRateFactor",0)

    stftLayer('Window',hamming(1280),'OverlapLength',900,...
    'OutputMode','spatial','Name','STFT')

    convolution2dLayer(5,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2)

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numel(categories(ads.Labels)))
    softmaxLayer
    ];
dlnet = dlnetwork(layers);
```

The sequence input layer is followed by a 1-D convolution layer consisting of a single filter with 5 coefficients. This is a finite impulse response filter. Convolutional layers in deep learning networks by default implement an affine operation on the input features. To obtain a strictly linear (filtering) operation, use the default `'BiasInitializer'` which is `'zeros'` and set the bias learn rate factor of the layer to 0. This means that the bias is initialized to 0 and never changes during training. The network uses a custom initialization of the filter weights to be a scaled Kronecker delta sequence. This is an allpass filter, which performs no filtering of the input.

`stftLayer` takes the filtered batch of input signals and obtains their magnitude STFTs. The magnitude STFT is a 2-D representation of the signal, which is amenable to use in 2-D convolutional networks.

While the weights of the STFT are not changed here during training, the layer supports backpropagation, which enables the filter coefficients in the "pre-emphasis-filter" layer to be learned.

Network Training

Set the training options for the custom training loop. Use 25 epochs with a minbatch size of 128. Set the initial learn rate to 0.001.

```
NumEpochs = 25;
miniBatchSize = 128;
learnRate = 0.001;
```

In the custom training loop, use a `minibatchqueue` object. The `processSpeechMB` function reads in a minibatch and applies a one-hot encoding scheme to the labels.

```
mbqTrain = minibatchqueue(transTrain, 2,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFormat', {'CBT', 'CB'}, ...
    'MiniBatchFcn', @processSpeechMB);
```

Train the network and plot the loss for each iteration. Use an Adam optimizer to update the network learnable parameters. To plot the loss as training progress, set the value of `progress` in the following code to "training-progress".

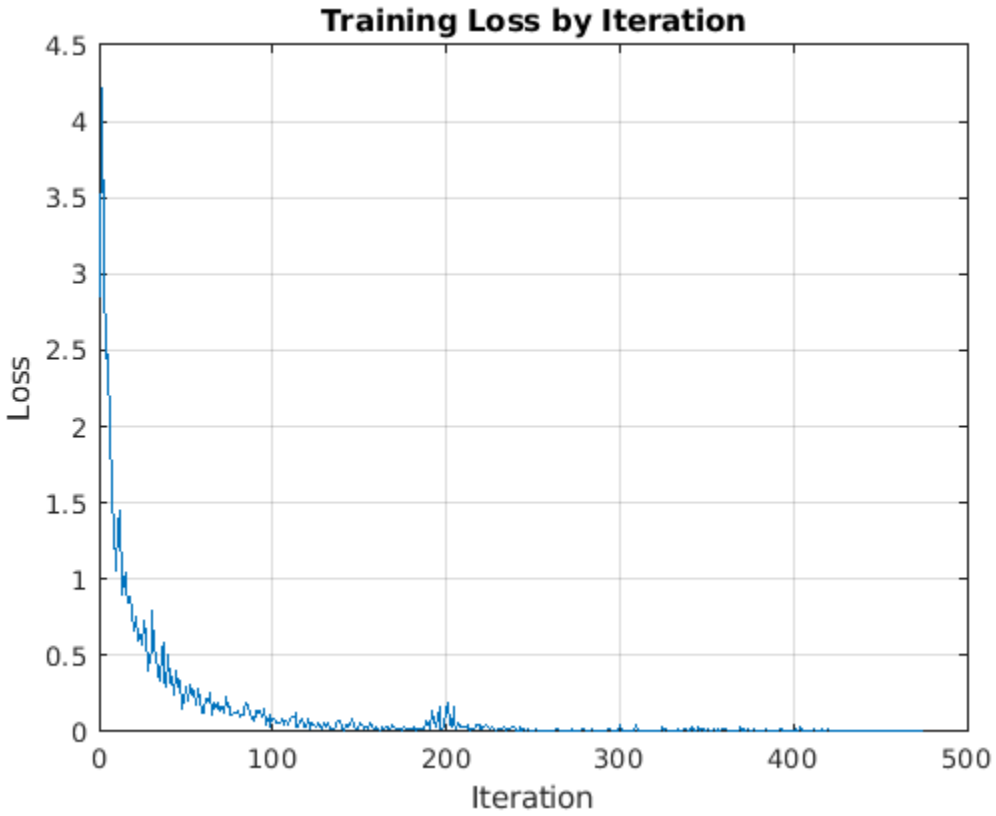
```
progress = "final-loss";
if progress == "training-progress"
    figure
    lineLossTrain = animatedline;
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end

% Initialize some training loop variables.
trailingAvg = [];
trailingAvgSq = [];
iteration = 0;
lossByIteration = 0;

% Loop over epochs. Time the epochs.
start = tic;

for epoch = 1:NumEpochs
    reset(mbqTrain)
    shuffle(mbqTrain)

    % Loop over mini-batches.
    while hasdata(mbqTrain)
        iteration = iteration + 1;

        % Get the next minibatch and one-hot coded targets
        [dlX,Y] = next(mbqTrain);
```

```
        % Evaluate the model gradients and loss
        [gradients, loss, state] = dlfeval(@modelGradSTFT,dlnet,dlX,Y);
        if progress == "final-loss"
            lossByIteration(iteration) = loss;
        end

        % Update the network state
        dlnet.State = state;

        % Update the network parameters using an Adam optimizer.
        [dlnet,trailingAvg,trailingAvgSq] = adamupdate(...
            dlnet, gradients, trailingAvg, trailingAvgSq, iteration, learnRate);

        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        if progress == "training-progress"
            addpoints(lineLossTrain,iteration,loss)
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
        end

    end
    disp("Training loss after epoch " + epoch + ": " + loss);

end
```

```
Training loss after epoch 1: 0.78569
Training loss after epoch 2: 0.33833
Training loss after epoch 3: 0.27921
Training loss after epoch 4: 0.11701
Training loss after epoch 5: 0.15688
Training loss after epoch 6: 0.032381
Training loss after epoch 7: 0.021219
Training loss after epoch 8: 0.048071
Training loss after epoch 9: 0.019537
Training loss after epoch 10: 0.055428
Training loss after epoch 11: 0.029689
Training loss after epoch 12: 0.021452
Training loss after epoch 13: 0.023566
Training loss after epoch 14: 0.010125
Training loss after epoch 15: 0.0027084
Training loss after epoch 16: 0.0074854
Training loss after epoch 17: 0.0053942
Training loss after epoch 18: 0.029233
Training loss after epoch 19: 0.016945
Training loss after epoch 20: 0.0096544
Training loss after epoch 21: 0.0023757
Training loss after epoch 22: 0.0028348
Training loss after epoch 23: 0.0041876
Training loss after epoch 24: 0.0017663
Training loss after epoch 25: 0.000395
```

```
if progress == "final-loss"
        plot(1:iteration,lossByIteration)
        grid on
        title('Training Loss by Iteration')
        xlabel("Iteration")
```

```
        ylabel("Loss")
end
```



Test the trained network on the held-out test set. Use a `minibatchqueue` object with a minibatch size of 32.

```
miniBatchSize = 32;
mbqTest = minibatchqueue(transTest, 2,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFormat', {'CBT', 'CB'}, ...
    'MiniBatchFcn', @processSpeechMB);
```

Loop over the test set and predict the class labels for each minibatch.

```
numObservations = numel(adsTest.Files);
classes = string(unique(adsTest.Labels));

predictions = [];

% Loop over mini-batches.
while hasdata(mbqTest)
    % Read mini-batch of data.
    dlX = next(mbqTest);
    % Make predictions on the minibatch
    dlYPred = predict(dlnet,dlX,'Acceleration','none');
    % Determine corresponding classes.
    predBatch = onehotdecode(dlYPred,classes,1);
    predictions = [predictions predBatch];
```

```
end
```

Evaluate the classification accuracy on the 600 examples in the held-out test set.

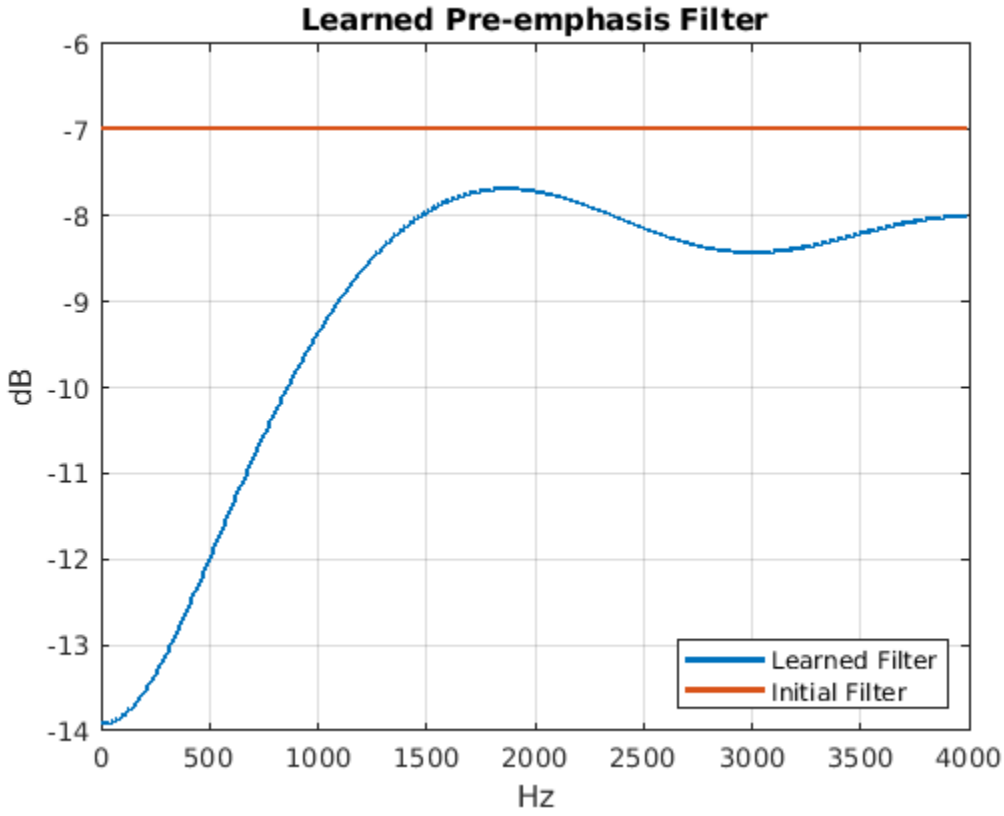
```
accuracy = mean(predictions' == categorical(adsTest.Labels))
```

```
accuracy = 0.9833
```

Test performance is approximately 98%. You can comment out the 1-D convolution layer and retrain the network without the pre-emphasis filter. The test performance without the pre-emphasis filter is also excellent at approximately 96%, but the use of the pre-emphasis filter makes a small improvement. It is noteworthy, that while the use of the learned pre-emphasis filter has only improved the test accuracy slightly, this was achieved by adding only 5 learnable parameters to the network.

To examine the learned pre-emphasis filter, extract the weights of the 1-D convolutional layer. Plot the frequency response. Recall that the sampling frequency of the data is 8 kHz. Because we initialized the filter to a scaled Kronecker delta sequence (allpass filter), we can easily compare the frequency response of the initialized filter with the learned response.

```
FIRFilter = dlnet.Layers(2).Weights;
[H,W] = freqz(FIRFilter,1,[],8000);
delta = kronDelta([5 1 1]);
Hinit = freqz(delta,1,[],4000);
plot(W,20*log10(abs([H Hinit])),'linewidth',2)
grid on
xlabel('Hz')
ylabel('dB')
legend('Learned Filter','Initial Filter','Location','SouthEast')
title('Learned Pre-emphasis Filter')
```



This example showed how to learn a pre-emphasis filter as a preprocessing step in a 2-D convolutional network based on short-time Fourier transforms of the signals. The ability of `stftLayer` to support backpropagation enabled gradient-based optimization of the filter weights inside the deep network. While this resulted in only a small improvement in the performance of the network on the test set, it achieved this improvement with a trivial increase in the number of learnable parameters.

Appendix: Helper Functions

```
function Labels = helpergenLabels(ads)
% This function is only for use in the "Learn Pre-Emphasis Filter using
% Deep Learning" example. It may change or be removed in a
% future release.

tmp = cell(numel(ads.Files),1);
expression = "[0-9]+_";
for nf = 1:numel(ads.Files)
    idx = regexp(ads.Files{nf},expression);
    tmp{nf} = ads.Files{nf}(idx);
end
Labels = categorical(tmp);
end

function [out,info] = helperReadData(x,info)
% This function is only for use in the "Learn Pre-Emphasis Filter using
% Deep Learning" example. It may change or be removed in a
% future release.
```

```
N = numel(x);
x = single(x);
if N > 8192
    x = x(1:8192);
elseif N < 8192
    pad = 8192-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = (x-mean(x))./std(x);
x = x(:)';
out = {x,info.Label};
end

function [dlX,dlY] = processSpeechMB(Xcell,Ycell)
% This function is only for use in the "Learn Pre-Emphasis Filter using
% Deep Learning" example. It may change or be removed in a
% future release.

Xcell = cellfun(@(x)reshape(x,1,1,[]),Xcell,'uni',false);
dlX = cat(2,Xcell{:});
dlY = cat(2,Ycell{:});
dlY = onehotencode(dlY,1);
end

function [grads, loss, state] = modelGradSTFT(net, X, T)
% This function is only for use in the "Learn Pre-Emphasis Filter using
% Deep Learning" example. It may change or be removed in a
% future release.

[y, state] = net.forward(X);
loss = crossentropy(y, T);
grads = dlgradient(loss,net.Learnables);
loss = double(gather(extractdata(loss)));
end

function delta = kronDelta(sz)
% This function is only for use in the "Learn Pre-Emphasis Filter using
% Deep Learning" example. It may change or be removed in a
% future release.

L = sz(1);
delta = zeros(L,sz(2),sz(3),'single');
delta(1) = 1/sqrt(L);

end
```

See Also

Apps
Deep Network Designer

Objects
`dlarray` | `dlnetwork` | `stftLayer`

Functions
`dlstft` | `stft` | `istft` | `stftmag2sig`

Related Examples

- “List of Deep Learning Layers” on page 1-21

Denoise EEG Signals Using Deep Learning Regression

This example shows how to remove electro-oculogram (EOG) noise from electroencephalogram (EEG) signals using the *EEGdenoiseNet* benchmark dataset [1] on page 12-0 and deep learning regression. The *EEGdenoiseNet* dataset contains 4514 clean EEG segments and 3400 ocular artifact segments that can be used to synthesize noisy EEG segments with the ground-truth clean EEG (the dataset also contains muscular artifact segments, but these will not be used in this example).

This example uses clean and EOG-contaminated EEG signals to train a long short-term memory (LSTM) model to remove the EOG artifacts. The regression model was trained with raw input signals and with signals transformed by the short-time Fourier transform (STFT). The STFT model improves performance especially at degraded SNR values.

Create the Dataset

The EEGdenoiseNet dataset contains 4514 clean EEG segments and 3400 EOG segments that can be used to generate three datasets for training, validating, and testing a deep learning model. The sample rate of all the signal segments is 256 Hz. For convenience, the dataset has been uploaded to this location: https://ssd.mathworks.com/supportfiles/SPT/data/EEGEOGDenoisingData.zip

Download the dataset using the `downloadSupportFile` function.

```
% Download the data
datasetZipFile = matlab.internal.examples.downloadSupportFile('SPT','data/EEGEOGDenoisingData.zi
datasetFolder = fullfile(fileparts(datasetZipFile),'EEG_EOG_Denoising_Dataset');
if ~exist(datasetFolder,'dir')
    unzip(datasetZipFile,fileparts(datasetZipFile));
end
```

After downloading the data, the location in `datasetFolder` contains two MAT files:

- `EEG_all_epochs.mat` contains a matrix with 4514 clean EEG segments of length 512 samples
- `EOG_all_epochs.mat` contains a matrix with 3400 EOG segments of length 512 samples

Use the `createDataset` helper function to generate training, validation, and testing datasets. The function combines clean EEG and EOG signals to generate pairs of clean and noisy EEG segments with different signal-to-noise ratios (SNR). For any EEG and EOG pair you can use the following combination equation to obtain a noisy segment with a given SNR:

$$\text{noisyEEG} = \text{EEG} + \lambda \cdot \text{EOG}$$

$$\text{SNR} = 10 \cdot \log_{10}\left(\frac{\text{rms(EEG)}}{\text{rms}(\lambda \cdot \text{EOG})}\right)$$

You vary parameter λ to control the artifact power and achieve a particular SNR value.

To create the training dataset `createDataset` combines the first 2720 pairs of EEG and EOG segments ten times each with random SNRs in the [-7, 2] dB interval for a total of 27200 training pairs. Each training pair is stored in a MAT file inside a folder named `train`. Each MAT file includes:

- A clean EEG segment (stored under a variable named `EEG`)
- An EOG segment (stored under a variable named `EOG`)
- A noisy EEG segment (stored under a variable named `noisyEEG`)

- The SNR of the noisy segment (stored under a variable named `SNR`)
- The sample rate value of the signal segments (stored under a variable named `Fs`)

To create the validation dataset `createDataset` combines the next 340 pairs of the EEG and EOG segments ten times each with random SNRs in the [–7, 2] dB interval for a total of 3400 validation segments. Validation data is stored in MAT files inside a folder named `validate`. Each MAT file contains the same variables as the ones described for the training set.

Finally, to create the test dataset `createDataset` combines the next 340 pairs of EEG and EOG segments ten times each with deterministic SNR values of –7, –6, –5, –4, –3, –2, –1, 0, 1, and 2 dB. The test data is stored in MAT files inside a folder named `test`. Test MAT files with the same SNR value are grouped under a common subfolder to make it easier to analyze the denoising performance of the trained model for a given SNR. For example, files with test signals with an SNR of -3 dB are stored in a folder with name `data_SNR_-3`.

Call the `createDataset` function to create the dataset (this may take a few seconds). Set the `createDatasetFlag` to false if you already have the dataset in the `datasetFolder` and want to skip this step.

```
createDatasetFlag = True;
if createDatasetFlag
    createDataset(datasetFolder);
end
```

Prepare Datastores to Consume the Data

The generated dataset is quite large (~430 MB), so it is convenient to use datastores to access the data without having to read it all at once into memory. Create signal datastores to access the training and validation data. Use the `SignalVariableNames` parameter to specify the variables you want to read from the MAT files (in the order you want them read). Also specify the `ReadOutputOrientation` as "row" to ensure the data is compatible with the LSTM network.

```
ds_Train = signalDatastore(fullfile(datasetFolder,"train"),SignalVariableNames=["noisyEEG","EEG"]
```

```
ds_Train =
  signalDatastore with properties:

                      Files:{
                             ' .../supportfiles/SPT/data/EEG_EOG_Denoising_Dataset/train/data_1.m
                             ' .../supportfiles/SPT/data/EEG_EOG_Denoising_Dataset/train/data_10
                             ' .../supportfiles/SPT/data/EEG_EOG_Denoising_Dataset/train/data_100
                              ... and 27197 more
                            }
                    Folders: {'/home/fboucher/Documents/MATLAB/Examples/R2021b/supportfiles/SPT/
    AlternateFileSystemRoots: [0×0 string]
                   ReadSize: 1
         SignalVariableNames: ["noisyEEG"    "EEG"]
       ReadOutputOrientation: "row"
```

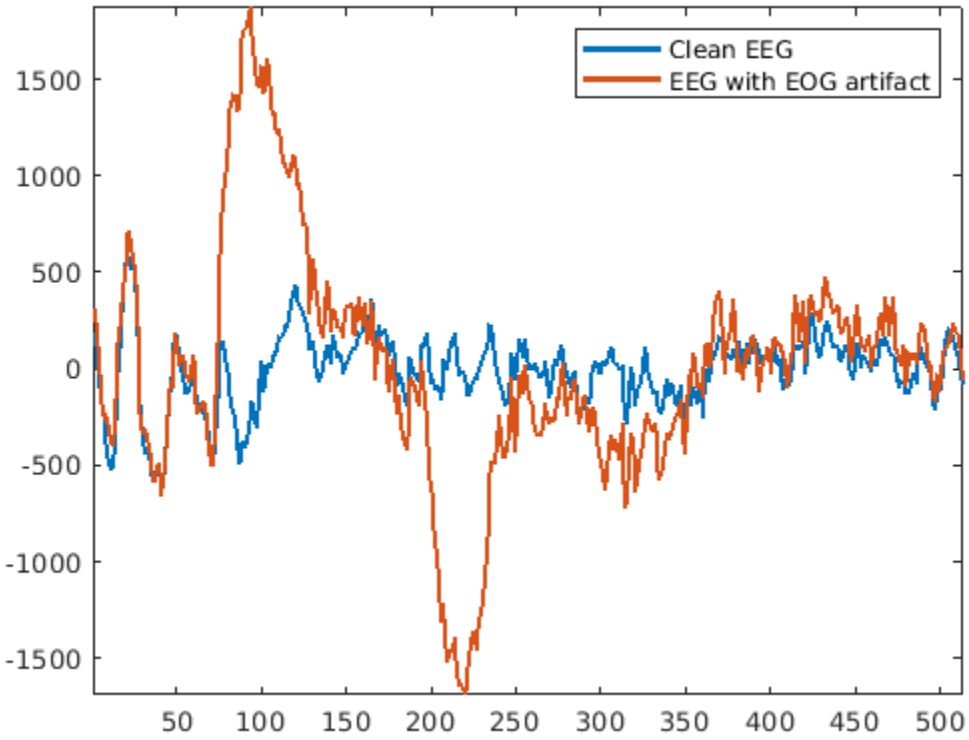
```
ds_Validate = signalDatastore(fullfile(datasetFolder,"validate"),SignalVariableNames=["noisyEEG"
```

Read the data from the first training file and plot the clean and noisy EEG signals. A call to preview or read methods of the datastore yields a 1x2 cell array with the first element containing a noisy EEG segment, and the second element containing a clean EEG segment.

```
data = preview(ds_Train)
```

```
data=1×2 cell array
    {[293.5459 312.8255 158.2003 54.3755 -122.9328 -245.2081 -263.6249 -231.0821 -265.4603 -326.4
```

```
plot([data{2}.' data{1}.'],LineWidth=2)
legend('Clean EEG','EEG with EOG artifact')
axis tight
```



The performance of a regression network is usually improved if the input and output signals are normalized. You can transform the signal datastores to apply normalization to each signal as it is read from disk. The `normalizeData helper` function is listed at the end of this example. It simply subtracts the signal mean and divides the result by the signal's standard deviation.

```
ds_Train_T = transform(ds_Train,@normalizeData);
ds_Validate_T = transform(ds_Validate,@normalizeData);
```

Train a Regression Model to Denoise EEG Signals

Train a network to denoise signals by passing noisy EEG signals into the network input and requesting the desired EEG clean ground-truth signals at the network output. A long-short term memory (LSTM) architecture is chosen because it is capable of learning features from time sequences.

Define the network architecture: the number of features is set to one as a single sequence is input to the network and a single sequence is output from the network. Use a dropout layer to reduce

overfitting of the model on the training data. Use a regression layer as the output layer since the model is being trained to perform regression. Note that normalization must be applied to input and output signals so it is more convenient to use transformed datastores than to use the `Normalization` option of the `sequenceInputLayer` that only normalizes the inputs.

```
numFeatures = 1;
numHiddenUnits = 100;

layers = [
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    dropoutLayer(0.2)
    fullyConnectedLayer(numFeatures)
    regressionLayer];
```

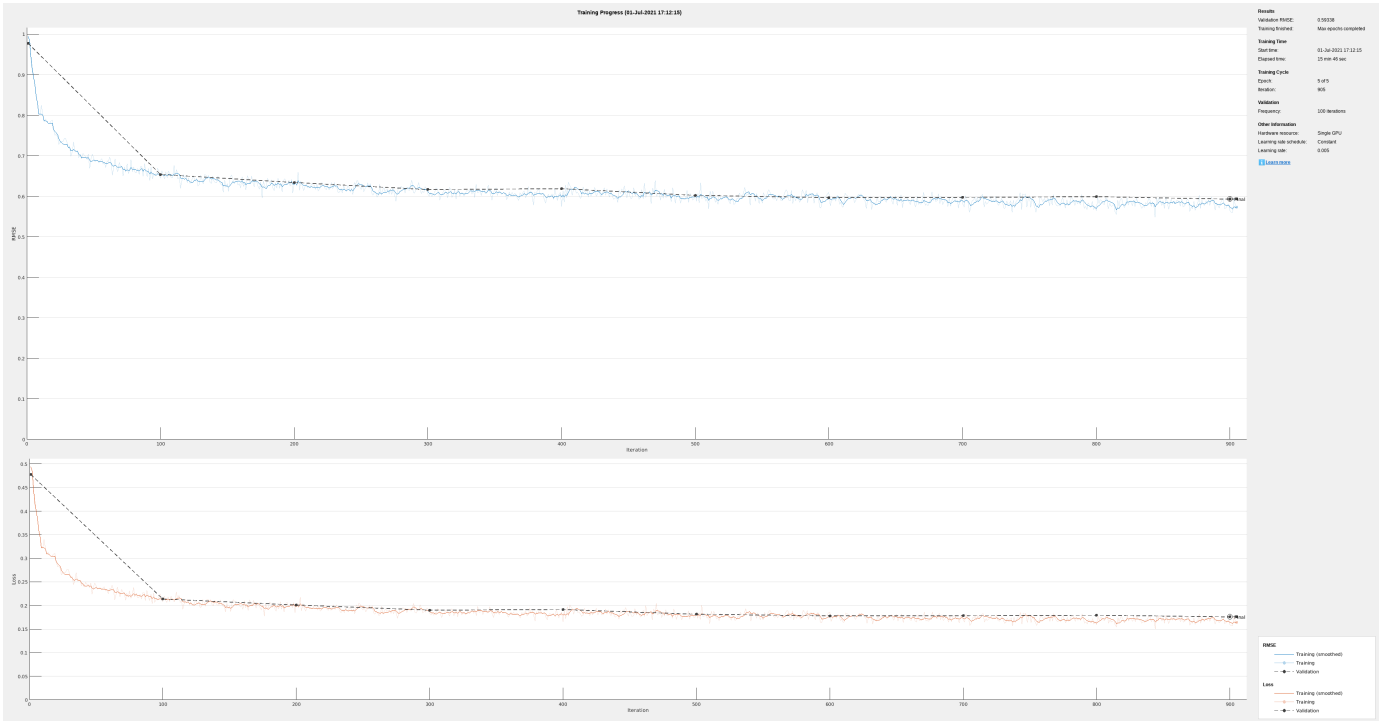
Define the training option parameters: use an Adam optimizer and choose to shuffle the data at every epoch. Also, specify the validation datastore `ds_Validate_T` as the source for the validation data.

```
maxEpochs = 5;
miniBatchSize = 150;

options = trainingOptions('adam', ...
    MaxEpochs=maxEpochs, ...
    MiniBatchSize=miniBatchSize, ...
    InitialLearnRate=0.005, ...
    GradientThreshold=1, ...
    Plots="training-progress", ...
    Shuffle="every-epoch", ...
    Verbose=false,...
    ValidationData=ds_Validate_T ,...
    ValidationFrequency=100, ...
    OutputNetwork="best-validation-loss");
```

Use the `trainNetwork` function to train the model. You can directly pass the transformed train datastore into the function because the datastore outputs a 1x2 cell array, with input and output signals, at each call to the `read` method.

The training steps will take several minutes. You can skip these steps by downloading the pre-trained networks using the selector below. If you want to train the network as the example runs, select '`Train Networks`'. If you want to skip the training steps, select '`Download Networks`' and a MAT file containing two pre-trained networks -`rawNet`, and stft`Net`- will be downloaded into your machine.

```
trainingFlag = Train networks ;
if trainingFlag == "Train networks"
    rawNet = trainNetwork(ds_Train_T,layers,options);
else
    % Download the pre-trained networks
    modelsZipFile = matlab.internal.examples.downloadSupportFile('SPT','data/EEGEOGDenoisingNetwo
    modelsFolder = fullfile(fileparts(datasetZipFile),'EEG_EOG_Denoising_Networks');
    if ~exist(modelsFolder,'dir')
        unzip(modelsZipFile,fileparts(modelsZipFile));
    end
    modelsFile = fullfile(modelsFolder,'trainedNetworks.mat');
    load(modelsFile)
end
```



Analyze the Denoising Performance of the Trained Model

Use the test dataset to analyze the denoising performance of the `rawNet` network. Recall that the test dataset contains multiple test files for each SNR value in [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2] dB. The performance metric is chosen as the mean-squared error (MSE) between the clean baseline EEG signal and the denoised EEG signal. The MSE of the clean EEG signal and the noisy EEG signal is also computed to show the worst-case MSE when no denoising is applied. At each SNR compute 340 MSE values for each of the 340 available test EEG segments and obtain the average MSE.

Create a `signalDatastore` to consume the test data and use a transformed datastore to setup data normalization. Since the data is now inside subfolders of the test folder, specify `IncludeSubfolders` as true. Further, use the folders2labels function to get the list of folder names for each file in the test dataset so that you can get data for each SNR.

```
ds_Test = signalDatastore(fullfile(datasetFolder,"test"),SignalVariableNames=["noisyEEG","EEG"],I
ds_Test_T = transform(ds_Test,@normalizeData);

% Get labels that contain the SNR value for each file in the datastore
labels = folders2labels(ds_Test)
```

```
labels = 3400×1 categorical
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
     data_SNR_-1
```

```
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
data_SNR_-1
  ⋮
```

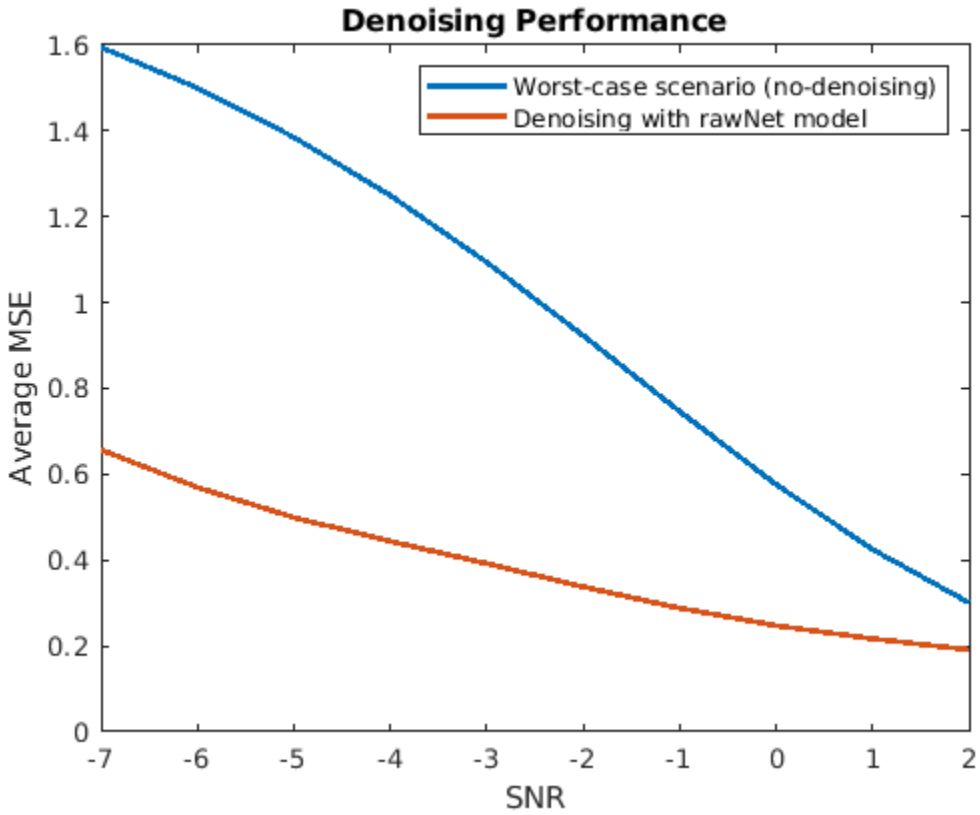
For each SNR value, denoise the test signals and compute the average MSE value. Use the `subset` function of the datastore to get a datastore pointing to the data for each SNR. To denoise a signal call the `predict` function passing the trained network and the noisy data as inputs.

```
SNRs = (-7:2);
MSE_Denoised_rawNet = zeros(numel(SNRs),1); % Measure denoising performance
MSE_No_Denoise = zeros(numel(SNRs),1); % Measure worst-case MSE when no denoising is applied

for idx = 1:numel(SNRs)
    lblIdx = find(labels == "data_SNR_"+num2str(SNRs(idx)));
    ds_Test_SNR = subset(ds_Test_T,lblIdx); % New datastore pointing to files with current SNR va

    % Denoise the data using the predict function of the trained model
    pred = predict(rawNet,ds_Test_SNR);

    % Use an array datastore to loop over the 340 denoised signals for the
    % current SNR value. Transform the datastore to add the normalization
    % step.
    ds_Pred = transform(arrayDatastore(pred,OutputType="same"),@normalizeData);

    mse = 0;
    mseWorstCase = 0;
    cnt = 0;
    while hasdata(ds_Pred)

        testData = read(ds_Test_SNR);
        denoisedData = read(ds_Pred);

        % MSE performance of denoiser - testData{2} contains clean EEG,
        % testData{1} contains noisy EEG.
        mse = mse + sum((testData{2} - denoisedData{1}).^2)/numel(denoisedData{1});

        % Worst-case MSE performance when no denoising is applied.
        % Convert data to single precession as denoisedData is single
        % precision.
```

```
        mseWorstCase = mseWorstCase + sum((single(testData{2}) - single(testData{1})).^2)/numel(
        cnt = cnt+1;
    end

    % Average MSE of denoised signals
    MSE_Denoised_rawNet(idx) = mse/cnt;

    % Worst-case average MSE
    MSE_No_Denoise(idx) = mseWorstCase/cnt;
end
```

Plot the average MSE results.

```
plot(SNRs,[MSE_No_Denoise,MSE_Denoised_rawNet],LineWidth=2)
xlabel("SNR")
ylabel("Average MSE")
title("Denoising Performance")
legend("Worst-case scenario (no-denoising)","Denoising with rawNet model")
```



Improve Performance Using Short-Time Fourier Transform Feature Extraction

A common approach to improve performance of a deep learning model is to use extracted features in place of the original raw signal data. The features provide a representation of the input data that makes it easier for the network to learn the most important aspects of the signals.

Choose a short-time Fourier transformation (STFT) with a window length of 64 samples and overlap length of 63 samples. This transformation will effectively create 33 complex features with a length of

449 samples each. LSTM networks do not support complex inputs so the complex features can be separated into real and imaginary components by stacking the real part of the features on top of the imaginary part of the features to yield 66 real features each one of length 449 samples.

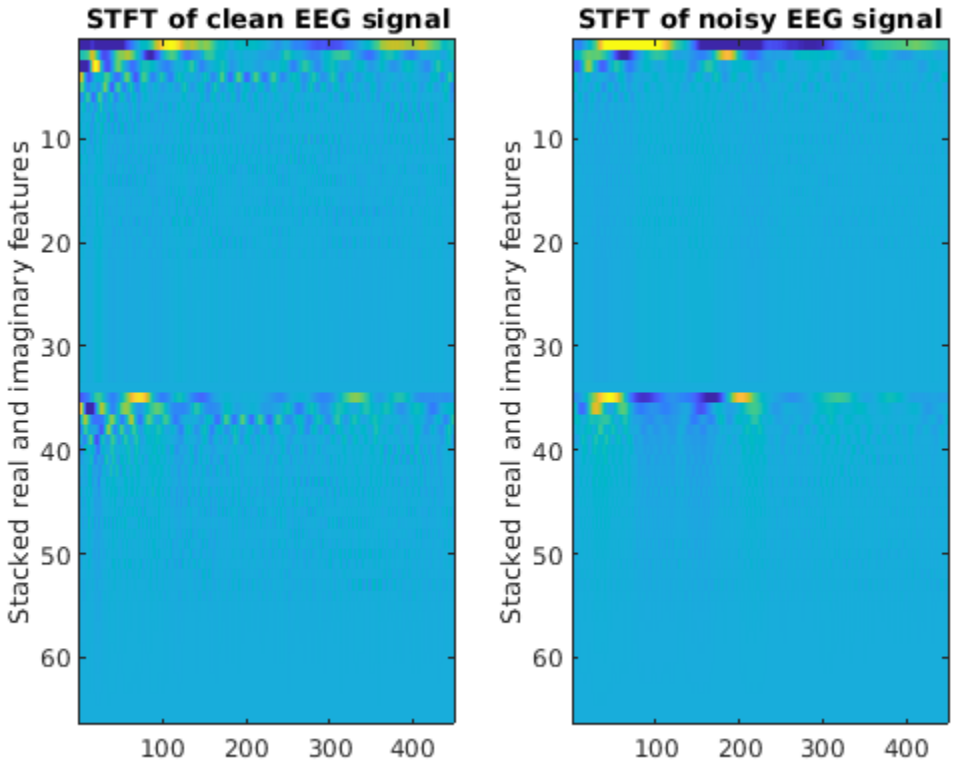
```
winLength = 64;
overlapLength = 63;
```

The `transformSTFT` helper function listed at the end of this example normalizes the input signal and then computes its STFT. The function stacks the real and imaginary components to create a real output matrix. Further, if a GPU is available, the function moves the data to the GPU to accelerate the STFT computations and mitigate the increased complexity of computing the transforms. If you do not wish to use the GPU, set `useGPUFlag` to `false`.

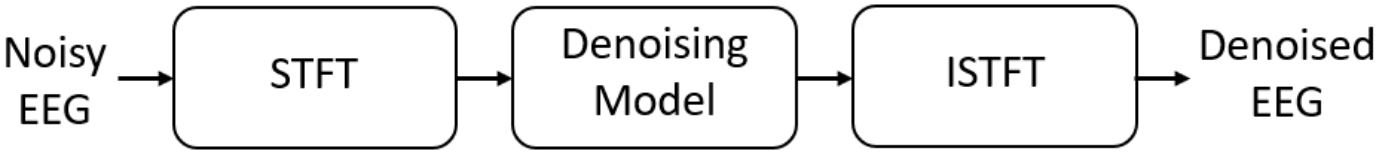
```
useGPUFlag = True;
```

Compute and plot the STFT of a pair of clean and noisy EEG signals using the `transformSTFT helper function.`

```
data = preview(ds_Train);
P = transformSTFT(data,winLength,overlapLength,useGPUFlag);
figure
subplot(1,2,1)
h = imagesc(P{2});
h.Parent.CLim = [-40 57];
title('STFT of clean EEG signal')
ylabel("Stacked real and imaginary features")
subplot(1,2,2)
h = imagesc(P{1});
h.Parent.CLim = [-40 57];
ylabel("Stacked real and imaginary features")
title('STFT of noisy EEG signal')
```



The idea is to train a network so that it can produce denoised STFT signal representations based on STFT inputs corresponding to noisy signals. Note that the target outcome is a denoised signal, not its denoised STFT representation, so a final step must be added to compute the inverse STFT (ISTFT) to recover the denoised signal as depicted on the block diagram below.



The helper function, `transformISTFT`, listed at the end of this example takes the denoised STFT network output, converts the stacked real and imaginary features back to complex features and computes the inverse STFT. As a final step the function normalizes the resulting signal. If a GPU is available and `useGPU`flag is true, the function performs all the computations in the GPU to reduce the processing time.

Create train, validation, and test datastores to apply STFT using the `transformSTFT` function.

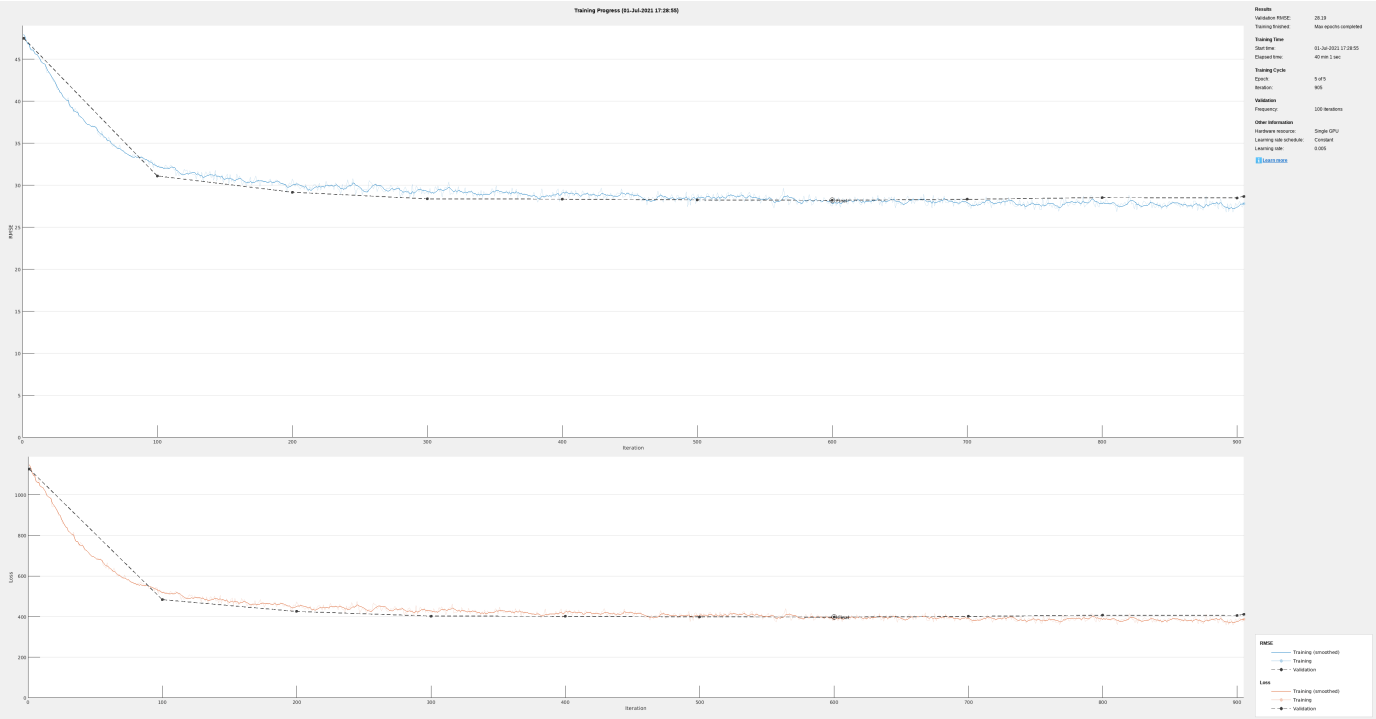
```
ds_Train_STFT = transform(ds_Train,@(d,wl,ol,fl)transformSTFT(d,winLength,overlapLength,useGPUFla
ds_Validate_STFT = transform(ds_Validate,@(d,wl,ol,fl)transformSTFT(d,winLength,overlapLength,use
ds_Test_STFT = transform(ds_Test,@(d,wl,ol,fl)transformSTFT(d,winLength,overlapLength,useGPUFlag)
```

Update the network architecture to account for 66 input and output features and specify the new validation data in the training options. Every other network parameter or option is unchanged.

```
numFeatures = winLength + 2;

layers = [
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    dropoutLayer(0.2)
    fullyConnectedLayer(numFeatures)
    regressionLayer];

options.ValidationData = ds_Validate_STFT;
```

Train the network if `trainingFlag` is `"Train networks"`.

```
if trainingFlag == "Train networks"
    stftNet = trainNetwork(ds_Train_STFT,layers,options);
end
```



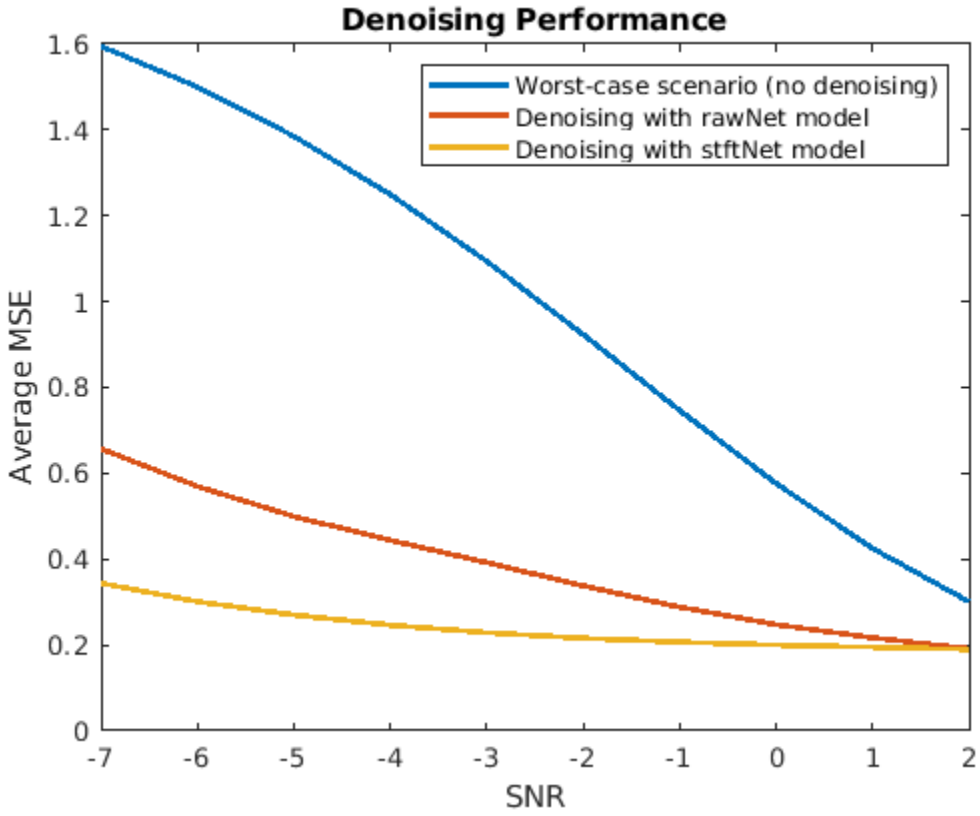
Use the trained network to denoise EEG signals using the test data. Compute average MSE values by comparing denoised and clean baseline EEG signals.

```
MSE_Denoised_stftNet = zeros(numel(SNRs),1); % Measure denoising performance
for idx = 1:numel(SNRs)
    lblIdx = find(labels == "data_SNR_"+num2str(SNRs(idx)));
    % New datastores pointing to files with current SNR value
    ds_Test_SNR = subset(ds_Test_T,lblIdx); % Raw EEG signals to compute MSE
    ds_Test_STFT_SNR = subset(ds_Test_STFT,lblIdx); % STFT transforms

    % Denoise the data using the predict function of the trained model.
    pred = predict(stftNet,ds_Test_STFT_SNR);

    % Use an array datastore to loop over the 340 denoised signals for the
    % current SNR value. Transform the datastore to compute the inverse
```

```
    % STFT and recover the actual denoised signal.
    ds_Pred = transform(arrayDatastore(pred,OutputType="same"),@(P,wl,ol)transformISTFT(P,winLeng

    mse = 0;
    cnt = 0;
    while hasdata(ds_Pred)

        testData = read(ds_Test_SNR);
        denoisedData = read(ds_Pred);

        % MSE performance of denoiser - testData{2} contains clean EEG
        mse = mse + sum((testData{2} - denoisedData).^2)/numel(denoisedData);
        cnt = cnt+1;
    end

    % Average MSE of denoised signals
    MSE_Denoised_stftNet(idx) = mse/cnt;
end
```

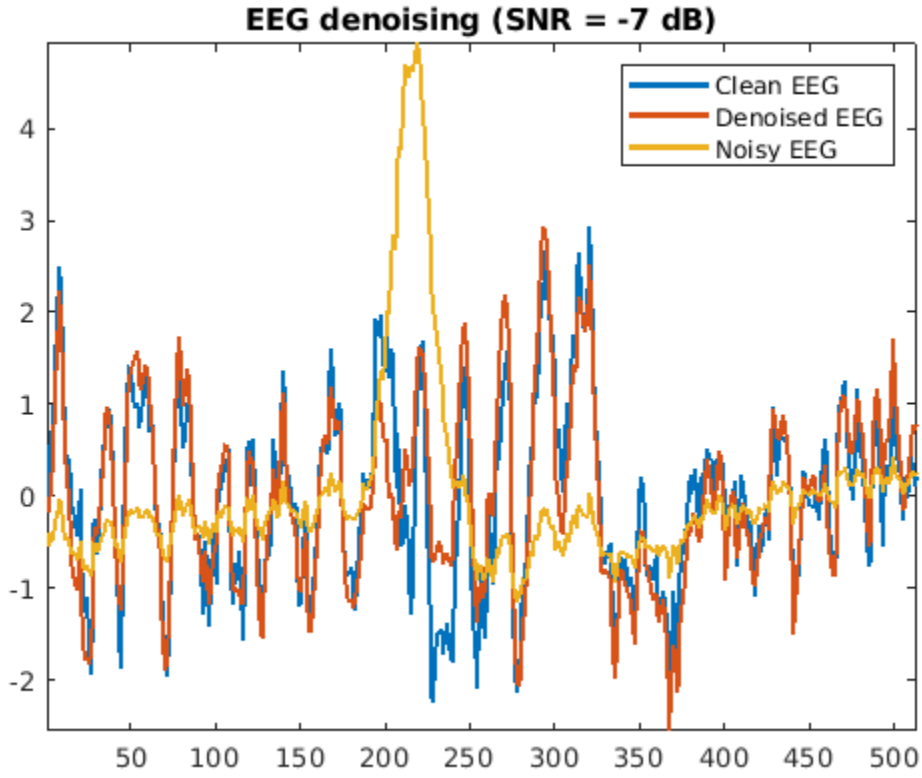
Plot the average MSE obtained with no denoising, denoising with a network trained with raw input signals, and denoising with a network trained with STFT transformed signals. You can see that the addition of the STFT step has improved the performance especially at the lower SNR values.

```
figure
plot(SNRs,[MSE_No_Denoise,MSE_Denoised_rawNet,MSE_Denoised_stftNet],LineWidth=2)
xlabel("SNR")
ylabel("Average MSE")
title("Denoising Performance")
legend("Worst-case scenario (no denoising)","Denoising with rawNet model","Denoising with stftNe
```



Plot noisy and denoised signals for different SNRs. The `getRandomEEG` helper function listed at the end of this example gets a random EEG signal with a specified SNR from the test dataset.

```
SNR = -7; % dB
data = getRandomEEG(datasetFolder,SNR);
noisyEEG = normalizeData(data{1});
cleanEEG = normalizeData(data{2});
stftInput = transformSTFT(noisyEEG,winLength,overlapLength,useGPUFlag);
denoisedEEG = transformISTFT(predict(stftNet,stftInput),winLength,overlapLength);

plot([cleanEEG.' denoisedEEG.' noisyEEG.'],LineWidth=2)
title("EEG denoising (SNR = " + SNR + " dB)")
legend("Clean EEG", "Denoised EEG","Noisy EEG")
axis tight
```



Conclusion

In this example you learned how to train a deep network to perform regression for signal denoising. You compared two models, one trained with raw clean and noisy EEG signals, the other trained with features extracted using a short-time Fourier transform. You learned that you can use complex features by stacking their real and imaginary components and treating them as independent real features. The use of STFT sequences provides greater performance improvement at worse SNRs and both approaches converge in performance as the SNR improves.

References

[1] Haoming Zhang, Mingqi Zhao, Chen Wei, Dante Mantini, Zherui Li, Quanying Liu. "A benchmark dataset for deep learning solutions of EEG denoising." https://arxiv.org/abs/2009.11662

Helper Functions

`normalizeData` - this function normalizes input signals by subtracting the mean and dividing by the standard deviation.

```
function y = normalizeData(x)
% This function is only intended to support examples in the Signal
% Processing Toolbox. It may be changed or removed in a future release.

if iscell(x)
    y = cell(1,numel(x));
    y{1} = (x{1}-mean(x{1}))/std(x{1});
```

```
    if numel(x) == 2
        y{2} = (x{2}-mean(x{2}))/std(x{2});
    end
else
    y = (x - mean(x))/std(x);
end
end
```

`transformSTFT` - this function normalizes the signals in input `data` and computes their short-time Fourier transform. It converts the complex STFT results into a real matrix by stacking the real and imaginary components one on top of the other.

```
function P = transformSTFT(data,winLength,overlapLength,useGPUFlag)
% This function is only intended to support examples in the Signal
% Processing Toolbox. It may be changed or removed in a future release.

if ~iscell(data)
    data = {data};
end

P = cell(1,numel(data));

x = data{1};
if useGPUFlag
    x = gpuArray(x);
end
x = normalizeData(x);
y = stft(x,Window=rectwin(winLength),OverlapLength=overlapLength,FrequencyRange="onesided");
P{1} = [real(y);imag(y)];

if numel(data) == 2
    x = data{2};
    if useGPUFlag
        x = gpuArray(x);
    end
    x = normalizeData(x);
    y = stft(x,Window=rectwin(winLength),OverlapLength=overlapLength,FrequencyRange="onesided");
    P{2} = [real(y);imag(y)];
end
end
```

`transformISTFT` - this function takes a matrix with stacked real and imaginary STFT elements and combines them back to a complex STFT matrix. The function then computes the inverse STFT transform and normalizes the resulting reconstructed signals.

```
function data = transformISTFT(P,winLength,overlapLength)
% This function is only intended to support examples in the Signal
% Processing Toolbox. It may be changed or removed in a future release.
PP = P{1};
NumRows = size(PP,1);
X = PP(1:NumRows/2,:)+1i*PP(1+NumRows/2:end,:);
data = istft(X,Window=rectwin(winLength),OverlapLength=overlapLength,ConjugateSymmetric=true,Fre
data = normalizeData(data);
end
```

`createDataset` - this function combines clean EEG signal segments with EOG segments to create training, validation and testing datasets to train an EEG denoiser neural network.

```
function createDataset(dataDir)
% This function is only intended to support examples in the Signal
% Processing Toolbox. It may be changed or removed in a future release.

% Create training, validation, and testing datasets consisting of clean EEG
% signals and noisy EEG signals contaminated by EOG segments.

load(fullfile(dataDir,"EEG_all_epochs.mat"),"EEG_all_epochs");
load(fullfile(dataDir,"EOG_all_epochs.mat"),"EOG_all_epochs");

EEG_all_epochs = EEG_all_epochs(1:3400,:).';
EOG_all_epochs = EOG_all_epochs.';
Fs = 256;
trainingPercentage = 80;
validationPercentage = 10;
N = size(EEG_all_epochs,2);

% Create a training dataset consisting of mat files containing two signals
% - a clean EEG signal, and an EEG signal contaminated by EOG artifacts.
% Combine each of 2720 pairs of EEG and EOG segments ten times with random
% SNRs in the range -7dB to 2dB to obtain 27200 training segments.

EEG_training = EEG_all_epochs(:,1:N*trainingPercentage/100);
EOG_training = EOG_all_epochs(:,1:N*trainingPercentage/100);

M = size(EEG_training,2);
cnt = 0;
if ~exist(fullfile(dataDir,"train"),'dir')
    mkdir(fullfile(dataDir,"train"))
end
for idx = 1:M
    for kk = 1:10
        cnt = cnt + 1;
        EEG = EEG_training(:,idx).';
        EOG = EOG_training(:,idx).';
        [noisyEEG,SNR] = createNoisySegment(EEG,EOG,[-7,2]);
        save(fullfile(dataDir,"train","data_" + num2str(cnt) + ".mat"),"EEG","EOG","noisyEEG","Fs
    end
end

% Create a validation dataset by combining 340 pairs of EEG and EOG
% segments ten times with random SNRs in (-7:2) dB

EEG_validation = EEG_all_epochs(:,1+N*trainingPercentage/100:N*trainingPercentage/100+N*validatio
EOG_validation = EOG_all_epochs(:,1+N*trainingPercentage/100:N*trainingPercentage/100+N*validatio

M = size(EEG_validation,2);
cnt = 0;
if ~exist(fullfile(dataDir,"validate"),'dir')
    mkdir(fullfile(dataDir,"validate"))
end
for idx = 1:M
    for kk = 1:10
        cnt = cnt + 1;
        EEG = EEG_validation(:,idx).';
        EOG = EOG_validation(:,idx).';
        [noisyEEG,SNR] = createNoisySegment(EEG,EOG,[-7,2]);
```

```
        save(fullfile(dataDir,"validate","data_" + num2str(cnt) + ".mat"),"EEG","EOG","noisyEEG",
    end
end

% Create a test dataset by combining 340 pairs of EEG and EOG segments ten
% times with 10 SNR values [-7 -6 -5 -4 -3 -2 -1 0 1 2] dB. Store the
% training sets in folders with names that identify the SNR value so that
% it is easy to analyze performance by accessing files with a specific SNR.

EEG_test = EEG_all_epochs(:,1+N*trainingPercentage/100+N*validationPercentage/100:end);
EOG_test = EOG_all_epochs(:,1+N*trainingPercentage/100+N*validationPercentage/100:end);

M = size(EEG_test,2);
SNRVect = (-7:2);
for kk = 1:numel(SNRVect)
    cnt = 0;
    if ~exist(fullfile(dataDir,"test","data_SNR_" + num2str(SNRVect(kk))),'dir')
        mkdir(fullfile(dataDir,"test","data_SNR_" + num2str(SNRVect(kk))));
    end
    for idx = 1:M
        cnt = cnt + 1;
        EEG = EEG_test(:,idx).';
        EOG = EOG_test(:,idx).';
        [noisyEEG,SNR] = createNoisySegment(EEG,EOG,SNRVect(kk));
        save(fullfile(dataDir,"test","data_SNR_" + num2str(SNR)+"/" + "data_"+num2str(cnt) + ".ma
    end
end
end

function [y,SNROut] = createNoisySegment(eeg,artifact,SNR)
% Combine EEG and artifact signals with a specified SNR in dB. If SNR is a
% two-element vector, its value is chosen randomly from a uniform
% distribution over the interval [SNR(1) SNR(2)]

if numel(SNR) == 2
    SNR = SNR(1) + (SNR(2)-SNR(1)).*rand(1,1);
end

k = 10^(SNR/10);
lambda = (1/k)*rms(eeg)/rms(artifact);

y = eeg + lambda * artifact;

SNROut = SNR;
end
```

getRandomEEG - this function reads the data from a random EEG test file with a specified SNR.

```
function data = getRandomEEG(datasetFolder,SNR)
sds  = signalDatastore(fullfile(datasetFolder,"test","data_SNR_"+num2str(SNR)),SignalVariableName
n = numel(sds.Files);
idx = randi(n,1);
```

```
data = read(subset(sds,idx));
end
```

See Also

`folders2labels` | `signalDatastore` | `trainNetwork`

Hand Gesture Classification Using Radar Signals and Deep Learning

This example shows how to classify ultra-wideband (UWB) impulse radar signal data using a multiple-input, single-output convolutional neural network (CNN).

Introduction

Movement-based signal data acquired using sensors, like UWB impulse radars, contain patterns specific to different gestures. Correlating motion data with movement benefits several avenues of work. For example, hand gesture recognition is important for contactless human-computer interaction. This example aims to use a deep learning solution to automate feature extraction from patterns within a hand gesture dataset and provide a label for every signal sample.

UWB-gestures is a publicly available dataset of dynamic hand gestures [1 on page 12-0]. It contains a total of 9600 samples gathered from 8 different human volunteers. To obtain each recording, the examiners placed a separate UWB impulse radar at the left, top, and right sides of their experimental setup, resulting in 3 received radar signal data matrices. Volunteers performed hand gestures from a gesture vocabulary consisting of 12 dynamic hand movements:

1. left-right swipe (L-R swipe)
2. right-left swipe (R-L swipe)
3. up-down swipe (U-D swipe)
4. down-up swipe (D-U swipe)
5. diagonal-left-right-up-down swipe (Diag-LR-UD swipe)
6. diagonal-left-right-down-up swipe (Diag-LR-DU swipe)
7. diagonal-right-left-up-down swipe (Diag-RL-UD swipe)
8. diagonal-right-left-down-up swipe (Diag-RL-DU swipe)
9. clockwise rotation
10. counterclockwise rotation
11. inward push
12. empty gesture

As each hand gesture motion is captured by 3 independent UWB impulse radars, we will use a CNN architecture that accepts 3 signals as separate inputs. The CNN model will extract feature information from each signal before combining it to make a final gesture label prediction. As such, a multiple-input, single-output CNN will use minimally pre-processed radar signal data matrices to classify different gestures.

Download the Data

Each radar signal data matrix is labeled as the hand gesture that generated it. 8 different human volunteers performed 12 separate hand gestures, for a total of 96 trials stored in 96 MAT-files. Each MAT-file contains 3 radar data matrices, corresponding to the 3 radars used in the experimental setup. They are named `Left`, `Top`, and `Right`. The files are available at the following location:

https://ssd.mathworks.com/supportfiles/SPT/data/uwb-gestures.zip

Download the data files into your MATLAB Examples directory.

```
datasetZipFolder = matlab.internal.examples.downloadSupportFile("SPT","data/uwb-gestures.zip");
datasetFolder = erase(datasetZipFolder,".zip");
if ~exist(datasetFolder,"dir")
    downloadLocation = fileparts(datasetZipFolder);
    unzip(datasetZipFolder,downloadLocation);
end
```

You can also choose to download a separate file which includes a pre-trained network, `misoNet`, stored in a MAT-file named `pretrainedNetwork.mat`. It is available at the following location:

https://ssd.mathworks.com/supportfiles/SPT/data/uwb-gestures-network.zip

You can skip the training steps and use the pre-trained network for classification by setting `doTraining` to `false`. If `doTraining` is set to `false`, the pre-trained network will be downloaded later in this example. If you want to train the network as the example runs, make sure to set `doTraining` to `true`.

```
doTraining = true;
```

Explore the Data

Create a signal datastore to access the data in the files. Specify the signal variable names you want to read from each file using the `SignalVariableNames` parameter. This example assumes the dataset has been stored in your MATLAB Examples directory under the *uwb-gestures* folder. If this is not the case, change the path to the data in the `datasetFolder` variable.

```
sds = signalDatastore(datasetFolder,...
                      "IncludeSubfolders",true,...
                      "SignalVariableNames",["Left","Top","Right"],...
                      "FileExtensions",".mat",...
                      "ReadOutputOrientation","row")
```

```
sds =
  signalDatastore with properties:

                       Files:{
                              ' .../R2021b/supportfiles/SPT/data/uwb-gestures/HV_01/HV_01_G10_RawD
                              ' .../R2021b/supportfiles/SPT/data/uwb-gestures/HV_01/HV_01_G11_RawD
                              ' .../R2021b/supportfiles/SPT/data/uwb-gestures/HV_01/HV_01_G12_RawD
                               ... and 93 more
                             }
                     Folders: {'/home/ejoshi/Documents/MATLAB/Examples/R2021b/supportfiles/SPT/da
    AlternateFileSystemRoots: [0×0 string]
                    ReadSize: 1
         SignalVariableNames: ["Left"    "Top"    "Right"]
       ReadOutputOrientation: "row"
```

The datastore returns a three-element cell array containing the radar signal matrices for the left, top, and right radars, in that order.

```
preview(sds)
```

```
ans=1×3 cell array
    {9000×189 double}    {9000×189 double}    {9000×189 double}
```

The rows and columns in each radar signal matrix represent the duration of the hand gesture (slow-time) and the distance of the hand from the radar (fast-time), respectively. During data acquisition,

examiners recorded a subject repeating a particular hand gesture for 450 seconds, corresponding to 9000 (slow-time) rows. There is 1 complete gesture motion in 90 slow-time frames. As such, each radar signal matrix contains 100 complete hand gesture motion samples.The range of each UWB radar is 1.2 meters, corresponding to 189 fast-time bins.

```
slowTimeFrames = 90;
recordedTimePerSample = 4.5;
radarRange = 1.2;
```

To visualize a hand gesture motion, specify a UWB radar location, gesture, and gesture sample (between 1 and 100).

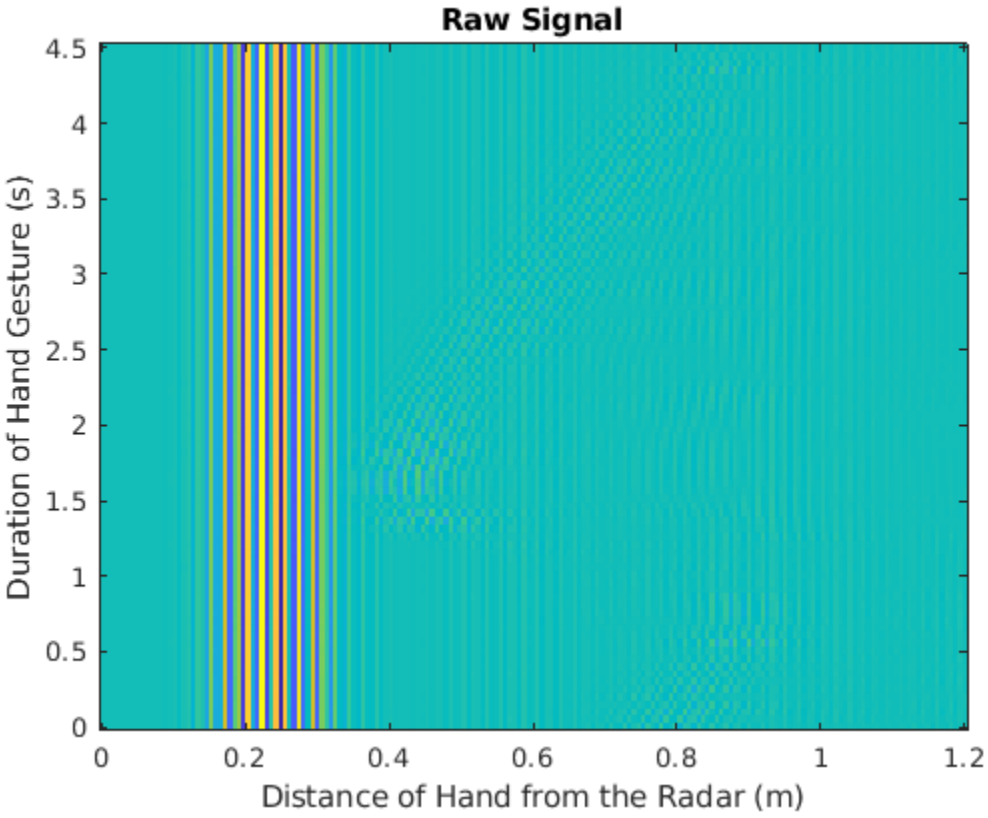
```
radarToPlot = left;
gestureToPlot = left-right swipe;
gestureSample = 50;
Visualize my choices
```

Obtain the radar signal matrix for the chosen hand gesture and radar location.

```
sdssubset = subset(sds,contains(sds.Files,gestureToPlot));
radarDataMatrix = read(sdssubset);
radarDataMatrix = radarDataMatrix{radarToPlot};
```

Use `normalize` to transform the gesture signal data to range between 0 and 1, and use `imagesc` to visualize the hand gesture motion sample.

```
normalizedRadarData = normalize(radarDataMatrix,2,"range",[0 1]);
imagesc([0 radarRange],...
        [0 recordedTimePerSample],...
        normalizedRadarData(slowTimeFrames*(gestureSample-1)+1:slowTimeFrames*gestureSample,:),.
        [0 1]);
set(gca,"YDir","normal")
title("Raw Signal")
xlabel("Distance of Hand from the Radar (m)")
ylabel("Duration of Hand Gesture (s)")
```



As you can see, it is difficult to discern a motion pattern.

The raw signal contains environmental reflections from body parts or other static objects present within the radar"s range. These unwanted reflections are known as "clutter" and can be removed using a pulse canceller that performs an exponential moving average. The transfer function for this operation is

$$H(z) = \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

such that α is a value $0 \le \alpha \le 1$ that controls the amount of averaging [2 on page 12-0]. Use `filter` with the numerator coefficients and denominator coefficients set as [1 -1] and [1 -0.9], respectively, to remove clutter from the raw signal.

```
clutterRemovedSignal = filter([1 -1],[1 -0.9],radarDataMatrix,[],1);
```

Visualize the clutter-removed signal to see the difference.

```
normalizedClutterRemovedSignal = normalize(clutterRemovedSignal,2,"range",[0 1]);
imagesc([0 radarRange],...
        [0 recordedTimePerSample],...
        normalizedClutterRemovedSignal(slowTimeFrames*(gestureSample-1)+1:slowTimeFrames*gestureS
        [0 1]);
set(gca,"YDir","normal")
title("Clutter-Removed Signal")
xlabel("Distance of Hand from the Radar (m)")
ylabel("Duration of Hand Gesture (s)")
```

Note that the motion pattern is much more visible now. For example, if you choose to visualize a *left-right swipe* from the perspective of the left radar, you will see that the distance of the hand from the radar increases over the duration of the hand gesture.

Prepare Data for Training

The MAT-file names contain gesture codes (G1, G2,..., G12) corresponding to labels for each radar signal matrix. Convert these codes to labels within the gesture vocabulary, using a categorical array.

```
[~,filenames] = fileparts(sds.Files);
gestureLabels = extract(filenames,"G"+digitsPattern);
gestureCodes = ["G1","G2","G3","G4",...
                "G5","G6","G7","G8",...
                "G9","G10","G11","G12"];
gestureVocabulary = ["L-R swipe",       "R-L swipe",       "U-D swipe",       "D-U swipe",...
                     "Diag-LR-UD swipe","Diag-LR-DU swipe","Diag-RL-UD swipe","Diag-RL-DU swipe",
                     "clockwise",       "counterclockwise","inward push",     "empty"];
gestureLabels = categorical(gestureLabels,gestureCodes,gestureVocabulary);
```

Collect the labels in an array datastore.

```
labelDs = arrayDatastore(gestureLabels,"OutputType","cell");
```

Combine the signal datastore and array datastore to obtain a single datastore that contains the signal data from each radar and a categorical label. Shuffle the resulting datastore to randomize the order in which it stores the MAT-files.

```
allDataDs = combine(sds,labelDs);
allDataDs = shuffle(allDataDs);
preview(allDataDs)
```

```
ans=1×4 cell array
    {9000×189 double}    {9000×189 double}    {9000×189 double}    {[Diag-LR-UD swipe]}
```

The transform function allows the helper function, `processData`, to be applied to data as it is read by a datastore. `processData` performs the normalization and filtering that was described in the above section to standardize data and remove clutter. In addition, it divides the radar signal matrix into separate hand gesture motion samples.

```
allDataDs = transform(allDataDs,@processData);
preview(allDataDs)
```

```
ans=8×4 cell array
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
    {90×189 double}    {90×189 double}    {90×189 double}    {[Diag-LR-UD swipe]}
```

Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. Since the data is being read from individual samples, the data will need to be read into memory, before being re-shuffled and inserted into another datastore for training.

Because the entire training dataset fits in memory, it is possible to transform the data in parallel, if Parallel Computing Toolbox is available, and then gather it into the workspace. Use `readall` with the `UseParallel` flag set to true to utilize a parallel pool to read all of the signal data and labels into the workspace. If the data fits into the memory of your computer, importing the data into the workspace enables faster training because the data is read and transformed only once. Note that if the data does not fit in memory, you must to pass the datastore into the training function, and the transformations are performed at every training epoch.

```
allData = readall(allDataDs,"UseParallel",true);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

The labels are returned as the last column in `allData`. Use `countlabels` to obtain proportions of label values in the dataset. Note that the gestures are balanced and well-represented across the dataset.

```
countlabels(allData(:,4))
```

```
ans=12×3 table
         Label           Count    Percent
    ________________     _____    _______

    D-U swipe             793     8.2664 
    Diag-LR-DU swipe      800     8.3394 
```

```
    Diag-LR-UD swipe     800     8.3394
    Diag-RL-DU swipe     800     8.3394
    Diag-RL-UD swipe     800     8.3394
    L-R swipe            800     8.3394
    R-L swipe            800     8.3394
    U-D swipe            800     8.3394
    clockwise            800     8.3394
    counterclockwise     800     8.3394
    empty                800     8.3394
    inward push          800     8.3394
```

Divide the data randomly into training and validation sets, while making sure to leave testing data for later. In this example, training, validation, and testing splits will be 70%, 15%, and 15%, respectively. Use `splitlabels` to split the data into training, validation, and testing sets that maintain the same label proportions as the original dataset. Specify the `randomized` option to shuffle the data randomly across the three sets.

```
idxs = splitlabels(allData(:,4),[0.7 0.15],"randomized");
trainIdx = idxs{1}; valIdx = idxs{2}; testIdx = idxs{3};
```

Avoid selecting samples from the same trial by randomizing the indices once more. Store the in-memory training and validation data in array datastores so that they can be used to train a multi-input network.

```
trainData = allData(trainIdx(randperm(length(trainIdx))),:);
valData = allData(valIdx(randperm(length(valIdx))),:);
trainDataDs = arrayDatastore(trainData,"OutputType","same");
valDataDs = arrayDatastore(valData,"OutputType","same");
```

Prepare Network for Training

Define the network architecture before training. Since each hand gesture motion is captured by 3 independent UWB impulse radars, we will use a CNN architecture that accepts 3 signals as separate inputs. The results achieved after training this proposed multiple-input, single-output CNN are considerably better than those achieved with an alternative single-input, single-output CNN whose input is a 90 x 189 x 3 stack of radar data matrices.

`repeatBranch` contains operations that will be performed separately on the three radar data signal matrices. The CNN model needs to combine the extracted feature information from each signal to make a final gesture label prediction. `mainBranch` contains operations that will concatenate the 3 `repeatBranch` outputs and estimate labels. Specify an `imageInputLayer` of size 90 x 189 to accept the hand gesture motion samples. Specify an `additionLayer` with number of inputs set to 3, to collect the outputs of the 3 branches and pass them into the classification section of the model. Specify a `fullyConnectedLayer` with an output size of 12, one for each of the hand gestures. Add a `softmaxLayer` and a `classificationLayer` to output the estimated labels.

```
repeatBranch = [
    imageInputLayer([90 189 1],"Normalization", "none")

    convolution2dLayer(3,8,"Padding",1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,16,"Padding",1)
    batchNormalizationLayer
```

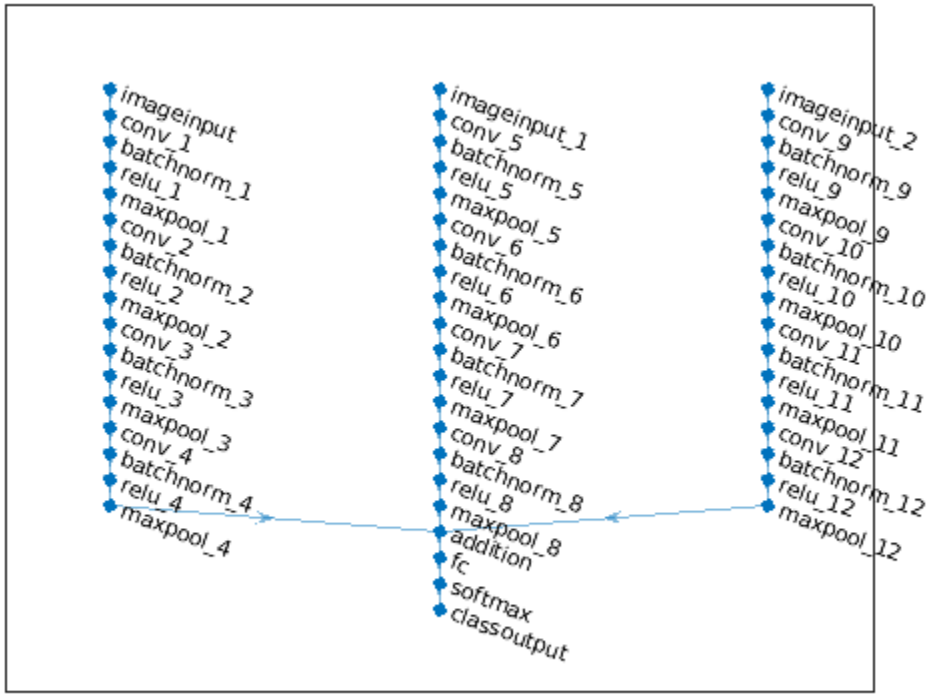
```
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,32,"Padding",1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)

    convolution2dLayer(3,64,"Padding",1)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(2,"Stride",2)];

mainBranch = [
    additionLayer(3)
    fullyConnectedLayer(12)
    softmaxLayer
    classificationLayer];
```

Define a `layerGraph` to which `repeatBranch` is added 3 times and `mainBranch` is added once. Connect the outputs of the final `maxPooling2dLayer` in each `repeatBranch` to the inputs of `additionLayer`.

```
misoCNN = layerGraph();
misoCNN = addLayers(misoCNN, repeatBranch);
misoCNN = addLayers(misoCNN, repeatBranch);
misoCNN = addLayers(misoCNN, repeatBranch);
misoCNN = addLayers(misoCNN, mainBranch);
misoCNN = connectLayers(misoCNN, "maxpool_4", "addition/in1");
misoCNN = connectLayers(misoCNN, "maxpool_8", "addition/in2");
misoCNN = connectLayers(misoCNN, "maxpool_12", "addition/in3");
```

Visualize the multiple-input, single-output CNN.

```
plot(misoCNN);
```



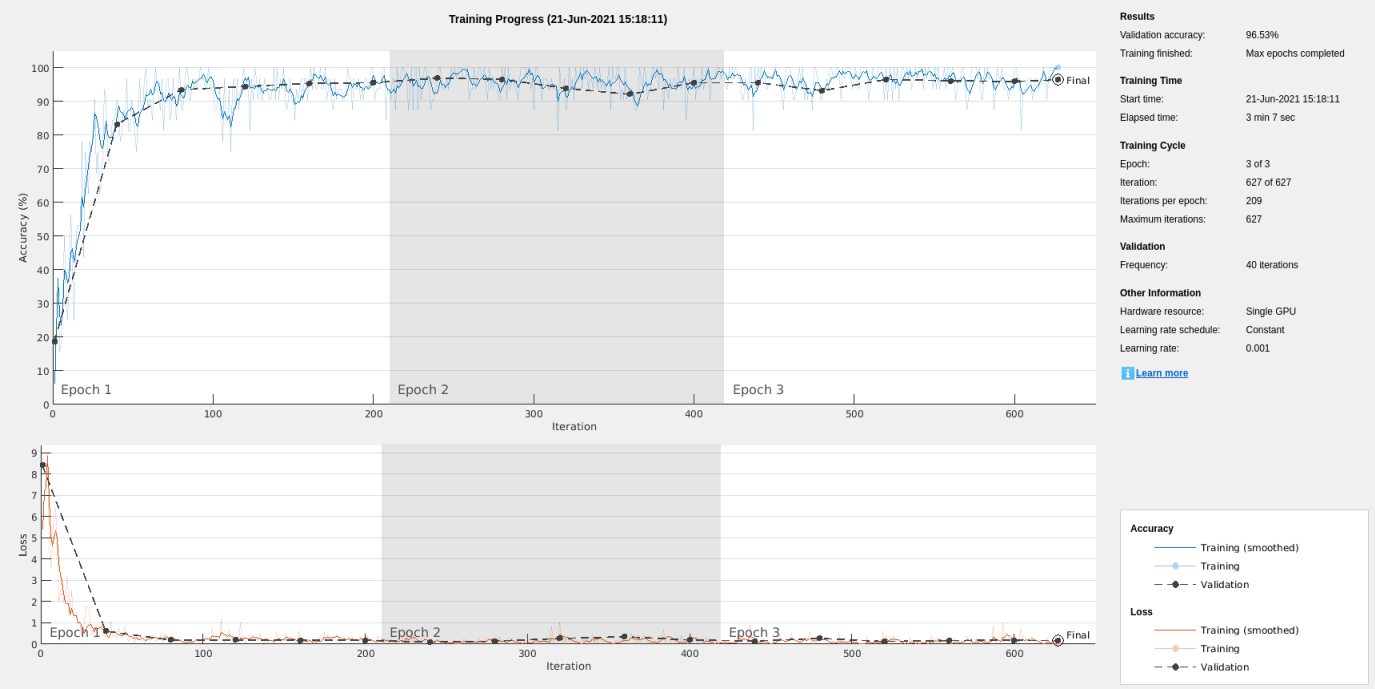
Choose options for the training process that ensure good network performance. Make sure to specify `valDataDs` as `ValidationData`, so that it is used for validation during training. Refer to the `trainingOptions` (Deep Learning Toolbox) documentation for a description of each parameter.

```
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-3, ...
    "MaxEpochs",3,...
    "MiniBatchSize",32, ...
    "ValidationData",valDataDs,...
    "ValidationFrequency",40,...
    "Verbose",false,...
    "Plots","training-progress");
```

Train Network

Use the `trainNetwork` command to train the CNN.

```
if doTraining == true
    misoNet = trainNetwork(trainDataDs,misoCNN,options);
else
    pretrainedNetworkZipFolder = matlab.internal.examples.downloadSupportFile("SPT","data/uwb-ges
    pretrainedNetworkFolder = erase(pretrainedNetworkZipFolder,".zip");
    if ~exist(pretrainedNetworkFolder,"dir")
        downloadLocation = fileparts(pretrainedNetworkZipFolder);
        unzip(pretrainedNetworkZipFolder,downloadLocation);
    end
    load(fullfile(pretrainedNetworkFolder,"pretrainedNetwork.mat"),"misoNet");
end
```



Classify Testing Data

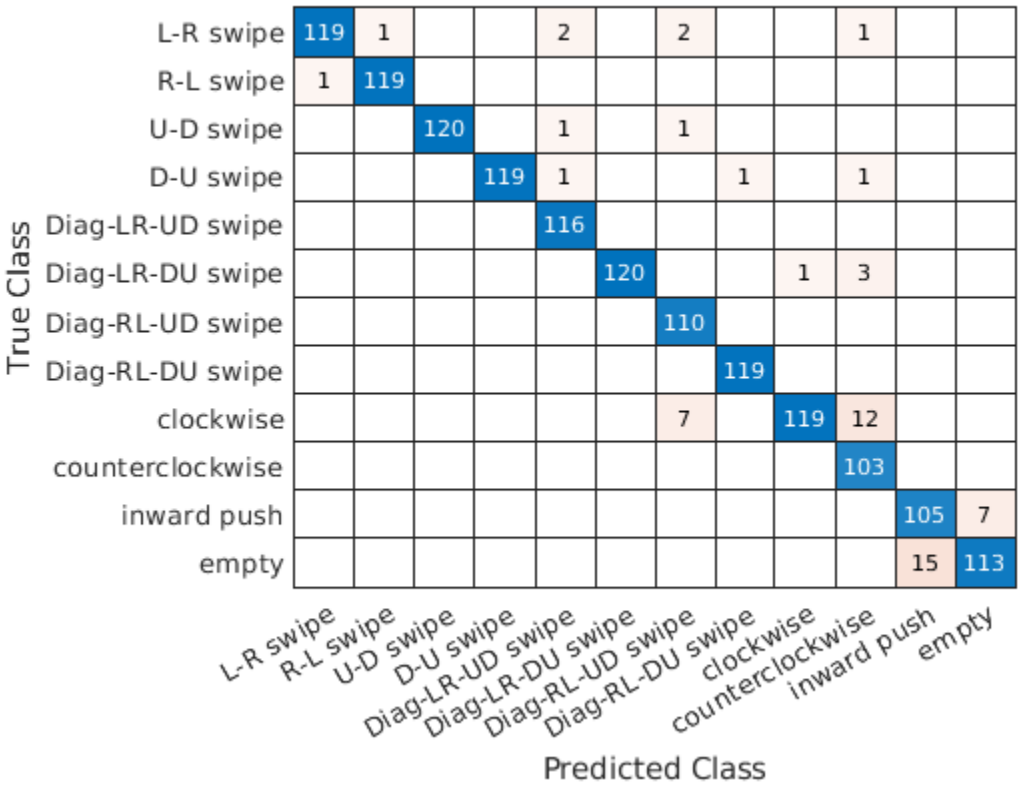
Classify the testing data using the trained CNN and the `classify` command.

```
testData = allData(testIdx,:);
testData = {cat(4,testData{:,1}),cat(4,testData{:,2}),cat(4,testData{:,3}),cat(1,testData{:,4})};
actualLabels = testData{4};
predictedLabels = classify(misoNet,testData{1},testData{2},testData{3});
accuracy = mean(predictedLabels==actualLabels);
fprintf("Accuracy on the test set is %0.2f%%",100*accuracy)
```

```
Accuracy on the test set is 96.04%
```

Visualize classification performance using a confusion matrix.

```
confusionchart(predictedLabels,actualLabels);
```



The largest confusion is between *counterclockwise* and *clockwise* movements and *inward push* and *empty* movements.

Explore Network Predictions

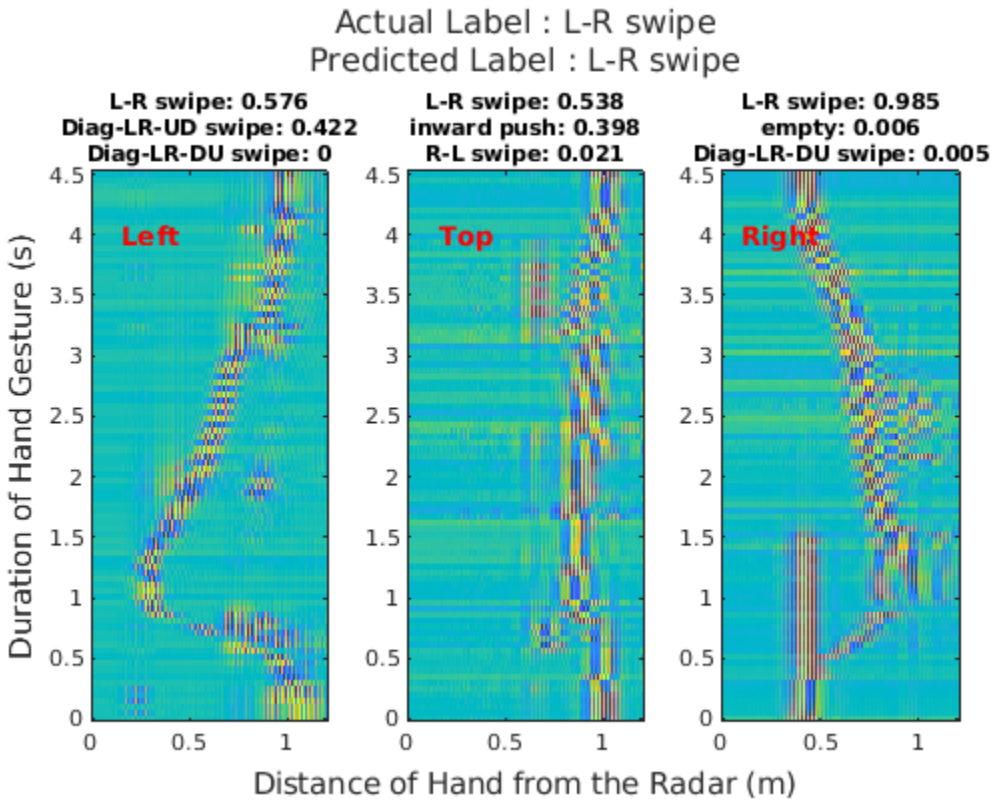
You can obtain the scores from the final max-pooling layers in each input branch to get a better understanding of how data from each radar contributed to final network confidence. The helper function, `getActivationData`, returns softmax-normalized scores (probabilities for class membership) and indices corresponding to the 3 highest scores.

```
gestureToPlot = left-right swipe;
gestureToPlotIndices = find(matches(string(actualLabels),gestureToPlot));
gestureSelection = randsample(gestureToPlotIndices,1);
actualLabel = actualLabels(gestureSelection);
predictedLabel = predictedLabels(gestureSelection);
allLabels = categories(actualLabels);

[leftScores, leftClassIds] = getActivationData(misoNet,testData,gestureSelection,"maxpool_4");
[topScores, topClassIds] = getActivationData(misoNet,testData,gestureSelection,"maxpool_8");
[rightScores, rightClassIds] = getActivationData(misoNet,testData,gestureSelection,"maxpool_12")
```

Use the helper function `plotActivationData` to visualize the data from each radar and overlay the labels corresponding to the 3 highest scores after the operations in each input branch are completed.

```
t = tiledlayout(1,3);
plotActivationData(testData, allLabels, leftScores, leftClassIds,...
```

```
    gestureSelection, [0 radarRange],[0 recordedTimePerSample], "Left");
plotActivationData(testData, allLabels, topScores, topClassIds,...
    gestureSelection, [0 radarRange],[0 recordedTimePerSample], "Top");
plotActivationData(testData, allLabels, rightScores, rightClassIds,...
    gestureSelection, [0 radarRange],[0 recordedTimePerSample], "Right");
title(t,["Actual Label : "+string(actualLabel);"Predicted Label : "+string(predictedLabel)],"Fon
xlabel(t,"Distance of Hand from the Radar (m)","FontSize",11)
ylabel(t,"Duration of Hand Gesture (s)","FontSize",11)
```



Conclusion

In this example, you learned how to use a multiple-input CNN to extract information from 3 independent radar data matrices and classify hand gesture movements. The multiple-input architecture allowed us to take advantage of data generated by multiple sensors recording the same event.

Helper Functions

```
function dataOut = processData(dataIn)
    label = dataIn(end);
    dataIn(end) = [];
    dataOut = cellfun(@(x) filter([1 -1],[1 -0.9],x,[],1),dataIn,"UniformOutput",false);
    dataOut = cellfun(@(x) normalize(x,2,"range",[0 1]),dataOut,"UniformOutput",false);
    dataOut = cat(1,dataOut{:});
    dataOut = mat2cell(dataOut,90*ones(1,size(dataOut,1)/90));
    dataOut = reshape(dataOut,[],3);
    dataOut(:,4) = label;
end
```

```
function [scores, classIds] = getActivationData(net, testData, index, layer)
    activation = activations(net,testData{1}(:,:,index),testData{2}(:,:,index),testData{3}(:,:,in
    fcWeights = net.Layers(end-2).Weights;
    fcBias = net.Layers(end-2).Bias;
    scores = fcWeights*activation + fcBias;
    scores = exp(scores)/sum(exp(scores));
    [~,classIds] = maxk(scores,3);
end

function plotActivationData(testData, labels, scores, ids, sampleIdx, xlimits, ylimits, plotTitle
    if plotTitle == "Left"
        gesture = 1;
    elseif plotTitle == "Top"
        gesture = 2;
    elseif plotTitle == "Right"
        gesture = 3;
    end
    nexttile;
    imagesc(xlimits,ylimits,testData{gesture}(:,:,sampleIdx),[0 1])
    text(0.3,4,plotTitle,"Color","red","FontWeight","bold","HorizontalAlignment","center")
    set(gca,"YDir","normal")
    title(string(labels(ids)) + ": " + string(round(scores(ids),3)),"FontSize",8);
end
```

References

[1] Ahmed, S., Wang, D., Park, J. et al. UWB-gestures, a public dataset of dynamic hand gestures acquired using impulse radar sensors. Sci Data 8, 102 (2021). https://doi.org/10.1038/s41597-021-00876-0

[2] Lazaro A, Girbau D, Villarino R. Techniques for clutter suppression in the presence of body movements during the detection of respiratory activity through UWB radars. Sensors (Basel, Switzerland). 2014 Feb;14(2):2595-2618. DOI: 10.3390/s140202595.

See Also

`arrayDatastore` | `signalDatastore` | `splitlabels` | `trainNetwork`

Waveform Segmentation Using Deep Learning

This example shows how to segment human electrocardiogram (ECG) signals using recurrent deep learning networks and time-frequency analysis.

Introduction

The electrical activity in the human heart can be measured as a sequence of amplitudes away from a baseline signal. For a single normal heartbeat cycle, the ECG signal can be divided into the following beat morphologies [1 on page 12-0]:

- P wave — A small deflection before the QRS complex representing atrial depolarization
- QRS complex — Largest-amplitude portion of the heartbeat
- T wave — A small deflection after the QRS complex representing ventricular repolarization

The segmentation of these regions of ECG waveforms can provide the basis for measurements useful for assessing the overall health of the human heart and the presence of abnormalities [2 on page 12-0]. Manually annotating each region of the ECG signal can be a tedious and time-consuming task. Signal processing and deep learning methods potentially can help streamline and automate region-of-interest annotation.

This example uses ECG signals from the publicly available QT Database [3 on page 12-0] [4 on page 12-0]. The data consists of roughly 15 minutes of ECG recordings, with a sample rate of 250 Hz, measured from a total of 105 patients. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, resulting in a two-channel signal. The database provides signal region labels generated by an automated expert system [2 on page 12-0]. This example aims to use a deep learning solution to provide a label for every ECG signal sample according to the region where the sample is located. This process of labeling regions of interest across a signal is often referred to as *waveform segmentation*.

To train a deep neural network to classify signal regions, you can use a Long Short-Term Memory (LSTM) network. This example shows how signal preprocessing techniques and time-frequency analysis can be used to improve LSTM segmentation performance. In particular, the example uses the Fourier synchrosqueezed transform to represent the nonstationary behavior of the ECG signal.

Download and Prepare the Data

Each channel of the 105 two-channel ECG signals was labeled independently by the automated expert system and is treated independently, for a total of 210 ECG signals that were stored together with the region labels in 210 MAT-files. The files are available at the following location: https://www.mathworks.com/supportfiles/SPT/data/QTDatabaseECGData.zip.

Download the data files into your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. If you want to place the data files in a folder different from `tempdir`, change the directory name in the subsequent instructions.

```
% Download the data
dataURL = 'https://www.mathworks.com/supportfiles/SPT/data/QTDatabaseECGData1.zip';
datasetFolder = fullfile(tempdir,'QTDataset');
zipFile = fullfile(tempdir,'QTDatabaseECGData.zip');
if ~exist(datasetFolder,'dir')
     websave(zipFile,dataURL);
     unzip(zipFile,tempdir);
end
```

The `unzip` operation creates the `QTDatabaseECGData` folder in your temporary directory with 210 MAT-files in it. Each file contains an ECG signal in variable `ecgSignal` and a table of region labels in variable `signalRegionLabels`. Each file also contains the sample rate of the signal in variable `Fs`. In this example all signals have a sample rate of 250 Hz.

Create a signal datastore to access the data in the files. This example assumes the dataset has been stored in your temporary directory under the `QTDatabaseECGData` folder. If this is not the case, change the path to the data in the code below. Specify the signal variable names you want to read from each file using the `SignalVariableNames` parameter.

```
sds = signalDatastore(datasetFolder,'SignalVariableNames',["ecgSignal","signalRegionLabels"])
```

```
sds =
  signalDatastore with properties:

                      Files:{
                             '/tmp/QTDataset/ecg1.mat';
                             '/tmp/QTDataset/ecg10.mat';
                             '/tmp/QTDataset/ecg100.mat'
                              ... and 207 more
                             }
    AlternateFileSystemRoots: [0×0 string]
                    ReadSize: 1
         SignalVariableNames: ["ecgSignal"    "signalRegionLabels"]
```

The datastore returns a two-element cell array with an ECG signal and a table of region labels each time you call the `read` function. Use the `preview` function of the datastore to see that the content of the first file is a 225,000 samples long ECG signal and a table containing 3385 region labels.

```
data = preview(sds)
```

```
data=2×1 cell array
    {225000×1 double}
    {  3385×2 table }
```

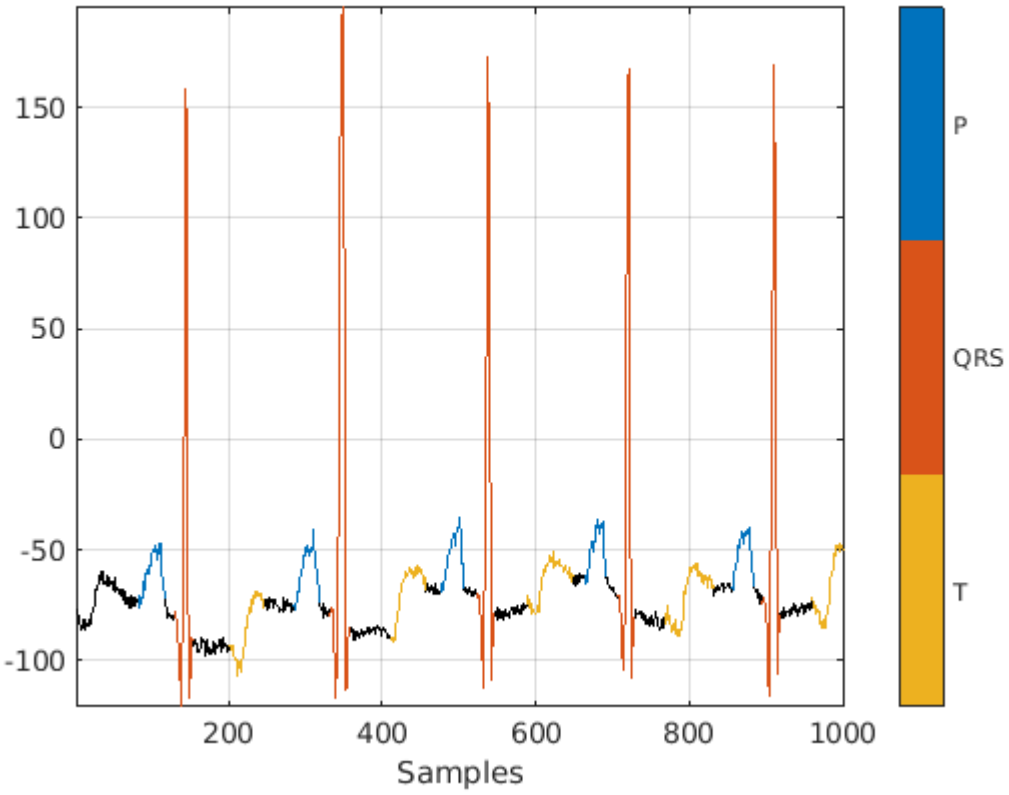
Look at the first few rows of the region labels table and observe that each row contains the region limit indices and the region class value (P, T, or QRS).

```
head(data{2})
```

```
ans=8×2 table
    ROILimits     Value
    _________    _____

     83    117     P
    130    153     QRS
    201    246     T
    285    319     P
    332    357     QRS
    412    457     T
    477    507     P
    524    547     QRS
```

Visualize the labels for the first 1000 samples using a `signalMask` object.

```
M = signalMask(data{2});
plotsigroi(M,data{1}(1:1000))
```



The usual machine learning classification procedure is the following:

1. Divide the database into training and testing datasets.
2. Train the network using the training dataset.
3. Use the trained network to make predictions on the testing dataset.

The network is trained with 70% of the data and tested with the remaining 30%.

For reproducible results, reset the random number generator. Use the `dividerand` function to get random indices to shuffle the files, and the `subset` function of `signalDatastore` to divide the data into training and testing datastores.

```
rng default
[trainIdx,~,testIdx] = dividerand(numel(sds.Files),0.7,0,0.3);
trainDs = subset(sds,trainIdx);
testDs = subset(sds,testIdx);
```

In this segmentation problem, the input to the LSTM network is an ECG signal and the output is a sequence or mask of labels with the same length as the input signal. The network task is to label each signal sample with the name of the region it belongs to. For this reason, it is necessary to transform the region labels on the dataset to sequences containing one label per signal sample. Use a transformed datastore and the `getmask` helper function to transform the region labels. The `getmask` function adds a label category, `"n/a"`, to label samples that do not belong to any region of interest.

```
type getmask.m
```

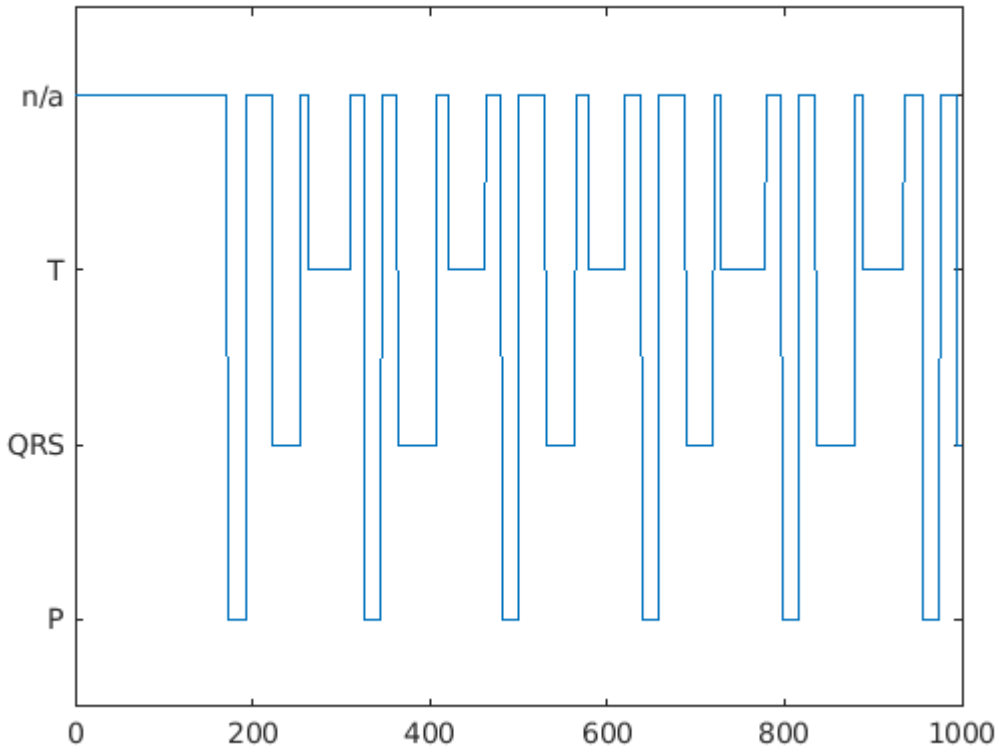
```
function outputCell = getmask(inputCell)
%GETMASK Convert region labels to a mask of labels of size equal to the
%size of the input ECG signal.
%
%   inputCell is a two-element cell array containing an ECG signal vector
%   and a table of region labels.
%
%   outputCell is a two-element cell array containing the ECG signal vector
%   and a categorical label vector mask of the same length as the signal.

% Copyright 2020 The MathWorks, Inc.

sig = inputCell{1};
roiTable = inputCell{2};
L = length(sig);
M = signalMask(roiTable);

% Get categorical mask and give priority to QRS regions when there is overlap
mask = catmask(M,L,'OverlapAction','prioritizeByList','PriorityList',[2 1 3]);

% Set missing values to "n/a"
mask(ismissing(mask)) = "n/a";

outputCell = {sig,mask};
end
```

Preview the transformed datastore to observe that it returns a signal vector and a label vector of equal lengths. Plot the first 1000 element of the categorical mask vector.

```
trainDs = transform(trainDs, @getmask);
testDs = transform(testDs, @getmask);

transformedData = preview(trainDs)
```

```
transformedData=1×2 cell array
    {224993×1 double}    {224993×1 categorical}
```

```
plot(transformedData{2}(1:1000))
```



Passing very long input signals into the LSTM network can result in estimation performance degradation and excessive memory usage. To avoid these effects, break the ECG signals and their corresponding label masks using a transformed datastore and the `resizeData` helper function. The helper function creates as many 5000-sample segments as possible and discards the remaining samples. A preview of the output of the transformed datastore shows that the first ECG signal and its label mask are broken into 5000-sample segments. Note that preview of the transformed datastore only shows the first 8 elements of the otherwise `floor(224993/5000)` = 44 element cell array that would result if we called the datastore `read` function.

```
trainDs = transform(trainDs,@resizeData);
testDs = transform(testDs,@resizeData);
preview(trainDs)
```

```
ans=8×2 cell array
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
    {1×5000 double}    {1×5000 categorical}
```

Choose to Train Networks or Download Pre-Trained Networks

The next sections of this example compare three different approaches to train LSTM networks. Due to the large size of the dataset, the training process of each network may take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB automatically uses the GPU for faster training. Otherwise, it uses the CPU.

You can skip the training steps and download the pre-trained networks using the selector below. If you want to train the networks as the example runs, select 'Train Networks'. If you want to skip the training steps, select 'Download Networks' and a file containing all three pre-trained networks - `rawNet`, `filteredNet`, and `fsstNet-` will be downloaded into your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. If you want to place the downloaded file in a folder different from `tempdir`, change the directory name in the subsequent instructions.

```
actionFlag = Train networks;
if actionFlag == "Download networks"
    % Download the pre-trained networks
    dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/QTDatabaseECGSegmentationNetworks
    modelsFolder = fullfile(tempdir,'QTDatabaseECGSegmentationNetworks');
    modelsFile = fullfile(modelsFolder,'trainedNetworks.mat');
    zipFile = fullfile(tempdir,'QTDatabaseECGSegmentationNetworks.zip');
    if ~exist(modelsFolder,'dir')
        websave(zipFile,dataURL);
        unzip(zipFile,fullfile(tempdir,'QTDatabaseECGSegmentationNetworks'));
    end
    load(modelsFile)
end
```

Results between the downloaded networks and newly trained networks may vary slightly since the networks are trained using random initial weights.

Input Raw ECG Signals Directly into the LSTM Network

First, train an LSTM network using the raw ECG signals from the training dataset.

Define the network architecture before training. Specify a `sequenceInputLayer` of size 1 to accept one-dimensional time series. Specify an LSTM layer with the `'sequence'` output mode to provide classification for each sample in the signal. Use 200 hidden nodes for optimal performance. Specify a `fullyConnectedLayer` with an output size of 4, one for each of the waveform classes. Add a `softmaxLayer` and a `classificationLayer` to output the estimated labels.

```
layers = [ ...
    sequenceInputLayer(1)
    lstmLayer(200,'OutputMode','sequence')
    fullyConnectedLayer(4)
    softmaxLayer
    classificationLayer];
```

Choose options for the training process that ensure good network performance. Refer to the `trainingOptions` documentation for a description of each parameter.

```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
    'MiniBatchSize',50, ...
    'InitialLearnRate',0.01, ...
    'LearnRateDropPeriod',3, ...
```

```
    'LearnRateSchedule','piecewise', ...
    'GradientThreshold',1, ...
    'Plots','training-progress',...
    'shuffle','every-epoch',...
    'Verbose',0,...
    'DispatchInBackground',true);
```

Because the entire training dataset fits in memory, it is possible to use the `tall` function of the datastore to transform the data in parallel, if Parallel Computing Toolbox™ is available, and then gather it into the workspace. Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. If the data fits into the memory of your computer, importing the data into the workspace enables faster training because the data is read and transformed only once. Note that if the data does not fit in memory, you must to pass the datastore into the training function, and the transformations are performed at every training epoch.

Create tall arrays for both the training and test sets. Depending on your system, the number of workers in the parallel pool that MATLAB creates may be different.

```
tallTrainSet = tall(trainDs);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

```
tallTestSet = tall(testDs);
```

Now call the `gather` function of the tall arrays to compute the transformations over the entire dataset and obtain cell arrays with the training and test signals and labels.

```
 trainData = gather(tallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 11 sec
Evaluation completed in 12 sec
```

```
 trainData(1,:)
```

```
ans=1×2 cell array
    {1×5000 double}    {1×5000 categorical}
```

```
 testData = gather(tallTestSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 2.9 sec
Evaluation completed in 3.1 sec
```

Train Network

Use the `trainNetwork` command to train the LSTM network.

```
if actionFlag == "Train networks"
     rawNet = trainNetwork(trainData(:,1),trainData(:,2),layers,options);
end
```

The training accuracy and loss subplots in the figure track the training progress across all iterations. Using the raw signal data, the network correctly classifies about 77% of the samples as belonging to a P wave, a QRS complex, a T wave, or an unlabeled region `"n/a"`.

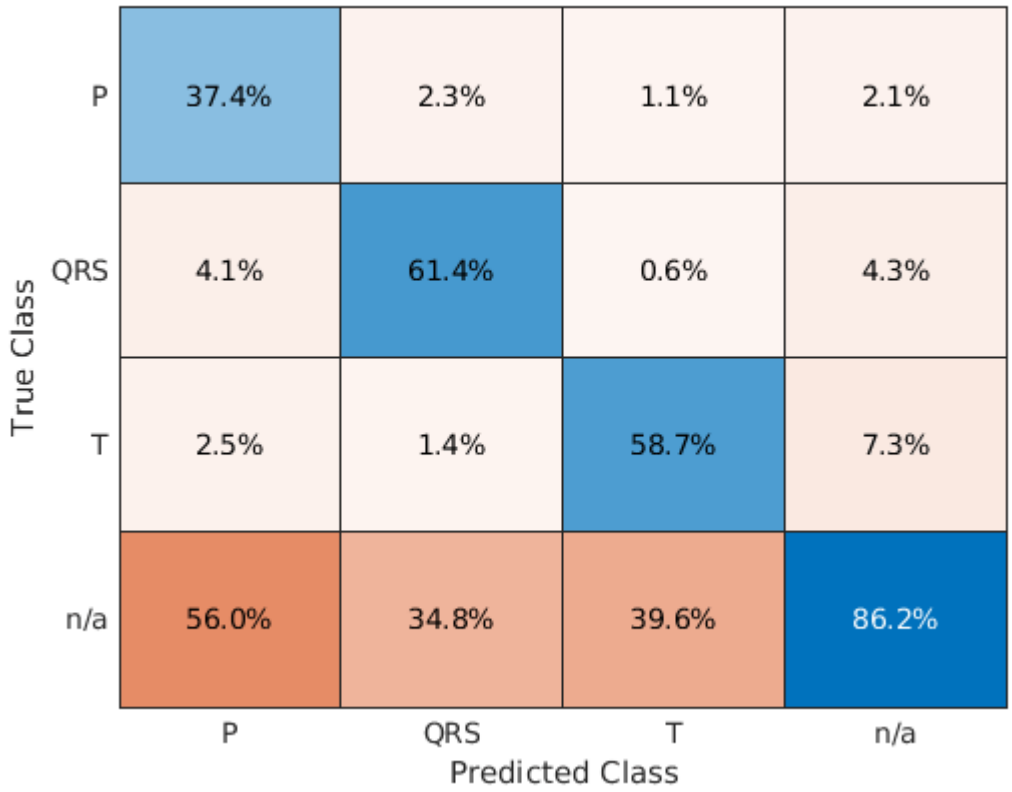
Classify Testing Data

Classify the testing data using the trained LSTM network and the `classify` command. Specify a mini-batch size of 50 to match the training options.

```
predTest = classify(rawNet,testData(:,1),'MiniBatchSize',50);
```

A confusion matrix provides an intuitive and informative means to visualize classification performance. Use the `confusionchart` command to calculate the overall classification accuracy for the testing data predictions. For each input, convert the cell array of categorical labels to a row vector. Specify a column-normalized display to view results as percentages of samples for each class.

```
confusionchart([predTest{:}],[testData{:,2}],'Normalization','column-normalized');
```



Using the raw ECG signal as input to the network, only about 60% of T-wave samples, 40% of P-wave samples, and 60% of QRS-complex samples were correct. To improve performance, apply some knowledge of the ECG signal characteristics prior to input to the deep learning network, for instance the baseline wandering caused by a patient's respiratory motion.

Apply Filtering Methods to Remove Baseline Wander and High-Frequency Noise

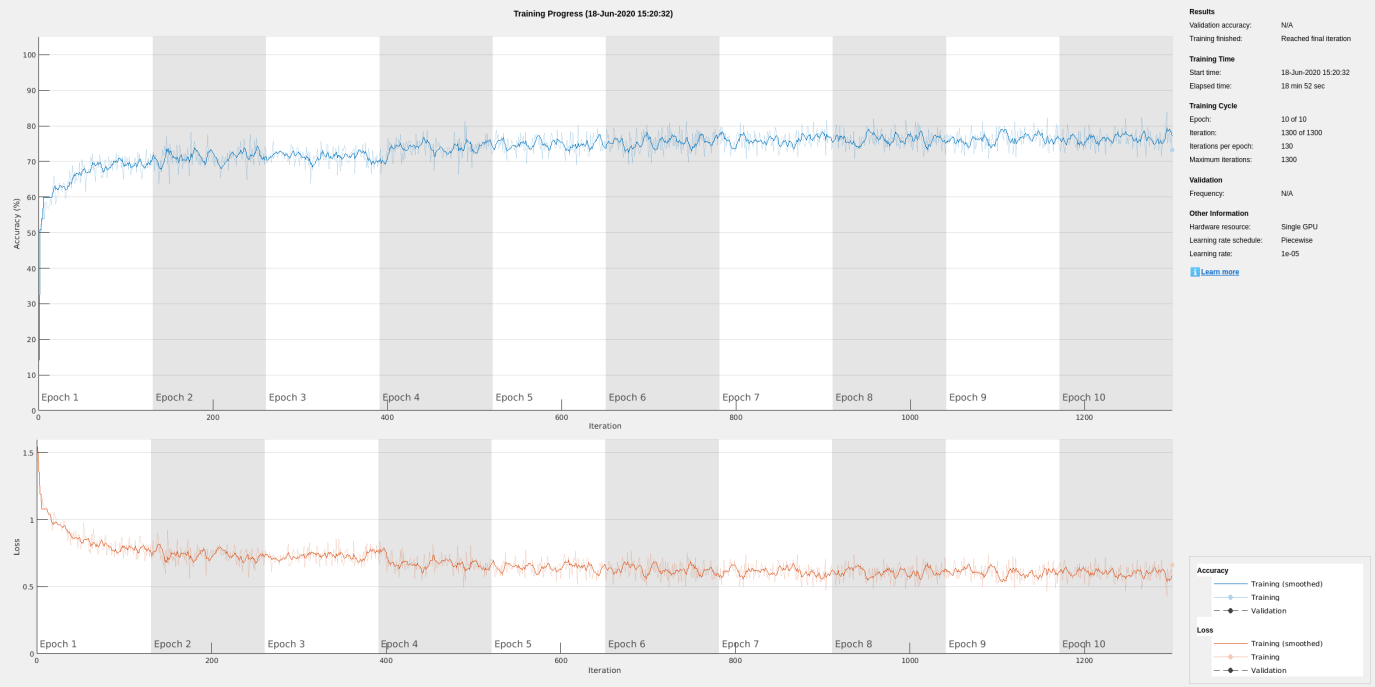
The three beat morphologies occupy different frequency bands. The spectrum of the QRS complex typically has a center frequency around 10–25 Hz, and its components lie below 40 Hz. The P and T waves occur at even lower frequencies: P-wave components are below 20 Hz, and T-wave components are below 10 Hz [5 on page 12-0].

Baseline wander is a low-frequency (< 0.5 Hz) oscillation caused by the patient's breathing motion. This oscillation is independent from the beat morphologies and does not provide meaningful information [6 on page 12-0].

Design a bandpass filter with passband frequency range of [0.5, 40] Hz to remove the wander and any high frequency noise. Removing these components improves the LSTM training because the network does not learn irrelevant features. Use `cellfun` on the tall data cell arrays to filter the dataset in parallel.

```
% Bandpass filter design
hFilt = designfilt('bandpassiir', 'StopbandFrequency1',0.4215,'PassbandFrequency1', 0.5, ...
    'PassbandFrequency2',40,'StopbandFrequency2',53.345,...
    'StopbandAttenuation1',60,'PassbandRipple',0.1,'StopbandAttenuation2',60,...
    'SampleRate',250,'DesignMethod','ellip');
```

```
% Create tall arrays from the transformed datastores and filter the signals
tallTrainSet = tall(trainDs);
tallTestSet = tall(testDs);

filteredTrainSignals = gather(cellfun(@(x)filter(hFilt,x),tallTrainSet(:,1),'UniformOutput',false

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```



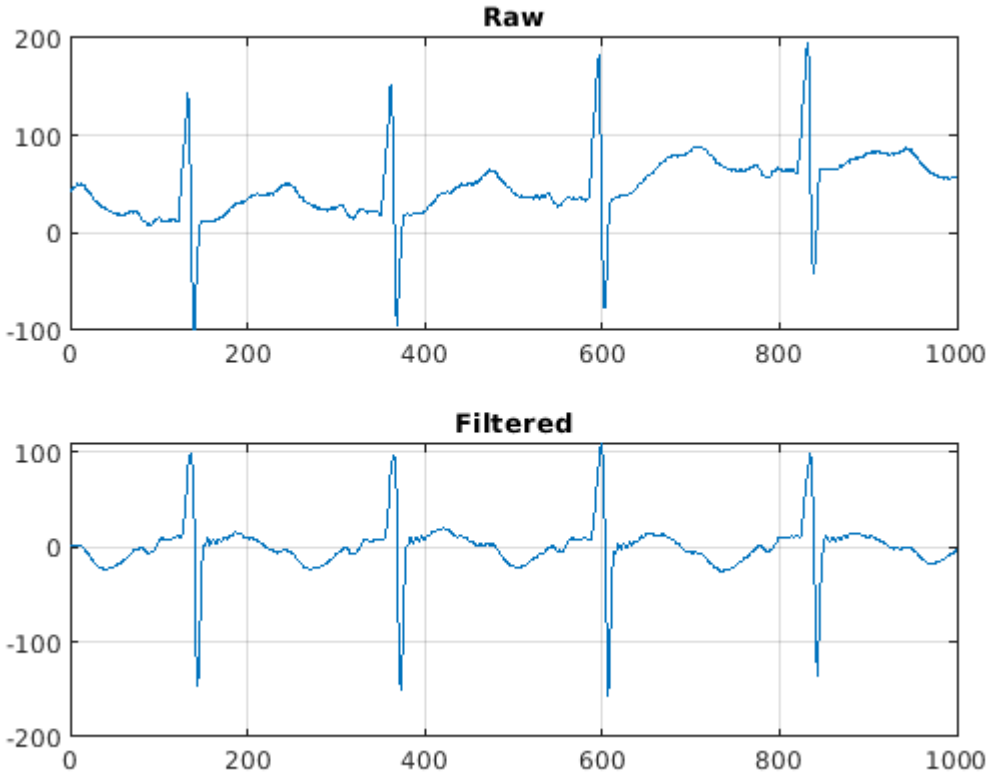

```
- Pass 1 of 1: Completed in 13 sec
Evaluation completed in 14 sec

trainLabels = gather(tallTrainSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 3.6 sec
Evaluation completed in 4 sec

filteredTestSignals = gather(cellfun(@(x)filter(hFilt,x),tallTestSet(:,1),'UniformOutput',false)

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 2.4 sec
Evaluation completed in 2.5 sec

testLabels = gather(tallTestSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1.9 sec
Evaluation completed in 2 sec
```

Plot the raw and filtered signals for a typical case.

```
trainData = gather(tallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4 sec
Evaluation completed in 4.2 sec
```

```
figure
subplot(2,1,1)
plot(trainData{95,1}(2001:3000))
title('Raw')
grid
subplot(2,1,2)
plot(filteredTrainSignals{95}(2001:3000))
title('Filtered')
grid
```

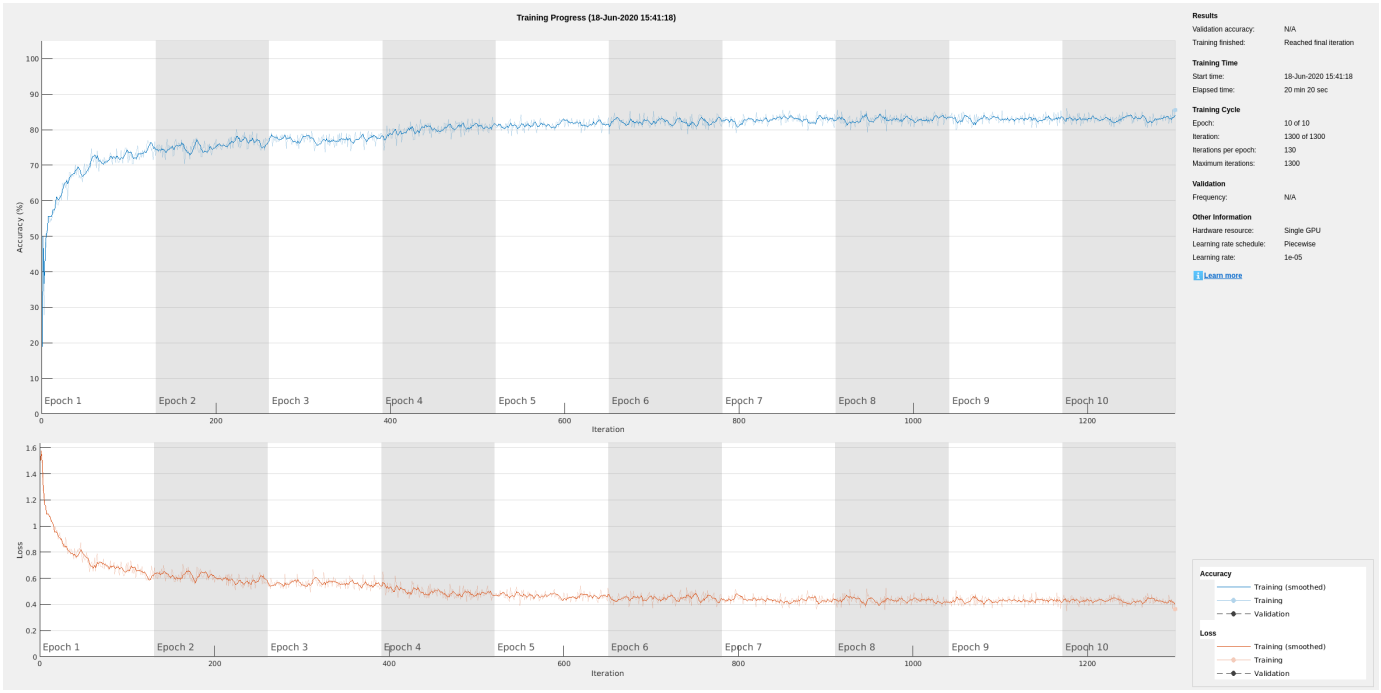


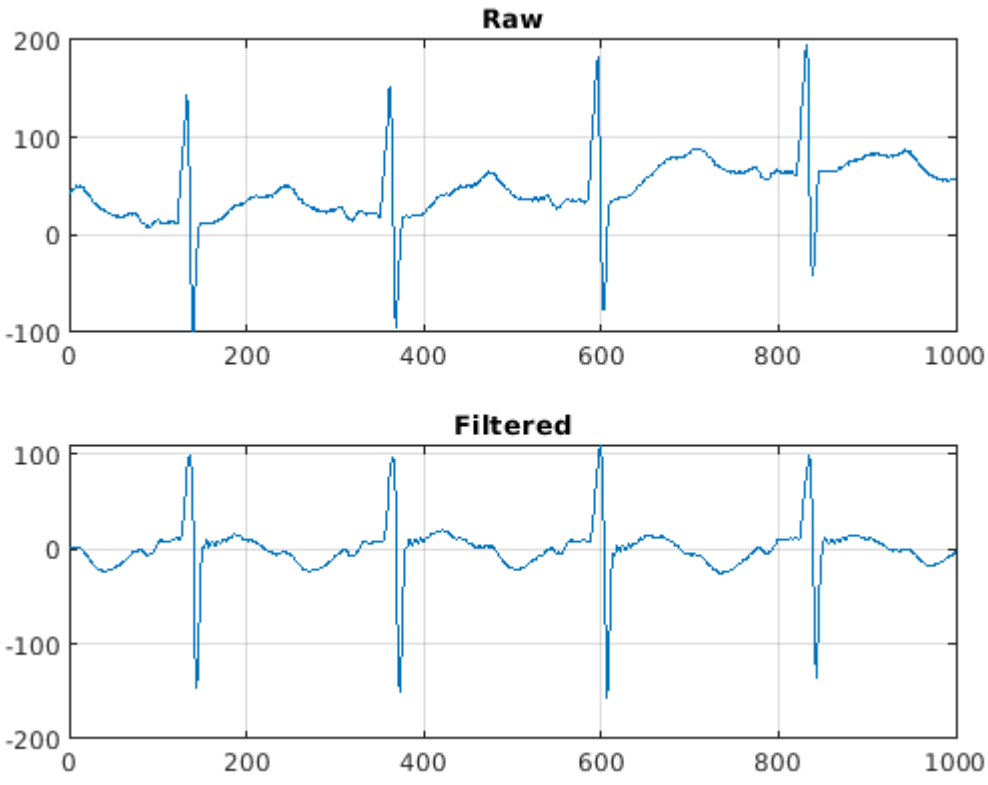
Even though the baseline of the filtered signals may confuse a physician that is used to traditional ECG measurements on medical devices, the network will actually benefit from the wandering removal.

Train Network with Filtered ECG Signals

Train the LSTM network on the filtered ECG signals using the same network architecture as before.

```
if actionFlag == "Train networks"
    filteredNet = trainNetwork(filteredTrainSignals,trainLabels,layers,options);
end
```





Preprocessing the signals improves the training accuracy to better than 80%.

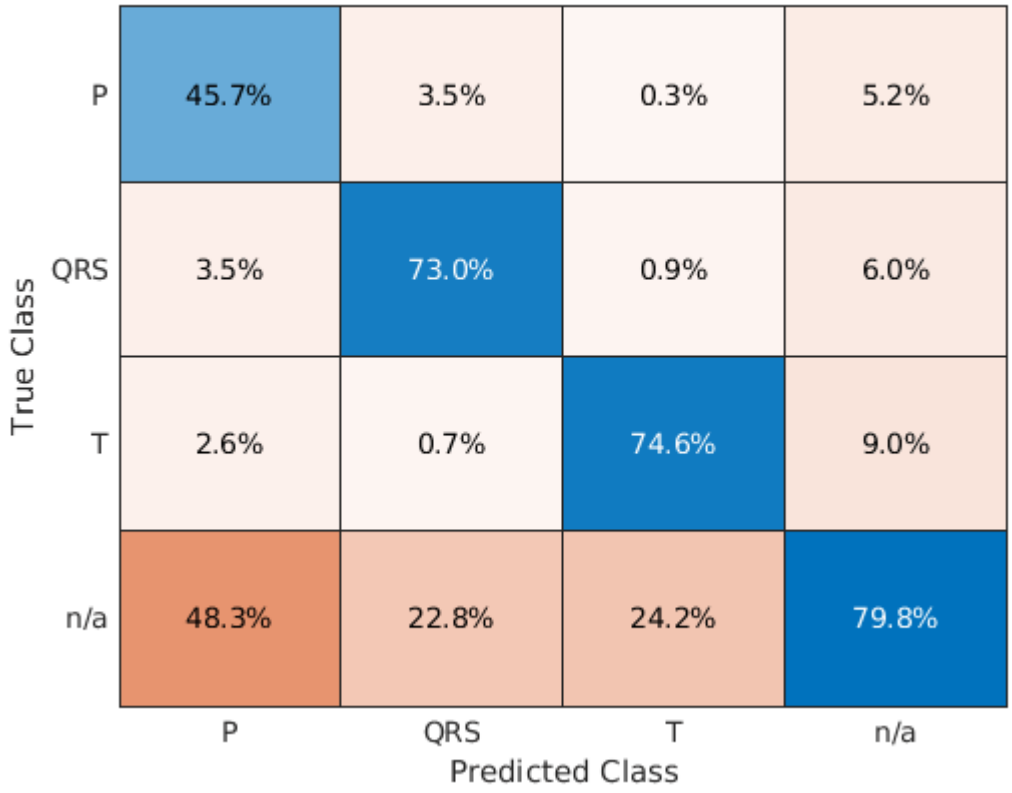
Classify Filtered ECG Signals

Classify the preprocessed test data with the updated LSTM network.

```
predFilteredTest = classify(filteredNet,filteredTestSignals,'MiniBatchSize',50);
```

Visualize the classification performance as a confusion matrix.

```
figure
confusionchart([predFilteredTest{:}],[testLabels{:}],'Normalization','column-normalized');
```



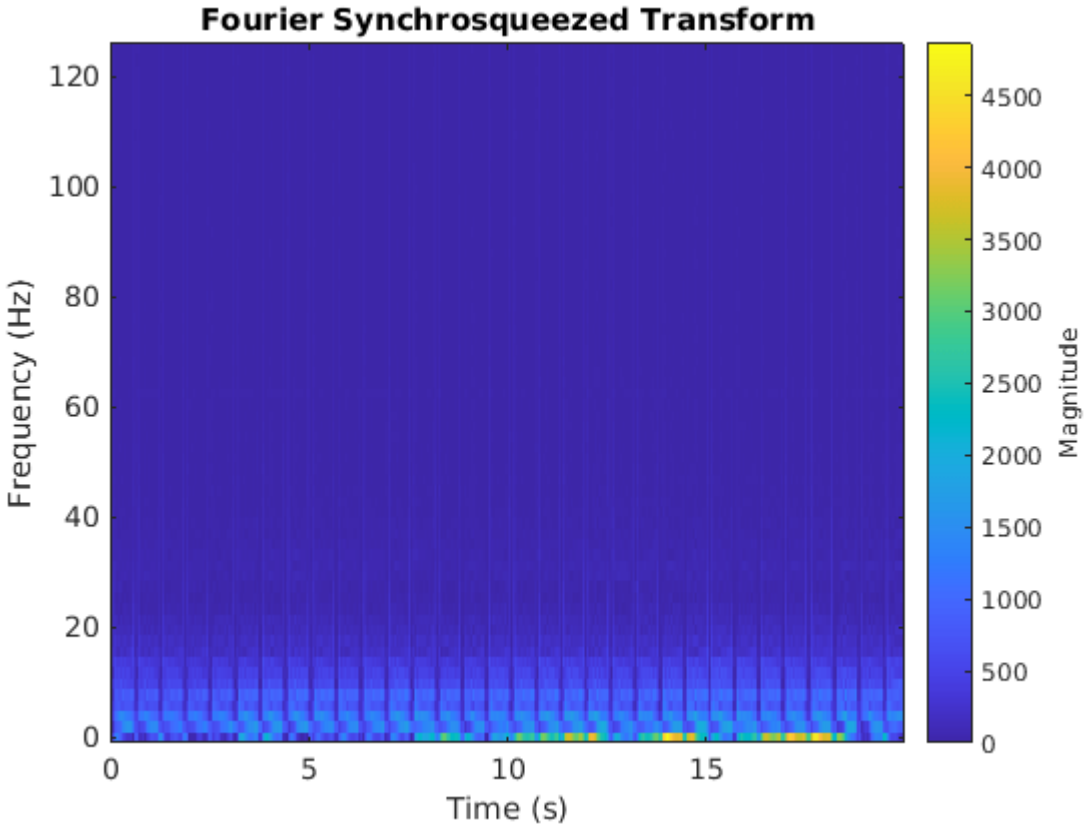
Simple preprocessing improves T-wave classification by about 15%, and QRS-complex and P-wave classification by about 10%.

Time-Frequency Representation of ECG Signals

A common approach for successful classification of time-series data is to extract time-frequency features and feed them to the network instead of the original data. The network then learns patterns across time and frequency simultaneously [7 on page 12-0].

The Fourier synchrosqueezed transform (FSST) computes a frequency spectrum for each signal sample so it is ideal for the segmentation problem at hand where we need to maintain the same time resolution as the original signals. Use the `fsst` function to inspect the transform of one of the training signals. Specify a Kaiser window of length 128 to provide adequate frequency resolution.

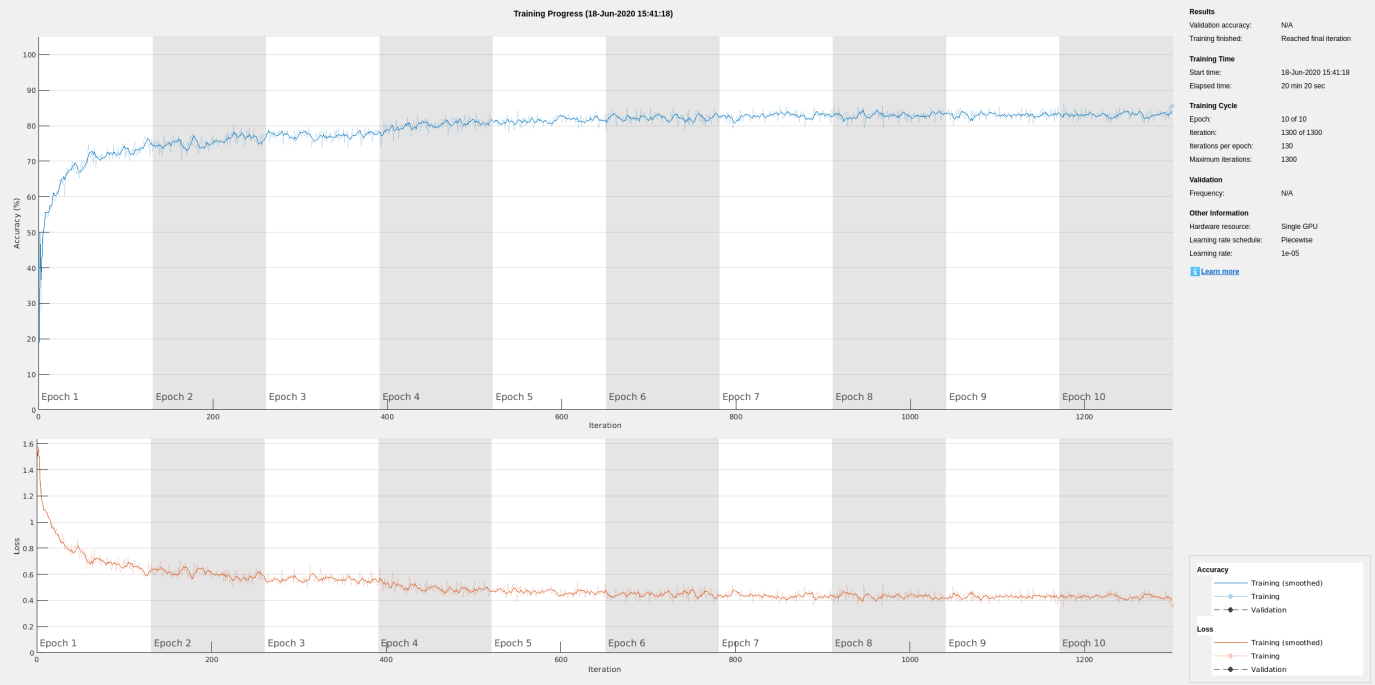
```
data =  preview(trainDs);
figure
fsst(data{1,1},250,kaiser(128),'yaxis')
```



Calculate the FSST of each signal in the training dataset over the frequency range of interest, [0.5, 40] Hz. Treat the real and imaginary parts of the FSST as separate features and feed both components into the network. Furthermore, standardize the training features by subtracting the mean and dividing by the standard deviation. Use a transformed datastore, the `extractFSSTFeatures` helper function, and the `tall` function to process the data in parallel.

```
fsstTrainDs = transform(trainDs,@(x)extractFSSTFeatures(x,250));
fsstTallTrainSet = tall(fsstTrainDs);
fsstTrainData = gather(fsstTallTrainSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```




```
- Pass 1 of 1: Completed in 2 min 35 sec
Evaluation completed in 2 min 35 sec
```

Repeat this procedure for the testing data.

```
fsstTTestDs = transform(testDs,@(x)extractFSSTFeatures(x,250));
fsstTallTestSet = tall(fsstTTestDs);
fsstTestData = gather(fsstTallTestSet);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 4 sec
Evaluation completed in 1 min 4 sec
```

Adjust Network Architecture

Modify the LSTM architecture so that the network accepts a frequency spectrum for each sample instead of a single value. Inspect the size of the FSST to see the number of frequencies.

```
size(fsstTrainData{1,1})
```

```
ans = 1×2

          40        5000
```

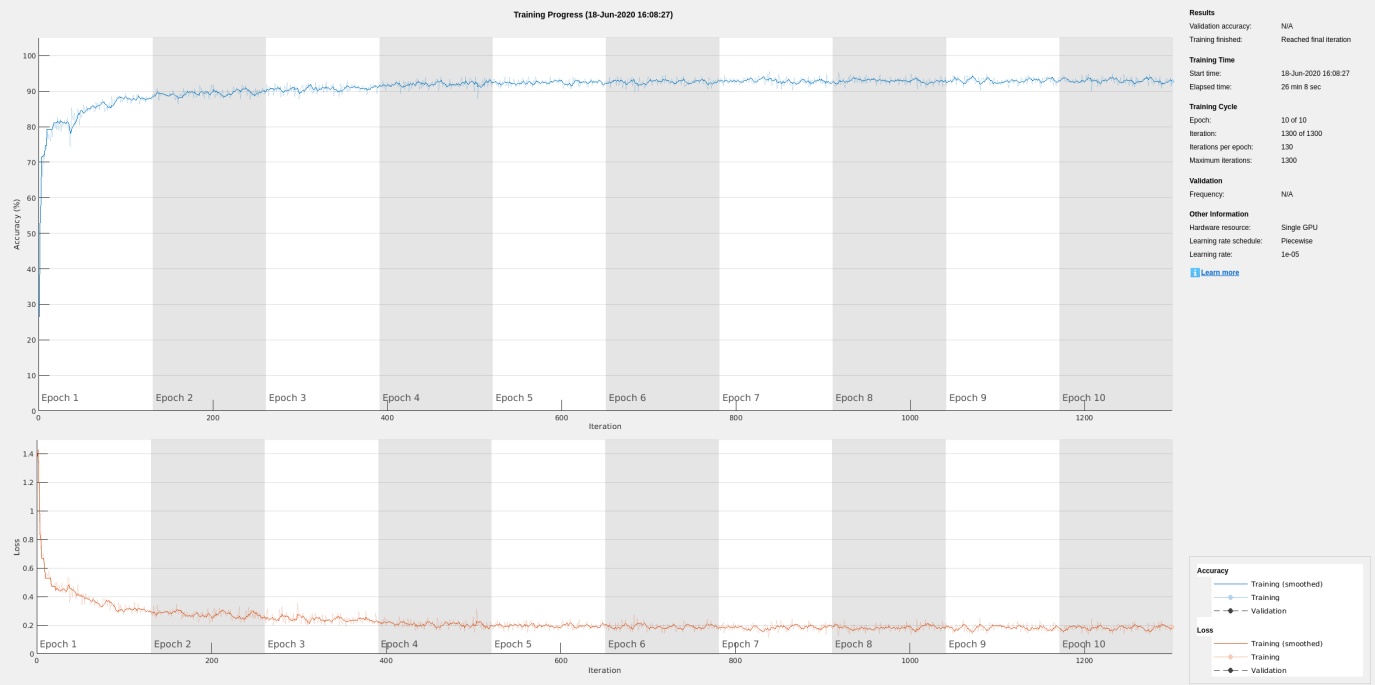
Specify a `sequenceInputLayer` of 40 input features. Keep the rest of the network parameters unchanged.

```
layers = [ ...
    sequenceInputLayer(40)
    lstmLayer(200,'OutputMode','sequence')
    fullyConnectedLayer(4)
    softmaxLayer
    classificationLayer];
```

Train Network with FSST of ECG Signals

Train the updated LSTM network with the transformed dataset.

```
if actionFlag == "Train networks"
    fsstNet = trainNetwork(fsstTrainData(:,1),fsstTrainData(:,2),layers,options);
end
```



Using time-frequency features improves the training accuracy, which now exceeds 90%.

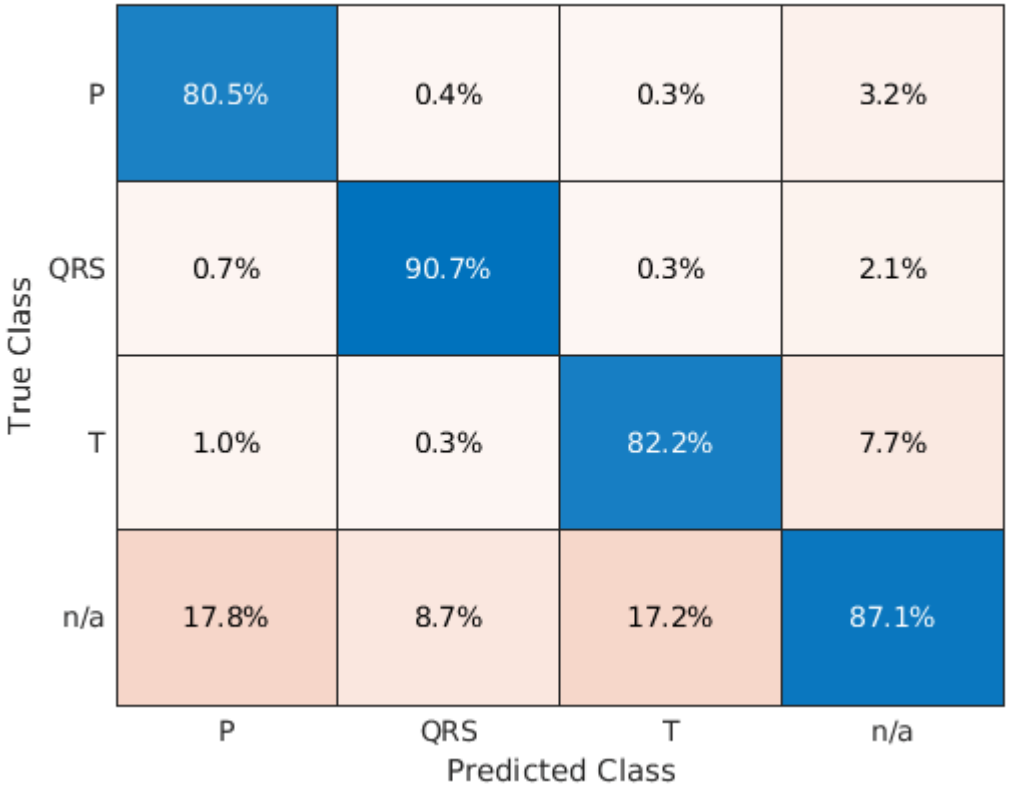
Classify Test Data with FSST

Using the updated LSTM network and extracted FSST features, classify the testing data.

```
predFsstTest = classify(fsstNet,fsstTestData(:,1),'MiniBatchSize',50);
```

Visualize the classification performance as a confusion matrix.

```
confusionchart([predFsstTest{:}],[fsstTestData{:,2}],'Normalization','column-normalized');
```

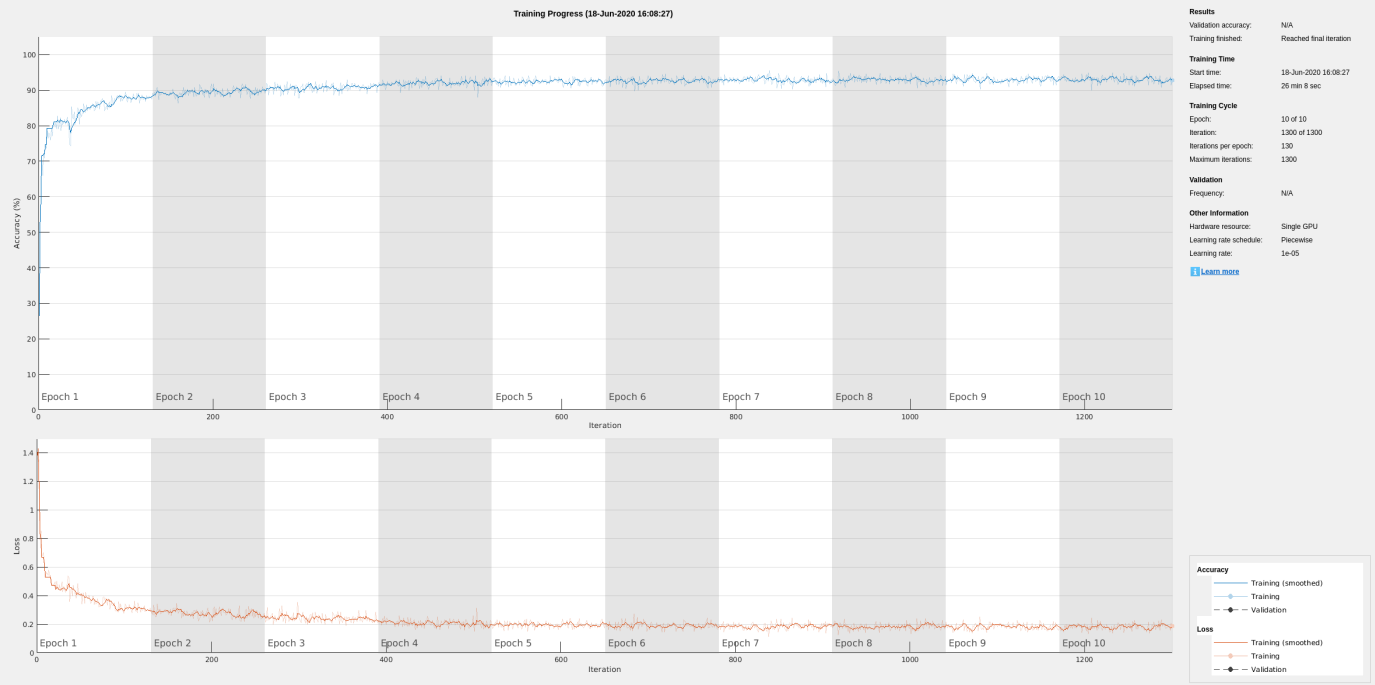


Using a time-frequency representation improves T-wave classification by about 25%, P-wave classification by about 40%, and QRS-complex classification by 30%, when compared to the raw data results.

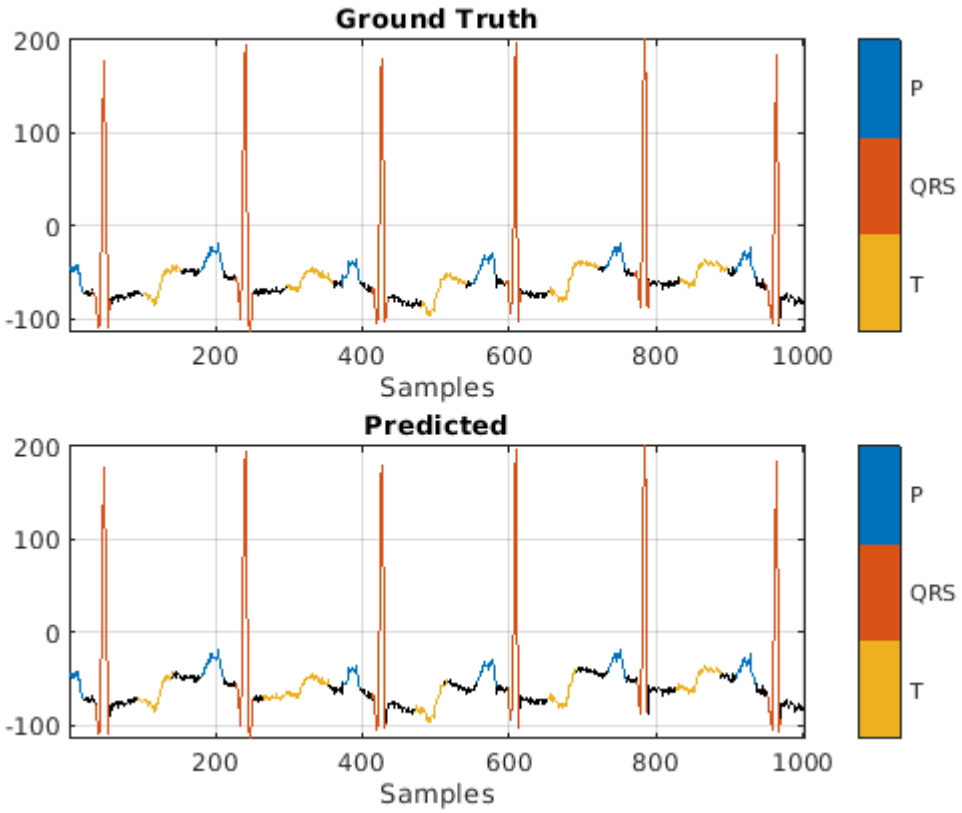
Use a `signalMask` object to compare the network prediction to the ground truth labels for a single ECG signal. Ignore the `"n/a"` labels when plotting the regions of interest.

```
testData = gather(tall(testDs));
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: 0% complete
Evaluation 0% complete
```




```
- Pass 1 of 1: Completed in 37 sec
Evaluation completed in 37 sec

Mtest = signalMask(testData{1,2}(3000:4000));
Mtest.SpecifySelectedCategories = true;
Mtest.SelectedCategories = find(Mtest.Categories ~= "n/a");

figure
subplot(2,1,1)
plotsigroi(Mtest,testData{1,1}(3000:4000))
title('Ground Truth')

Mpred = signalMask(predFsstTest{1}(3000:4000));
Mpred.SpecifySelectedCategories = true;
Mpred.SelectedCategories = find(Mpred.Categories ~= "n/a");

subplot(2,1,2)
plotsigroi(Mpred,testData{1,1}(3000:4000))
title('Predicted')
```



Conclusion

This example showed how signal preprocessing and time-frequency analysis can improve LSTM waveform segmentation performance. Bandpass filtering and Fourier-based synchrosqueezing result in an average improvement across all output classes from 55% to around 85%.

References

[1] McSharry, Patrick E., et al. "A dynamical model for generating synthetic electrocardiogram signals." *IEEE® Transactions on Biomedical Engineering.* Vol. 50, No. 3, 2003, pp. 289–294.

[2] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic detection of wave boundaries in multilead ECG signals: Validation with the CSE database." *Computers and Biomedical Research*. Vol. 27, No. 1, 1994, pp. 45–60.

[3] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation*. Vol. 101, No. 23, 2000, pp. e215–e220. [Circulation Electronic Pages; http://circ.ahajournals.org/content/101/23/e215.full].

[4] Laguna, Pablo, Roger G. Mark, Ary L. Goldberger, and George B. Moody. "A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG." *Computers in Cardiology.* Vol.24, 1997, pp. 673–676.

[5] Sörnmo, Leif, and Pablo Laguna. "Electrocardiogram (ECG) signal processing." *Wiley Encyclopedia of Biomedical Engineering,* 2006.

[6] Kohler, B-U., Carsten Hennig, and Reinhold Orglmeister. "The principles of software QRS detection." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 21, No. 1, 2002, pp. 42–57.

[7] Salamon, Justin, and Juan Pablo Bello. "Deep convolutional neural networks and data augmentation for environmental sound classification." *IEEE Signal Processing Letters.* Vol. 24, No. 3, 2017, pp. 279–283.

See Also

`confusionchart` | `fsst` | `labeledSignalSet` | `lstmLayer` | `trainingOptions` | `trainNetwork`

More About

- “Long Short-Term Memory Networks” on page 1-75
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42

Classify ECG Signals Using Long Short-Term Memory Networks

This example shows how to classify heartbeat electrocardiogram (ECG) data from the PhysioNet 2017 Challenge using deep learning and signal processing. In particular, the example uses Long Short-Term Memory networks and time-frequency analysis.

Introduction

ECGs record the electrical activity of a person's heart over a period of time. Physicians use ECGs to detect visually if a patient's heartbeat is normal or irregular.

Atrial fibrillation (AFib) is a type of irregular heartbeat that occurs when the heart's upper chambers, the atria, beat out of coordination with the lower chambers, the ventricles.

This example uses ECG data from the PhysioNet 2017 Challenge [1 on page 12-0], [2 on page 12-0], [3 on page 12-0], which is available at https://physionet.org/challenge/2017/. The data consists of a set of ECG signals sampled at 300 Hz and divided by a group of experts into four different classes: Normal (N), AFib (A), Other Rhythm (O), and Noisy Recording (~). This example shows how to automate the classification process using deep learning. The procedure explores a binary classifier that can differentiate Normal ECG signals from signals showing signs of AFib.

This example uses long short-term memory (LSTM) networks, a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. The LSTM layer (`lstmLayer`) can look at the time sequence in the forward direction, while the bidirectional LSTM layer (`bilstmLayer`) can look at the time sequence in both forward and backward directions. This example uses a bidirectional LSTM layer.

To accelerate the training process, run this example on a machine with a GPU. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB® automatically uses the GPU for training; otherwise, it uses the CPU.

Load and Examine the Data

Run the `ReadPhysionetData` script to download the data from the PhysioNet website and generate a MAT-file (`PhysionetData.mat`) that contains the ECG signals in the appropriate format. Downloading the data might take a few minutes. Use a conditional statement that runs the script only if `PhysionetData.mat` does not already exist in the current folder.

```
if ~isfile('PhysionetData.mat')
    ReadPhysionetData
end
load PhysionetData
```

The loading operation adds two variables to the workspace: `Signals` and `Labels`. `Signals` is a cell array that holds the ECG signals. `Labels` is a categorical array that holds the corresponding ground-truth labels of the signals.

```
Signals(1:5)
```

```
ans=5×1 cell array
    {1×9000  double}
    {1×9000  double}
    {1×18000 double}
    {1×9000  double}
```

```
    {1×18000 double}
```

```
Labels(1:5)
```

```
ans = 5×1 categorical
     N
     N
     N
     A
     A
```

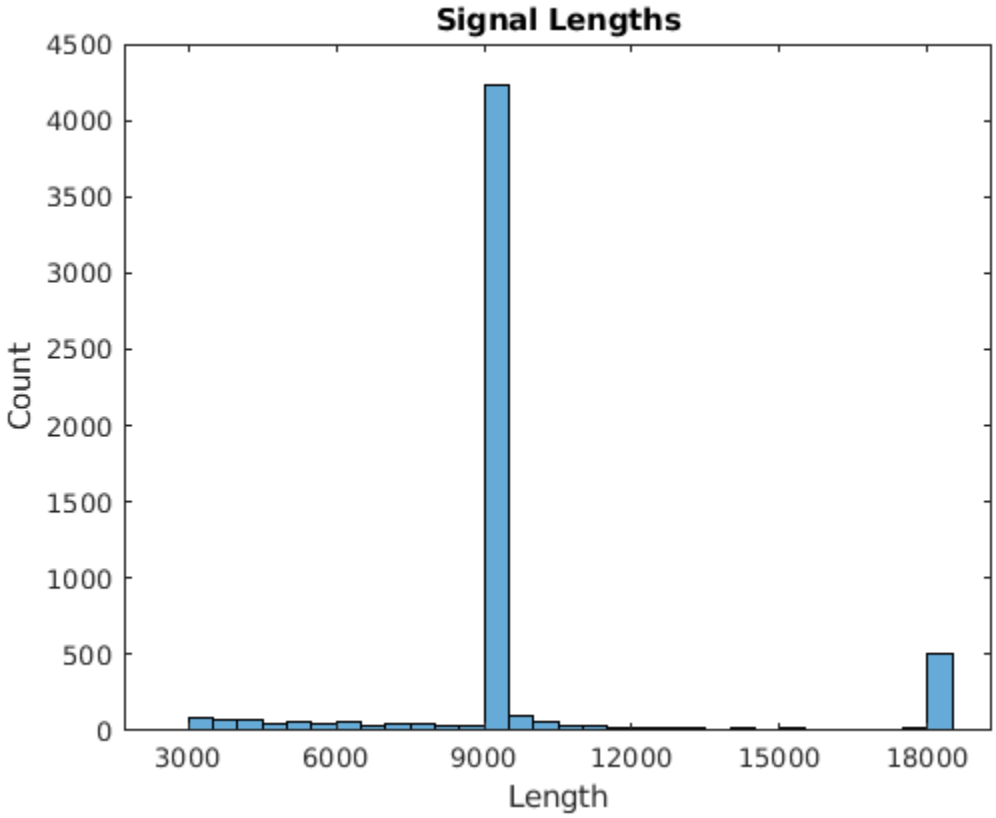
Use the `summary` function to see how many AFib signals and Normal signals are contained in the data.

```
summary(Labels)
```

```
     A       738
     N      5050
```

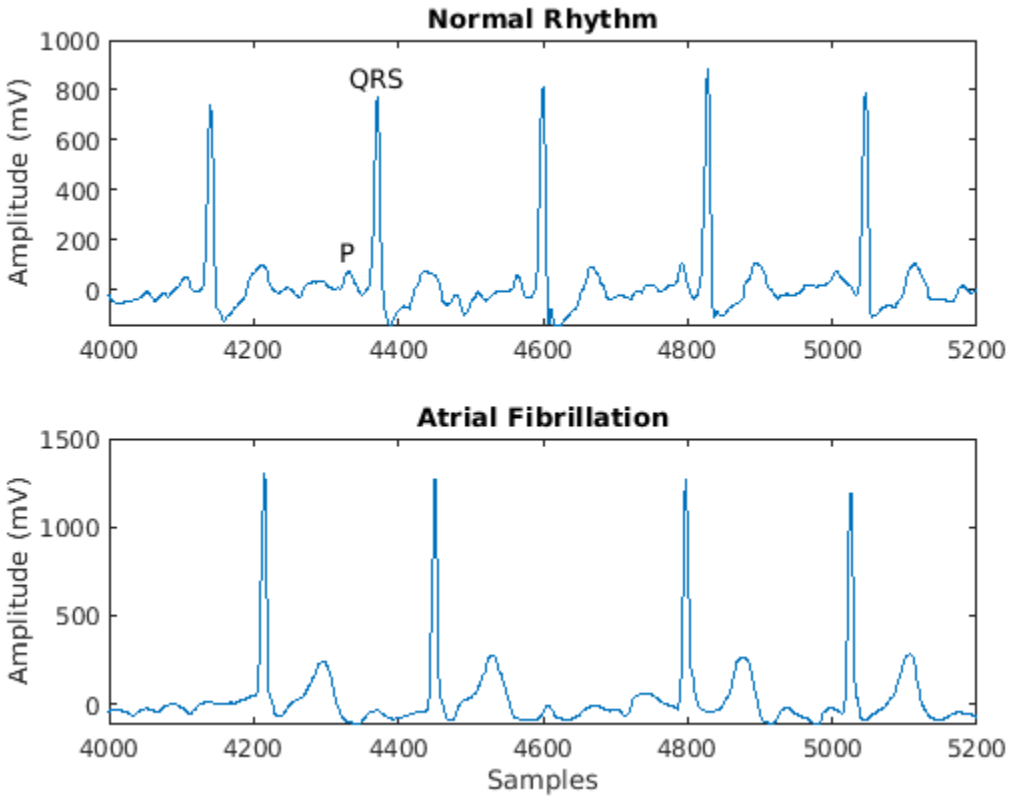
Generate a histogram of signal lengths. Most of the signals are 9000 samples long.

```
L = cellfun(@length,Signals);
h = histogram(L);
xticks(0:3000:18000);
xticklabels(0:3000:18000);
title('Signal Lengths')
xlabel('Length')
ylabel('Count')
```



Visualize a segment of one signal from each class. AFib heartbeats are spaced out at irregular intervals while Normal heartbeats occur regularly. AFib heartbeat signals also often lack a P wave, which pulses before the QRS complex in a Normal heartbeat signal. The plot of the Normal signal shows a P wave and a QRS complex.

```
normal = Signals{1};
aFib = Signals{4};

subplot(2,1,1)
plot(normal)
title('Normal Rhythm')
xlim([4000,5200])
ylabel('Amplitude (mV)')
text(4330,150,'P','HorizontalAlignment','center')
text(4370,850,'QRS','HorizontalAlignment','center')

subplot(2,1,2)
plot(aFib)
title('Atrial Fibrillation')
xlim([4000,5200])
xlabel('Samples')
ylabel('Amplitude (mV)')
```



Prepare the Data for Training

During training, the `trainNetwork` function splits the data into mini-batches. The function then pads or truncates signals in the same mini-batch so they all have the same length. Too much padding or truncating can have a negative effect on the performance of the network, because the network might interpret a signal incorrectly based on the added or removed information.

To avoid excessive padding or truncating, apply the `segmentSignals` function to the ECG signals so they are all 9000 samples long. The function ignores signals with fewer than 9000 samples. If a signal has more than 9000 samples, `segmentSignals` breaks it into as many 9000-sample segments as possible and ignores the remaining samples. For example, a signal with 18500 samples becomes two 9000-sample signals, and the remaining 500 samples are ignored.

```
[Signals,Labels] = segmentSignals(Signals,Labels);
```

View the first five elements of the `Signals` array to verify that each entry is now 9000 samples long.

```
Signals(1:5)
```

```
ans=5×1 cell array
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
    {1×9000 double}
```

Train the Classifier Using Raw Signal Data

To design the classifier, use the raw signals generated in the previous section. Split the signals into a training set to train the classifier and a testing set to test the accuracy of the classifier on new data.

Use the `summary` function to show that the ratio of AFib signals to Normal signals is 718:4937, or approximately 1:7.

```
summary(Labels)
```

```
     A       718
     N      4937
```

Because about 7/8 of the signals are Normal, the classifier would learn that it can achieve a high accuracy simply by classifying all signals as Normal. To avoid this bias, augment the AFib data by duplicating AFib signals in the dataset so that there is the same number of Normal and AFib signals. This duplication, commonly called oversampling, is one form of data augmentation used in deep learning.

Split the signals according to their class.

```
afibX = Signals(Labels=='A');
afibY = Labels(Labels=='A');

normalX = Signals(Labels=='N');
normalY = Labels(Labels=='N');
```

Next, use `dividerand` to divide targets from each class randomly into training and testing sets.

```
[trainIndA,~,testIndA] = dividerand(718,0.9,0.0,0.1);
[trainIndN,~,testIndN] = dividerand(4937,0.9,0.0,0.1);

XTrainA = afibX(trainIndA);
YTrainA = afibY(trainIndA);

XTrainN = normalX(trainIndN);
YTrainN = normalY(trainIndN);

XTestA = afibX(testIndA);
YTestA = afibY(testIndA);

XTestN = normalX(testIndN);
YTestN = normalY(testIndN);
```

Now there are 646 AFib signals and 4443 Normal signals for training. To achieve the same number of signals in each class, use the first 4438 Normal signals, and then use `repmat` to repeat the first 634 AFib signals seven times.

For testing, there are 72 AFib signals and 494 Normal signals. Use the first 490 Normal signals, and then use `repmat` to repeat the first 70 AFib signals seven times. By default, the neural network randomly shuffles the data before training, ensuring that contiguous signals do not all have the same label.

```
XTrain = [repmat(XTrainA(1:634),7,1); XTrainN(1:4438)];
YTrain = [repmat(YTrainA(1:634),7,1); YTrainN(1:4438)];

XTest = [repmat(XTestA(1:70),7,1); XTestN(1:490)];
YTest = [repmat(YTestA(1:70),7,1); YTestN(1:490);];
```

The distribution between Normal and AFib signals is now evenly balanced in both the training set and the testing set.

```
summary(YTrain)
```

```
     A      4438
     N      4438
```

```
summary(YTest)
```

```
     A      490
     N      490
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer`, as it looks at the sequence in both forward and backward directions.

Because the input signals have one dimension each, specify the input size to be sequences of size 1. Specify a bidirectional LSTM layer with an output size of 100 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map the input time series into 100 features and then prepares the output for the fully connected layer. Finally, specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(1)
    bilstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 1 dimensions
     2   ''   BiLSTM                 BiLSTM with 100 hidden units
     3   ''   Fully Connected         2 fully connected layer
     4   ''   Softmax                softmax
     5   ''   Classification Output   crossentropyex
```

Next specify the training options for the classifier. Set the `'MaxEpochs'` to 10 to allow the network to make 10 passes through the training data. A `'MiniBatchSize'` of 150 directs the network to look at 150 training signals at a time. An `'InitialLearnRate'` of 0.01 helps speed up the training process. Specify a `'SequenceLength'` of 1000 to break the signal into smaller pieces so that the machine does not run out of memory by looking at too much data at one time. Set '`GradientThreshold`' to 1 to stabilize the training process by preventing gradients from getting too large. Specify `'Plots'` as `'training-progress'` to generate plots that show a graphic of the training progress as the number of iterations increases. Set `'Verbose'` to `false` to suppress the table output that corresponds to the data shown in the plot. If you want to see this table, set `'Verbose'` to `true`.

This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with RNNs like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

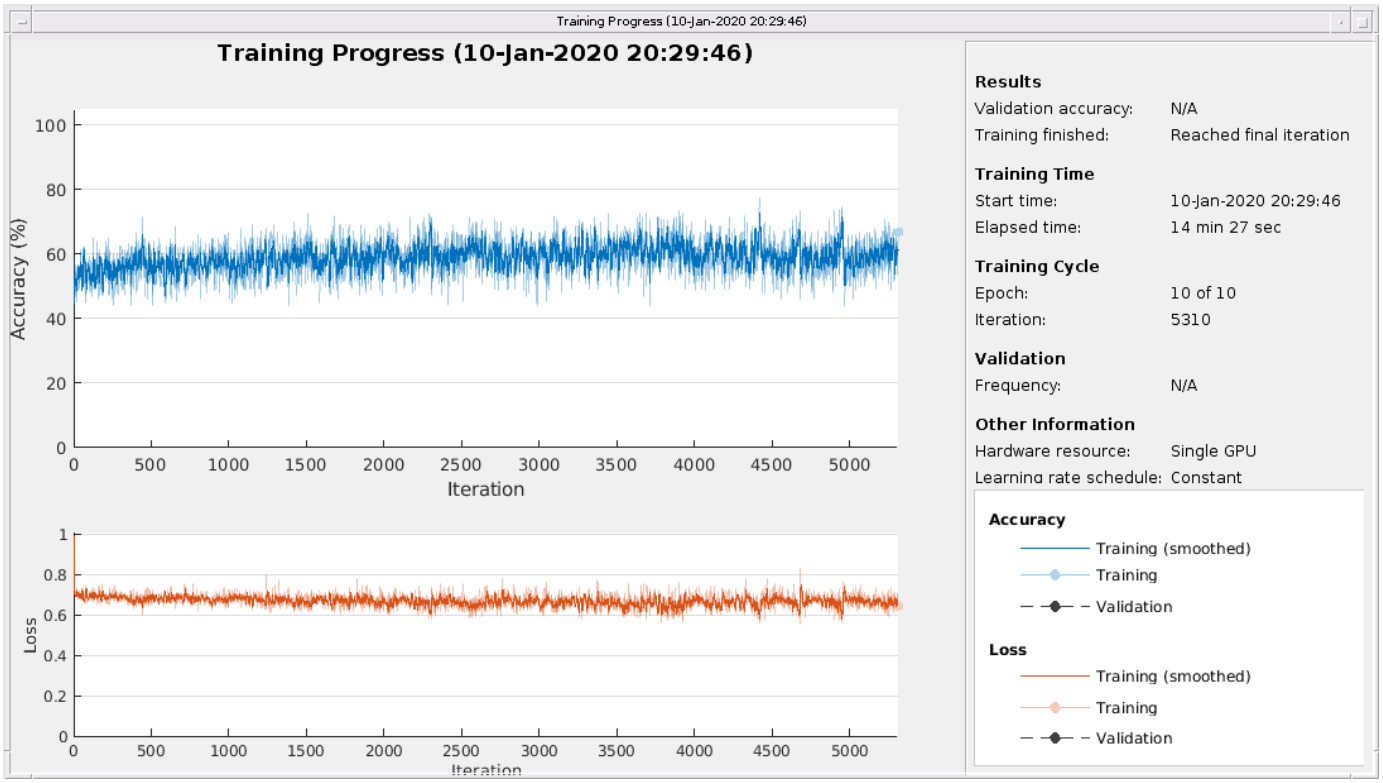
```
options = trainingOptions('adam', ...
    'MaxEpochs',10, ...
```

```
'MiniBatchSize', 150, ...
'InitialLearnRate', 0.01, ...
'SequenceLength', 1000, ...
'GradientThreshold', 1, ...
'ExecutionEnvironment',"auto",...
'plots','training-progress', ...
'Verbose',false);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture by using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
net = trainNetwork(XTrain,YTrain,layers,options);
```



The top subplot of the training-progress plot represents the training accuracy, which is the classification accuracy on each mini-batch. When training progresses successfully, this value typically increases towards 100%. The bottom subplot displays the training loss, which is the cross-entropy loss on each mini-batch. When training progresses successfully, this value typically decreases towards zero.

If the training is not converging, the plots might oscillate between values without trending in a certain upward or downward direction. This oscillation means that the training accuracy is not improving and the training loss is not decreasing. This situation can occur from the start of training, or the plots might plateau after some preliminary improvement in training accuracy. In many cases, changing the training options can help the network achieve convergence. Decreasing `MiniBatchSize` or decreasing `InitialLearnRate` might result in a longer training time, but it can help the network learn better.

The classifier's training accuracy oscillates between about 50% and about 60%, and at the end of 10 epochs, it already has taken several minutes to train.

Visualize the Training and Testing Accuracy

Calculate the training accuracy, which represents the accuracy of the classifier on the signals on which it was trained. First, classify the training data.

```
trainPred = classify(net,XTrain,'SequenceLength',1000);
```

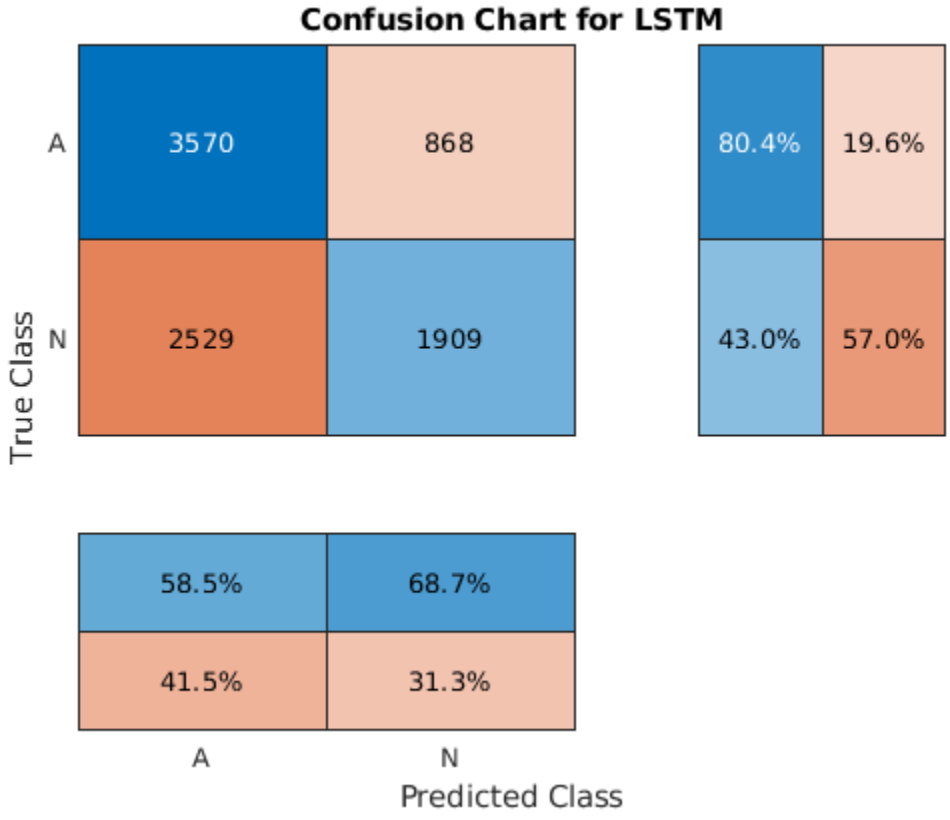
In classification problems, confusion matrices are used to visualize the performance of a classifier on a set of data for which the true values are known. The Target Class is the ground-truth label of the signal, and the Output Class is the label assigned to the signal by the network. The axes labels represent the class labels, AFib (A) and Normal (N).

Use the `confusionchart` command to calculate the overall classification accuracy for the testing data predictions. Specify `'RowSummary'` as `'row-normalized'` to display the true positive rates and false positive rates in the row summary. Also, specify `'ColumnSummary'` as `'column-normalized'` to display the positive predictive values and false discovery rates in the column summary.

```
LSTMAccuracy = sum(trainPred == YTrain)/numel(YTrain)*100
```

```
LSTMAccuracy = 61.7283
```

```
figure
confusionchart(YTrain,trainPred,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```



Now classify the testing data with the same network.

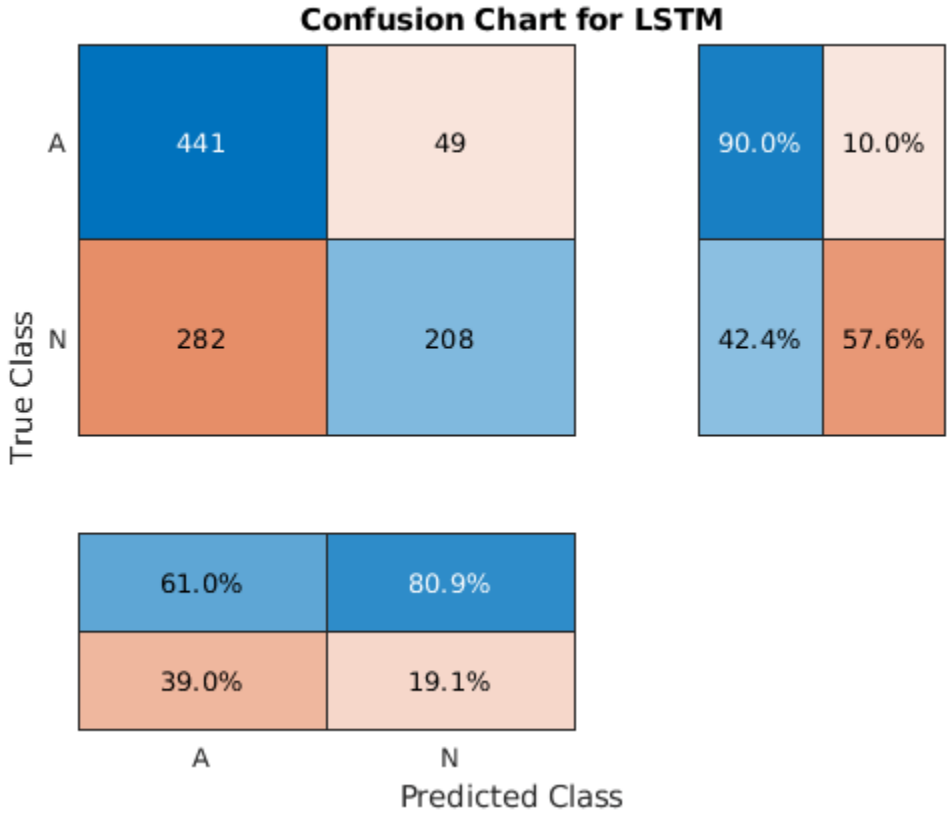
```
testPred = classify(net,XTest,'SequenceLength',1000);
```

Calculate the testing accuracy and visualize the classification performance as a confusion matrix.

```
LSTMAccuracy = sum(testPred == YTest)/numel(YTest)*100
```

```
LSTMAccuracy = 66.2245
```

```
figure
confusionchart(YTest,testPred,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```



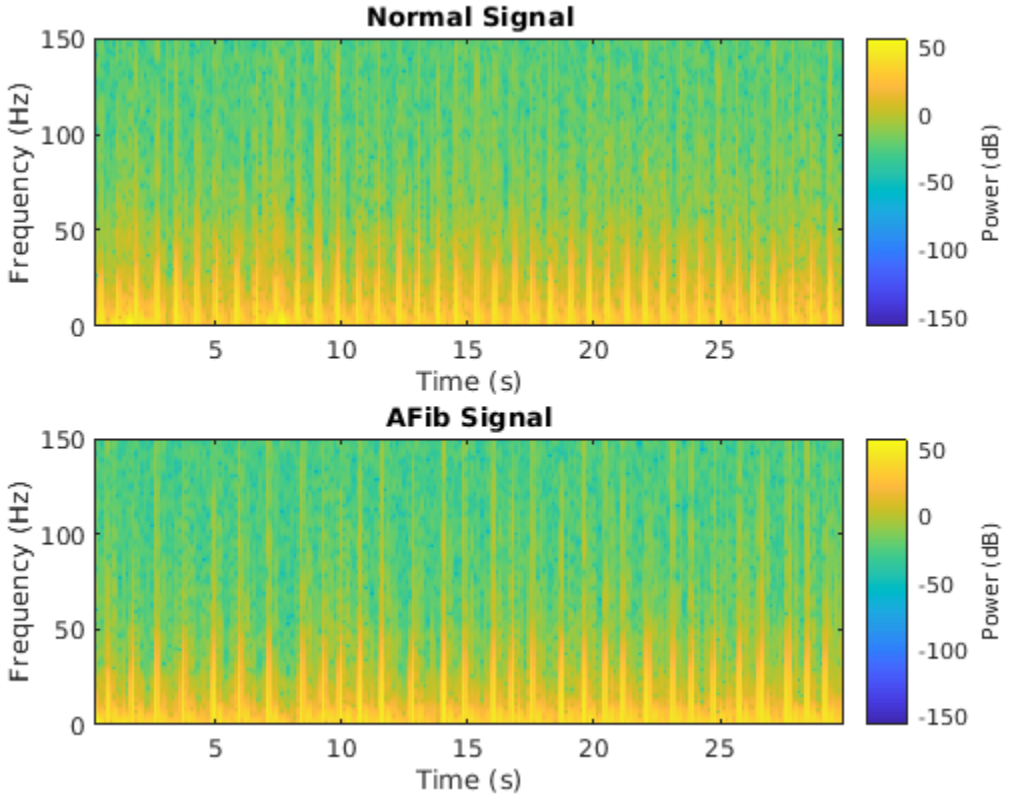
Improve the Performance with Feature Extraction

Feature extraction from the data can help improve the training and testing accuracies of the classifier. To decide which features to extract, this example adapts an approach that computes time-frequency images, such as spectrograms, and uses them to train convolutional neural networks (CNNs) [4 on page 12-0], [5 on page 12-0].

Visualize the spectrogram of each type of signal.

```
fs = 300;

figure
subplot(2,1,1);
pspectrum(normal,fs,'spectrogram','TimeResolution',0.5)
```

```
title('Normal Signal')

subplot(2,1,2);
pspectrum(aFib,fs,'spectrogram','TimeResolution',0.5)
title('AFib Signal')
```



Because this example uses an LSTM instead of a CNN, it is important to translate the approach so it applies to one-dimensional signals. Time-frequency (TF) moments extract information from the spectrograms. Each moment can be used as a one-dimensional feature to input to the LSTM.

Explore two TF moments in the time domain:

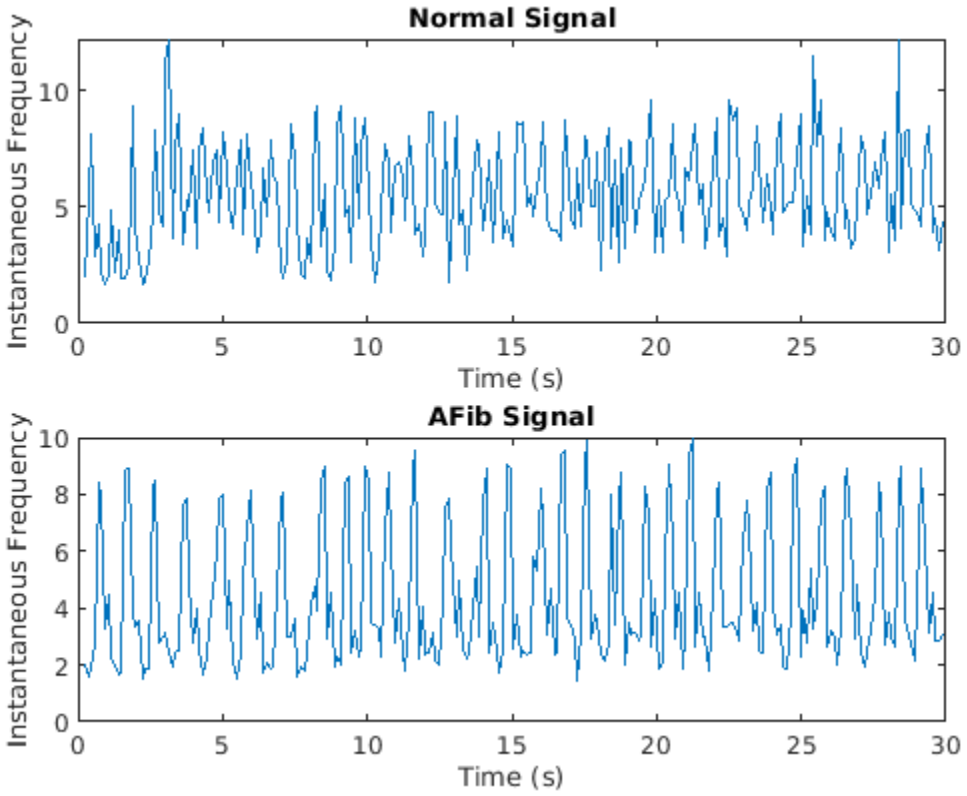
- Instantaneous frequency (`instfreq`)
- Spectral entropy (`pentropy`)

The `instfreq` function estimates the time-dependent frequency of a signal as the first moment of the power spectrogram. The function computes a spectrogram using short-time Fourier transforms over time windows. In this example, the function uses 255 time windows. The time outputs of the function correspond to the centers of the time windows.

Visualize the instantaneous frequency for each type of signal.

```
[instFreqA,tA] = instfreq(aFib,fs);
[instFreqN,tN] = instfreq(normal,fs);

figure
subplot(2,1,1);
```

```
plot(tN,instFreqN)
title('Normal Signal')
xlabel('Time (s)')
ylabel('Instantaneous Frequency')

subplot(2,1,2);
plot(tA,instFreqA)
title('AFib Signal')
xlabel('Time (s)')
ylabel('Instantaneous Frequency')
```



Use `cellfun` to apply the `instfreq` function to every cell in the training and testing sets.

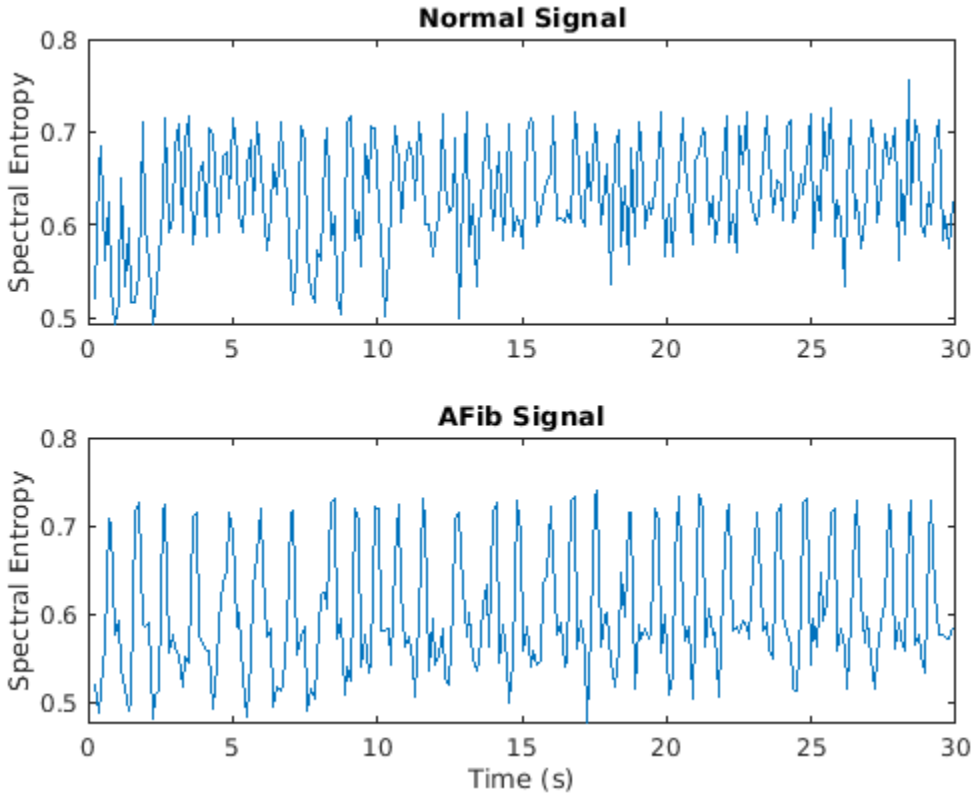
```
instfreqTrain = cellfun(@(x)instfreq(x,fs)',XTrain,'UniformOutput',false);
instfreqTest = cellfun(@(x)instfreq(x,fs)',XTest,'UniformOutput',false);
```

The spectral entropy measures how spiky flat the spectrum of a signal is. A signal with a spiky spectrum, like a sum of sinusoids, has low spectral entropy. A signal with a flat spectrum, like white noise, has high spectral entropy. The `pentropy` function estimates the spectral entropy based on a power spectrogram. As with the instantaneous frequency estimation case, `pentropy` uses 255 time windows to compute the spectrogram. The time outputs of the function correspond to the center of the time windows.

Visualize the spectral entropy for each type of signal.

```
[pentropyA,tA2] = pentropy(aFib,fs);
[pentropyN,tN2] = pentropy(normal,fs);
```

```
figure

subplot(2,1,1)
plot(tN2,pentropyN)
title('Normal Signal')
ylabel('Spectral Entropy')

subplot(2,1,2)
plot(tA2,pentropyA)
title('AFib Signal')
xlabel('Time (s)')
ylabel('Spectral Entropy')
```



Use `cellfun` to apply the `pentropy` function to every cell in the training and testing sets.

```
pentropyTrain = cellfun(@(x)pentropy(x,fs)',XTrain,'UniformOutput',false);
pentropyTest = cellfun(@(x)pentropy(x,fs)',XTest,'UniformOutput',false);
```

Concatenate the features such that each cell in the new training and testing sets has two dimensions, or two features.

```
XTrain2 = cellfun(@(x,y)[x;y],instfreqTrain,pentropyTrain,'UniformOutput',false);
XTest2 = cellfun(@(x,y)[x;y],instfreqTest,pentropyTest,'UniformOutput',false);
```

Visualize the format of the new inputs. Each cell no longer contains one 9000-sample-long signal; now it contains two 255-sample-long features.

```
XTrain2(1:5)
```

```
ans=5×1 cell array
    {2×255 double}
    {2×255 double}
    {2×255 double}
    {2×255 double}
    {2×255 double}
```

Standardize the Data

The instantaneous frequency and the spectral entropy have means that differ by almost one order of magnitude. Furthermore, the instantaneous frequency mean might be too high for the LSTM to learn effectively. When a network is fit on data with a large mean and a large range of values, large inputs could slow down the learning and convergence of the network [6 on page 12-0].

```
mean(instFreqN)
```

```
ans = 5.5615
```

```
mean(pentropyN)
```

```
ans = 0.6326
```

Use the training set mean and standard deviation to standardize the training and testing sets. Standardization, or z-scoring, is a popular way to improve network performance during training.

```
XV = [XTrain2{:}];
mu = mean(XV,2);
sg = std(XV,[],2);

XTrainSD = XTrain2;
XTrainSD = cellfun(@(x)(x-mu)./sg,XTrainSD,'UniformOutput',false);

XTestSD = XTest2;
XTestSD = cellfun(@(x)(x-mu)./sg,XTestSD,'UniformOutput',false);
```

Show the means of the standardized instantaneous frequency and spectral entropy.

```
instFreqNSD = XTrainSD{1}(1,:);
pentropyNSD = XTrainSD{1}(2,:);

mean(instFreqNSD)
```

```
ans = -0.3211
```

```
mean(pentropyNSD)
```

```
ans = -0.2416
```

Modify the LSTM Network Architecture

Now that the signals each have two dimensions, it is necessary to modify the network architecture by specifying the input sequence size as 2. Specify a bidirectional LSTM layer with an output size of 100, and output the last element of the sequence. Specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(2)
```

```
    bilstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 2 dimensions
     2   ''   BiLSTM                 BiLSTM with 100 hidden units
     3   ''   Fully Connected         2 fully connected layer
     4   ''   Softmax                softmax
     5   ''   Classification Output   crossentropyex
```

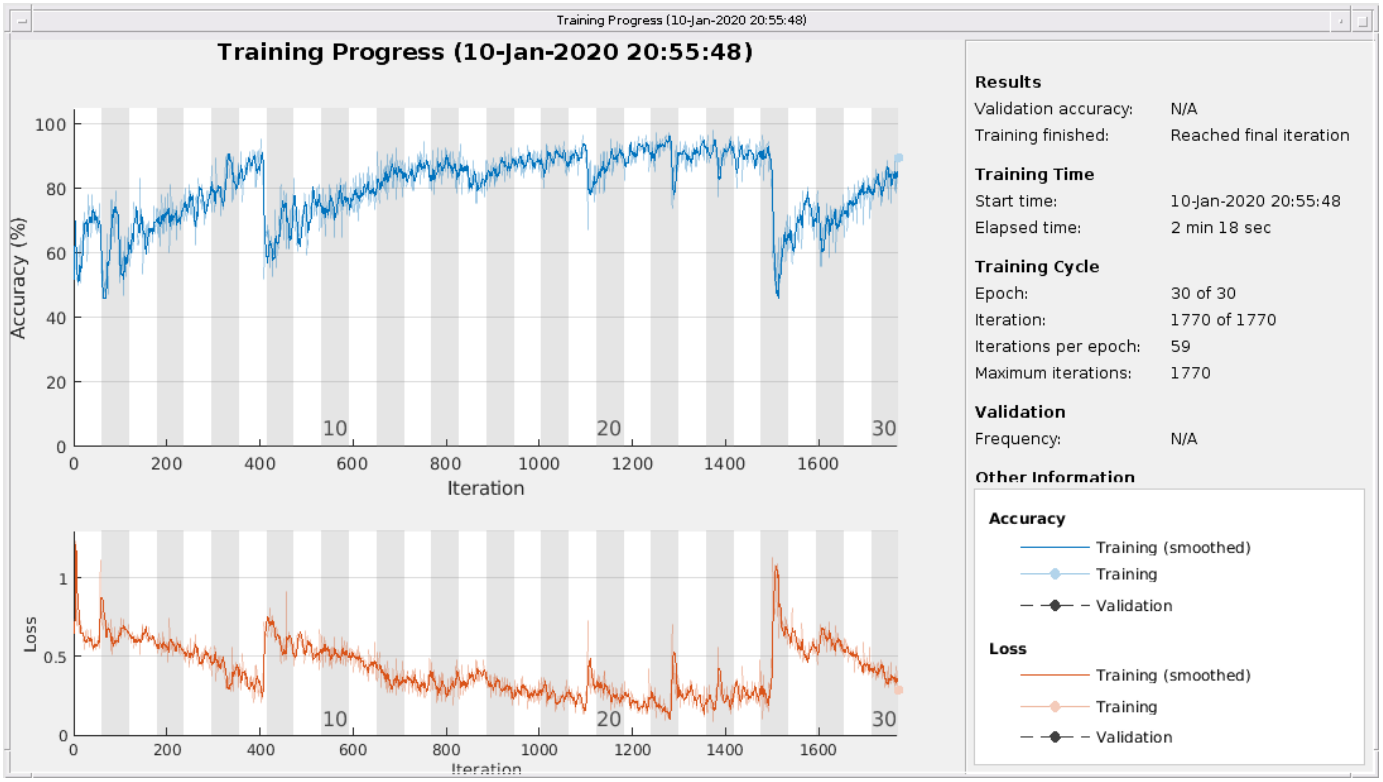
Specify the training options. Set the maximum number of epochs to 30 to allow the network to make 30 passes through the training data.

```
options = trainingOptions('adam', ...
    'MaxEpochs',30, ...
    'MiniBatchSize', 150, ...
    'InitialLearnRate', 0.01, ...
    'GradientThreshold', 1, ...
    'ExecutionEnvironment',"auto",...
    'plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM Network with Time-Frequency Features

Train the LSTM network with the specified training options and layer architecture by using `trainNetwork`.

```
net2 = trainNetwork(XTrainSD,YTrain,layers,options);
```



There is a great improvement in the training accuracy. The cross-entropy loss trends towards 0. Furthermore, the time required for training decreases because the TF moments are shorter than the raw sequences.

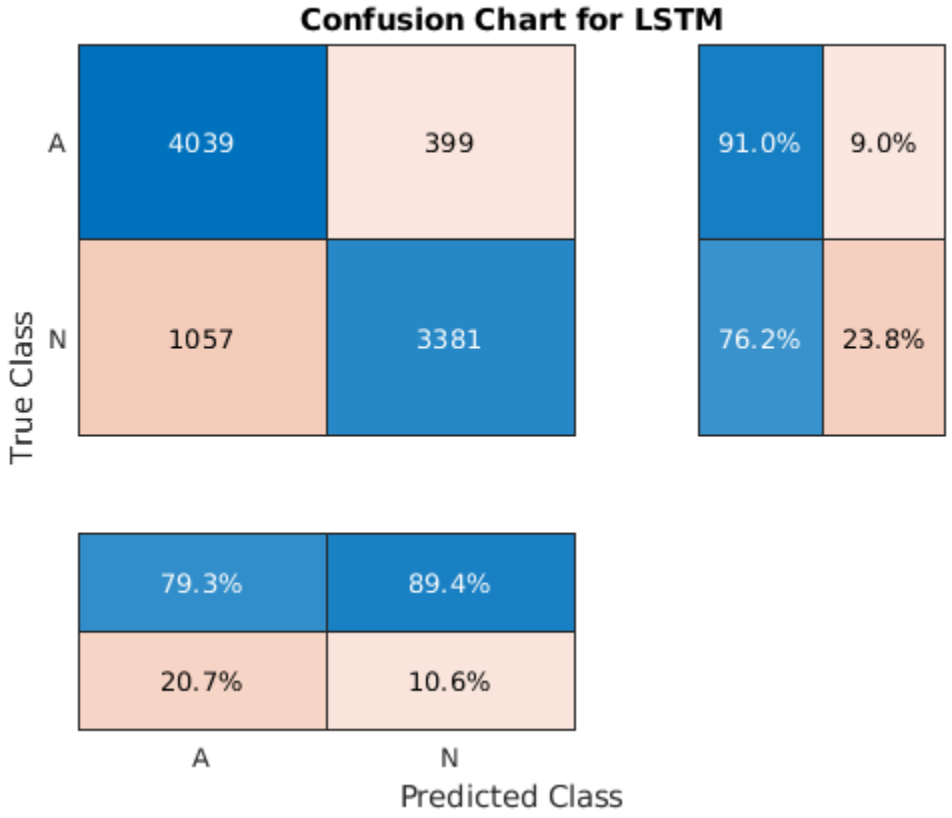
Visualize the Training and Testing Accuracy

Classify the training data using the updated LSTM network. Visualize the classification performance as a confusion matrix.

```
trainPred2 = classify(net2,XTrainSD);
LSTMAccuracy = sum(trainPred2 == YTrain)/numel(YTrain)*100
```

```
LSTMAccuracy = 83.5962
```

```
figure
confusionchart(YTrain,trainPred2,'ColumnSummary','column-normalized',...
              'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```

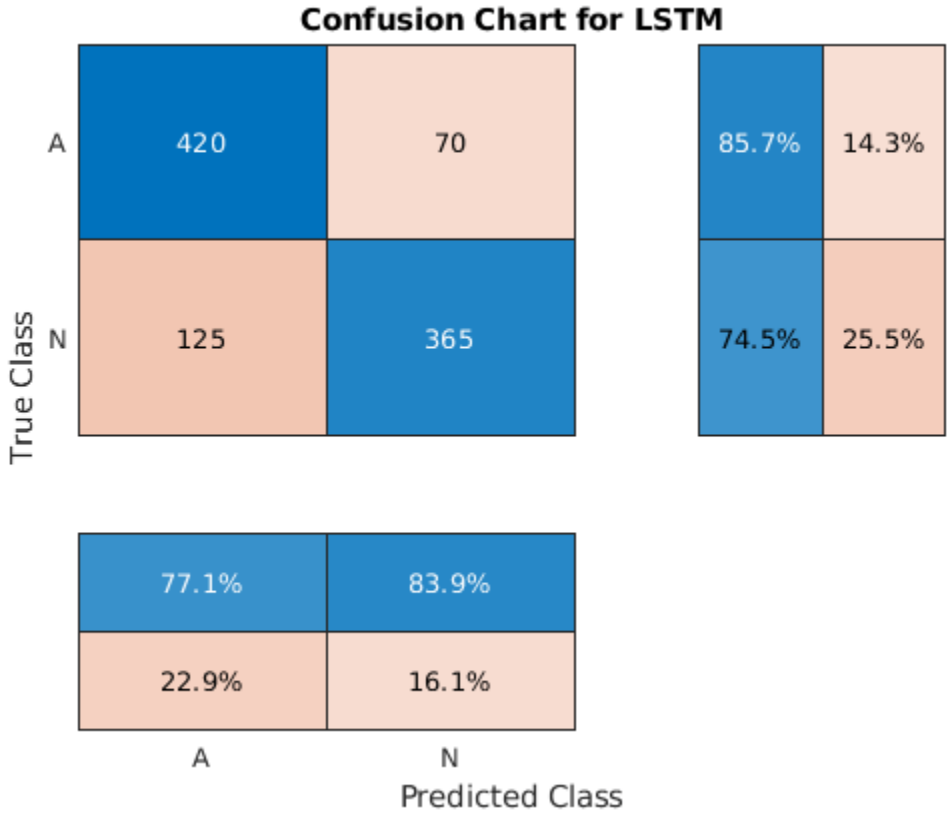


Classify the testing data with the updated network. Plot the confusion matrix to examine the testing accuracy.

```
testPred2 = classify(net2,XTestSD);

LSTMAccuracy = sum(testPred2 == YTest)/numel(YTest)*100
```

```
LSTMAccuracy = 80.1020
```

```
figure
confusionchart(YTest,testPred2,'ColumnSummary','column-normalized',...
               'RowSummary','row-normalized','Title','Confusion Chart for LSTM');
```



Conclusion

This example shows how to build a classifier to detect atrial fibrillation in ECG signals using an LSTM network. The procedure uses oversampling to avoid the classification bias that occurs when one tries to detect abnormal conditions in populations composed mainly of healthy patients. Training the LSTM network using raw signal data results in a poor classification accuracy. Training the network using two time-frequency-moment features for each signal significantly improves the classification performance and also decreases the training time.

References

[1] *AF Classification from a Short Single Lead ECG Recording: the PhysioNet/Computing in Cardiology Challenge, 2017.* https://physionet.org/challenge/2017/

[2] Clifford, Gari, Chengyu Liu, Benjamin Moody, Li-wei H. Lehman, Ikaro Silva, Qiao Li, Alistair Johnson, and Roger G. Mark. "AF Classification from a Short Single Lead ECG Recording: The PhysioNet Computing in Cardiology Challenge 2017." *Computing in Cardiology* (Rennes: IEEE). Vol. 44, 2017, pp. 1–4.

[3] Goldberger, A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals". *Circulation*. Vol. 101, No. 23, 13 June 2000, pp. e215–e220. http://circ.ahajournals.org/content/101/23/e215.full

[4] Pons, Jordi, Thomas Lidy, and Xavier Serra. "Experimenting with Musically Motivated Convolutional Neural Networks". *14th International Workshop on Content-Based Multimedia Indexing (CBMI)*. June 2016.

[5] Wang, D. "Deep learning reinvents the hearing aid," *IEEE Spectrum*, Vol. 54, No. 3, March 2017, pp. 32–37. doi: 10.1109/MSPEC.2017.7864754.

[6] Brownlee, Jason. *How to Scale Data for Long Short-Term Memory Networks in Python*. 7 July 2017. https://machinelearningmastery.com/how-to-scale-data-for-long-short-term-memory-networks-in-python/.

See Also

Functions

`instfreq` | `pentropy` | `trainingOptions` | `trainNetwork` | `bilstmLayer` | `lstmLayer`

More About

- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Generate Synthetic Signals Using Conditional GAN

This example shows how to generate synthetic pump signals using a conditional generative adversarial network.

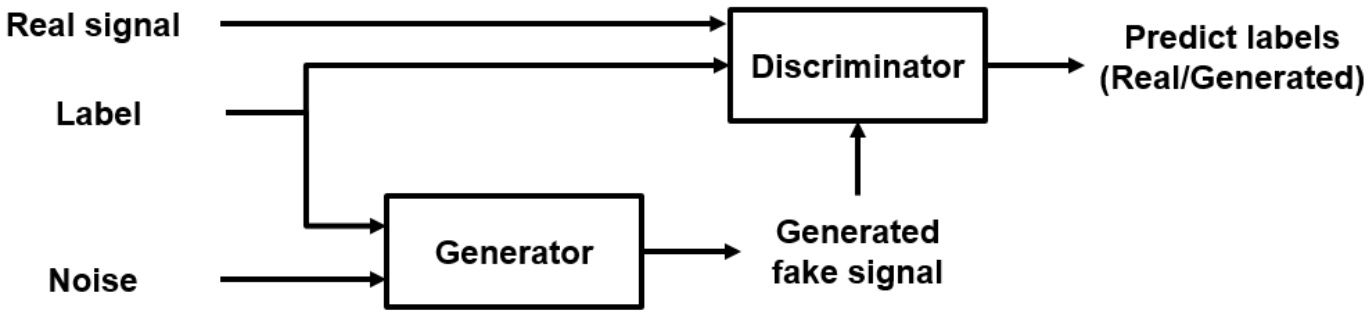
Generative adversarial networks (GANs) can be used to produce synthetic data that resembles real data input to the networks. GANs are useful when simulations are computationally expensive or experiments are costly. Conditional GANs (CGANs) can use data labels during the training process to generate data belonging to specific categories.

This example treats simulated signals obtained by a pump Simulink™ model as the "real" data that plays the role of training data set for a CGAN. The CGAN uses 1-D convolutional networks and is trained using a custom training loop and a deep learning array. In addition, this example uses principal component analysis (PCA) to visually compare the characteristics of generated and real signals.

CGAN for Signal Synthesis

CGANs consist of two networks that train together as adversaries:

1. *Generator* network — Given a label and random array as input, this network generates data with the same structure as the training data observations corresponding to the same label. The objective of the generator is to generate labeled data that the discriminator classifies as "real."
2. *Discriminator* network — Given batches of labeled data containing observations from both training data and generated data from the generator, this network attempts to classify the observations as "real" or "generated." The objective of the discriminator is to not be "fooled" by the generator when given batches of both real and generated labeled data.



Ideally, these strategies result in a generator that generates convincingly realistic data corresponding to the input labels and a discriminator that has learned strong features characteristic of the training data for each label.

Load Data

The simulated data is generated by the pump Simulink model presented in the “Multi-Class Fault Detection Using Simulated Data” (Predictive Maintenance Toolbox) example. The Simulink model is configured to model three types of faults: cylinder leaks, blocked inlets, and increased bearing friction. The data set contains 1575 pump output flow signals, of which 760 are healthy signals and 815 have a single fault, combinations of two faults, or combinations of three faults. Each signal has 1201 signal samples with a sample rate of 1000 Hz.

Download and unzip the data in your temporary directory, whose location is specified by MATLAB® `tempdir` command. If you have the data in a folder different from that specified by `tempdir`, change the directory name in the following code.

```
% Download the data
dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/PumpSignalGAN.zip';
saveFolder = fullfile(tempdir,'PumpSignalGAN');
zipFile = fullfile(tempdir,'PumpSignalGAN.zip');
if ~exist(saveFolder,'dir')
    websave(zipFile,dataURL);
end

% Unzip the data
unzip(zipFile,saveFolder)
```

The zip file contains the training data set and a pretrained CGAN:

- `simulatedDataset` — Simulated signals and their corresponding categorical labels
- `GANModel` — Generator and discriminator trained on the simulated data

Load the training data set and standardize the signals to have zero mean and unit variance.

```
load(fullfile(saveFolder,'simulatedDataset.mat')) % load data set
meanFlow = mean(flow,2);
flowNormalized = flow-meanFlow;
stdFlow = std(flowNormalized(:));
flowNormalized = flowNormalized/stdFlow;
```

Healthy signals are labeled as 1 and faulty signals are labeled as 2.

Define Generator Network

Define the following two-input network, which generates flow signals given 1-by-1-by-100 arrays of random values and corresponding labels.

The network:

- Projects and reshapes the 1-by-1-by-100 arrays of noise to 4-by-1-by-1024 arrays by a custom layer.
- Converts the categorical labels to embedding vectors and reshapes them to a 4-by-1-by-1 arrays.
- Concatenates the results from the two inputs along the channel dimension. The output is a 4-by-1-by-1025 array.
- Upsamples the resulting arrays to 1201-by-1-by-1 arrays using a series of 1-D transposed convolution layers with batch normalization and ReLU layers.

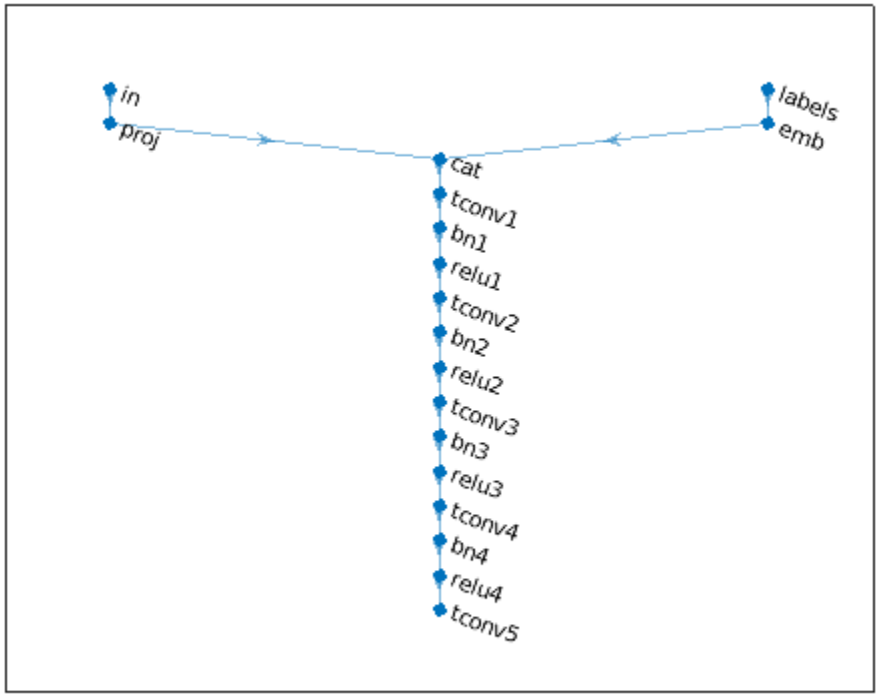
To project and reshape the noise input, use the custom layer `projectAndReshapeLayer`, attached to this example as a supporting file. The `projectAndReshapeLayer` object upscales the input using a fully connected layer and reshapes the output to the specified size.

To input the labels into the network, use an `imageInputLayer` object and specify a size of 1-by-1. To embed and reshape the label input, use the custom layer `embedAndReshapeLayer`, attached to this example as a supporting file. The `embedAndReshapeLayer` object converts a categorical label to a one-channel array of the specified size using an embedding and a fully connected operation. For categorical inputs, use an embedding dimension of 100.

```
% Generator Network

numFilters = 64;
numLatentInputs = 100;
projectionSize = [4 1 1024];
numClasses = 2;
embeddingDimension = 100;

layersGenerator = [
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj');
    concatenationLayer(3,2,'Name','cat');
    transposedConv2dLayer([5 1],8*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bn1','Epsilon',5e-5)
    reluLayer('Name','relu1')
    transposedConv2dLayer([10 1],4*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv2')
    batchNormalizationLayer('Name','bn2','Epsilon',5e-5)
    reluLayer('Name','relu2')
    transposedConv2dLayer([12 1],2*numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv3')
    batchNormalizationLayer('Name','bn3','Epsilon',5e-5)
    reluLayer('Name','relu3')
    transposedConv2dLayer([5 1],numFilters,'Stride',4,'Cropping',[1 0],'Name','tconv4')
    batchNormalizationLayer('Name','bn4','Epsilon',5e-5)
    reluLayer('Name','relu4')
    transposedConv2dLayer([7 1],1,'Stride',2,'Cropping',[1 0],'Name','tconv5')
    ];

lgraphGenerator = layerGraph(layersGenerator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(projectionSize(1:2),embeddingDimension,numClasses,'emb')];

lgraphGenerator = addLayers(lgraphGenerator,layers);
lgraphGenerator = connectLayers(lgraphGenerator,'emb','cat/in2');
```

Plot the network structure for the generator.

```
plot(lgraphGenerator)
```



To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Define Discriminator Network

Define the following two-input network, which classifies real and generated 1201-by-1 signals given a set of signals and their corresponding labels.

This network:

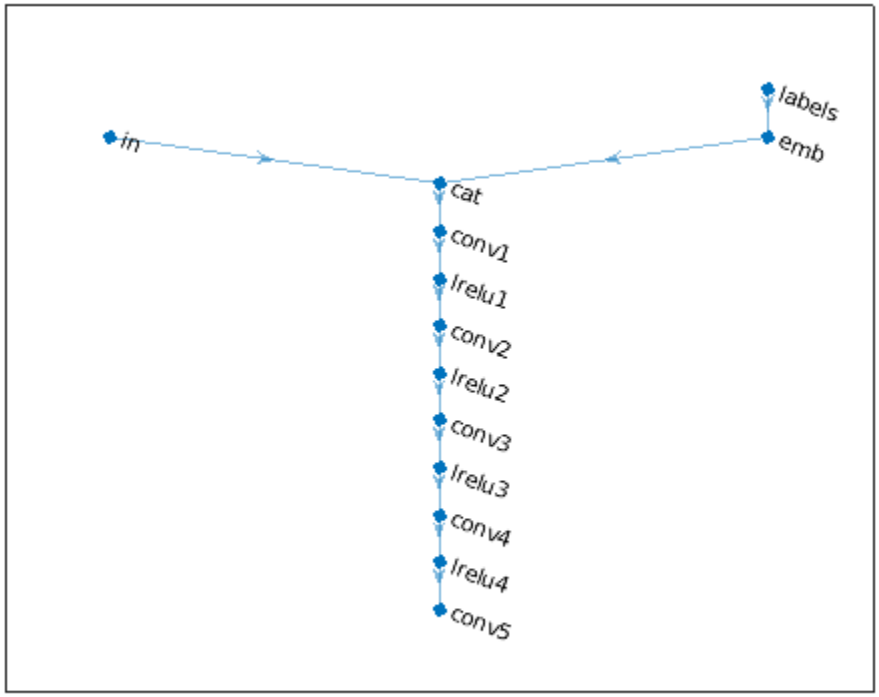
- Takes 1201-by-1-by-1 signals as input.
- Converts categorical labels to embedding vectors and reshapes them to a 1201-by-1-by-1 arrays.
- Concatenates the results from the two inputs along the channel dimension. The output is a 1201-by-1-by-1025 array.
- Downsamples the resulting arrays to scalar prediction scores, which are 1-by-1-by-1 arrays, using a series of 1-D convolution layers with leaky ReLU layers with a scale of 0.2.

```
% Discriminator Network

scale = 0.2;
inputSize = [1201 1 1];

layersDiscriminator = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
```

```
    concatenationLayer(3,2,'Name','cat')
    convolution2dLayer([17 1],8*numFilters,'Stride',2,'Padding',[1 0],'Name','conv1')
    leakyReluLayer(scale,'Name','lrelu1')
    convolution2dLayer([16 1],4*numFilters,'Stride',4,'Padding',[1 0],'Name','conv2')
    leakyReluLayer(scale,'Name','lrelu2')
    convolution2dLayer([16 1],2*numFilters,'Stride',4,'Padding',[1 0],'Name','conv3')
    leakyReluLayer(scale,'Name','lrelu3')
    convolution2dLayer([8 1],numFilters,'Stride',4,'Padding',[1 0],'Name','conv4')
    leakyReluLayer(scale,'Name','lrelu4')
    convolution2dLayer([8 1],1,'Name','conv5')];

lgraphDiscriminator = layerGraph(layersDiscriminator);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(inputSize,embeddingDimension,numClasses,'emb')];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,'emb','cat/in2');
```

Plot the network structure for the discriminator.

```
plot(lgraphDiscriminator)
```



To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

Train Model

Train the CGAN model using a custom training loop. Loop over the training data and update the network parameters at each iteration. To monitor the training progress, display generated healthy and faulty signals using two fixed arrays of random values to input into the generator as well as a plot of the scores of the two networks.

For each epoch, shuffle the training data and loop over mini-batches of data.

For each mini-batch:

- Generate a `dlarray` object containing an array of random values for the generator network.
- For GPU training, convert the data to a `gpuArray` (Parallel Computing Toolbox) object.
- Evaluate the model gradients using `dlfeval` and the helper function `modelGradients`.
- Update the network parameters using the `adamupdate` function.

The helper function `modelGradients` takes as input the generator and discriminator networks, a mini-batch of input data, and an array of random values, and returns the gradients of the loss with respect to the learnable parameters in the networks and the scores of the two networks. The loss function is defined in the helper function `ganLoss`.

Specify Training Options

Set the training parameters.

```
params.numLatentInputs = numLatentInputs;
params.numClasses = numClasses;
params.sizeData = [inputSize length(labels)];
params.numEpochs = 1000;
params.miniBatchSize = 256;

% Specify the options for Adam optimizer
params.learnRate = 0.0002;
params.gradientDecayFactor = 0.5;
params.squaredGradientDecayFactor = 0.999;
```

Set the execution environment to run the CGANs on the CPU. To run the CGANs on the GPU, set `executionEnvironment` to "`gpu`" or select the "Run on GPU" option in Live Editor. Using a GPU requires Parallel Computing Toolbox™. To see which GPUs are supported, see “GPU Support by Release” (Parallel Computing Toolbox).

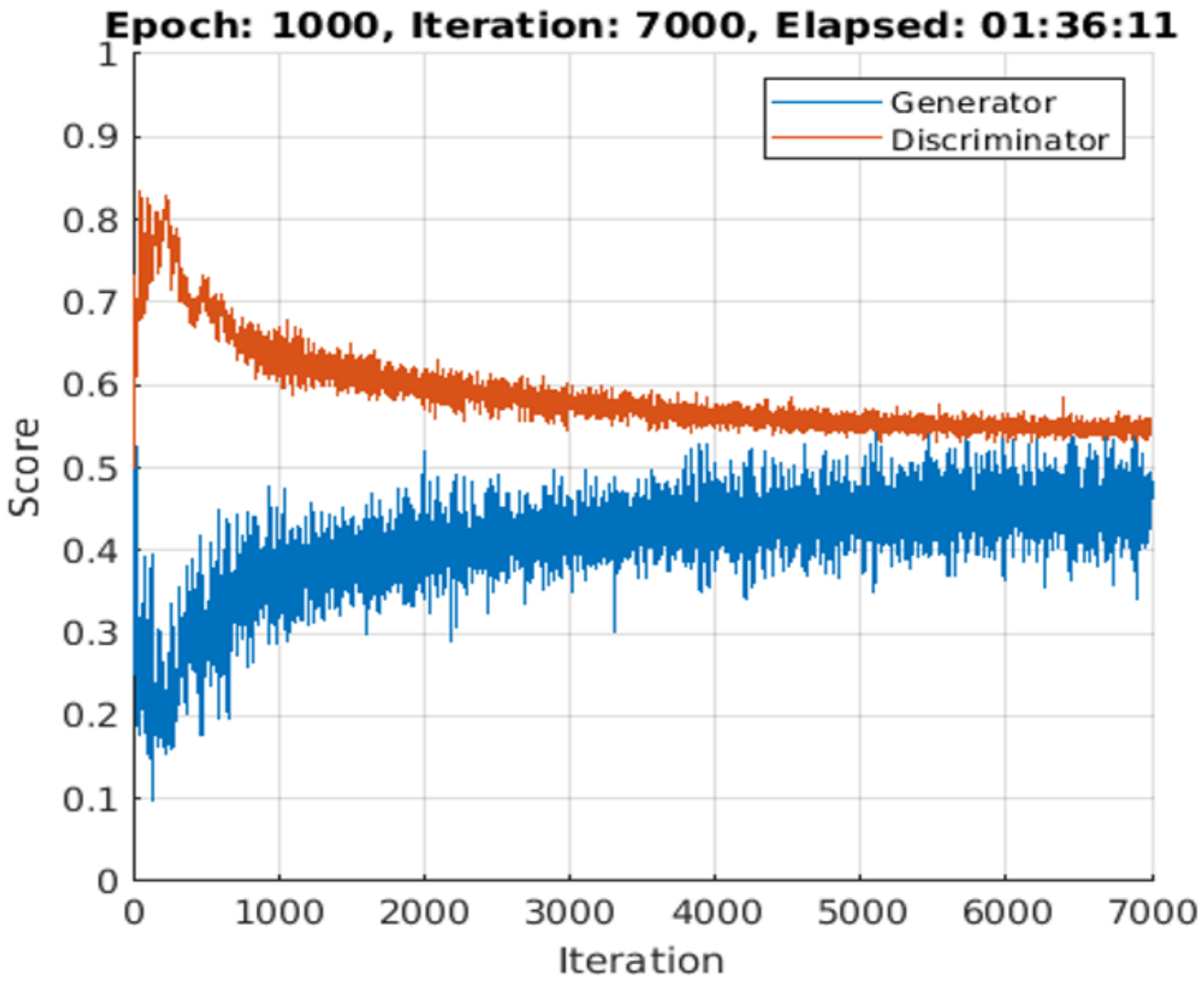
```
executionEnvironment = Run on CPU;
params.executionEnvironment = executionEnvironment;
```

Skip the training process by loading the pretrained network. To train the network on your computer, set `trainNow` to `true` or select the "Train CGAN now" option in Live Editor.

```
trainNow = Use trained CGAN;
if trainNow
    % Train the CGAN
    [dlnetGenerator,dlnetDiscriminator] = trainGAN(dlnetGenerator, ...
        dlnetDiscriminator,flowNormalized,labels,params); %#ok
else
    % Use pretrained CGAN (default)
```

```
    load(fullfile(tempdir,'PumpSignalGAN','GANModel.mat')) % load data set
end
```

The training plot below shows an example of scores of the generator and discriminator networks. To learn more about how to interpret the network scores, see “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-182. In this example, the scores of both the generator and discriminator converge close to 0.5, indicating that the training performance is good.



Synthesize Flow Signals

Create a `dlarray` object containing a batch of 2000 1-by-1-by-100 arrays of random values to input into the generator network. Reset the random number generator for reproducible results.

```
rng default

numTests = 2000;
```

```
ZNew = randn(1,1,numLatentInputs,numTests,'single');
dlZNew = dlarray(ZNew,'SSCB');
```

Specify that the first 1000 random arrays are healthy and the rest are faulty.

```
TNew = ones(1,1,1,numTests,'single');
TNew(1,1,1,numTests/2+1:end) = single(2);
dlTNew = dlarray(TNew,'SSCB');
```

To generate signals using the GPU, convert the data to `gpuArray` objects.

```
if executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
    dlTNew = gpuArray(dlTNew);
end
```

Use the `predict` function on the generator with the batch of 1-by-1-by-100 arrays of random values and labels to generate synthetic signals and revert the standardization step that you performed on the original flow signals.

```
dlXGeneratedNew = predict(dlnetGenerator,dlZNew,dlTNew)*stdFlow+meanFlow;
```

Signal Feature Visualization

Unlike images and audio signals, general signals have characteristics that make them difficult for human perception to tell apart. To compare real and generated signals or healthy and faulty signals, you can apply principal component analysis (PCA) to the statistical features of the real signals and then project the features of the generated signals to the same PCA subspace.

Feature Extraction

Combine the original real signal and the generated signals in one data matrix. Use the helper function `extractFeatures` to extract the feature including common signal statistics such as the mean and variance as well as spectral characteristics.

```
idxGenerated = 1:numTests;
idxReal = numTests+1:numTests+size(flow,2);

XGeneratedNew = squeeze(extractdata(gather(dlXGeneratedNew)));
x = [XGeneratedNew single(flow)];

features = zeros(size(x,2),14,'like',x);

for ii = 1:size(x,2)
    features(ii,:) = extractFeatures(x(:,ii));
end
```

Each row of `features` corresponds to the features of one signal.

Modify the labels for the generated healthy and faulty signals as well as real healthy and faulty signals.

```
L = [squeeze(TNew)+2;labels.'];
```

The labels now have these definitions:

- 1 — Generated healthy signals

- 2 — Generated faulty signals
- 3 — Real healthy signals
- 4 — Real faulty signals

Principal Component Analysis

Perform PCA on the features of the real signals and project the features of the generated signals to the same PCA subspace. `W` is the coefficient and `Y` is the score.

```
% PCA via svd
featuresReal = features(idxReal,:);
mu = mean(featuresReal,1);
[~,S,W] = svd(featuresReal-mu);
S = diag(S);
Y = (features-mu)*W;
```

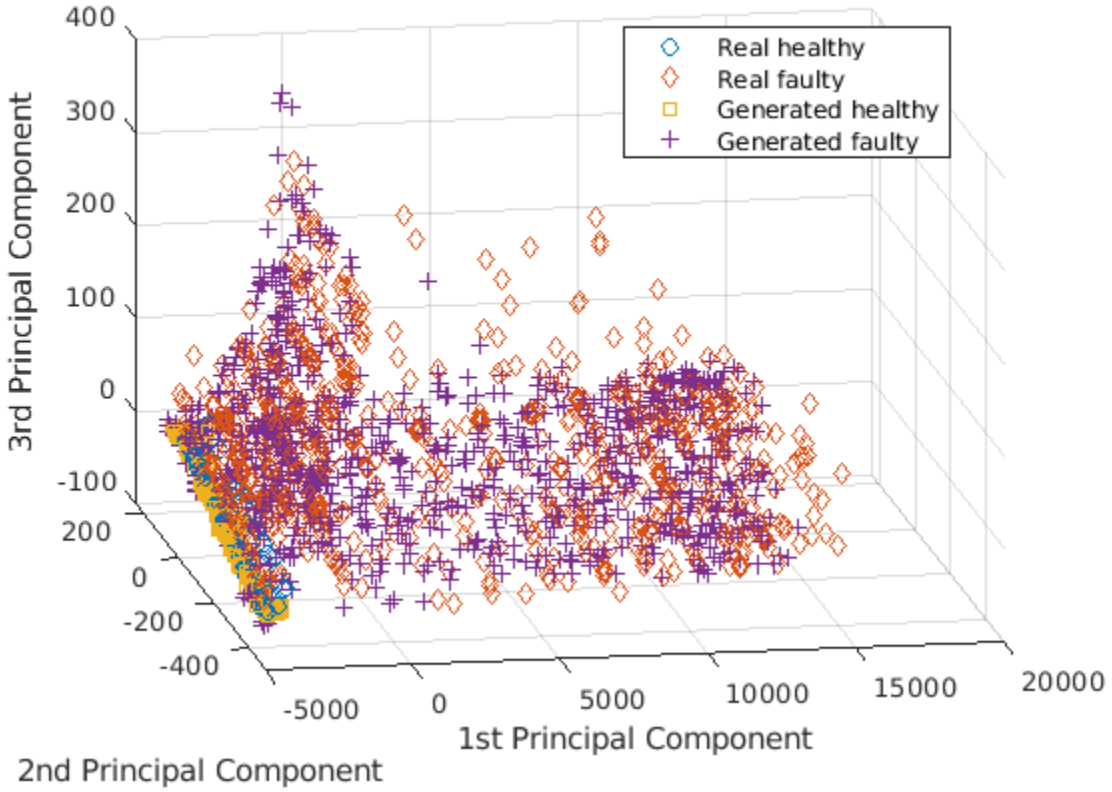
From the singular vector `S`, the first three singular values make up 99% of the energy in `S`. You can visualize the signal features by taking advantage of the first three principal components.

```
sum(S(1:3))/sum(S)
```

```
ans = single
    0.9923
```

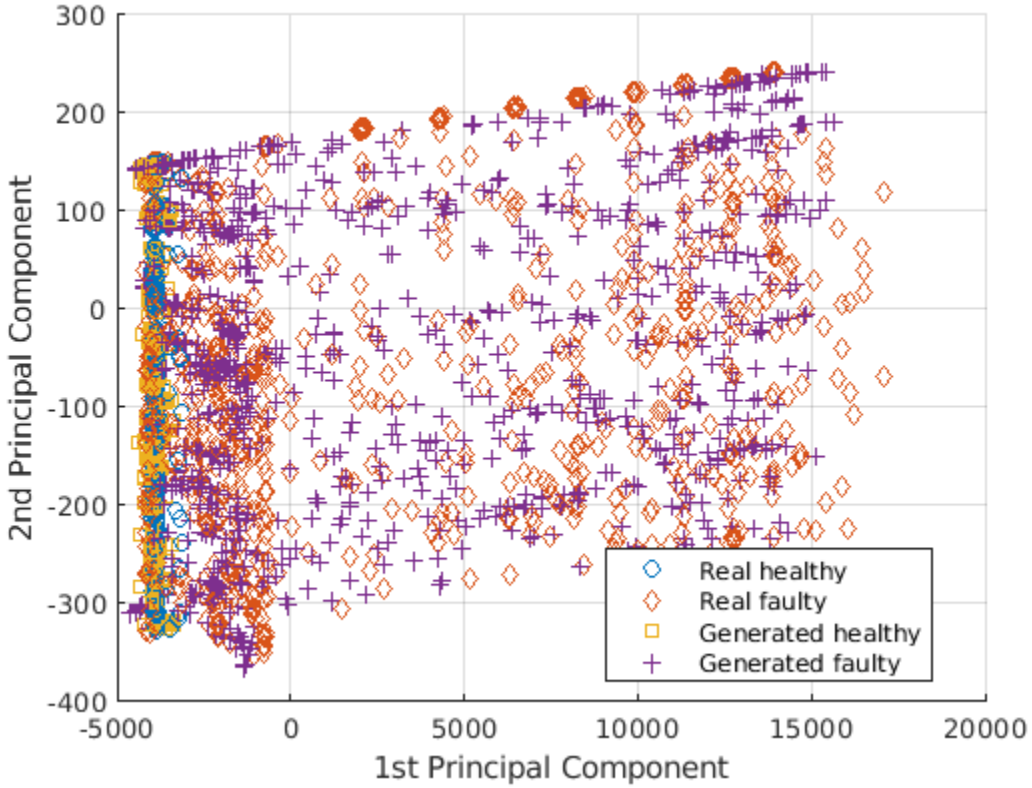
Plot the features of all the signals using the first three principal components. In the PCA subspace, the distribution of the generated signals is similar to the distribution of the real signals.

```
idxHealthyR = L==1;
idxFaultR = L==2;

idxHealthyG = L==3;
idxFaultG = L==4;

pp = Y(:,1:3);

figure
scatter3(pp(idxHealthyR,1),pp(idxHealthyR,2),pp(idxHealthyR,3),'o')
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
zlabel('3rd Principal Component')
hold on
scatter3(pp(idxFaultR,1),pp(idxFaultR,2),pp(idxFaultR,3),'d')
scatter3(pp(idxHealthyG,1),pp(idxHealthyG,2),pp(idxHealthyG,3),'s')
scatter3(pp(idxFaultG,1),pp(idxFaultG,2),pp(idxFaultG,3),'+')
view(-10,20)
legend('Real healthy','Real faulty','Generated healthy','Generated faulty', ...
    'Location','Best')
hold off
```



To better capture the difference between the real signals and generated signals, plot the subspace using the first two principal components.

```
view(2)
```



Healthy and faulty signals lie in the same area of the PCA subspace regardless of their being real or generated, demonstrating that the generated signals have features similar to those of the real signals.

Predict Labels of Real Signals

To further illustrate the performance of the CGAN, train an SVM classifier based on the generated signals and then predict whether a real signal is healthy or faulty.

Set the generated signals as the training data set and the real signals as the test data set. Change the numeric labels to character vectors.

```
LABELS = {'Healthy','Faulty'};
strL = LABELS([squeeze(TNew);labels.']).';

dataTrain = features(idxGenerated,:);
dataTest = features(idxReal,:);

labelTrain = strL(idxGenerated);
labelTest = strL(idxReal);

predictors = dataTrain;
response = labelTrain;
cvp = cvpartition(size(predictors,1),'KFold',5);
```

Train an SVM classifier using the generated signals.

```
SVMClassifier = fitcsvm( ...
    predictors(cvp.training(1),:), ...
    response(cvp.training(1)),'KernelFunction','polynomial', ...
    'PolynomialOrder',2, ...
    'KernelScale','auto', ...
    'BoxConstraint',1, ...
    'ClassNames',LABELS, ...
    'Standardize',true);
```

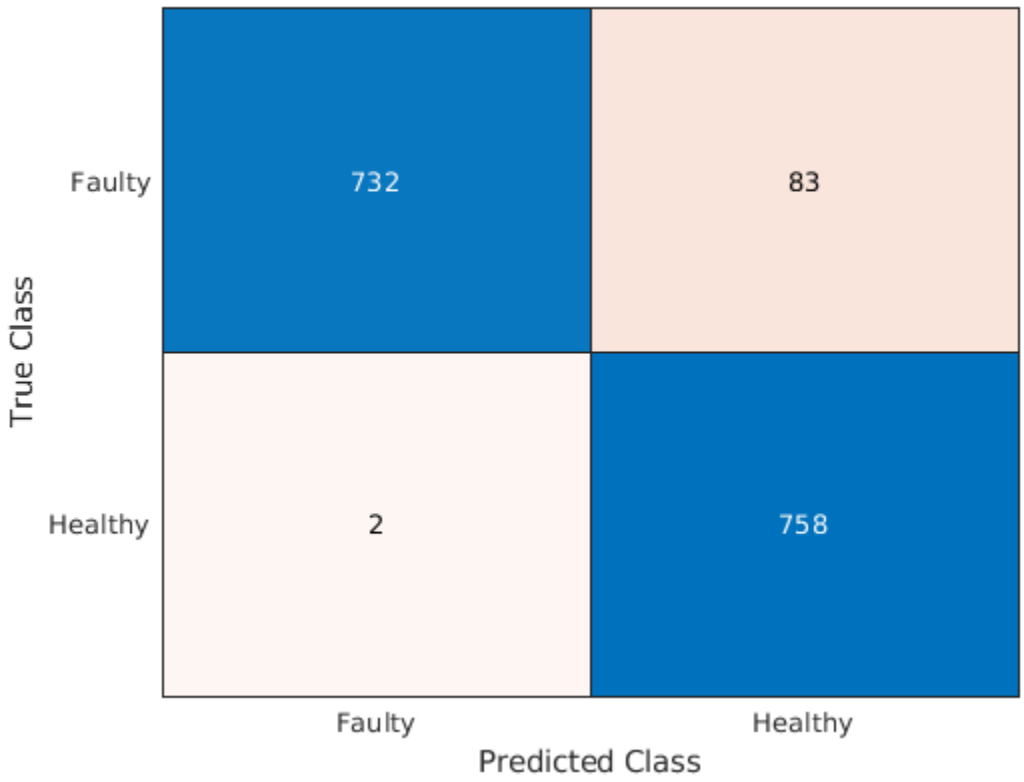
Use the trained classifier to obtain the predicted labels for the real signals. The classifier achieves a prediction accuracy above 90%.

```
actualValue = labelTest;
predictedValue = predict(SVMClassifier,dataTest);
predictAccuracy = mean(cellfun(@strcmp,actualValue,predictedValue))
```

```
predictAccuracy = 0.9460
```

Use a confusion matrix to view detailed information about prediction performance for each category. The confusion matrix shows that, in each category, the classifier trained based on the generated signals achieves a high degree of accuracy.

```
figure
confusionchart(actualValue,predictedValue)
```

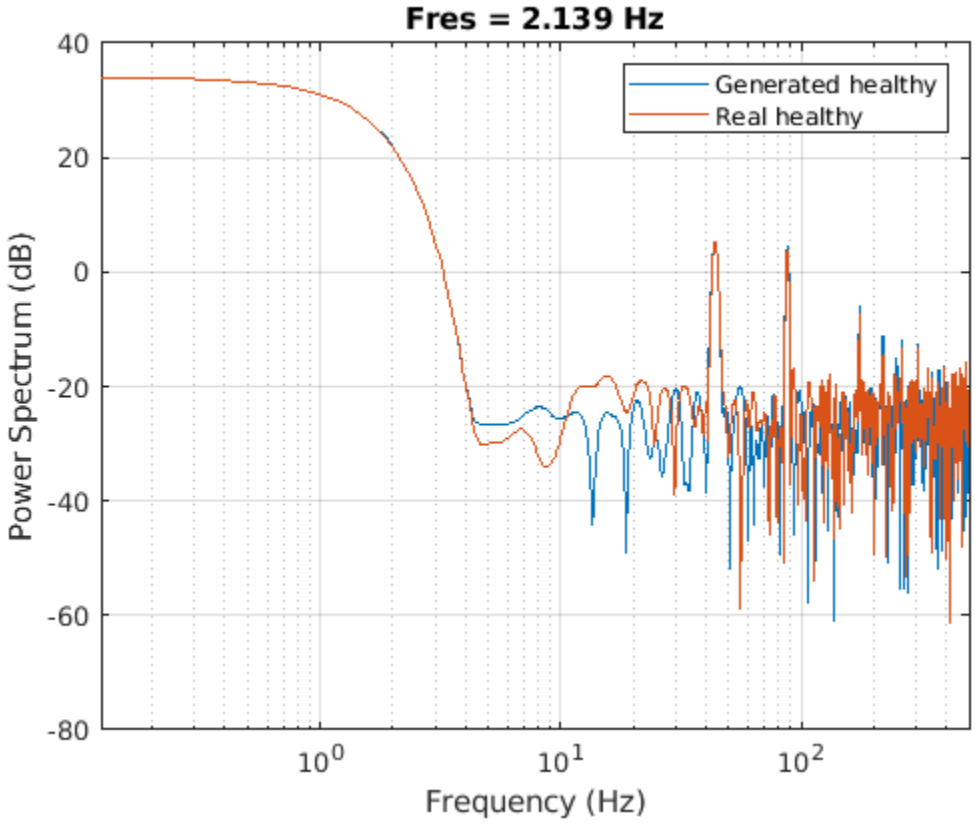


Case Study

Compare the spectral characteristics of real and generated signals. Due to the nondeterministic behavior of GPU training, if you train the CGAN model yourself, your results might differ from the results in this example.

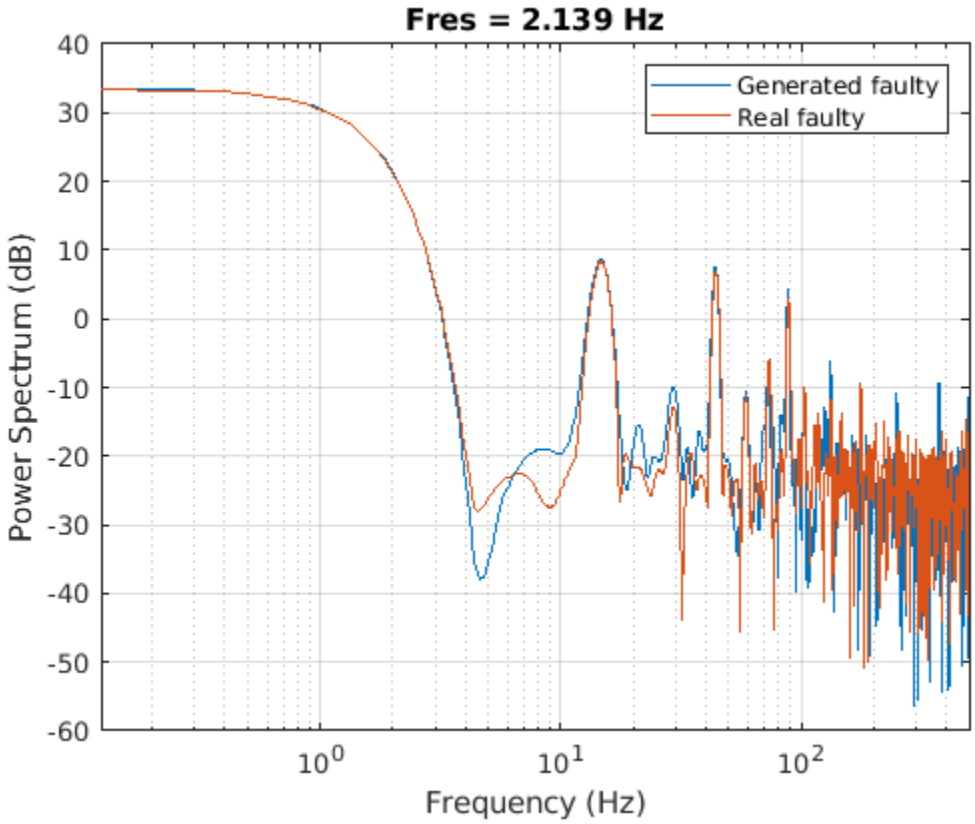
The pump motor speed is 950 rpm, or 15.833 Hz, and since the pump has three cylinders the flow is expected to have a fundamental at 3 times 15.833 Hz, or 47.5 Hz, and harmonics at multiples of 47.5 Hz. Plot the spectrum for one case of the real and generated healthy signals. From the plot, the generated healthy signal has relatively high power values at 47.5 Hz and 2 times 47.5 Hz, which is exactly the same as the real healthy signal.

```
Fs = 1000;
pspectrum([x(:,1) x(:,2006)],Fs)
set(gca,'XScale','log')
legend('Generated healthy','Real healthy')
```



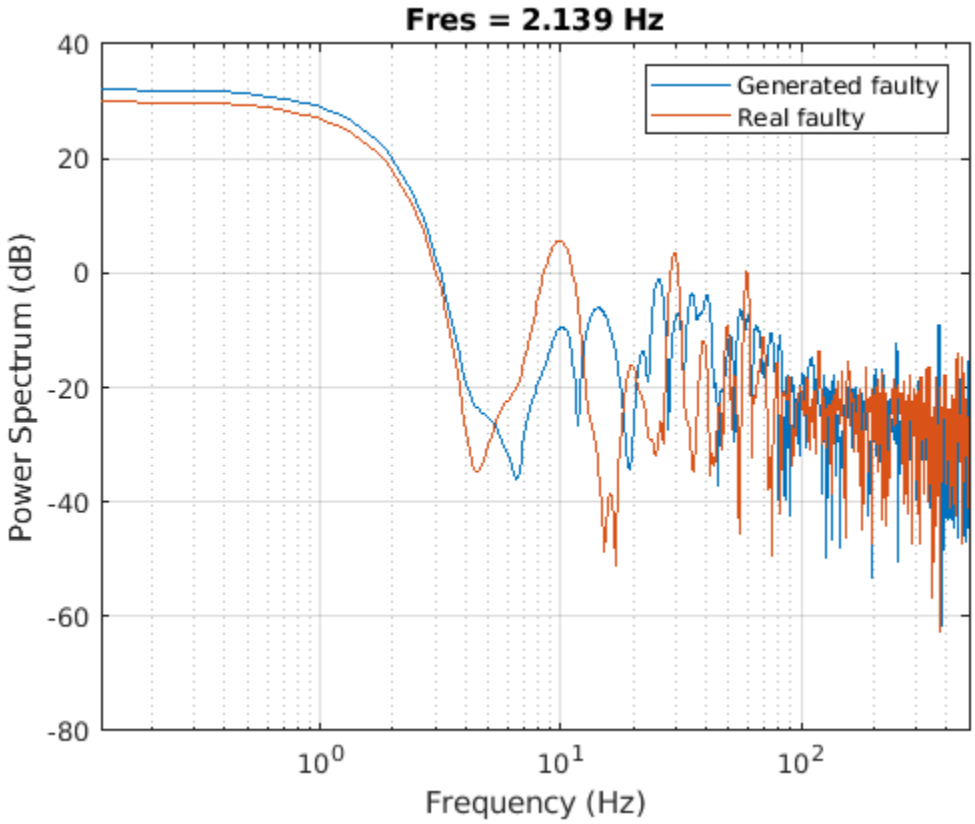
If faults exist, resonances will occur at the pump motor speed, 15.833 Hz, and its harmonics. Plot the spectra for one case of real and generated faulty signals. The generated signal has relatively high power values at around 15.833 Hz and its harmonics, which is similar to the real faulty signal.

```
pspectrum([x(:,1011) x(:,2100)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```



Plot spectra for another case of real and generated faulty signals. The spectral characteristics of the generated faulty signals do not match the theoretical analysis very well and are different from the real faulty signal. The CGAN can still be possibly improved by tuning the network structure or hyperparameters.

```
pspectrum([x(:,1001) x(:,2600)],Fs)
set(gca,'XScale','log')
legend('Generated faulty','Real faulty')
```



Computation Time

The Simulink simulation takes about 14 hours to generate 2000 pump flow signals. This duration can be reduced to about 1.7 hours with eight parallel workers if you have Parallel Computing Toolbox™.

The CGAN takes 1.5 hours to train and 70 seconds to generate the same amount of synthetic data with an NVIDIA Titan V GPU.

Classify Time Series Using Wavelet Analysis and Deep Learning

This example shows how to classify human electrocardiogram (ECG) signals using the continuous wavelet transform (CWT) and a deep convolutional neural network (CNN).

Training a deep CNN from scratch is computationally expensive and requires a large amount of training data. In various applications, a sufficient amount of training data is not available, and synthesizing new realistic training examples is not feasible. In these cases, leveraging existing neural networks that have been trained on large data sets for conceptually similar tasks is desirable. This leveraging of existing neural networks is called transfer learning. In this example we adapt two deep CNNs, GoogLeNet and SqueezeNet, pretrained for image recognition to classify ECG waveforms based on a time-frequency representation.

GoogLeNet and SqueezeNet are deep CNNs originally designed to classify images in 1000 categories. We reuse the network architecture of the CNN to classify ECG signals based on images from the CWT of the time series data. The data used in this example are publicly available from PhysioNet.

Data Description

In this example, you use ECG data obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). In total you use 162 ECG recordings from three PhysioNet databases: MIT-BIH Arrhythmia Database [3][7], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1][3]. More specifically, 96 recordings from persons with arrhythmia, 30 recordings from persons with congestive heart failure, and 36 recordings from persons with normal sinus rhythms. The goal is to train a classifier to distinguish between ARR, CHF, and NSR.

Download Data

The first step is to download the data from the GitHub repository. To download the data from the website, click `Code` and select `Download ZIP`. Save the file `physionet_ECG_data-main.zip` in a folder where you have write permission. The instructions for this example assume you have downloaded the file to your temporary directory, `tempdir`, in MATLAB. Modify the subsequent instructions for unzipping and loading the data if you choose to download the data in folder different from `tempdir`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir,'physionet_ECG_data-main.zip'),tempdir)
```

Unzipping creates the folder `physionet-ECG_data-main` in your temporary directory. This folder contains the text file `README.md` and `ECGData.zip`. The `ECGData.zip` file contains

- `ECGData.mat`
- `Modified_physionet_data.txt`
- `License.txt`

`ECGData.mat` holds the data used in this example. The text file, `Modified_physionet_data.txt`, is required by PhysioNet's copying policy and provides the source attributions for the data as well as a description of the preprocessing steps applied to each ECG recording.

Unzip `ECGData.zip` in `physionet-ECG_data-main`. Load the data file into your MATLAB workspace.

```
unzip(fullfile(tempdir,'physionet_ECG_data-main','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-main'))
load(fullfile(tempdir,'physionet_ECG_data-main','ECGData.mat'))
```

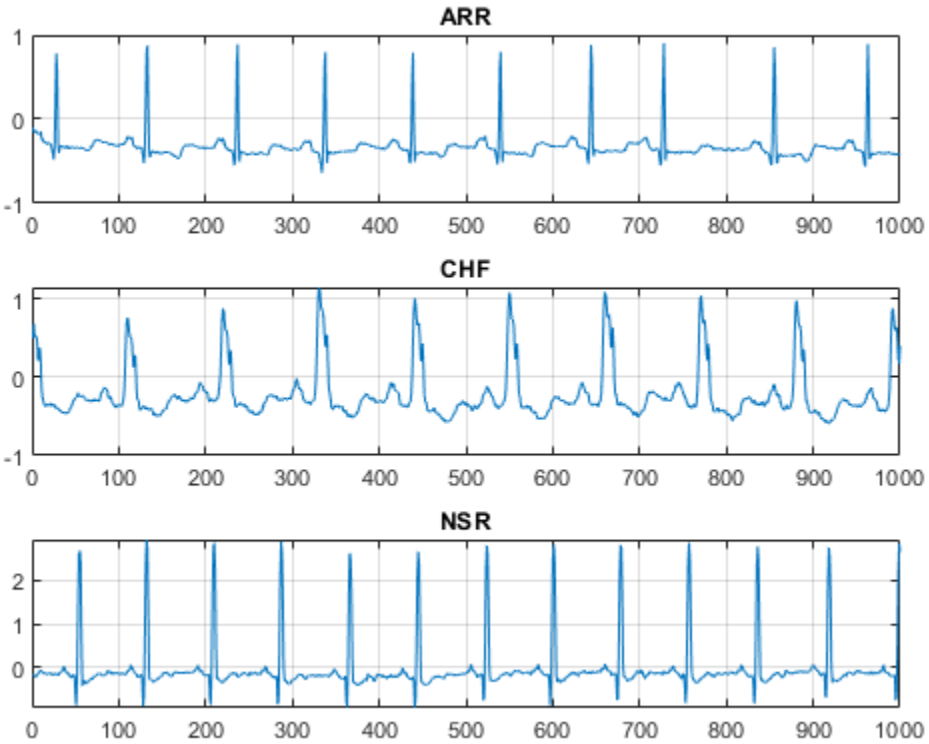
`ECGData` is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one for each row of `Data`. The three diagnostic categories are: `'ARR'`, `'CHF'`, and `'NSR'`.

To store the preprocessed data of each category, first create an ECG data directory `dataDir` inside `tempdir`. Then create three subdirectories in `'data'` named after each ECG category. The helper function `helperCreateECGDirectories` does this. `helperCreateECGDirectories` accepts `ECGData`, the name of an ECG data directory, and the name of a parent directory as input arguments. You can replace `tempdir` with another directory where you have write permission. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

```
parentDir = tempdir;
dataDir = 'data';
helperCreateECGDirectories(ECGData,parentDir,dataDir)
```

Plot a representative of each ECG category. The helper function `helperPlotReps` does this. `helperPlotReps` accepts `ECGData` as input. You can find the source code for this helper function in the Supporting Functions section at the end of this example.

```
helperPlotReps(ECGData)
```



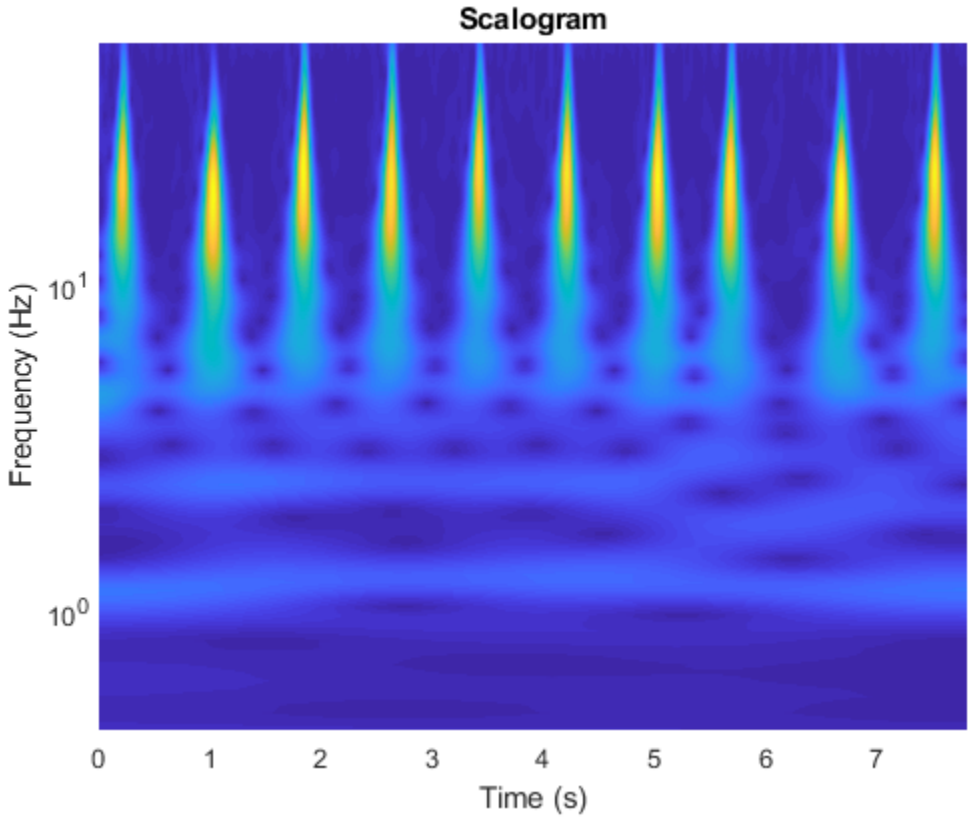
Create Time-Frequency Representations

After making the folders, create time-frequency representations of the ECG signals. These representations are called scalograms. A scalogram is the absolute value of the CWT coefficients of a signal.

To create the scalograms, precompute a CWT filter bank. Precomputing the CWT filter bank is the preferred method when obtaining the CWT of many signals using the same parameters.

Before generating the scalograms, examine one of them. Create a CWT filter bank using `cwtfilterbank` (Wavelet Toolbox) for a signal with 1000 samples. Use the filter bank to take the CWT of the first 1000 samples of the signal and obtain the scalogram from the coefficients.

```
Fs = 128;
fb = cwtfilterbank('SignalLength',1000,...
    'SamplingFrequency',Fs,...
    'VoicesPerOctave',12);
sig = ECGData.Data(1,1:1000);
[cfs,frq] = wt(fb,sig);
t = (0:999)/Fs;figure;pcolor(t,frq,abs(cfs))
set(gca,'yscale','log');shading interp;axis tight;
title('Scalogram');xlabel('Time (s)');ylabel('Frequency (Hz)')
```



Use the helper function `helperCreateRGBfromTF` to create the scalograms as RGB images and write them to the appropriate subdirectory in `dataDir`. The source code for this helper function is in the Supporting Functions section at the end of this example. To be compatible with the GoogLeNet architecture, each RGB image is an array of size 224-by-224-by-3.

```
helperCreateRGBfromTF(ECGData,parentDir,dataDir)
```

Divide into Training and Validation Data

Load the scalogram images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an ImageDatastore object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a CNN.

```
allImages = imageDatastore(fullfile(parentDir,dataDir),...
    'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

Randomly divide the images into two groups, one for training and the other for validation. Use 80% of the images for training, and the remainder for validation. For purposes of reproducibility, we set the random seed to the default value.

```
rng default
[imgsTrain,imgsValidation] = splitEachLabel(allImages,0.8,'randomized');
disp(['Number of training images: ',num2str(numel(imgsTrain.Files))]);
```

```
Number of training images: 130
```

```
disp(['Number of validation images: ',num2str(numel(imgsValidation.Files))]);
```

```
Number of validation images: 32
```

GoogLeNet

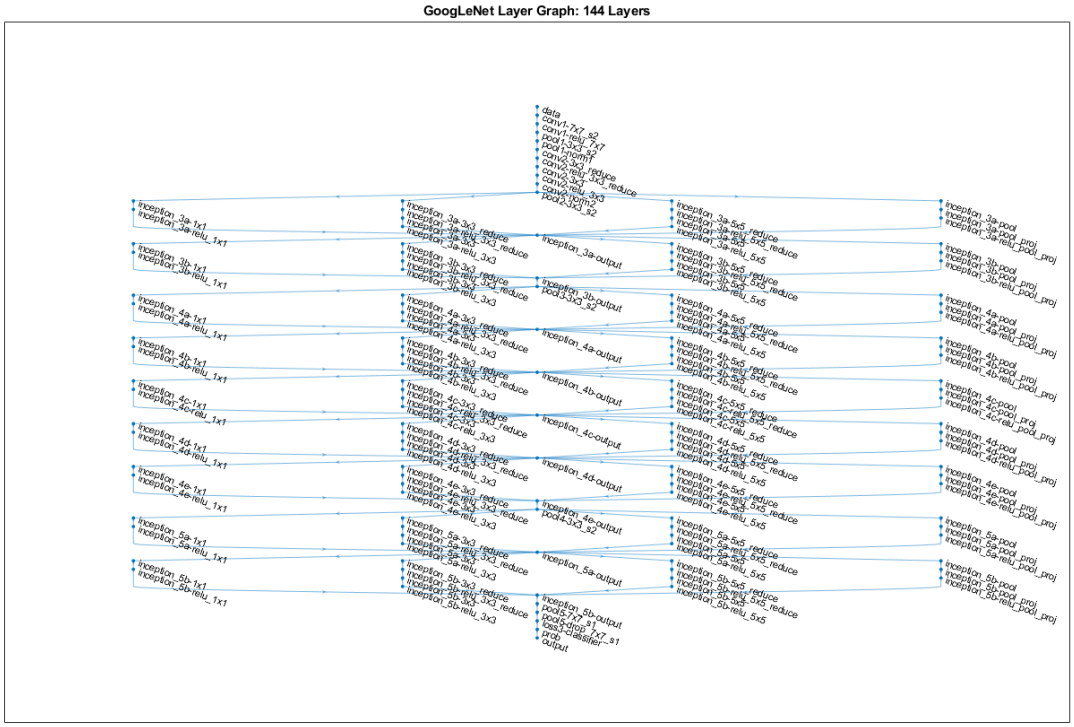
Load

Load the pretrained GoogLeNet neural network. If Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, the software provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

```
net = googlenet;
```

Extract and display the layer graph from the network.

```
lgraph = layerGraph(net);
numberOfLayers = numel(lgraph.Layers);
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8]);
plot(lgraph)
title(['GoogLeNet Layer Graph: ',num2str(numberOfLayers),' Layers']);
```



Inspect the first element of the network Layers property. Confirm that GoogLeNet requires RGB images of size 224-by-224-by-3.

```
net.Layers(1)
```

```
ans =
  ImageInputLayer with properties:

                 Name: 'data'
            InputSize: [224 224 3]

   Hyperparameters
    DataAugmentation: 'none'
       Normalization: 'zerocenter'
                Mean: [224×224×3 single]
```

Modify GoogLeNet Network Parameters

Each layer in the network architecture can be considered a filter. The earlier layers identify more common features of images, such as blobs, edges, and colors. Subsequent layers focus on more specific features in order to differentiate categories. GoogLeNet is pretrained to classify images into 1000 object categories. You must retrain GoogLeNet for our ECG classification problem.

To prevent overfitting, a dropout layer is used. A dropout layer randomly sets input elements to zero with a given probability. See `dropoutLayer` for more information. The default probability is 0.5.

Replace the final dropout layer in the network, `'pool5-drop_7x7_s1'`, with a dropout layer of probability 0.6.

```
newDropoutLayer = dropoutLayer(0.6,'Name','new_Dropout');
lgraph = replaceLayer(lgraph,'pool5-drop_7x7_s1',newDropoutLayer);
```

The convolutional layers of the network extract image features that the last learnable layer and final classification layer use to classify the input image. These two layers, `'loss3-classifier'` and `'output'` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain GoogLeNet to classify the RGB images, replace these two layers with new layers adapted to the data.

Replace the fully connected layer `'loss3-classifier'` with a new fully connected layer with the number of filters equal to the number of classes. To learn faster in the new layers than in the transferred layers, increase the learning rate factors of the fully connected layer.

```
numClasses = numel(categories(imgsTrain.Labels));
newConnectedLayer = fullyConnectedLayer(numClasses,'Name','new_fc',...
    'WeightLearnRateFactor',5,'BiasLearnRateFactor',5);
lgraph = replaceLayer(lgraph,'loss3-classifier',newConnectedLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'output',newClassLayer);
```

Set Training Options and Train GoogLeNet

Training a neural network is an iterative process that involves minimizing a loss function. To minimize the loss function, a gradient descent algorithm is used. In each iteration, the gradient of the loss function is evaluated and the descent algorithm weights are updated.

Training can be tuned by setting various options. `InitialLearnRate` specifies the initial step size in the direction of the negative gradient of the loss function. `MiniBatchSize` specifies how large of a subset of the training set to use in each iteration. One epoch is a full pass of the training algorithm over the entire training set. `MaxEpochs` specifies the maximum number of epochs to use for training. Choosing the right number of epochs is not a trivial task. Decreasing the number of epochs has the effect of underfitting the model, and increasing the number of epochs results in overfitting.

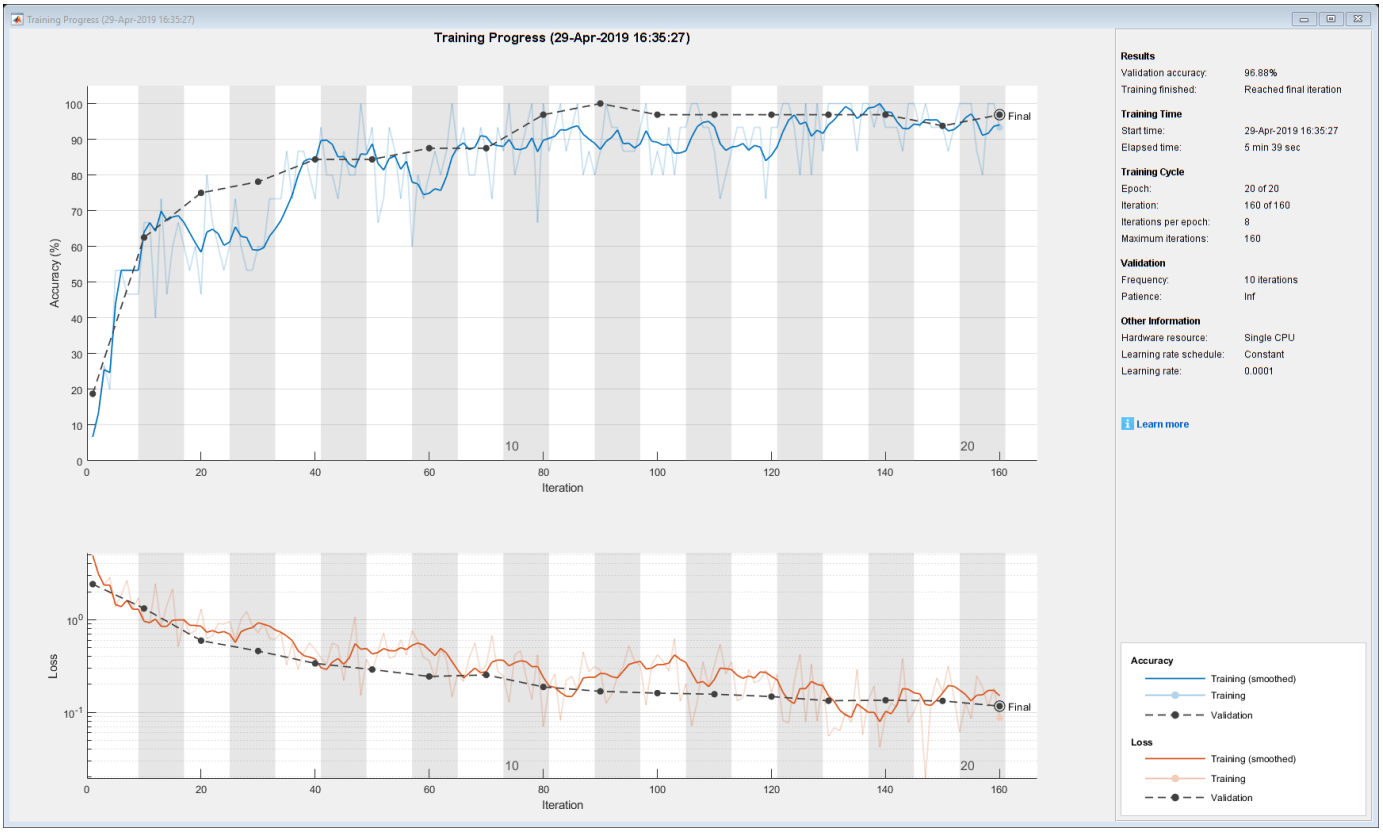
Use the `trainingOptions` function to specify the training options. Set `MiniBatchSize` to 10, `MaxEpochs` to 10, and `InitialLearnRate` to 0.0001. Visualize training progress by setting `Plots` to `training-progress`. Use the stochastic gradient descent with momentum optimizer. By default, training is done on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™. To see which GPUs are supported, see “GPU Support by Release” (Parallel Computing Toolbox). For purposes of reproducibility, set `ExecutionEnvironment` to `cpu` so that `trainNetwork` used the CPU. Set the random seed to the default value. Run times will be faster if you are able to use a GPU.

```
options = trainingOptions('sgdm',...
    'MiniBatchSize',15,...
    'MaxEpochs',20,...
    'InitialLearnRate',1e-4,...
    'ValidationData',imgsValidation,...
    'ValidationFrequency',10,...
    'Verbose',1,...
```

```
    'ExecutionEnvironment','cpu',...
    'Plots','training-progress');
rng default
```

Train the network. The training process usually takes 1-5 minutes on a desktop CPU. The command window displays training information during the run. Results include epoch number, iteration number, time elapsed, mini-batch accuracy, validation accuracy, and loss function value for the validation data.

```
trainedGN = trainNetwork(imgsTrain,lgraph,options);
```




```
Initializing input data normalization.
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:03 |        6.67% |       18.75% |       4.9207 |       2.4
|       2 |          10 |       00:00:23 |       66.67% |       62.50% |       0.9589 |       1.1
|       3 |          20 |       00:00:43 |       46.67% |       75.00% |       1.2973 |       0.5
|       4 |          30 |       00:01:04 |       60.00% |       78.13% |       0.7219 |       0.4
|       5 |          40 |       00:01:25 |       73.33% |       84.38% |       0.4750 |       0.3
|       7 |          50 |       00:01:46 |       93.33% |       84.38% |       0.2714 |       0.2
|       8 |          60 |       00:02:07 |       80.00% |       87.50% |       0.3617 |       0.2
|       9 |          70 |       00:02:29 |       86.67% |       87.50% |       0.3246 |       0.2
|      10 |          80 |       00:02:50 |      100.00% |       96.88% |       0.0701 |       0.1
|      12 |          90 |       00:03:11 |       86.67% |      100.00% |       0.2836 |       0.1
|      13 |         100 |       00:03:32 |       86.67% |       96.88% |       0.4160 |       0.1
|      14 |         110 |       00:03:53 |       86.67% |       96.88% |       0.3237 |       0.1
```

```
|      15 |         120 |       00:04:14 |       93.33% |       96.88% |       0.1646 |       0.1
|      17 |         130 |       00:04:35 |      100.00% |       96.88% |       0.0551 |       0.1
|      18 |         140 |       00:04:57 |       93.33% |       96.88% |       0.0927 |       0.1
|      19 |         150 |       00:05:18 |       93.33% |       93.75% |       0.1666 |       0.1
|      20 |         160 |       00:05:39 |       93.33% |       96.88% |       0.0873 |       0.1
|=========================================================================================================
```

Inspect the last layer of the trained network. Confirm the Classification Output layer includes the three classes.

```
trainedGN.Layers(end)
```

```
ans =
  ClassificationOutputLayer with properties:

            Name: 'new_classoutput'
         Classes: [ARR    CHF    NSR]
      OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Evaluate GoogLeNet Accuracy

Evaluate the network using the validation data.

```
[YPred,probs] = classify(trainedGN,imgsValidation);
accuracy = mean(YPred==imgsValidation.Labels);
disp(['GoogLeNet Accuracy: ',num2str(100*accuracy),'%'])
```

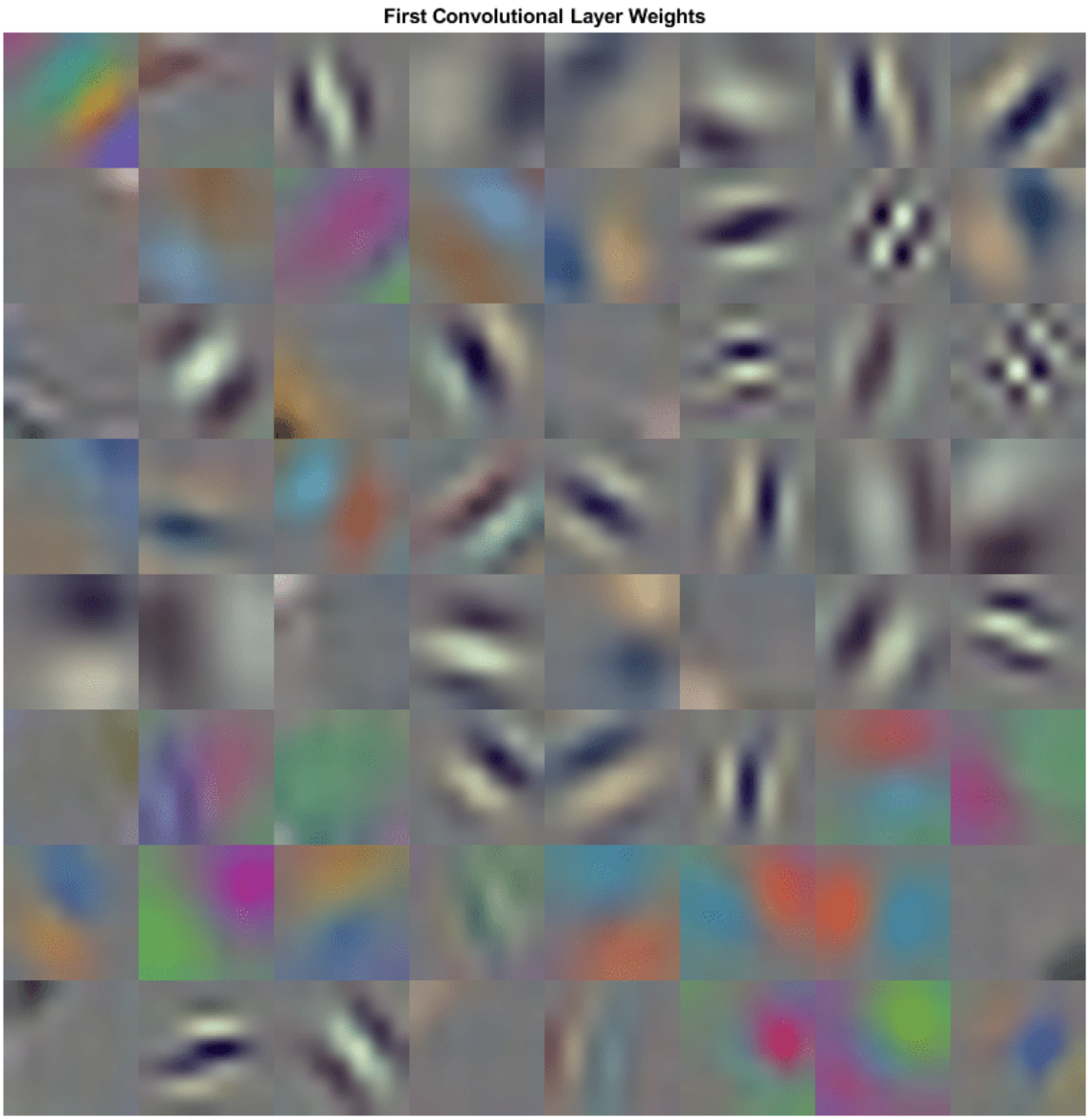
```
GoogLeNet Accuracy: 96.875%
```

The accuracy is identical to the validation accuracy reported on the training visualization figure. The scalograms were split into training and validation collections. Both collections were used to train GoogLeNet. The ideal way to evaluate the result of the training is to have the network classify data it has not seen. Since there is an insufficient amount of data to divide into training, validation, and testing, we treat the computed validation accuracy as the network accuracy.

Explore GoogLeNet Activations

Each layer of a CNN produces a response, or activation, to an input image. However, there are only a few layers within a CNN that are suitable for image feature extraction. The layers at the beginning of the network capture basic image features, such as edges and blobs. To see this, visualize the network filter weights from the first convolutional layer. There are 64 individual sets of weights in the first layer.

```
wghts = trainedGN.Layers(2).Weights;
wghts = rescale(wghts);
wghts = imresize(wghts,5);
figure
montage(wghts)
title('First Convolutional Layer Weights')
```

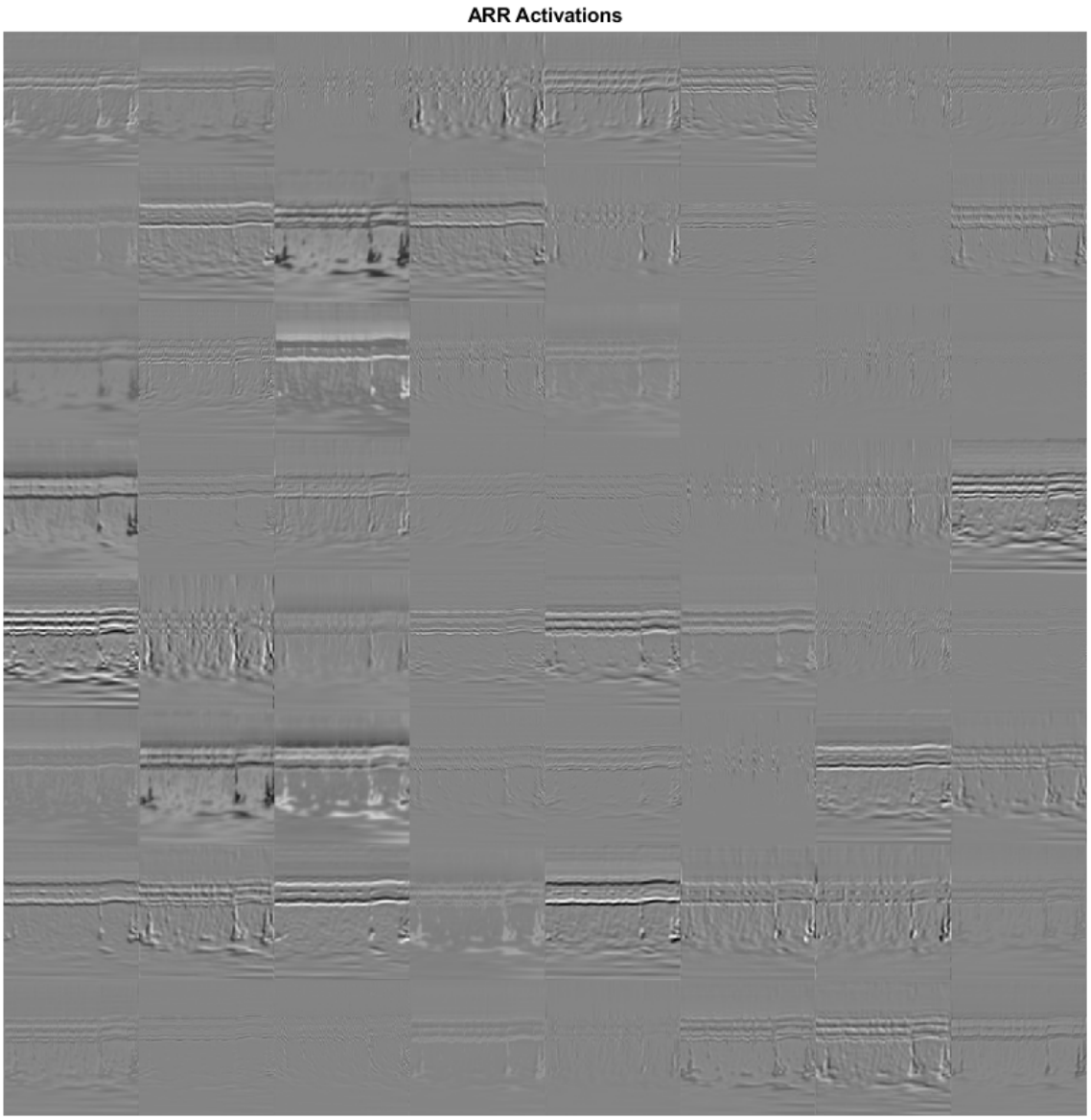


You can examine the activations and discover which features GoogLeNet learns by comparing areas of activation with the original image. For more information, see “Visualize Activations of a Convolutional Neural Network” on page 5-141 and “Visualize Features of a Convolutional Neural Network” on page 5-156.

Examine which areas in the convolutional layers activate on an image from the ARR class. Compare with the corresponding areas in the original image. Each layer of a convolutional neural network consists of many 2-D arrays called *channels*. Pass the image through the network and examine the output activations of the first convolutional layer, `'conv1-7x7_s2'`.

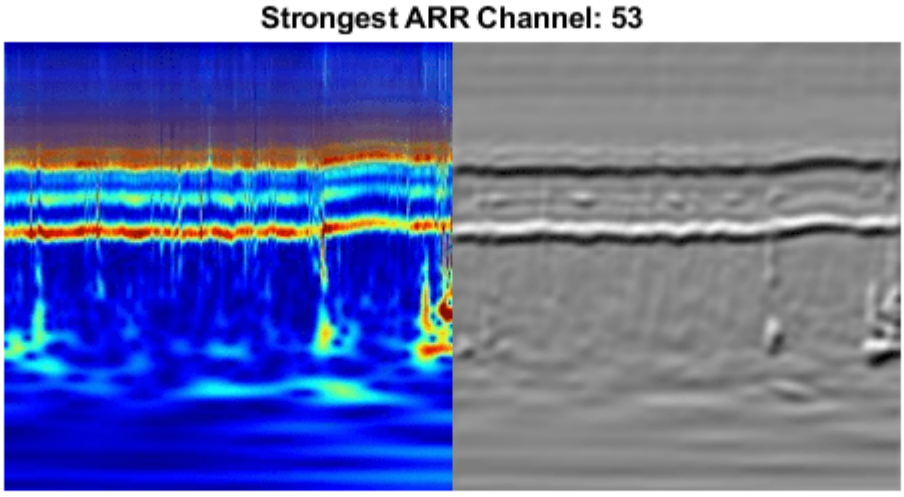
```
convLayer = 'conv1-7x7_s2';
```

```
imgClass = 'ARR';
imgName = 'ARR_10.jpg';
imarr = imread(fullfile(parentDir,dataDir,imgClass,imgName));

trainingFeaturesARR = activations(trainedGN,imarr,convLayer);
sz = size(trainingFeaturesARR);
trainingFeaturesARR = reshape(trainingFeaturesARR,[sz(1) sz(2) 1 sz(3)]);
figure
montage(rescale(trainingFeaturesARR),'Size',[8 8])
title([imgClass,' Activations'])
```



Find the strongest channel for this image. Compare the strongest channel with the original image.

```
imgSize = size(imarr);
imgSize = imgSize(1:2);
[~,maxValueIndex] = max(max(max(trainingFeaturesARR)));
arrMax = trainingFeaturesARR(:,:,:,maxValueIndex);
arrMax = rescale(arrMax);
arrMax = imresize(arrMax,imgSize);
figure;
imshowpair(imarr,arrMax,'montage')
title(['Strongest ',imgClass,' Channel: ',num2str(maxValueIndex)])
```



SqueezeNet

SqueezeNet is a deep CNN whose architecture supports images of size 227-by-227-by-3. Even though the image dimensions are different for GoogLeNet, you do not have to generate new RGB images at the SqueezeNet dimensions. You can use the original RGB images.

Load

Load the pretrained SqueezeNet neural network. If Deep Learning Toolbox™ Model *for SqueezeNet Network* support package is not installed, the software provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

```
sqz = squeezenet;
```

Extract the layer graph from the network. Confirm SqueezeNet has fewer layers than GoogLeNet. Also confirm that SqueezeNet is configured for images of size 227-by-227-by-3

```
lgraphSqz = layerGraph(sqz);
disp(['Number of Layers: ',num2str(numel(lgraphSqz.Layers))])
```

```
Number of Layers: 68
```

```
disp(lgraphSqz.Layers(1).InputSize)
```

```
   227   227     3
```

Modify SqueezeNet Network Parameters

To retrain SqueezeNet to classify new images, make changes similar to those made for GoogLeNet.

Inspect the last six network layers.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
  6x1 Layer array with layers:

     1   'drop9'                               Dropout                 50% dropout
     2   'conv10'                              Convolution             1000 1x1x512 convolutions wi
     3   'relu_conv10'                         ReLU                    ReLU
     4   'pool10'                              Average Pooling         14x14 average pooling with s
     5   'prob'                                Softmax                 softmax
     6   'ClassificationLayer_predictions'     Classification Output   crossentropyex with 'tench'
```

Replace the `'drop9'` layer, the last dropout layer in the network, with a dropout layer of probability 0.6.

```
tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);
```

Unlike GoogLeNet, the last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, `'conv10'`, and not a fully connected layer. Replace the `'conv10'` layer with a new convolutional layer with the number of filters equal to the number of classes. As was done with GoogLeNet, increase the learning rate factors of the new layer.

```
numClasses = numel(categories(imgsTrain.Labels));
tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);
```

Replace the classification layer with a new one without class labels.

```
tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Inspect the last six layers of the network. Confirm the dropout, convolutional, and output layers have been changed.

```
lgraphSqz.Layers(63:68)
```

```
ans =
  6x1 Layer array with layers:

     1   'new_dropout'       Dropout                 60% dropout
     2   'new_conv'          Convolution             3 1x1 convolutions with stride [1  1] and pa
     3   'relu_conv10'       ReLU                    ReLU
     4   'pool10'            Average Pooling         14x14 average pooling with stride [1  1] and
     5   'prob'              Softmax                 softmax
     6   'new_classoutput'   Classification Output   crossentropyex
```

Prepare RGB Data for SqueezeNet

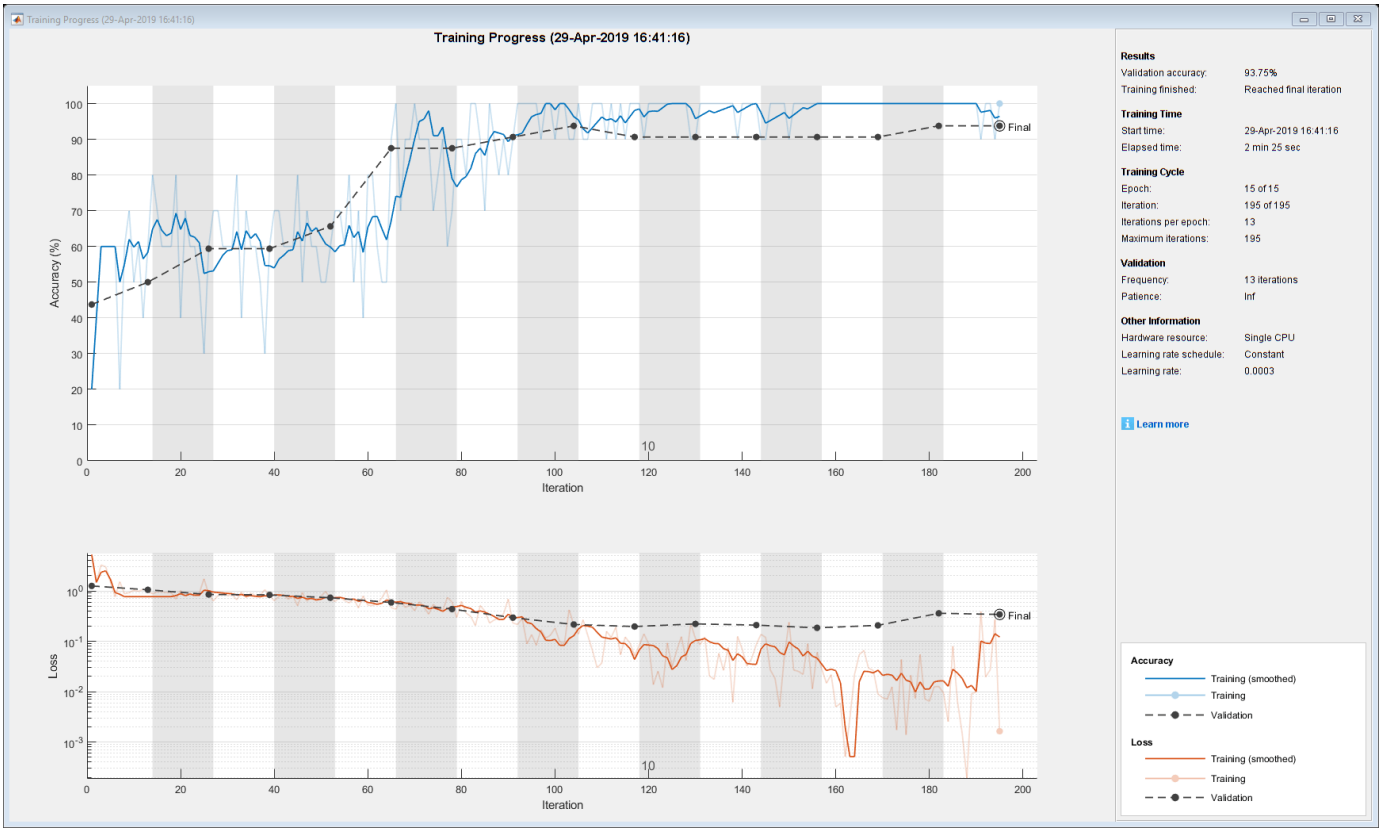
The RGB images have dimensions appropriate for the GoogLeNet architecture. Create augmented image datastores that automatically resize the existing RGB images for the SqueezeNet architecture. For more information, see `augmentedImageDatastore`.

```
augimgsTrain = augmentedImageDatastore([227 227],imgsTrain);
augimgsValidation = augmentedImageDatastore([227 227],imgsValidation);
```

Set Training Options and Train SqueezeNet

Create a new set of training options to use with SqueezeNet. Set the random seed to the default value and train the network. The training process usually takes 1-5 minutes on a desktop CPU.

```
ilr = 3e-4;
miniBatchSize = 10;
maxEpochs = 15;
valFreq = floor(numel(augimgsTrain.Files)/miniBatchSize);
opts = trainingOptions('sgdm',...
    'MiniBatchSize',miniBatchSize,...
    'MaxEpochs',maxEpochs,...
    'InitialLearnRate',ilr,...
    'ValidationData',augimgsValidation,...
    'ValidationFrequency',valFreq,...
    'Verbose',1,...
    'ExecutionEnvironment','cpu',...
    'Plots','training-progress');

rng default
trainedSN = trainNetwork(augimgsTrain,lgraphSqz,opts);
```




```
Initializing input data normalization.
|=========================================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)  |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================|
|       1 |           1 |       00:00:01 |       20.00% |       43.75% |       5.2508 |       1.2
|       1 |          13 |       00:00:11 |       60.00% |       50.00% |       0.9912 |       1.0
|       2 |          26 |       00:00:20 |       60.00% |       59.38% |       0.8554 |       0.8
|       3 |          39 |       00:00:30 |       60.00% |       59.38% |       0.8120 |       0.8
|       4 |          50 |       00:00:38 |       50.00% |              |       0.7885 |
|       4 |          52 |       00:00:40 |       60.00% |       65.63% |       0.7091 |       0.7
|       5 |          65 |       00:00:49 |       90.00% |       87.50% |       0.4639 |       0.5
|       6 |          78 |       00:00:59 |       70.00% |       87.50% |       0.6021 |       0.4
|       7 |          91 |       00:01:08 |       90.00% |       90.63% |       0.2307 |       0.2
|       8 |         100 |       00:01:15 |       90.00% |              |       0.1827 |
|       8 |         104 |       00:01:18 |       90.00% |       93.75% |       0.2139 |       0.2
|       9 |         117 |       00:01:28 |      100.00% |       90.63% |       0.0521 |       0.1
|      10 |         130 |       00:01:38 |       90.00% |       90.63% |       0.1134 |       0.2
|      11 |         143 |       00:01:47 |      100.00% |       90.63% |       0.0855 |       0.2
|      12 |         150 |       00:01:52 |       90.00% |              |       0.2394 |
|      12 |         156 |       00:01:57 |      100.00% |       90.63% |       0.0606 |       0.1
|      13 |         169 |       00:02:06 |      100.00% |       90.63% |       0.0090 |       0.2
|      14 |         182 |       00:02:16 |      100.00% |       93.75% |       0.0127 |       0.1
|      15 |         195 |       00:02:25 |      100.00% |       93.75% |       0.0016 |       0.1
|=========================================================================================================|
```

Inspect the last layer of the network. Confirm the Classification Output layer includes the three classes.

```
trainedSN.Layers(end)

ans =
  ClassificationOutputLayer with properties:

            Name: 'new_classoutput'
         Classes: [ARR    CHF    NSR]
      OutputSize: 3

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Evaluate SqueezeNet Accuracy

Evaluate the network using the validation data.

```
[YPred,probs] = classify(trainedSN,augimgsValidation);
accuracy = mean(YPred==imgsValidation.Labels);
disp(['SqueezeNet Accuracy: ',num2str(100*accuracy),'%'])
```

```
SqueezeNet Accuracy: 93.75%
```

Conclusion

This example shows how to use transfer learning and continuous wavelet analysis to classify three classes of ECG signals by leveraging the pretrained CNNs GoogLeNet and SqueezeNet. Wavelet-based time-frequency representations of ECG signals are used to create scalograms. RGB images of the scalograms are generated. The images are used to fine-tune both deep CNNs. Activations of different network layers were also explored.

This example illustrates one possible workflow you can use for classifying signals using pretrained CNN models. Other workflows are possible. “Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox) and “Deploy Signal Classifier Using Wavelets and Deep Learning on Raspberry Pi” (Wavelet Toolbox) show how to deploy code onto hardware for signal classification. GoogLeNet and SqueezeNet are models pretrained on a subset of the ImageNet database [10], which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [8]. The ImageNet collection contains images of real-world objects such as fish, birds, appliances, and fungi. Scalograms fall outside the class of real-world objects. In order to fit into the GoogLeNet and SqueezeNet architecture, the scalograms also underwent data reduction. Instead of fine-tuning pretrained CNNs to distinguish different classes of scalograms, training a CNN from scratch at the original scalogram dimensions is an option.

References

1. Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661–670.
2. Engin, M. "ECG beat classification using neuro-fuzzy network." *Pattern Recognition Letters*. Vol. 25, Number 15, 2004, pp.1715–1722.
3. Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit,and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23: e215–e220. [Circulation Electronic Pages; `http://circ.ahajournals.org/content/101/23/e215.full`]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.

4 Leonarduzzi, R. F., G. Schlotthauer, and M. E. Torres. "Wavelet leader based multifractal analysis of heart rate variability during myocardial ischaemia." In *Engineering in Medicine and Biology Society (EMBC), Annual International Conference of the IEEE*, 110–113. Buenos Aires, Argentina: IEEE, 2010.

5 Li, T., and M. Zhou. "ECG classification using wavelet packet entropy and random forests." *Entropy*. Vol. 18, Number 8, 2016, p.285.

6 Maharaj, E. A., and A. M. Alonso. "Discriminant analysis of multivariate time series: Application to diagnosis based on ECG signals." *Computational Statistics and Data Analysis*. Vol. 70, 2014, pp. 67–87.

7 Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20. Number 3, May-June 2001, pp. 45–50. (PMID: 11446209)

8 Russakovsky, O., J. Deng, and H. Su et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*. Vol. 115, Number 3, 2015, pp. 211–252.

9 Zhao, Q., and L. Zhang. "ECG feature extraction and classification using wavelet transform and support vector machines." In *IEEE International Conference on Neural Networks and Brain*, 1089–1092. Beijing, China: IEEE, 2005.

10 *ImageNet*. `http://www.image-net.org`

Supporting Functions

helperCreateECGDataDirectories creates a data directory inside a parent directory, then creates three subdirectories inside the data directory. The subdirectories are named after each class of ECG signal found in `ECGData`.

```
function helperCreateECGDirectories(ECGData,parentFolder,dataFolder)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

rootFolder = parentFolder;
localFolder = dataFolder;
mkdir(fullfile(rootFolder,localFolder))

folderLabels = unique(ECGData.Labels);
for i = 1:numel(folderLabels)
    mkdir(fullfile(rootFolder,localFolder,char(folderLabels(i))));
end
end
```

helperPlotReps plots the first thousand samples of a representative of each class of ECG signal found in `ECGData`.

```
function helperPlotReps(ECGData)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

folderLabels = unique(ECGData.Labels);

for k=1:3
    ecgType = folderLabels{k};
    ind = find(ismember(ECGData.Labels,ecgType));
    subplot(3,1,k)
    plot(ECGData.Data(ind(1),1:1000));
    grid on
```

```
    title(ecgType)
end
end
```

helperCreateRGBfromTF uses `cwtfilterbank` (Wavelet Toolbox) to obtain the continuous wavelet transform of the ECG signals and generates the scalograms from the wavelet coefficients. The helper function resizes the scalograms and writes them to disk as jpeg images.

```
function helperCreateRGBfromTF(ECGData,parentFolder,childFolder)
% This function is only intended to support the ECGAndDeepLearningExample.
% It may change or be removed in a future release.

imageRoot = fullfile(parentFolder,childFolder);

data = ECGData.Data;
labels = ECGData.Labels;

[~,signalLength] = size(data);

fb = cwtfilterbank('SignalLength',signalLength,'VoicesPerOctave',12);
r = size(data,1);

for ii = 1:r
    cfs = abs(fb.wt(data(ii,:)));
    im = ind2rgb(im2uint8(rescale(cfs)),jet(128));

    imgLoc = fullfile(imageRoot,char(labels(ii)));
    imFileName = strcat(char(labels(ii)),'_',num2str(ii),'.jpg');
    imwrite(imresize(im,[224 224]),fullfile(imgLoc,imFileName));
end
end
```

See Also

`cwtfilterbank` | `googlenet` | `squeezenet` | `trainNetwork` | `trainingOptions` | `imageDatastore` | `augmentedImageDatastore`

Related Examples

- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Pretrained Deep Neural Networks” on page 1-8
- “Deep Learning in MATLAB” on page 1-2

Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning

This example shows how to generate and deploy a CUDA® executable that classifies human electrocardiogram (ECG) signals using features extracted by the continuous wavelet transform (CWT) and a pretrained convolutional neural network (CNN).

SqueezeNet is a deep CNN originally designed to classify images in 1000 categories. We reuse the network architecture of the CNN to classify ECG signals based on their scalograms. A scalogram is the absolute value of the CWT of the signal. After training SqueezeNet to classify ECG signals, you create a CUDA executable that generates a scalogram of an ECG signal and then uses the CNN to classify the signal. The executable and CNN are both deployed to the NVIDIA hardware.

This example uses the same data as used in “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox). In that example, transfer learning with GoogLeNet and SqueezeNet are used to classify ECG waveforms into one of three categories. The description of the data and how to obtain it are repeated here for convenience.

ECG Data Description and Download

The ECG data is obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). In total there are 162 ECG recordings from three PhysioNet databases: MIT-BIH Arrhythmia Database [2][3], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1] [3]. More specifically, 96 recordings from persons with arrhythmia, 30 recordings from persons with congestive heart failure, and 36 recordings from persons with normal sinus rhythms. The goal is to train a model to distinguish between ARR, CHF, and NSR.

You can obtain this data from the MathWorks GitHub repository. To download the data from the website, click `Code` and select `Download ZIP`. Save the file `physionet_ECG_data-main.zip` in a folder where you have write permission. The instructions for this example assume you have downloaded the file to your temporary directory, `tempdir`, in MATLAB. Modify the subsequent instructions for unzipping and loading the data if you choose to download the data in a folder different from `tempdir`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir,'physionet_ECG_data-main.zip'),tempdir)
```

Unzipping creates the folder `physionet-ECG_data-main` in your temporary directory. This folder contains the text file `README.md` and `ECGData.zip`. The `ECGData.zip` file contains:

- `ECGData.mat`
- `Modified_physionet_data.txt`
- `License.txt`

`ECGData.mat` holds the data used in this example. The text file `Modified_physionet_data.txt` is required by PhysioNet's copying policy and provides the source attributions for the data as well as a description of the preprocessing steps applied to each ECG recording.

Unzip `ECGData.zip` in `physionet-ECG_data-main`. Load the data file into your MATLAB workspace.

```
unzip(fullfile(tempdir,'physionet_ECG_data-main','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-main'))
load(fullfile(tempdir,'physionet_ECG_data-main','ECGData.mat'))
```

ECGData is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one label for each row of `Data`. The three diagnostic categories are: `'ARR'`, `'CHF'`, and `'NSR'`.

Feature Extraction

After downloading the data, you must generate scalograms of the signals. The scalograms are the "input" images to the CNN.

To store the scalograms of each category, first create an ECG data directory `'data'` inside `tempdir`. Then create three subdirectories in `'data'` named after each ECG category. The helper function `helperCreateECGDirectories` does this for you. `helperCreateECGDirectories` accepts `ECGData`, the name of an ECG data directory, and the name of a parent directory as input arguments. You can replace `tempdir` with another directory where you have write permission. You can find the source code for this helper function in the Supporting Functions on page 12-0 section at the end of this example.

```
parentDir = tempdir;
dataDir = 'data';
helperCreateECGDirectories(ECGData,parentDir,dataDir)
```

After making the folders, create scalograms of the ECG signals as RGB images and write them to the appropriate subdirectory in `dataDir`. To create the scalograms, first precompute a CWT filter bank. Precomputing the filter bank is the preferred method when obtaining the CWT of many signals using the same parameters. The helper function `helperCreateRGBfromTF` does this.The source code for this helper function is in the Supporting Functions on page 12-0 section at the end of this example. To be compatible with the SqueezeNet architecture, each RGB image is an array of size 227-by-227-by-3.

```
helperCreateRGBfromTF(ECGData,parentDir,dataDir)
```

Divide Data Set into Training and Validation Data

Load the scalogram images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images when training a CNN.

```
allImages = imageDatastore(fullfile(tempdir,dataDir),...
    'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

Randomly divide the images into two groups, one for training and the other for validation. Use 80% of the images for training and the remainder for validation. For purposes of reproducibility, we set the random seed to the default value.

```
rng default
[imgsTrain,imgsValidation] = splitEachLabel(allImages,0.8,'randomized');
disp(['Number of training images: ',num2str(numel(imgsTrain.Files))]);
```

```
Number of training images: 130
```

```
disp(['Number of validation images: ',num2str(numel(imgsValidation.Files))]);
```

```
Number of validation images: 32
```

SqueezeNet

SqueezeNet is a pretrained CNN that can classify images into 1000 categories. You need to retrain SqueezeNet for our ECG classification problem. Prior to retraining, you modify several network layers and set various training options. After retraining is complete, you save the CNN in a `.mat` file. The CUDA executable will use the `.mat` file.

Specify an experiment trial index and a results directory. If necessary, create the directory.

```
trial = 1;
ResultDir = 'results';
if ~exist(ResultDir,'dir')
    mkdir(ResultDir)
end
MatFile = fullfile(ResultDir,sprintf('SqueezeNet_Trial%d.mat',trial));
```

Load SqeezeNet. Extract the layer graph and inspect the last five layers.

```
sqz = squeezenet;
lgraph = layerGraph(sqz);
lgraph.Layers(end-4:end)
```

```
ans =
  5×1 Layer array with layers:

     1   'conv10'                            Convolution              1000 1×1×512 convolutions w
     2   'relu_conv10'                       ReLU                     ReLU
     3   'pool10'                            Global Average Pooling   Global average pooling
     4   'prob'                              Softmax                  softmax
     5   'ClassificationLayer_predictions'   Classification Output    crossentropyex with 'tench
```

To retrain SqueezeNet to classify the three classes of ECG signals, replace the `'conv10'` layer with a new convolutional layer with the number of filters equal to the number of ECG classes. Replace the classification layer with a new one without class labels.

```
numClasses = numel(categories(imgsTrain.Labels));
new_conv10_WeightLearnRateFactor = 1;
new_conv10_BiasLearnRateFactor = 1;
newConvLayer = convolution2dLayer(1,numClasses,...
        'Name','new_conv10',...
        'WeightLearnRateFactor',new_conv10_WeightLearnRateFactor,...
        'BiasLearnRateFactor',new_conv10_BiasLearnRateFactor);
lgraph = replaceLayer(lgraph,'conv10',newConvLayer);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassLayer);
lgraph.Layers(end-4:end)
```

```
ans =
  5×1 Layer array with layers:

     1   'new_conv10'        Convolution              3 1×1 convolutions with stride [1  1] and p
     2   'relu_conv10'       ReLU                     ReLU
     3   'pool10'            Global Average Pooling   Global average pooling
     4   'prob'              Softmax                  softmax
     5   'new_classoutput'   Classification Output    crossentropyex
```

Create a set of training options to use with SqueezeNet.

```
OptimSolver = 'sgdm';
MiniBatchSize = 15;
MaxEpochs = 20;
InitialLearnRate = 1e-4;
Momentum = 0.9;
ExecutionEnvironment = 'cpu';

options = trainingOptions(OptimSolver,...
    'MiniBatchSize',MiniBatchSize,...
    'MaxEpochs',MaxEpochs,...
    'InitialLearnRate',InitialLearnRate,...
    'ValidationData',imgsValidation,...
    'ValidationFrequency',10,...
    'ExecutionEnvironment',ExecutionEnvironment,...
    'Momentum',Momentum);
```

Save all the parameters in a structure. The trained network and structure will be later saved in a `.mat` file.

```
TrialParameter.new_conv10_WeightLearnRateFactor = new_conv10_WeightLearnRateFactor;
TrialParameter.new_conv10_BiasLearnRateFactor = new_conv10_BiasLearnRateFactor;
TrialParameter.OptimSolver = OptimSolver;
TrialParameter.MiniBatchSize = MiniBatchSize;
TrialParameter.MaxEpochs = MaxEpochs;
TrialParameter.InitialLearnRate = InitialLearnRate;
TrialParameter.Momentum = Momentum;
TrialParameter.ExecutionEnvironment = ExecutionEnvironment;
```

Set the random seed to the default value and train the network. Save the trained network, trial parameters, training run time, and image datastore containing the validation images. The training process usually takes 1-5 minutes on a desktop CPU. If you want to use a trained CNN from a previous trial, set `trial` to the index number of that trial and `LoadModel` to `true`.

```
LoadModel = false;
if ~LoadModel
    rng default
    tic;
    trainedModel = trainNetwork(imgsTrain,lgraph,options);
    trainingTime = toc;
    fprintf('Total training time: %.2e sec\n',trainingTime);
    save(MatFile,'TrialParameter','trainedModel','trainingTime','imgsValidation');
else
    disp('Load ML model from the file')
    load(MatFile,'trainedModel','imgsValidation');
end
```

```
Initializing input data normalization.
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:02 |       26.67% |       25.00% |       4.1769 |       2.9
|       2 |          10 |       00:00:12 |       73.33% |       59.38% |       0.9877 |       1.1
|       3 |          20 |       00:00:21 |       60.00% |       56.25% |       0.9164 |       0.9
|       4 |          30 |       00:00:31 |       86.67% |       68.75% |       0.6698 |       0.7
|       5 |          40 |       00:00:40 |       66.67% |       68.75% |       0.9053 |       0.7
```

```
|       7 |          50 |       00:00:50 |       80.00% |       78.13% |       0.5422 |       0.6
|       8 |          60 |       00:00:59 |      100.00% |       81.25% |       0.4187 |       0.6
|       9 |          70 |       00:01:08 |       93.33% |       84.38% |       0.3561 |       0.5
|      10 |          80 |       00:01:18 |       73.33% |       84.38% |       0.5141 |       0.4
|      12 |          90 |       00:01:27 |       86.67% |       84.38% |       0.4220 |       0.4
|      13 |         100 |       00:01:36 |       93.33% |       90.63% |       0.1923 |       0.3
|      14 |         110 |       00:01:46 |      100.00% |       90.63% |       0.1472 |       0.3
|      15 |         120 |       00:01:55 |      100.00% |       93.75% |       0.0791 |       0.2
|      17 |         130 |       00:02:04 |       86.67% |       93.75% |       0.2486 |       0.2
|      18 |         140 |       00:02:14 |      100.00% |       93.75% |       0.0386 |       0.2
|      19 |         150 |       00:02:23 |      100.00% |       93.75% |       0.0487 |       0.2
|      20 |         160 |       00:02:32 |      100.00% |       93.75% |       0.0224 |       0.2
|=========================================================================================================
```

```
Total training time: 1.61e+02 sec
```

Save only the trained network in a separate `.mat` file. This file will be used by the CUDA executable.

```
ModelFile = fullfile(ResultDir,sprintf('SqueezeNet_Trial%d.mat',trial));
OutMatFile = fullfile('ecg_model.mat');

data = load(ModelFile,'trainedModel');
net = data.trainedModel;
save(OutMatFile,'net');
```

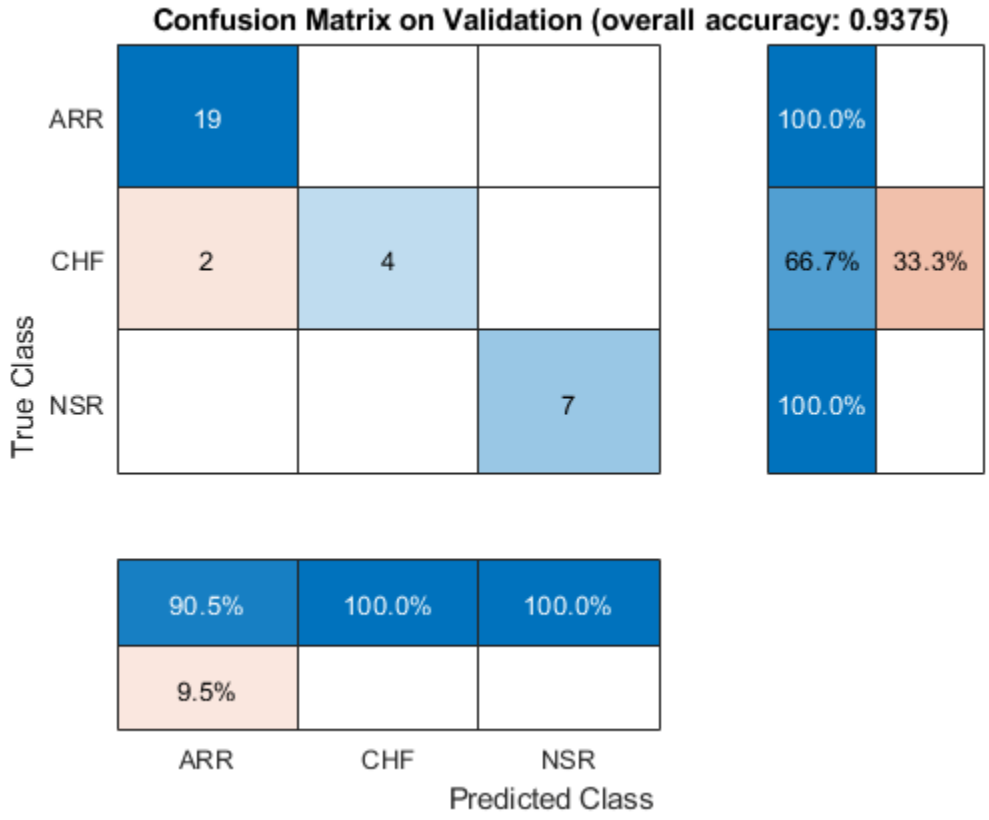
Use the trained network to predict the classes for the validation set.

```
[YPred, probs] = classify(trainedModel,imgsValidation);
accuracy = mean(YPred==imgsValidation.Labels)
```

```
accuracy = 0.9375
```

Summarize the performance of the trained network on the validation set with a confusion chart. Display the precision and recall for each class by using column and row summaries. Save the figure. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure
confusionMat = confusionmat(imgsValidation.Labels,YPred);
confusionchart(imgsValidation.Labels,YPred, ...
    'Title',sprintf('Confusion Matrix on Validation (overall accuracy: %.4f)',accuracy),...
    'ColumnSummary','column-normalized','RowSummary','row-normalized');
```




```
AccFigFile = fullfile(ResultDir,sprintf('SqueezeNet_ValidationAccuracy_Trial%d.fig',trial));
saveas(gcf,AccFigFile);
```

Display the size of the trained network.

```
info = whos('trainedModel');
ModelMemSize = info.bytes/1024;
fprintf('Trained network size: %g kB\n',ModelMemSize)
```

```
Trained network size: 2981.55 kB
```

Determine the average time it takes the network to classify an image.

```
NumTestForPredTime = 20;
TrialParameter.NumTestForPredTime = NumTestForPredTime;

fprintf('Test prediction time (number of tests: %d)... ',NumTestForPredTime)
```

```
Test prediction time (number of tests: 20)...
```

```
imageSize = trainedModel.Layers(1).InputSize;
PredTime = zeros(NumTestForPredTime,1);
for i = 1:NumTestForPredTime
    x = randn(imageSize);
    tic;
    [YPred, probs] = classify(trainedModel,x,'ExecutionEnvironment',ExecutionEnvironment);
    PredTime(i) = toc;
end
```

```
AvgPredTimePerImage = mean(PredTime);
fprintf('Average prediction time (execution environment: %s): %.2e sec \n',...
    ExecutionEnvironment,AvgPredTimePerImage);
```

```
Average prediction time (execution environment: cpu): 2.94e-02 sec
```

Save the results.

```
if ~LoadModel
    save(MatFile,'accuracy','confusionMat','PredTime','ModelMemSize', ...
        'AvgPredTimePerImage','-append')
end
```

GPU Code Generation — Define Functions

The scalogram of a signal is the input "image" to a deep CNN. Create a function, `cwt_ecg_jetson_ex`, that computes the scalogram of an input signal and returns an image at the user-specified dimensions. The image uses the `jet(128)` colormap. The `%#codegen` directive in the function indicates that the function is intended for code generation. When using the `coder.gpu.kernelfun` pragma, code generation attempts to map the computations in the `cwt_ecg_jetson_ex` function to the GPU.

```
type cwt_ecg_jetson_ex.m
```

```
function im = cwt_ecg_jetson_ex(TimeSeriesSignal, ImgSize) %#codegen
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

coder.gpu.kernelfun();

%% Create Scalogram
cfs = cwt(TimeSeriesSignal, 'morse', 1, 'VoicesPerOctave', 12);
cfs = abs(cfs);

%% Image generation
cmapj128 = coder.load('cmapj128');
imx = ind2rgb_custom_ecg_jetson_ex(round(255*rescale(cfs))+1,cmapj128.cmapj128);

% resize to proper size and convert to uint8 data type
im = im2uint8(imresize(imx, ImgSize));

end
```

Create the entry-point function, `model_predict_ecg.m`, for code generation. The function takes an ECG signal as input and calls the `cwt_ecg_jetson_ex` function to create an image of the scalogram. The `model_predict_ecg` function uses the network contained in the `ecg_model.mat` file to classify the ECG signal.

```
type model_predict_ecg.m
```

```
function PredClassProb = model_predict_ecg(TimeSeriesSignal) %#codegen
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.
    coder.gpu.kernelfun();

    % parameters
    ModFile = 'ecg_model.mat'; % file that saves neural network model
    ImgSize = [227 227]; % input image size for the ML model
```

```
    % sanity check signal is a row vector of correct length
    assert(isequal(size(TimeSeriesSignal), [1 65536]))
    %% cwt transformation for the signal
    im = cwt_ecg_jetson_ex(TimeSeriesSignal, ImgSize);

    %% model prediction
    persistent model;
    if isempty(model)
        model = coder.loadDeepLearningNetwork(ModFile, 'mynet');
    end

    PredClassProb = predict(model, im);

end
```

To generate a CUDA executable that can be deployed to an NVIDIA target, create a custom main file (`main_ecg_jetson_ex.cu`) and a header file (`main_ecg_jetson_ex.h`). You can generate an example main file and use that as a template to rewrite new main and header files. For more information, see the `GenerateExampleMain` property of `coder.CodeConfig` (MATLAB Coder). The main file calls the code generated for the MATLAB entry-point function. The main file first reads the ECG signal from a text file, passes the data to the entry-point function, and writes the prediction results to a text file (`predClassProb.txt`). To maximize computation efficiency on the GPU, the executable processes single-precision data.

```
type main_ecg_jetson_ex.cu
```

```
//
// File: main_ecg_jetson_ex.cu
//
// This file is only intended to support wavelet deep learning examples.
// It may change or be removed in a future release.

//************************************************************************
// Include Files
#include "rt_nonfinite.h"
#include "model_predict_ecg.h"
#include "main_ecg_jetson_ex.h"
#include "model_predict_ecg_terminate.h"
#include "model_predict_ecg_initialize.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function Definitions

/* Read data from a file*/
int readData_real32_T(const char * const file_in, real32_T data[65536])
{
  FILE* fp1 = fopen(file_in, "r");
  if (fp1 == 0)
  {
    printf("ERROR: Unable to read data from %s\n", file_in);
    exit(0);
  }
  for(int i=0; i<65536; i++)
  {
      fscanf(fp1, "%f", &data[i]);
```

```
  }
  fclose(fp1);
  return 0;
}

/* Write data to a file*/
int writeData_real32_T(const char * const file_out, real32_T data[3])
{
  FILE* fp1 = fopen(file_out, "w");
  if (fp1 == 0)
  {
    printf("ERROR: Unable to write data to %s\n", file_out);
    exit(0);
  }
  for(int i=0; i<3; i++)
  {
    fprintf(fp1, "%f\n", data[i]);
  }
  fclose(fp1);
  return 0;
}

// model predict function
static void main_model_predict_ecg(const char * const file_in, const char * const file_out)
{
  real32_T PredClassProb[3];
  //  real_T b[65536];
  real32_T b[65536];

  // readData_real_T(file_in, b);
  readData_real32_T(file_in, b);

  model_predict_ecg(b, PredClassProb);

  writeData_real32_T(file_out, PredClassProb);

}

// main function
int32_T main(int32_T argc, const char * const argv[])
{
  const char * const file_out = "predClassProb.txt";
  // Initialize the application.
  model_predict_ecg_initialize();

  // Run prediction function
  main_model_predict_ecg(argv[1], file_out); // argv[1] = file_in

  // Terminate the application.
  model_predict_ecg_terminate();
  return 0;
}
```

```
type main_ecg_jetson_ex.h
```

```
//
// File: main_ecg_jetson_ex.h
//
```

```
// This file is only intended to support wavelet deep learning examples.
// It may change or be removed in a future release.

//
//*************************************************************************
#ifndef MAIN_H
#define MAIN_H

// Include Files
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "model_predict_ecg_types.h"

// Function Declarations
extern int32_T main(int32_T argc, const char * const argv[]);

#endif

//
// File trailer for main_ecg_jetson_ex.h
//
// [EOF]
//
```

GPU Code Generation — Specify Target

To create an executable that can be deployed to the target device, set `CodeGenMode` equal to 1. If you want to create an executable that runs locally and connects remotely to the target device, set `CodeGenMode` equal to 2.

The `main` function reads data from the text file specified by `signalFile` and writes the classification results to `resultFile`. Set `ExampleIndex` to choose a representative ECG signal. You will use this signal to test the executable against the `classify` function. `Jetson_BuildDir` specifies the directory for performing the remote build process on the target. If the specified build directory does not exist on the target, then the software creates a directory with the given name.

```
CodeGenMode = 1;
signalFile = 'signalData.txt';
resultFile = 'predClassProb.txt'; % consistent with "main_ecg_jetson_ex.cu"
Jetson_BuildDir = '~/projectECG';
ExampleIndex = 1; % 1,4: type ARR; 2,5: type CHF; 3,6: type NSR

Function_to_Gen = 'model_predict_ecg';
ModFile = 'ecg_model.mat'; % file that saves neural network model; consistent with "main_ecg_jets
ImgSize = [227 227]; % input image size for the ML model

switch ExampleIndex
    case 1 % ARR 7
        SampleSignalIdx = 7;
    case 2 % CHF 97
        SampleSignalIdx = 97;
    case 3 % NSR 132
        SampleSignalIdx = 132;
    case 4 % ARR 31
        SampleSignalIdx = 31;
    case 5 % CHF 101
```

```
        SampleSignalIdx = 101;
    case 6 % NSR 131
        SampleSignalIdx = 131;
end
signal_data = single(ECGData.Data(SampleSignalIdx,:));
ECGtype = ECGData.Labels{SampleSignalIdx};
```

GPU Code Generation — Connect to Hardware

To communicate with the NVIDIA hardware, you create a live hardware connection object using the `jetson` function. You must know the host name or IP address, username, and password of the target board to create a live hardware connection object.

Create a live hardware connection object for the Jetson hardware. During the hardware live object creation checking of hardware, IO server installation and gathering peripheral information on target are performed. This information is displayed in the Command Window.

```
hwobj = jetson('gpucoder-nano-2','ubuntu','ubuntu');
```

```
Checking for CUDA availability on the Target...
Checking for 'nvcc' in the target system path...
Checking for cuDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for prerequisite libraries is complete.
Gathering hardware details...
Checking for third-party library availability on the Target...

Gathering hardware details is complete.
 Board name         : NVIDIA Jetson TX1
 CUDA Version       : 10.0
 cuDNN Version      : 7.3
 TensorRT Version   : 5.0
 GStreamer Version  : 1.14.5
 V4L2 Version       : 1.14.2-1
 SDL Version        : 1.2
 Available Webcams  :
 Available GPUs     : NVIDIA Tegra X1
```

Use the `coder.checkGpuInstall` (GPU Coder) function and verify that the compilers and libraries needed for running this example are set up correctly on the hardware.

```
envCfg = coder.gpuEnvConfig('jetson');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.HardwareObject = hwobj;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg)
```

```
ans = struct with fields:
                 gpu: 1
                cuda: 1
               cudnn: 1
            tensorrt: 0
        basiccodegen: 0
       basiccodeexec: 0
         deepcodegen: 1
        deepcodeexec: 0
    tensorrtdatatype: 0
```

```
            profiling: 0
```

GPU Code Generation — Compile

Create a GPU code configuration object necessary for compilation. Use the `coder.hardware` function to create a configuration object for the Jetson platform and assign it to the `Hardware` property of the code configuration object `cfg`. Use `'NVIDIA Jetson'` for the Jetson TX1 or TX2 boards. The custom main file is a wrapper that calls the entry-point function in the generated code. The custom file is required for a deployed executable.

Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. The code generator takes advantage of NVIDIA® CUDA® deep neural network library (cuDNN) for NVIDIA GPUs. cuDNN is a GPU-accelerated library of primitives for deep neural networks.

```
if CodeGenMode == 1
    cfg = coder.gpuConfig('exe');
    cfg.Hardware = coder.hardware('NVIDIA Jetson');
    cfg.Hardware.BuildDir = Jetson_BuildDir;
    cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
    cfg.CustomSource = fullfile('main_ecg_jetson_ex.cu');
elseif CodeGenMode == 2
    cfg = coder.gpuConfig('lib');
    cfg.VerificationMode = 'PIL';
    cfg.Hardware = coder.hardware('NVIDIA Jetson');
    cfg.Hardware.BuildDir = Jetson_BuildDir;
    cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
end
```

To generate CUDA code, use the `codegen` function and pass the GPU code configuration along with the size and type of the input for the `model_predict_ecg` entry-point function. After code generation on the host is complete, the generated files are copied over and built on the target.

```
codegen('-config ',cfg,Function_to_Gen,'-args',{signal_data},'-report');
```

```
Code generation successful: View report
```

GPU Code Generation — Execute

If you compiled an executable to be deployed to the target, write the example ECG signal to a text file. Use the `putFile()` function of the hardware object to place the text file on the target. The `workspaceDir` property contains the path to the `codegen` folder on the target.

```
if CodeGenMode == 1
    fid = fopen(signalFile,'w');
    for i = 1:length(signal_data)
        fprintf(fid,'%f\n',signal_data(i));
    end
    fclose(fid);
    hwobj.putFile(signalFile,hwobj.workspaceDir);
end
```

Run the executable.

When running the deployed executable, delete the previous result file if it exists. Use the `runApplication()` function to launch the executable on the target hardware, and then the

`getFile()` function to retrieve the results. Because the results may not exist immediately after the `runApplication()` function call returns, and to allow for communication delays, set a maximum time for fetching the results to 90 seconds. Use the `evalc` function to suppress the command-line output.

```
if CodeGenMode == 1 % run deployed executable
    maxFetchTime = 90;
    resultFile_hw = fullfile(hwobj.workspaceDir,resultFile);
    if ispc
        resultFile_hw = strrep(resultFile_hw,'\','/');
    end

    ta = tic;

    hwobj.deleteFile(resultFile_hw)
    hwobj.runApplication(Function_to_Gen,signalFile);

    tf = tic;
    success = false;
    while toc(tf) < maxFetchTime
        try
            evalc('hwobj.getFile(resultFile_hw)');
            success = true;
        catch ME
        end
        if success
            break
        end
    end
    fprintf('Fetch time = %.3e sec\n',toc(tf));
    assert(success,'Unable to fetch the prediction')
    PredClassProb = readmatrix(resultFile);
    PredTime = toc(ta);
elseif CodeGenMode == 2 % run PIL executable
    ta = tic;
    eval(sprintf('PredClassProb = %s_pil(signal_data);',Function_to_Gen));
    PredTime = toc(ta);
    eval(sprintf('clear %s_pil;',Function_to_Gen)); % terminate PIL execution
end

### Launching the executable on the target...
Executable launched successfully with process ID 5672.
Displaying the simple runtime log for the executable...

Note: For the complete log, run the following command in the MATLAB command window:
system(hwobj,'cat /home/ubuntu/projectECG/MATLAB_ws/R2021a/C/Users/pkostele/OneDrive_-_MathWorks/

Fetch time = 9.743e+00 sec
```

Use the `classify` function to predict the class labels for the example signal.

```
ModData = load(ModFile,'net');
im = cwt_ecg_jetson_ex(signal_data,ImgSize);
[ModPred, ModPredProb] = classify(ModData.net,im);
PredCat = categories(ModPred)';
```

Compare the results.

```
PredTableJetson = array2table(PredClassProb(:)','VariableNames',matlab.lang.makeValidName(PredCa
fprintf('tPred = %.3e sec\nExample ECG Type: %s\n',PredTime,ECGtype)
```

```
tPred = 1.288e+01 sec
Example ECG Type: ARR
```

```
disp(PredTableJetson)
```

```
      ARR         CHF          NSR
    ________    _________    _________

    0.99872     0.001131     0.000153
```

```
PredTableMATLAB = array2table(ModPredProb(:)','VariableNames',matlab.lang.makeValidName(PredCat)
disp(PredTableMATLAB)
```

```
      ARR          CHF           NSR
    ________    __________    ___________

    0.99872     0.0011298     0.00015316
```

Close the hardware connection.

```
clear hwobj
```

Summary

This example shows how to create and deploy a CUDA executable that uses a CNN to classify ECG signals. You also have the option to create an executable the runs locally and connects to the remote target. A complete workflow is presented in this example. After the data is downloaded, the CWT is used to extract features from the ECG signals. Then SqueezeNet is retrained to classify the signals based on their scalograms. Two user-defined functions are created and compiled on the target NVIDIA device. Results of the executable are compared with MATLAB.

References

1. Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661–670.
2. Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit,and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23: e215–e220. [Circulation Electronic Pages; `http://circ.ahajournals.org/content/101/23/e215.full`]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.
3. Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20. Number 3, May-June 2001, pp. 45–50. (PMID: 11446209)

Supporting Functions

helperCreateECGDirectories

```
function helperCreateECGDirectories(ECGData,parentFolder,dataFolder)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.
```

```
rootFolder = parentFolder;
localFolder = dataFolder;
mkdir(fullfile(rootFolder,localFolder))

folderLabels = unique(ECGData.Labels);
for i = 1:numel(folderLabels)
    mkdir(fullfile(rootFolder,localFolder,char(folderLabels(i))));
end
end
```

helperPlotReps

```
function helperPlotReps(ECGData)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

folderLabels = unique(ECGData.Labels);

for k=1:3
    ecgType = folderLabels{k};
    ind = find(ismember(ECGData.Labels,ecgType));
    subplot(3,1,k)
    plot(ECGData.Data(ind(1),1:1000));
    grid on
    title(ecgType)
end
end
```

helperCreateRGBfromTF

```
function helperCreateRGBfromTF(ECGData,parentFolder, childFolder)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

imageRoot = fullfile(parentFolder,childFolder);

data = ECGData.Data;
labels = ECGData.Labels;

[~,signalLength] = size(data);

fb = cwtfilterbank('SignalLength',signalLength,'VoicesPerOctave',12);
r = size(data,1);

for ii = 1:r
    cfs = abs(fb.wt(data(ii,:)));
    im = ind2rgb(im2uint8(rescale(cfs)),jet(128));

    imgLoc = fullfile(imageRoot,char(labels(ii)));
    imFileName = strcat(char(labels(ii)),'_',num2str(ii),'.jpg');
    imwrite(imresize(im,[227 227]),fullfile(imgLoc,imFileName));
end
end
```

See Also

cwt | coder.DeepLearningConfig | cwtfilterbank

More About

- “GPU Acceleration of Scalograms for Deep Learning” (Wavelet Toolbox)

Deploy Signal Classifier Using Wavelets and Deep Learning on Raspberry Pi

This example shows the workflow to classify human electrocardiogram (ECG) signals using the Continuous Wavelet Transform (CWT) and a deep convolutional neural network (CNN). This example also provides information on how to generate and deploy the code and CNN for prediction on a Raspberry Pi target (ARM®-based device).

SqueezeNet is a deep CNN originally designed to classify images in 1000 categories. In the example "Classify Time Series Using Wavelet Analysis and Deep Learning" (Wavelet Toolbox), SqueezeNet is retrained to classify ECG waveforms based on their *scalograms*. A scalogram is a time-frequency representation of the signal and is the absolute value of the CWT of the signal. We reuse the retrained SqueezeNet in this example.

ECG Data Description

In this example, ECG data from PhysioNet is used. The ECG data is obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). The data set includes 96 recordings from persons with ARR, 30 recordings from persons with CHF, and 36 recordings from persons with NSR. The 162 ECG recordings are from three PhysioNet databases: MIT-BIH Arrhythmia Database [2][3], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1][3]. Shortened ECG data of the above references can be downloaded from the GitHub repository.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library version 19.05 (on the target ARM hardware)
- Environment variables for the compilers and libraries
- MATLAB Support Package for Raspberry Pi Hardware
- MATLAB Coder Interface for Deep Learning Libraries support package

For supported versions of libraries and for information about setting up environment variables, see "Prerequisites for Deep Learning with MATLAB Coder" (MATLAB Coder). This example is not supported in MATLAB Online™.

Functionality of Generated Code

The core function in the generated executable, `processECG`, uses 65,536 samples of single-precision ECG data as input. The function:

1. Takes the CWT of the ECG data.
2. Obtains scalogram from wavelet coefficients.
3. Converts the scalogram to an RGB mage of dimension 227-by-227-by-3. This makes the image compatible with the SqueezeNet network architecture.
4. Performs prediction to classify the image using SqueezeNet.

```
type processECG
```

```
function [YPred] = processECG(input)
% processECG function - converts 1D ECG to image and predicts the syndrome
% of heart disease
```

```
%
% This function is only intended to support the example:
% Signal Classification Code Generation Using Wavelets and
% Deep Learning on Raspberry Pi. It may change or be removed in a
% future release.

% Copyright 2020 The MathWorks, Inc.

    % colourmap for image transformation
    persistent net jetdata;
    if(isempty(jetdata))
        jetdata = colourmap(128,class(input));
    end

    % Squeezenet trained network
    if(isempty(net))
        net = coder.loadDeepLearningNetwork('trainedNet.mat');
    end

    % Wavelet Transformation & Image conversion
    cfs = ecg_to_Image(input);
    image = ind2rgb(im2uint8(rescale(cfs)),single(jetdata));
    image = im2uint8(imresize(image,[227,227]));

    % figure
    if isempty(coder.target)
        imshow(image);
    end

    % Prediction
    [YPred] = predict(net,image);


    %% ECG to image conversion
    function cfs = ecg_to_Image(input)

        %Wavelet Transformation
        persistent filterBank
        [~,siglen] = size(input);
        if isempty(filterBank)
            filterBank = cwtfilterbank('SignalLength',siglen,'VoicesPerOctave',6);
        end
        %CWT conversion
        cfs = abs(filterBank.wt(input));
    end


    %% Colourmap
    function J = colourmap(m,class)

        n = ceil(m/4);
        u = [(1:1:n)/n ones(1,n-1) (n:-1:1)/n]';
        g = ceil(n/2) - (mod(m,4)==1) + (1:length(u))';
        r = g + n;
        b = g - n;
        r1 = r(r<=128);
        g1 = g(g<=128);
        b1 = b(b >0);
```

```
        J = zeros(m,3);
        J(r1,1) = u(1:length(r1));
        J(g1,2) = u(1:length(g1));
        J(b1,3) = u(end-length(b1)+1:end);
        feval = str2func(class);
        J = feval(J);
    end
end
```

Create Code Generation Configuration Object

Create a code generation configuration object for generation of an executable program. Specify generation of C++ code.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Set Up Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the same version of the ARM Compute library as the one on the Raspberry Pi. Specify the architecture of the Raspberry Pi.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv7';
```

Attach Deep Learning Configuration Object to Code Generation Configuration Object

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object. Make the MATLAB Source Comments visible in the configuration object at the time of code generation.

```
cfg.DeepLearningConfig = dlcfg;
cfg.MATLABSourceComments = 1;
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi Support Package function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `'raspiname'` with the name of your Raspberry Pi
- `'pi'` with your user name
- `'password'` with your password

```
r = raspi('172.18.76.69','pi','raspberry');
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
buildDir = '~/remdirECG';
cfg.Hardware.BuildDir = buildDir;
```

Provide C++ Main File for Code Execution

The C++ main file reads the input ECG data, calls the `processECG` function to perform preprocessing and deep learning using CNN on the ECG data, and displays the classification probability.

Specify the main file in the code generation configuration object. To learn more about generating and customizing `main_ecg_raspi.cpp`, refer to “Generating Standalone C/C++ Executables from MATLAB Code” (MATLAB Coder).

```
cfg.CustomSource = 'main_ecg_raspi.cpp';
```

Generate Source C++ Code Using codegen

Use the `codegen` function to generate the C++ code. When `codegen` is used with the MATLAB Support Package for Raspberry Pi Hardware, the executable is built on the Raspberry Pi board.

Make sure to set the environment variables `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the Raspberry Pi. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

```
codegen -config cfg processECG -args {ones(1,65536,'single')} -d arm_compute
```

```
 Deploying code. This may take a few minutes.
```

Fetch Generated Executable Directory

To test the generated code on the Raspberry Pi, copy the input ECG signal to the generated code directory. You can find this directory manually or by using the `raspi.utils.getRemoteBuildDirectory` API. This function lists the directories of the binary files that are generated by using `codegen`.

```
applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName','processECG')
```

```
applicationDirPaths=1×4 cell array
    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}
```

The complete path to the remote build directory is derived from the present working directory. If you do not know which `applicationDirPaths` entry contains the generated code, use the helper function `helperFindTargetDir`. Otherwise, specify the proper directory.

```
directoryUnknown = true;

if directoryUnknown
    targetDirPath = helperFindTargetDir(applicationDirPaths);
else
    targetDirPath = applicationDirPaths{1}.directory;
end
```

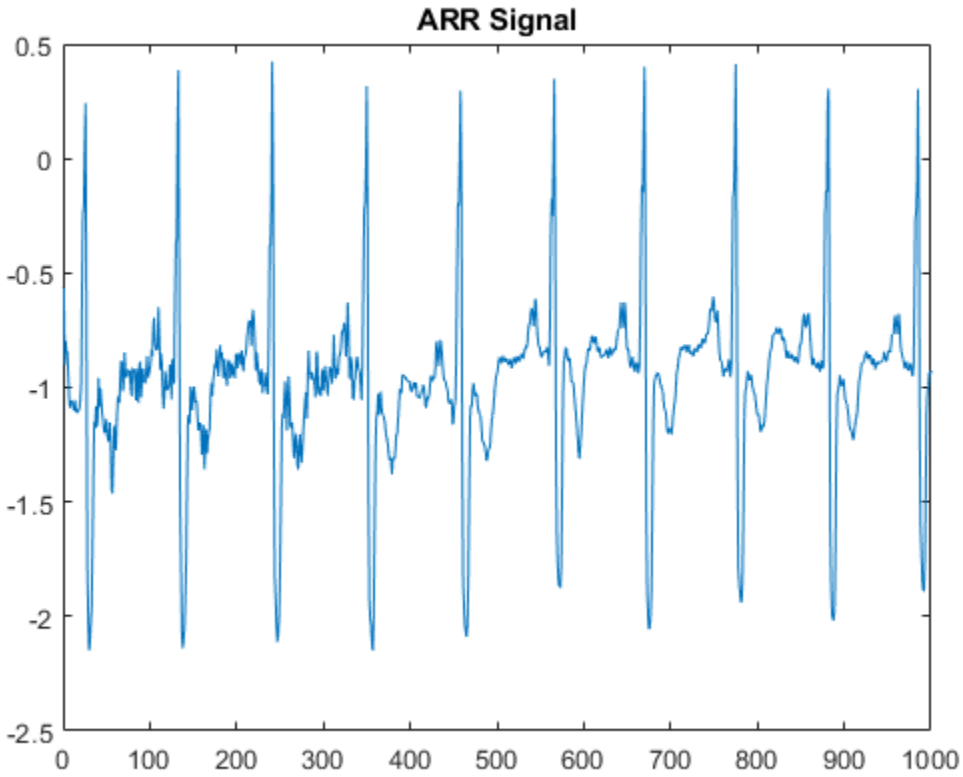
Copy Input File to Raspberry Pi

The text file `input_ecg_raspi.csv` contains the ECG samples of a representative ARR signal. To copy the file required to run the executable program, use `putFile`, which is available with the MATLAB Support Package for Raspberry Pi Hardware.

```
r.putFile('input_ecg_raspi.csv', targetDirPath);
```

For a pictorial representation, the first 1000 samples can be plotted by using these steps.

```
input = dlmread('input_ecg_raspi.csv');
plot(input(1:1000))
title('ARR Signal')
```



Run Executable on Raspberry Pi

Run the executable program on the Raspberry Pi from MATLAB and direct the output back to MATLAB. The input file name is passed as the command line argument for the executable.

```
exeName = 'processECG.elf';           % executable name
fileName = 'input_ecg_raspi.csv';     % Input ECG file that is pushed to target
command = ['cd ' targetDirPath ';./' exeName ' ' fileName];
output = system(r,command)
```

```
output =
    'Predicted Values on the Target Hardware
     ARR            CHF            NSR
     0.806078    0.193609    0.000313103
     '
```

References

1 Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661–670.

2 Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit,and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." Circulation. Vol. 101, Number 23: e215–e220. [Circulation Electronic Pages; `http://circ.ahajournals.org/content/101/23/e215.full`]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.

3 Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20. Number 3, May-June 2001, pp. 45–50. (PMID: 11446209)

Supporting Functions

helperFindTargetDir

```
function targetDir = helperFindTargetDir(dirPaths)
%
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

% find pwd
p = pwd;
if ispc
    % replace blank spaces with underscores
    p = strrep(p,' ','_');

    % split path into component folders
    pSplit = regexp(p,filesep,'split');

    % Since Windows uses colons, remove any colons that occur
    for k=1:numel(pSplit)
        pSplit{k} = erase(pSplit{k},':');
    end

    % now build the path using Linux file separation
    pLinux = '';
    for k=1:numel(pSplit)-1
        pLinux = [pLinux,pSplit{k},'/'];
    end
    pLinux = [pLinux,pSplit{end}];
else
    pLinux = p;
end

targetDir = '';
for k=1:numel(dirPaths)
    d = strfind(dirPaths{k}.directory,pLinux);
    if ~isempty(d)
        targetDir = dirPaths{k}.directory;
        break
    end
end

if numel(targetDir) == 0
    disp('Target directory not found.');
end
end
```

Deploy Signal Segmentation Deep Network on Raspberry Pi

This example details the workflow for waveform segmentation of an electrocardiogram (ECG) signal using short-time Fourier transform and a bidirectional long short-term memory (BiLSTM) network. The example also provides information on how to generate and deploy the code and the trained BiLSTM network for segmentation on a Raspberry Pi™ target (ARM®-based device).

The pretrained network in the example is similar to the “Waveform Segmentation Using Deep Learning” (Signal Processing Toolbox) example.

This example details:

- Processor-in-the-loop (PIL) based workflow to verify generated code deployed and running on a Raspberry Pi from MATLAB™
- Generation of a standalone executable

The PIL verification process is a crucial part of the design cycle to check that the behavior of the generated code matches the design before deploying a standalone executable.

ECG Dataset

This example uses ECG signals from the publicly available QT Database [1 on page 12-0] [2 on page 12-0]. The data consists of roughly 15 minutes of labeled ECG recordings, with a sample rate of 250 Hz, measured from a total of 105 patients.

The ECG signal can be divided into the following beat morphologies [3 on page 12-0]:

- P wave — A small deflection before the QRS complex representing atrial depolarization
- QRS complex — Largest amplitude portion of the heartbeat
- T wave — A small deflection after the QRS complex representing ventricular repolarization

The segmentation of these regions of ECG waveforms can provide the basis for measurements that assess the overall health of the human heart and the presence of abnormalities.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- MATLAB® Coder™
- Embedded Coder™
- Deep Learning Toolbox™
- Deep Learning Support for MATLAB Coder
- MATLAB Support Package for Raspberry Pi™

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder) (MATLAB Coder).

Functionality of Generated Code

The core function in the generated executable:

- Uses 15,000 samples of single-precision ECG data as input.
- Computes the short-time Fourier transform of the signal.
- Standardizes and normalizes the output.
- Labels regions of the signal using the pretrained BiLSTM network.
- Generates an output file with the labels.

`waveformSegmentation` Function

An *entry-point* function, also known as the *top-level* or *primary* function, is a function you define for code generation. You must define an entry-point function that calls code-generation-enabled functions and generates C/C++ code from the entry-point function. All functions within the entry-point function must support code generation.

In this example, w`aveformSegmentation` is the entry-point function. It takes an ECG signal as an input and passes it to the trained BiLSTM network for prediction. The `performPreprocessing` function preprocesses the raw signal and applies the short-time Fourier transform. The `genClassifiedResults` function passes the preprocessed signal to the network for prediction and displays the classification results.

```
type waveformSegmentation
```

```
function out = waveformSegmentation(in)
%#codegen
persistent net;

if isempty(net)
    net = coder.loadDeepLearningNetwork('trained-network-STFTBILSTM.mat', 'net');
end


preprocessedSignal = performPreprocessing(in);
out = cell(3,1);


for indx = 1:3
  out{indx,1} =  genClassifedResults(net.predict(preprocessedSignal{1,indx}));
end



end
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `'raspiname'` with the name of your Raspberry Pi
- `'pi'` with your username
- `'password'` with your password

```
r = raspi('raspiname','pi','password');
```

The example shows the PIL-based workflow for verification of code and design and then creates and deploys a standalone executable. Optionally, if you want to directly deploy a standalone executable, you can skip PIL execution and go to creating a standalone execution.

Generate PIL MEX Function

The first step shows a PIL-based workflow to generate a MEX function for the `waveformSegmentation` function.

Set Up Code Generation Configuration Object for a Static Library

Create a code configuration object for a static library and set the verification mode to `'PIL'`. Set the target language to `'C++'`.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
cfg.TargetLang = 'C++';
```

Set Up Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute library as the one on the Raspberry Pi. Specify the architecture of the Raspberry Pi. (This example requires ARM Compute Library v19.05).

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv7';
```

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object. Set the configuration object with MATLAB Source Comments visible in the code generation.

```
cfg.DeepLearningConfig = dlcfg;
cfg.MATLABSourceComments = 1;
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for the Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
cfg.Hardware.BuildDir = '~/waveformSegmentation';
```

Generate Source C++ Code Using codegen Function

Use the `codegen` function to generate the C++ code. When `codegen` is used with MATLAB Support Package for Raspberry Pi Hardware, the generated code is downloaded to the board and compiled there. A PIL MEX function is generated to communicate between MATLAB and the generated code running on the Raspberry Pi.

Make sure to set the environment variables `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the Raspberry Pi. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder) (MATLAB Coder).

```
codegen -config cfg waveformSegmentation -args {coder.typeof(single(ones(1,15000)),[1,15000],[0,0
```

```
### Target device has no native communication support. Checking connectivity configuration regist
 Deploying code. This may take a few minutes.
### Target device has no native communication support. Checking connectivity configuration regist
### Connectivity configuration for function 'waveformSegmentation': 'Raspberry Pi'
Location of the generated elf : /home/pi/waveformSegmentation/MATLAB_ws/R2020b/C/Users/eshashah/C
Code generation successful: View report
```

Run the Executable Program on Raspberry Pi

Load the MAT-file `ecgsignal_test`. The file stores a sample ECG signal on which you can test the generated code.

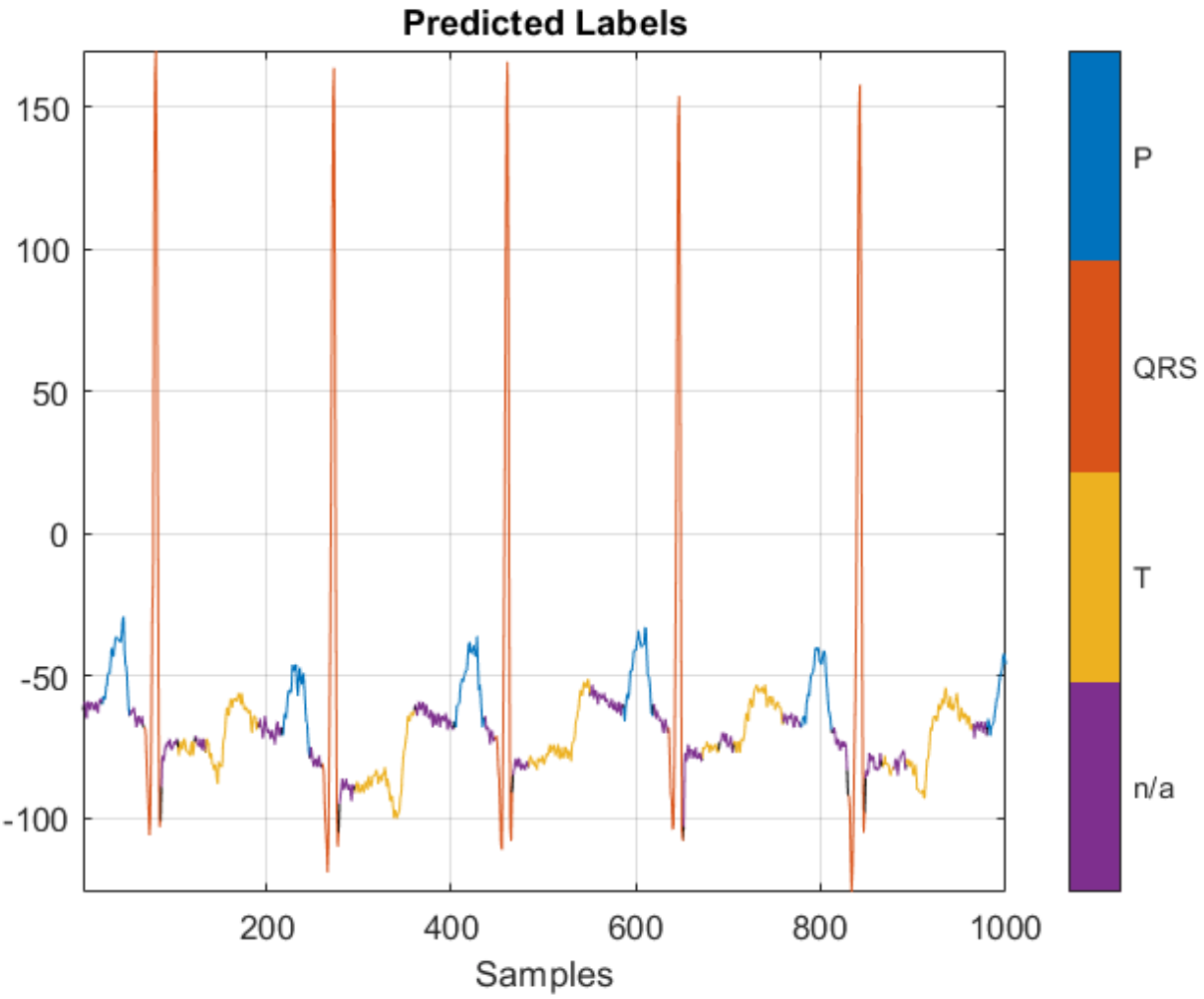
Run the generated `waveformSegmentation_pil` MEX function on the test signal.

```
load ecgsignal_test.mat;
out = waveformSegmentation_pil(test);
```

```
### Starting application: 'codegen\lib\waveformSegmentation\pil\waveformSegmentation.elf'
    To terminate execution: clear waveformSegmentation_pil
### Launching application waveformSegmentation.elf...
```

Display the signals with predicted labels.

```
labels = categorical(out{1}(1,2000:3000));
msk = signalMask(labels);
plotsigroi(msk,test(1,2000:3000))
title('Predicted Labels')
```



After the verifying the output of the PIL MEX function, you can create a standalone executable for the `waveformSegmentation` function.

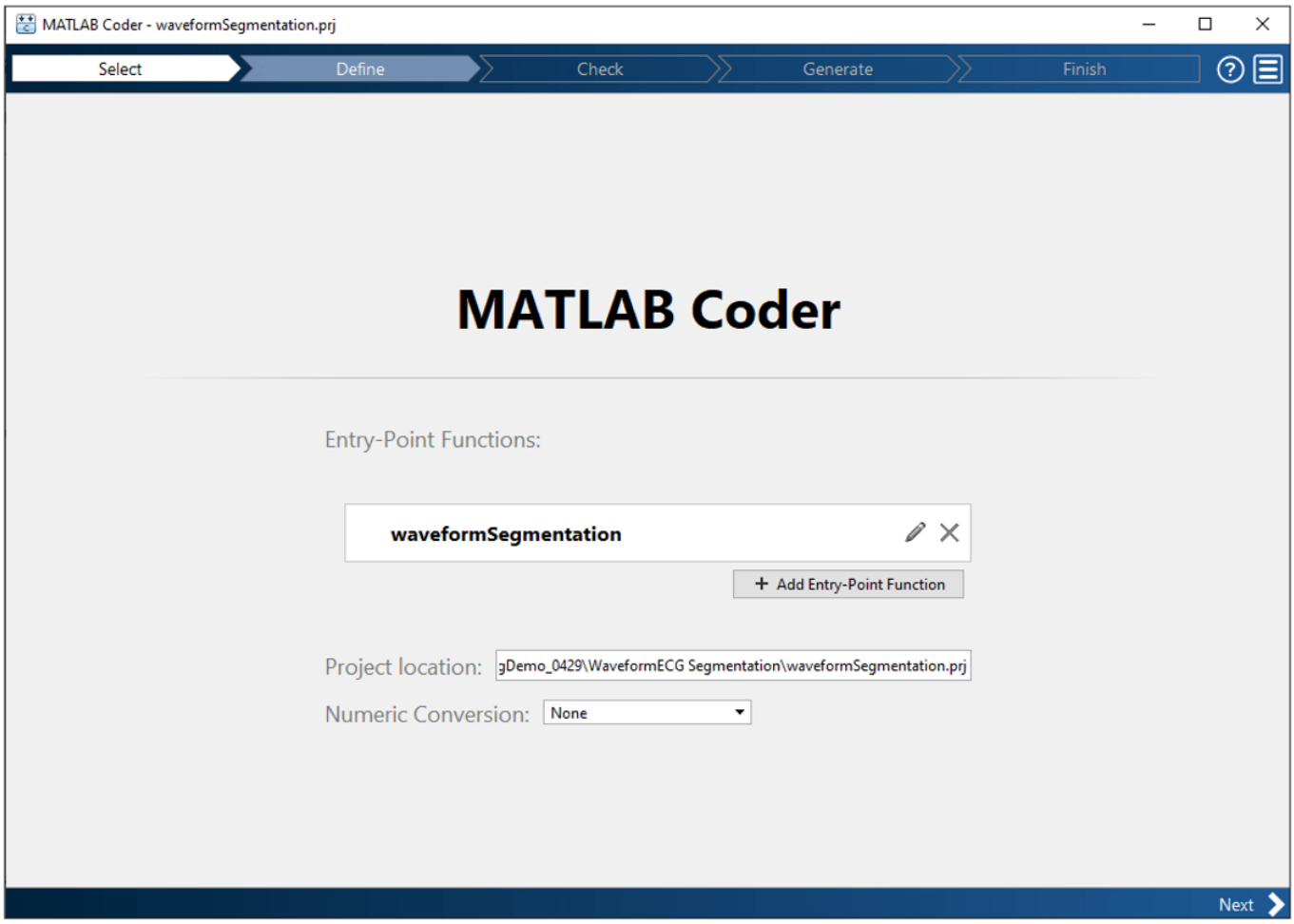
The next part shows the code generation workflow to create a generate and deploy a standalone executable in the code for prediction on a Raspberry Pi using the MATLAB Coder App.

Create a Standalone Executable Using the MATLAB Coder App

The **MATLAB Coder** app generates C or C++ code from MATLAB® code. The workflow-based user interface steps you through the code generation process. The following steps describe a brief workflow using the MATLAB Coder app. For more details, see MATLAB Coder (MATLAB Coder) and “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder).

Select the Entry-Point Function File

On the **Apps** tab, click the down arrow on the far right of the toolstrip to expand the apps gallery. Under **Code Generation**, click **MATLAB Coder**. The app opens the **Select Source Files** page. Enter or select the name of the entry-point function, `waveformSegmentation`.



Click **Next** to go to the **Define Input Types** page.

Define Input Types

1. Select **Let me enter input or global types directly** and set the value of the input `in` as single `(1x15000)`.

2. Click **Next** to go to the **Generate Code** step. Skip the **Check for Run-Time Issues** step because MEX generation is not supported for code generation with the ARM Compute Library.

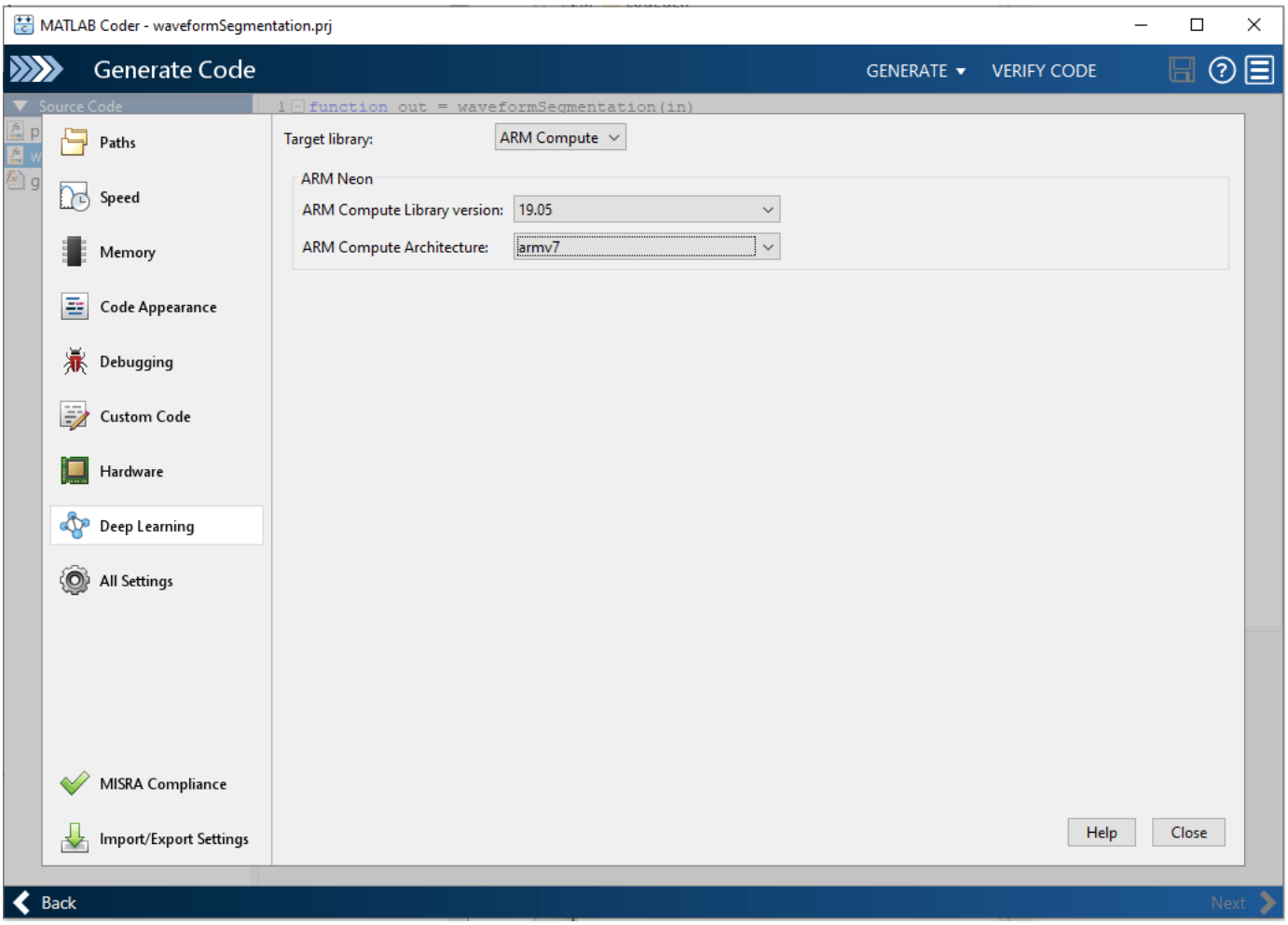
Generate Code

1. Set values in the generate code dialog box:

- Set **Build Type** to `Executable (.exe)`
- Set **Language** to `C++`
- Set **Hardware Board** as `Raspberry Pi`

2. Click the **More Settings** button:

- In the **Custom Code** pane, in additional source files, browse and select `ecgsegmentation_main.cpp`. For more information on writing a C/C++ main function, refer to "Structure of Generated Example C/C++ Main Function" (MATLAB Coder).
- In the **Hardware** pane, set the **username** and **password** for the Raspberry Pi board.
- In the **Deep Learning** pane, set **Target library** to `ARM Compute`. Specify **ARM Compute Library version** and **ARM Compute Architecture**.



3. Close the Settings window and generate code.

4. Click **Next** to go to the **Finish Workflow** page.

Fetch Generated Executable Directory

Once the code generation is completed the following lines of code to test the generated code on the Raspberry Pi, copy the input ECG signal to the generated code directory. You can find this directory manually or by using the `raspi.utils.getRemoteBuildDirectory` API. This function lists the directories of the binary files that are generated by using the `codegen` function. Assuming that the binary is found in only one directory, enter:

```
applicationDirPaths =...
```

```
raspi.utils.getRemoteBuildDirectory('applicationName','waveformSegmentation')
;

targetDirPath = applicationDirPaths{1}.directory;
```

Copy Input Files to the Raspberry Pi

To copy files required to run the executable program, use `putFile`, which is available with the MATLAB Support Package for Raspberry Pi Hardware. The `input.csv` file contains a sample ECG signal that is used to test the deployed code.

```
r.putFile('input.csv', targetDirPath);

input = dlmread('input.csv');
```

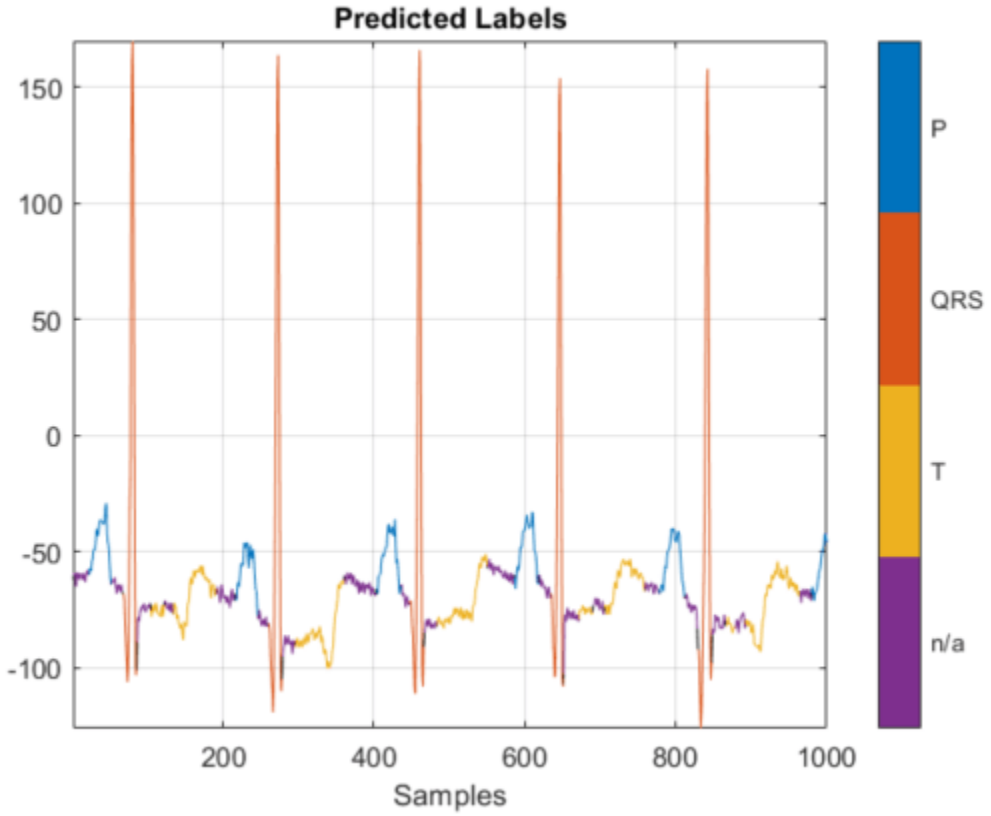
Run Executable program on Raspberry Pi

Run the executable program on the Raspberry Pi from MATLAB and get the output file to MATLAB. Input file name should be passed as the command line argument for the executable.

```
exeName = 'waveformSegmentation.elf'; % Executable name

command = ['cd ' targetDirPath ';./' exeName];

system(r,command)

outputPath = strcat(targetDirPath,'/*.txt');

getFile(r,outputPath)
```

Display the signals with predicted labels. The output is depicted in the figure.

```
load ecgsignal_test.mat;

labels = categorical(textread('out.txt','%s')');

msk = signalMask(labels(1,2000:3000));

plotsigroi(msk,test(1,2000:3000))

title('Predicted Labels')
```



References

[1] McSharry, Patrick E., et al. "A dynamical model for generating synthetic electrocardiogram signals." *IEEE® Transactions on Biomedical Engineering.* Vol. 50, No. 3, 2003, pp. 289–294.

[2] Laguna, Pablo, Raimon Jané, and Pere Caminal. "Automatic detection of wave boundaries in multilead ECG signals: Validation with the CSE database." *Computers and Biomedical Research*. Vol. 27, No. 1, 1994, pp. 45–60.

[3] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffery M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation.* Vol. 101, No. 23, 2000, pp. e215–e220. [Circulation Electronic Pages; http://circ.ahajournals.org/content/101/23/e215.full].

See Also

Apps
MATLAB Coder

Functions
`codegen` | `fsst` | `signalMask`

More About

- “Long Short-Term Memory Networks” on page 1-75
- “Generate C Code by Using the MATLAB Coder App” (MATLAB Coder)
- “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)
- “Structure of Generated Example C/C++ Main Function” (MATLAB Coder)

Anomaly Detection Using Autoencoder and Wavelets

This example shows how wavelet features can be used to detect arc faults in a DC system. For the safe operation of DC distribution systems, it is important to identify arc faults and prefault signals that can be caused by deterioration of wire insulation due to aging, abrasion, or rodent bites. These arc faults can result in shock, fires, and system failures in the microgrid. Unlike the fault signals in AC distribution systems, these prefault arc flash signals are difficult to identify as they do not generate significant power to trigger the circuit breakers. As a result, these signals can exist in the system for hours without being detected.

Arc fault detection using the wavelet transform was studied in [1] on page 12-0 . This example follows the feature extraction procedure detailed in [1] on page 12-0 . The feature extraction involves filtering the load signals using the Daubechies `db3` wavelet followed by normalization. Further, an autoencoder trained with signal features under normal conditions is used to detect arc faults in load signals. The DC arc model used to generate the fault signal and the pretrained network used to detect the arc faults are provided in the example folder. As training the network and arc detection in larger signals can take significantly long simulation time, in this example we only report the detection results.

Training and Testing Setup

The autoencoder is trained using the load signal generated by the Simulink™ model `DCNoArc` under normal conditions, i.e., without arc faults. The model `DCNoArc` was built using components from the `Specialized Power System` library in `Simscape Electrical™`.

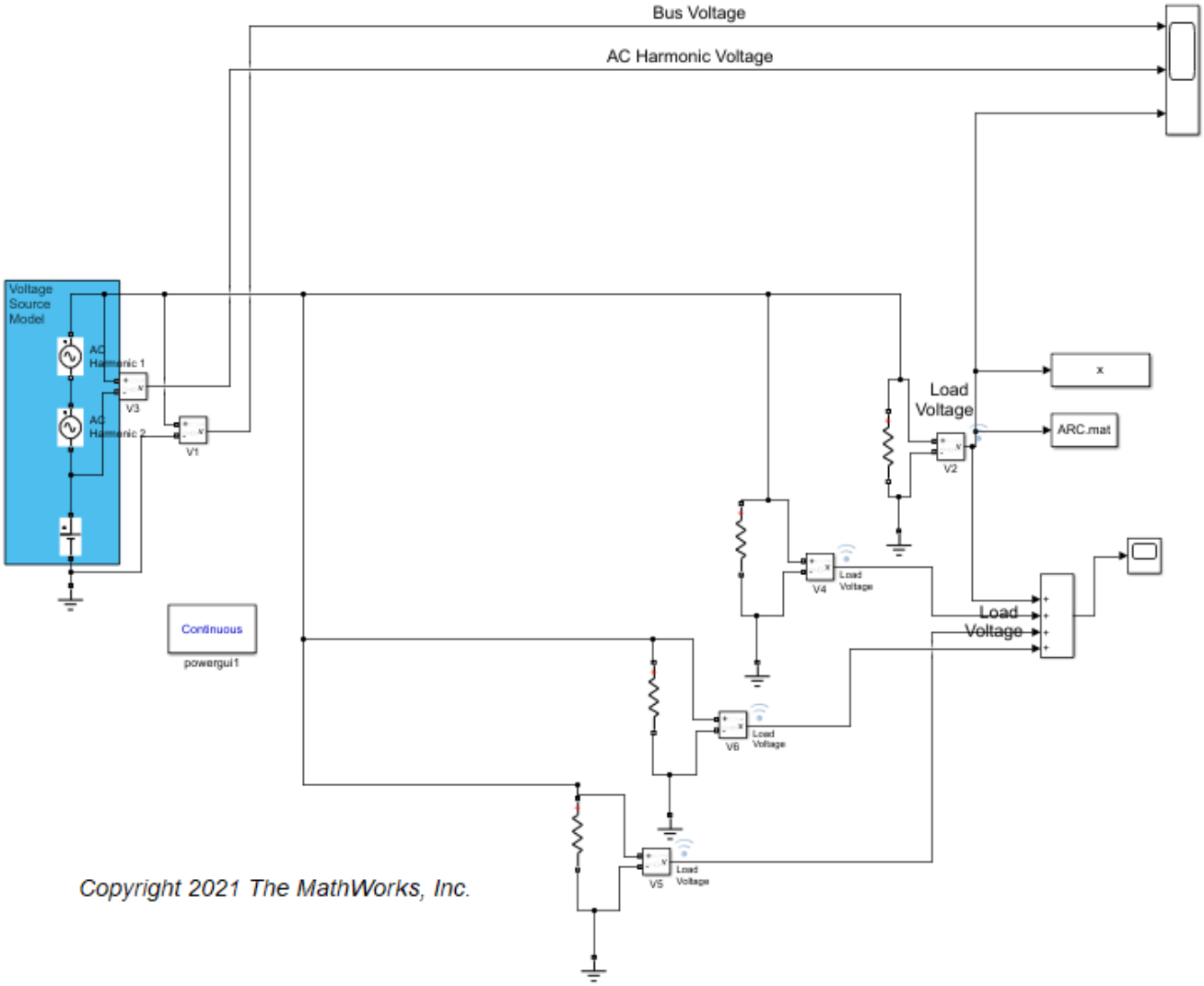


Figure 1: `DCNoArc` model for generating load signal under normal conditions.

The voltage sources are modeled using the following parameters:

- *AC Harmonic Source 1:* 10V ac voltage and 120 Hz frequency
- *AC Harmonic Source 2:* 20V ac voltage and 2000 Hz frequency
- *DC voltage source:* 1000V

In the model `DCArcModelFinal` we add arc fault generation in every load branch. The model uses the Cassie arc model for synthetic arc fault generation. The arc model works like an ideal conductance until the arc ignites at the contact separation time.

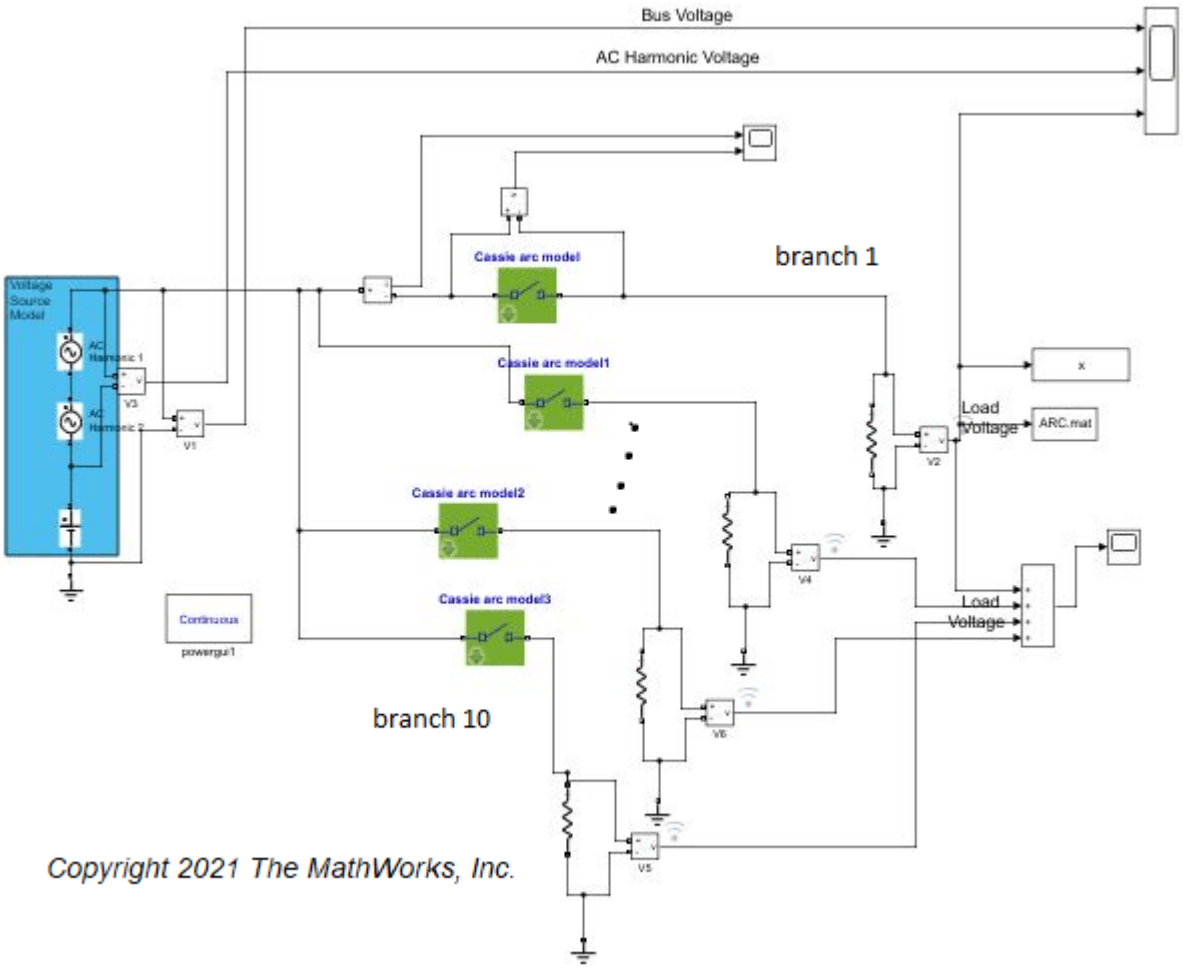


Figure 2: `DCArcModelFinal` model for generating load signal with arc fault.

Cassie Arc Model:

The Cassie arc model is one of the most studied black box models for generating synthetic arc. The model is described by the following differential equation:

$$\frac{dg}{dt} = \frac{g}{\tau}\left(\frac{u^2}{U_c^2} - 1\right)$$

where

- g is the conductance of the arc in siemens
- τ is the arc time constant in seconds
- u is the voltage across the arc in volts
- U_c is constant arc voltage in volts

The Cassie arc models were implemented in Simulink™ using the following parameter values:

- Initial conductance g(0) is 1e4 siemens
- Constant arc voltage U_c = 100 V
- Arc time constant 1.2e-6 seconds

The contact separation times for the arc models are chosen at random. All the parameters have been loaded in the `initFcn` callbacks in the **Model Properties** of the **Model Settings** tab. We use the `DCArcModelFinal` model to generate a faulty load signal to test the autoencoder.

Anomaly Detection with Autoencoder

Autoencoders are used to detect anomalies in a signal. To this end, the autoencoder accepts training data without anomalies as input and tries to reconstruct the input signal using a neural network. The network weights are calculated such that the reconstruction error is minimized. The statistics of the reconstruction error for the training data can be used to select the threshold in the anomaly detection block that determines the detection performance of the autoencoder. Every time the autoencoder encounters an anomaly in the testing data, it produces a large reconstruction error. If the error is above the threshold in the anomaly detection block, the encoder declares it to be an anomaly. In this example, we used root-mean-square error (RMSE) as the reconstruction error metric.

For this example, we trained two autoencoders using the load signal under normal conditions without arc fault. One autoencoder was trained using the raw load signal as training data. This encoder uses the raw faulty load signal to detect arc faults. The second autoencoder was trained using wavelet features. Arc fault detection is subsequently done on wavelet features as opposed to the raw data. For training and testing the network, we assume that the load consists of 10 parallel resistive branches with randomly chosen resistance values. For arc fault signal generation, we add a Cassie arc model in every load branch. The contact separation times of the models are such that they are triggered randomly throughout the simulation period. Just like in a real-time DC system, the load signals from both normal and faulty conditions have added white noise.

Feature Extraction

The wavelet-based autoencoder was trained and tested on signals filtered using the discrete wavelet transform (DWT). Following [1] on page 12-0 , the Daubechies `db3` wavelet was used.

The following figures show the wavelet-filtered load signals under normal and faulty conditions. The wavelet-filtered faulty signal captures the variation due to arc faults. For training and testing purposes, the wavelet-filtered signals are segmented into 100-sample frames.

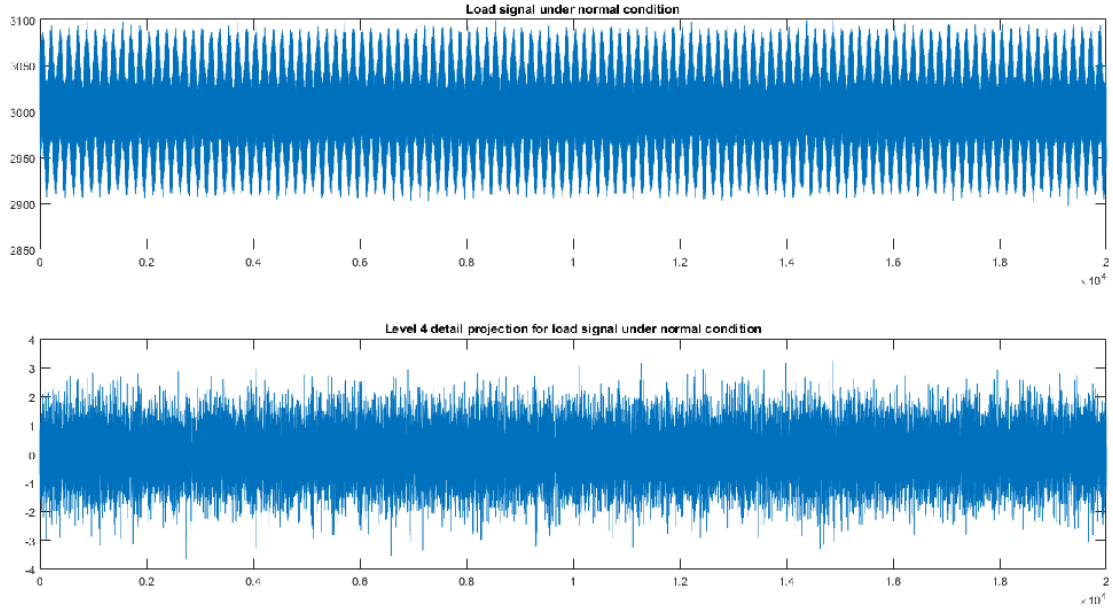


Figure 3: Raw load signal and wavelet-filtered feature under normal conditions.

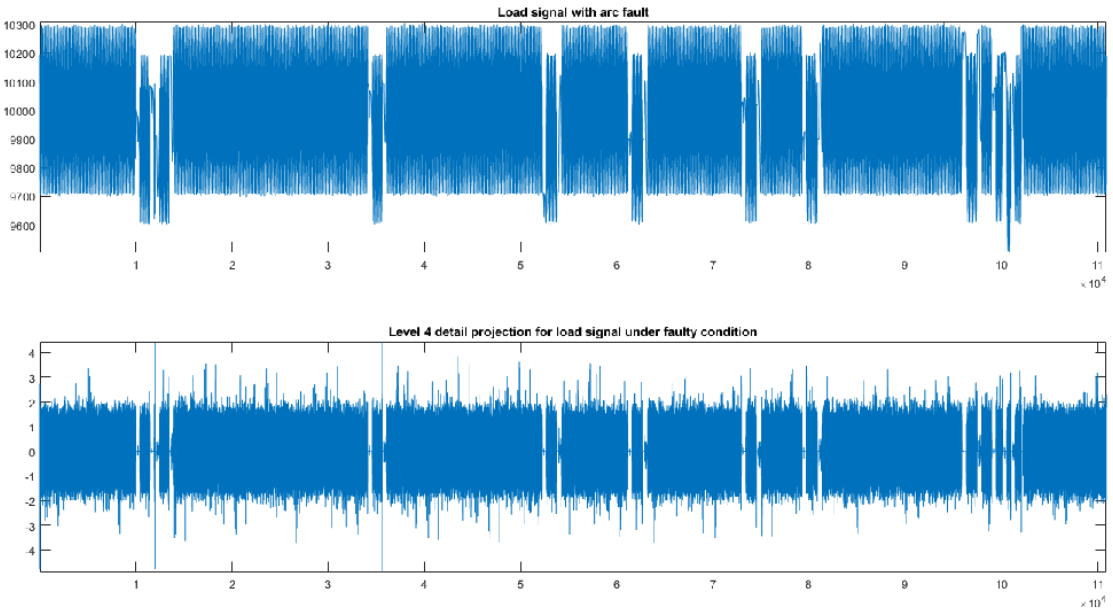


Figure 4: Raw load signal and wavelet-filtered feature under faulty conditions.

Model Training

The load signal under normal conditions is specified in the `normal.mat` file in the example folder. Use the normal load signal to obtain the training features that are used as the input of the autoencoder. Train the autoencoder using these layers and training options.

```
% Training data: load voltage under normal condition
featureDimension = 100;

% Create network layers
numHiddenUnits = 50;

layers = [ sequenceInputLayer(1, 'Name', 'in')
    bilstmLayer(32, 'Name', 'bilstm1')
    reluLayer('Name', 'relu1')
    bilstmLayer(16, 'Name', 'bilstm2')
    reluLayer('Name', 'relu2')
    bilstmLayer(32, 'Name', 'bilstm3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(1, 'Name', 'fc')
    regressionLayer('Name', 'out') ];

% Set options
options = trainingOptions('adam', ...
    'MaxEpochs',20, ...
    'MiniBatchSize',16, ...
    'Plots','training-progress');
```

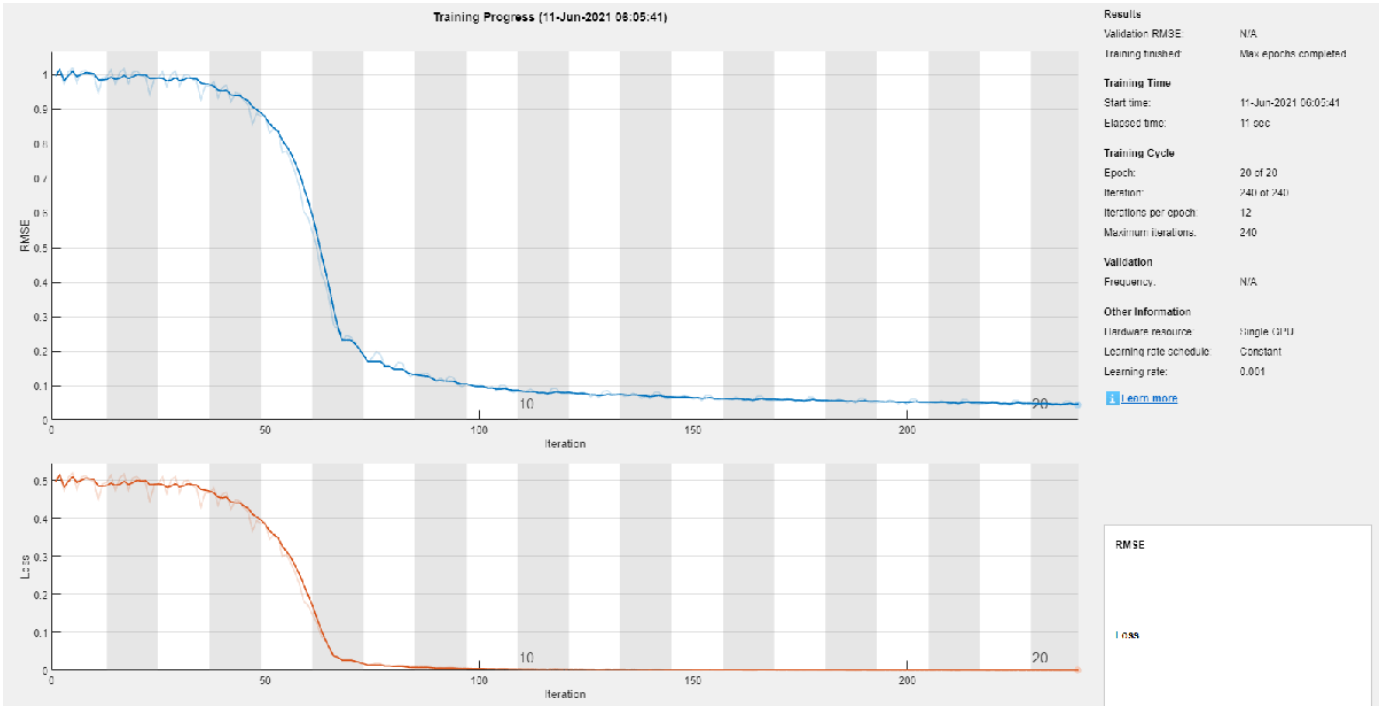


Figure 5: Training progress for the autoencoder.

The trained network is specified in the file `netData.mat` in the example folder. The trained network can be uploaded into the `Predict` block from Deep Learning Toolbox™ to detect the arc fault in the

load signal. The figure shows the histogram for the reconstruction error produced by the autoencoder when the input is the training data. You can use the statistics for the reconstruction error to choose the detection threshold. For instance, choose the detection threshold to be three times the standard deviation for the reconstruction error.

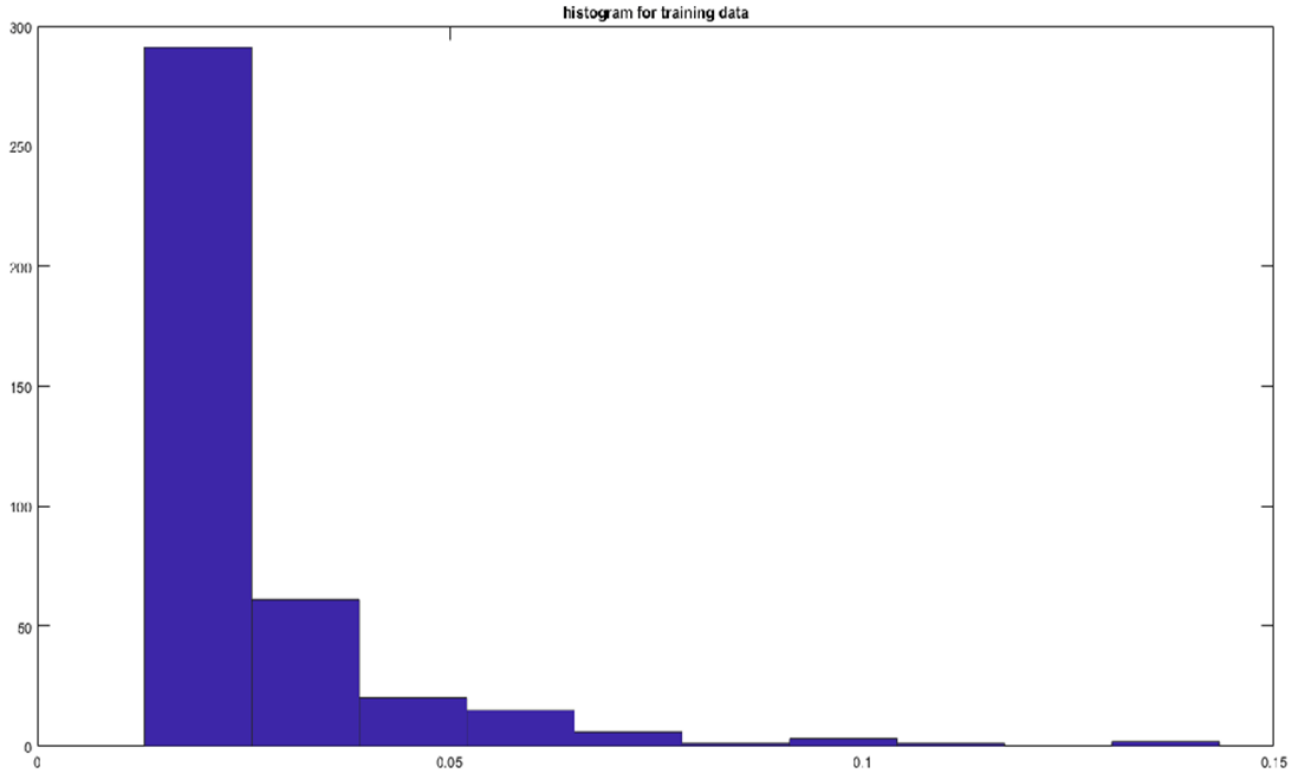


Figure 6: Histogram for the reconstruction error produced by the autoencoder when the input is the training data.

Model for Anomaly Detection Using Autoencoder

The `DCArcModelFinal` model is used for real-time detection of the arc fault in a DC load signal. Before running the model, you must specify the simulation stop time in seconds in the workspace variable `t`.

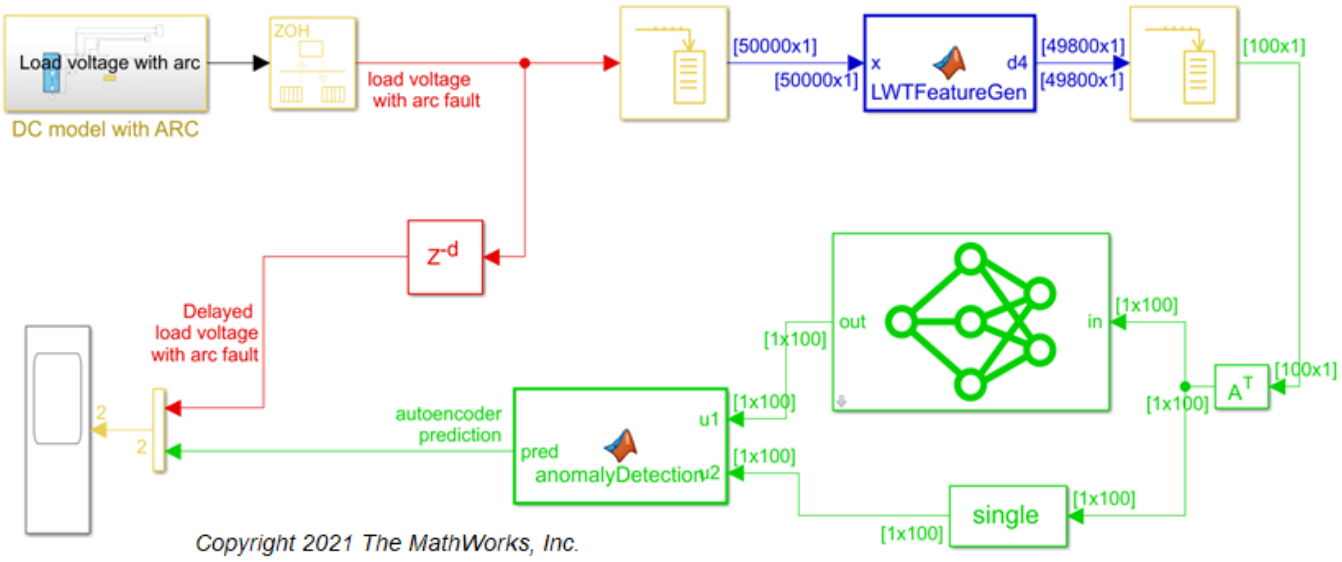


Figure 7: `DCArcModelFinal` for arc fault detection.

The first block generates noisy DC load signal with arc fault in continuous time. The load voltage is then converted into a discrete-time signal sampled at 20kHz by the `Rate transition` block in DSP System Toolbox™. The discrete time signal is then buffered to the `LWTFeatureGen` block that obtains the desired level 4 detail projection after preprocessing. The detail projection is then segmented in 100 sample frames that are the test features for the `Predict` block. The `Predict` block has been preloaded with the network pretrained using the load signal under normal conditions. The anomaly detection block then calculates the root-mean-square error (RMSE) for each frame and declares the presence of an arc fault if the error is above some predefined threshold. We can observe the prediction from the block adding a scope to the model. The red line in the plot indicates where the transition due to anomaly is detected. The autoencoder identified most of the arc faults.

Figure 8: Real-time arc fault detection as seen from scope.

This plot shows the regions predicted by the network when the wavelet-filtered features are used. The autoencoder was able to detect all 10 arc fault regions correctly. In other words, we obtained a 100% probability of detection in this case.

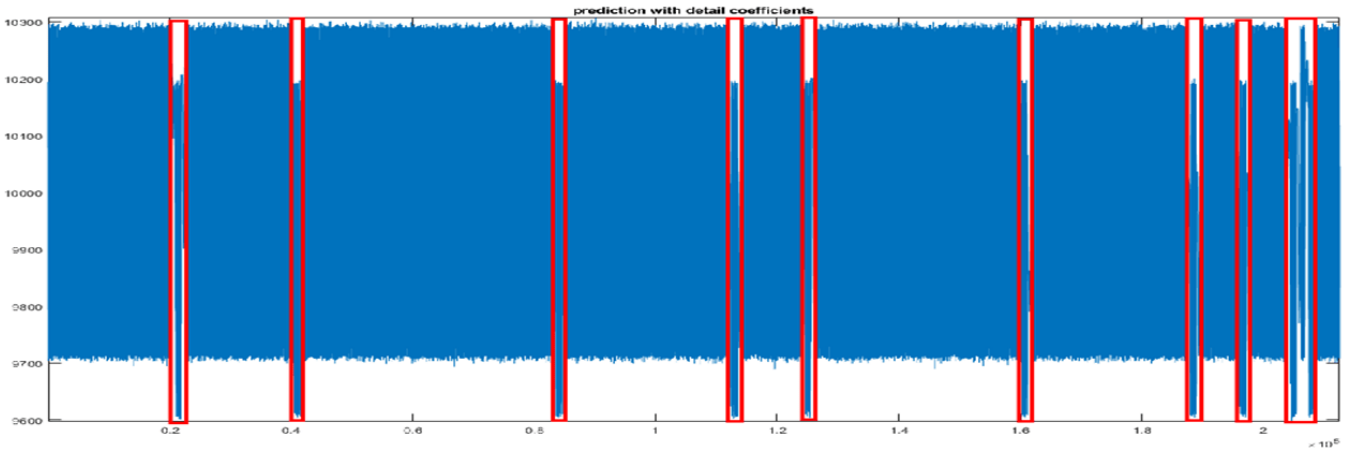


Figure 9: Detection performance for the autoencoder using wavelet-filtered features.

This plot shows the anomaly detection performance of the raw data trained autoencoder. When we used raw data for anomaly detection, the encoder was able to identify seven out of 10 regions correctly.

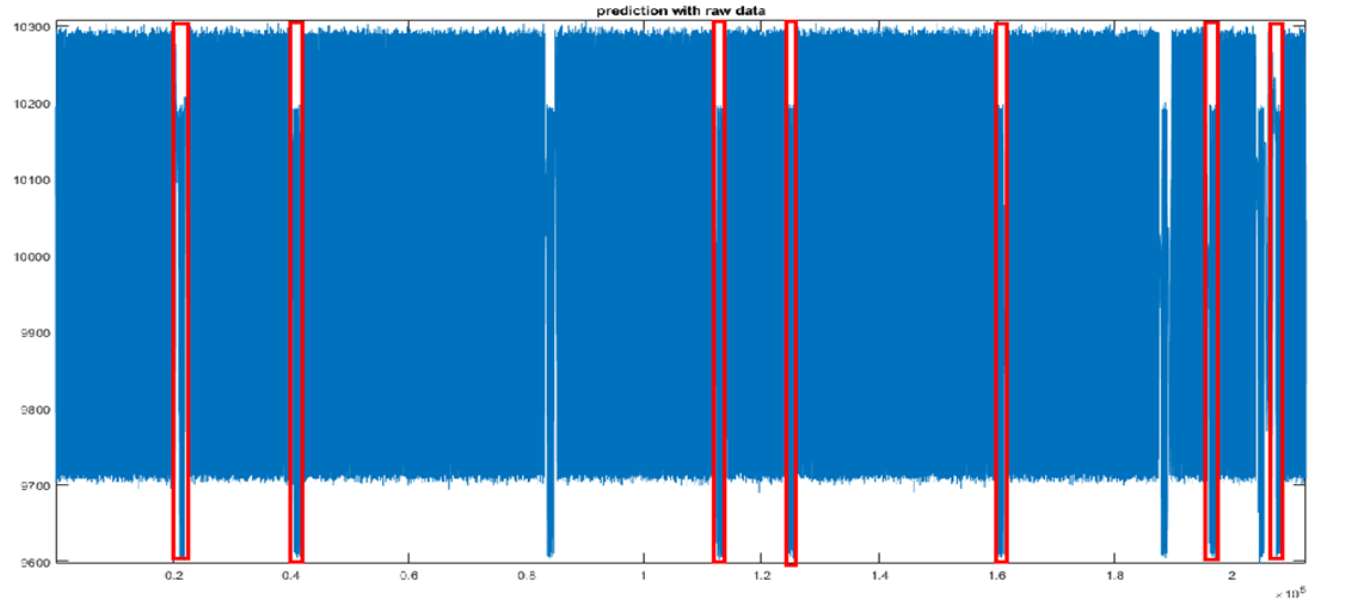


Figure 10: Detection performance for the autoencoder using raw load signal.

Finally, we generated a 50 second long anomalous signal with 40 arc fault regions (this data is not included with the example). When tested with the autoencoder trained with raw signals, the arc regions were detected with a 57.85% probability of detection. In contrast, the autoencoder trained with the wavelet-filtered signals, was able to detect the arc fault regions with a 97.52% probability of detection.

Summary

In this example, we demonstrated how autoencoders can be used to identify arc faults in DC systems. Both the raw and wavelet filtered load signals under normal conditions can be used as features to train the autoencoders. These anomaly detection mechanisms can be used to detect arc faults in a timely manner and thus protect a DC system from damages caused by the faults.

References

[1] Wang, Zhan, and Robert S. Balog. “Arc Fault and Flash Signal Analysis in DC Distribution Systems Using Wavelet Transformation.” *IEEE Transactions on Smart Grid* 6, no. 4 (July 2015): 1955–63. `https://doi.org/10.1109/TSG.2015.2407868`

See Also

Functions
`lwt` | `ilwt` | `lwtcoef`

Objects
`liftingScheme`

Related Examples

- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” (Wavelet Toolbox)

Fault Detection Using Wavelet Scattering and Recurrent Deep Networks

This example shows how to classify faults in acoustic recordings of air compressors using a wavelet scattering network paired with a recurrent neural network. The example provides the opportunity to use a GPU to accelerate the computation of the wavelet scattering transform. If you wish to utilize a GPU, you must have Parallel Computing Toolbox™ and a supported GPU. See “GPU Support by Release” (Parallel Computing Toolbox) for details.

Dataset

The dataset consists of acoustic recordings collected on a single stage reciprocating type air compressor [1 on page 12-0]. The data are sampled at 16 kHz. Specifications of the air compressor are as follows:

- Air Pressure Range: 0-500 lb/m2, 0-35 Kg/cm2
- Induction Motor: 5HP, 415V, 5Am, 50 Hz, 1440rpm
- Pressure Switch: Type PR-15, Range 100-213 PSI

Each recording represents one of 8 states which includes the healthy state and 7 faulty states. The 7 faulty states are:

1. Leakage inlet valve (LIV) fault
2. Leakage outlet valve (LOV) fault
3. Non-return valve (NRV) fault
4. Piston ring fault
5. Flywheel fault
6. Rider belt fault
7. Bearing fault

Download the dataset and unzip the data file in a folder where you have write permission. This example assumes you are downloading the data in the temporary directory designated as `tempdir` in MATLAB®. If you chose to use a different folder, substitute that folder for `tempdir` in the following. The recordings are stored as .wav files in folders named for their respective state.

```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'AirCompressorDataSet');
if ~exist(fullfile(tempdir,'AirCompressorDataSet'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'AirCompressorDataSet'))
end
```

Use an `audioDatastore` to manage data access. Each subfolder contains only recordings of the designated class. Use the folder names as the class labels.

```
datasetLocation = fullfile(tempdir,'AirCompressorDataSet','AirCompressorDataset');
ads = audioDatastore(datasetLocation,'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

Examine the number of examples in each class. There are 225 recordings in each class.

```
countcats(ads.Labels)
```

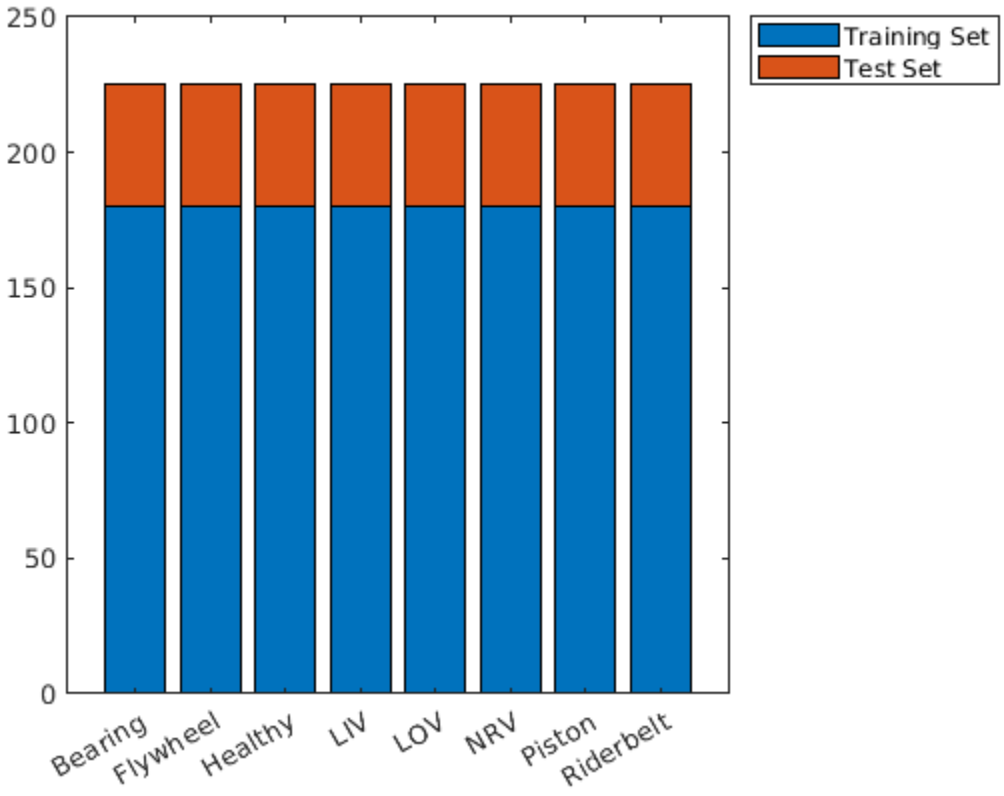
```
ans = 8×1

   225
   225
   225
   225
   225
   225
   225
   225
```

Split the data into training and test sets. Use 80% of the data for training and hold out the remaining 20% for testing. Shuffle the data once before splitting.

```
rng default
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8,0.2);
```

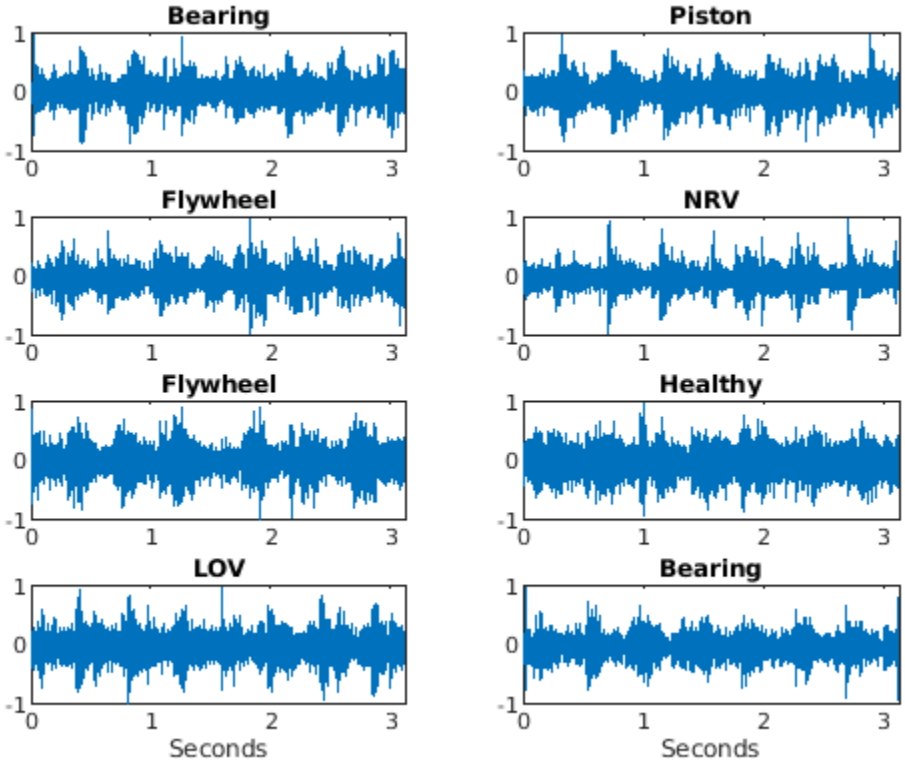
Verify that the number of examples in each class is the expected number.

```
uniqueLabels = unique(adsTrain.Labels);
tblTrain = countEachLabel(adsTrain);
tblTest = countEachLabel(adsTest);
H = bar(uniqueLabels,[tblTrain.Count, tblTest.Count],'stacked');
legend(H,["Training Set","Test Set"],'Location','NorthEastOutside')
```



Select some random examples from the training set for plotting.

```
idx = randperm(numel(adsTrain.Files),8);
Fs = 16e3;
for n = 1:numel(idx)
    x = audioread(adsTrain.Files{idx(n)});
    t = (0:size(x,1)-1)/Fs;
    subplot(4,2,n);
    plot(t,x);
    if n == 7 || n == 8
        xlabel('Seconds');
    end
    title(string(adsTrain.Labels(idx(n))));
end
```



Wavelet Scattering Network

Each record has 50,000 samples sampled at 16 kHz. Construct a wavelet scattering network based on the data characteristics. Set the invariance scale to be 0.5 seconds.

```
N = 5e4;
Fs = 16e3;
IS = 0.5;
sn = waveletScattering('SignalLength',N,'SamplingFrequency',Fs,...
    'InvarianceScale',0.5);
```

With these network settings, there are 330 scattering paths and 25 time windows per example. You can see this with the following code.

```
[~,npaths] = paths(sn);
Ncfs = numCoefficients(sn);
sum(npaths)
```

```
ans = 330
```

```
Ncfs
```

```
Ncfs = 25
```

Note this already represents a 6-fold reduction in the size of the data for each record. We reduced the data size from 50,000 samples to 8250 in total. Most importantly, we reduced the size of the data along the time dimension from 50,000 to 25 samples. This is crucial for our use of a recurrent network. Attempting to use a recurrent network on the original data with 50,000 samples would immediately result in memory problems.

Obtain the wavelet scattering features for the training and test sets. If you have a suitable GPU and Parallel Computing Toolbox, you can set `useGPU` to `true` to accelerate the scattering transform. The function `helperBatchScatFeatures` obtains the scattering transform of each example.

```
batchsize = 64;
useGPU = false;
scTrain = [];
while hasdata(adsTrain)
    sc = helperBatchScatFeatures(adsTrain,sn,N,batchsize,useGPU);
    scTrain = cat(3,scTrain,sc);
end
```

Repeat the process for the held out test set.

```
scTest = [];
while hasdata(adsTest)
   sc = helperBatchScatFeatures(adsTest,sn,N,batchsize,useGPU);
   scTest = cat(3,scTest,sc);
end
```

Remove the 0-th order scattering coefficients. For both the training and test sets, put each 330-by-25 scattering transform into an element of a cell array for use in training and testing the recurrent network.

```
TrainFeatures = scTrain(2:end,:,:);
TrainFeatures = squeeze(num2cell(TrainFeatures,[1 2]));
YTrain = adsTrain.Labels;
TestFeatures = scTest(2:end,:,:);
TestFeatures = squeeze(num2cell(TestFeatures,[1 2]));
YTest = adsTest.Labels;
```

Define Network

Recall there are 1440 training examples and 360 test set examples. Accordingly the `TrainFeatures` and `TestFeatures` cell arrays have 1440 and 360 elements respectively.

Use the number of scattering paths as the number of features. Create a recurrent network with a single LSTM layer having 512 hidden units. Follow the LSTM layer with a fully connected layer and finally a softmax layer. Use 'zscore' normalization across all scattering paths at the input to the network.

```
[inputSize, ~] = size(TrainFeatures{1});
```

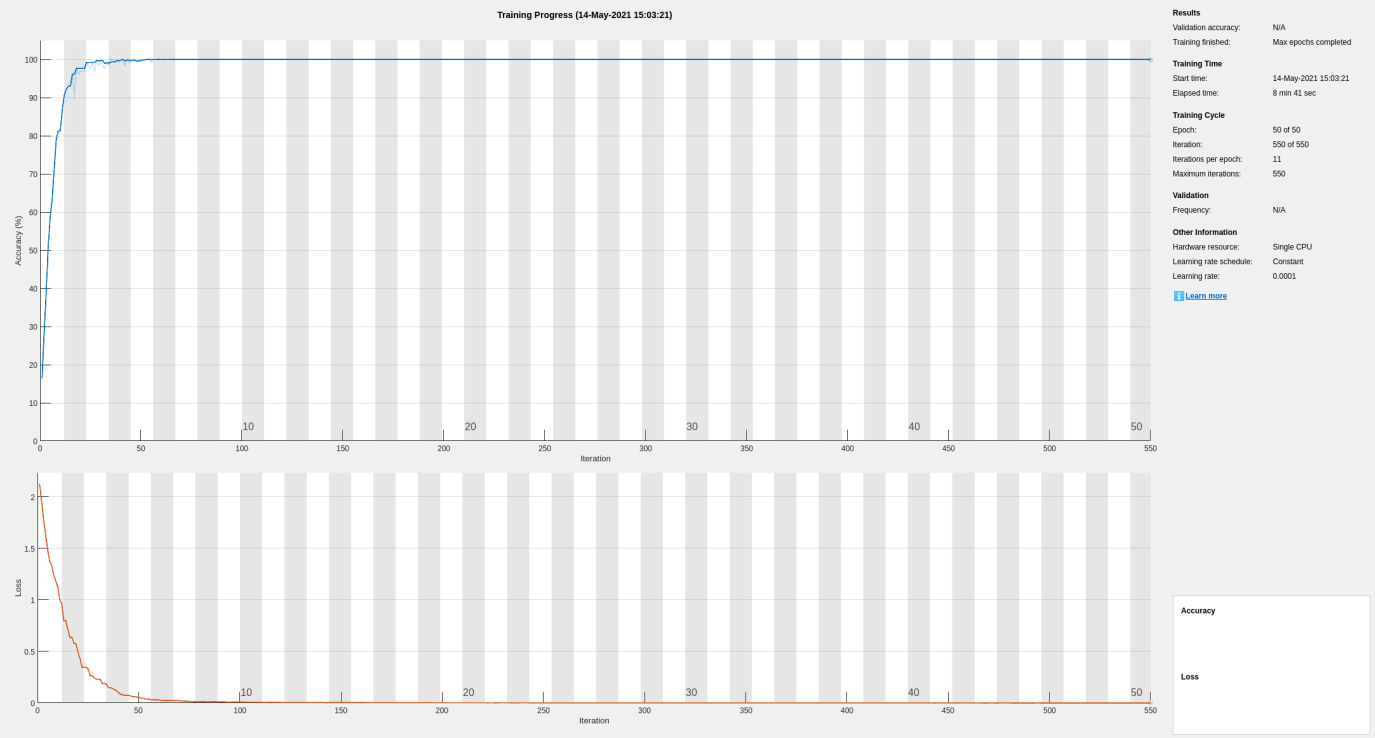
```
numHiddenUnits = 512;
numClasses = numel(unique(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize,'Normalization','zscore')
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Train Network

Train the network for 50 epochs with a mini batch size of 128. Use an Adam optimizer with an initial learn rate of 1e-4. Shuffle the data each epoch.

```
maxEpochs = 50;
miniBatchSize = 128;


options = trainingOptions('adam', ...
    'InitialLearnRate',1e-4,...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','shortest', ...
    'Shuffle','every-epoch',...
    'Plots','training-progress',...
    'Verbose',true);

net = trainNetwork(TrainFeatures,YTrain,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:08 |       16.41% |       2.1281 |      1.0000e-04 |
|       5 |          50 |       00:00:59 |      100.00% |       0.0536 |      1.0000e-04 |
|      10 |         100 |       00:01:51 |      100.00% |       0.0072 |      1.0000e-04 |
|      14 |         150 |       00:02:40 |      100.00% |       0.0064 |      1.0000e-04 |
|      19 |         200 |       00:03:27 |      100.00% |       0.0025 |      1.0000e-04 |
|      23 |         250 |       00:04:14 |      100.00% |       0.0015 |      1.0000e-04 |
|      28 |         300 |       00:05:00 |      100.00% |       0.0012 |      1.0000e-04 |
|      32 |         350 |       00:05:45 |      100.00% |       0.0007 |      1.0000e-04 |
|      37 |         400 |       00:06:29 |      100.00% |       0.0006 |      1.0000e-04 |
|      41 |         450 |       00:07:12 |      100.00% |       0.0005 |      1.0000e-04 |
|      46 |         500 |       00:07:55 |      100.00% |       0.0005 |      1.0000e-04 |
|      50 |         550 |       00:08:41 |      100.00% |       0.0004 |      1.0000e-04 |
|========================================================================================|
Training finished: Reached final iteration.
```



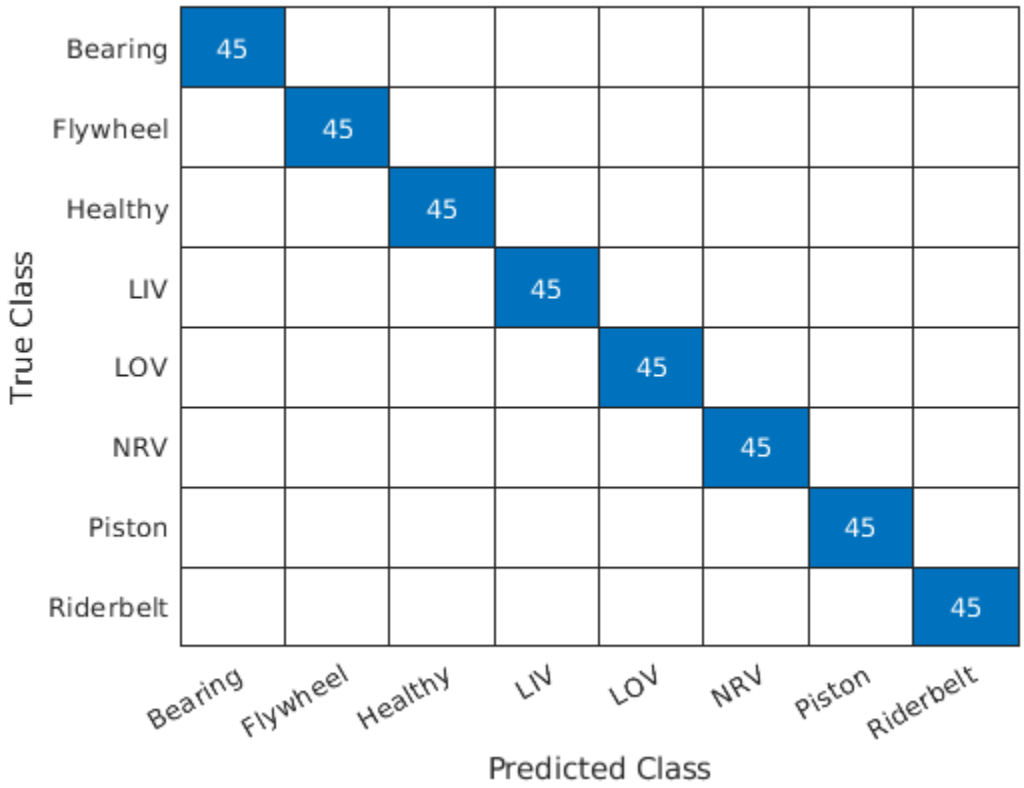
In training, the network has achieved near perfect performance. In order to ensure we have not overfit to the training data, use the held-out test set to determine how well our network generalizes to unseen data.

```
YPred = classify(net,TestFeatures);
accuracy = 100*sum(YPred == YTest) / numel(YTest)
```

```
accuracy = 100
```

In this case, we see that the performance on the held-out test set is also excellent.

```
figure
confusionchart(YTest, YPred)
```



Summary

In this example, the wavelet scattering transform was used with a simple recurrent network to classify faults in an air compressor. The scattering transform allowed us to extract robust features for our learning problem. Additionally, the data reduction achieved along the time dimension of the data by the use of the wavelet scattering transform was critical in order to create a computationally feasible problem for our recurrent network.

References

[1] Verma, Nishchal K., Rahul Kumar Sevakula, Sonal Dixit, and Al Salour. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability* 65, no. 1 (March 2016): 291–309. https://doi.org/10.1109/TR.2015.2459684.

helperbatchscatfeatures - This function returns the wavelet time scattering feature matrix for a given input signal. If `useGPU` is set to `true`, the scattering transform is computed on the GPU.

```
function sc = helperBatchScatFeatures(ds,sn,N,batchsize,useGPU)
% This function is only intended to support examples in the Wavelet
% Toolbox. It may be changed or removed in a future release.

% Read batch of data from audio datastore
batch = helperReadBatch(ds,N,batchsize);
if useGPU
    batch = gpuArray(batch);
end
```

```
% Obtain scattering features
S = sn.featureMatrix(batch,'transform','log');
gather(batch);
S = gather(S);

% Subsample the features
%sc = S(:,1:6:end,:);
sc = S;
end
```

helperReadBatch - This function reads batches of a specified size from a datastore and returns the output in single precision. Each column of the output is a separate signal from the datastore. The output may have fewer columns than the batch size if the datastore does not have enough records.

```
function batchout = helperReadBatch(ds,N,batchsize)
% This function is only in support of Wavelet Toolbox examples. It may
% change or be removed in a future release.
%
% batchout = readReadBatch(ds,N,batchsize) where ds is the Datastore and
%   ds is the Datastore
%   batchsize is the batchsize

kk = 1;

while(hasdata(ds)) && kk <= batchsize
    tmpRead = read(ds);
    batchout(:,kk) = cast(tmpRead(1:N),'single'); %#ok<AGROW>
    kk = kk+1;
end

end
```


See Also

`waveletScattering`

Related Examples

- “Air Compressor Fault Detection Using Wavelet Scattering” (Wavelet Toolbox)

More About

- “Wavelet Scattering” (Wavelet Toolbox)

Parasite Classification Using Wavelet Scattering and Deep Learning

This example shows how to classify parasitic infections in Giemsa stain images using wavelet image scattering and deep learning. The dataset is challenging for deep networks because it contains only 48 images. The images are divided evenly into three categories of parasitic infections: babesiosis, plasmodium-gametocyte, and trypanosomiasis.

Data

Obtain the data from the MATLAB® file exchange: Deploying Deep Neural Networks to Embedded GPUs and unzip the file. The file is in the same folder as this example.

```
url = "https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/" + ...
    "5918495a-0009-419e-8e10-77b06e3fe553/844e43fa-7c50-4f88-a435-f0afe04fc3a3/" + ...
    "packages/zip";
websave("classifyBloodSmearImages.zip",url);
unzip('classifyBloodSmearImages.zip')
```

Create an `ImageDatastore` to manage the access of the Giemsa stain images. The images are in RGB format with a common size of 300-by-300-by-3.

```
imagedir = fullfile('classifyBloodSmearImages','BloodSmearImages');
Imds = imageDatastore(imagedir,'IncludeSubFolders',true,'FileExtensions',...
    '.jpg','LabelSource','foldernames');
summary(Imds.Labels)
```

```
     babesiosis                16
     plasmodium-gametocyte     16
     trypanosomiasis           16
```

There are 16 images for each of the three parasite types. Split the data into training and hold-out test sets, with 70 percent of the images in the training set and 30 percent in the test set. Set the random number generator for reproducibility.

```
rng default
[trainImds,testImds] = splitEachLabel(Imds,0.7);
```

Verify that equal numbers of each parasite class are contained in both the training and test sets.

```
summary(trainImds.Labels)
```

```
     babesiosis                11
     plasmodium-gametocyte     11
     trypanosomiasis           11
```

```
% Perform the same for the test set.
summary(testImds.Labels)
```

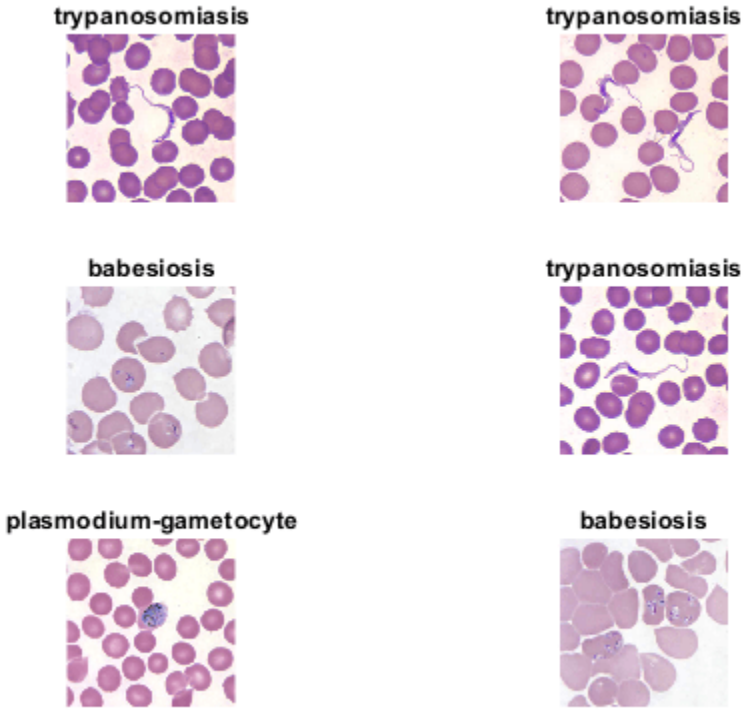
```
     babesiosis                5
     plasmodium-gametocyte     5
     trypanosomiasis           5
```

Because this is a small dataset, the entire training and test sets fit in memory. Read all images for both sets.

```
trainImages = readall(trainImds);
testImages = readall(testImds);
```

Plot some sample images from the training data.

```
idx = randperm(33,6);
figure
for ii = 1:length(idx)
    im = trainImages{idx(ii)};
    subplot(3,2,ii)
    imshow(im,[])
    title(string(trainImds.Labels(idx(ii))));
end
```



Wavelet Scattering Network

In this example, you use a wavelet scattering transform as the feature extractor for the machine learning approaches. The wavelet scattering transform helps to reduce the dimensionality of the data and increase the interclass dissimilarity. Construct a two-layer image scattering network with a 40-by-40 pixel invariance scale. Use two wavelets per octave in the first layer and one wavelet per octave in the second layer. Use two rotations of the wavelets per layer.

```
sn = waveletScattering2('ImageSize',[300 300],'InvarianceScale',40,...
    'QualityFactors',[2 1],'NumRotations',[2 2]);
[~,npaths] = paths(sn);
sum(npaths)
```

```
ans = 27
```

```
coefficientSize(sn)
```

```
ans = 1×2

    38    38
```

The specified wavelet scattering network has 27 paths. The image on each scattering path is reduced to 38-by-38-by-3. Even without further averaging of the scattering coefficients, this is a reduction in the size of each image's memory by more than a factor of 2. However, for classification we form a feature vector that averages the scattering coefficients over the spatial and channel dimensions. This results in feature vectors with only 27 elements, a real-valued scalar for each scattering path. This represents a reduction in the number of elements by a factor of 10,000 for each image.

The following code computes the wavelet scattering feature vectors for both the training and test sets. Concatenate the feature vectors so that you have *N*-by-27 matrices, where *N* is the number of examples in the training or test set and each row is a wavelet scattering feature vector for an example.

```
trainfeatures = cellfun(@(x)helperScatImages_mean(sn,x),trainImages,'Uni',0);
testfeatures = cellfun(@(x)helperScatImages_mean(sn,x),testImages,'Uni',0);
trainfeatures = cat(1,trainfeatures{:});
testfeatures = cat(1,testfeatures{:});
```

SVM Classification

Use an SVM classifier with the scattering features. Choose a cubic polynomial kernel. Use a one-vs-all coding scheme.

```
template = templateSVM(...
    'KernelFunction', 'polynomial', ...
    'PolynomialOrder', 3, ...
    'KernelScale', 1, ...
    'BoxConstraint', 314, ...
    'Standardize', true);
classificationSVM = fitcecoc(trainfeatures,trainImds.Labels,...
    'Learners', template, 'Coding', 'onevsall');
```

Estimate the accuracy on the training set using cross-validation with 5 folds.

```
kfoldmodel = crossval(classificationSVM, 'KFold', 5);
loss = kfoldLoss(kfoldmodel)*100;
crossvalAccuracy = 100-loss
```

```
crossvalAccuracy = single
    81.8182
```

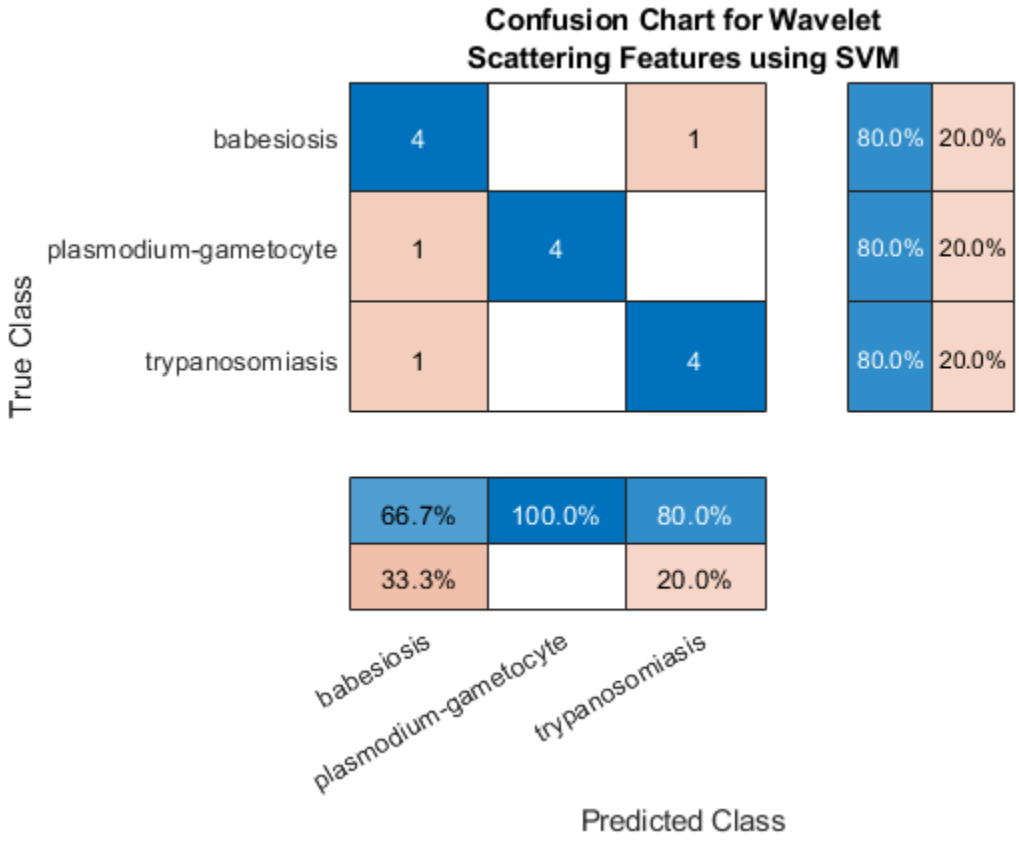
The cross-validation accuracy is approximately 80 percent. Now examine the accuracy on the held-out test set and plot the confusion chart.

```
[predLabels,scores] = predict(classificationSVM,testfeatures);
testAccuracy = ...
    sum(categorical(predLabels)== testImds.Labels)/numel(testImds.Labels)*100
```

```
testAccuracy = 80
```

```
figure
cchart = confusionchart(testImds.Labels,predLabels);
```

```
cchart.Title = ...
    {'Confusion Chart for Wavelet' ; 'Scattering Features using SVM'};
cchart.RowSummary = 'row-normalized';
cchart.ColumnSummary = 'column-normalized';
```



The overall test accuracy is 80 percent with the SVM model. The recall for each class is 80%. The precision is also good for the plasmodium-gametocyte and trypanosomiasis parasites, but worse for babesiosis. Examine the F1 scores for each class.

```
f1SVM = f1score(cchart.NormalizedValues);
disp(f1SVM)
```

```
                            F1
                          _______

    babesiosis            0.72727
    plasmodium-gametocyte 0.88889
    trypanosomiasis           0.8
```

All F1 scores are between approximately 0.7 and 0.9.

PCA classifier with scattering features

Support vector machines are powerful techniques for features that are not linearly separable, but they are designed for binary classification and may be suboptimal for multiclass problems. Here you complement the SVM analysis by using a simple PCA (linear) classifier with the same wavelet scattering features. The `helperPCAModel` function determines the `numcomp` eigenvectors

corresponding to the largest eigenvalues of the covariance matrix of the wavelet scattering features for each pathogen in the training set along with the class means.

`helperPCAClassifier` classifies each test sample. It does this by subtracting the model class means from each wavelet scattering feature vector in the test dataset and projecting the centered feature vectors onto the covariance-matrix eigenvectors for each class in the model. `helperPCAClassifier` assigns each test example to the pathogen with the smallest error, or residual. This is a principal components analysis (PCA) classifier.

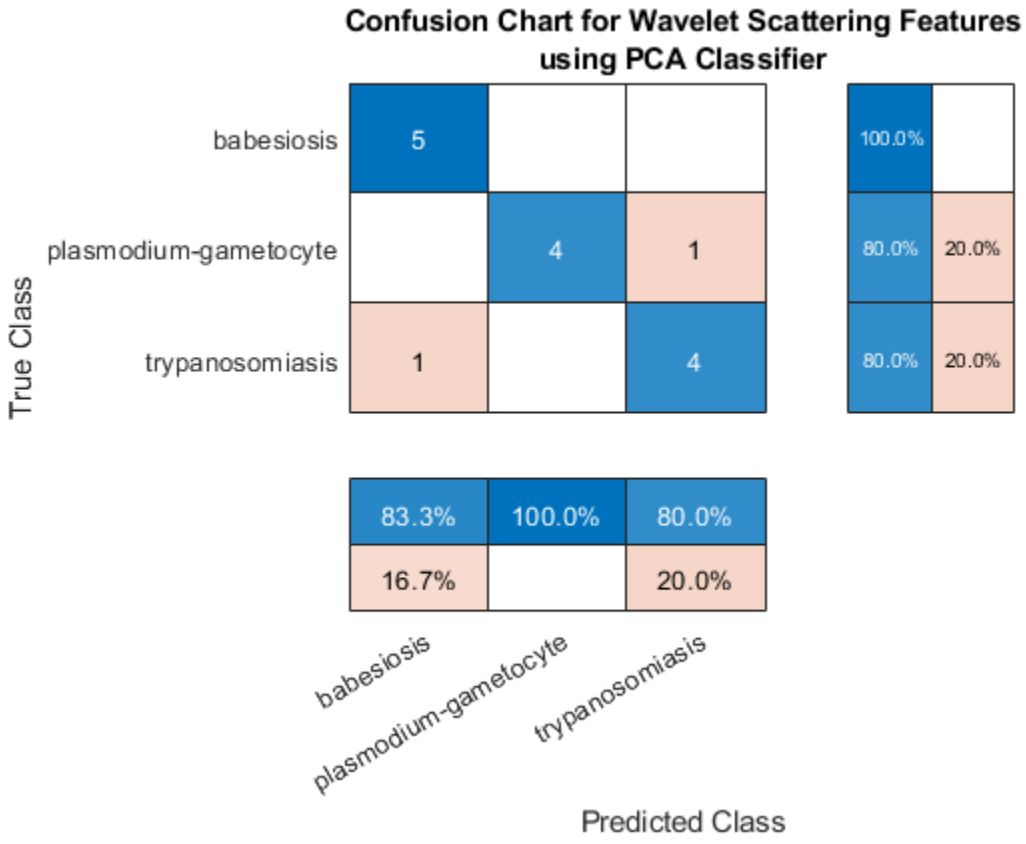
Remove the 0-th order scattering features from each feature vector. Set the number of principal components (eigenvectors) to 6.

```
numcomp = 6;
model = helperPCAModel(trainfeatures(:,2:end)',numcomp,trainImds.Labels);
PCALabels = helperPCAClassifier(testfeatures(:,2:end)',model);
testPCAacc = sum(PCALabels==testImds.Labels)/numel(testImds.Labels)*100
```

```
testPCAacc = 86.6667
```

The test accuracy is approximately 87% with the PCA classifier. Plot the confusion chart and calculate the F1 scores for each class.

```
figure
cchart = confusionchart(testImds.Labels,PCALabels);
cchart.Title = {'Confusion Chart for Wavelet Scattering Features' ; ...
    'using PCA Classifier'};
cchart.RowSummary = 'row-normalized';
cchart.ColumnSummary = 'column-normalized';
```



```
f1PCA = f1score(cchart.NormalizedValues);
disp(f1PCA)
```

```
                              F1
                            _______

    babesiosis              0.90909
    plasmodium-gametocyte   0.88889
    trypanosomiasis             0.8
```

The F1 scores for the PCA classifier with wavelet scattering features are quite strong, with all scores between 0.8 and 1.

Convolutional Deep Network

In this section, you attempt the same classification using deep convolutional networks. Deep networks provide state-of-art results for classification problems with large datasets and are capable of learning complicated nonlinear mappings, but their performance often suffers in small datasets. To mitigate this problem, use an image augmenter. `imageDataAugmenter` perturbs the data in each epoch, in effect creating new training examples.

```
augmenter = imageDataAugmenter('RandRotation',[0 180],'RandXTranslation', [-5 5], ...
    'RandYTranslation',[-5 5]);
augimds = augmentedImageDatastore([300 300 3],trainImds,'DataAugmentation',augmenter);
```

Define a small CNN consisting of two convolution layers followed by batch normalization layers and RELU activations. Follow the final RELU activation with max pooling, fully connected, and softmax layers.

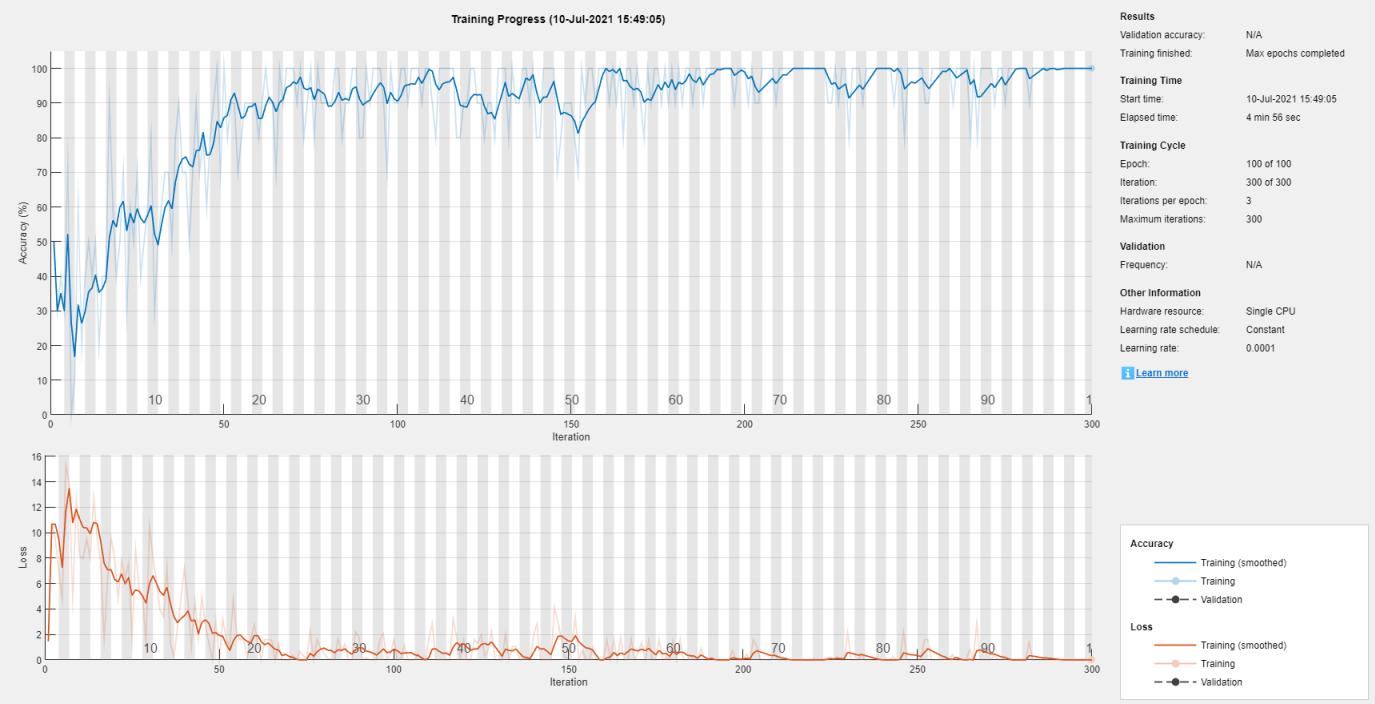
```
layers = [
    imageInputLayer([300 300 3])
    convolution2dLayer(7,16)
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,20)
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(4)
    fullyConnectedLayer(3)
    softmaxLayer
    classificationLayer];
```

Use stochastic gradient descent with a minibatch size of 10. Shuffle the data each epoch. Run the training for 100 epochs.

```
opts = trainingOptions('sgdm',...
    'InitialLearnRate',  0.0001, ...
    'MaxEpochs', 100, ...
    'MiniBatchSize',10,...
    'Shuffle','every-epoch',...
    'Plots', 'training-progress',...
    'Verbose',false,...
    'ExecutionEnvironment','cpu');
```

Train the network.

```
trainedNet = trainNetwork(augimds,layers,opts);
```

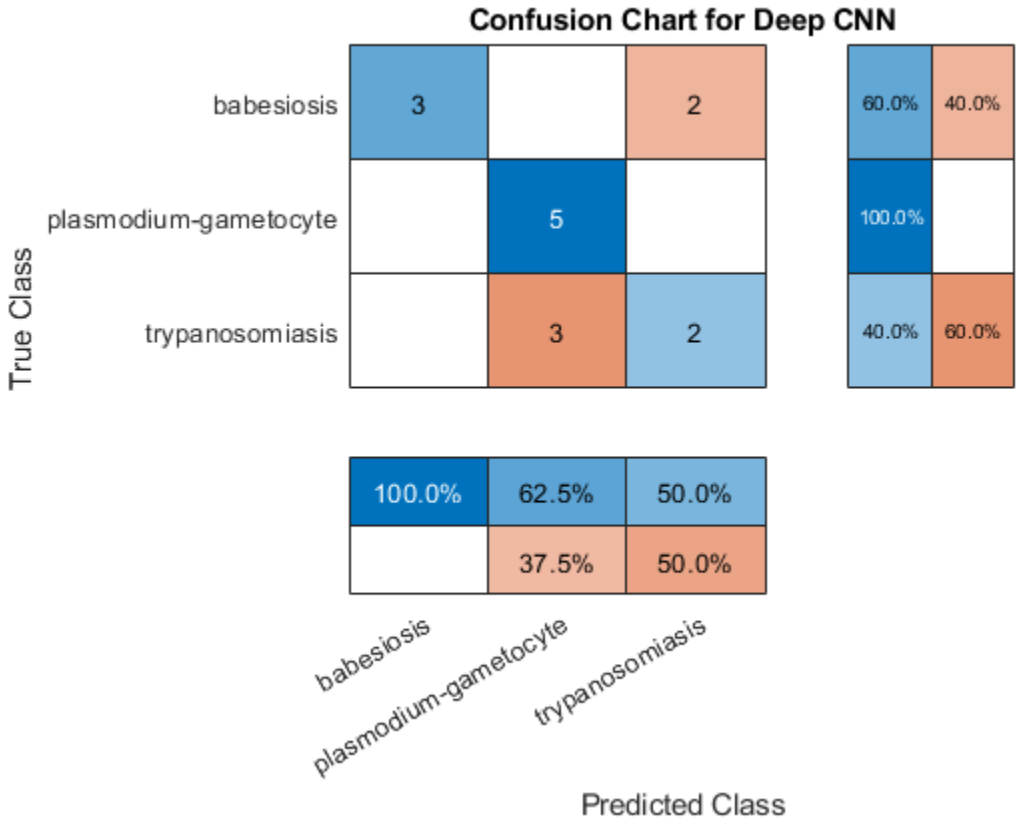


Examine the performance of the network on the held-out test set.

```
ypred = trainedNet.classify(testImds);
cnnAccuracy = sum(ypred == testImds.Labels)/numel(testImds.Labels)*100
```

```
cnnAccuracy = 66.6667
```

```
figure
cchart = confusionchart(testImds.Labels,ypred);
cchart.Title = 'Confusion Chart for Deep CNN';
cchart.RowSummary = 'row-normalized';
cchart.ColumnSummary = 'column-normalized';
```




```
f1CNN = f1score(cchart.NormalizedValues);
disp(f1CNN)
```

```
                            F1
                          _______

    babesiosis              0.75
    plasmodium-gametocyte  0.76923
    trypanosomiasis        0.44444
```

In spite of using an augmented dataset for training, the CNN has overfit the training set and the F1 scores are significantly worse than either the SVM or PCA model with the wavelet scattering features.

Next, use transfer learning with SqueezeNet. Modify the final convolutional layer to accommodate the fact that you have three classes of pathogens. SqueezeNet was constructed to recognize 1,000 classes.

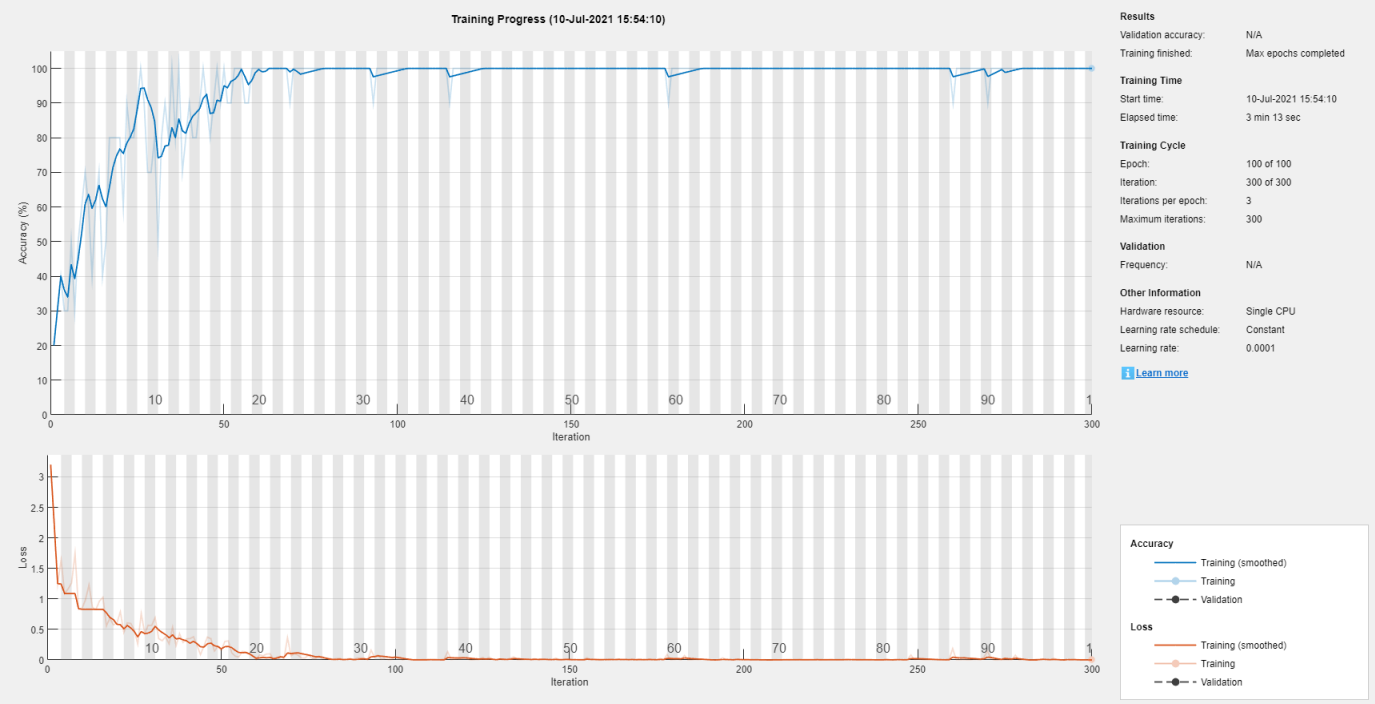
```
net = squeezenet;
lgraphSQZ = layerGraph(net);
numClasses = numel(categories(trainImds.Labels));
oldFinalConv = lgraphSQZ.Layers(end-4);
newFinalConv = convolution2dLayer(1,numClasses, ...
        'Name','new_conv');
setLearnRateFactor(newFinalConv,'Weights',10);
setLearnRateFactor(newFinalConv,'Bias',10)
```

```
ans =
  Convolution2DLayer with properties:

             Name: 'new_conv'

   Hyperparameters
        FilterSize: [1 1]
       NumChannels: 'auto'
        NumFilters: 3
            Stride: [1 1]
    DilationFactor: [1 1]
       PaddingMode: 'manual'
       PaddingSize: [0 0 0 0]
      PaddingValue: 0

   Learnable Parameters
           Weights: []
              Bias: []

  Show all properties
```

```
lgraphSQZ = replaceLayer(lgraphSQZ,oldFinalConv.Name,newFinalConv);
oldClassLayer= lgraphSQZ.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSQZ = replaceLayer(lgraphSQZ,oldClassLayer.Name,newClassLayer);
```

Reset the training and test datastores. Modify the datastore read function to resize images to be compatible with SqueezeNet, which expects 227-by-227-by-3 images. Set up the image augmenter and train the network.

```
reset(trainImds);
reset(testImds);
trainImds.ReadFcn = @(x)imresize(imread(x),'OutputSize',[227 227]);
testImds.ReadFcn = @(x)imresize(imread(x),'OutputSize',[227 227]);
augmenter = imageDataAugmenter('RandRotation',[0 180],'RandXTranslation', [-5 5], ...
    'RandYTranslation',[-5 5]);
augimds = augmentedImageDatastore([227 227 3],trainImds,...
    'DataAugmentation',augmenter);
trainedNet = trainNetwork(augimds,lgraphSQZ,opts);
```

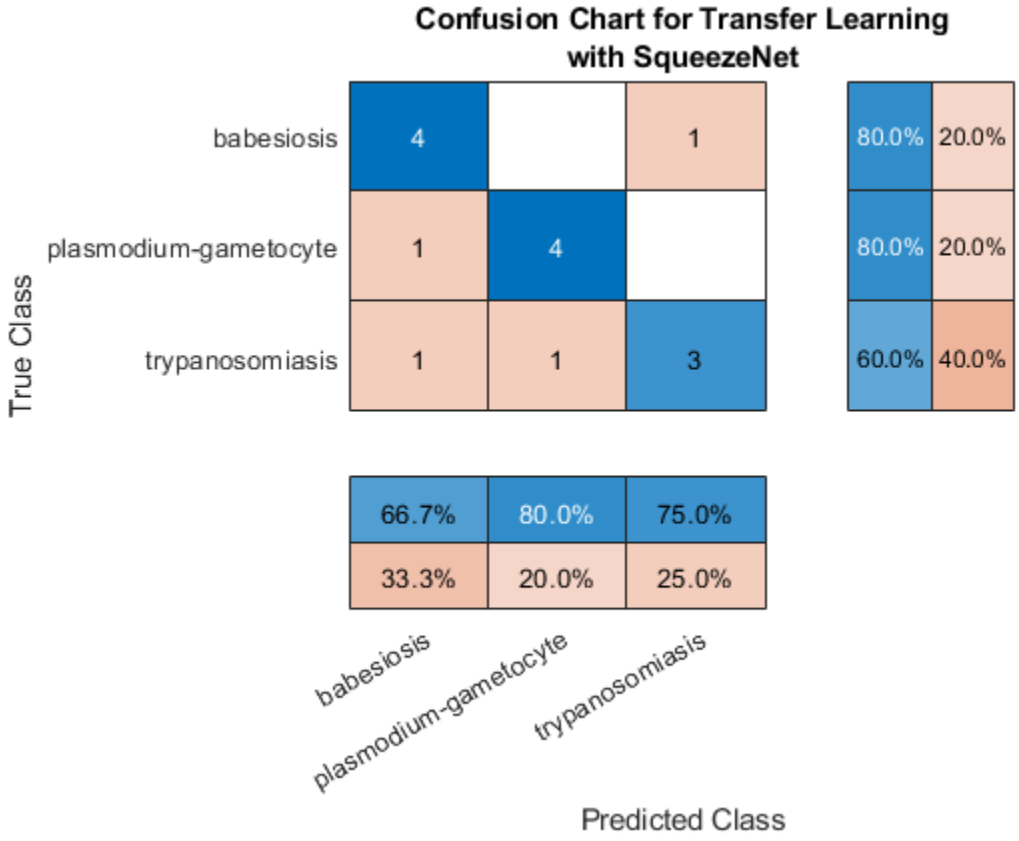


Obtain the SqueezeNet accuracy, plot the confusion chart, and compute the F1 scores.

```
ypred = trainedNet.classify(testImds);
sqznetAccuracy = sum(ypred == testImds.Labels)/numel(testImds.Labels)*100
```

```
sqznetAccuracy = 73.3333
```

```
figure
cchart = confusionchart(testImds.Labels,ypred);
cchart.Title = {'Confusion Chart for Transfer Learning' ; 'with SqueezeNet'};
cchart.RowSummary = 'row-normalized';
cchart.ColumnSummary = 'column-normalized';
```




```
f1SqueezeNet = f1score(cchart.NormalizedValues);
disp(f1SqueezeNet)
```

```
                            F1
                          _______

    babesiosis            0.72727
    plasmodium-gametocyte     0.8
    trypanosomiasis       0.66667
```

SqueezeNet performs better than the simpler CNN, particularly in terms of the F1 score for trypanosomiasis, but the performance does not match the accuracy of the simpler PCA classifier with the wavelet scattering features.

Summary

In this example, the wavelet scattering transform and deep learning frameworks were used to classify pathogens in Giemsa stain images. The limited dataset size provides challenges for training a deep learning classifier even when data augmentation is used. The example illustrated that the wavelet scattering transform can provide a useful alternative to deep networks in such cases. In forming feature vectors from the wavelet scattering transform, we reduced each transform output from a 27-by-38-by-38-by-3 tensor to a 27-element vector. Accordingly, we have used a global pooling of the scattering coefficients. It is possible to utilize other pooling schemes, which could yield better results.

Appendix — Supporting Functions

```
function features = helperScatImages_mean(sn,x)
smat = featureMatrix(sn,x);
features = mean(smat,2:4);
features = features';
end
function F1scores = f1score(cchartVal)
N = sum(cchartVal,'all');
probT = sum(cchartVal)./N;
classProbEst = diag(cchartVal)./N;
Prec = classProbEst'./probT;
probC = [5/15 5/15 5/15];
Recall = classProbEst'./probC;
F1scores = harmmean([Prec ; Recall]);
F1scores = F1scores';
F1scores = table(F1scores,'VariableNames',{'F1'},...
    'RowNames', {'babesiosis','plasmodium-gametocyte', 'trypanosomiasis'});
end

function labels = helperPCAClassifier(features,model)
% This function is only to support wavelet image scattering examples in
% Wavelet Toolbox. It may change or be removed in a future release.
% model is a structure array with fields, M, mu, v, and Labels
% features is the matrix of test data which is Ns-by-L, Ns is the number of
% scattering paths and L is the number of test examples. Each column of
% features is a test example.

% Copyright 2018-2021 MathWorks

labelIdx = determineClass(features,model);
labels = model.Labels(labelIdx);
% Returns as column vector to agree with imageDatastore Labels
labels = labels(:);


%--------------------------------------------------------------------------
function labelIdx = determineClass(features,model)
% Determine number of classes
Nclasses = numel(model.Labels);
% Initialize error matrix
errMatrix = Inf(Nclasses,size(features,2));
for nc = 1:Nclasses
    % class centroid
    mu = model.mu{nc};
    u = model.U{nc};
    % 1-by-L
    errMatrix(nc,:) = projectionError(features,mu,u);
end
% Determine minimum along class dimension
[~,labelIdx] = min(errMatrix,[],1);


%--------------------------------------------------------------------------
function totalerr = projectionError(features,mu,u)
    %
    Npc = size(u,2);
    L = size(features,2);
```

```
    % Subtract class mean: Ns-by-L minus Ns-by-1
    s = features-mu;
    % 1-by-L
    normSqX = sum(abs(s).^2,1)';
    err = Inf(Npc+1,L);
    err(1,:) = normSqX;
    err(2:end,:) = -abs(u'*s).^2;
    % 1-by-L
    totalerr = sqrt(sum(err,1));
end
end
end

function model = helperPCAModel(features,M,Labels)
% This function is only to support wavelet image scattering examples in
% Wavelet Toolbox. It may change or be removed in a future release.
% model = helperPCAModel(features,M,Labels)

% Copyright 2018-2021 MathWorks

% Initialize structure array to hold the affine model
model = struct('Dim',[],'mu',[],'U',[],'Labels',categorical([]),'S',[]);
model.Dim = M;
% Obtain the number of classes
LabelCategories = categories(Labels);
Nclasses = numel(categories(Labels));
for kk = 1:Nclasses
    Class = LabelCategories{kk};
    % Find indices corresponding to each class
    idxClass = Labels == Class;
    % Extract feature vectors for each class
    tmpFeatures = features(:,idxClass);
    % Determine the mean for each class
    model.mu{kk} = mean(tmpFeatures,2);
    [model.U{kk},model.S{kk}] = scatPCA(tmpFeatures);
    if size(model.U{kk},2) > M
        model.U{kk} = model.U{kk}(:,1:M);
        model.S{kk} = model.S{kk}(1:M);

    end
    model.Labels(kk) = Class;
end

    function [u,s,v] = scatPCA(x)
        % Calculate the principal components of x along the second dimension.
        [u,d] = eig(cov(x'));
        % Sort eigenvalues of covariance matrix in descending order
        [s,ind] = sort(diag(d),'descend');
        % sort eigenvector matrix accordingly
        u = u(:,ind);
    end
end
```

See Also

waveletScattering2

Related Examples

- “Texture Classification with Wavelet Image Scattering” (Wavelet Toolbox)

More About

- “Wavelet Scattering” (Wavelet Toolbox)

Wireless Comm Examples

Spectrum Sensing with Deep Learning to Identify 5G and LTE Signals

This example shows how to train a semantic segmentation network using deep learning for spectrum monitoring. One of the uses of spectrum monitoring is to characterize spectrum occupancy. The neural network in this example is trained to identify 5G NR and LTE signals in a wideband spectrogram.

Introduction

Computer vision uses the semantic segmentation technique to identify objects and their locations in an image or a video. In wireless signal processing, the objects of interest are wireless signals, and the locations of the objects are the frequency and time occupied by the signals. In this example we apply the semantic segmentation technique to wireless signals to identify spectral content in a wideband spectrogram.

In the following, you will:

1. Generate training signals.
2. Apply transfer learning to a semantic segmentation network to identify 5G NR and LTE signals in time and frequency.
3. Test the trained network with synthetic signals.
4. Use an SDR to test the network with over the air (OTA) signals.

Generate Training Data

One advantage of wireless signals in the deep learning domain is the fact that the signals are synthesized. Also, we have highly reliable channel and RF impairment models. As a result, instead of collecting and manually labeling signals, you can generate 5G NR signals using 5G Toolbox™ and LTE signals using LTE Toolbox™ functions. You can pass these signals through standards-specified channel models to create the training data.

Train the network with frames that contain only 5G NR or LTE signals and then shift these signals in frequency randomly within the band of interest. Each frame is 40 ms long, which is the duration of 40 subframes. The network assumes that the 5G NR or LTE signal occupies the same band for the whole frame duration. To test the network performance, create frames that contain both 5G NR and LTE signals on distinct random bands within the band of interest.

Use a sampling rate of 61.44 MHz. This rate is high enough to process most of the latest standard signals and several low-cost software defined radio (SDR) systems can sample at this rate providing about 50 MHz of useful bandwidth. To monitor a wider band, you can increase the sample rate, regenerate training frames and retrain the network.

Use the `helperSpecSenseTrainingData` function to generate training frames. This function generates 5G NR signals using the `helperSpecSenseNRSignal` function and LTE signals using the `helperSpecSenseLTESignal` function. This table lists 5G NR variable signal parameters.

5G NR Parameter	Value	Units
Bandwidth	[10 15 20 25 30 40 50]	MHz
Sub-Carrier Spacing (SCS)	[15 30]	kHz
SSB Block Pattern	["Case A" "Case B"]	
SSB Period	[20]	ms

This table lists LTE variable signal parameters.

LTE Parameter	Value	Units
Reference Channel	["R.2","R.6","R.8","R.9"}	
Bandwidth	[10 5 15 20]	MHz
Duplex Mode	FDD	

Use the `nrCDLChannel` (5G Toolbox) and the `lteFadingChannel` (LTE Toolbox) functions to add channel impairments. For details of the channel configurations, see the `helperSpecSenseTrainingData` function. This table lists channel parameters.

Channel Parameter	Value	Units
SNR	[40 50 100]	dB
Doppler	[0 10 500]	Hz

The `helperSpecSenseTrainingData` function uses the `helperSpecSenseSpectrogramImage` function to create spectrogram images from complex baseband signals. Calculate the spectrograms using an FFT length of 4096. Generate 256 by 256 RGB images. This Image size allows a large enough batch of images to fit in memory during training while providing enough resolution in time and frequency. If your GPU does not have sufficient memory, you can resize the images to smaller sizes or reduce the training batch size.

The `generateTrainData` variable determines whether training data is to be downloaded or generated. Choosing "Use downloaded data" sets the `generateTrainData` variable to `false`. Choosing "Generate training data" sets the `generateTrainData` variable to `true` to generate the training data from scratch. Data generation may take several hours depending on the configuration of your computer. Using a PC with Intel® Xeon® W-2133 CPU @ 3.60GHz and creating a parallel pool with six workers with the Parallel Computing Toolbox, training data generation takes about an hour. Choose "Train network now" to train the network. This process takes about 20 minutes with the same PC and NVIDIA® Titan V GPU. Choose "Use trained network" to skip network training. Instead, the example downloads the trained network.

Use 900 frames from each set of signals: 5G NR only, LTE only and 5G NR and LTE both. If you increase the number of possible values for the system parameters, increase the number of training frames .

```
imageSize = [256 256];    % pixels
sampleRate = 61.44e6;     % Hz
numSubFrames = 40;        % corresponds to 40 ms
frameDuration = numSubFrames*1e-3;    % seconds
trainDir = fullfile(pwd,'TrainingData');
```

```
generateTrainData = Use downloaded data ;
trainNow = Use trained network ;
if ~generateTrainData || ~trainNow
  helperSpecSenseDownloadData()
end
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/SpectrumSensing/SpectrumSenseTrainingDataNetwork.t
Download complete. Extracting files.
Extract complete.
```

```
if generateTrainData
  numFramesPerStandard = 900;
  helperSpecSenseTrainingData(numFramesPerStandard,imageSize,trainDir,numSubFrames,sampleRate);
end
```

Load Training Data

Use the `imageDatastore` function to load training images with the spectrogram of 5G NR and LTE signals. The `imageDatastore` function enables you to efficiently load a large collection of images from disk. Spectrogram images are stored in .png files.

```
imds = imageDatastore(trainDir,'IncludeSubfolders',false,'FileExtensions','.png');
```

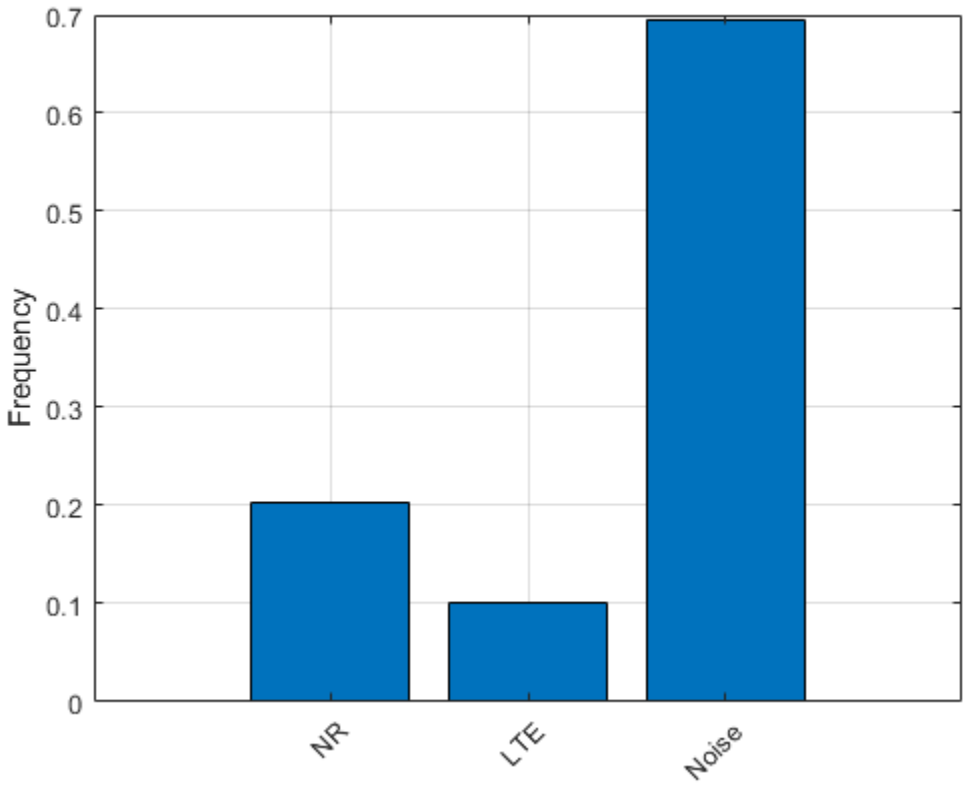
Use the `pixelLabelDatastore` (Computer Vision Toolbox) function to load spectrogram pixel label image data. Each pixel is labeled as one of "NR", "LTE" or "Noise". A pixel label datastore encapsulates the pixel label data and the label ID to a class name mapping. Pixel labels are stored in .hdf files.

```
classNames = ["NR" "LTE" "Noise"];
pixelLabelID = [127 255 0];
pxdsTruth = pixelLabelDatastore(trainDir,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
```

Analyze Dataset Statistics

To see the distribution of class labels in the training dataset, use the `countEachLabel` (Computer Vision Toolbox) function to count the number of pixels by class label, and plot the pixel counts by class.

```
tbl = countEachLabel(pxdsTruth);
frequency = tbl.PixelCount/sum(tbl.PixelCount);
figure
bar(1:numel(classNames),frequency)
grid on
xticks(1:numel(classNames))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes would have an equal number of observations. However, with wireless signals it is common for the classes in the training set to be imbalanced. 5G NR signals may have larger bandwidth than LTE signals, and noise fills the background. Because the learning is biased in favor of the dominant classes, imbalance in the number of observations per class can be detrimental to the learning process. In the Balance Classes Using Class Weighting on page 13-0 section, class weighting is used to mitigate bias caused by imbalance in the number of observations per class.

Prepare Training, Validation, and Test Sets

The deep neural network uses 80% of the single signal images from the dataset for training and, 20% of the images for validation. The `helperSpecSensePartitionData` function randomly splits the image and pixel label data into training and validation sets.

```
[imdsTrain,pxdsTrain,imdsVal,pxdsVal] = helperSpecSensePartitionData(imds,pxdsTruth,[80 20]);
cdsTrain = pixelLabelImageDatastore(imdsTrain,pxdsTrain,'OutputSize',imageSize);
cdsVal = pixelLabelImageDatastore(imdsVal,pxdsVal,'OutputSize',imageSize);
```

Train Deep Neural Network

Use the `deeplabv3plusLayers` (Computer Vision Toolbox) function to create a semantic segmentation neural network. Choose `resnet50` as the base network and specify the input image size (number of pixels used to represent time and frequency axes) and the number of classes. If the Deep Learning Toolbox™ Model *for ResNet-50 Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**. Check that the installation is successful by

typing `resnet50` at the command line. If the required support package is installed, then the function returns a `DAGNetwork` object.

```
baseNetwork = resnet50;
lgraph = deeplabv3plusLayers(imageSize,numel(classNames),baseNetwork);
```

Balance Classes Using Class Weighting

To improve training when classes in the training set are not balanced, you can use class weighting to balance the classes. Use the pixel label counts computed earlier with the `countEachLabel` function and calculate the median frequency class weights.

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;
```

Specify the class weights using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights
lgraph = replaceLayer(lgraph,"classification",pxLayer);
```

Select Training Options

Configure training using the `trainingOptions` function to specify the stochastic gradient descent with momentum (SGDM) optimization algorithm and the hyper-parameters used for SGDM. To get the best performance from the network, you can use the Experiment Manager to optimize training options.

```
opts = trainingOptions("sgdm",...
  MiniBatchSize = 40,...
  MaxEpochs = 20, ...
  LearnRateSchedule = "piecewise",...
  InitialLearnRate = 0.02,...
  LearnRateDropPeriod = 10,...
  LearnRateDropFactor = 0.1,...
  ValidationData = cdsVal,...
  ValidationPatience = 5,...
  Shuffle="every-epoch",...
  OutputNetwork = "best-validation-loss",...
  Plots = 'training-progress')
```

```
opts =
  TrainingOptionsSGDM with properties:

                     Momentum: 0.9000
               InitialLearnRate: 0.0200
              LearnRateSchedule: 'piecewise'
            LearnRateDropFactor: 0.1000
            LearnRateDropPeriod: 10
               L2Regularization: 1.0000e-04
       GradientThresholdMethod: 'l2norm'
             GradientThreshold: Inf
                     MaxEpochs: 20
                 MiniBatchSize: 40
                       Verbose: 1
               VerboseFrequency: 50
                 ValidationData: [1×1 pixelLabelImageDatastore]
            ValidationFrequency: 50
```

```
              ValidationPatience: 5
                         Shuffle: 'every-epoch'
                  CheckpointPath: ''
            ExecutionEnvironment: 'auto'
                      WorkerLoad: []
                       OutputFcn: []
                           Plots: 'training-progress'
                  SequenceLength: 'longest'
              SequencePaddingValue: 0
          SequencePaddingDirection: 'right'
              DispatchInBackground: 0
           ResetInputNormalization: 1
      BatchNormalizationStatistics: 'population'
                   OutputNetwork: 'best-validation-loss'
```

Train the network using the combined training data store, `cdsTrain`. The combined training data store contains single signal frames and true pixel labels.

```
if trainNow
  [net,trainInfo] = trainNetwork(cdsTrain,lgraph,opts); %#ok<UNRCH>
else
  load specSenseTrainedNet net
end
```

Test with Synthetic Signals

Test the network signal identification performance using signals that contain both 5G NR and LTE signals. Use the `semanticseg` (Computer Vision Toolbox) function to get the pixel estimates of the spectrogram images in the test data set. Use the `evaluateSemanticSegmentation` (Computer Vision Toolbox) function to compute various metrics to evaluate the quality of the semantic segmentation results.

```
dataDir = fullfile(trainDir,'LTE_NR');
imds = imageDatastore(dataDir,'IncludeSubfolders',false,'FileExtensions','.png');
pxdsResults = semanticseg(imds,net,"WriteLocation",tempdir);
```

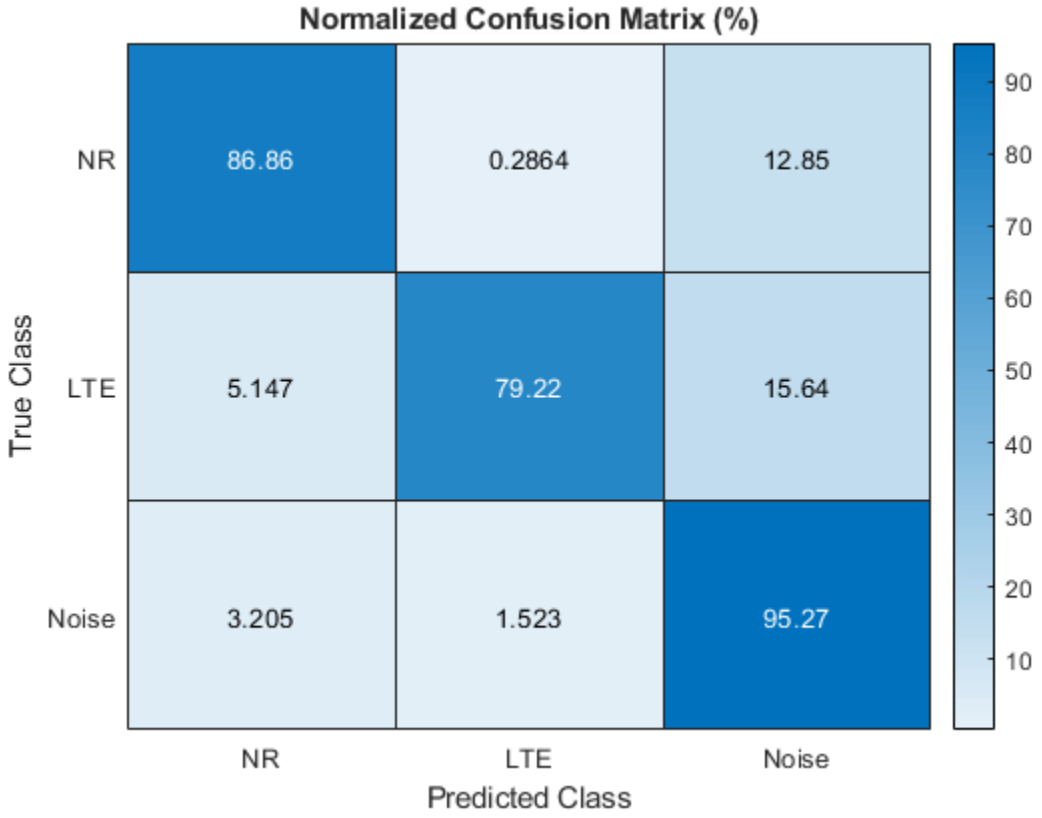
```
Running semantic segmentation network
-------------------------------------
* Processed 900 images.
```

```
pxdsTruth = pixelLabelDatastore(dataDir,classNames,pixelLabelID,...
  'IncludeSubfolders',false,'FileExtensions','.hdf');
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTruth);
```

```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 900 images.
* Finalizing... Done.
* Data set metrics:

    GlobalAccuracy    MeanAccuracy    MeanIoU    WeightedIoU    MeanBFScore
    ______________    ____________    _______    ___________    ___________

       0.88609           0.87117       0.79066      0.79601        0.65624
```

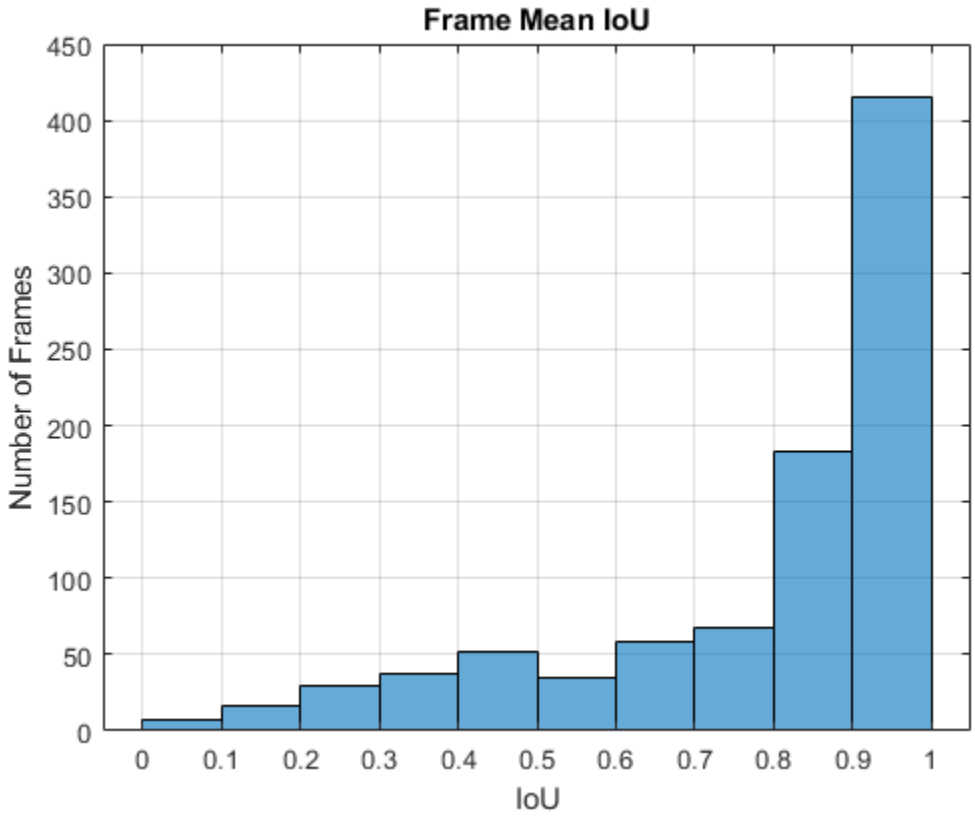
Plot the normalized confusion matrix for all test frames as a heat map.

```
normConfMatData = metrics.NormalizedConfusionMatrix.Variables;
figure
h = heatmap(classNames,classNames,100*normConfMatData);
h.XLabel = 'Predicted Class';
h.YLabel = 'True Class';
h.Title = 'Normalized Confusion Matrix (%)';
```



Plot the histogram of the per-image intersection over union (IoU). For each class, IoU is the ratio of correctly classified pixels to the total number of ground truth and predicted pixels in that class.

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
grid on
xlabel('IoU')
ylabel('Number of Frames')
title('Frame Mean IoU')
```



Inspecting low SNR frames shows that the spectrogram images do not contain visual features that can help the network identify the low SNR frames correctly. Repeat the same process, considering only the frames with average SNR of 50dB or 100dB and ignoring the frames with average SNR of 40dB.

```
files = dir(fullfile(dataDir,'*.mat'));
dataFiles = {};
labelFiles = {};
for p=1:numel(files)
  load(fullfile(files(p).folder,files(p).name),'params');
  if params.SNRdB > 40
    [~,name] = fileparts(files(p).name);
    dataFiles = [dataFiles; fullfile(files(p).folder,[name '.png'])]; %#ok<AGROW>
    labelFiles = [labelFiles; fullfile(files(p).folder,[name '.hdf'])]; %#ok<AGROW>
  end
end
imds = imageDatastore(dataFiles);
pxdsResults = semanticseg(imds,net,"WriteLocation",tempdir);
```

```
Running semantic segmentation network
-------------------------------------
* Processed 608 images.
```

```
pxdsTruth = pixelLabelDatastore(labelFiles,classNames,pixelLabelID);
metrics = evaluateSemanticSegmentation(pxdsResults,pxdsTruth);
```

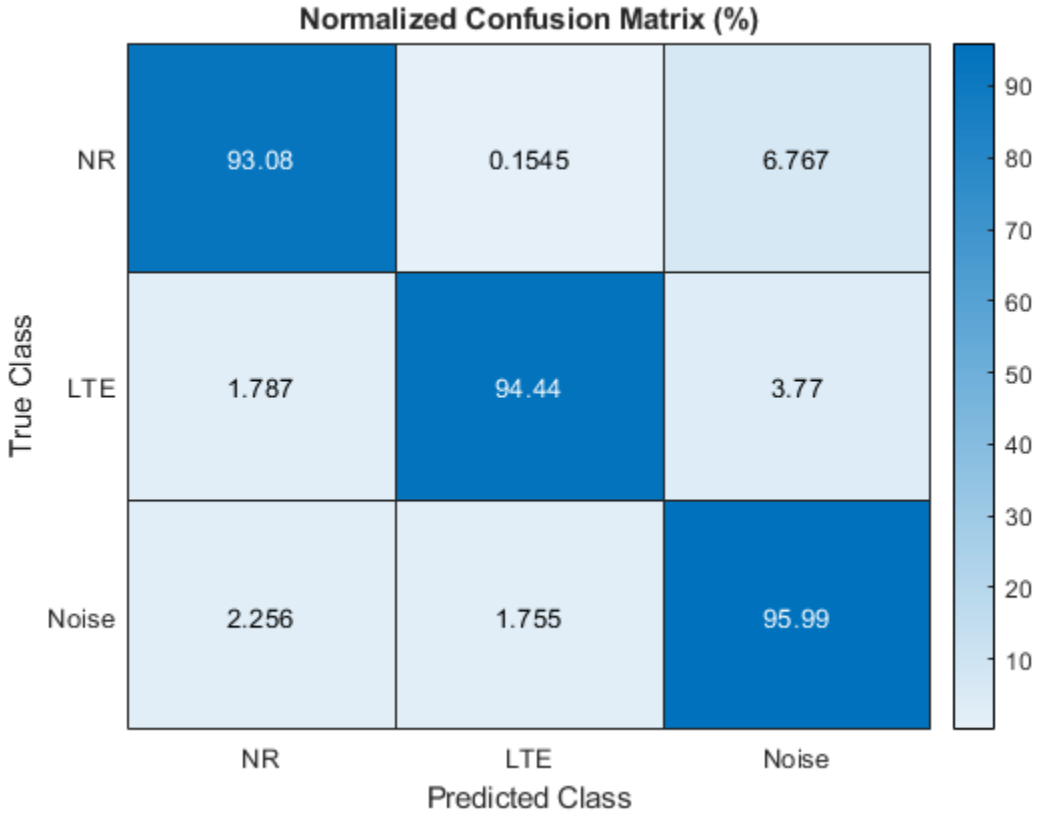
```
Evaluating semantic segmentation results
----------------------------------------
```

```
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 608 images.
* Finalizing... Done.
* Data set metrics:
```

GlobalAccuracy	MeanAccuracy	MeanIoU	WeightedIoU	MeanBFScore
0.94487	0.94503	0.89799	0.89582	0.74699

Considering only the set of frames with higher SNR, replot the normalized confusion matrix and observe the improved network accuracy.

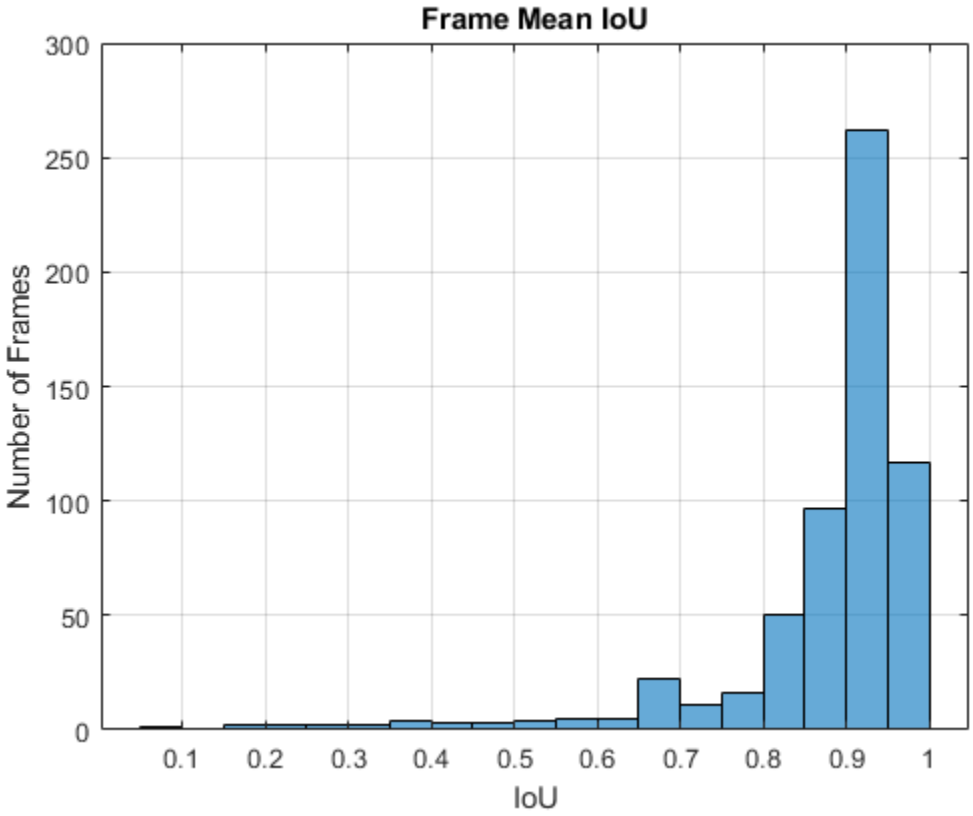
```
normConfMatData = metrics.NormalizedConfusionMatrix.Variables;
figure
h = heatmap(classNames,classNames,100*normConfMatData);
h.XLabel = 'Predicted Class';
h.YLabel = 'True Class';
h.Title = 'Normalized Confusion Matrix (%)';
```



Considering only the set of frames with higher SNR, replot the per-image IoU histogram and observe the improved distribution.

```
imageIoU = metrics.ImageMetrics.MeanIoU;
figure
histogram(imageIoU)
grid on
xlabel('IoU')
```

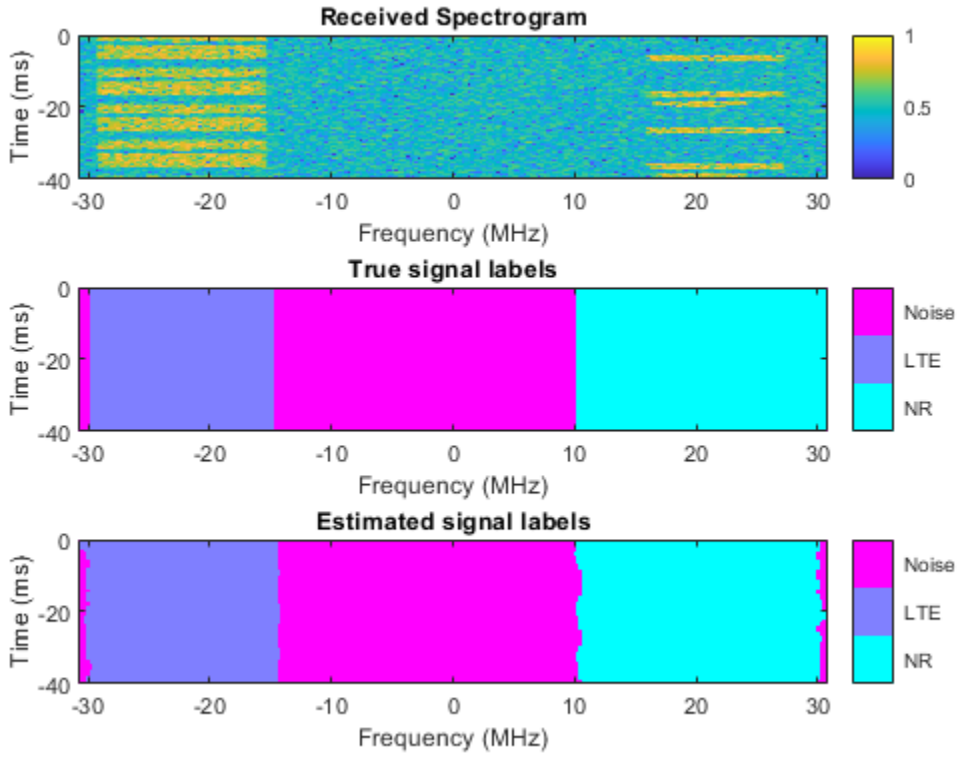
```
ylabel('Number of Frames')
title('Frame Mean IoU')
```



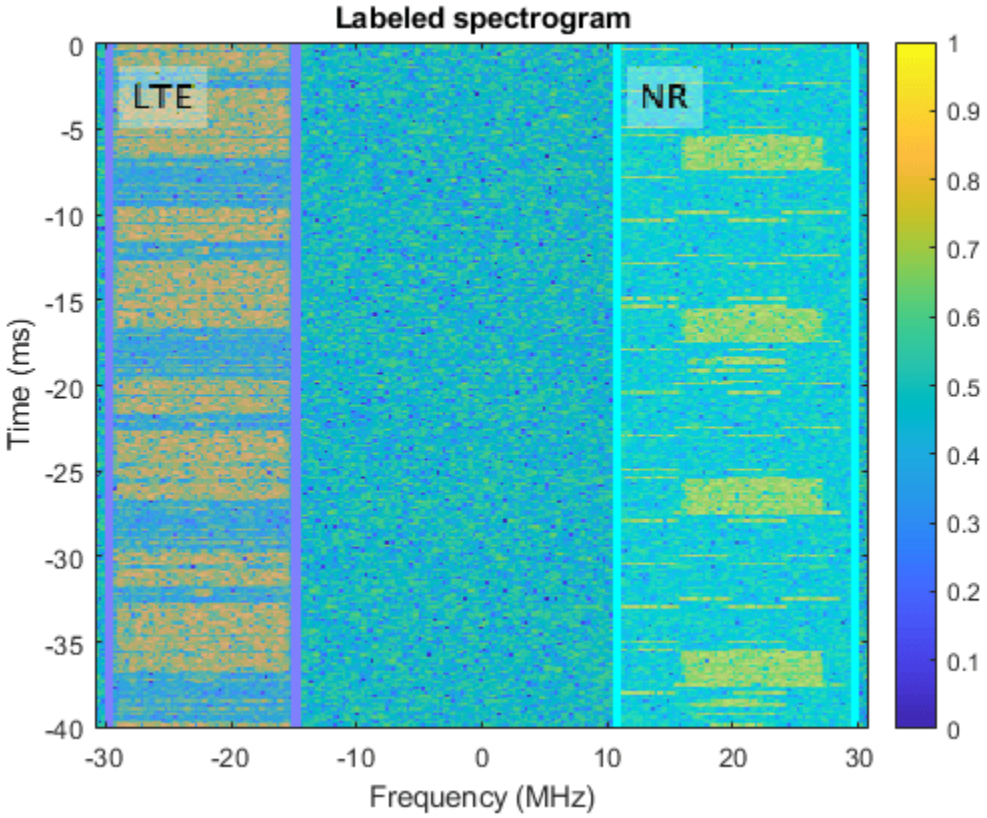
Identify 5G NR and LTE Signals in Spectrogram

Visualize the received spectrum, true labels, and predicted labels for the image with index 602.

```
imgIdx = 602;
rcvdSpectrogram = readimage(imds,imgIdx);
trueLabels = readimage(pxdsTruth,imgIdx);
predictedLabels = readimage(pxdsResults,imgIdx);
figure
helperSpecSenseDisplayResults(rcvdSpectrogram,trueLabels,predictedLabels, ...
  classNames,sampleRate,0,frameDuration)
```




```
figure
helperSpecSenseDisplayIdentifiedSignals(rcvdSpectrogram,predictedLabels, ...
  classNames,sampleRate,0,frameDuration)
```



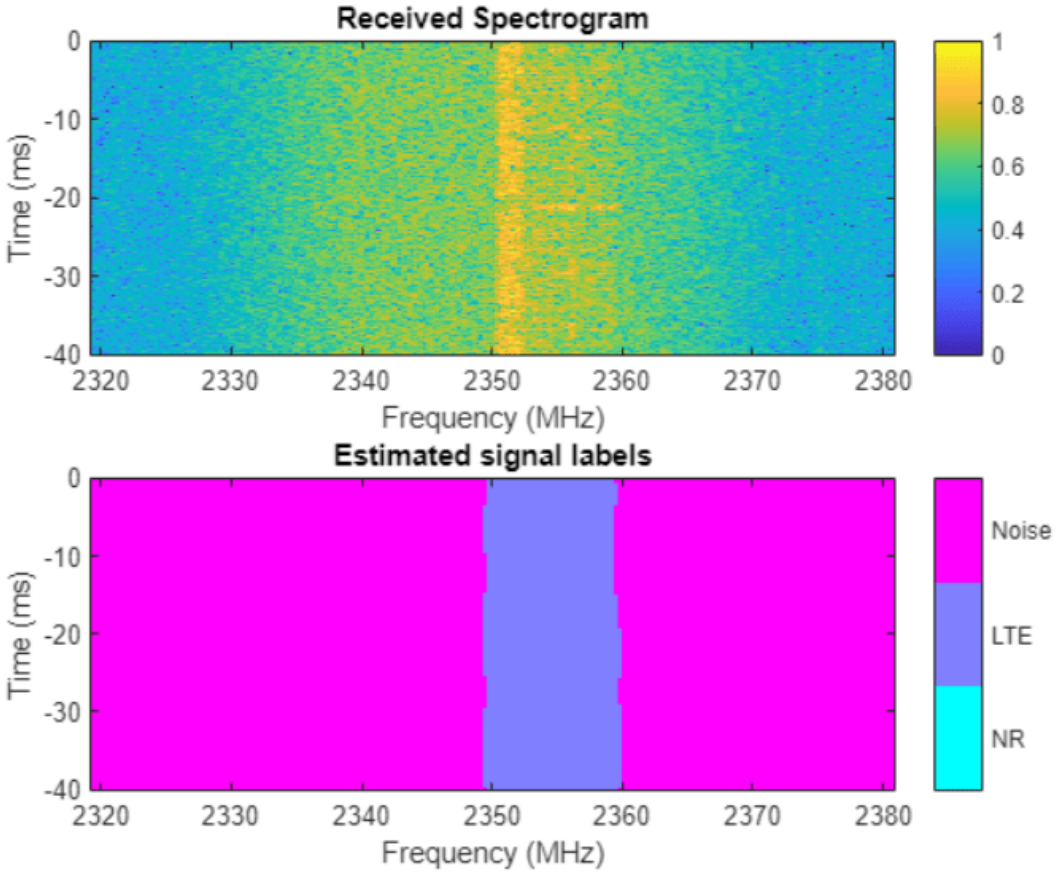
Test with Over-the-Air Signals

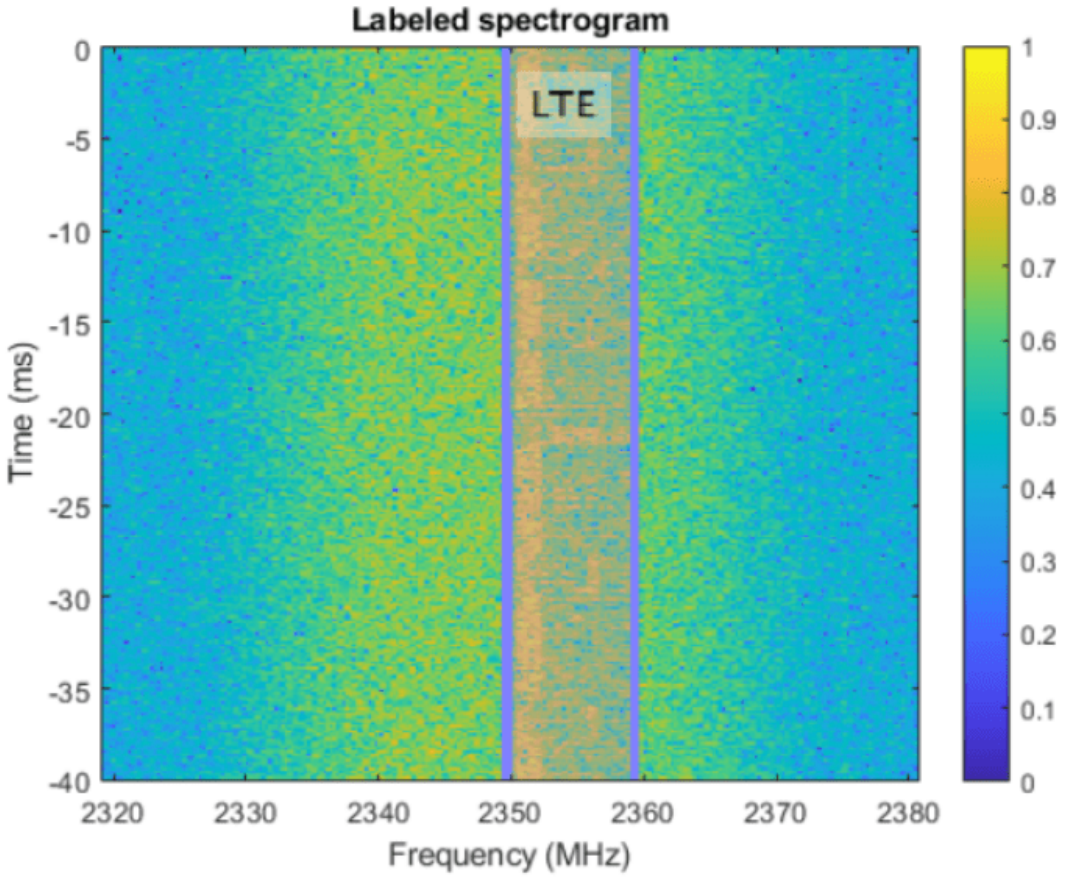
Test the performance of the trained network using over-the-air signal captures. Find a nearby base station and tune the center frequency of your radio to cover the band of the signals you want to identify. This example sets the center frequency to 2.35 GHz. If you have at least one ADALM-PLUTO radio and have installed Communication Toolbox Support Package for ADALM-PLUTO Radio, you can run this section of the code. In case you do not have access to an ADALM-PLUTO radio, this example shows results of a test conducted using captured signals.

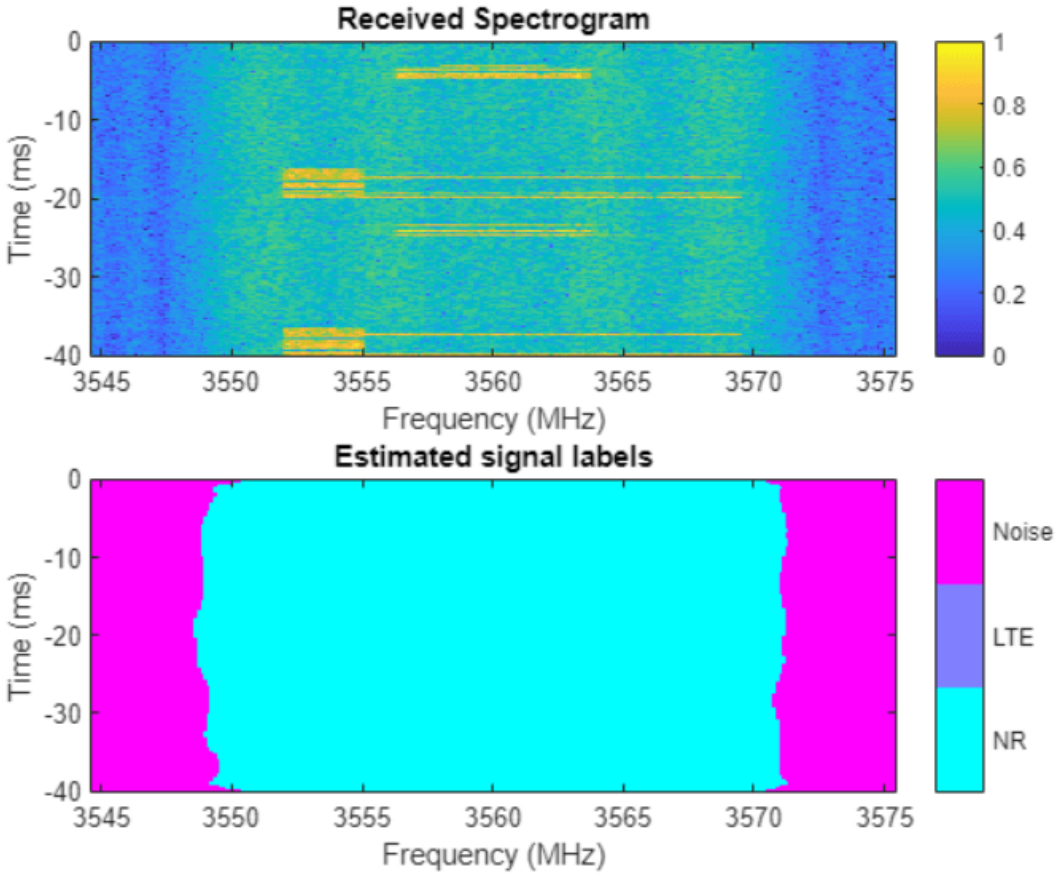
```
runSDRSection = false;
if helperIsPlutoSDRInstalled()
  radios = findPlutoRadio();
  if length(radios) >= 1
    runSDRSection = true;
  else
    disp("At least one ADALM-PLUTO radios is needed. Skipping SDR test.")
  end
else
    disp("Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found.
    disp("Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package."
    disp("Skipping SDR test.")
end
```

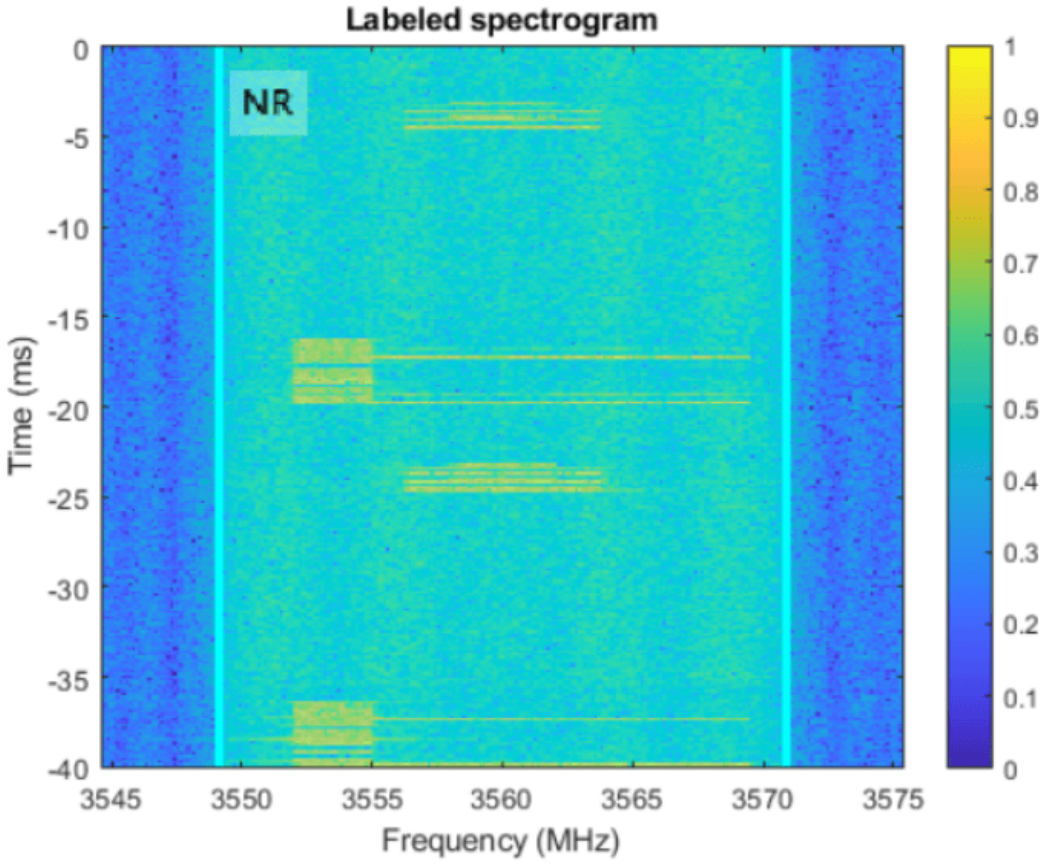
```
Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found.

Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package.

Skipping SDR test.
```

```
if runSDRSection
  % Set up PlutoSDR receiver
  rx = sdrrx('Pluto');
  rx.CenterFrequency = 2.35e9;
  rx.BasebandSampleRate = sampleRate;
  rx.SamplesPerFrame = frameDuration*rx.BasebandSampleRate;
  rx.OutputDataType = 'single';
  rx.EnableBurstMode = true;
  rx.NumFramesInBurst = 1;
  Nfft = 4096;
  overlap = 10;

  meanAllScores = zeros([imageSize numel(classNames)]);
  segResults = zeros([imageSize 10]);
  for frameCnt=1:10
    rxWave = rx();
    rxSpectrogram = helperSpecSenseSpectrogramImage(rxWave,Nfft,sampleRate,imageSize);

    [segResults(:,:,frameCnt),scores,allScores] = semanticseg(rxSpectrogram,net);
    meanAllScores = (meanAllScores*(frameCnt-1) + allScores) / frameCnt;
  end
  release(rx)

  [~,predictedLabels] = max(meanAllScores,[],3);
  figure
  helperSpecSenseDisplayResults(rxSpectrogram,[],predictedLabels,classNames,...
    sampleRate,rx.CenterFrequency,frameDuration)
  figure
  freqBand = helperSpecSenseDisplayIdentifiedSignals(rxSpectrogram,predictedLabels,...
    classNames,sampleRate,rx.CenterFrequency,frameDuration)
else
  figure
  imshow('lte_capture_result1.png')
  figure
  imshow('lte_capture_result2.png')
  figure
  imshow('nr_capture_result1.png')
  figure
  imshow('nr_capture_result2.png')
end
```


Received Spectrogram
Time (ms)
0
-10
-20
-30
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
1
0.8
0.6
0.4
0.2
0
Estimated signal labels
Time (ms)
0
-10
-20
-30
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
Noise
LTE
NR


Labeled spectrogram
LTE
Time (ms)
0
-5
-10
-15
-20
-25
-30
-35
-40
2320
2330
2340
2350
2360
2370
2380
Frequency (MHz)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0


Received Spectrogram
Time (ms)
0
-10
-20
-30
-40
3545
3550
3555
3560
3565
3570
3575
Frequency (MHz)
1
0.8
0.6
0.4
0.2
0
Estimated signal labels
Time (ms)
0
-10
-20
-30
-40
3545
3550
3555
3560
3565
3570
3575
Frequency (MHz)
Noise
LTE
NR



Conclusions and Further Exploration

The trained network can distinguish 5G NR and LTE signals including two example captures from real base stations. The network may not be able to identify every captured signal correctly. In such cases, enhance the training data either by generating more representative synthetic signals or capturing over-the-air signals and including these in the training set.

You can use the LTE “Cell Search, MIB and SIB1 Recovery” (LTE Toolbox) and the “NR Cell Search and MIB and SIB1 Recovery” (5G Toolbox) examples to identify LTE and 5G NR base stations manually to capture training data, respectively.

If you need to monitor wider bands of spectrum, increase the `sampleRate`, regenerate the training data and retrain the network.

See Also

`classificationLayer` | `featureInputLayer` | `fullyConnectedLayer` | `reluLayer` | `softmaxLayer` | `pixelLabelDatastore` | `countEachLabel` | `pixelClassificationLayer`

More About

- “Deep Learning in MATLAB” on page 1-2

Autoencoders for Wireless Communications

This example shows how to model an end-to-end communications system with an autoencoder to reliably transmit information bits over a wireless channel.

Introduction

A traditional autoencoder is an unsupervised neural network that learns how to efficiently compress data, which is also called encoding. The autoencoder also learns how to reconstruct the data from the compressed representation such that the difference between the original data and the reconstructed data is minimal.

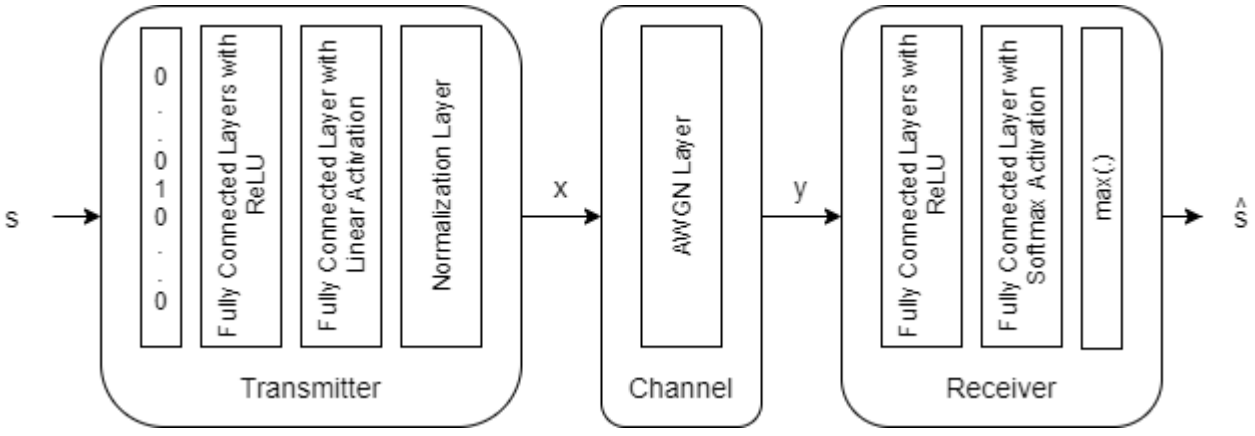
Traditional wireless communication systems are designed to provide reliable data transfer over a channel that impairs the transmitted signals. These systems have multiple components such as channel coding, modulation, equalization, synchronization, etc. Each component is optimized independently based on mathematical models that are simplified to arrive at closed form expressions. On the contrary, an autoencoder jointly optimizes the transmitter and the receiver as a whole. This joint optimization has the potential of providing a better performance than the traditional systems [1] on page 13-0 ,[2] on page 13-0 .

Traditional autoencoders are usually used to compress images, in other words remove redundancies in an image and reduce its dimension. A wireless communication system on the other hand uses channel coding and modulation techniques to add redundancy to the information bits. With this added redundancy, the system can recover the information bits that are impaired by the wireless channel. So, a wireless autoencoder actually adds redundancy and tries to minimize the number of errors in the received information for a given channel while learning to apply both channel coding and modulation in an unsupervised way.

Basic Autoencoder System

The following is the block diagram of a wireless auto encoder system. The encoder (transmitter) first maps k information bits into a message s such that $s \in \{1, \ldots, M\}$, where $M = 2^k$. Then message s is mapped to n real number to create $\mathbf{x} = f(s) \in \mathbb{R}^n$. The last layer of the encoder imposes constraints on $\mathbf{x}$ to further restrict the encoded symbols. The following are possible such constraints and are implemented using the normalization layer:

- Energy constraint: $\|x\|_2^2 \le n$
- Average power constraint: $\mathbb{E}[|x_i|^2] \le 1, \forall i$



Define the communication rate of this system as $R = k/n$ [bits/channel use], where (n,k) means that the system sends one of $M = 2^k$ messages using n channel uses. The channel impairs encoded (i.e. transmitted) symbols to generate $\mathbf{y} \in \mathbb{R}^n$. The decoder (i.e. receiver) produces an estimate, $\hat{s}$, of the transmitted message, s.

The input message is defined as a one-hot vector $\mathbf{1}_s \in \mathbb{R}^M$, which is defined as a vector whose elements are all zeros except the s^{th} one. The channel is additive white Gaussian noise (AWGN) that adds noise to achieve a given energy per bit to noise power density ratio, E_b/N_o.

Define a (7,4) autoencoder network with energy normalization and a training E_b/N_o of 3 dB. In [1] on page 13-0 , authors showed that two fully connected layers for both the encoder (transmitter) and the decoder (receiver) provides the best results with minimal complexity. Input layer (`featureInputLayer`) accepts a one-hot vector of length M. The encoder has two fully connected layers (`fullyConnectedLayer`). The first one has M inputs and M outputs and is followed by an ReLU layer (`reluLayer`). The second fully connected layer has M inputs and n outputs and is followed by the normalization layer (helperAEWNormalizationLayer.m). The encoder layers are followed by the AWGN channel layer (helperAEWAWGNLayer.m). The output of the channel is passed to the decoder layers. The first decoder layer is a fully connected layer that has n inputs and M outputs and is followed by an ReLU layer. Second fully connected layer has M inputs and M outputs and is followed by a softmax layer (`softmaxLayer`), which outputs the probability of each M symbols. The classification layer (`classificationLayer`) outputs the most probable transmitted symbol from 0 to M-1.

```
k = 4;    % number of input bits
M = 2^k;  % number of possible input symbols
n = 7;    % number of channel uses
EbNo = 3; % Eb/No in dB

wirelessAutoencoder = [
  featureInputLayer(M,"Name","One-hot input","Normalization","none")

  fullyConnectedLayer(M,"Name","fc_1")
  reluLayer("Name","relu_1")

  fullyConnectedLayer(n,"Name","fc_2")

  helperAEWNormalizationLayer("Method", "Energy", "Name", "wnorm")

  helperAEWAWGNLayer("Name","channel",...
    "NoiseMethod","EbNo",...
    "EbNo",EbNo,...
    "BitsPerSymbol",2,...
    "SignalPower",1)

  fullyConnectedLayer(M,"Name","fc_3")
  reluLayer("Name","relu_2")

  fullyConnectedLayer(M,"Name","fc_4")
  softmaxLayer("Name","softmax")

  classificationLayer("Name","classoutput")]
```

```
wirelessAutoencoder =
  11×1 Layer array with layers:
```

```
     1   'One-hot input'   Feature Input            16 features
     2   'fc_1'            Fully Connected          16 fully connected layer
     3   'relu_1'          ReLU                     ReLU
     4   'fc_2'            Fully Connected          7 fully connected layer
     5   'wnorm'           Wireless Normalization   Energy normalization layer
     6   'channel'         AWGN Channel             AWGN channel with EbNo = 3
     7   'fc_3'            Fully Connected          16 fully connected layer
     8   'relu_2'          ReLU                     ReLU
     9   'fc_4'            Fully Connected          16 fully connected layer
    10   'softmax'         Softmax                  softmax
    11   'classoutput'     Classification Output    crossentropyex
```

The `helperAEWTrainWirelessAutoencoder.m` function defines such a network based on the (n,k), normalization method and the E_b/N_o values. The Wireless Autoencoder Training Function section on page 13-0 shows the contents of the `helperAEWTrainWirelessAutoencoder.m` function.

Train Autoencoder

Run the `helperAEWTrainWirelessAutoencoder.m` function to train a (2,2) autoencoder with energy normalization. This function uses the `trainingOptions` function to select

- Adam (adaptive moment estimation) optimizer,
- Initial learning rate of 0.01,
- Maximum epochs of 15,
- Minibatch size of 20*M,
- Piecewise learning schedule with drop period of 10 and drop factor of 0.1.

Then, the `helperAEWTrainWirelessAutoencoder.m` function runs the `trainNetwork` function to train the autoencoder network with the selected options. Finally, this function separates the network into encoder and decoder parts. Encoder starts with the input layer and ends after the normalization layer. Decoder starts after the channel layer and ends with the classification layer. A feature input layer is added at the beginning of the decoder.

Train the autoencoder with an E_b/N_o value that is low enough to result in some errors but not too low such that the training algorithm cannot extract any useful information from the received symbols, y. Set E_b/N_o to 3 dB.

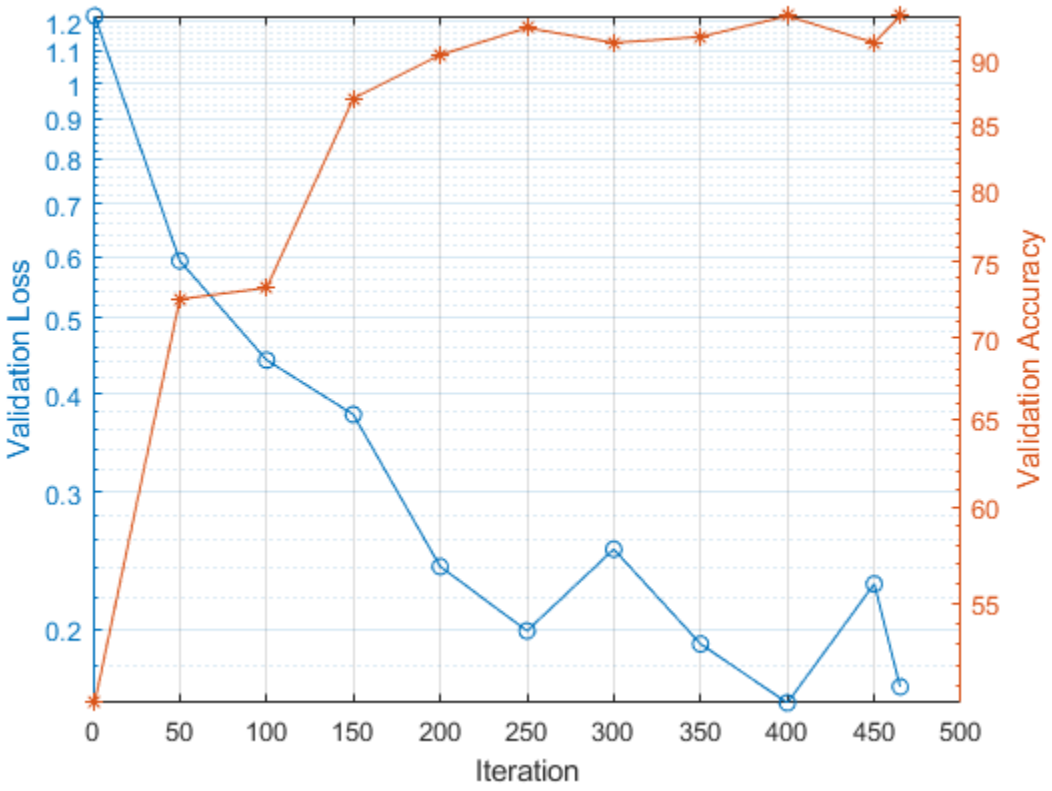
Training an autoencoder may take several minutes. Set `trainNow` to false to use saved networks.

```
trainNow = false; %#ok<*NASGU>

n = 2;                      % number of channel uses
k = 2;                      % bits per data symbol
EbNo = 3;                   % dB
normalization = "Energy";   % Normalization "Energy" | "Average power"

if trainNow
  [txNet22e,rxNet22e,info22e,wirelessAutoEncoder22e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo); %#ok<*UNRCH>
else
  load trainedNet_n2_k2_energy txNet rxNet info trainedNet
  txNet22e = txNet;
  rxNet22e = rxNet;
  info22e = info;
```

```
  wirelessAutoEncoder22e = trainedNet;
end
```

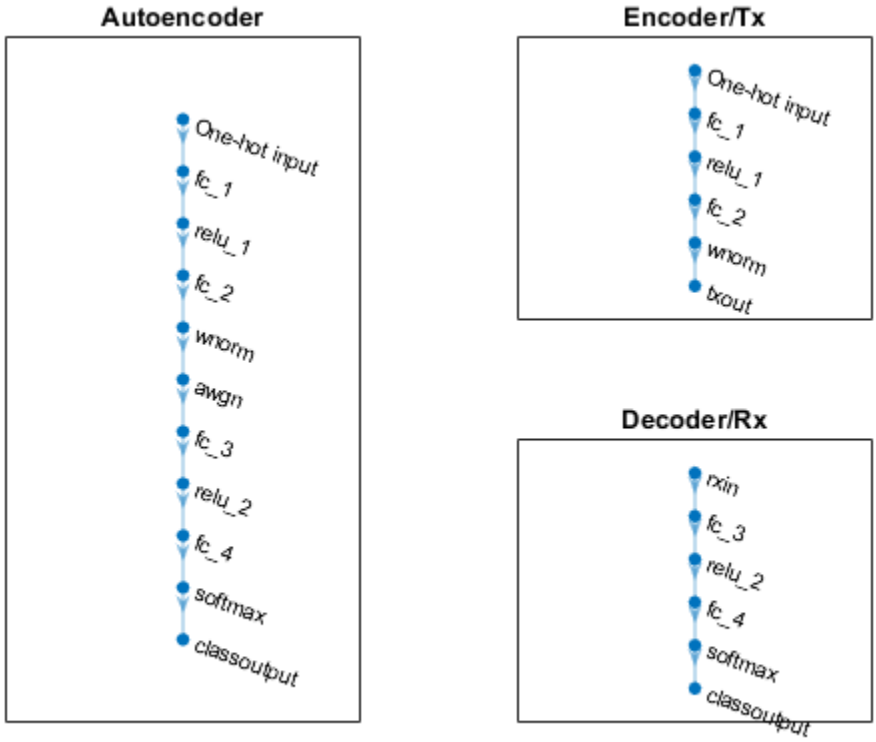
Plot the traning progress. The validation accuracy quickly reaches more than 90% while the validation loss keeps slowly decreasing. This behavior shows that the training E_b/N_o value was low enough to cause some errors but not too low to avoid convergence. For definitions of validation accuracy and validation loss, see “Monitor Deep Learning Training Progress” on page 5-115 section.

```
figure
helperAEWPlotTrainingPerformance(info22e)
```



Use the plot object function of the trained network objects to show the layer graphs of the full autoencoder, the encoder network, i.e. the transmitter, and the decoder network, i.e. the receiver.

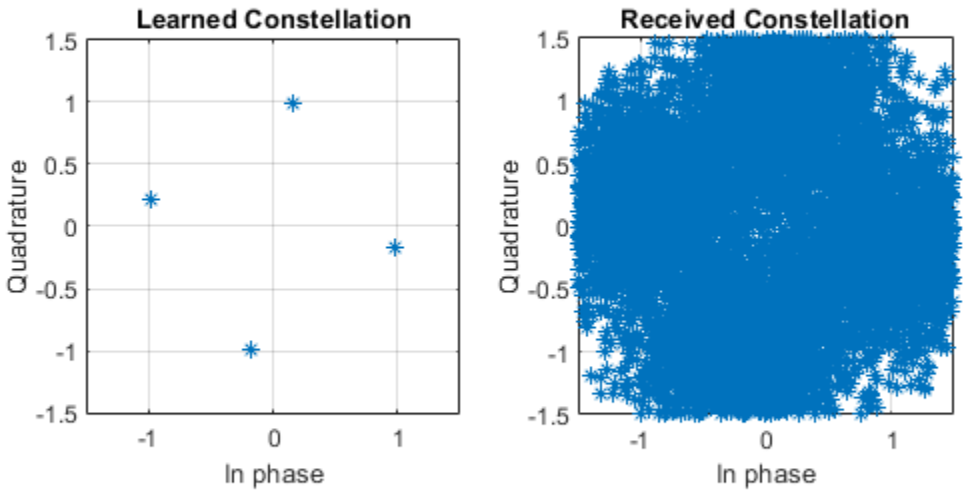
```
figure
tiledlayout(2,2)
nexttile([2 1])
plot(wirelessAutoEncoder22e)
title('Autoencoder')
nexttile
plot(txNet22e)
title('Encoder/Tx')
nexttile
plot(rxNet22e)
title('Decoder/Rx')
```



Plot Transmitted and Received Constellation

Plot the constellation learned by the autoencoder to send symbols through the AWGN channel together with the received constellation. For a (2,2) configuration, autoencoder learns a QPSK ($M = 2^k = 4$) constellation with a phase rotation. The received constellation is basically the activation values at the output of the channel layer obtained using the `activations` function and treated as interleaved complex numbers.

```
subplot(1,2,1)
helperAEWPlotConstellation(txNet22e)
title('Learned Constellation')
subplot(1,2,2)
helperAEWPlotReceivedConstellation(wirelessAutoEncoder22e)
title('Received Constellation')
```



Simulate BLER Performance

Simulate the block error rate (BLER) performance of the (2,2) autoencoder. Setup simulation parameters.

```
simParams.EbNoVec = 0:0.5:8;
simParams.MinNumErrors = 10;
simParams.MaxNumFrames = 300;
simParams.NumSymbolsPerFrame = 10000;
simParams.SignalPower = 1;
```

Generate random integers in the [0 M-1] range that represents k random information bits. Encode these information bits into complex symbols with `helperAEWEncode.m` function. The `helperAEWEncode` function runs the encoder part of the autoencoder then maps the real valued $\mathbf{x}$ vector into a complex valued $\mathbf{x_c}$ vector such that the odd and even elements are mapped into the in-phase and the quadrature component of a complex symbol, respectively, where $\mathbf{x}_c = \mathbf{x}(1:2:end) + j\mathbf{x}(2:2:end)$. In other words, treat the $\mathbf{x}$ array as an interleaved complex array.

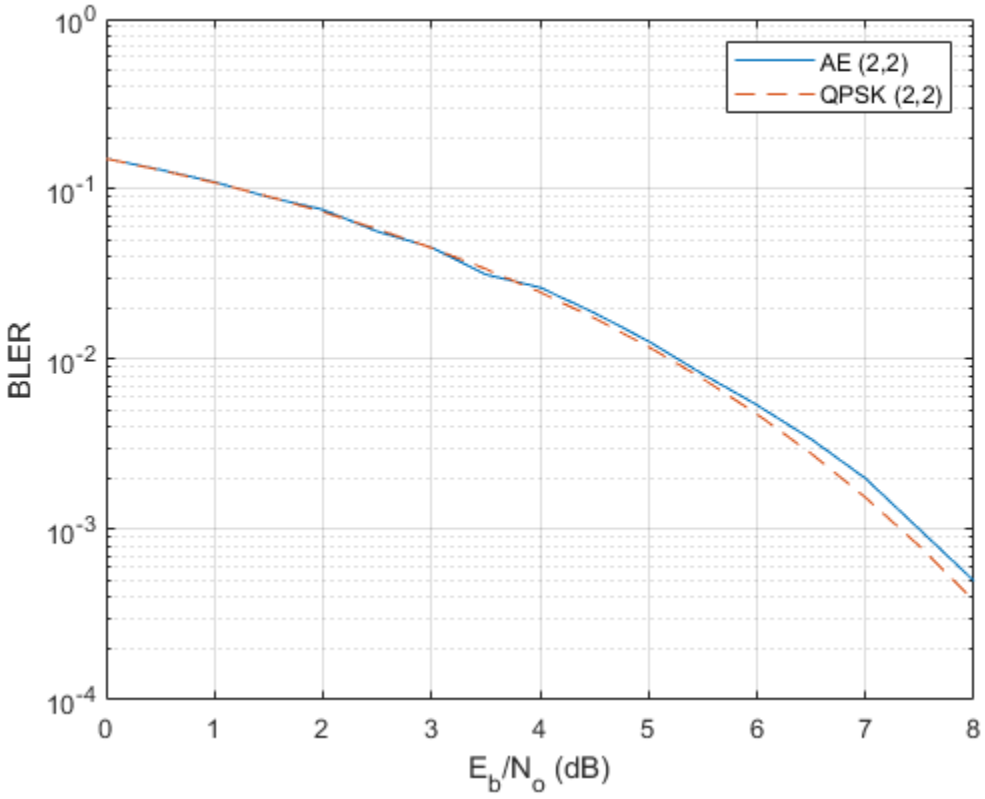
Pass the complex symbols through an AWGN channel. Decode the channel impaired complex symbols with the `helperAEWDecode.m` function. The following code runs the simulation for each E_b/N_o point for at least 10 block errors. To obtain more accurate results, increase minimum number of errors to at least 100. If Parallel Computing Toolbox™ is installed and a license is available, the simulation will run on a parallel pool. Compare the results with that of an uncoded QPSK system with block length 2.

```
EbNoVec = simParams.EbNoVec;
R = k/n;
```

```
M = 2^k;
BLER = zeros(size(EbNoVec));
parfor EbNoIdx = 1:length(EbNoVec)
  EbNo = EbNoVec(EbNoIdx) + 10*log10(R);
  chan = comm.AWGNChannel("BitsPerSymbol",2, ...
    "EbNo", EbNo, "SamplesPerSymbol", 1, "SignalPower", 1);

  numBlockErrors = 0;
  frameCnt = 0;
  while (numBlockErrors < simParams.MinNumErrors) ...
      && (frameCnt < simParams.MaxNumFrames) %#ok<PFBNS>

    d = randi([0 M-1],simParams.NumSymbolsPerFrame,1);    % Random information bits
    x = helperAEWEncode(d,txNet22e);                      % Encoder
    y = chan(x);                                          % Channel
    dHat = helperAEWDecode(y,rxNet22e);                   % Decoder

    numBlockErrors = numBlockErrors + sum(d ~= dHat);
    frameCnt = frameCnt + 1;
  end
  BLER(EbNoIdx) = numBlockErrors / (frameCnt*simParams.NumSymbolsPerFrame);
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
figure
semilogy(simParams.EbNoVec,BLER,'-')
hold on
qpsk22BLER = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^2;
semilogy(simParams.EbNoVec,qpsk22BLER,'--')
hold off
ylim([1e-4 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('AE (2,2)','QPSK (2,2)')
```



The well formed constellation together with the BLER results show that training for 15 epochs is enough to get a satisfactory convergence.

Compare Constellation Diagrams

Compare learned constellations of several autoencoders normalized to unit energy and unit average power. Train (2,4) autoencoder normalized to unit energy.

```
n = 2;      % number of channel uses
k = 4;      % bits per data symbol
EbNo = 3;   % dB
normalization = "Energy";
if trainNow
  [txNet24e,rxNet24e,info24e,wirelessAutoEncoder24e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n2_k4_energy txNet rxNet info trainedNet
  txNet24e = txNet;
  rxNet24e = rxNet;
  info24e = info;
  wirelessAutoEncoder24e = trainedNet;
end
```

Train (2,4) autoencoder normalized to unit average power.

```
n = 2;      % number of channel uses
k = 4;      % bits per data symbol
```

```
EbNo = 3;   % dB
normalization = "Average power";
if trainNow
  [txNet24p,rxNet24p,info24p,wirelessAutoEncoder24p] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n2_k4_power txNet rxNet info trainedNet
  txNet24p = txNet;
  rxNet24p = rxNet;
  info24p = info;
  wirelessAutoEncoder24p = trainedNet;
end
```

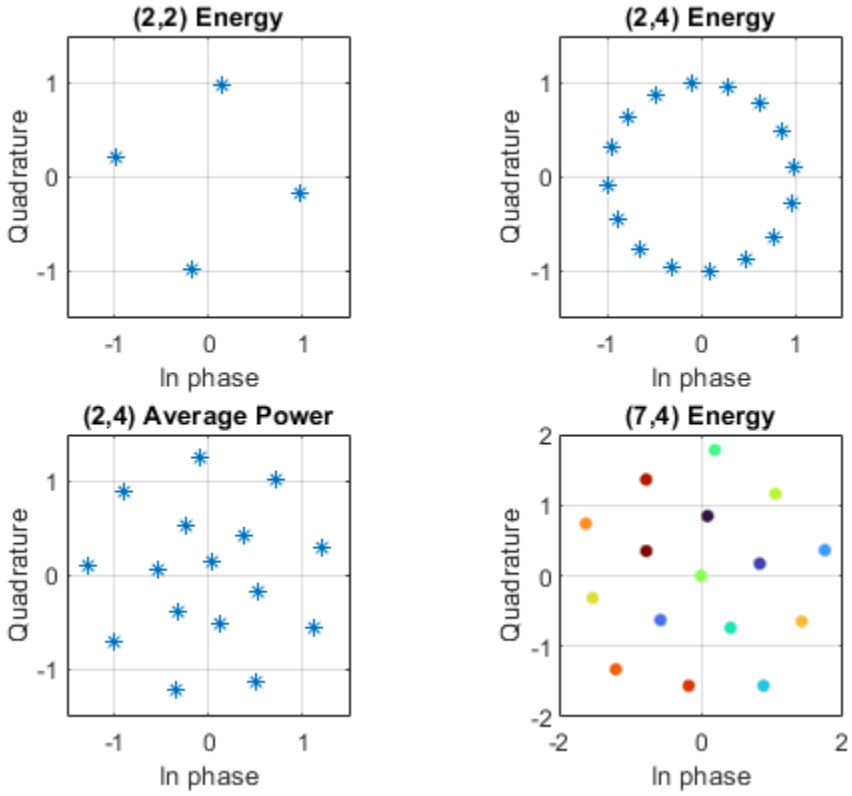
Train (7,4) autoencoder normalized to unit energy.

```
n = 7;      % number of channel uses
k = 4;      % bits per data symbol
EbNo = 3;   % dB
normalization = "Energy";
if trainNow
  [txNet74e,rxNet74e,info74e,wirelessAutoEncoder74e] = ...
    helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
else
  load trainedNet_n7_k4_energy txNet rxNet info trainedNet
  txNet74e = txNet;
  rxNet74e = rxNet;
  info74e = info;
  wirelessAutoEncoder74e = trainedNet;
end
```

Plot the constellation using the `helperAEWPlotConstellation.m` function. The trained (2,2) autoencoder converges on a QPSK constellation with a phase shift as the optimal constellation for the channel conditions experienced. The (2,4) autoencoder with energy normalization converges to a 16PSK constellation with a phase shift. Note that, energy normalization forces every symbol to have unit energy and places the symbols on the unit circle. Given this constraint, best constellation is a PSK constellation with equal angular distance between symbols. The (2,4) autoencoder with average power normalization converges to a three-tier constellation of 1-6-9 symbols. Average power normalization forces the symbols to have unity average power over time. This constraint results in an APSK constellation, which is different than the conventional QAM or APSK schemes. Note that, this network configuration may also converge to a two-tier constellation with 7-9 symbols based on the random initial condition used during training. The last plot shows the 2-D mapping of the 7-D constellation generated by the (7,4) autoencoder with energy constraint. 2-D mapping is obtained using the t-Distributed Stochastic Neighbor Embedding (t-SNE) method (see `tsne` (Statistics and Machine Learning Toolbox) function).

```
figure
subplot(2,2,1)
helperAEWPlotConstellation(txNet22e)
title('(2,2) Energy')
subplot(2,2,2)
helperAEWPlotConstellation(txNet24e)
title('(2,4) Energy')
subplot(2,2,3)
helperAEWPlotConstellation(txNet24p)
title('(2,4) Average Power')
subplot(2,2,4)
```

```
helperAEWPlotConstellation(txNet74e,'t-sne')
title('(7,4) Energy')
```

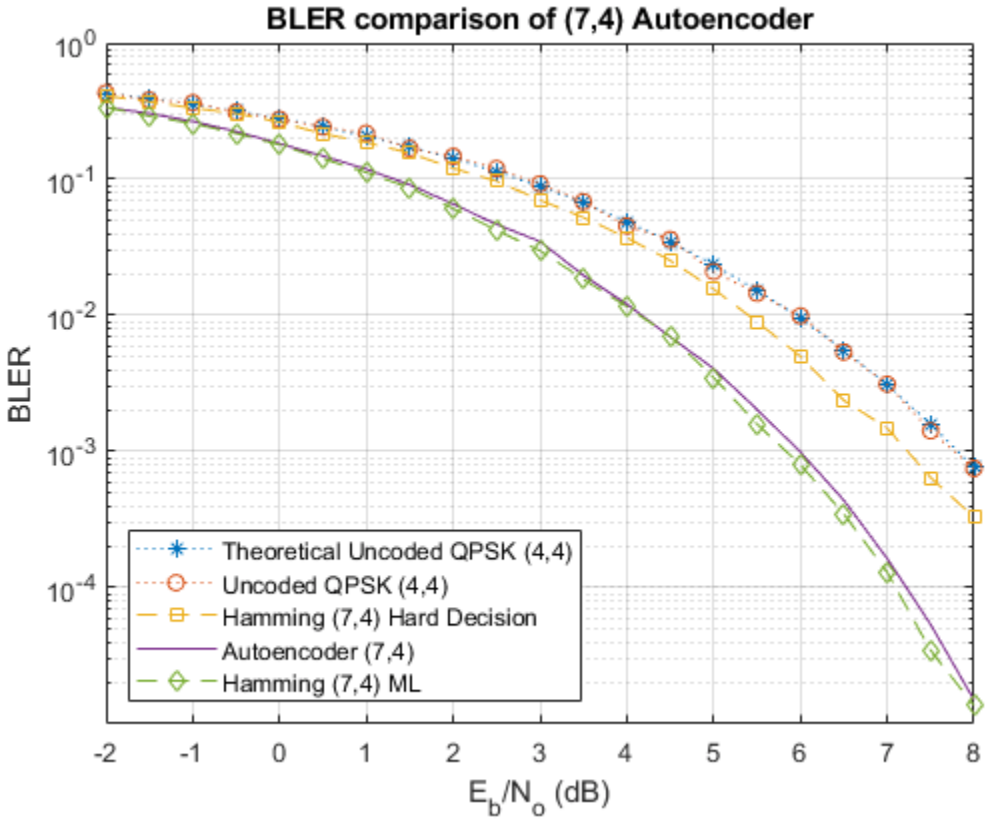


Compare BLER Performance of Autoencoders with Coded and Uncoded QPSK

Simulate the BLER performance of a (7,4) autoencoder with that of (7,4) Hamming code with QPSK modulation for both hard decision and maximum likelihood (ML) decoding. Use uncoded (4,4) QPSK as a baseline. (4,4) uncoded QPSK is basically a QPSK modulated system that sends blocks of 4 bits and measures BLER. The data for the following figures is obtained using helperAEWSimulateBLER.mlx and helperAEWPrepareAutoencoders.mlx files.

```
load codedBLERResults.mat
figure
qpsk44BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^4;
semilogy(simParams.EbNoVec,qpsk44BLERTh,':*')
hold on
semilogy(simParams.EbNoVec,qpsk44BLER,':o')
semilogy(simParams.EbNoVec,hammingHard74BLER,'--s')
semilogy(simParams.EbNoVec,ae74eBLER,'-')
semilogy(simParams.EbNoVec,hammingML74BLER,'--d')
hold off
ylim([1e-5 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('Theoretical Uncoded QPSK (4,4)','Uncoded QPSK (4,4)','Hamming (7,4) Hard Decision',...
```

```
  'Autoencoder (7,4)','Hamming (7,4) ML','Location','southwest')
title('BLER comparison of (7,4) Autoencoder')
```

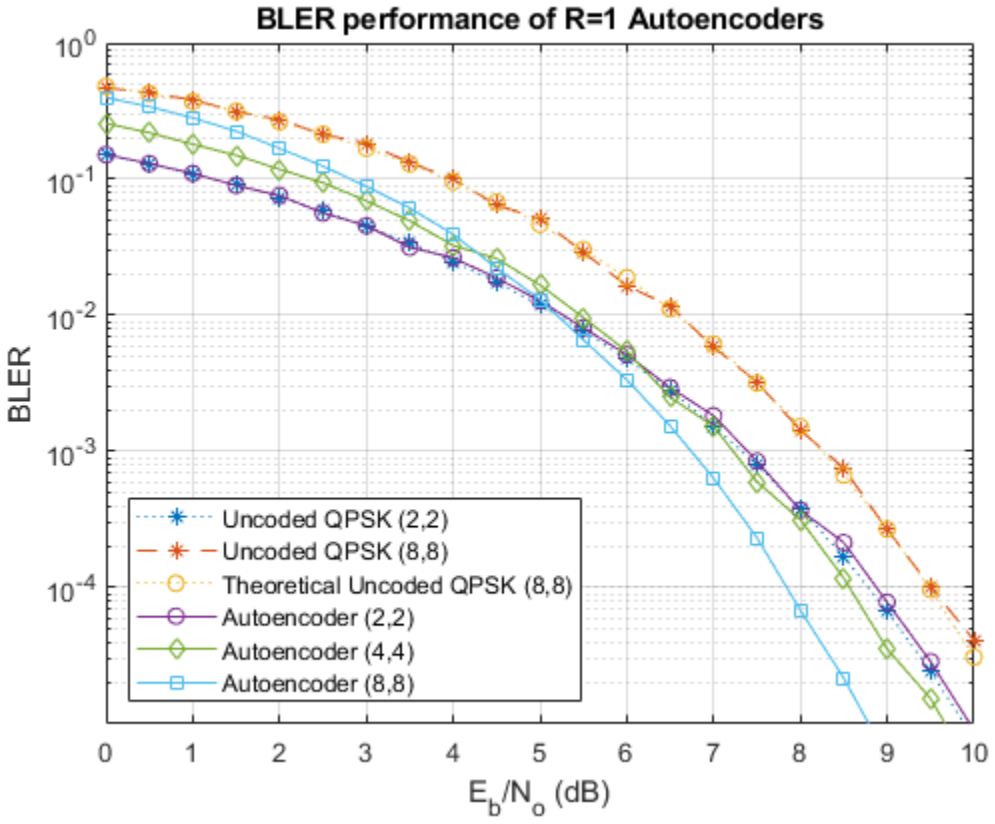


As expected, hard decision (7,4) Hamming code with QPSK modulation provides about 0.6 dB E_b/N_o advantage over uncoded QPSK, while the ML decoding of (7,4) Hamming code with QPSK modulation provides another 1.5 dB advantage for a BLER of 10^{-3}. The (7,4) autoencoder BLER performance approaches the ML decoding of (7,4) Hamming code, when trained with 3 dB E_b/N_o. This BLER performance shows that the autoencoder is able to learn not only modulation but also channel coding to achieve a coding gain of about 2 dB for a coding rate of R=4/7.

Next, simulate the BLER performance of autoencoders with R=1 with that of uncoded QPSK systems. Use uncoded (2,2) and (8,8) QPSK as baselines. Compare BLER performance of these systems with that of (2,2), (4,4) and (8,8) autoencoders.

```
load uncodedBLERResults.mat
qpsk22BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^2;
semilogy(simParams.EbNoVec,qpsk22BLERTh,':*')
hold on
semilogy(simParams.EbNoVec,qpsk88BLER,'--*')
qpsk88BLERTh = 1-(1-berawgn(simParams.EbNoVec,'psk',4,'nondiff')).^8;
semilogy(simParams.EbNoVec,qpsk88BLERTh,':o')
semilogy(simParams.EbNoVec,ae22eBLER,'-o')
semilogy(simParams.EbNoVec,ae44eBLER,'-d')
semilogy(simParams.EbNoVec,ae88eBLER,'-s')
hold off
ylim([1e-5 1])
```

```
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('Uncoded QPSK (2,2)','Uncoded QPSK (8,8)','Theoretical Uncoded QPSK (8,8)',...
  'Autoencoder (2,2)','Autoencoder (4,4)','Autoencoder (8,8)','Location','southwest')
title('BLER performance of R=1 Autoencoders')
```



Bit error rate of QPSK is the same for both (8,8) and (2,2) cases. However, the BLER depends on the block length, n, and gets worse as n increases as given by $BLER = 1 - (1 - BER)^n$. As expected, BLER performance of (8,8) QPSK is worse than the (2,2) QPSK system. The BLER performance of (2,2) autoencoder matches the BLER performance of (2,2) QPSK. On the other hand, (4,4) and (8,8) autoencoders optimize the channel coder and the constellation jointly to obtain a coding gain with respect to the corresponding uncoded QPSK systems.

Effect of Training Eb/No on BLER Performance

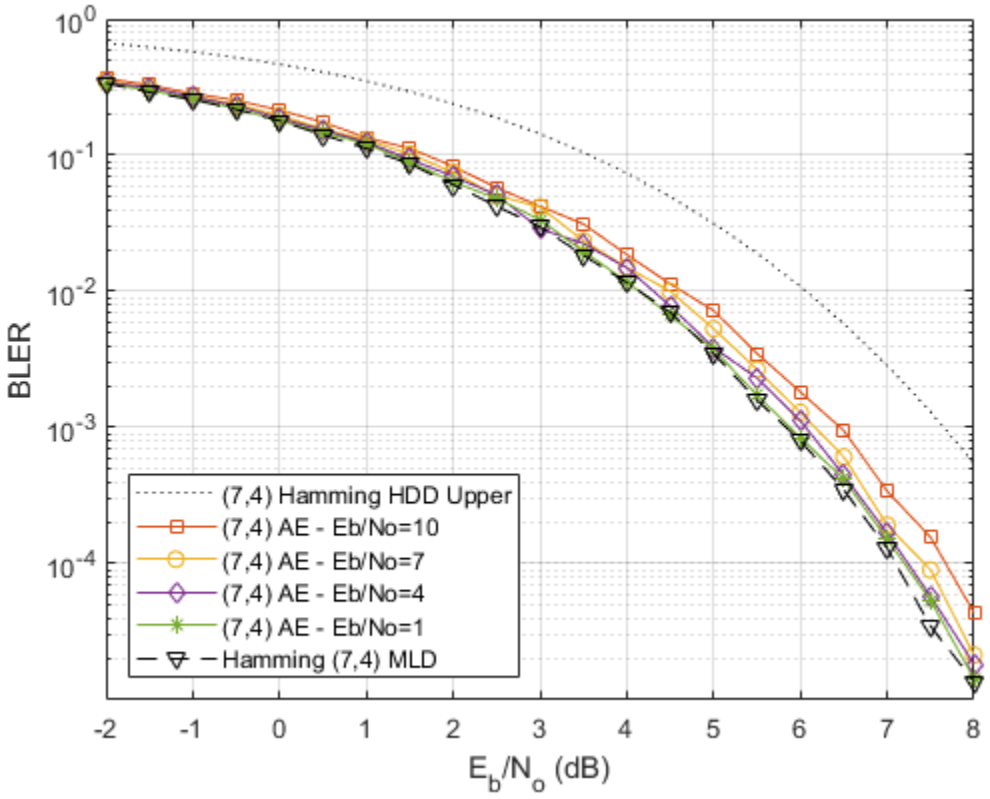
Train the (7,4) autoencoder with energy normalization under different E_b/N_o values and compare the BLER performance.

```
n = 7;
k = 4;
normalization = 'Energy';

EbNoVec = 1:3:10;
if trainNow
  for EbNoIdx = 1:length(EbNoVec)
    EbNo = EbNoVec(EbNoIdx);
```

```
    [txNetVec{EbNoIdx},rxNetVec{EbNoIdx},infoVec{EbNoIdx},trainedNetVec{EbNoIdx}] = ...
      helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo);
    BLERVec{EbNoIdx} = helperAEWAutoencoderBLER(txNetVec{EbNoIdx},rxNetVec{EbNoIdx},simParams);
  end
else
  load ae74TrainedEbNo1to10 BLERVec trainParams simParams txNetVec rxNetVec infoVec trainedNetVe
end
```

Plot the BLER performance together with theoretical upper bound for hard decision decoded Hamming (7,4) code and simulated BLER of maximum likelihood decoded (MLD) Hamming (7,4) code. The BLER performance of the (7,4) autoencoder gets closer to the Hamming (7,4) code with MLD as the training E_b/N_o decreases from 10 dB to 1 dB, at which point it almost matches the MLD Hamming (7,4) code.

```
berHamming = bercoding(simParams.EbNoVec,'hamming','hard',7);
blerHamming = 1-(1-berHamming).^7;
load codedBLERResults hammingML74BLER
figure
semilogy(simParams.EbNoVec,blerHamming,':k')
hold on
linespec = {'-*','-d','-o','-s',};
for EbNoIdx=length(EbNoVec):-1:1
  semilogy(simParams.EbNoVec,BLERVec{EbNoIdx},linespec{EbNoIdx})
end
semilogy(simParams.EbNoVec,hammingML74BLER,'--vk')
hold off
ylim([1e-5 1])
grid on
xlabel('E_b/N_o (dB)')
ylabel('BLER')
legend('(7,4) Hamming HDD Upper','(7,4) AE - Eb/No=10','(7,4) AE - Eb/No=7',...
  '(7,4) AE - Eb/No=4','(7,4) AE - Eb/No=1','Hamming (7,4) MLD','location','southwest')
```



Conclusions and Further Exploration

The BLER results show that it is possible for autoencoders to learn joint coding and modulation schemes in an unsupervised way. It is even possible to train an autoencoder with R=1 to obtain a coding gain as compared to traditional methods. The example also shows the effect of hyperparameters such as E_b/N_o on the BLER performance.

The results are obtained using the following default settings for training and BLER simulations:

```
trainParams.Plots = 'none';
trainParams.Verbose = false;
trainParams.MaxEpochs = 15;
trainParams.InitialLearnRate = 0.01;
trainParams.LearnRateSchedule = 'piecewise';
trainParams.LearnRateDropPeriod = 10;
trainParams.LearnRateDropFactor = 0.1;
trainParams.MiniBatchSize = 20*2^k;

simParams.EbNoVec = -2:0.5:8;
simParams.MinNumErrors = 100;
simParams.MaxNumFrames = 300;
simParams.NumSymbolsPerFrame = 10000;
simParams.SignalPower = 1;
```

Vary these parameters to train different autoencoders and test their BLER performance. Experiment with different n, k, normalization and E_b/N_o values. See the help for

helperAEWTrainWirelessAutoencoder.m, helperAEWPrepareAutoencoders.mlx and helperAEWAutoencoderBLER.m for more information.

List of helper functions

- helperAEWAWGNLayer.m
- helperAEWNormalizationLayer.m
- helperAEWEncode.m
- helperAEWDecode.m
- helperAEWTrainWirelessAutoencoder.m
- helperAEWPlotConstellation.m
- helperAEWPlotTrainingPerformance.m
- helperAEWAutoencoderBLER.m
- helperAEWPrepareAutoencoders.mlx
- helperAEWSimulateBLER.mlx
- helperAEWPlotReceivedConstellation.m

Wireless Autoencoder Training Function

This section shows the content of the helperAEWTrainWirelessAutoencoder function. To open the runnable version of the function in the MATLAB editor, click helperAEWTrainWirelessAutoencoder.m.

```
type helperAEWTrainWirelessAutoencoder
```

```
function [txNet,rxNet,info,trainedNet] = ...
  helperAEWTrainWirelessAutoencoder(n,k,normalization,EbNo,varargin)
%helperAEWTrainWirelessAutoencoder Train wireless autoencoder
%   [TX,RX,INFO,AE] = helperAEWTrainWirelessAutoencoder(N,K,NORM,EbNo)
%   trains an autoencoder, AE, with (N,K), where K is the number of input
%   bits and N is the number of channel uses. The autoencoder employs NORM
%   normalization. NORM must be one of 'Energy' and 'Average power'. The
%   channel is an AWGN channel with Eb/No set to EbNo. TX and Rx are the
%   encoder and decoder parts of the autoencoder that can be used in the
%   helperAEWEncoder and helperAEWDecoder functions, respectively. INFO is
%   the training information that can be used to check the convergence
%   behavior of the training process.
%
%   [TX,RX,INFO,AE] = helperAEWTrainWirelessAutoencoder(...,TP) provides
%   training parameters as follows:
%     TP.Plots: Plots to display during network training defined as one of
%               'none' (default) or 'training-progress'.
%     TP.Verbose: Indicator to display training progress information
%               defined as 1 (true) (default) or 0 (false).
%     TP.MaxEpochs: Maximum number of epochs defined as a positive integer.
%               The default is 15.
%     TP.InitialLearnRate: Initial learning rate as a floating point number
%               between 0 and 1. The default is 0.01;
%     TP.LearnRateSchedule: Learning rate schedule defined as one of
%               'piecewise' (default) or 'none'.
%     TP.LearnRateDropPeriod: Number of epochs for dropping the learning
%               rate as a positive integer. The default is 10.
%     TP.LearnRateDropFactor: Factor for dropping the learning rate,
%               defined as a scalar between 0 and 1. The default is 0.1.
```

```
%     TP.MiniBatchSize: Size of the mini-batch to use for each training
%               iteration, defined as a positive integer. The default is
%               20*M.
%
%   See also AutoencoderForWirelessCommunicationsExample, helperAEWEncode,
%   helperAEWDecode, helperAEWNormalizationLayer, helperAEWAWGNLayer.

%   Copyright 2020 The MathWorks, Inc.

% Derived parameters
M = 2^k;
R = k/n;

if nargin > 4
  trainParams = varargin{1};
else
  % Set default training options. Set maximum epochs to 15. SGD requires a
  % representative mini-batch that has enough symbols to achieve
  % convergence. Therefore, increase the mini-batch size with M. Set the
  % initial learning rate to 0.01 and reduce the learning rate by a factor
  % of 10 every 10 epochs. Do not plot or print training progress.
  trainParams.MaxEpochs = 15;
  trainParams.MiniBatchSize = 20*M;
  trainParams.InitialLearnRate = 0.01;
  trainParams.LearnRateSchedule = 'piecewise';
  trainParams.LearnRateDropPeriod = 10;
  trainParams.LearnRateDropFactor = 0.1;
  trainParams.Plots = 'none';
  trainParams.Verbose = false;
end

% Convert Eb/No to channel Eb/No values using the code rate
EbNoChannel = EbNo + 10*log10(R);

% As the number of possible input symbols increase, we need to increase the
% number of training symbols to give the network a chance to experience a
% large number of possible input combinations. The same is true for number
% of validation symbols.
numTrainSymbols = 2500 * M;
numValidationSymbols = 100 * M;

% Define autoencoder network. Input is a one-hot vector of length M. The
% encoder has two fully connected layers. The first one has M inputs and M
% outputs and is followed by an ReLU layer. The second fully connected
% layer has M inputs and n outputs and is followed by the normalization
% layer. Normalization layer imposes constraints on the encoder output and
% available methods are energy and average power normalization. The encoder
% layers are followed by the AWGN channel layer. Set BitsPerSymbol to 2
% since two output values are mapped onto a complex symbol. Set the signal
% power to 1 since the normalization layer outputs signals with unity
% power. The output of the channel is passed to the decoder layers. The
% first decoder layer is a fully connected layer that has n inputs and M
% outputs and is followed by an ReLU layer. Second fully connected layer
% has M inputs and M outputs and is followed by a softmax layer. The output
% of the decoder is chosen as the most probable transmitted symbol from 0
% to M-1.
wirelessAutoEncoder = [
  featureInputLayer(M,"Name","One-hot input","Normalization","none")
```

```
  fullyConnectedLayer(M,"Name","fc_1")
  reluLayer("Name","relu_1")

  fullyConnectedLayer(n,"Name","fc_2")

  helperAEWNormalizationLayer("Method", normalization)

  helperAEWAWGNLayer("NoiseMethod","EbNo",...
    "EbNo",EbNoChannel,...
    "BitsPerSymbol",2,...
    "SignalPower",1)

  fullyConnectedLayer(M,"Name","fc_3")
  reluLayer("Name","relu_2")

  fullyConnectedLayer(M,"Name","fc_4")
  softmaxLayer("Name","softmax")

  classificationLayer("Name","classoutput")];

% Generate random training data. Create one-hot input vectors and labels.
d = randi([0 M-1],numTrainSymbols,1);
trainSymbols = zeros(numTrainSymbols,M);
trainSymbols(sub2ind([numTrainSymbols, M],...
  (1:numTrainSymbols)',d+1)) = 1;
trainLabels = categorical(d);

% Generate random validation data. Create one-hot input vectors and labels.
d = randi([0 M-1],numValidationSymbols,1);
validationSymbols = zeros(numValidationSymbols,M);
validationSymbols(sub2ind([numValidationSymbols, M],...
  (1:numValidationSymbols)',d+1)) = 1;
validationLabels = categorical(d);

% Set training options
options = trainingOptions('adam', ...
  'InitialLearnRate',trainParams.InitialLearnRate, ...
  'MaxEpochs',trainParams.MaxEpochs, ...
  'MiniBatchSize',trainParams.MiniBatchSize, ...
  'Shuffle','every-epoch', ...
  'ValidationData',{validationSymbols,validationLabels}, ...
  'LearnRateSchedule', trainParams.LearnRateSchedule, ...
  'LearnRateDropPeriod', trainParams.LearnRateDropPeriod, ...
  'LearnRateDropFactor', trainParams.LearnRateDropFactor, ...
  'Plots', trainParams.Plots, ...
  'Verbose', trainParams.Verbose);

% Train the autoencoder network
[trainedNet,info] = trainNetwork(trainSymbols,trainLabels,wirelessAutoEncoder,options);

% Separate the network into encoder and decoder parts. Encoder starts with
% the input layer and ends after the normalization layer.
for idxNorm = 1:length(trainedNet.Layers)
  if isa(trainedNet.Layers(idxNorm), 'helperAEWNormalizationLayer')
    break
  end
end
```

```
lgraph = addLayers(layerGraph(trainedNet.Layers(1:idxNorm)), ...
  regressionLayer('Name', 'txout'));
lgraph = connectLayers(lgraph,'wnorm','txout');
txNet = assembleNetwork(lgraph);

% Decoder starts after the channel layer and ends with the classification
% layer. Add a feature input layer at the beginning.
for idxChan = idxNorm:length(trainedNet.Layers)
  if isa(trainedNet.Layers(idxChan), 'helperAEWAWGNLayer')
    break
  end
end
firstLayerName = trainedNet.Layers(idxChan+1).Name;
n = trainedNet.Layers(idxChan+1).InputSize;
lgraph = addLayers(layerGraph(featureInputLayer(n,'Name','rxin')), ...
  trainedNet.Layers(idxChan+1:end));
lgraph = connectLayers(lgraph,'rxin',firstLayerName);
rxNet = assembleNetwork(lgraph);
```

References

[1] T. O’Shea and J. Hoydis, "An Introduction to Deep Learning for the Physical Layer," in IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563-575, Dec. 2017, doi: 10.1109/TCCN.2017.2758370.

[2] S. Dörner, S. Cammerer, J. Hoydis and S. t. Brink, "Deep Learning Based Communication Over the Air," in *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 132-143, Feb. 2018, doi: 10.1109/JSTSP.2017.2784180.

See Also

`classificationLayer` | `featureInputLayer` | `fullyConnectedLayer` | `reluLayer` | `softmaxLayer`

More About

- “Deep Learning in MATLAB” on page 1-2

Modulation Classification with Deep Learning

This example shows how to use a convolutional neural network (CNN) for modulation classification. You generate synthetic, channel-impaired waveforms. Using the generated waveforms as training data, you train a CNN for modulation classification. You then test the CNN with software-defined radio (SDR) hardware and over-the-air signals.

Predict Modulation Type Using CNN

The trained CNN in this example recognizes these eight digital and three analog modulation types:

- Binary phase shift keying (BPSK)
- Quadrature phase shift keying (QPSK)
- 8-ary phase shift keying (8-PSK)
- 16-ary quadrature amplitude modulation (16-QAM)
- 64-ary quadrature amplitude modulation (64-QAM)
- 4-ary pulse amplitude modulation (PAM4)
- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast FM (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

```
modulationTypes = categorical(["BPSK", "QPSK", "8PSK", ...
  "16QAM", "64QAM", "PAM4", "GFSK", "CPFSK", ...
  "B-FM", "DSB-AM", "SSB-AM"]);
```

First, load the trained network. For details on network training, see the Training a CNN on page 13-0 section.

```
load trainedModulationClassificationNetwork
trainedNet
```

```
trainedNet = SeriesNetwork with properties:
         Layers: [28×1 nnet.cnn.layer.Layer]
     InputNames: {'Input Layer'}
    OutputNames: {'Output'}
```

The trained CNN takes 1024 channel-impaired samples and predicts the modulation type of each frame. Generate several PAM4 frames that are impaired with Rician multipath fading, center frequency and sampling time drift, and AWGN. Use following function to generate synthetic signals to test the CNN. Then use the CNN to predict the modulation type of the frames.

- `randi`: Generate random bits
- `pammod` (Communications Toolbox) PAM4-modulate the bits
- `rcosdesign` (Signal Processing Toolbox): Design a square-root raised cosine pulse shaping filter
- `filter`: Pulse shape the symbols
- `comm.RicianChannel` (Communications Toolbox): Apply Rician multipath channel

- `comm.PhaseFrequencyOffset` (Communications Toolbox): Apply phase and/or frequency shift due to clock offset
- `interp1`: Apply timing drift due to clock offset
- `awgn` (Communications Toolbox): Add AWGN

```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
rng(123456)
% Random bits
d = randi([0 3], 1024, 1);
% PAM4 modulation
syms = pammod(d,4);
% Square-root raised cosine filter
filterCoeffs = rcosdesign(0.35,4,8);
tx = filter(filterCoeffs,1,upsample(syms,8));

% Channel
SNR = 30;
maxOffset = 5;
fc = 902e6;
fs = 200e3;
multipathChannel = comm.RicianChannel(...
  'SampleRate', fs, ...
  'PathDelays', [0 1.8 3.4] / 200e3, ...
  'AveragePathGains', [0 -2 -10], ...
  'KFactor', 4, ...
  'MaximumDopplerShift', 4);

frequencyShifter = comm.PhaseFrequencyOffset(...
  'SampleRate', fs);

% Apply an independent multipath channel
reset(multipathChannel)
outMultipathChan = multipathChannel(tx);

% Determine clock offset factor
clockOffset = (rand() * 2*maxOffset) - maxOffset;
C = 1 + clockOffset / 1e6;

% Add frequency offset
frequencyShifter.FrequencyOffset = -(C-1)*fc;
outFreqShifter = frequencyShifter(outMultipathChan);

% Add sampling time drift
t = (0:length(tx)-1)' / fs;
newFs = fs * C;
tp = (0:length(tx)-1)' / newFs;
outTimeDrift = interp1(t, outFreqShifter, tp);

% Add noise
rx = awgn(outTimeDrift,SNR,0);

% Frame generation for classification
unknownFrames = helperModClassGetNNFrames(rx);

% Classification
[prediction1,score1] = classify(trainedNet,unknownFrames);
```

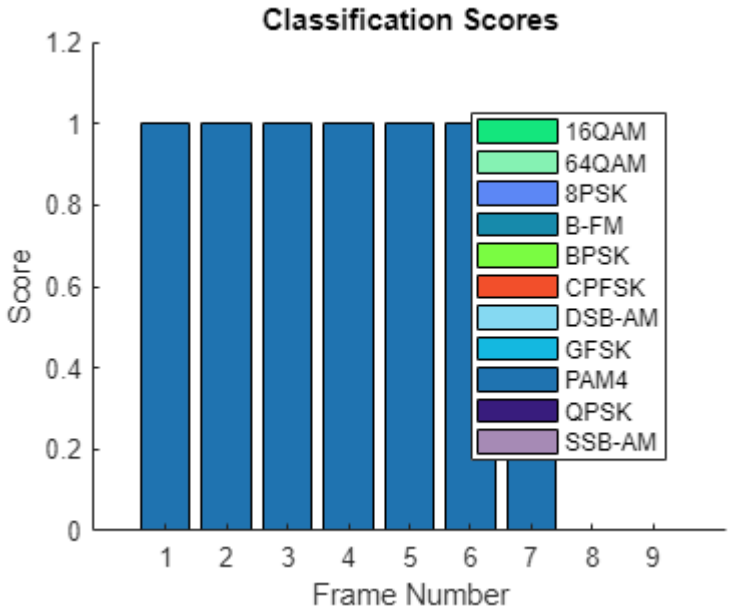
Return the classifier predictions, which are analogous to hard decisions. The network correctly identifies the frames as PAM4 frames. For details on the generation of the modulated signals, see helperModClassGetModulator function.

```
prediction1
```

```
prediction1 = 7×1 categorical
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
     PAM4
```

The classifier also returns a vector of scores for each frame. The score corresponds to the probability that each frame has the predicted modulation type. Plot the scores.

```
helperModClassPlotScores(score1,modulationTypes)
```



Before we can use a CNN for modulation classification, or any other task, we first need to train the network with known (or labeled) data. The first part of this example shows how to use Communications Toolbox features, such as modulators, filters, and channel impairments, to generate synthetic training data. The second part focuses on defining, training, and testing the CNN for the task of modulation classification. The third part tests the network performance with over-the-air signals using software defined radio (SDR) platforms.

Waveform Generation for Training

Generate 10,000 frames for each modulation type, where 80% is used for training, 10% is used for validation and 10% is used for testing. We use training and validation frames during the network training phase. Final classification accuracy is obtained using test frames. Each frame is 1024 samples long and has a sample rate of 200 kHz. For digital modulation types, eight samples represent

a symbol. The network makes each decision based on single frames rather than on multiple consecutive frames (as in video). Assume a center frequency of 902 MHz and 100 MHz for the digital and analog modulation types, respectively.

To run this example quickly, use the trained network and generate a small number of training frames. To train the network on your computer, choose the "Train network now" option (i.e. set trainNow to true).

```
trainNow = Use trained network ;
if trainNow == true
  numFramesPerModType = 10000;
else
  numFramesPerModType = 200;
end
percentTrainingSamples = 80;
percentValidationSamples = 10;
percentTestSamples = 10;

sps = 8;                % Samples per symbol
spf = 1024;             % Samples per frame
symbolsPerFrame = spf / sps;
fs = 200e3;             % Sample rate
fc = [902e6 100e6];     % Center frequencies
```

Create Channel Impairments

Pass each frame through a channel with

- AWGN
- Rician multipath fading
- Clock offset, resulting in center frequency offset and sampling time drift

Because the network in this example makes decisions based on single frames, each frame must pass through an independent channel.

AWGN

The channel adds AWGN with an SNR of 30 dB. Implement the channel using `awgn` (Communications Toolbox) function.

Rician Multipath

The channel passes the signals through a Rician multipath fading channel using the `comm.RicianChannel` (Communications Toolbox) System object. Assume a delay profile of [0 1.8 3.4] samples with corresponding average path gains of [0 -2 -10] dB. The K-factor is 4 and the maximum Doppler shift is 4 Hz, which is equivalent to a walking speed at 902 MHz. Implement the channel with the following settings.

Clock Offset

Clock offset occurs because of the inaccuracies of internal clock sources of transmitters and receivers. Clock offset causes the center frequency, which is used to downconvert the signal to baseband, and the digital-to-analog converter sampling rate to differ from the ideal values. The channel simulator uses the clock offset factor C, expressed as $C = 1 + \frac{\Delta_{\text{clock}}}{10^6}$, where Δ_{clock} is the clock

offset. For each frame, the channel generates a random Δ_{clock} value from a uniformly distributed set of values in the range $[-\max\Delta_{\text{clock}}\ \max\Delta_{\text{clock}}]$, where $\max\Delta_{\text{clock}}$ is the maximum clock offset. Clock offset is measured in parts per million (ppm). For this example, assume a maximum clock offset of 5 ppm.

```
maxDeltaOff = 5;
deltaOff = (rand()*2*maxDeltaOff) - maxDeltaOff;
C = 1 + (deltaOff/1e6);
```

Frequency Offset

Subject each frame to a frequency offset based on clock offset factor C and the center frequency. Implement the channel using `comm.PhaseFrequencyOffset` (Communications Toolbox).

Sampling Rate Offset

Subject each frame to a sampling rate offset based on clock offset factor C. Implement the channel using the `interp1` function to resample the frame at the new rate of $C \times f_s$.

Combined Channel

Use the helperModClassTestChannel object to apply all three channel impairments to the frames.

```
channel = helperModClassTestChannel(...
  'SampleRate', fs, ...
  'SNR', SNR, ...
  'PathDelays', [0 1.8 3.4] / fs, ...
  'AveragePathGains', [0 -2 -10], ...
  'KFactor', 4, ...
  'MaximumDopplerShift', 4, ...
  'MaximumClockOffset', 5, ...
  'CenterFrequency', 902e6)
```

```
channel = helperModClassTestChannel with properties:
                    SNR: 30
         CenterFrequency: 902000000
              SampleRate: 200000
              PathDelays: [0 9.0000e-06 1.7000e-05]
        AveragePathGains: [0 -2 -10]
                 KFactor: 4
     MaximumDopplerShift: 4
      MaximumClockOffset: 5
```

You can view basic information about the channel using the info object function.

```
chInfo = info(channel)
```

```
chInfo = struct with fields:
               ChannelDelay: 6
      MaximumFrequencyOffset: 4510
     MaximumSampleRateOffset: 1
```

Waveform Generation

Create a loop that generates channel-impaired frames for each modulation type and stores the frames with their corresponding labels in MAT files. By saving the data into files, you eliminate the need to generate the data every time you run this example. You can also share the data more effectively.

Remove a random number of samples from the beginning of each frame to remove transients and to make sure that the frames have a random starting point with respect to the symbol boundaries.

```
% Set the random number generator to a known state to be able to regenerate
% the same frames every time the simulation is run
rng(1235)
tic

numModulationTypes = length(modulationTypes);

channelInfo = info(channel);
transDelay = 50;
dataDirectory = fullfile(tempdir,"ModClassDataFiles");
disp("Data file directory is " + dataDirectory)
```

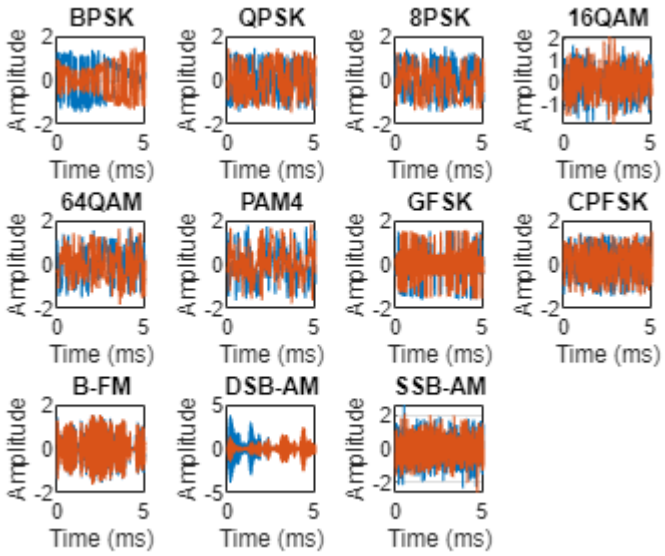
```
Data file directory is C:\TEMP\ModClassDataFiles
```

```
fileNameRoot = "frame";

% Check if data files exist
dataFilesExist = false;
if exist(dataDirectory,'dir')
  files = dir(fullfile(dataDirectory,sprintf("%s*",fileNameRoot)));
  if length(files) == numModulationTypes*numFramesPerModType
    dataFilesExist = true;
  end
end

if ~dataFilesExist
  disp("Generating data and saving in data files...")
  [success,msg,msgID] = mkdir(dataDirectory);
  if ~success
    error(msgID,msg)
  end
  for modType = 1:numModulationTypes
    elapsedTime = seconds(toc);
    elapsedTime.Format = 'hh:mm:ss';
    fprintf('%s - Generating %s frames\n', ...
      elapsedTime, modulationTypes(modType))

    label = modulationTypes(modType);
    numSymbols = (numFramesPerModType / sps);
    dataSrc = helperModClassGetSource(modulationTypes(modType), sps, 2*spf, fs);
    modulator = helperModClassGetModulator(modulationTypes(modType), sps, fs);
    if contains(char(modulationTypes(modType)), {'B-FM','DSB-AM','SSB-AM'})
      % Analog modulation types use a center frequency of 100 MHz
      channel.CenterFrequency = 100e6;
    else
      % Digital modulation types use a center frequency of 902 MHz
      channel.CenterFrequency = 902e6;
    end
```

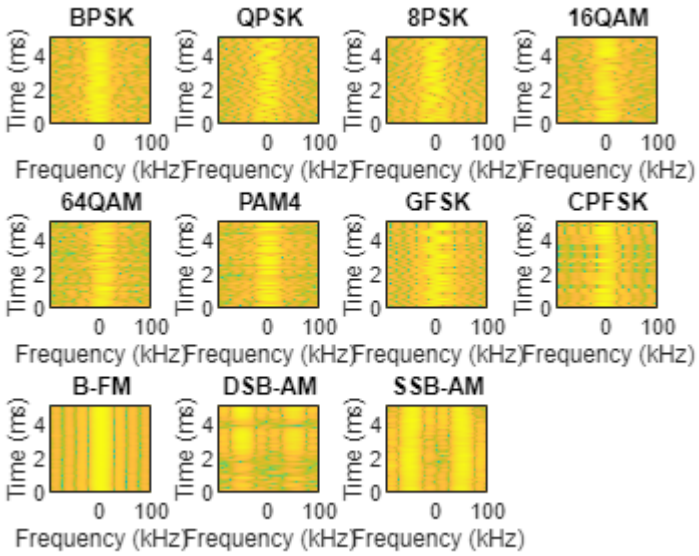
```
    for p=1:numFramesPerModType
      % Generate random data
      x = dataSrc();

      % Modulate
      y = modulator(x);

      % Pass through independent channels
      rxSamples = channel(y);

      % Remove transients from the beginning, trim to size, and normalize
      frame = helperModClassFrameGenerator(rxSamples, spf, spf, transDelay, sps);

      % Save data file
      fileName = fullfile(dataDirectory,...
        sprintf("%s%s%03d",fileNameRoot,modulationTypes(modType),p));
      save(fileName,"frame","label")
    end
  end
else
  disp("Data files exist. Skip data generation.")
end
```

```
Generating data and saving in data files...

00:00:00 - Generating BPSK frames
00:00:01 - Generating QPSK frames
00:00:02 - Generating 8PSK frames
00:00:04 - Generating 16QAM frames
00:00:05 - Generating 64QAM frames
00:00:06 - Generating PAM4 frames
00:00:08 - Generating GFSK frames
00:00:09 - Generating CPFSK frames
00:00:11 - Generating B-FM frames
00:00:12 - Generating DSB-AM frames
00:00:13 - Generating SSB-AM frames
```

```
% Plot the amplitude of the real and imaginary parts of the example frames
% against the sample number
helperModClassPlotTimeDomain(dataDirectory,modulationTypes,fs)
```




```
% Plot the spectrogram of the example frames
helperModClassPlotSpectrogram(dataDirectory,modulationTypes,fs,sps)
```



Create a Datastore

Use a `signalDatastore` object to manage the files that contain the generated complex waveforms. Datastores are especially useful when each individual file fits in memory, but the entire collection does not necessarily fit.

```
frameDS = signalDatastore(dataDirectory,'SignalVariableNames',["frame","label"]);
```

Transform Complex Signals to Real Arrays

The deep learning network in this example expects real inputs while the received signal has complex baseband samples. Transform the complex signals into real valued 4-D arrays. The output frames have size 1-by-spf-by-2-by-N, where the first page (3rd dimension) is in-phase samples and the second page is quadrature samples. When the convolutional filters are of size 1-by-spf, this approach ensures that the information in the I and Q gets mixed even in the convolutional layers and makes better use of the phase information. See helperModClassIQAsPages for details.

```
frameDSTrans = transform(frameDS,@helperModClassIQAsPages);
```

Split into Training, Validation, and Test

Next divide the frames into training, validation, and test data. See helperModClassSplitData for details.

```
splitPercentages = [percentTrainingSamples,percentValidationSamples,percentTestSamples];
[trainDSTrans,validDSTrans,testDSTrans] = helperModClassSplitData(frameDSTrans,splitPercentages);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Import Data Into Memory

Neural network training is iterative. At every iteration, the datastore reads data from files and transforms the data before updating the network coefficients. If the data fits into the memory of your computer, importing the data from the files into the memory enables faster training by eliminating this repeated read from file and transform process. Instead, the data is read from the files and transformed once. Training this network using data files on disk takes about 110 minutes while training using in-memory data takes about 50 min.

Import all the data in the files into memory. The files have two variables: `frame` and `label` and each `read` call to the datastore returns a cell array, where the first element is the `frame` and the second element is the `label`. Use the `transform` functions helperModClassReadFrame and helperModClassReadLabel to read frames and labels. Use `readall` with `"UseParallel"` option set to `true` to enable parallel processing of the transform functions, in case you have Parallel Computing Toolbox license. Since `readall` function, by default, concatenates the output of the `read` function over the first dimension, return the frames in a cell array and manually concatenate over the 4th dimension.

```
% Read the training and validation frames into the memory
pctExists = parallelComputingLicenseExists();
trainFrames = transform(trainDSTrans, @helperModClassReadFrame);
rxTrainFrames = readall(trainFrames,"UseParallel",pctExists);
rxTrainFrames = cat(4, rxTrainFrames{:});
validFrames = transform(validDSTrans, @helperModClassReadFrame);
rxValidFrames = readall(validFrames,"UseParallel",pctExists);
rxValidFrames = cat(4, rxValidFrames{:});

% Read the training and validation labels into the memory
trainLabels = transform(trainDSTrans, @helperModClassReadLabel);
rxTrainLabels = readall(trainLabels,"UseParallel",pctExists);
validLabels = transform(validDSTrans, @helperModClassReadLabel);
rxValidLabels = readall(validLabels,"UseParallel",pctExists);
```

Train the CNN

This example uses a CNN that consists of six convolution layers and one fully connected layer. Each convolution layer except the last is followed by a batch normalization layer, rectified linear unit (ReLU) activation layer, and max pooling layer. In the last convolution layer, the max pooling layer is replaced with an average pooling layer. The output layer has softmax activation. For network design guidance, see “Deep Learning Tips and Tricks” on page 1-67.

```
modClassNet = helperModClassCNN(modulationTypes,sps,spf);
```

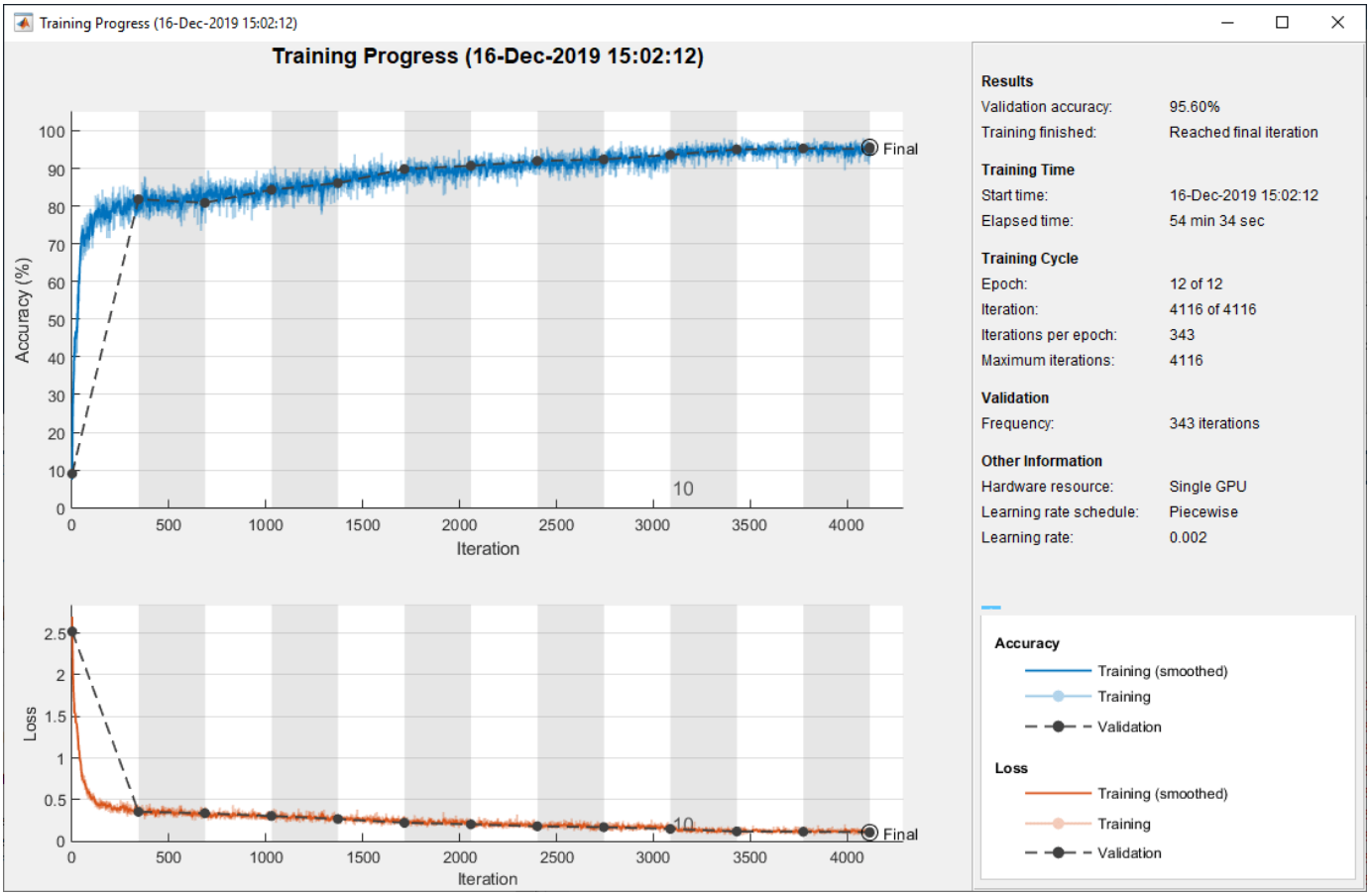
Next configure `TrainingOptionsSGDM` to use an SGDM solver with a mini-batch size of 256. Set the maximum number of epochs to 12, since a larger number of epochs provides no further training advantage. By default, the `'ExecutionEnvironment'` property is set to `'auto'`, where the `trainNetwork` function uses a GPU if one is available or uses the CPU, if not. To use the GPU, you must have a Parallel Computing Toolbox license. Set the initial learning rate to $2x10^{-2}$. Reduce the learning rate by a factor of 10 every 9 epochs. Set `'Plots'` to `'training-progress'` to plot the training progress. On an NVIDIA Titan Xp GPU, the network takes approximately 25 minutes to train. .

```
maxEpochs = 12;
miniBatchSize = 256;
options = helperModClassTrainingOptions(maxEpochs,miniBatchSize,...
  numel(rxTrainLabels),rxValidFrames,rxValidLabels);
```

Either train the network or use the already trained network. By default, this example uses the trained network.

```
if trainNow == true
  elapsedTime = seconds(toc);
  elapsedTime.Format = 'hh:mm:ss';
  fprintf('%s - Training the network\n', elapsedTime)
  trainedNet = trainNetwork(rxTrainFrames,rxTrainLabels,modClassNet,options);
else
  load trainedModulationClassificationNetwork
end
```

As the plot of the training progress shows, the network converges in about 12 epochs to more than 95% accuracy.



Evaluate the trained network by obtaining the classification accuracy for the test frames. The results show that the network achieves about 94% accuracy for this group of waveforms.

```
elapsedTime = seconds(toc);
elapsedTime.Format = 'hh:mm:ss';
fprintf('%s - Classifying test frames\n', elapsedTime)
```

```
00:01:25 - Classifying test frames
```

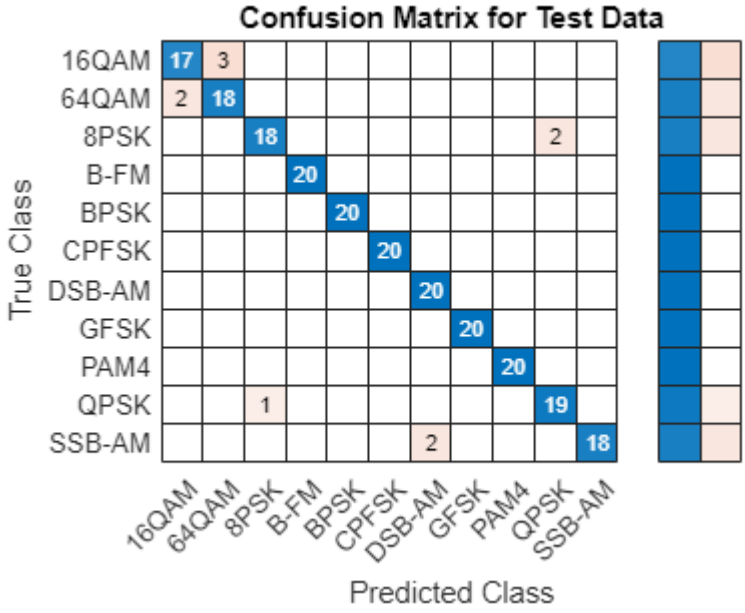
```
% Read the test frames into the memory
testFrames = transform(testDSTrans, @helperModClassReadFrame);
rxTestFrames = readall(testFrames,"UseParallel",pctExists);
rxTestFrames = cat(4, rxTestFrames{:});

% Read the test labels into the memory
testLabels = transform(testDSTrans, @helperModClassReadLabel);
rxTestLabels = readall(testLabels,"UseParallel",pctExists);

rxTestPred = classify(trainedNet,rxTestFrames);
testAccuracy = mean(rxTestPred == rxTestLabels);
disp("Test accuracy: " + testAccuracy*100 + "%")
```

```
Test accuracy: 95.4545%
```

Plot the confusion matrix for the test frames. As the matrix shows, the network confuses 16-QAM and 64-QAM frames. This problem is expected since each frame carries only 128 symbols and 16-QAM is a

subset of 64-QAM. The network also confuses QPSK and 8-PSK frames, since the constellations of these modulation types look similar once phase-rotated due to the fading channel and frequency offset.

```
figure
cm = confusionchart(rxTestLabels, rxTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
cm.Parent.Position = [cm.Parent.Position(1:2) 740 424];
```



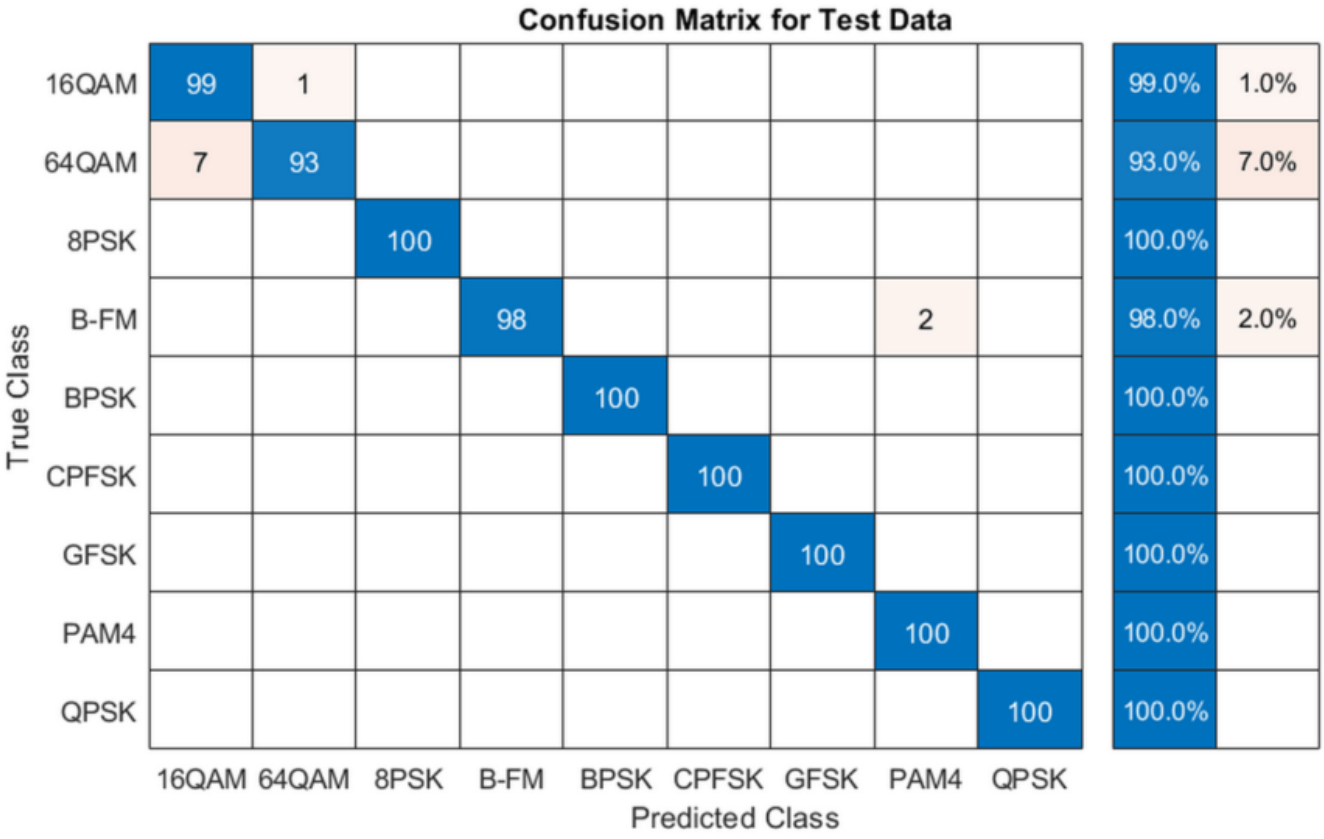
Test with SDR

Test the performance of the trained network with over-the-air signals using the helperModClassSDRTest function. To perform this test, you must have dedicated SDRs for transmission and reception. You can use two ADALM-PLUTO radios, or one ADALM-PLUTO radio for transmission and one USRP® radio for reception. You must install Communications Toolbox Support Package for ADALM-PLUTO Radio. If you are using a USRP® radio, you must also install Communications Toolbox Support Package for USRP® Radio. The `helperModClassSDRTest` function uses the same modulation functions as used for generating the training signals, and then transmits them using an ADALM-PLUTO radio. Instead of simulating the channel, capture the channel-impaired signals using the SDR that is configured for signal reception (ADALM-PLUTO or USRP® radio). Use the trained network with the same `classify` function used previously to predict the modulation type. Running the next code segment produces a confusion matrix and prints out the test accuracy.

```
radioPlatform = "ADALM-PLUTO";

switch radioPlatform
  case "ADALM-PLUTO"
    if helperIsPlutoSDRInstalled() == true
      radios = findPlutoRadio();
      if length(radios) >= 2
        helperModClassSDRTest(radios);
```

```
      else
        disp('Selected radios not found. Skipping over-the-air test.')
      end
    end
  case {"USRP B2xx","USRP X3xx","USRP N2xx"}
    if (helperIsUSRPInstalled() == true) && (helperIsPlutoSDRInstalled() == true)
      txRadio = findPlutoRadio();
      rxRadio = findsdru();
      switch radioPlatform
        case "USRP B2xx"
          idx = contains({rxRadio.Platform}, {'B200','B210'});
        case "USRP X3xx"
          idx = contains({rxRadio.Platform}, {'X300','X310'});
        case "USRP N2xx"
          idx = contains({rxRadio.Platform}, 'N200/N210/USRP2');
      end
      rxRadio = rxRadio(idx);
      if (length(txRadio) >= 1) && (length(rxRadio) >= 1)
        helperModClassSDRTest(rxRadio);
      else
        disp('Selected radios not found. Skipping over-the-air test.')
      end
    end
end
```

When using two stationary ADALM-PLUTO radios separated by about 2 feet, the network achieves 99% overall accuracy with the following confusion matrix. Results will vary based on experimental setup.



Further Exploration

It is possible to optimize the hyperparameters parameters, such as number of filters, filter size, or optimize the network structure, such as adding more layers, using different activation layers, etc. to improve the accuracy.

Communication Toolbox provides many more modulation types and channel impairments. For more information see “Modulation” (Communications Toolbox) and “Propagation and Channel Models” (Communications Toolbox) sections. You can also add standard specific signals with LTE Toolbox, WLAN Toolbox, and 5G Toolbox. You can also add radar signals with Phased Array System Toolbox.

helperModClassGetModulator function provides the MATLAB functions used to generate modulated signals. You can also explore the following functions and System objects for more details:

- helperModClassGetModulator.m
- helperModClassTestChannel.m
- helperModClassGetSource.m
- helperModClassFrameGenerator.m
- helperModClassCNN.m
- helperModClassTrainingOptions.m

References

1 O'Shea, T. J., J. Corgan, and T. C. Clancy. "Convolutional Radio Modulation Recognition Networks." Preprint, submitted June 10, 2016. https://arxiv.org/abs/1602.04105

2 O'Shea, T. J., T. Roy, and T. C. Clancy. "Over-the-Air Deep Learning Based Radio Signal Classification." IEEE Journal of Selected Topics in Signal Processing. Vol. 12, Number 1, 2018, pp. 168–179.

3 Liu, X., D. Yang, and A. E. Gamal. "Deep Neural Network Architectures for Modulation Classification." Preprint, submitted January 5, 2018. https://arxiv.org/abs/1712.00443v3

See Also

`trainingOptions` | `trainNetwork`

More About

• “Deep Learning in MATLAB” on page 1-2

Training and Testing a Neural Network for LLR Estimation

This example shows how to generate signals and channel impairments to train a neural network, called LLRNet, to estimate exact log likelihood ratios (LLR).

Most modern communication systems, such as 5G New Radio (NR) and Digital Video Broadcasting for Satellite, Second Generation (DVB-S.2) use forward error correction algorithms that benefit from soft demodulated bit values. These systems calculate soft bit values using the LLR approach. LLR is defined as the log of the ratio of probability of a bit to be 0 to the probability of a bit to be 1 or

$$l_i \triangleq \log\left(\frac{P_r(c_i = 0 \,|\, \hat{s})}{P_r(c_i = 1 \,|\, \hat{s})}\right),\ i = 1, \ldots, k$$

where $\hat{s}$ is an k-bit received symbol, and c_i is the i^{th}bit of the symbol. Assuming an additive white Gaussian noise (AWGN) channel, the exact computation of the LLR expression is

$$l_i \triangleq \log\left(\frac{\sum_{s \in c_i^0} \exp\left(-\frac{\|\hat{s}-s\|_2^2}{\sigma^2}\right)}{\sum_{s \in c_i^1} \exp\left(-\frac{\|\hat{s}-s\|_2^2}{\sigma^2}\right)}\right)$$

where σ^2 is the noise variance. Exponential and logarithmic calculations are very costly especially in embedded systems. Therefore, most practical systems use the max-log approximation. For a given array x, the max-log approximation is

$$\log\left(\sum_j \exp\left(-x_j^2\right)\right) \approx \max_j\left(-x_j^2\right).$$

Substituting this in the exact LLR expression results in the max-log LLR approximation [1] on page 13-0

$$l_i \approx \frac{1}{\sigma^2}\left(\min_{s \in C_i^1} \|\hat{s}-s\|_2^2 - \min_{s \in C_i^0} \|\hat{s}-s\|_2^2\right).$$

LLRNet uses a neural network to estimate the exact LLR values given the baseband complex received symbol for a given SNR value. A shallow network with a small number of hidden layers has the potential to estimate the exact LLR values at a complexity similar to the approximate LLR algorithm [1] on page 13-0 .

Compare Exact LLR, Max-Log Approximate LLR and LLRNet for M-ary QAM

5G NR uses M-ary QAM modulation. This section explores the accuracy of LLRNet in estimating the LLR values for 16-, 64-, and 256-QAM modulation. Assume an M-ary QAM system that operates under AWGN channel conditions. This assumption is valid even when the channel is frequency selective but symbols are equalized. The following shows calculated LLR values for the following three algorithms:

- Exact LLR
- Max-log approximate LLR

- LLRNet

16-QAM LLR Estimation Performance

Calculate exact and approximate LLR values for symbol values that cover the 99.7% ($\pm 3\sigma$) of the possible received symbols. Assuming AWGN, 99.7% ($\pm 3\sigma$) of the received signals will be in the range $\left[\max_{s \in C}(Re(s) + 3\sigma) \min_{s \in C}(Re(s) - 3\sigma)\right] + i\left[\max_{s \in C}(Im(s) + 3\sigma) \min_{s \in C}(Im(s) - 3\sigma)\right]$. Generate uniformly distributed I/Q symbols over this space and use `qamdemod` (Communications Toolbox) function to calculate exact LLR and approximate LLR values.

```
M = 16;             % Modulation order
k = log2(M);        % Bits per symbols
SNRValues = -5:5:5; % in dB
numSymbols = 1e4;
numSNRValues = length(SNRValues);
symOrder = llrnetQAMSymbolMapping(M);

const = qammod(0:15,M,symOrder,'UnitAveragePower',1);
maxConstReal = max(real(const));
maxConstImag = max(imag(const));

numBits = numSymbols*k;
exactLLR = zeros(numBits,numSNRValues);
approxLLR = zeros(numBits,numSNRValues);
rxSym = zeros(numSymbols,numSNRValues);
for snrIdx = 1:numSNRValues
    SNR = SNRValues(snrIdx);
    noiseVariance = 10^(-SNR/10);
    sigma = sqrt(noiseVariance);

    maxReal = maxConstReal + 3*sigma;
    minReal = -maxReal;
    maxImag = maxConstImag + 3*sigma;
    minImag = -maxImag;

    r = (rand(numSymbols,1)*(maxReal-minReal)+minReal) + ...
        1i*(rand(numSymbols,1)*(maxImag-minImag)+minImag);
    rxSym(:,snrIdx) = r;

    exactLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','llr','NoiseVariance',noiseVariance);
    approxLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','approxllr','NoiseVariance',noiseVariance);
end
```

Set up and Train Neural Network

Set up a shallow neural network with one input layer, one hidden layer, and one output layer. Input a received symbol to the network and train it to estimate the exact LLR values. Since the network expects real inputs, create a two column vector, where the first column is the real values of the received symbol and the second column is the imaginary values of the received symbol. Also, the output must be a $k \times N$ vector, where k is the number of bits per symbol and N is the number of symbols.

```
nnInput = zeros(numSymbols,2,numSNRValues);
nnOutput = zeros(numSymbols,k,numSNRValues);
```

```
for snrIdx = 1:numSNRValues
    rxTemp = rxSym(:,snrIdx);
    rxTemp = [real(rxTemp) imag(rxTemp)];
    nnInput(:,:,snrIdx) = rxTemp;

    llrTemp = exactLLR(:,snrIdx);
    nnOutput(:,:,snrIdx) = reshape(llrTemp, k, numSymbols)';
end
```

For 16-QAM symbols, the hidden layer has 8 neurons and the output layer has 4 neurons, which corresponds to the number of bits per symbol. The llrnetNeuralNetwork function returns preconfigured neural network. Train the neural network for three different SNR values. Use the exact LLR values calculated using the `qamdemod` function as the expected output values.

```
hiddenLayerSize = 8;
trainedNetworks = cell(1,numSNRValues);
for snrIdx=1:numSNRValues
    fprintf('Training neural network for SNR = %1.1fdB\n', ...
        SNRValues(snrIdx))
    x = nnInput(:,:,snrIdx)';
    y = nnOutput(:,:,snrIdx)';

    MSExactLLR = mean(y(:).^2);
    fprintf('\tMean Square LLR = %1.2f\n', MSExactLLR)

    % Train the Network. Use parallel pool, if available. Train three times
    % and pick the best one.
    mse = inf;
    for p=1:3
        netTemp = llrnetNeuralNetwork(hiddenLayerSize);
        if parallelComputingLicenseExists()
            [netTemp,tr] = train(netTemp,x,y,'useParallel','yes');
        else
            [netTemp,tr] = train(netTemp,x,y);
        end
        % Test the Network
        predictedLLRSNR = netTemp(x);
        mseTemp = perform(netTemp,y,predictedLLRSNR);
        fprintf('\t\tTrial %d: MSE = %1.2e\n', p, mseTemp)
        if mse > mseTemp
            mse = mseTemp;
            net = netTemp;
        end
    end

    % Store the trained network
    trainedNetworks{snrIdx} = net;
    fprintf('\tBest MSE = %1.2e\n', mse)
end
```

```
Training neural network for SNR = -5.0dB

    Mean Square LLR = 4.42

        Trial 1: MSE = 1.95e-06
        Trial 2: MSE = 1.22e-04
        Trial 3: MSE = 4.54e-06

    Best MSE = 1.95e-06
```

```
Training neural network for SNR = 0.0dB

    Mean Square LLR = 15.63

        Trial 1: MSE = 1.90e-03
        Trial 2: MSE = 5.03e-03
        Trial 3: MSE = 8.95e-05

    Best MSE = 8.95e-05

Training neural network for SNR = 5.0dB

    Mean Square LLR = 59.29

        Trial 1: MSE = 2.25e-02
        Trial 2: MSE = 2.23e-02
        Trial 3: MSE = 7.40e-02

    Best MSE = 2.23e-02
```

Performance metric for this network is mean square error (MSE). The final MSE values show that the neural network converges to an MSE value that is at least 40 dB less than the mean square exact LLR values. Note that, as SNR increases so do the LLR values, which results in relatively higher MSE values.

Results for 16-QAM

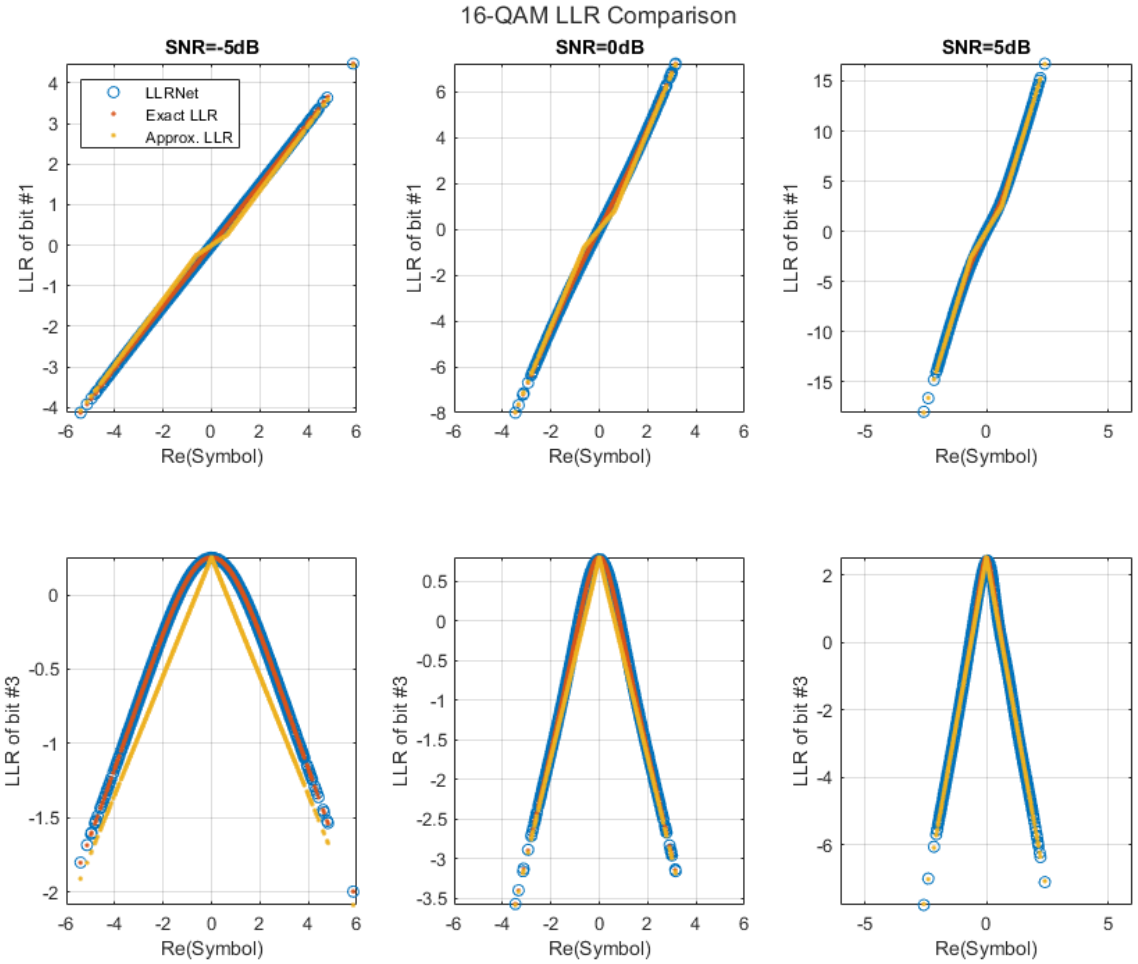
Compare the LLR estimates of LLRNet to that of exact LLR and approximate LLR. Simulate 1e4 16-QAM symbols and calculate LLR values using all three methods. Do not use the symbols that we generated in the previous section so as not to give LLRNet an unfair advantage, since those symbols were used to train the LLRNet.

```
numBits = numSymbols*k;
d = randi([0 1], numBits, 1);

txSym = qammod(d,M,symOrder,'InputType','bit','UnitAveragePower',1);

exactLLR = zeros(numBits,numSNRValues);
approxLLR = zeros(numBits,numSNRValues);
predictedLLR = zeros(numBits,numSNRValues);
rxSym = zeros(length(txSym),numSNRValues);
for snrIdx = 1:numSNRValues
    SNR = SNRValues(snrIdx);
    sigmas = 10^(-SNR/10);
    r = awgn(txSym,SNR);
    rxSym(:,snrIdx) = r;

    exactLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','llr','NoiseVariance',sigmas);
    approxLLR(:,snrIdx) = qamdemod(r,M,symOrder,...
        'UnitAveragePower',1,'OutputType','approxllr','NoiseVariance',sigmas);

    net = trainedNetworks{snrIdx};
    x = [real(r) imag(r)]';
    tempLLR = net(x);
    predictedLLR(:,snrIdx) = reshape(tempLLR, numBits, 1);
end

qam16Results.exactLLR = exactLLR;
```

```
qam16Results.approxLLR = approxLLR;
qam16Results.predictedLLR = predictedLLR;
qam16Results.RxSymbols = rxSym;
qam16Results.M = M;
qam16Results.SNRValues = SNRValues;
qam16Results.HiddenLayerSize = hiddenLayerSize;
qam16Results.NumSymbols = numSymbols;
```

The following figure shows exact LLR, max-log approximate LLR, and LLRNet estimate of LLR values versus the real part of the received symbol for odd bits. LLRNet matches the exact LLR values even for low SNR values.

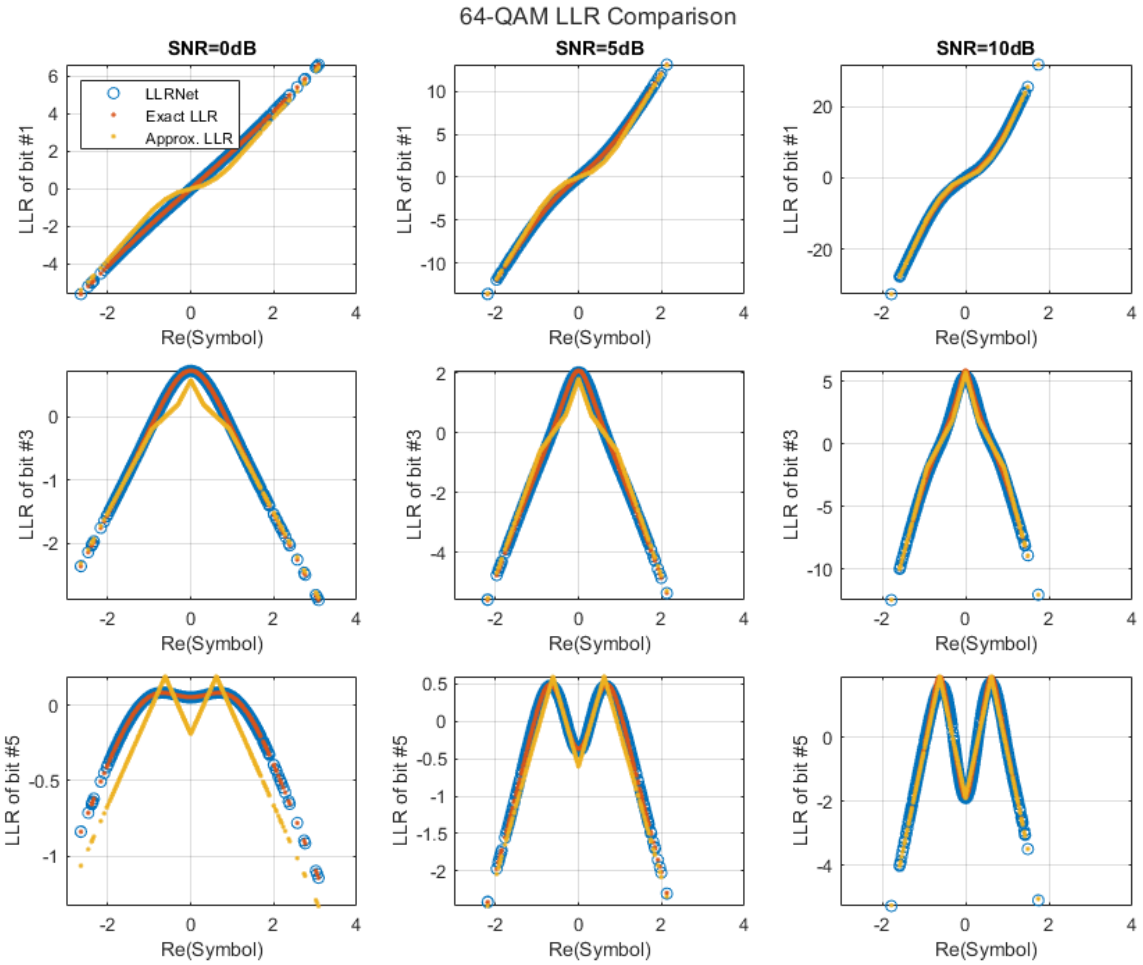
```
llrnetPlotLLR(qam16Results,'16-QAM LLR Comparison')
```

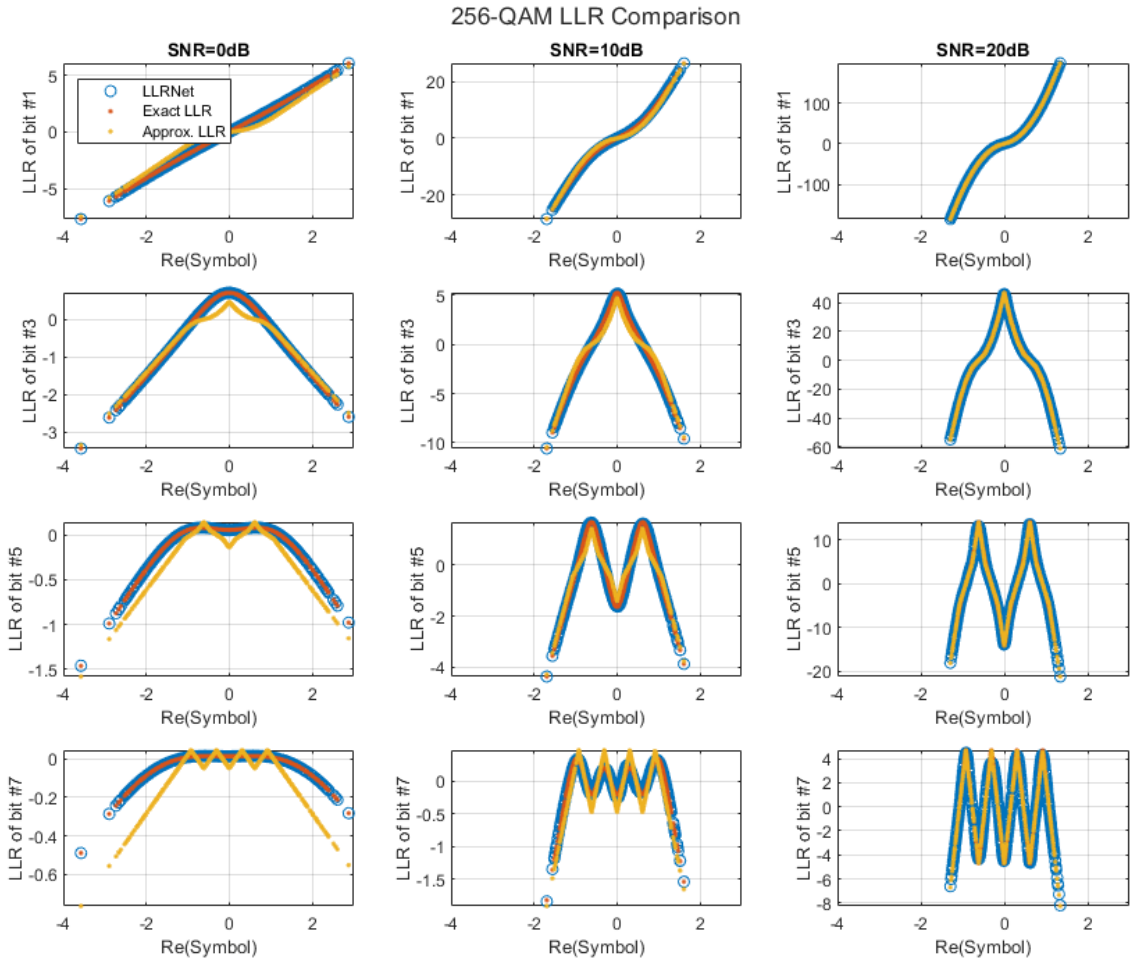


64-QAM and 256-QAM LLR Estimation Performance

Check if the LLRNet can estimate the LLR values for higher order QAM. Repeat the same process you followed for 16-QAM for 64-QAM and 256-QAM using the llrnetQAMLLR helper function. The following figures show exact LLR, max-log approximate LLR, and LLRNet estimate of LLR values versus the real part of the received symbol for odd bits.

```
trainNow = Show saved results;
if trainNow
    % Parameters for 64-QAM
    simParams(1).M = 64; %#ok<UNRCH>
    simParams(1).SNRValues = 0:5:10;
    simParams(1).HiddenLayerSize = 16;
    simParams(1).NumSymbols = 1e4;
    simParams(1).UseReLU = false;

    % Parameters for 256-QAM
    simParams(2).M = 256;
    simParams(2).SNRValues = 0:10:20;
    simParams(2).HiddenLayerSize = 32;
    simParams(2).NumSymbols = 1e4;
    simParams(2).UseReLU = false;

    simResults = llrnetQAMLLR(simParams);
    llrnetPlotLLR(simResults(1),sprintf('%d-QAM LLR Comparison',simResults(1).M))
    llrnetPlotLLR(simResults(2),sprintf('%d-QAM LLR Comparison',simResults(2).M))
else
    load('llrnetQAMPerformanceComparison.mat', 'simResults')
    for p=1:length(simResults)
        llrnetPlotLLR(simResults(p),sprintf('%d-QAM LLR Comparison',simResults(p).M))
    end
end
```


64-QAM LLR Comparison
SNR=0dB
SNR=5dB
SNR=10dB
LLRNet
Exact LLR
Approx. LLR
LLR of bit #1
LLR of bit #3
LLR of bit #5
Re(Symbol)



DVB-S.2 Packet Error Rate

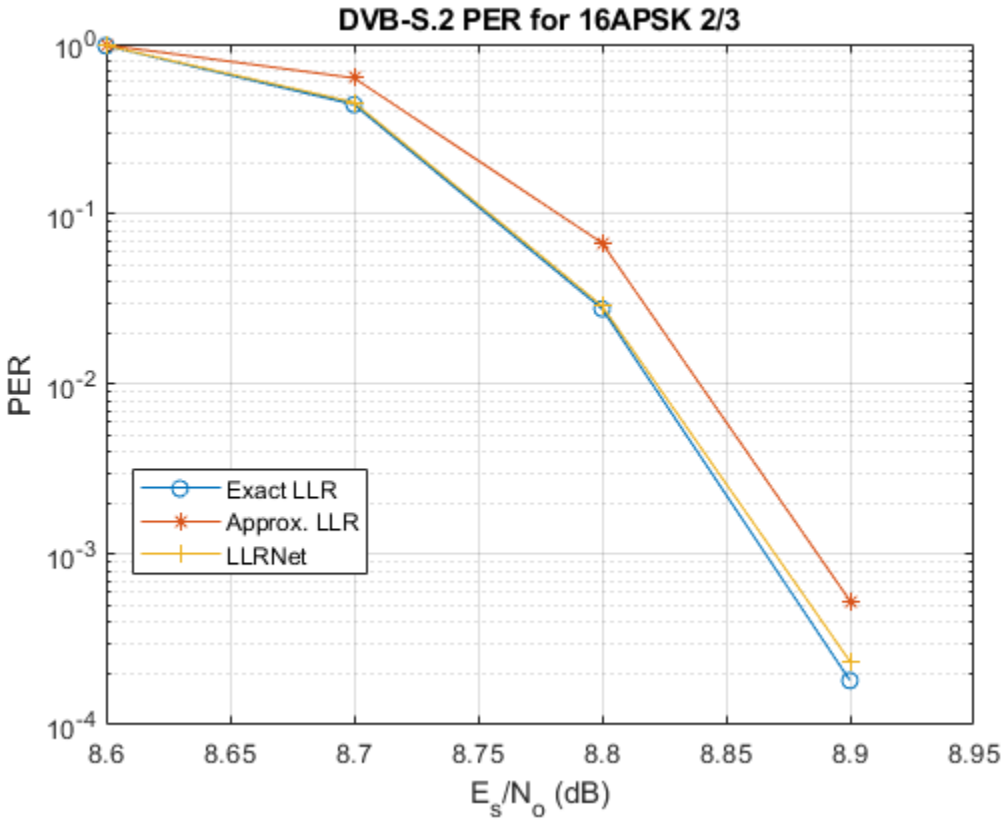
DVB-S.2 system uses a soft demodulator to generate inputs for the LDPC decoder. Simulate the packet error rate (PER) of a DVB-S.2 system with 16-APSK modulation and 2/3 LDPC code using exact LLR, approximate LLR, and LLRNet using llrNetDVBS2PER function. This function uses the `comm.PSKDemodulator` (Communications Toolbox) System object and the `dvbsapskdemod` (Communications Toolbox) function to calculate exact and approximate LLR values and the `comm.AWGNChannel` (Communications Toolbox) System object to simulate the channel.

Set `simulateNow` to true (or select "Simulate" in the dropdown) to run the PER simulations for the values of `subsystemType`, `EsNoValues`, and `numSymbols` using the llrnetDVBS2PER function. If Parallel Computing Toolbox™ is installed, this function uses the `parfor` command to run the simulations in parallel. On an Intel® Xeon® W-2133 CPU @ 3.6GHz and running a “Run Code on Parallel Pools” (Parallel Computing Toolbox) of size 6, the simulation takes about 40 minutes. Set `simulateNow` to `false` (or select "Plot saved results" in the dropdown), to load the PER results for the values of `subsystemType='16APSK 2/3'`, `EsNoValues=8.6:0.1:8.9`, and `numSymbols=10000`.

Set `trainNow` to `true` (or select "Train LLRNet" in the dropdown) to train LLR neural networks for each value of `EsNoValues`, for the given `subsystemType` and `numSymbols`. If Parallel Computing Toolbox™ is installed, the `train` function can be called with the optional name-value pair `'useParallel'` set to `'yes'` to run the simulations in parallel. On an Intel® Xeon® W-2133 CPU @ 3.6GHz and running a “Run Code on Parallel Pools” (Parallel Computing Toolbox) of size 6, the simulation takes about 21 minutes. Set `trainNow` to false (or select "Use saved networks" in the dropdown) to load LLR neural networks trained for `subsystemType='16APSK 2/3'`, `EsNoValues=8.6:0.1:8.9`.

For more information on the DVB-S.2 PER simulation, see the “DVB-S.2 Link, Including LDPC Coding in Simulink” (Communications Toolbox) example. For more information on training the network, refer to the llrnetTrainDVBS2LLRNetwork function and [1] on page 13-0 .

```
simulateNow = Plot saved results;
if simulateNow
    subsystemType = '16APSK 2/3'; %#ok<UNRCH>
    EsNoValues = 8.6:0.1:8.9;     % in dB
    numFrames = 10000;
    numErrors = 200;

    trainNow = Use saved networks;
    if trainNow && (~strcmp(subsystemType,'16APSK 2/3') || ~isequal(EsNoValues,8.6:0.1:9))
        % Train the networks for each EsNo value
        numTrainSymbols = 1e4;
        hiddenLayerSize = 64;
        llrNets = llrnetTrainDVBS2LLRNetwork(subsystemType, EsNoValues, numTrainSymbols, hiddenLa
    else
        load('llrnetDVBS2Networks','llrNets','subsystemType','EsNoValues');
    end

    % Simulate PER with exact LLR, approximate LLR, and LLRNet
    [perLLR,perApproxLLR,perLLRNet] = llrnetDVBS2PER(subsystemType,EsNoValues,llrNets,numFrames,n
    llrnetPlotLLRvsEsNo(perLLR,perApproxLLR,perLLRNet,EsNoValues,subsystemType)
else
    load('llrnetDVBS2PERResults.mat','perApproxLLR','perLLR','perLLRNet',...
        'subsystemType','EsNoValues');
    llrnetPlotLLRvsEsNo(perLLR,perApproxLLR,perLLRNet,EsNoValues,subsystemType)
end
```



The results show that the LLRNet almost matches the performance of exact LLR without using any expensive operations such as logarithm and exponential.

Further Exploration

Try different modulation and coding schemes for the DVB-S.2 system. The full list of modulation types and coding rates are given in the “DVB-S.2 Link, Including LDPC Coding in Simulink” (Communications Toolbox) example. You can also try different sizes for the hidden layer of the network to reduce the number of operations and measure the performance loss as compared to exact LLR.

The example uses these helper functions. Examine these files to learn about details of the implementation.

- llrnetDVBS2PER.m: Simulate DVB-S.2 PER using exact LLR, approximate LLR, and LLRNet LLR
- llrnetTrainDVBS2LLRNetwork.m: Train neural networks for DVB-S.2 LLR estimation
- llrnetQAMLLR.m: Train neural networks for M-ary QAM LLR estimation and calculate exact LLR, approximate LLR, and LLRNet LLR
- llrnetNeuralNetwork.m: Configure a shallow neural network for LLR estimation

References

[1] O. Shental and J. Hoydis, ""Machine LLRning": Learning to Softly Demodulate," 2019 IEEE Globecom Workshops (GC Wkshps), Waikoloa, HI, USA, 2019, pp. 1-7.

See Also

More About

- “Deep Learning in MATLAB” on page 1-2

Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation

This example shows how to design a radio frequency (RF) fingerprinting convolutional neural network (CNN) with simulated data. You train the CNN with simulated wireless local area network (WLAN) beacon frames from known and unknown routers for RF fingerprinting. You then compare the media access control (MAC) address of received signals and the RF fingerprint detected by the CNN to detect WLAN router impersonators.

For more information on how to test the designed neural network with signals captured from real Wi-Fi routers, see the “Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation” (Communications Toolbox) example.

Detect Router Impersonation Using RF Fingerprinting

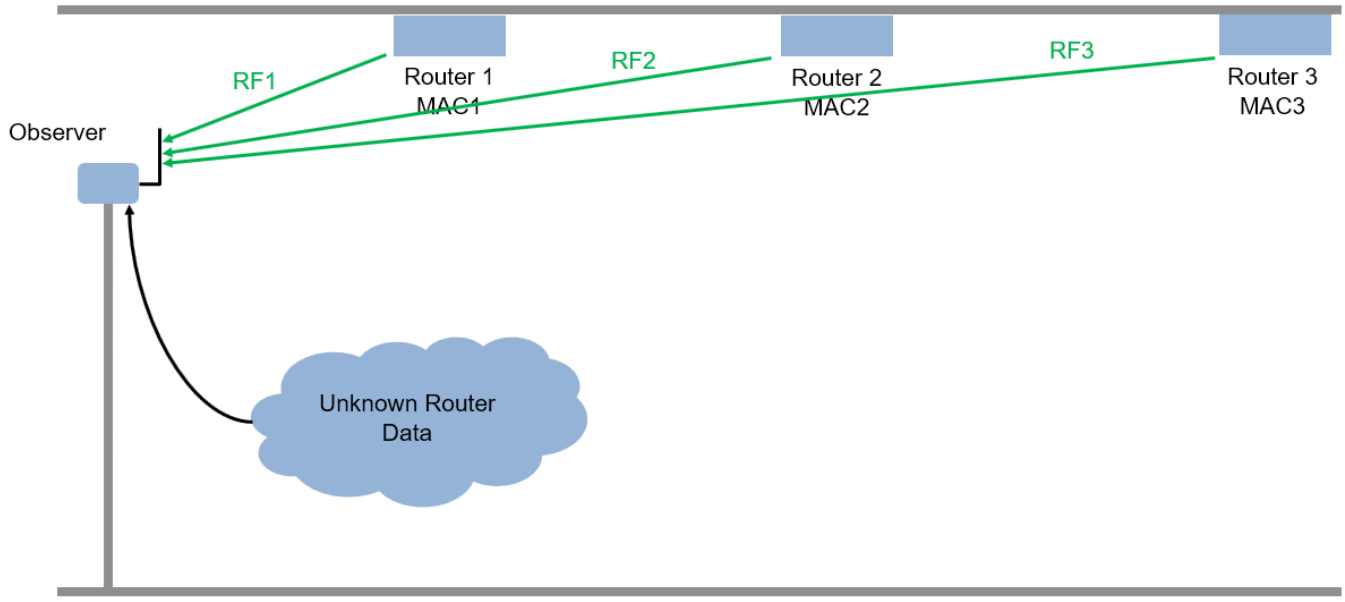
Router impersonation is a form of attack on a WLAN network where a malicious agent tries to impersonate a legitimate router and trick network users to connect to it. Security identification solutions based on simple digital identifiers, such as MAC addresses, IP addresses, and SSID, are not effective in detecting such an attack. These identifiers can be easily spoofed. Therefore, a more secure solution uses other information, such as the RF signature of the radio link, in addition to these simple digital identifiers.

A wireless transmitter-receiver pair creates a unique RF signature at the receiver that is a combination of the channel and RF impairments. *RF Fingerprinting* is the process of distinguishing transmitting radios in a shared spectrum through these signatures. In [1] on page 13-0 , authors designed a deep learning (DL) network that consumes raw baseband in-phase/quadrature (IQ) samples and identifies the transmitting radio. The network can identify the transmitting radios if the RF impairments are dominant or the channel profile stays constant during the operation time. Most WLAN networks have fixed routers that create a static channel profile when the receiver location is also fixed. In such a scenario, the deep learning network can identify router impersonators by comparing the received signal's RF fingerprint and MAC address pair to that of the known routers.

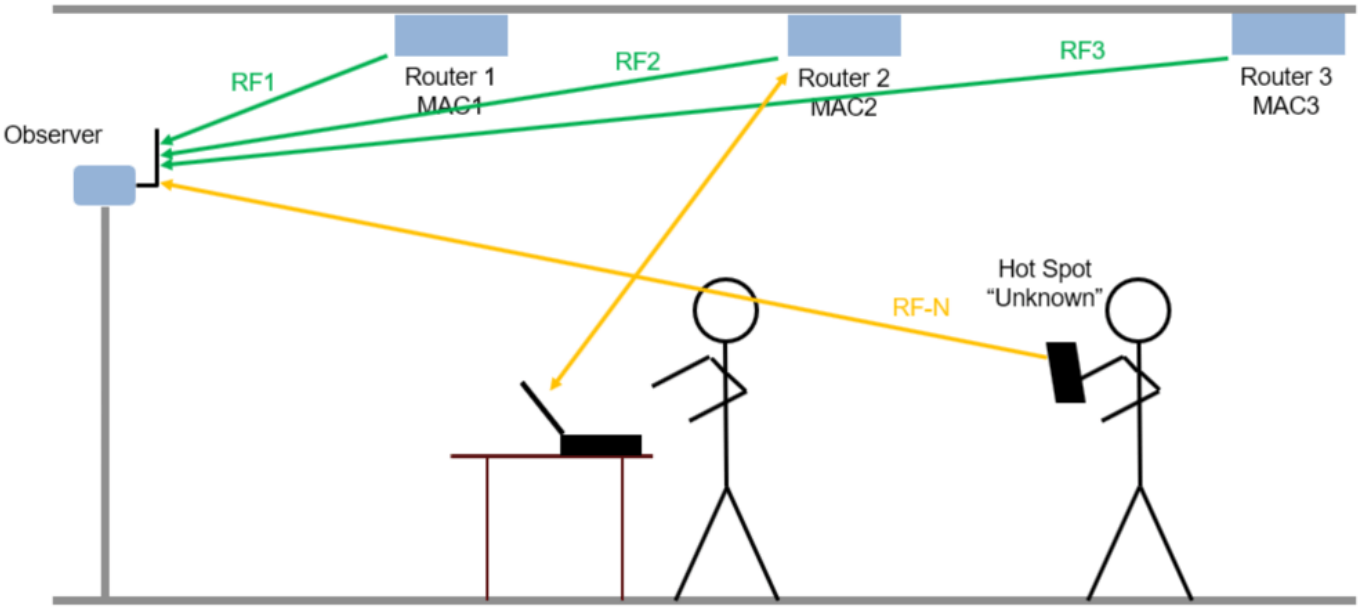
This example simulates a WLAN system with several fixed routers and a fixed observer using the WLAN Toolbox™ and trains a neural network (NN) with the simulated data using Deep Learning Toolbox™.

System Description

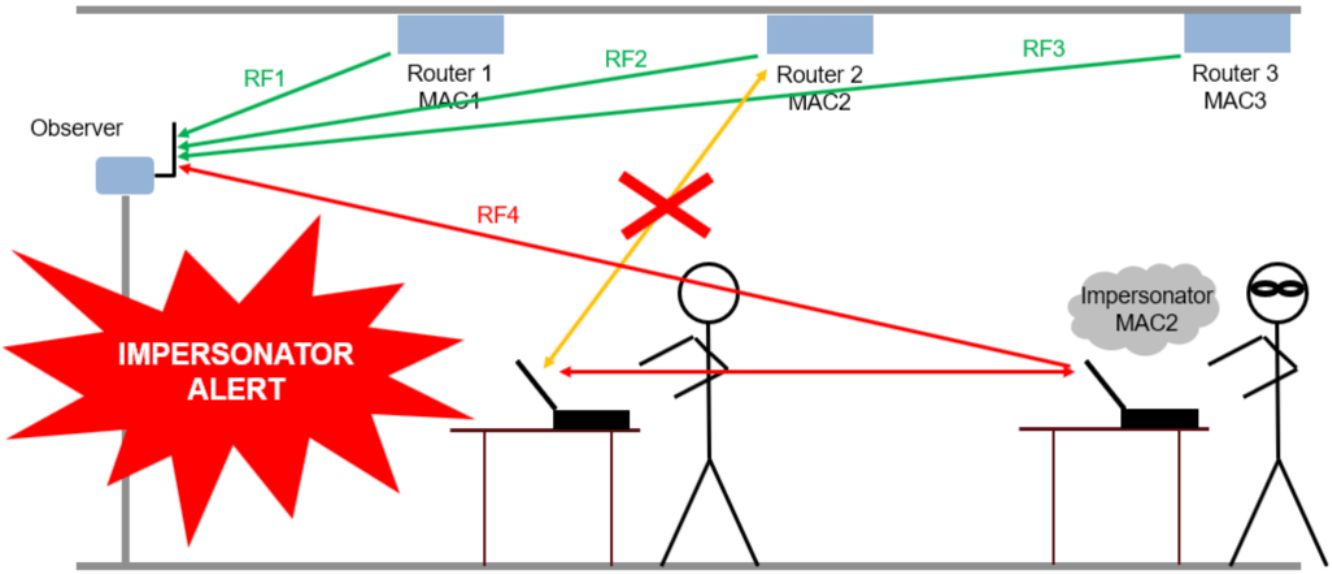
Assume an indoor space with a number of trusted routers with known MAC addresses, which we will refer to as known routers. Also, assume that unknown routers may enter the observation area, some of which may be router impersonators. The class "Unknown" represents any transmitting device that is not contained in the known set. The following figure shows a scenario where there are three known routers. The observer collects non-high throughput (non-HT) beacon signals from these routers and uses the (legacy) long training field (L-LTF) to identify the RF fingerprint. Transmitted L-LTF signals are the same for all routers that enables the algorithm to avoid any data dependency. Since the routers and the observer are fixed, the RF fingerprints (combination of multipath channel profile and RF impairments) RF1, RF2, and RF3 do not vary in time. Unknown router data is a collection of random RF fingerprints, which are different than the known routers.



The following figure shows a user connected to a router and a mobile hot spot. After training, the observer receives beacon frames and decodes the MAC address. Also, the observer extracts the L-LTF signal and uses this signal to classify the RF fingerprint of the source of the beacon frame. If the MAC address and the RF fingerprint matches, as in the case of Router 1, Router 2, and Router3, then the observer declares the source as a "known" router. If the MAC address of the beacon is not in the database and the RF fingerprint does not match any of the known routers, as in the case of a mobile hot spot, then the observer declares the source as an "unknown" router.



The following figure shows a router impersonator in action. A router impersonator (a.k.a. evil twin) can replicate the MAC address of a known router and transmit beacon frames. Then, the hacker can jam the original router and force the user to connect to the evil twin. The observer receives the beacon frames from the evil twin too and decodes the MAC address. The decoded MAC address matches the MAC address of a known router but the RF fingerprint does not match. The observer declares the source as a router impersonator.



Set System Parameters

Generate a dataset of 5,000 Non-HT WLAN beacon frames for each router. Use MAC addresses as labels for the known routers; the remaining are be labeled as "Unknown". A NN is trained to classify the known routers as well as to detect any unknown ones. Split the dataset into training, validation and test, where the splitting ratios are 80%, 10%, and 10%, respectively. Consider an SNR of 20 dB, working on the 5 GHz band. The number of simulated devices is set to 4 but it can be modified by choosing a different value for `numKnownRouters`. Set the number of unknown routers more than the known ones to represent in the dataset the variability in the unknown router RF fingerprints.

```
numKnownRouters = 4;
numUnknownRouters = 10;
numTotalRouters = numKnownRouters+numUnknownRouters;
SNR = 20;                   % dB
channelNumber = 153;        % WLAN channel number
channelBand = 5;            % GHz
frameLength = 160;          % L-LTF sequence length in samples
```

By default, this example downloads training data and trained network from https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingSimulatedData.tar.gz. If you do not have an Internet connection, you can download the file manually on a computer that is connected to the Internet and save to the same directory as the current example files.

To run this example quickly, download the pretrained trained network and generate a small number of frames, for example 10. To train the network on your computer, choose the "Train network now" option (i.e. set `trainNow` to true). Generating 5000 frames of data takes about 50 minutes on an Intel(R) Xeon(R) W-2133 CPU @ 3.6 GHz with 64 MB memory. Training this network takes about 5

minutes with an Nvidia(R) Titan Xp GPU. Training on a CPU may result in a very long training duration.

```
trainNow = Use trained network ;

if trainNow
  numTotalFramesPerRouter = 5000; %#ok<UNRCH>
else
  numTotalFramesPerRouter = 10;
  rfFingerprintingDownloadData('simulated')
end

Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingSimulatedData.ta
Download and extracting files done

  numTrainingFramesPerRouter = numTotalFramesPerRouter*0.8;
  numValidationFramesPerRouter = numTotalFramesPerRouter*0.1;
  numTestFramesPerRouter = numTotalFramesPerRouter*0.1;
```

Generate WLAN Waveforms

Wi-Fi routers that implement 802.11a/g/n/ac protocols transmit beacon frames in the 5 GHz band to broadcast their presence and capabilities using the OFDM non-HT format. The beacon frame consists of two main parts: preamble (SYNC) and payload (DATA). The preamble has two parts: short training and long training. In this example, the payload contains the same bits except the MAC address for each router. The CNN uses the L-LTF part of the preamble as training units. Reusing the L-LTF signal for RF fingerprinting provides an overhead-free fingerprinting solution. Use `wlanMACFrameConfig` (WLAN Toolbox), `wlanMACFrame` (WLAN Toolbox), `wlanNonHTConfig` (WLAN Toolbox), and `wlanWaveformGenerator` (WLAN Toolbox) functions to generate WLAN beacon frames.

```
% Create Beacon frame-body configuration object
frameBodyConfig = wlanMACManagementConfig;

% Create Beacon frame configuration object
beaconFrameConfig = wlanMACFrameConfig('FrameType', 'Beacon', ...
  "ManagementConfig", frameBodyConfig);

% Generate Beacon frame bits
[~, mpduLength] = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

% Create a wlanNONHTConfig object, 20 MHz bandwidth and MCS 1 are used
nonHTConfig = wlanNonHTConfig(...
  'ChannelBandwidth', "CBW20",...
  "MCS", 1,...
  "PSDULength", mpduLength);
```

The rfFingerprintingNonHTFrontEnd object performs front-end processing including extracting the L-LTF signal. The object is configured with a channel bandwidth of 20 MHz to process non-HT signals.

```
rxFrontEnd = rfFingerprintingNonHTFrontEnd('ChannelBandwidth', 'CBW20');

fc = helperWLANChannelFrequency(channelNumber, channelBand);
fs = wlanSampleRate(nonHTConfig);
```

Setup Channel and RF Impairments

Pass each frame through a channel with

- Rayleigh multipath fading
- Radio impairments, such as phase noise, frequency offset and DC offset
- AWGN

Rayleigh Multipath and AWGN

The channel passes the signals through a Rayleigh multipath fading channel using the `comm.RayleighChannel` (Communications Toolbox) System object. Assume a delay profile of [0 1.8 3.4] samples with corresponding average path gains of [0 -2 -10] dB. Since the channel is static, set maximum Doppler shift to zero to make sure that the channel does not change for the same radio. Implement the multipath channel with these settings. Add noise using the `awgn` (Communications Toolbox) function,

```
multipathChannel = comm.RayleighChannel(...
  'SampleRate', fs, ...
  'PathDelays', [0 1.8 3.4]/fs, ...
  'AveragePathGains', [0 -2 -10], ...
  'MaximumDopplerShift', 0);
```

Radio Impairments

The RF impairments, and their corresponding range of values are:

- Phase noise [0.01, 0.3] rms (degrees)
- Frequency offset [-4, 4] ppm
- DC offset: [-50, -32] dBc

See helperRFImpairments on page 13-0 function for more details on RF impairment simulation. This function uses `comm.PhaseFrequencyOffset` (Communications Toolbox) and `comm.PhaseNoise` (Communications Toolbox) System objects.

```
phaseNoiseRange = [0.01, 0.3];
freqOffsetRange = [-4, 4];
dcOffsetRange = [-50, -32];

rng(123456)  % Fix random generator

% Assign random impairments to each simulated radio within the previously
% defined ranges
radioImpairments = repmat(...
  struct('PhaseNoise', 0, 'DCOffset', 0, 'FrequencyOffset', 0), ...
  numTotalRouters, 1);
for routerIdx = 1:numTotalRouters
  radioImpairments(routerIdx).PhaseNoise = ...
    rand*(phaseNoiseRange(2)-phaseNoiseRange(1)) + phaseNoiseRange(1);
  radioImpairments(routerIdx).DCOffset = ...
    rand*(dcOffsetRange(2)-dcOffsetRange(1)) + dcOffsetRange(1);
  radioImpairments(routerIdx).FrequencyOffset = ...
    fc/1e6*(rand*(freqOffsetRange(2)-freqOffsetRange(1)) + freqOffsetRange(1));
end
```

Apply Channel Impairments and Generate Data Frames for Training

Apply the RF and channel impairments defined previously. Reset the channel object for each radio to generate an independent channel. Use rfFingerprintingNonHTFrontEnd function to process the received frames. Finally, extract the L-LTF from every transmitted WLAN frame. Split the received L-LTF signals into training, validation and test sets.

```
% Create variables that will store the training, validation and testing
% datasets
xTrainingFrames = zeros(frameLength, numTrainingFramesPerRouter*numTotalRouters);
xValFrames = zeros(frameLength, numValidationFramesPerRouter*numTotalRouters);
xTestFrames = zeros(frameLength, numTestFramesPerRouter*numTotalRouters);

% Index vectors for train, validation and test data units
trainingIndices = 1:numTrainingFramesPerRouter;
validationIndices = 1:numValidationFramesPerRouter;
testIndices = 1:numTestFramesPerRouter;

tic
generatedMACAddresses = strings(numTotalRouters, 1);
rxLLTF = zeros(frameLength, numTotalFramesPerRouter);     % Received L-LTF sequences
for routerIdx = 1:numTotalRouters

  % Generate a 12-digit random hexadecimal number as a MAC address for
  % known routers. Set the MAC address of all unknown routers to
  % 'AAAAAAAAAAAA'.
  if (routerIdx<=numKnownRouters)
    generatedMACAddresses(routerIdx) = string(dec2hex(bi2de(randi([0 1], 12, 4)))');
  else
    generatedMACAddresses(routerIdx) = 'AAAAAAAAAAAA';
  end

  fprintf('%s - Generating frames for router %d with MAC address %s\n', ...
    datestr(toc/86400,'HH:MM:SS'), routerIdx, generatedMACAddresses(routerIdx))

  % Set MAC address into the wlanFrameConfig object
  beaconFrameConfig.Address2 = generatedMACAddresses(routerIdx);

  % Generate beacon frame bits
  beacon = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

  txWaveform = wlanWaveformGenerator(beacon, nonHTConfig);

  txWaveform = helperNormalizeFramePower(txWaveform);

  % Add zeros to account for channel delays
  txWaveform = [txWaveform; zeros(160,1)]; %#ok<AGROW>

  % Reset multipathChannel object to generate a new static channel
  reset(multipathChannel)

  frameCount= 0;
  while frameCount<numTotalFramesPerRouter

    rxMultipath = multipathChannel(txWaveform);

    rxImpairment = helperRFImpairments(rxMultipath, radioImpairments(routerIdx), fs);
```

```
    rxSig = awgn(rxImpairment,SNR,0);

    % Detect the WLAN packet and return the received L-LTF signal using
    % rfFingerprintingNonHTFrontEnd object
    [valid, ~, ~, ~, ~, LLTF] = rxFrontEnd(rxSig);

    % Save successfully received L-LTF signals
    if valid
      frameCount=frameCount+1;
      rxLLTF(:,frameCount) = LLTF;
    end

    if mod(frameCount,500) == 0
      fprintf('%s - Generated %d/%d frames\n', ...
        datestr(toc/86400,'HH:MM:SS'), frameCount, numTotalFramesPerRouter)
    end
  end

  rxLLTF = rxLLTF(:, randperm(numTotalFramesPerRouter));

  % Split data into training, validation and test
  xTrainingFrames(:, trainingIndices+(routerIdx-1)*numTrainingFramesPerRouter) ...
    = rxLLTF(:, trainingIndices);
  xValFrames(:, validationIndices+(routerIdx-1)*numValidationFramesPerRouter)...
    = rxLLTF(:, validationIndices+ numTrainingFramesPerRouter);
  xTestFrames(:, testIndices+(routerIdx-1)*numTestFramesPerRouter)...
    = rxLLTF(:, testIndices + numTrainingFramesPerRouter+numValidationFramesPerRouter);
end
```

```
00:00:00 - Generating frames for router 1 with MAC address 71153FFD7ACA
00:00:01 - Generating frames for router 2 with MAC address 5F4A8EAD6AD2
00:00:01 - Generating frames for router 3 with MAC address A91A85793DAA
00:00:01 - Generating frames for router 4 with MAC address 841F1BE784B0
00:00:02 - Generating frames for router 5 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 6 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 7 with MAC address AAAAAAAAAAAA
00:00:02 - Generating frames for router 8 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 9 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 10 with MAC address AAAAAAAAAAAA
00:00:03 - Generating frames for router 11 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 12 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 13 with MAC address AAAAAAAAAAAA
00:00:04 - Generating frames for router 14 with MAC address AAAAAAAAAAAA
```

```
% Label received frames. Label the first numKnownRouters with their MAC
% address. Label the rest with "Unknown”.
labels = generatedMACAddresses;
labels(generatedMACAddresses == generatedMACAddresses(numTotalRouters)) = "Unknown";

yTrain = repelem(labels, numTrainingFramesPerRouter);
yVal = repelem(labels, numValidationFramesPerRouter);
yTest = repelem(labels, numTestFramesPerRouter);
```

Create Real-Valued Input Matrices

The Deep Learning model only works on real numbers. Thus, I and Q are split into two separate columns. Then, the data is rearranged into a 2 x frameLength x 1 x numFrames matrix, as required

by the Deep Learning Toolbox. Additionally, the training set is shuffled, and the label variables are saved as categorical variables.

```
% Rearrange datasets into a one-column vector
xTrainingFrames = xTrainingFrames(:);
xValFrames = xValFrames(:);
xTestFrames = xTestFrames(:);

% Separate between I and Q
xTrainingFrames = [real(xTrainingFrames), imag(xTrainingFrames)];
xValFrames = [real(xValFrames), imag(xValFrames)];
xTestFrames = [real(xTestFrames), imag(xTestFrames)];

% Reshape training data into a 2 x frameLength x 1 x
% numTrainingFramesPerRouter*numTotalRouters matrix
xTrainingFrames = permute(...
  reshape(xTrainingFrames,[frameLength,numTrainingFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]);

% Shuffle data
vr = randperm(numTotalRouters*numTrainingFramesPerRouter);
xTrainingFrames = xTrainingFrames(:,:,:,vr);

% Create label vector and shuffle
yTrain = categorical(yTrain(vr));

% Reshape validation data into a 2 x frameLength x 1 x
% numValidationFramesPerRouter*numTotalRouters matrix
xValFrames = permute(...
  reshape(xValFrames,[frameLength,numValidationFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]);

% Create label vector
yVal = categorical(yVal);

% Reshape test dataset into a numTestFramesPerRouter*numTotalRouter matrix
xTestFrames = permute(...
  reshape(xTestFrames,[frameLength,numTestFramesPerRouter*numTotalRouters, 2, 1]),...
  [1 3 4 2]); %#ok<NASGU>

% Create label vector
yTest = categorical(yTest); %#ok<NASGU>
```

Train the Neural Network

This example uses a neural network (NN) architecture that consists of two convolutional and three fully connected layers. The intuition behind this design is that the first layer will learn features independently in I and Q. Note that the filter sizes are 1x7. Then, the next layer will use a filter size of 2x7 that will extract features combining I and Q together. Finally, the last three fully connected layers will behave as a classifier using the extracted features in the previous layers [1] on page 13-0 .

```
poolSize = [2 1];
strideSize = [2 1];
layers = [
  imageInputLayer([frameLength 2 1], 'Normalization', 'none', 'Name', 'Input Layer')

  convolution2dLayer([7 1], 50, 'Padding', [1 0], 'Name', 'CNN1')
  batchNormalizationLayer('Name', 'BN1')
```

```
  leakyReluLayer('Name', 'LeakyReLu1')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool1')

  convolution2dLayer([7 2], 50, 'Padding', [1 0], 'Name', 'CNN2')
  batchNormalizationLayer('Name', 'BN2')
  leakyReluLayer('Name', 'LeakyReLu2')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool2')

  fullyConnectedLayer(256, 'Name', 'FC1')
  leakyReluLayer('Name', 'LeakyReLu3')
  dropoutLayer(0.5, 'Name', 'DropOut1')

  fullyConnectedLayer(80, 'Name', 'FC2')
  leakyReluLayer('Name', 'LeakyReLu4')
  dropoutLayer(0.5, 'Name', 'DropOut2')

  fullyConnectedLayer(numKnownRouters+1, 'Name', 'FC3')
  softmaxLayer('Name', 'SoftMax')
  classificationLayer('Name', 'Output')
  ]
```

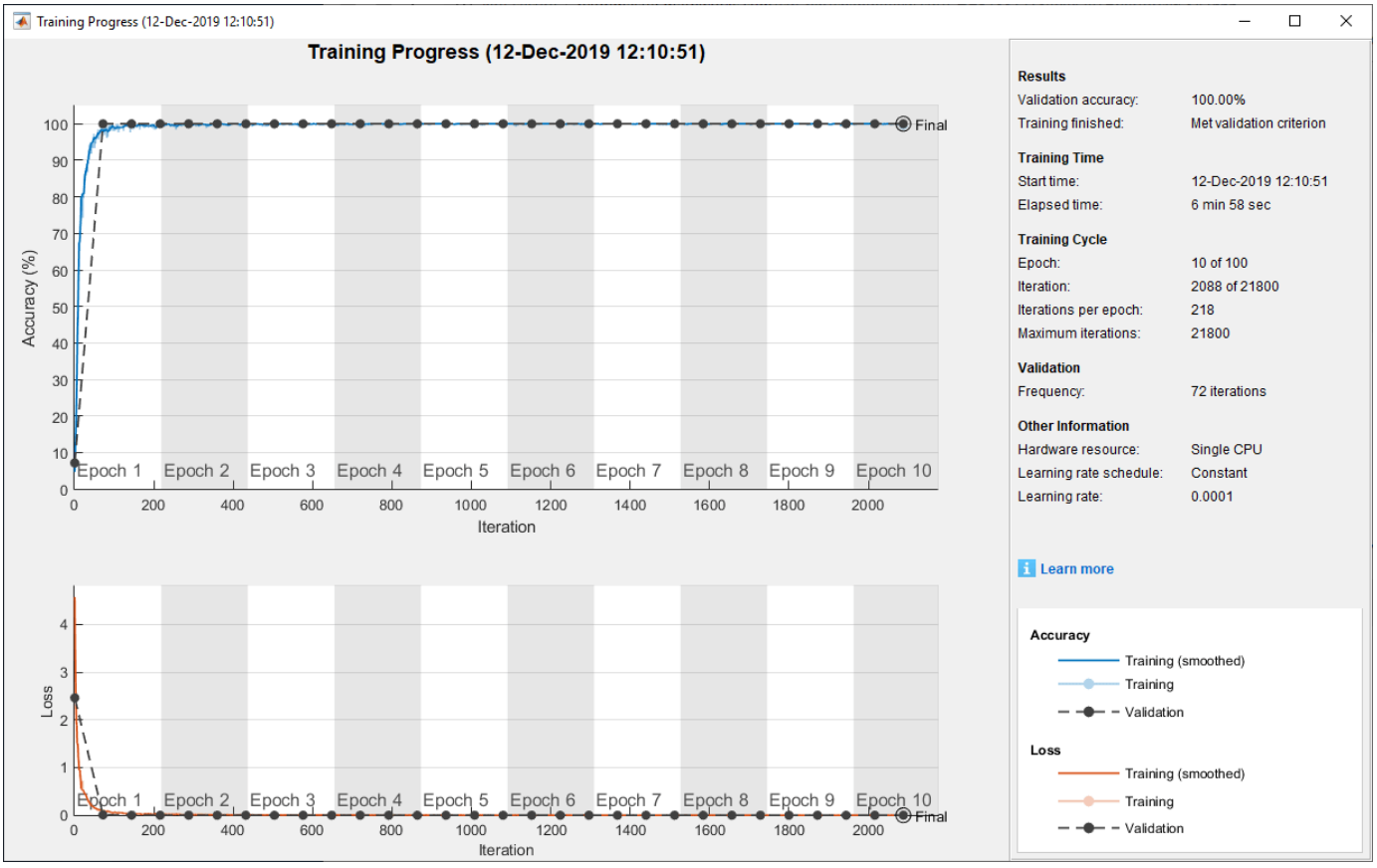
```
layers =
  18×1 Layer array with layers:

     1   'Input Layer'   Image Input             160×2×1 images
     2   'CNN1'          Convolution             50 7×1 convolutions with stride [1  1] and paddi
     3   'BN1'           Batch Normalization     Batch normalization
     4   'LeakyReLu1'    Leaky ReLU              Leaky ReLU with scale 0.01
     5   'MaxPool1'      Max Pooling             2×1 max pooling with stride [2  1] and padding
     6   'CNN2'          Convolution             50 7×2 convolutions with stride [1  1] and paddi
     7   'BN2'           Batch Normalization     Batch normalization
     8   'LeakyReLu2'    Leaky ReLU              Leaky ReLU with scale 0.01
     9   'MaxPool2'      Max Pooling             2×1 max pooling with stride [2  1] and padding
    10   'FC1'           Fully Connected         256 fully connected layer
    11   'LeakyReLu3'    Leaky ReLU              Leaky ReLU with scale 0.01
    12   'DropOut1'      Dropout                 50% dropout
    13   'FC2'           Fully Connected         80 fully connected layer
    14   'LeakyReLu4'    Leaky ReLU              Leaky ReLU with scale 0.01
    15   'DropOut2'      Dropout                 50% dropout
    16   'FC3'           Fully Connected         5 fully connected layer
    17   'SoftMax'       Softmax                 softmax
    18   'Output'        Classification Output   crossentropyex
```

Configure the training options to use the ADAM optimizer with a mini-batch size of 256. By default, '`ExecutionEnvironment`' is set to 'auto', which uses a GPU for training if one is available. Otherwise, `trainNetwork` uses a CPU for training. To explicitly set the execution environment, set '`ExecutionEnvironment`' to one of '`cpu`', '`gpu`', '`multi-gpu`', or '`parallel`'. Choosing '`cpu`' may result in a very long training duration.

```
if trainNow

  miniBatchSize = 256; %#ok<UNRCH>

  % Training options
  options = trainingOptions('adam', ...
    'MaxEpochs',100, ...
    'ValidationData',{xValFrames, yVal}, ...
    'ValidationFrequency',floor(numTrainingFramesPerRouter*numTotalRouters/miniBatchSize/3), ...
```

```
    'Verbose',false, ...
    'L2Regularization', 0.0001, ...
    'InitialLearnRate', 0.0001, ...
    'MiniBatchSize', miniBatchSize, ...
    'ValidationPatience', 3, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch');

  % Train the network
  simNet = trainNetwork(xTrainingFrames, yTrain, layers, options);
else
  % Load trained network (simNet), testing dataset (xTestFrames and
  % yTest) and the used MACAddresses (generatedMACAddresses)
  load('rfFingerprintingSimulatedDataTrainedNN.mat',...
    'generatedMACAddresses',...
    'simNet',...
    'xTestFrames',...
    'yTest')
end
```

As the plot of the training progress shows, the network converges in about 2 epochs to almost 100% accuracy.



Classify test frames and calculate the final accuracy off the neural network.

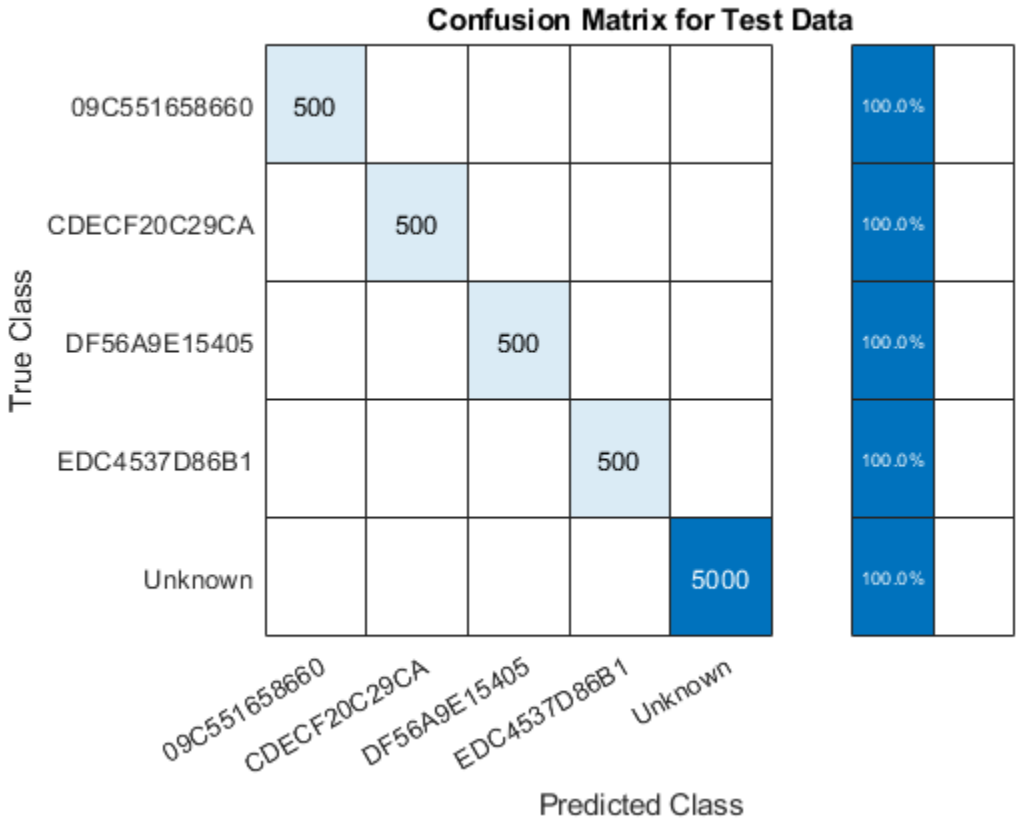
```
% Obtain predicted classes for xTestFrames
yTestPred = classify(simNet,xTestFrames);
```

```
% Calculate test accuracy
testAccuracy = mean(yTest == yTestPred);
disp("Test accuracy: " + testAccuracy*100 + "%")
```

```
Test accuracy: 100%
```

Plot the confusion matrix for the test frames. As mentioned before, perfect classification accuracy is achieved with the synthetic dataset.

```
figure
cm = confusionchart(yTest, yTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
```



Detect Router Impersonator

Generate beacon frames with the known MAC addresses and one unknown MAC address. Generate a new set of RF impairments and multipath channel. Since the impairments are all new, the RF fingerprint for these frames should be classified as "Unknown". The frames with known MAC addresses represent router impersonators while the frames with unknown MAC addresses are simply unknown routers.

```
framesPerRouter = 4;
knownMACAddresses = generatedMACAddresses(1:numKnownRouters);

% Assign random impairments to each simulated radio within the previously
% defined ranges
```

```
for routerIdx = 1:numTotalRouters
  radioImpairments(routerIdx).PhaseNoise = rand*( phaseNoiseRange(2)-phaseNoiseRange(1) ) + phase
  radioImpairments(routerIdx).DCOffset = rand*( dcOffsetRange(2)-dcOffsetRange(1) ) + dcOffsetRan
  radioImpairments(routerIdx).FrequencyOffset = fc/1e6*(rand*( freqOffsetRange(2)-freqOffsetRange
end
% Reset multipathChannel object to generate a new static channel
reset(multipathChannel)

% Run for all known routers and one unknown
for macIndex = 1:(numKnownRouters+1)

  beaconFrameConfig.Address2 = generatedMACAddresses(macIndex);

  % Generate Beacon frame bits
  beacon = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

  txWaveform = wlanWaveformGenerator(beacon, nonHTConfig);

  txWaveform = helperNormalizeFramePower(txWaveform);

  % Add zeros to account for channel delays
  txWaveform = [txWaveform; zeros(160,1)]; %#ok<AGROW>

  % Create an unseen multipath channel. In other words, create an unseen
  % RF fingerprint.
  reset(multipathChannel)

  frameCount= 0;
  while frameCount<framesPerRouter

    rxMultipath = multipathChannel(txWaveform);

    rxImpairment = helperRFImpairments(rxMultipath, radioImpairments(routerIdx), fs);

    rxSig = awgn(rxImpairment,SNR,0);

    % Detect the WLAN packet and return the received L-LTF signal using
    % rfFingerprintingNonHTFrontEnd object
    [payloadFull, cfgNonHT, rxNonHTData, chanEst, noiseVar, LLTF] = ...
      rxFrontEnd(rxSig);

    if payloadFull
      frameCount = frameCount+1;
      recBits = wlanNonHTDataRecover(rxNonHTData, chanEst, ...
        noiseVar, cfgNonHT, 'EqualizationMethod', 'ZF');

      % Decode and evaluate recovered bits
      mpduCfg = wlanMPDUDecode(recBits, cfgNonHT);

      % Separate I and Q and reshape for neural network
      LLTF= [real(LLTF), imag(LLTF)];
      LLTF = permute(reshape(LLTF,frameLength ,[] , 2, 1), [1 3 4 2]);

      ypred = classify(simNet, LLTF);

      if sum(contains(knownMACAddresses, mpduCfg.Address2)) ~= 0
        if categorical(convertCharsToStrings(mpduCfg.Address2))~=ypred
          disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint mismatch, ROUTER
```

```
            else
              disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint match"))
            end
          else
            disp(strcat("MAC Address ", mpduCfg.Address2," is not recognized, unknown device"))
          end
        end

        % Reset multipathChannel object to generate a new static channel
        reset(multipathChannel)
      end
    end
```

```
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 09C551658660 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address CDECF20C29CA is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address DF56A9E15405 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EDC4537D86B1 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
MAC Address AAAAAAAAAAAA is not recognized, unknown device
```

Further Exploration

You can test the system under different channel and RF impairments by modifying the

- Multipath profile (`PathDelays` and `AveragePathGains` properties of Rayleigh channel object),
- Channel noise level (SNR input of `awgn` function),
- RF impairments (`phaseNoiseRange`, `freqOffsetRange`, and `dcOffsetRange` variables).

You can also modify the neural network structure by changing

- Convolutional layer parameters (filter size, number of filters, padding),
- Number of fully connected layers,
- Number of convolutional layers.

Appendix: Helper Functions

```
function [impairedSig] = helperRFImpairments(sig, radioImpairments, fs)
% helperRFImpairments Apply RF impairments
%   IMPAIREDSIG = helperRFImpairments(SIG, RADIOIMPAIRMENTS, FS) returns signal
%   SIG after applying the impairments defined by RADIOIMPAIRMENTS
%   structure at the sample rate FS.
```

```
% Apply frequency offset
fOff = comm.PhaseFrequencyOffset('FrequencyOffset', radioImpairments.FrequencyOffset,  'SampleRa

% Apply phase noise
phaseNoise = helperGetPhaseNoise(radioImpairments);
phNoise = comm.PhaseNoise('Level', phaseNoise, 'FrequencyOffset', abs(radioImpairments.FrequencyO

impFOff = fOff(sig);
impPhNoise = phNoise(impFOff);

% Apply DC offset
impairedSig = impPhNoise + 10^(radioImpairments.DCOffset/10);

end

function [phaseNoise] = helperGetPhaseNoise(radioImpairments)
% helperGetPhaseNoise Get phase noise value
load('Mrms.mat','Mrms','MyI','xI');
[~, iRms] = min(abs(radioImpairments.PhaseNoise - Mrms));
[~, iFreqOffset] = min(abs(xI - abs(radioImpairments.FrequencyOffset)));
phaseNoise = -abs(MyI(iRms, iFreqOffset));
end
```

Selected Bibliography

[1] K. Sankhe, M. Belgiovine, F. Zhou, S. Riyaz, S. Ioannidis and K. Chowdhury, "ORACLE: Optimized Radio clAssification through Convolutional neuraL nEtworks," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 370-378.

See Also

More About

- “Deep Learning in MATLAB” on page 1-2

Test a Deep Neural Network with Captured Data to Detect WLAN Router Impersonation

This example shows how to train a radio frequency (RF) fingerprinting convolutional neural network (CNN) with captured data. You capture wireless local area network (WLAN) beacon frames from real routers using a software defined radio (SDR). You program a second SDR to transmit unknown beacon frames and capture them. You train the CNN using these captured signals. You then program a software-defined radio (SDR) as a router impersonator that transmits beacon signals with the media access control (MAC) address of one of the known routers and use the CNN to identify it as an impersonator.

For more information on router impersonation and validation of the network design with simulated data, see the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” (Communications Toolbox) example.

Train with Captured Data

Collect a dataset of 802.11a/g/n/ac OFDM non-high throughput (non-HT) beacon frames from real WLAN routers. As described in the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” (Communications Toolbox) example, only the legacy long training field (L-LTF) field present in preambles are used as training units in order to avoid any data dependency.

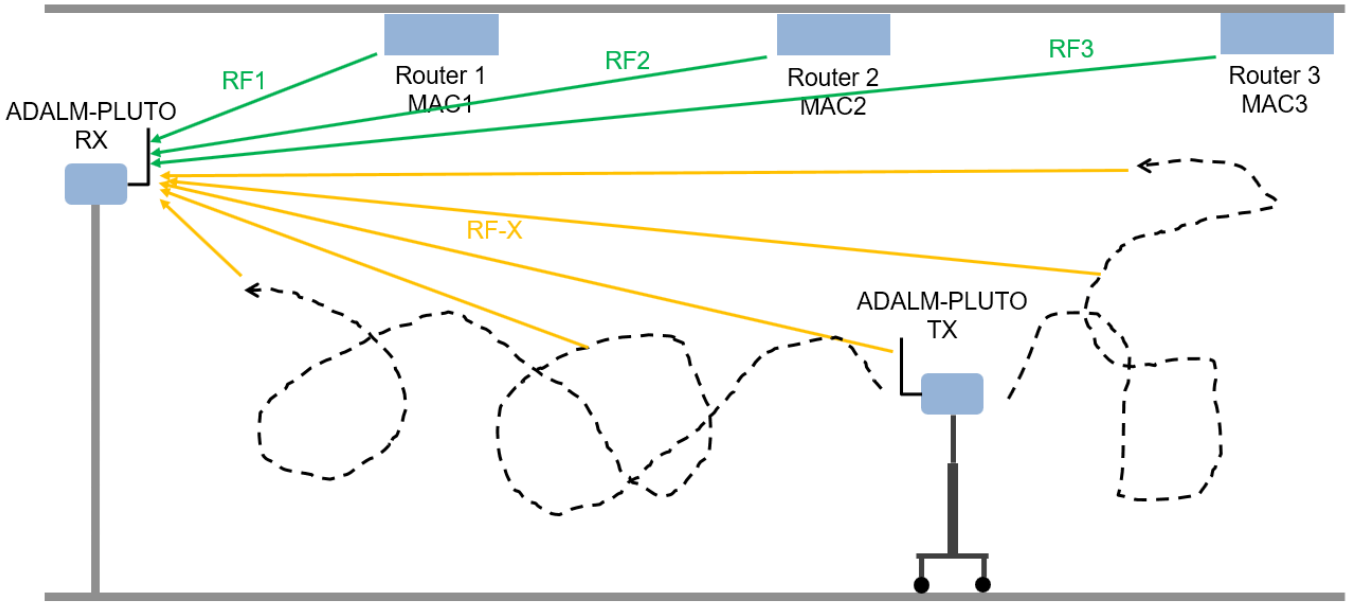
In this example, the data was collected using the scenario depicted in the following figure. The observer is a stationary ADALM-PLUTO radio. Known router data was collected as follows:

1. Set the observer's center frequency based on the WLAN channel used by the routers
2. Receive a beacon frame
3. Extract the L-LTF signal
4. Decode the MAC address to use as the label
5. Save the L-LTF signal together with its label
6. Repeat steps 2-5 to collect `numFramesPerRouter` frames from `numKnownRouters` routers.

Unknown router beacon frames are simulated using a mobile ADALM-PLUTO radio as a transmitter. This radio repeatedly transmits beacon frames with a random MAC address. Unknown router data was collected as follows:

1. Generate beacon frames with a random MAC address
2. Start transmitting the beacon frames repeatedly using the ADALM-PLUTO radio
3. Collect `NUMFRAMES` beacon frames
4. Extract the L-LTF signal
5. Save the L-LTF frames with label "Unknown"
6. Move the radio to another location
7. Repeat steps 3-6 to collect data from `NUMLOC` locations

This combined dataset of known and unknown routers is used to train the same DL model as in the “Design a Deep Neural Network with Simulated Data to Detect WLAN Router Impersonation” (Communications Toolbox) example.



This example downloads training data and trained network from https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData.tar.gz. If you do not have an Internet connection, you can download the file manually on a computer that is connected to the Internet and save to the same directory as the current example files. For privacy reasons, MAC addresses have been anonymized in the downloaded data. To replicate the results of this example, capture your own data as described in Appendix: Known and Unknown Router Data Collection on page 13-0 .

```
rfFingerprintingDownloadData('captured')
```

```
Starting download of data files from:
    https://www.mathworks.com/supportfiles/spc/RFFingerprinting/RFFingerprintingCapturedData.tar
Download and extracting files done
```

To run this example quickly, use the downloaded pretrained network. To train the network on your computer, choose the "Train network now" option (i.e. set `trainNow` to true). Training this network takes about 5 minutes with an Nvidia(R) Titan Xp GPU. Training on a CPU may result in a very long training duration.

`trainNow =` Use trained network ; `%#ok<*UNRCH>`

This example uses data from four known routers. The dataset contains 3600 frames per router, where 90% is used as training frames and 10% is used as test frames.

```
numKnownRouters = 4;
numFramesPerRouter = 3600;
numTrainingFramesPerRouter = numFramesPerRouter * 0.9;
numTestFramesPerRouter = numFramesPerRouter * 0.1;
frameLength = 160;
```

Preprocess Known and Unknown Router Data

Separate collected complex baseband data into its in-phase and quadrature components and reshape it into a *2 x frameLength x 1 x numFramesPerRouter*numKnownRouters* matrix. Repeat the same process for the unknown router data. The following code uses previously collected and pre-processed data. To use your own data, first collect data as described in Appendix: Known and Unknown Router Data Collection on page 13-0 . Copy the new data files named `rfFingerprintingCapturedDataUser.mat` and `rfFingerprintingCapturedUnknownFramesUser.mat` to the same directory as this example. Then update the `load` commands to load these files.

```
if trainNow
  % Load known router data
  load('rfFingerprintingCapturedData.mat')

  % Create label vectors
  yTrain = repelem(MACAddresses, numTrainingFramesPerRouter);
  yTest = repelem(MACAddresses, numTestFramesPerRouter);

  % Separate between I and Q
  numTrainingSamples = numTrainingFramesPerRouter*numKnownRouters*frameLength;
  xTrainingFrames = xTrainingFrames(1:numTrainingSamples,1);
  xTrainingFrames = [real(xTrainingFrames), imag(xTrainingFrames)];
  numTestSamples = numTestFramesPerRouter*numKnownRouters*frameLength;
  xTestFrames = xTestFrames(1:numTestSamples,1);
  xTestFrames = [real(xTestFrames), imag(xTestFrames)];

  % Reshape dataset into an 2 x frameLength x 1 x numTrainingFramesPerRouter*numKnownRouters mat
  xTrainingFrames = permute(...
    reshape(xTrainingFrames,[frameLength,numTrainingFramesPerRouter*numKnownRouters, 2, 1]),...
    [1 3 4 2]);

  % Reshape dataset into an 2 x frameLength x 1 x numTestFramesPerRouter*numKnownRouters matrix
  xTestFrames = permute(...
    reshape(xTestFrames,[frameLength,numTestFramesPerRouter*numKnownRouters, 2, 1]),...
    [1 3 4 2]);

  % Load unknown router data
  load('rfFingerprintingCapturedUnknownFrames.mat')

  % Number of training units
  numUnknownFrames = size(unknownFrames, 4);

  % Split data into 90% training and 10% test
  numUnknownTrainingFrames = floor(numUnknownFrames*0.9);
  numUnknownTest = numUnknownFrames - numUnknownTrainingFrames;

  % Add ADALM-PLUTO data into training and test datasets
  xTrainingFrames(:,:,:,(1:numUnknownTrainingFrames) + numTrainingFramesPerRouter*numKnownRouters
    = unknownFrames(:,:,:, 1:numUnknownTrainingFrames);
  xTestFrames(:,:,:,(1:numUnknownTest) + numTestFramesPerRouter*numKnownRouters) ...
    = unknownFrames(:,:,:, (1:numUnknownTest) + numUnknownTrainingFrames);

  % Shuffle data
  vr = randperm(numKnownRouters*numTrainingFramesPerRouter+numUnknownTrainingFrames);
  xTrainingFrames = xTrainingFrames(:,:,:,vr);
```

```
  % Add "unknown" label and shuffle
  yTrain = [yTrain, repmat("Unknown", [1, numUnknownTrainingFrames])];
  yTrain = categorical(yTrain(vr));

  yTest = [yTest, repmat("Unknown", [1, numUnknownTest])];
  yTest = categorical(yTest);
end
```

Train the CNN

Use the same NN architecture and training options as in the training with simulated data example.

```
poolSize = [2 1];
strideSize = [2 1];
% Create network architecture
layers = [
  imageInputLayer([frameLength 2 1], 'Normalization', 'none', 'Name', 'Input Layer')

  convolution2dLayer([7 1], 50, 'Padding', [1 0], 'Name', 'CNN1')
  batchNormalizationLayer('Name', 'BN1')
  leakyReluLayer('Name', 'LeakyReLu1')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool1')

  convolution2dLayer([7 2], 50, 'Padding', [1 0], 'Name', 'CNN2')
  batchNormalizationLayer('Name', 'BN2')
  leakyReluLayer('Name', 'LeakyReLu2')
  maxPooling2dLayer(poolSize, 'Stride', strideSize, 'Name', 'MaxPool2')

  fullyConnectedLayer(256, 'Name', 'FC1')
  leakyReluLayer('Name', 'LeakyReLu3')
  dropoutLayer(0.5, 'Name', 'DropOut1')

  fullyConnectedLayer(80, 'Name', 'FC2')
  leakyReluLayer('Name', 'LeakyReLu4')
  dropoutLayer(0.5, 'Name', 'DropOut2')

  fullyConnectedLayer(numKnownRouters+1, 'Name', 'FC3')
  softmaxLayer('Name', 'SoftMax')
  classificationLayer('Name', 'Output')
  ];
```

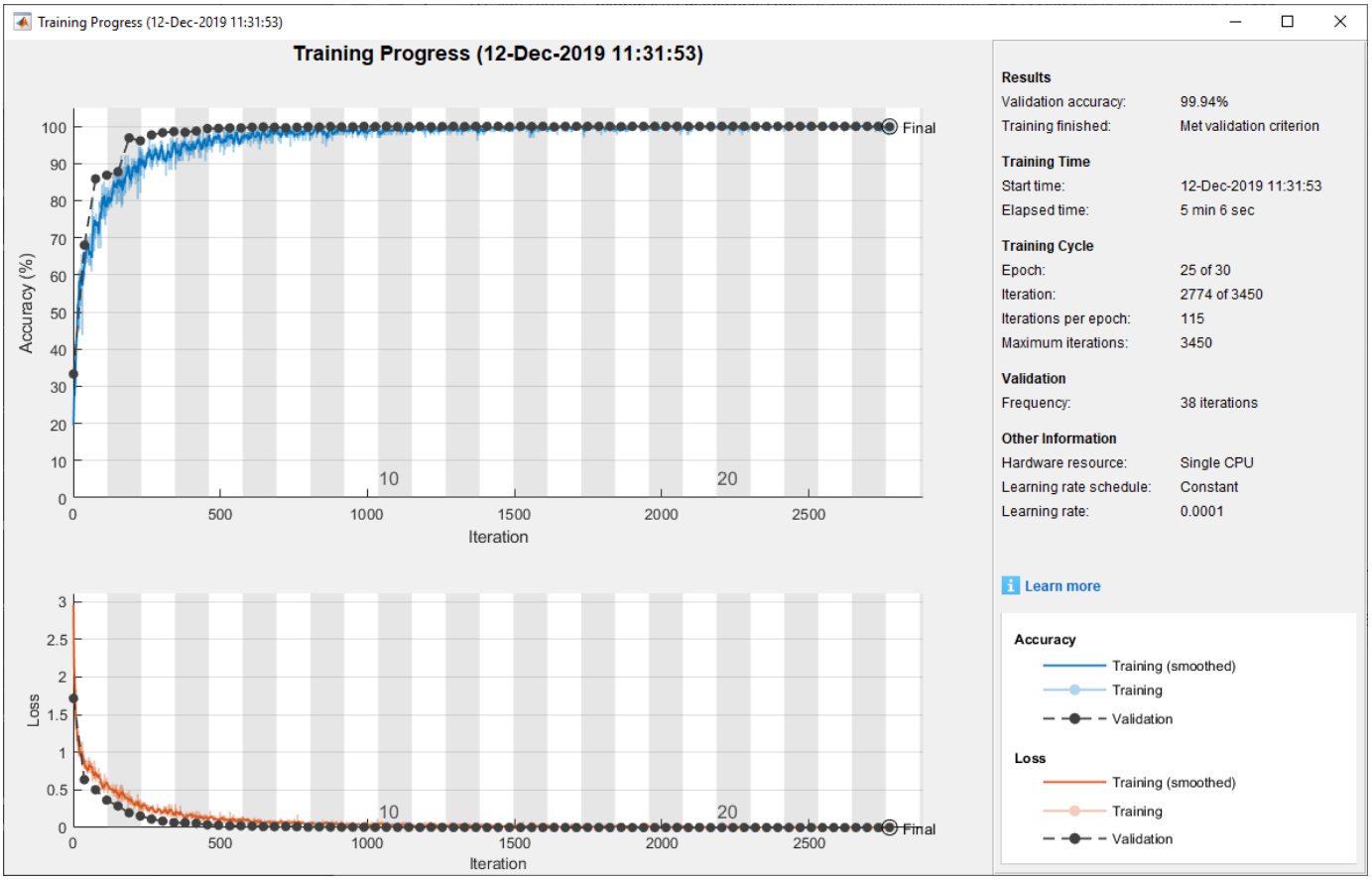
Configure the training options to use ADAM optimizer with a mini-batch size of 128. Use test frames for validation since optimization of hyperparameters were done in [1] on page 13-0 .

By default, `ExecutionEnvironment` is set to `'auto'`, which uses a GPU for training if one is available. Otherwise, `trainNetwork` uses the CPU for training. To explicitly set the execution environment, set `ExecutionEnvironment` to one of `'cpu'`, `'gpu'`, `'multi-gpu'`, or `'parallel'`. Choosing `'cpu'` may result in a very long training duration.

```
if trainNow
  miniBatchSize = 128;

  % Training options
  options = trainingOptions('adam', ...
    'MaxEpochs',30, ...
    'ValidationData',{xTestFrames, yTest}, ...
    'ValidationFrequency',floor((numTrainingFramesPerRouter*numKnownRouters + numUnknownTrainingF
    'Verbose',false, ...
```

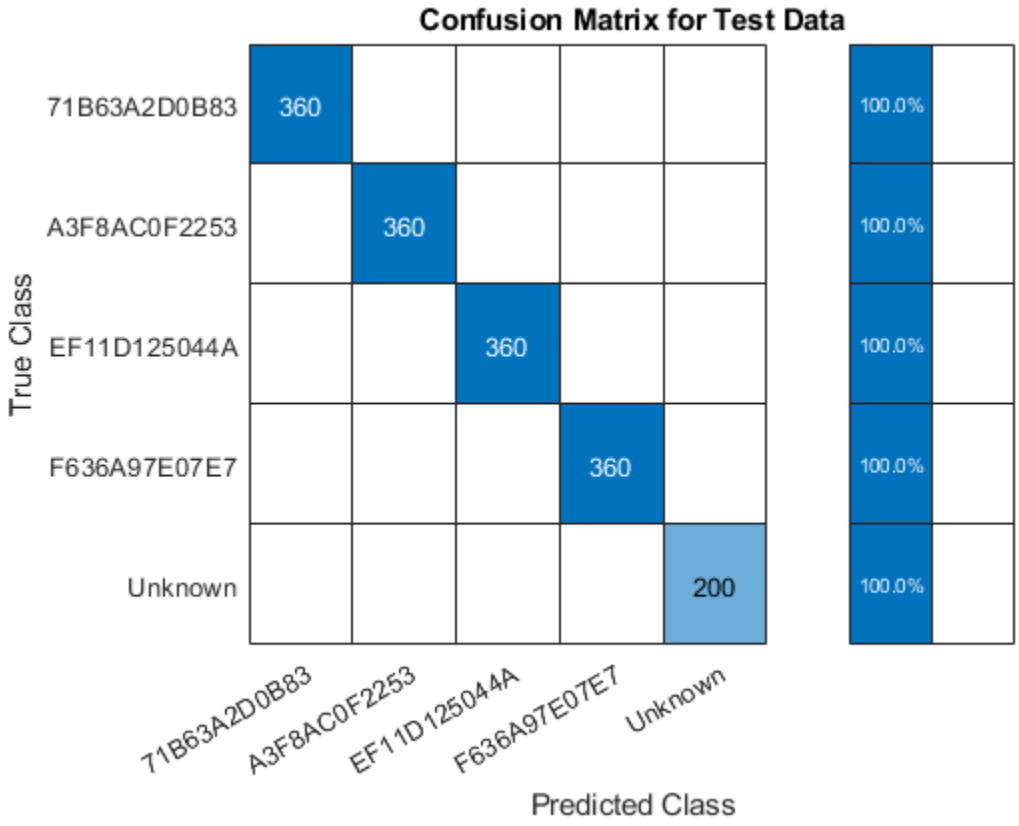
```
    'L2Regularization', 0.0001, ...
    'InitialLearnRate', 0.0001, ...
    'MiniBatchSize', miniBatchSize, ...
    'ValidationPatience', 5, ...
    'Plots','training-progress', ...
    'Shuffle', 'every-epoch');

  % Train the network
  capturedDataNet = trainNetwork(xTrainingFrames, yTrain, layers, options);
else
  load('rfFingerprintingCapturedDataTrainedNN.mat','capturedDataNet','xTestFrames','yTest','MACAd
end
```

The following plot shows the training progress of the network run on a computer with a single Nvidia Titan Xp GPU, where the network converged in about 10 epochs to almost 100% accuracy. The final accuracy of the network is 100%.



Generate the confusion matrix.

```
figure
yTestPred = classify(capturedDataNet,xTestFrames);
cm = confusionchart(yTest, yTestPred);
cm.Title = 'Confusion Matrix for Test Data';
cm.RowSummary = 'row-normalized';
```



Test with SDR

Test the performance of the trained network on the class "Unknown". Generate beacon frames with MAC addresses of the known routers and one unknown router. Transmit these frames using an ADALM-PLUTO radio and receive using another ADALM-PLUTO radio. Since the channel and RF impairments created between these two radios are different than the ones created between the real routers and the observer, the neural network should classify all of the received signals as "Unknown". If the received MAC address is a known one, then the system declares the source as a router impersonator. If the received MAC address is an unknown one, then the system declares the source as an unknown router. To perform this test, you need two ADALM-PLUTO radios for transmission and reception. Also, you need to install Communication Toolbox Support Package for ADALM-PLUTO Radio.

Waveform Generation

Generate a transmission waveform consisting of beacon frames with different MAC addresses. The transmitter repeatedly transmits these WLAN frames. The receiver captures the WLAN frames and determines if it is a router impersonator using the received MAC address and RF fingerprint detected by the trained NN.

```
chanBW='CBW20';     % Channel Bandwidth
osf = 2;            % Oversampling Factor
frameLength=160;    % Frame Length in samples
% Create Beacon frame-body configuration object
frameBodyConfig = wlanMACManagementConfig;
```

```
% Create Beacon frame configuration object
beaconFrameConfig = wlanMACFrameConfig('FrameType', 'Beacon');
beaconFrameConfig.ManagementConfig = frameBodyConfig;

% Create interpolation and decimation objects
decimator = dsp.FIRDecimator('DecimationFactor',osf);

% Save known MAC addresses
knownMACAddresses = MACAddresses;
MACAddressesToSimulate = [MACAddresses, "ABCDEFABCDEF"];

% Create WLAN waveform with the MAC addresses of known routers and an
% unknown router
txWaveform = zeros(1540,5);
for i = 1:length(MACAddressesToSimulate)

  % Set MAC Address
  beaconFrameConfig.Address2 = MACAddressesToSimulate(i);

  % Generate Beacon frame bits
  [beacon, mpduLength] = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

  nonHTcfg = wlanNonHTConfig(...
    'ChannelBandwidth', chanBW,...
    "MCS", 1,...
    "PSDULength", mpduLength);
  txWaveform(:,i) = [wlanWaveformGenerator(beacon, nonHTcfg); zeros(20,1)];
end

txWaveform = txWaveform(:);

% Get center frequency for channel 153 in 5 GHz band
fc = helperWLANChannelFrequency(153, 5);
fs = wlanSampleRate(nonHTcfg);

txSig  = resample(txWaveform,osf,1);

% Samples per frame in Burst Mode
spf = length(txSig)/length(MACAddressesToSimulate);

runSDRSection = false;
if helperIsPlutoSDRInstalled()
  radios = findPlutoRadio();
  if length(radios) >= 2
    runSDRSection = true;
  else
    disp("Two ADALM-PLUTO radios are needed. Skipping SDR test.")
  end
else
    disp("Communications Toolbox Support Package for Analog Devices ADALM-PLUTO Radio not found.
    disp("Click Add-Ons in the Home tab of the MATLAB toolstrip to install the support package."
    disp("Skipping SDR test.")
end


if runSDRSection
  % Set up PlutoSDR transmitter
  deviceNameSDR = 'Pluto';
```

```
  txGain = 0;
  txSDR = sdrtx(deviceNameSDR);
  txSDR.RadioID = 'usb:0';
  txSDR.BasebandSampleRate = fs*osf;
  txSDR.CenterFrequency = fc;
  txSDR.Gain = txGain;

  % Set up PlutoSDR Receiver
  rxSDR = sdrrx(deviceNameSDR);
  rxSDR.RadioID = 'usb:1';
  rxSDR.BasebandSampleRate = txSDR.BasebandSampleRate;
  rxSDR.CenterFrequency = txSDR.CenterFrequency;
  rxSDR.GainSource ='Manual';
  rxSDR.Gain = 30;
  rxSDR.OutputDataType = 'double';
  rxSDR.EnableBurstMode=true;
  rxSDR.NumFramesInBurst = 20;
  rxSDR.SamplesPerFrame = osf*spf;
end
```

L-LTF for Classification

The L-LTF sequence present in each beacon frame preamble is used as input units to the NN. rfFingerprintingNonHTFrontEnd System object is used to detect the WLAN packets, perform synchronization tasks and, extract the L-LTF sequences and data. In addition, the MAC address is also decoded. In addition, the data is pre-processed and classified using the trained network.

```
if runSDRSection
  numLLTF = 20;       % Number of L-LTF captured for Testing

  rxFrontEnd = rfFingerprintingNonHTFrontEnd('ChannelBandwidth', 'CBW20');

  disp("The known MAC addresses are:");
  disp(knownMACAddresses)

  % Set PlutoSDR to transmit repeatedly
  disp('Starting transmitter')
  transmitRepeat(txSDR, txSig);

  % Captured Frames counter
  numCapturedFrames = 0;

  disp('Starting receiver')
  % Loop until numLLTF frames are collected
  while numCapturedFrames < numLLTF

    % Receive data using PlutoSDR
    rxSig = rxSDR();
    rxSig = decimator(rxSig);

    % Perform front-end processing and payload buffering
    [payloadFull, cfgNonHT, rxNonHTData, chanEst, noiseVar, LLTF] = ...
      rxFrontEnd(rxSig);

    if payloadFull

      % Recover payload bits
      recBits = wlanNonHTDataRecover(rxNonHTData, chanEst, ...
```

```
        noiseVar, cfgNonHT, 'EqualizationMethod', 'ZF');

      % Decode and evaluate recovered bits
      [mpduCfg, ~, success] = wlanMPDUDecode(recBits, cfgNonHT);

      if success == wlanMACDecodeStatus.Success
        % Update counter
        numCapturedFrames = numCapturedFrames+1;

        % Create real-valued input
        LLTF = [real(LLTF), imag(LLTF)];
        LLTF = permute(reshape(LLTF,frameLength ,[] , 2, 1), [1 3 4 2]);

        ypred = classify(capturedDataNet, LLTF);

        if sum(contains(knownMACAddresses, mpduCfg.Address2)) ~= 0
          if categorical(convertCharsToStrings(mpduCfg.Address2))~=ypred
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint mismatch, ROUTE
          else
            disp(strcat("MAC Address ", mpduCfg.Address2," is known, fingerprint match"))
          end
        else
          disp(strcat("MAC Address ", mpduCfg.Address2," is not recognized, unknown device"));
        end
      end
    end
  end
  release(txSDR)
end
```

```
The known MAC addresses are:

    "71B63A2D0B83"    "A3F8AC0F2253"    "EF11D125044A"    "F636A97E07E7"

Starting transmitter

## Establishing connection to hardware. This process can take several seconds.
## Waveform transmission has started successfully and will repeat indefinitely.
## Call the release method to stop the transmission.

Starting receiver

## Establishing connection to hardware. This process can take several seconds.

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device
```

```
MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address EF11D125044A is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED

MAC Address ABCDEFABCDEF is not recognized, unknown device

MAC Address A3F8AC0F2253 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address F636A97E07E7 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
MAC Address 71B63A2D0B83 is known, fingerprint mismatch, ROUTER IMPERSONATOR DETECTED
```

Further Exploration

Capture data from your own routers as explained in Appendix: Known and Unknown Router Data Collection, on page 13-0 train the neural network with this data, and test the performance of the network.

Appendix: Helper Functions

- rfFingerprintingRouterDataCollection
- rfFingerprintingUnknownClassDataCollectionTx
- rfFingerprintingUnknownClassDataCollectionRx
- rfFingerprintingNonHTFrontEnd
- rfFingerprintingNonHTReceiver

Appendix: Known and Unknown Router Data Collection

Use `rfFingerprintingRouterDataCollection` to collect data from known (i.e. trusted) routers. This function extracts L-LTF signals present in 802.11a/g/n/ac OFDM Non-HT beacons frames transmitted from commercial 802.11 hardware. For more information see the “IEEE® 802.11™ WLAN - OFDM Beacon Receiver with USRP® Hardware” (Communications Toolbox Support Package for USRP Radio) example. L-LTF signals and corresponding router MAC addresses are used to train the RF fingerprinting neural network. This method works best if the routers and their antennas are fixed and hard to move unintentionally. For example, in most office environments, routers are mounted on the ceiling. Follow these steps:

1. Connect an ADALM-PLUTO radio to your PC to use as the observer radio.
2. Place the radio in a central location where it can receive signals from as many routers as possible. Fix the radio so that it does not move. If possible, place the observer radio on the ceiling or high on a wall.
3. Determine the channel number of the routers. You can use a Wi-Fi analyzer app on your phone to find out the channel numbers.
4. Start data collection by running "`rfFingerprintingRouterDataCollection(channel)`" where channel is the Wi-Fi channel number
5. Monitor the "max(abs(LLTF))" value. If it is above 1.2 or smaller than 0.01, adjust the gain of the receiver using the GAIN input of `rfFingerprintingRouterDataCollection` function.

Use the helper functions `rfFingerprintingUnknownClassDataCollectionTx` and `rfFingerprintingUnknownClassDataCollectionRx` to collect data from unknown routers. These functions set two ADALM-PLUTO radios to transmit and receive L-LTF signals. The received signals are combined with the known router signals to train the neural network. You need two ADALM-PLUTO radios, preferably connected to two separate PCs. Follow these steps:

1. Connect an ADALM-PLUTO radio to a stationary PC to act as the unknown router
2. Start transmissions by running "`rfFingerprintingUnknownClassDataCollectionTx`"
3. Connect another ADALM-PLUTO radio to a mobile PC to act as the observer
4. Start data collection by running "`rfFingerprintingUnknownClassDataCollectionRx`". This function by default collects 200 frames per location. Each location represents a different unknown router.
5. When the function instructs you to move to a new location, move the observer radio to a new location. By default, this function collects data from 10 locations.
6. If the observer does not receive any beacons or it rarely receives beacons, move the observer closer to the transmitter.
7. Once the data collection is done, call "`release(sdrTransmitter)`" in the transmitting radio's MATLAB session.

Selected Bibliography

[1] K. Sankhe, M. Belgiovine, F. Zhou, S. Riyaz, S. Ioannidis and K. Chowdhury, "ORACLE: Optimized Radio clAssification through Convolutional neuraL nEtworks," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 370-378.

See Also

More About

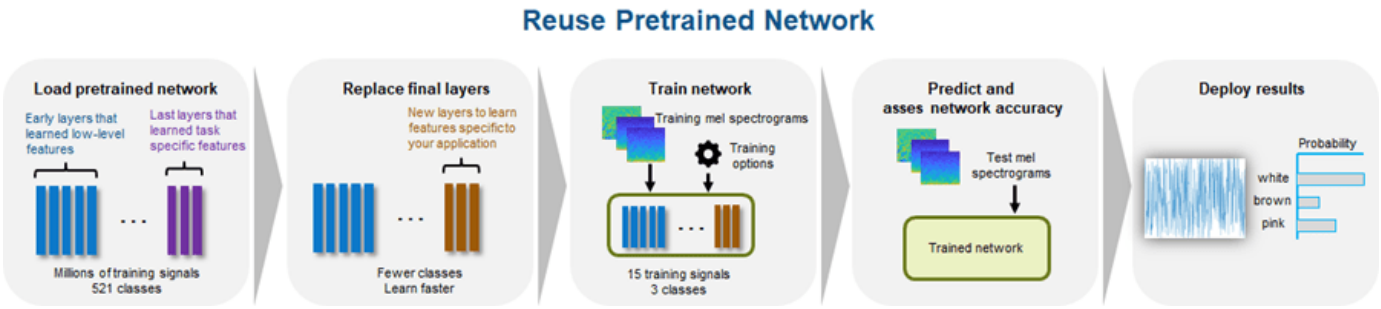
- “Deep Learning in MATLAB” on page 1-2

Audio Examples

Transfer Learning with Pretrained Audio Networks

This example shows how to use transfer learning to retrain YAMNet, a pretrained convolutional neural network, to classify a new set of audio signals. To get started with audio deep learning from scratch, see “Classify Sound Using Deep Learning” (Audio Toolbox).

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training signals.



Audio Toolbox™ additionally provides the `classifySound` (Audio Toolbox) function, which implements necessary preprocessing for YAMNet and convenient postprocessing to interpret the results. Audio Toolbox also provides the pretrained VGGish network (`vggish` (Audio Toolbox)) as well as the `vggishFeatures` (Audio Toolbox) function, which implements preprocessing and postprocessing for the VGGish network.

Create Data

Generate 100 white noise signals, 100 brown noise signals, and 100 pink noise signals. Each signal represents a duration of 0.98 seconds assuming a 16 kHz sample rate.

```
fs = 16e3;
duration = 0.98;
N = duration*fs;
numSignals = 100;

wNoise = 2*rand([N,numSignals]) - 1;
wLabels = repelem(categorical("white"),numSignals,1);

bNoise = filter(1,[1,-0.999],wNoise);
bNoise = bNoise./max(abs(bNoise),[],'all');
bLabels = repelem(categorical("brown"),numSignals,1);

pNoise = pinknoise([N,numSignals]);
pLabels = repelem(categorical("pink"),numSignals,1);
```

Split the data into training and test sets. Normally, the training set consists of most of the data. However, to illustrate the power of transfer learning, you will use only a few samples for training and the majority for validation.

```
K = 5 ;
```

```
trainAudio = [wNoise(:,1:K),bNoise(:,1:K),pNoise(:,1:K)];
trainLabels = [wLabels(1:K);bLabels(1:K);pLabels(1:K)];

validationAudio = [wNoise(:,K+1:end),bNoise(:,K+1:end),pNoise(:,K+1:end)];
validationLabels = [wLabels(K+1:end);bLabels(K+1:end);pLabels(K+1:end)];

fprintf("Number of samples per noise color in train set = %d\n" + ...
        "Number of samples per noise color in validation set = %d\n",K,numSignals-K);
```

```
Number of samples per noise color in train set = 5
Number of samples per noise color in validation set = 95
```

Extract Features

Use `melSpectrogram` (Audio Toolbox) to extract log-mel spectrograms from both the training set and the validation set using the same parameters as the YAMNet model was trained on.

```
FFTLength = 512;
numBands = 64;
frequencyRange = [125 7500];
windowLength = 0.025*fs;
overlapLength = 0.015*fs;

trainFeatures = melSpectrogram(trainAudio,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');

trainFeatures = log(trainFeatures + single(0.001));
trainFeatures = permute(trainFeatures,[2,1,4,3]);

validationFeatures = melSpectrogram(validationAudio,fs, ...
    'Window',hann(windowLength,'periodic'), ...
    'OverlapLength',overlapLength, ...
    'FFTLength',FFTLength, ...
    'FrequencyRange',frequencyRange, ...
    'NumBands',numBands, ...
    'FilterBankNormalization','none', ...
    'WindowNormalization',false, ...
    'SpectrumType','magnitude', ...
    'FilterBankDesignDomain','warped');

validationFeatures = log(validationFeatures + single(0.001));
validationFeatures = permute(validationFeatures,[2,1,4,3]);
```

Transfer Learning

To load the pretrained network, call `yamnet` (Audio Toolbox). If the Audio Toolbox model for YAMNet is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path. The YAMNet model can classify audio into one of 521 sound categories, including white noise and pink noise (but not brown noise).

```
net = yamnet;
net.Layers(end).Classes

ans = 521×1 categorical
     Speech
     Child speech, kid speaking
     Conversation
     Narration, monologue
     Babbling
     Speech synthesizer
     Shout
     Bellow
     Whoop
     Yell
     Children shouting
     Screaming
     Whispering
     Laughter
     Baby laughter
     Giggle
     Snicker
     Belly laugh
     Chuckle, chortle
     Crying, sobbing
     Baby cry, infant cry
     Whimper
     Wail, moan
     Sigh
     Singing
     Choir
     Yodeling
     Chant
     Mantra
     Child singing
      ⋮
```

Prepare the model for transfer learning by first converting the network to a `layerGraph`. Use `replaceLayer` to replace the fully-connected layer with an untrained fully-connected layer. Replace the classification layer with a classification layer that classifies the input as "white", "pink", or "brown". See “List of Deep Learning Layers” on page 1-21 for deep learning layers supported in MATLAB®.

```
uniqueLabels = unique(trainLabels);
numLabels = numel(uniqueLabels);

lgraph = layerGraph(net.Layers);

lgraph = replaceLayer(lgraph,"dense",fullyConnectedLayer(numLabels,"Name","dense"));
lgraph = replaceLayer(lgraph,"Sound",classificationLayer("Name","Sounds","Classes",uniqueLabels))
```

To define training options, use `trainingOptions`.

```
options = trainingOptions('adam','ValidationData',{single(validationFeatures),validationLabels})
```

To train the network, use `trainNetwork`. The network achieves a validation accuracy of 100% using only 5 signals per noise type.

```
trainNetwork(single(trainFeatures),trainLabels,lgraph,options);
```

```
Training on single CPU.
|=========================================================================================================
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validati
|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss
|=========================================================================================================
|       1 |           1 |       00:00:02 |       20.00% |       88.77% |       1.1922 |       0.6
|      30 |          30 |       00:00:14 |      100.00% |      100.00% |   9.1076e-06 |   5.0431e
|=========================================================================================================
```

Speech Command Recognition in Simulink

This example shows a Simulink model that detects the presence of speech commands in audio. The model uses a pretrained convolutional neural network to recognize a given set of commands.

Speech Command Recognition Model

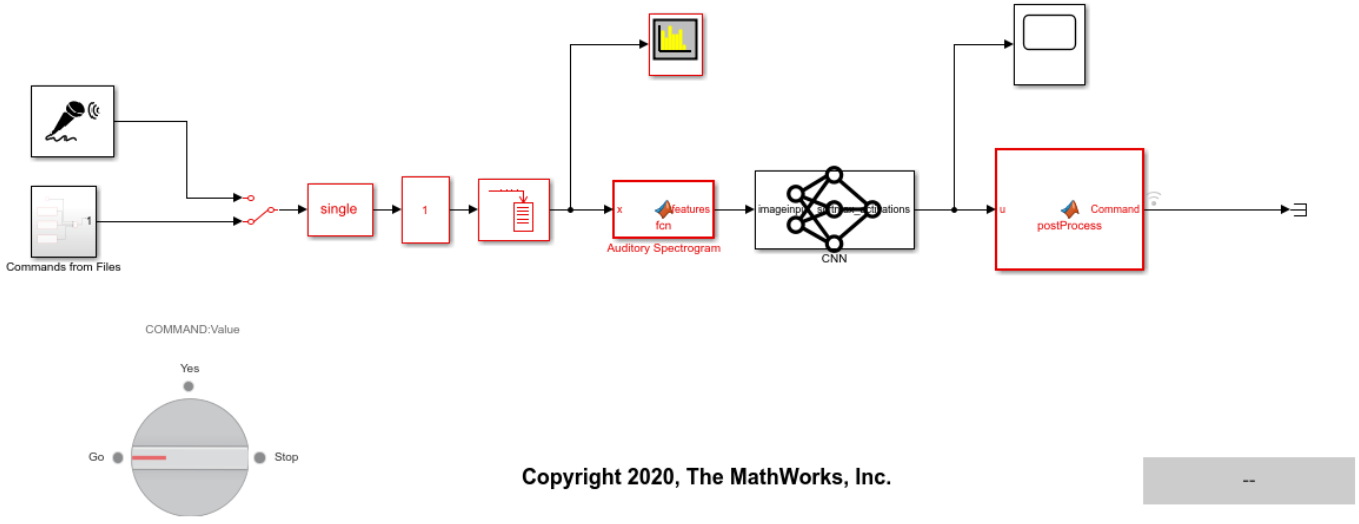
The model recognizes the following speech commands:

- "yes"
- "no"
- "up"
- "down"
- "left"
- "right"
- "on"
- "off"
- "stop"
- "go"

The model uses a pretrained convolutional deep learning network. Refer to the example “Speech Command Recognition Using Deep Learning” (Audio Toolbox) for details on the architecture of this network and how you train it.

Open the model.

```
model = 'speechCommandRecognition';
open_system(model)
```



The model breaks the audio stream into one-second overlapping segments. A mel spectrogram is computed from each segment. The spectrograms are fed to the pretrained network.

Use the manual switch to select either a live stream from your microphone or read commands stored in audio files. For commands on file, use the rotary switch to select one of three commands (Go, Yes and Stop).

Auditory Spectrogram Extraction

The deep learning network is trained on auditory spectrograms computed using an audioFeatureExtractor. For the model to classify commands properly, you must extract auditory spectrograms in a manner identical to the trainind stage.

Define the parameters of the feature extraction. `frameDuration` is the duration of each frame for spectrum calculation. `hopDuration` is the time step between each spectrum. `numBands` is the number of filters in the auditory spectrogram.

```
fs = 16000;
frameDuration = 0.025;
frameSamples = round(frameDuration*fs);
hopDuration = 0.010;
hopSamples = round(hopDuration*fs);
numBands = 50;
```

Define an audioFeatureExtractor object to perform the feature extraction. The object is identical to the one used in “Speech Command Recognition Using Deep Learning” (Audio Toolbox) to extract the training spectrograms.

```
afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',512, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',frameSamples - hopSamples, ...
    'barkSpectrum',true);

setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);
```

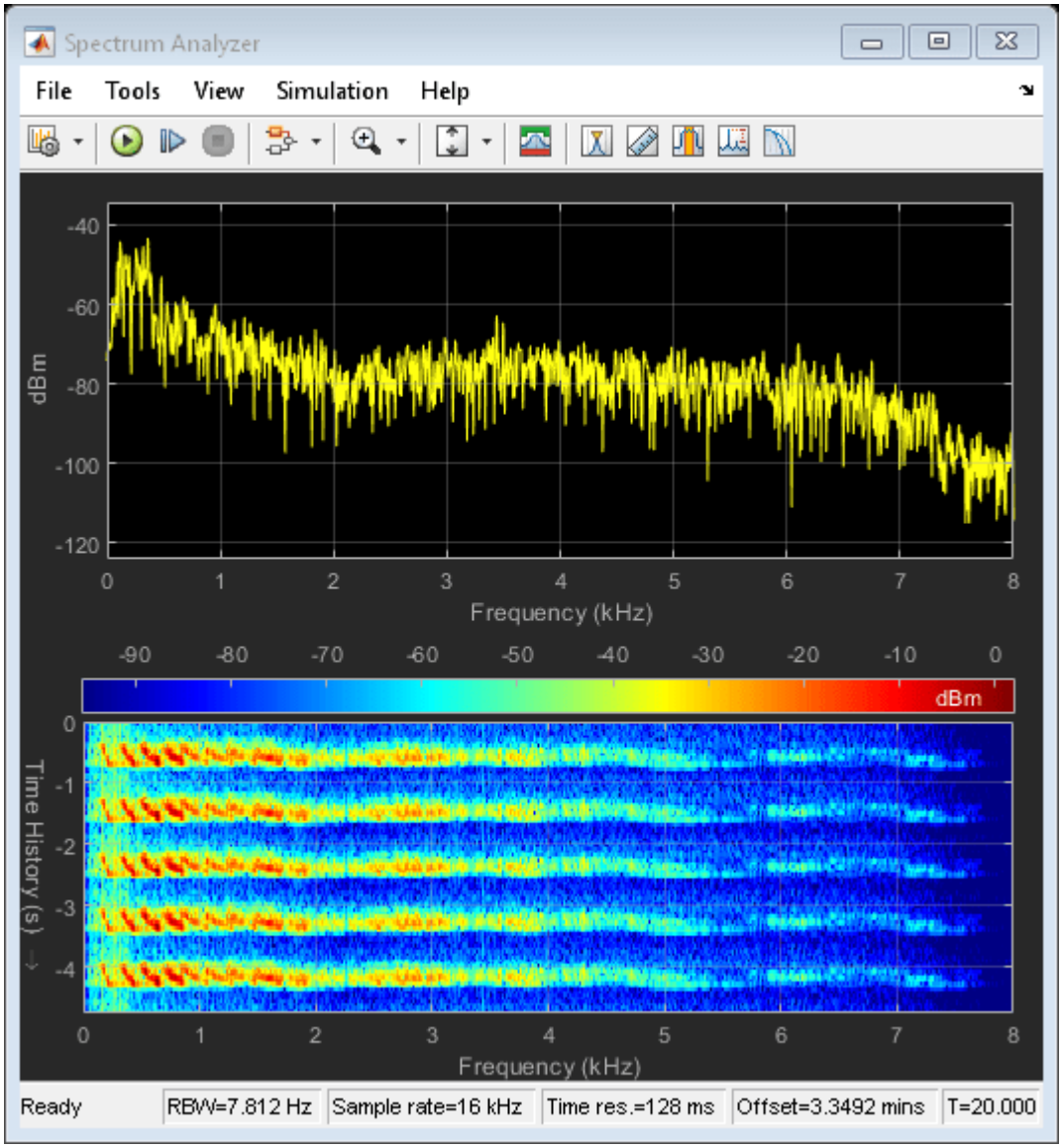
Call generateMATLABFunction to create a feature extraction function. This function is called from the `Auditory Spectrogram` MATLAB Function block in the model. This ensures that the feature extraction used in the model matches the one used in training.

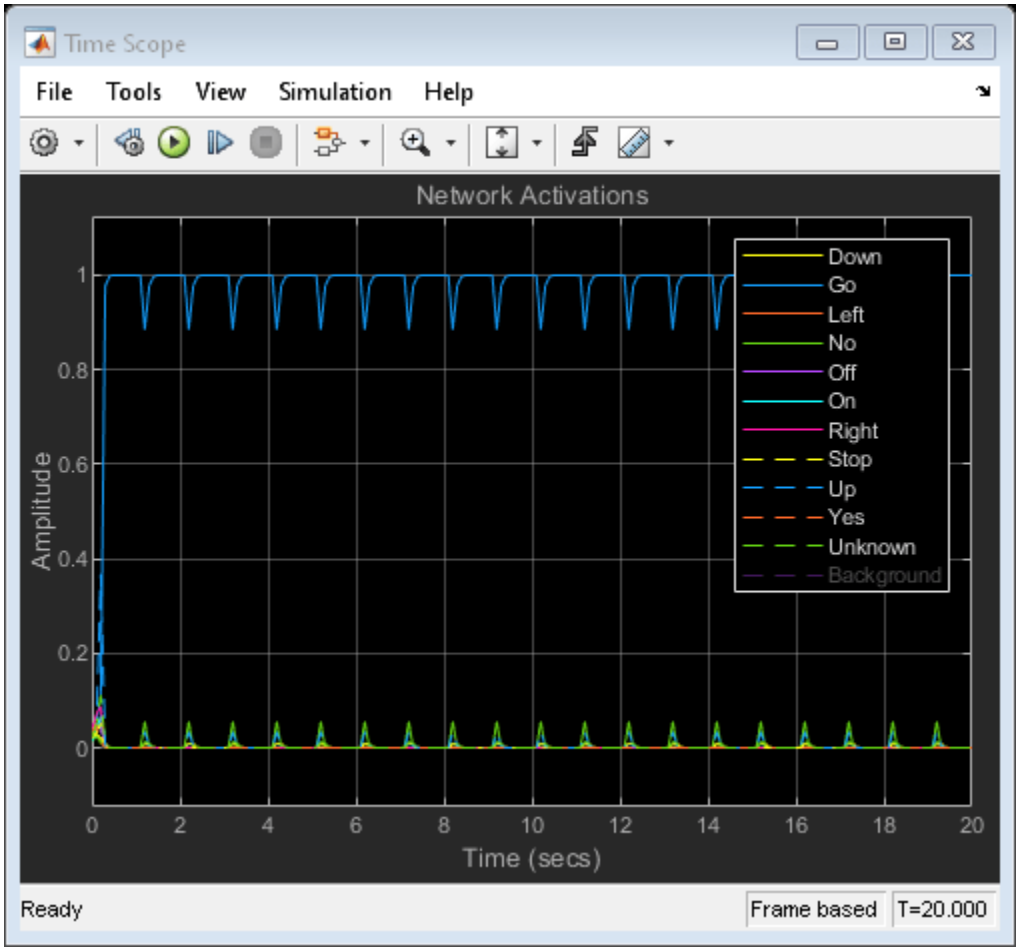
```
generateMATLABFunction(afe,'extractSpeechFeatures')
```

Run the model

Simulate the model for 20 seconds. To run the model indefinitely, set the stop time to Inf.

```
set_param(model,'StopTime','20');
sim(model);
```


Spectrum Analyzer
File
Tools
View
Simulation
Help
dBm
-40
-60
-80
-100
-120
0
1
2
3
4
5
6
7
8
Frequency (kHz)
-90
-80
-70
-60
-50
-40
-30
-20
-10
0
dBm
Time History (s)
0
-1
-2
-3
-4
Frequency (kHz)
Ready
RBW=7.812 Hz
Sample rate=16 kHz
Time res.=128 ms
Offset=3.3492 mins
T=20.000



The recognized command is printed in the display block. The speech spectrogram is displayed in a Spectrum Analyzer scope. The network activations, which give a level of confidence in the different supported commands, are displayed in a time scope.

Close the model.

```
close_system(model,0)
```

Speaker Identification Using Custom SincNet Layer and Deep Learning

In this example, you train three convolutional neural networks (CNNs) to perform speaker verification and then compare the performances of the architectures. The architectures of the three CNNs are all equivalent except for the first convolutional layer in each:

1. In the first architecture, the first convolutional layer is a "standard" convolutional layer, implemented using `convolution2dLayer`.
2. In the second architecture, the first convolutional layer is a constant sinc filterbank, implemented using a custom layer.
3. In the third architecture, the first convolutional layer is a trainable sinc filterbank, implemented using a custom layer. This architecture is referred to as *SincNet* [1] on page 14-0 .

[1] on page 14-0 shows that replacing the standard convolutional layer with a filterbank layer leads to faster training convergence and higher accuracy. [1] on page 14-0 also shows that making the parameters of the filter bank learnable yields additional performance gains.

Introduction

Speaker identification is a prominent research area with a variety of applications including forensics and biometric authentication. Many speaker identification systems depend on precomputed features such as i-vectors or MFCCs, which are then fed into machine learning or deep learning networks for classification. Other deep learning speech systems bypass the feature extraction stage and feed the audio signal directly to the network. In such end-to-end systems, the network directly learns low-level audio signal characteristics.

In this example, you first train a traditional end-to-end speaker identification CNN. The filters learned tend to have random shapes that do not correspond to perceptual evidence or knowledge of how the human ear works, especially in scenarios where the amount of training data is limited [1] on page 14-0 . You then replace the first convolutional layer in the network with a custom sinc filterbank layer that introduces structure and constraints based on perceptual evidence. Finally, you train the SincNet architecture, which adds learnability to the sinc filterbank parameters.

The three neural network architectures explored in the example are summarized as follows:

1. **Standard Convolutional Neural Network** - The input waveform is directly connected to a randomly initialized convolutional layer which attempts to learn features and capture characteristics from the raw audio frames.
2. **ConstantSincLayer** - The input waveform is convolved with a set of fixed-width sinc functions (bandpass filters) equally spaced on the mel scale.
3. **SincNetLayer** - The input waveform is convolved with a set of sinc functions whose parameters are learned by the network. In the SincNet architecture, the network tunes parameters of the sinc functions while training.

This example defines and trains the three neural networks proposed above and evaluates their performance on the LibriSpeech Dataset [2] on page 14-0 .

Data Set

Download Dataset

In this example, you use a subset of the LibriSpeech Dataset [2] on page 14-0 . The LibriSpeech Dataset is a large corpus of read English speech sampled at 16 kHz. The data is derived from audiobooks read from the LibriVox project.

```
downloadDatasetFolder = tempdir;

filename = "train-clean-100.tar.gz";
url = "http://www.openSLR.org/resources/12/" + filename;

datasetFolder = fullfile(downloadDatasetFolder,"LibriSpeech","train-clean-100");

if ~isfolder(datasetFolder)
    gunzip(url,downloadDatasetFolder);
    unzippedFile = fullfile(downloadDatasetFolder,filename);
    untar(unzippedFile{1}(1:end-3),downloadDatasetFolder);
end
```

Create an `audioDatastore` object to access the LibriSpeech audio data.

```
ADS = audioDatastore(datasetFolder,'IncludeSubfolders',1);
```

Extract the speaker label from the file path.

```
ADS.Labels = extractBetween(ADS.Files,fullfile(datasetFolder,filesep),filesep);
```

The full `dev-train-100` dataset is around 6 GB of data. To train the network with data from all 251 speakers, set `reduceDataset` to `false`. To run this example quickly with data from just six speakers, set `reduceDataset` to `true`.

```
reducedDataSet = false;
if reducedDataSet
    indices = cellfun(@(c)str2double(c)<50,ADS.Labels);  %#ok
    ADS = subset(ADS,indices);
end
ADS = splitEachLabel(ADS,0.1);
```

Split the audio files into training and test data. 80% of the audio files are assigned to the training set and 20% are assigned to the test set.

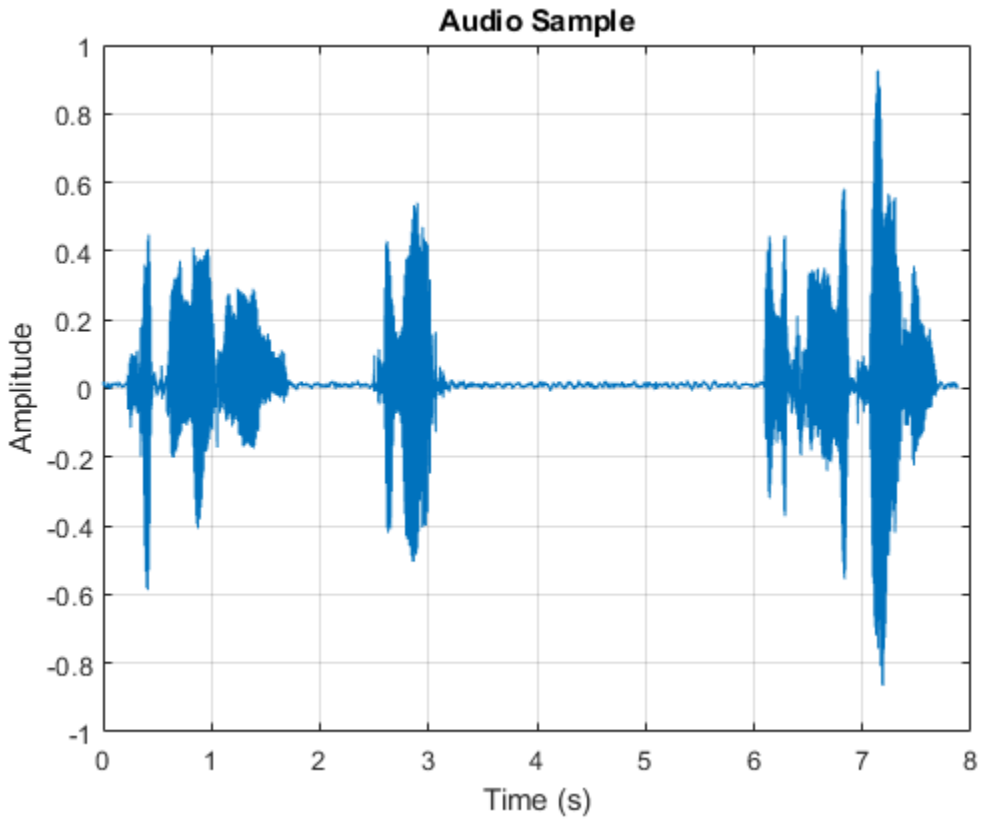
```
[ADSTrain,ADSTest] = splitEachLabel(ADS,0.8);
```

Sample Speech Signal

Plot one of the audio files and listen to it.

```
[audioIn,dsInfo] = read(ADSTrain);
Fs = dsInfo.SampleRate;

sound(audioIn,Fs)

t = (1/Fs)*(0:length(audioIn)-1);

plot(t,audioIn)
```

```
title("Audio Sample")
xlabel("Time (s)")
ylabel("Amplitude")
grid on
```



Reset the training datastore.

```
reset(ADSTrain)
```

Data Preprocessing

CNNs expect inputs to have consistent dimensions. You will preprocess the audio by removing regions of silence and then break the remaining speech into 200 ms frames with 40 ms overlap.

Set the parameters for preprocessing.

```
frameDuration = 200e-3;
overlapDuration = 40e-3;
frameLength = floor(Fs*frameDuration);
overlapLength = round(Fs*overlapDuration);
```

Use the supporting function, preprocessAudioData on page 14-0 , to preprocess the training and test data. `XTrain` and `XTest` contain the train and test speech frames, respectively. `YTrain` and `YTest` contain the train and test labels, respectively.

```
[XTrain,YTrain] = preprocessAudioData(ADSTrain,frameLength,overlapLength,Fs);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
[XTest,YTest] = preprocessAudioData(ADSTest,frameLength,overlapLength,Fs);
```

Standard CNN

Define Layers

The standard CNN is inspired by the neural network architecture in [1] on page 14-0 .

```
numFilters = 80;
filterLength = 251;
numSpeakers = numel(unique(ADS.Labels));

layers = [
    imageInputLayer([1 frameLength 1])

    % First convolutional layer

    convolution2dLayer([1 filterLength],numFilters)
    batchNormalizationLayer
    leakyReluLayer(0.2)
    maxPooling2dLayer([1 3])

    % This layer is followed by 2 convolutional layers

    convolution2dLayer([1 5],60)
    batchNormalizationLayer
    leakyReluLayer(0.2)
    maxPooling2dLayer([1 3])

    convolution2dLayer([1 5],60)
    batchNormalizationLayer
    leakyReluLayer(0.2)
    maxPooling2dLayer([1 3])

    % This is followed by 3 fully-connected layers

    fullyConnectedLayer(256)
    batchNormalizationLayer
    leakyReluLayer(0.2)

    fullyConnectedLayer(256)
    batchNormalizationLayer
    leakyReluLayer(0.2)

    fullyConnectedLayer(256)
    batchNormalizationLayer
    leakyReluLayer(0.2)

    fullyConnectedLayer(numSpeakers)
    softmaxLayer
    classificationLayer];
```

Analyze the layers of the neural network using the `analyzeNetwork` function

```
analyzeNetwork(layers)
```

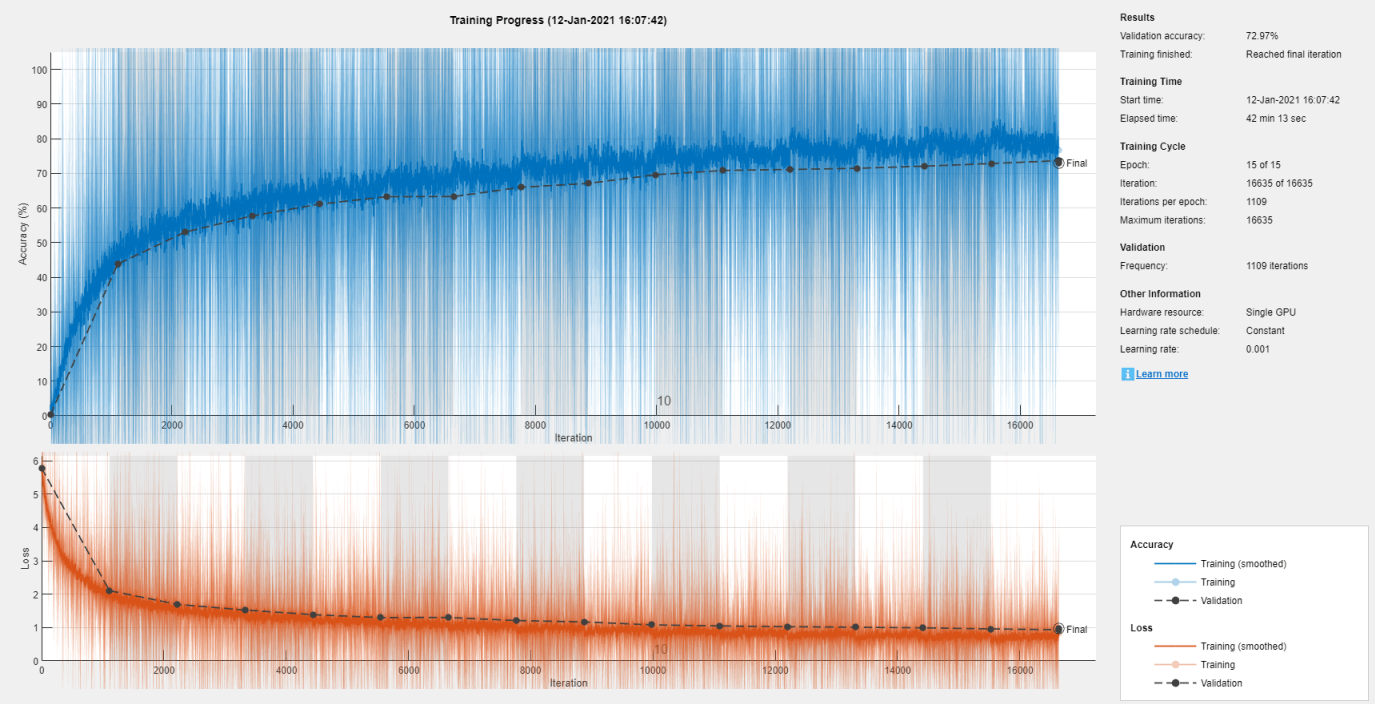
Train Network

Train the neural network for 15 epochs using `adam` optimization. Shuffle the training data before every epoch. The training options for the neural network are set using `trainingOptions`. Use the test data as the validation data to observe how the network performance improves as training progresses.

```
numEpochs = 15;
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);

options = trainingOptions("adam", ...
    "Shuffle","every-epoch", ...
    "MiniBatchSize",miniBatchSize, ...
    "Plots","training-progress", ...
    "Verbose",false,"MaxEpochs",numEpochs, ...
    "ValidationData",{XTest,categorical(YTest)}, ...
    "ValidationFrequency",validationFrequency);
```

To train the network, call `trainNetwork`.

```
[convNet,convNetInfo] = trainNetwork(XTrain,YTrain,layers,options);
```

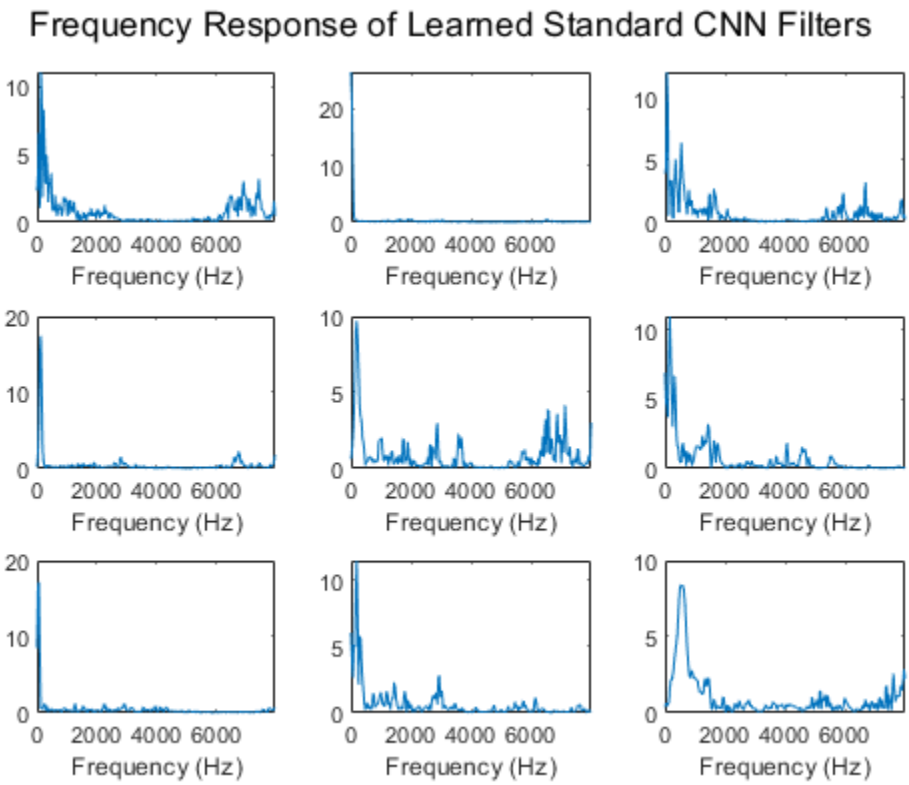


Inspect Frequency Response of First Convolutional Layer

Plot the magnitude frequency response of nine filters learned from the standard CNN network. The shape of these filters is not intuitive and does not correspond to perceptual knowledge. The next section explores the effect of using constrained filter shapes.

```
F = squeeze(convNet.Layers(2,1).Weights);
H = zeros(size(F));
Freq = zeros(size(F));
```

```
for ii = 1:size(F,2)
   [h,f] = freqz(F(:,ii),1,251,Fs);
    H(:,ii) = abs(h);
    Freq(:,ii) = f;
end
idx = linspace(1,size(F,2),9);
idx = round(idx);

figure
for jj = 1:9
   subplot(3,3,jj)
   plot(Freq(:,idx(jj)),H(:,idx(jj)))
   sgtitle("Frequency Response of Learned Standard CNN Filters")
   xlabel("Frequency (Hz)")
end
```



Constant Sinc Filterbank

In this section, you replace the first convolutional layer in the standard CNN with a constant sinc filterbank layer. The constant sinc filterbank layer convolves the input frames with a bank of fixed bandpass filters. The bandpass filters are a linear combination of two sinc filters in the time domain. The frequencies of the bandpass filters are spaced linearly on the mel scale.

Define Layers

The implementation for the constant sinc filterbank layer can be found in the `constantSincLayer.m` file (attached to this example). Define parameters for a `ConstantSincLayer`. Use 80 filters and a filter length of 251.

```
numFilters = 80;
filterLength = 251;
numChannels = 1;
name = 'constant_sinc';
```

Change the first convolutional layer from the standard CNN to the `ConstantSincLayer` and keep the other layers unchanged.

```
cSL = constantSincLayer(numFilters,filterLength,Fs,numChannels,name)
```

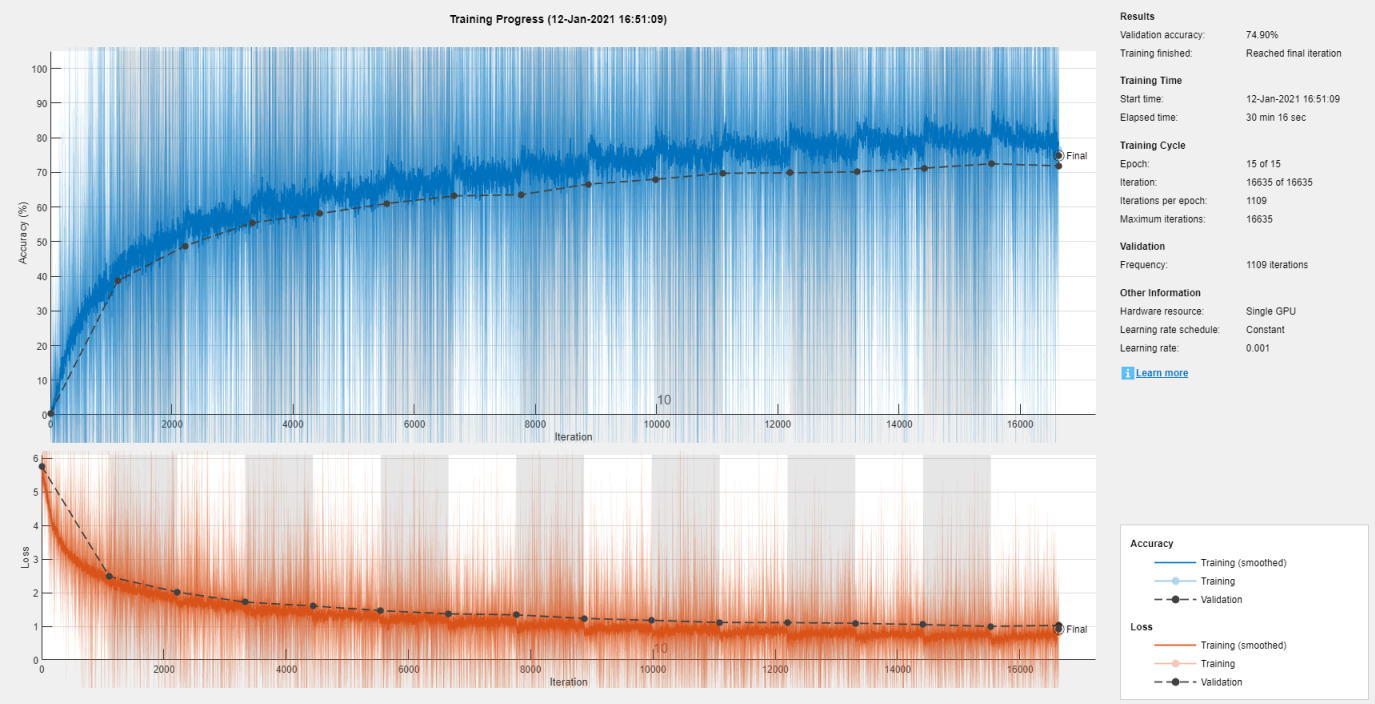
```
cSL =
  constantSincLayer with properties:

                Name: 'constant_sinc'
          NumFilters: 80
          SampleRate: 16000
        FilterLength: 251
         NumChannels: []
             Filters: [1×251×1×80 single]
    MinimumFrequency: 50
    MinimumBandwidth: 50
    StartFrequencies: [1×80 double]
          Bandwidths: [1×80 double]

  Show all properties
```

```
layers(2) = cSL;
```

Train Network

Train the network using the `trainNetwork` function. Use the same training options defined previously.

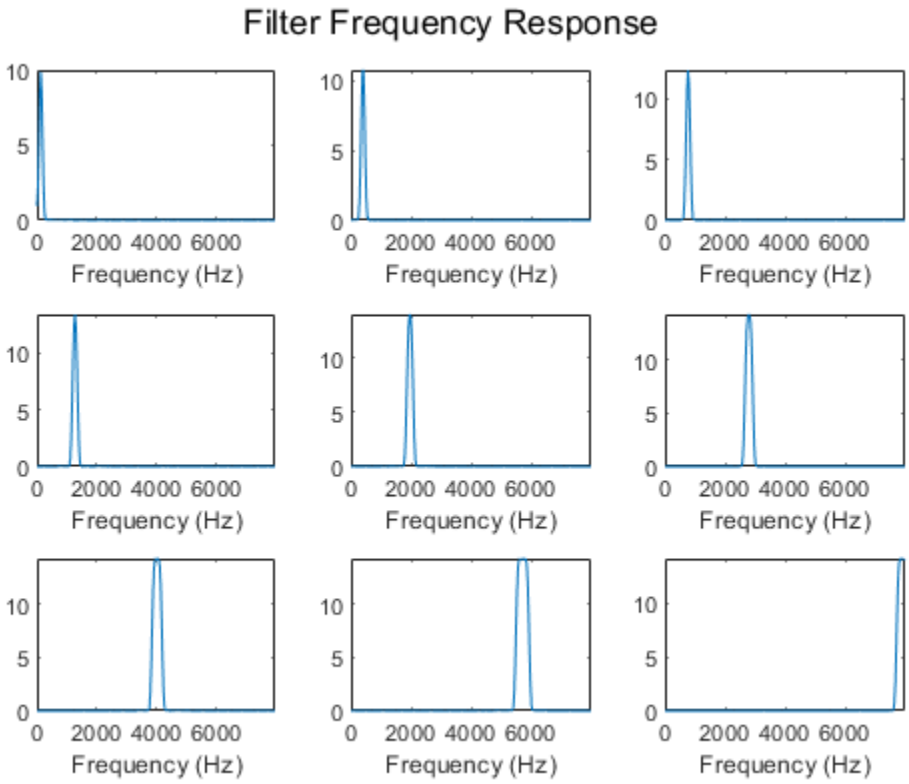
```
[constSincNet,constSincInfo] = trainNetwork(XTrain,YTrain,layers,options);
```



Inspect Frequency Response of First Convolutional Layer

The `plotNFilters` method plots the magnitude frequency response of `n` filters with equally spaced filter indices. Plot the magnitude frequency response of nine filters in the `ConstantSincLayer`.

```
figure
n = 9;
plotNFilters(constSincNet.Layers(2),n)
```



SincNet

In this section, you use a trainable SincNet layer as the first convolutional layer in your network. The SincNet layer convolves the input frames with a bank of bandpass filters. The bandwidth and the initial frequencies of the SincNet filters are initialized as equally spaced in the mel scale. The SincNet layer attempts to learn better parameters for these bandpass filters within the neural network framework.

Define Layers

The implementation for the SincNet layer filterbank layer can be found in the `sincNetLayer.m` file (attached to this example). Define parameters for a `SincNetLayer`. Use 80 filters and a filter length of 251.

```
numFilters = 80;
filterLength = 251;
numChannels = 1;
name = 'sinc';
```

Replace the `ConstantSincLayer` from the previous network with the `SincNetLayer`. This new layer has two learnable parameters: `FilterFrequencies` and `FilterBandwidths`.

```
sNL = sincNetLayer(numFilters,filterLength,Fs,numChannels,name)
```

```
sNL =
  sincNetLayer with properties:
```

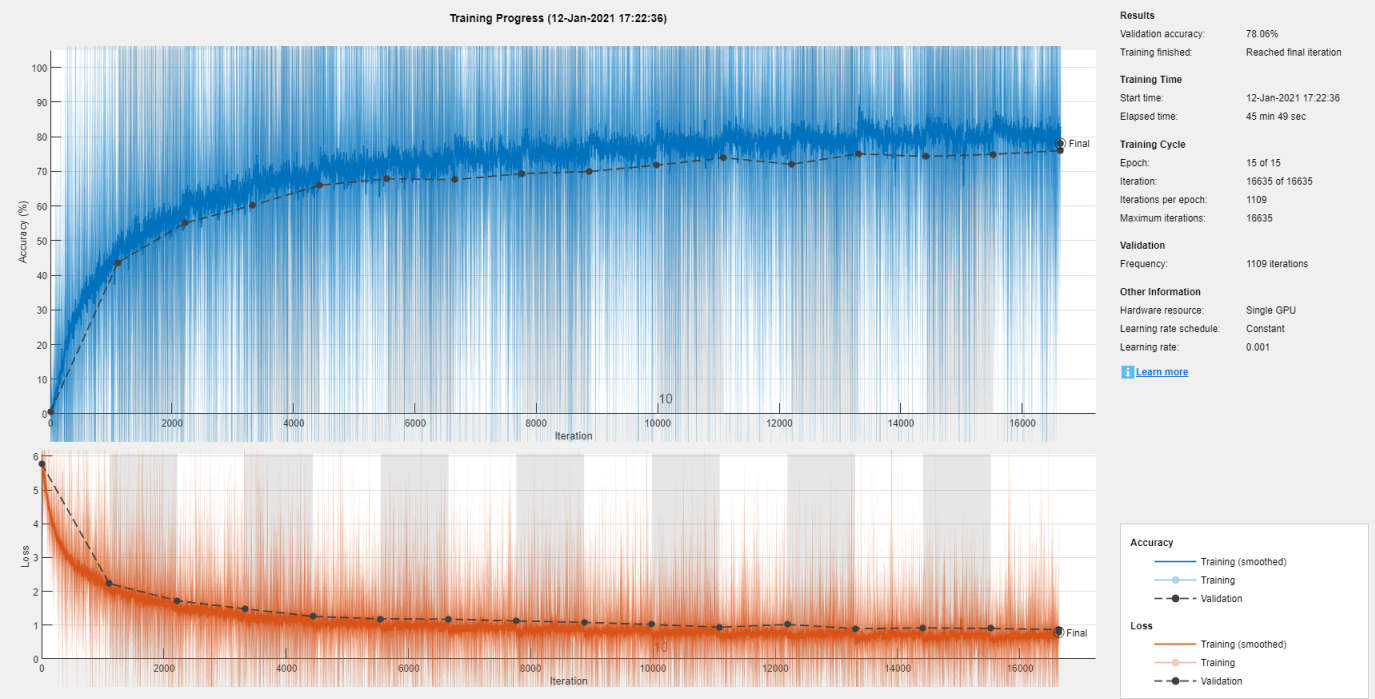
```
                Name: 'sinc'
          NumFilters: 80
          SampleRate: 16000
        FilterLength: 251
         NumChannels: []
              Window: [1×251 double]
          TimeStamps: [1×251 double]
    MinimumFrequency: 50
    MinimumBandwidth: 50

  Learnable Parameters
    FilterFrequencies: [1×80 double]
    FilterBandwidths: [1×80 double]

  Show all properties
```

```
layers(2) = sNL;
```

Train Network

Train the network using the `trainNetwork` function. Use the same training options defined previously.

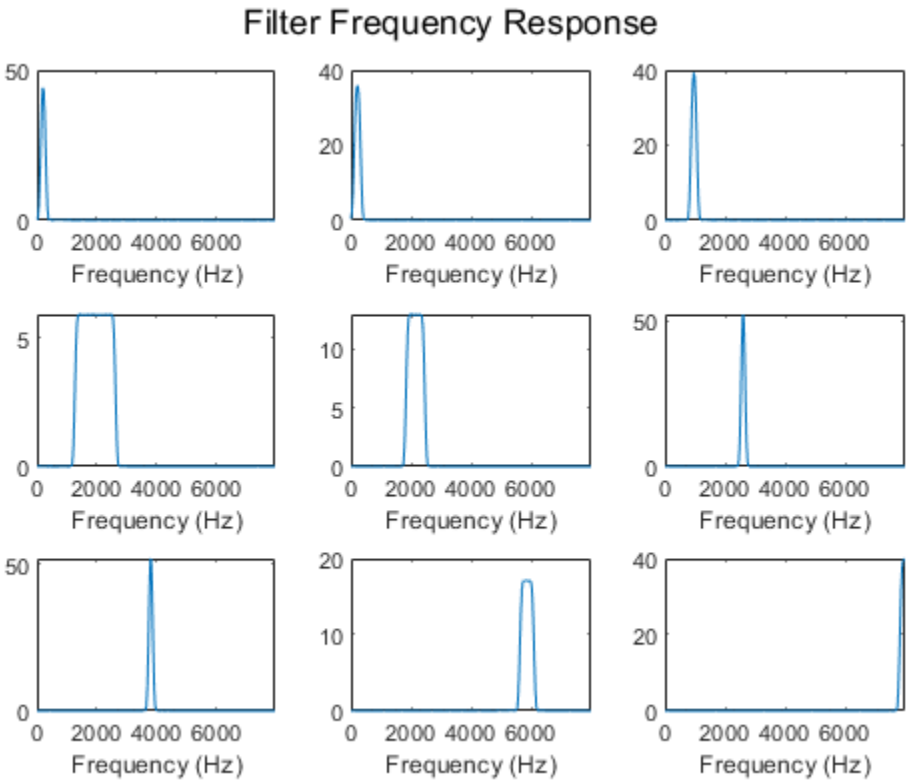
```
[sincNet,sincNetInfo] = trainNetwork(XTrain,YTrain,layers,options);
```



Inspect Frequency Response of First Convolutional Layer

Use the `plotNFilters` method of `SincNetLayer` to visualize the magnitude frequency response of nine filters with equally spaced indices learned by SincNet.

```
figure
plotNFilters(sincNet.Layers(2),9)
```



Results Summary

Accuracy

The table summarizes the frame accuracy for all three neural networks.

```
NetworkType = {'Standard CNN','Constant Sinc Layer','SincNet Layer'}';
Accuracy = [convNetInfo.FinalValidationAccuracy;constSincInfo.FinalValidationAccuracy;sincNetInfo
```

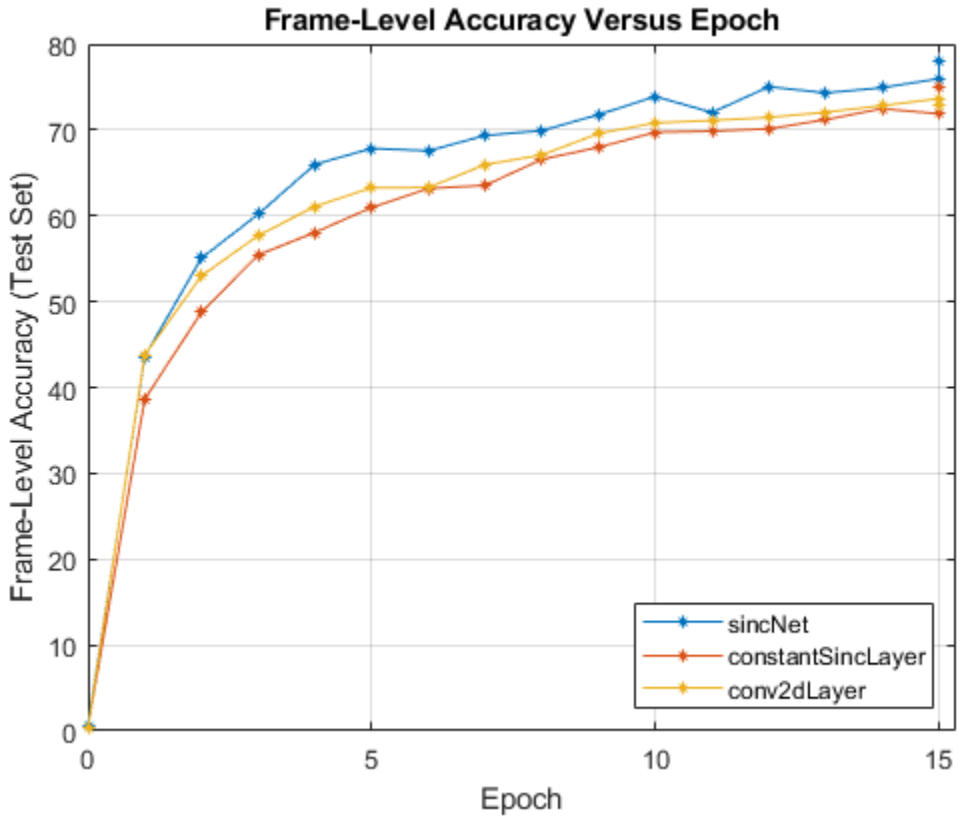
```
resultsSummary = table(NetworkType,Accuracy)
```

```
resultsSummary=3×2 table
          NetworkType          Accuracy
    ___________________________    ________

    {'Standard CNN'       }      72.97
    {'Constant Sinc Layer'}     74.902
    {'SincNet Layer'      }     78.062
```

Performance with Respect to Epochs

Plot the accuracy on the test set against the epoch number to see how well the networks learn as the number of epochs increase. SincNet outperforms the `ConstantSincLayer` network, especially during the early stages of training. This shows that updating the parameters of the bandpass filters within the neural network framework leads to faster convergence. This behavior is only observed when the dataset is large enough, so it might not be seen when `reduceDataSet` is set to `true`.

```
epoch = linspace(0,numEpochs,numel(sincNetInfo.ValidationAccuracy(~isnan(sincNetInfo.ValidationA
epoch = [epoch,numEpochs];

sinc_valAcc = [sincNetInfo.ValidationAccuracy(~isnan(sincNetInfo.ValidationAccuracy)),...
    sincNetInfo.FinalValidationAccuracy];
const_sinc_valAcc = [constSincInfo.ValidationAccuracy(~isnan(constSincInfo.ValidationAccuracy)),
    constSincInfo.FinalValidationAccuracy];
conv_valAcc = [convNetInfo.ValidationAccuracy(~isnan(convNetInfo.ValidationAccuracy)),...
    convNetInfo.FinalValidationAccuracy];

figure
plot(epoch,sinc_valAcc,'-*','MarkerSize',4)
hold on
plot(epoch,const_sinc_valAcc,'-*','MarkerSize',4)
plot(epoch,conv_valAcc,'-*','MarkerSize',4)
ylabel('Frame-Level Accuracy (Test Set)')
xlabel('Epoch')
xlim([0 numEpochs+0.3])
title('Frame-Level Accuracy Versus Epoch')
legend("sincNet","constantSincLayer","conv2dLayer","Location","southeast")
grid on
```



In the figure above, the final frame accuracy is a bit different from the frame accuracy that is computed in the last iteration. While training, the batch normalization layers perform normalization over mini-batches. However, at the end of training, the batch normalization layers normalize over the entire training data, which results in a slight change in performance.

Supporting Functions

```
function [X,Y] = preprocessAudioData(ADS,SL,OL,Fs)

    if ~isempty(ver('parallel'))
        pool = gcp;
        numPar = numpartitions(ADS,pool);
    else
        numPar = 1;
    end

    parfor ii = 1:numPar

        X = zeros(1,SL,1,0);
        Y = zeros(0);
        subADS = partition(ADS,numPar,ii);

        while hasdata(subADS)
            [audioIn,dsInfo] = read(subADS);

            speechIdx = detectSpeech(audioIn,Fs);
            numChunks = size(speechIdx,1);
            audioData = zeros(1,SL,1,0);

            for chunk = 1:numChunks
                % Remove trail end audio
                audio_chunk = audioIn(speechIdx(chunk,1):speechIdx(chunk,2));
                audio_chunk = buffer(audio_chunk,SL,OL);
                q = size(audio_chunk,2);

                % Split audio into 200 ms chunks
                audio_chunk = reshape(audio_chunk,1,SL,1,q);

                % Concatenate with existing audio
                audioData = cat(4,audioData,audio_chunk);
            end

            audioLabel = str2double(dsInfo.Label{1});

            % Generate labels for training and testing by replecating matrix
            audioLabelsTrain = repmat(audioLabel,1,size(audioData,4));

            % Add data points for current speaker to existing data
            X = cat(4,X,audioData);
            Y = cat(2,Y,audioLabelsTrain);
        end

        XC{ii} = X;
        YC{ii} = Y;
    end

    X = cat(4,XC{:});
    Y = cat(2,YC{:});

    Y = categorical(Y);

end
```

References

[1] M. Ravanelli and Y. Bengio, "Speaker Recognition from Raw Waveform with SincNet," *2018 IEEE Spoken Language Technology Workshop (SLT)*, Athens, Greece, 2018, pp. 1021-1028, doi: 10.1109/SLT.2018.8639585.

[2] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Brisbane, QLD, 2015, pp. 5206-5210, doi: 10.1109/ICASSP.2015.7178964

Dereverberate Speech Using Deep Learning Networks

This example shows how to train a U-Net fully convolutional network (FCN) [1] on page 14-0 to dereverberate a speech signals.

Introduction

Reverberation occurs when a speech signal is reflected off objects in space, causing multiple reflections to build up and eventually leads to degradation of speech quality. Dereverberation is the process of reducing the reverberation effects in a speech signal.

Dereverberate Speech Signal Using Pretrained Network

Before going into the training process in detail, use a pretrained network to dereverberate a speech signal.

Download the pretrained network. This network was trained on 56-speaker versions of the training datasets. The example walks through training on the 28-speaker version.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/dereverbnet.zip';
downloadFolder = tempdir;
networkDataFolder = fullfile(downloadFolder,'derevernet');

if ~exist(networkDataFolder,'dir')
    disp('Downloading pretrained network ...')
    unzip(url,downloadFolder)
end
load(fullfile(networkDataFolder,'dereverbNet.mat'))
```

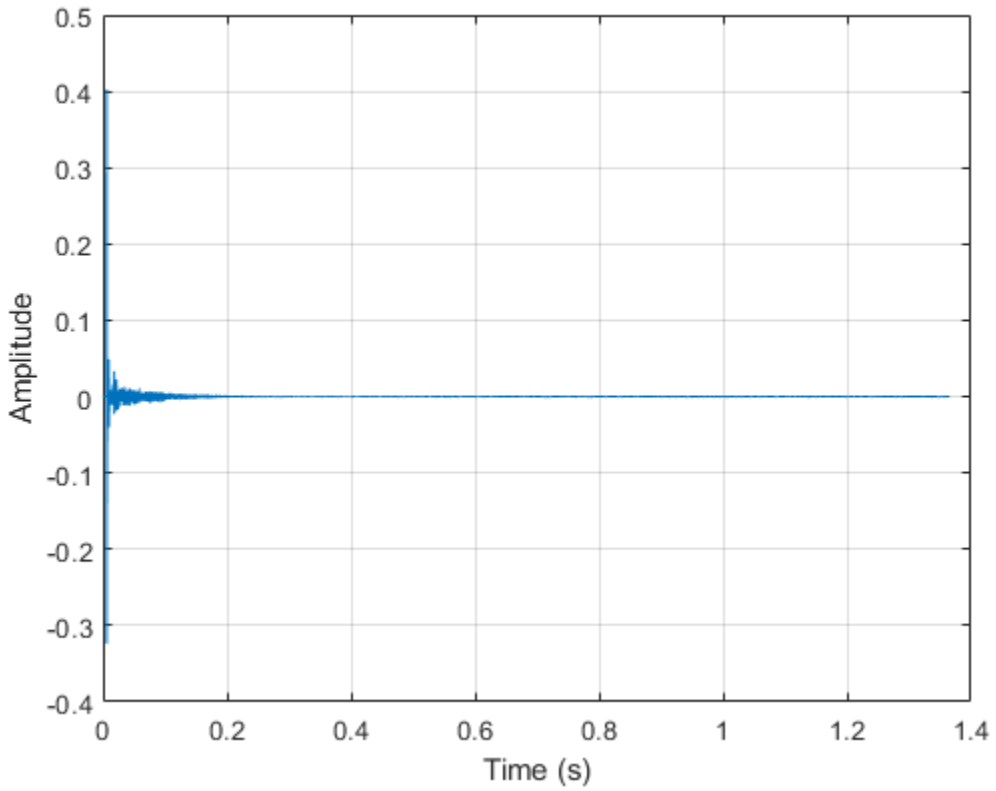
Listen to a clean speech signal sampled at 16 kHz.

```
[cleanAudio,fs] = audioread('clean_speech_signal.wav');

sound(cleanAudio,fs)
```

An acoustic path can be modelled using a room impulse response. You can model reverberation by convolving an anechoic signal with a room impulse response.

Load and plot a room impulse response.

```
[rirAudio,fsR] = audioread('room_impulse_response.wav');

tAxis = (1/fsR)*(0:numel(rirAudio)-1);

figure
plot(tAxis,rirAudio)
xlabel('Time (s)')
ylabel('Amplitude')
grid on
```



Convolve the clean speech with the room impulse response to obtain reverberated speech. Align the lengths and amplitudes of the reverberated and clean speech signals.

```
revAudio = conv(cleanAudio,rirAudio);

revAudio = revAudio(1:numel(cleanAudio));
revAudio = revAudio.*(max(abs(cleanAudio))/max(abs(revAudio)));
```

Listen to the reverberated speech signal.

```
sound(revAudio,fs)
```

The input to the pretrained network is the log-magnitude short-time Fourier transform (STFT) of the reverberant audio. The network predicts the log-magnitude STFT of the dereverberated input. To estimate the original time-domain audio signal, you perform an inverse STFT and assume the phase of the reverberant audio.

Use the following parameters to compute the STFT.

```
params.WindowdowLength = 512;
params.Window = hamming(params.WindowdowLength,"periodic");
params.OverlapLength = round(0.75*params.WindowdowLength);
params.FFTLength = params.WindowdowLength;
```

Use `stft` to compute the one-sided log-magnitude STFT. Use single precision when computing features to better utilize memory usage and to speed up the training. Even though the one-sided STFT yields 257 frequency bins, consider only 256 bins and ignore the highest frequency bin.

```
revAudio = single(revAudio);
audioSTFT = stft(revAudio,'Window',params.Window,'OverlapLength',params.OverlapLength, ...
                 'FFTLength',params.FFTLength,'FrequencyRange','onesided');
Eps = realmin('single');
reverbFeats = log(abs(audioSTFT(1:end-1,:)) + Eps);
```

Extract the phase of the STFT.

```
phaseOriginal = angle(audioSTFT(1:end-1,:));
```

Each input will have dimensions 256-by-256 (frequency bins by time steps). Split the log-magnitude STFT into segments of 256 time-steps.

```
params.NumSegments = 256;
params.NumFeatures = 256;
totalFrames = size(reverbFeats,2);
chunks = ceil(totalFrames/params.NumSegments);
reverbSTFTSegments = mat2cell(reverbFeats,params.NumFeatures, ...
    [params.NumSegments*ones(1,chunks - 1),(totalFrames - (chunks-1)*params.NumSegments)]);
reverbSTFTSegments{chunks} = reverbFeats(:,end-params.NumSegments + 1:end);
```

Scale the segmented features to the range [-1,1]. Retain the minimum and maximum values used to scale for reconstructing the dereverberated signal.

```
minVals = num2cell(cellfun(@(x)min(x,[],'all'),reverbSTFTSegments));
maxVals = num2cell(cellfun(@(x)max(x,[],'all'),reverbSTFTSegments));

featNorm = cellfun(@(feat,minFeat,maxFeat)2.*(feat - minFeat)./(maxFeat - minFeat) - 1, ...
    reverbSTFTSegments,minVals,maxVals,'UniformOutput',false);
```

Reshape the features so that chunks are along the fourth dimension.

```
featNorm = reshape(cell2mat(featNorm),params.NumFeatures,params.NumSegments,1,chunks);
```

Predict the log-magnitude spectra of the reverberated speech signal using the pretrained network.

```
predictedSTFT4D = predict(dereverbNet,featNorm);
```

Reshape to 3-dimensions and scale the predicted STFTs to the original range using the saved minimum-maximum pairs.

```
predictedSTFT = squeeze(mat2cell(predictedSTFT4D,params.NumFeatures,params.NumSegments,1,ones(1,c
featDeNorm = cellfun(@(feat,minFeat,maxFeat) (feat + 1).*(maxFeat-minFeat)./2 + minFeat, ...
    predictedSTFT,minVals,maxVals,'UniformOutput',false);
```

Reverse the log-scaling.

```
predictedSTFT = cellfun(@exp,featDeNorm,'UniformOutput',false);
```

Concatenate the predicted 256-by-256 magnitude STFT segments to obtain the magnitude spectrogram of original length.

```
predictedSTFTAll = predictedSTFT(1:chunks - 1);
predictedSTFTAll = cat(2,predictedSTFTAll{:});
predictedSTFTAll(:,totalFrames - params.NumSegments + 1:totalFrames) = predictedSTFT{chunks};
```

Before taking the inverse STFT, append zeros to the predicted log-magnitude spectrum and the phase in lieu of the highest frequency bin which was excluded when preparing input features.

```
nCount = size(predictedSTFTAll,3);
predictedSTFTAll = cat(1,predictedSTFTAll,zeros(1,totalFrames,nCount));
phase = cat(1,phaseOriginal,zeros(1,totalFrames,nCount));
```

Use the inverse STFT function to reconstruct the dereverberated time-domain speech signal using the predicted log-magnitude STFT and the phase of reverberant speech signal.

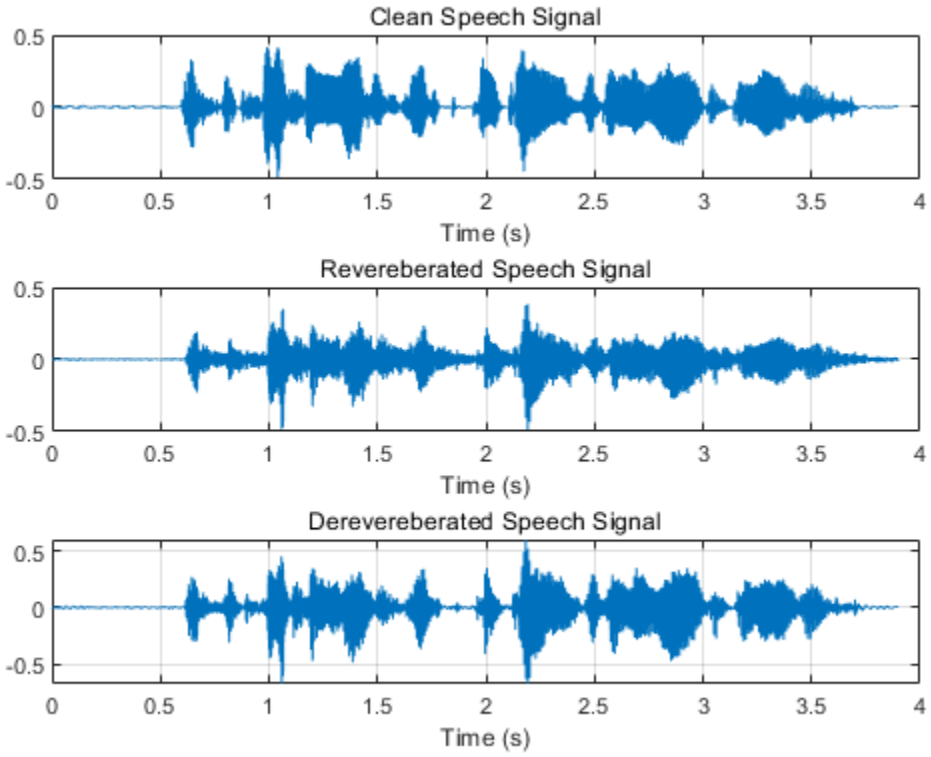
```
oneSidedSTFT = predictedSTFTAll.*exp(1j*phase);
dereverbedAudio = istft(oneSidedSTFT, ...
    'Window',params.Window,'OverlapLength',params.OverlapLength, ...
    'FFTLength',params.FFTLength,'ConjugateSymmetric',true, ...
    'FrequencyRange','onesided');

dereverbedAudio = dereverbedAudio./max(abs([dereverbedAudio;revAudio]));
dereverbedAudio = [dereverbedAudio;zeros(length(revAudio) - numel(dereverbedAudio), 1)];
```

Listen to the dereverberated audio signal.

```
sound(dereverbedAudio,fs)
```

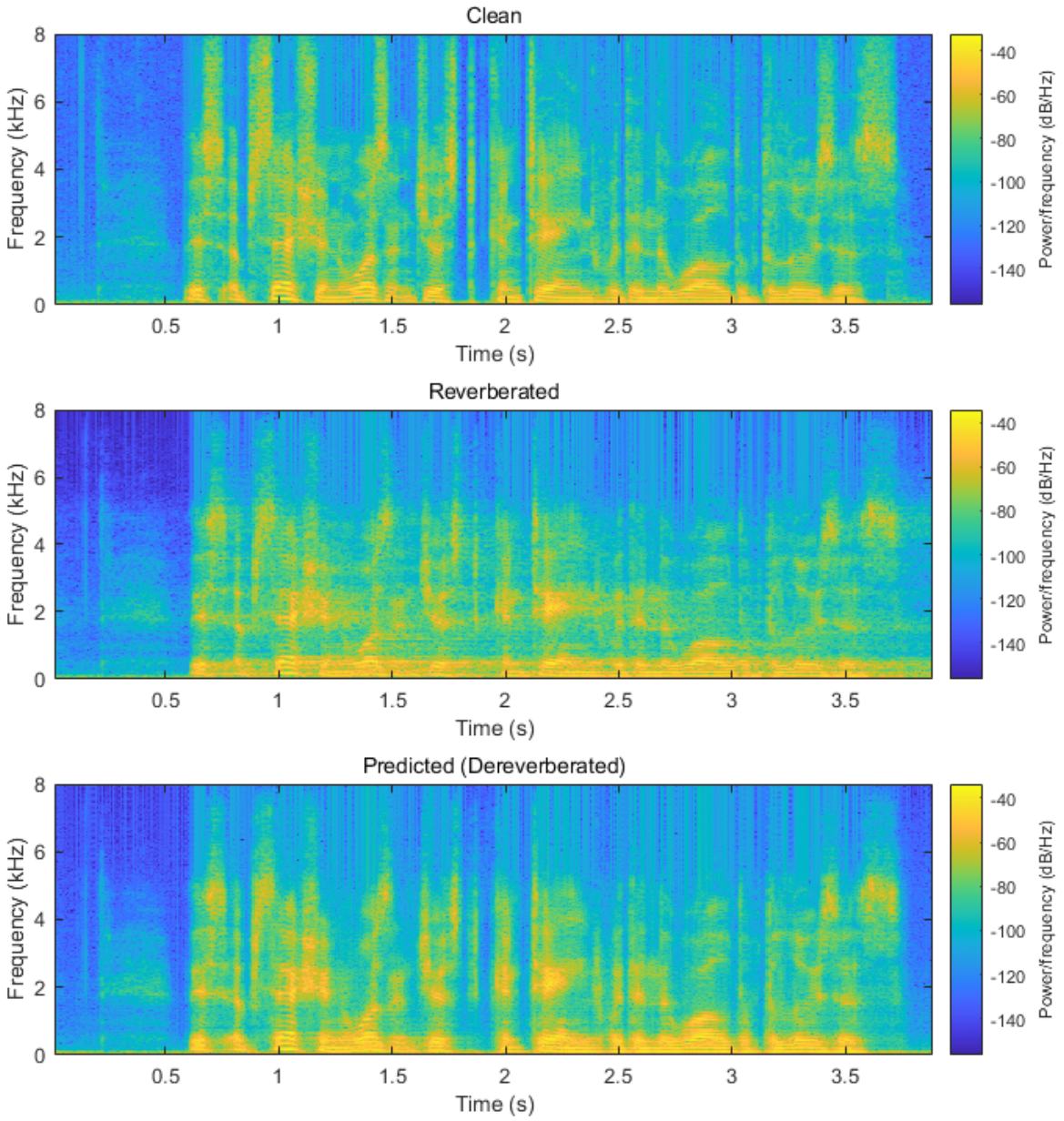
Plot the clean, reverberant, and dereverberated speech signals.

```
t = (1/fs)*(0:numel(cleanAudio)-1);

figure

subplot(3,1,1)
plot(t,cleanAudio)
xlabel('Time (s)')
grid on
subtitle('Clean Speech Signal')

subplot(3,1,2)
plot(t,revAudio)
xlabel('Time (s)')
grid on
subtitle('Revereberated Speech Signal')

subplot(3,1,3)
plot(t,dereverbedAudio)
xlabel('Time (s)')
grid on
subtitle('Derevereberated Speech Signal')
```



Visualize the spectrograms of the clean, reverberant, and dereverberated speech signals.

```
figure('Position',[100,100,800,800])

subplot(3,1,1)
spectrogram(cleanAudio,params.Window,params.OverlapLength,params.FFTLength,fs,'yaxis');
subtitle('Clean')

subplot(3,1,2)
spectrogram(revAudio,params.Window,params.OverlapLength,params.FFTLength,fs,'yaxis');
subtitle('Reverberated')

subplot(3,1,3)
spectrogram(dereverbedAudio,params.Window,params.OverlapLength,params.FFTLength,fs,'yaxis');
subtitle('Predicted (Dereverberated)')
```



Download the Dataset

This example uses the Reverberant Speech Database [2] on page 14-0 and the corresponding Clean Speech Database [3] on page 14-0 to train the network.

Download the clean speech data set.

```
url1 = 'https://datashare.is.ed.ac.uk/bitstream/handle/10283/2791/clean_trainset_28spk_wav.zip';
url2 = 'https://datashare.is.ed.ac.uk/bitstream/handle/10283/2791/clean_testset_wav.zip';
```

```
downloadFolder = tempdir;
cleanDataFolder = fullfile(downloadFolder,'DS_10283_2791');

if ~exist(cleanDataFolder,'dir')
    disp('Downloading data set (6 GB) ...')
    unzip(url1,cleanDataFolder)
    unzip(url2,cleanDataFolder)
end
```

Download the reverberated speech dataset.

```
url3 = 'https://datashare.is.ed.ac.uk/bitstream/handle/10283/2031/reverb_trainset_28spk_wav.zip'
url4 = 'https://datashare.is.ed.ac.uk/bitstream/handle/10283/2031/reverb_testset_wav.zip';
downloadFolder = tempdir;
reverbDataFolder = fullfile(downloadFolder,'DS_10283_2031');

if ~exist(reverbDataFolder,'dir')
    disp('Downloading data set (6 GB) ...')
    unzip(url3,reverbDataFolder)
    unzip(url4,reverbDataFolder)
end
```

Data Preprocessing and Feature Extraction

Once the data is downloaded, preprocess the downloaded data and extract features before training the DNN model:

1. Synthetically generate reverberant data using the `reverberator` object
2. Split each speech signal into small segments of 2.072s duration
3. Discard segments which contain significant silent regions
4. Extract log-magnitude STFTs as predictor and target features
5. Scale and reshape features

First, create two `audioDatastore` objects that point to the clean and reverberant speech datasets.

```
adsCleanTrain = audioDatastore(fullfile(cleanDataFolder,'clean_trainset_28spk_wav'),'IncludeSubf
adsReverbTrain = audioDatastore(fullfile(reverbDataFolder,'reverb_trainset_28spk_wav'),'IncludeSu
```

Synthetic Reverberant Speech Data Generation

The amount of reverberation in the original data is relatively small. You will augment the reverberant speech data with significant reverberation effects using the `reverberator` object.

Create an `audioDatastore` that points to the clean speech dataset allocated for synthetic reverberant data generation.

```
adsSyntheticCleanTrain = subset(adsCleanTrain,10e3+1:length(adsCleanTrain.Files));
adsCleanTrain = subset(adsCleanTrain,1:10e3);
adsReverbTrain = subset(adsReverbTrain,1:10e3);
```

Resample from 48 kHz to 16 kHz.

```
adsSyntheticCleanTrain = transform(adsSyntheticCleanTrain,@(x)resample(x,16e3,48e3));
adsCleanTrain = transform(adsCleanTrain,@(x)resample(x,16e3,48e3));
adsReverbTrain = transform(adsReverbTrain,@(x)resample(x,16e3,48e3));
```

Combine the two audio datastores, maintaining the correspondence between the clean and reverberant speech samples.

```
adsCombinedTrain = combine(adsCleanTrain,adsReverbTrain);
```

The applyReverb on page 14-0 function creates a `reverberator` object, updates the pre delay, decay factor, and wet-dry mix parameters as specified, and then applies reverberation. Use `audioDataAugmenter` to create synthetically generated reverberant data.

```
augmenter = audioDataAugmenter('AugmentationMode','independent','NumAugmentations', 1,'ApplyAddNo
    'ApplyTimeStretch',0,'ApplyPitchShift',0,'ApplyVolumeControl',0,'ApplyTimeShift',0);
algorithmHandle = @(y,preDelay,decayFactor,wetDryMix,samplingRate) ...
    applyReverb(y,preDelay,decayFactor,wetDryMix,samplingRate);

addAugmentationMethod(augmenter,'Reverb',algorithmHandle, ...
    'AugmentationParameter',{'PreDelay','DecayFactor','WetDryMix','SamplingRate'}, ...
    'ParameterRange',{[0.15,0.25],[0.2,0.5],[0.3,0.45],[16000,16000]})

augmenter.ReverbProbability = 1;
disp(augmenter)
```

```
  audioDataAugmenter with properties:

              AugmentationMode: 'independent'
    AugmentationParameterSource: 'random'
              NumAugmentations: 1
              ApplyTimeStretch: 0
               ApplyPitchShift: 0
            ApplyVolumeControl: 0
                ApplyAddNoise: 0
               ApplyTimeShift: 0
                  ApplyReverb: 1
                PreDelayRange: [0.1500 0.2500]
              DecayFactorRange: [0.2000 0.5000]
                WetDryMixRange: [0.3000 0.4500]
             SamplingRateRange: [16000 16000]
```

Create a new `audioDatastore` corresponding to synthetically generated reverberant data by calling `transform` to apply data augmentation.

```
adsSyntheticReverbTrain = transform(adsSyntheticCleanTrain,@(y)deal(augment(augmenter,y,16e3).Au
```

Combine the two audio datastores.

```
adsSyntheticCombinedTrain = combine(adsSyntheticCleanTrain,adsSyntheticReverbTrain);
```

Next, based on the dimensions of the input features to the network, segment the audio into chunks of 2.072 s duration with an overlap of 50%.

Having too many silent segments can adversely affect the DNN model training. Remove the segments which are mostly silent (more than 50% of the duration) and exclude those from the model training. Do not completely remove silence because the model will not be robust to silent regions and slight reverberation effects could be identified as silence. `detectSpeech` can identify the start and end points of silent regions. After these two steps, the feature extraction process can be carried out as explained in the first section. helperFeatureExtract on page 14-0 implements these steps.

Define the feature extraction parameters. By setting `reduceDataSet` to true, you choose a small subset of the datasets to perform the subsequent steps.

```
reduceDataSet = true;
params.fs = 16000;
params.WindowdowLength = 512;
params.Window = hamming(params.WindowdowLength,"periodic");
params.OverlapLength = round(0.75*params.WindowdowLength);
params.FFTLength = params.WindowdowLength;
samplesPerMs = params.fs/1000;
params.samplesPerImage = (24+256*8)*samplesPerMs;
params.shiftImage = params.samplesPerImage/2;
params.NumSegments = 256;
params.NumFeatures = 256
```

```
params = struct with fields:
    WindowdowLength: 512
             Window: [512×1 double]
      OverlapLength: 384
          FFTLength: 512
        NumSegments: 256
        NumFeatures: 256
                 fs: 16000
    samplesPerImage: 33152
         shiftImage: 16576
```

To speed up processing, distribute the preprocessing and feature extraction task across multiple workers using `parfor`.

Determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(adsCombinedTrain,pool);
else
    numPar = 1;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

For each partition, read from the datastore, preprocess the audio signal, and then extract the features.

```
if reduceDataSet
    adsCombinedTrain = shuffle(adsCombinedTrain); %#ok
    adsCombinedTrain = subset(adsCombinedTrain,1:200);

    adsSyntheticCombinedTrain = shuffle(adsSyntheticCombinedTrain);
    adsSyntheticCombinedTrain = subset(adsSyntheticCombinedTrain,1:200);
end

allCleanFeatures = cell(1,numPar);
allReverbFeatures = cell(1,numPar);

parfor iPartition = 1:numPar
    combinedPartition = partition(adsCombinedTrain,numPar,iPartition);
    combinedSyntheticPartition = partition(adsSyntheticCombinedTrain,numPar,iPartition);
```

```
    cPartitionSize = numel(combinedPartition.UnderlyingDatastores{1}.UnderlyingDatastores{1}.File
    cSyntheticPartitionSize = numel(combinedSyntheticPartition.UnderlyingDatastores{1}.Underlying
    partitionSize = cPartitionSize + cSyntheticPartitionSize;

    cleanFeaturesPartition = cell(1,partitionSize);
    reverbFeaturesPartition = cell(1,partitionSize);

    for idx = 1:partitionSize
        if idx <= cPartitionSize
            audios = read(combinedPartition);
        else
            audios = read(combinedSyntheticPartition);
        end
        cleanAudio = single(audios(:,1));
        reverbAudio = single(audios(:,2));
        [featuresClean,featuresReverb] = helperFeatureExtract(cleanAudio,reverbAudio,false,params
        cleanFeaturesPartition{idx} = featuresClean;
        reverbFeaturesPartition{idx} = featuresReverb;
    end
    allCleanFeatures{iPartition} = cat(2,cleanFeaturesPartition{:});
    allReverbFeatures{iPartition} = cat(2,reverbFeaturesPartition{:});
end

allCleanFeatures = cat(2,allCleanFeatures{:});
allReverbFeatures = cat(2,allReverbFeatures{:});
```

Normalize the extracted features to the range [-1,1] and then reshape as explained in the first section, using the featureNormalizeAndReshape on page 14-0 function.

```
trainClean = featureNormalizeAndReshape(allCleanFeatures);
trainReverb = featureNormalizeAndReshape(allReverbFeatures);
```

Now that you have extracted the log-magnitude STFT features from the training datasets, follow the same procedure to extract features from the validation datasets. For reconstruction purposes, retain the phase of the reverberant speech samples of the validation dataset. In addition, retain the audio data for both the clean and reverberant speech samples in the validation set to use in the evaluation process (next section).

```
adsCleanVal = audioDatastore(fullfile(cleanDataFolder,'clean_testset_wav'),'IncludeSubfolders',t
adsReverbVal = audioDatastore(fullfile(reverbDataFolder,'reverb_testset_wav'),'IncludeSubfolders
```

Resample from 48 kHz to 16 kHz.

```
adsCleanVal = transform(adsCleanVal,@(x)resample(x,16e3,48e3));
adsReverbVal = transform(adsReverbVal,@(x)resample(x,16e3,48e3));

adsCombinedVal = combine(adsCleanVal,adsReverbVal);

if reduceDataSet
    adsCombinedVal = shuffle(adsCombinedVal);%#ok
    adsCombinedVal = subset(adsCombinedVal,1:50);
end

allValCleanFeatures = cell(1,numPar);
allValReverbFeatures = cell(1,numPar);
allValReverbPhase = cell(1,numPar);
allValCleanAudios = cell(1,numPar);
```

```
allValReverbAudios = cell(1,numPar);

parfor iPartition = 1:numPar
    combinedPartition = partition(adsCombinedVal,numPar,iPartition);

    partitionSize = numel(combinedPartition.UnderlyingDatastores{1}.UnderlyingDatastores{1}.Files

    cleanFeaturesPartition = cell(1,partitionSize);
    reverbFeaturesPartition = cell(1,partitionSize);
    reverbPhasePartition = cell(1,partitionSize);
    cleanAudiosPartition = cell(1,partitionSize);
    reverbAudiosPartition = cell(1,partitionSize);

    for idx = 1:partitionSize
        audios = read(combinedPartition);

        cleanAudio = single(audios(:,1));
        reverbAudio = single(audios(:,2));

        [a,b,c,d,e] = helperFeatureExtract(cleanAudio,reverbAudio,true,params);

        cleanFeaturesPartition{idx} = a;
        reverbFeaturesPartition{idx} = b;
        reverbPhasePartition{idx} = c;
        cleanAudiosPartition{idx} = d;
        reverbAudiosPartition{idx} = e;
    end
    allValCleanFeatures{iPartition} = cat(2,cleanFeaturesPartition{:});
    allValReverbFeatures{iPartition} = cat(2,reverbFeaturesPartition{:});
    allValReverbPhase{iPartition} = cat(2,reverbPhasePartition{:});
    allValCleanAudios{iPartition} = cat(2,cleanAudiosPartition{:});
    allValReverbAudios{iPartition} = cat(2,reverbAudiosPartition{:});
end

allValCleanFeatures = cat(2,allValCleanFeatures{:});
allValReverbFeatures = cat(2,allValReverbFeatures{:});
allValReverbPhase = cat(2,allValReverbPhase{:});
allValCleanAudios = cat(2,allValCleanAudios{:});
allValReverbAudios = cat(2,allValReverbAudios{:});

valClean = featureNormalizeAndReshape(allValCleanFeatures);
```

Retain the minimum and maximum values of each feature of the reverberant validation set. You will use these values in the reconstruction process.

```
[valReverb,valMinMaxPairs] = featureNormalizeAndReshape(allValReverbFeatures);
```

Define Neural Network Architecture

A fully convolutional network architecture named **U-Net** was adapted for this speech dereverberation task as proposed in [1] on page 14-0 . "U-Net" is an encoder-decoder network with skip connections. In the U-Net model, each layer downsamples its input (stride of 2) until a bottleneck layer is reached (encoding path). In subsequent layers, the input is upsampled by each layer until the output is returned to the original shape (decoding path). To minimize the loss of low-level information during the downsampling process, connections are made between the mirrored layers by directly concatenating outputs of corresponding layers (*skip connections*).

Define the network architecture and return the layer graph with connections.

```
params.WindowdowLength = 512;
params.FFTLength = params.WindowdowLength;
params.NumFeatures = params.FFTLength/2;
params.NumSegments = 256;

filterH = 6;
filterW = 6;
numChannels = 1;
nFilters = [64,128,256,512,512,512,512,512];

inputLayer = imageInputLayer([params.NumFeatures,params.NumSegments,numChannels], ...
    'Normalization','none','Name','input');
layers = inputLayer;

% U-Net squeezing path
layers = [layers;
    convolution2dLayer([filterH,filterW],nFilters(1),'Stride',2,'Padding','same','Name',"conv"+s
    leakyReluLayer(0.2,'Name',"leaky-relu"+string(1))];

for i = 2:8
    layers =  [layers;
        convolution2dLayer([filterH,filterW],nFilters(i),'Stride',2,'Padding','same','Name',"conv
        batchNormalizationLayer('Name',"batchnorm"+string(i))];%#ok
    if i ~= 8
        layers = [layers;leakyReluLayer(0.2,'Name',"leaky-relu"+string(i))];%#ok
    else
        layers = [layers;reluLayer('Name',"relu"+string(i))];%#ok
    end
end

% U-Net expanding path
for i = 7:-1:0
    nChannels = numChannels;
    if i > 0
        nChannels = nFilters(i);
    end
    layers = [layers;
        transposedConv2dLayer([filterH,filterW],nChannels,'Stride',2,'Cropping','same','Name',"de
    if i > 0
        layers = [layers; batchNormalizationLayer('Name',"de-batchnorm" +string(i))];%#ok
    end
    if i > 4
        layers = [layers;dropoutLayer(0.5,'Name',"de-dropout"+string(i))];%#ok
    end
    if i > 0
        layers = [layers;
            reluLayer('Name',"de-relu"+string(i));
            concatenationLayer(3,2,'Name',"concat"+string(i))];%#ok
    else
        layers = [layers;tanhLayer('Name',"de-tanh"+string(i))];%#ok
    end
end

layers = [layers;regressionLayer('Name','output')];

unetLayerGraph = layerGraph(layers);

% Define skip-connections
```

```
for i = 1:7
    unetLayerGraph = connectLayers(unetLayerGraph,'leaky-relu'+string(i),'concat'+string(i)+'/in2');
end
```

Use `analyzeNetwork` to view the model architecture. This is a good way to visualize the connections between layers.

```
analyzeNetwork(unetLayerGraph);
```

Train the Network

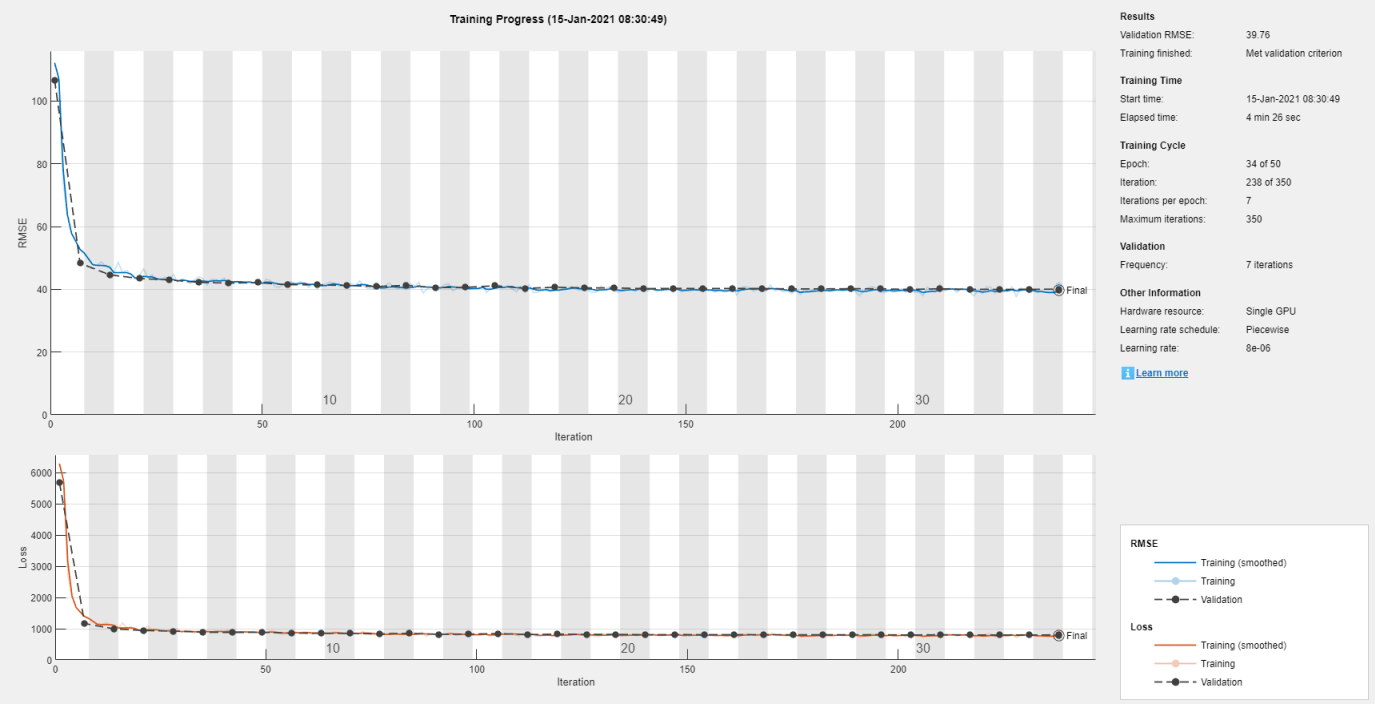
You will use the mean squared error (MSE) between the log-magnitude spectra of the dereverberated speech sample (output of the model) and the corresponding clean speech sample (target) as the loss function. Use the `adam` optimizer and a mini-batch size of 128 for the training. Allow the model to train for a maximum of 50 epochs. If the validation loss doesn't improve for 5 consecutive epochs, terminate the training process. Reduce the learning rate by a factor of 10 every 15 epochs.

Define the training options as below. Change the execution environment and whether to perform background dispatching depending on your hardware availability and whether you have access to Parallel Computing Toolbox™.

```
initialLearnRate = 8e-4;
miniBatchSize = 64;

options = trainingOptions("adam", ...
        "MaxEpochs", 50, ...
        "InitialLearnRate",initialLearnRate, ...
        "MiniBatchSize",miniBatchSize, ...
        "Shuffle","every-epoch", ...
        "Plots","training-progress", ...
        "Verbose",false, ...
        "ValidationFrequency",max(1,floor(size(trainReverb,4)/miniBatchSize)), ...
        "ValidationPatience",5, ...
        "LearnRateSchedule","piecewise", ...
        "LearnRateDropFactor",0.1, ...
        "LearnRateDropPeriod",15, ...
        "ExecutionEnvironment","gpu", ...
        "DispatchInBackground",true, ...
        "ValidationData",{valReverb,valClean});
```

Train the network.

```
dereverbNet = trainNetwork(trainReverb,trainClean,unetLayerGraph,options);
```



Evaluate Network Performance

Prediction and Reconstruction

Predict the log-magnitude spectra of the validation set.

```
predictedSTFT4D = predict(dereverbNet,valReverb);
```

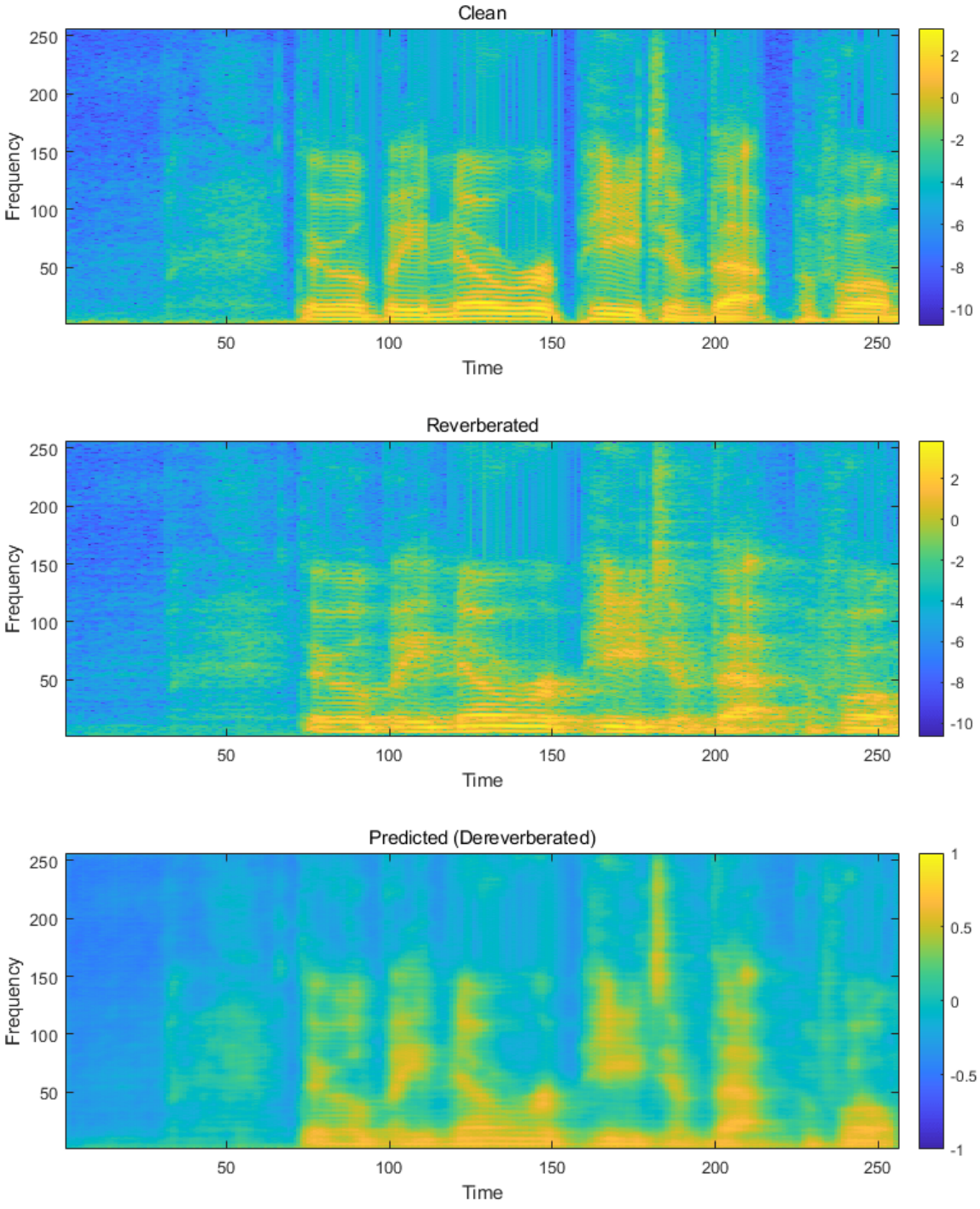
Use the helperReconstructPredictedAudios on page 14-0 function to reconstruct the predicted speech. This function performs actions outlined in the first section.

```
params.WindowdowLength = 512;
params.Window = hamming(params.WindowdowLength,"periodic");
params.OverlapLength = round(0.75*params.WindowdowLength);
params.FFTLength = params.WindowdowLength;
params.fs = 16000;

dereverbedAudioAll = helperReconstructPredictedAudios(predictedSTFT4D,valMinMaxPairs,allValReverb
```

Visualize the log-magnitude STFTs of the clean, reverberant, and corresponding dereverberated speech signals.

```
figure('Position',[100,100,1024,1200])

subplot(3,1,1)
imagesc(squeeze(allValCleanFeatures{1}))
set(gca,'Ydir','normal')
subtitle('Clean')
xlabel('Time')
ylabel('Frequency')
colorbar
```

```
subplot(3,1,2)
imagesc(squeeze(allValReverbFeatures{1}))
set(gca,'Ydir','normal')
subtitle('Reverberated')
xlabel('Time')
ylabel('Frequency')
colorbar

subplot(3,1,3)
imagesc(squeeze(predictedSTFT4D(:,:,:,1)))
set(gca,'Ydir','normal')
subtitle('Predicted (Dereverberated)')
xlabel('Time')
ylabel('Frequency')
caxis([-1,1])
colorbar
```


Clean
Frequency
Time
Reverberated
Frequency
Time
Predicted (Dereverberated)
Frequency
Time

Evaluation Metrics

You will use a subset of objective measures used in [1] on page 14-0 to evaluate the performance of the network. These metrics are computed on the time-domain signals.

- Cepstrum distance (CD) - Provides an estimate of the log spectral distance between two spectra (predicted and clean). Smaller values indicate better quality.
- Log likelihood ratio (LLR) - Linear predictive coding (LPC) based objective measurement. Smaller values indicate better quality.

Compute these measurements for the reverberant speech and the dereverberated speech signals.

```
[summaryMeasuresReconstructed,allMeasuresReconstructed] = calculateObjectiveMeasures(dereverbedAu
[summaryMeasuresReverb,allMeasuresReverb] = calculateObjectiveMeasures(allValReverbAudios,allVal
disp(summaryMeasuresReconstructed)
```

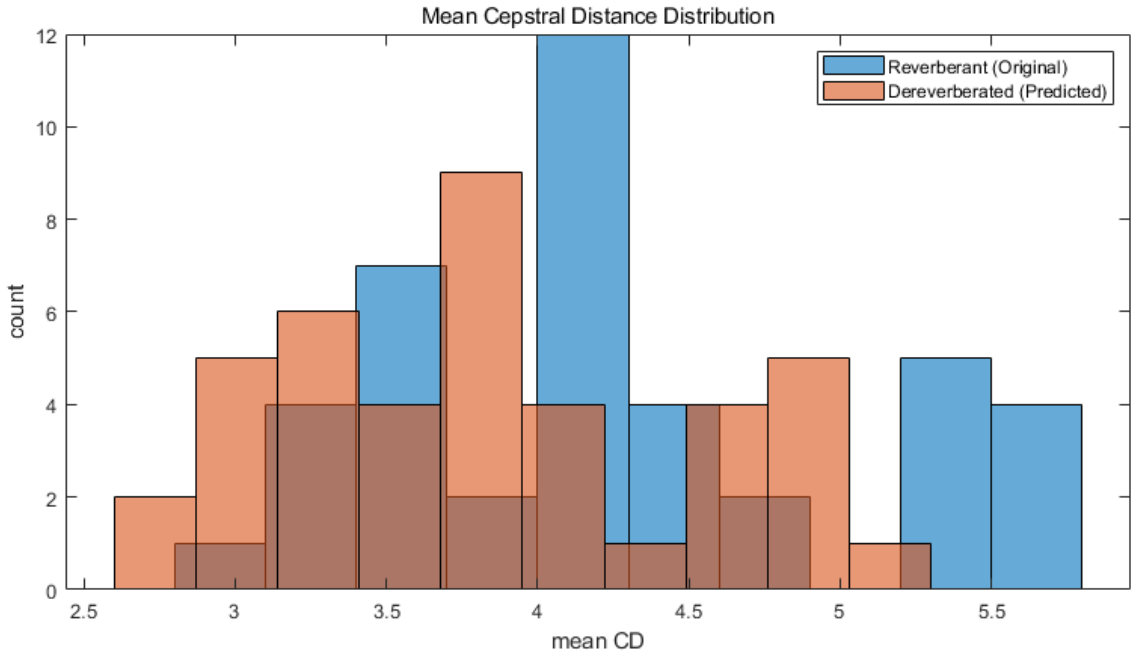
```
       avgCdMean: 3.8386
     avgCdMedian: 3.3671
      avgLlrMean: 0.9152
    avgLlrMedian: 0.8096
```

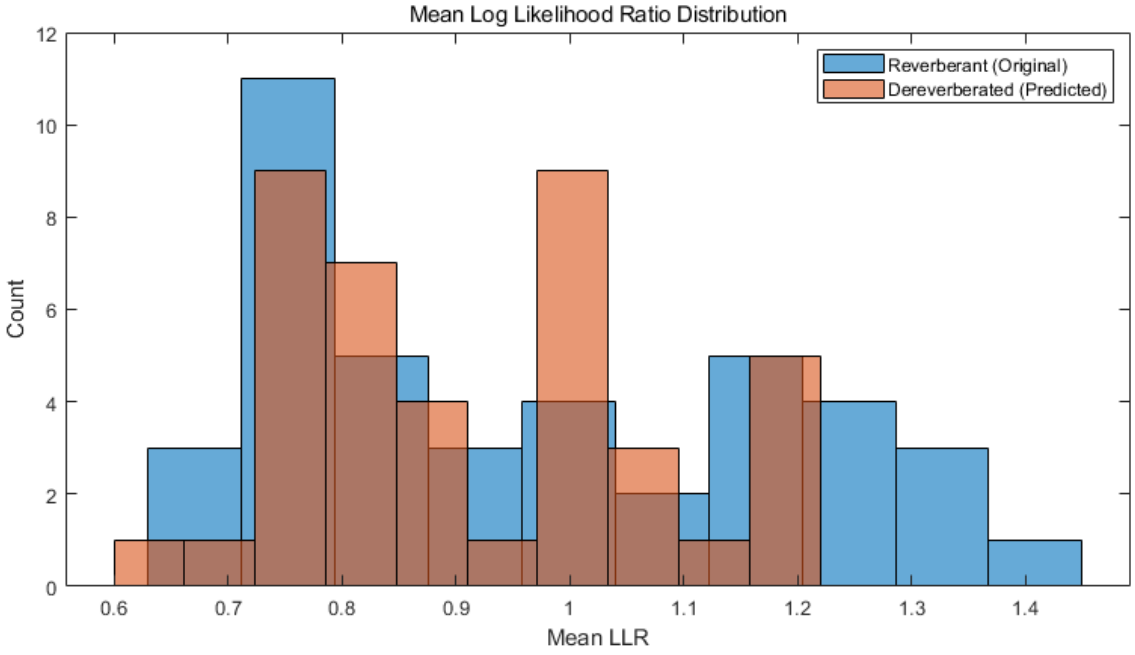
```
disp(summaryMeasuresReverb)
```

```
       avgCdMean: 4.2591
     avgCdMedian: 3.6336
      avgLlrMean: 0.9726
    avgLlrMedian: 0.8714
```

The histograms illustrate the distribution of mean CD, mean SRMR and mean LLR of the reverberant and dereverberated data.

```
figure('position',[50,50,1100,1300])

subplot(2,1,1)
histogram(allMeasuresReverb.cdMean,10)
hold on
histogram(allMeasuresReconstructed.cdMean, 10)
subtitle('Mean Cepstral Distance Distribution')
ylabel('count')
xlabel('mean CD')
legend('Reverberant (Original)','Dereverberated (Predicted)')

subplot(2,1,2)
histogram(allMeasuresReverb.llrMean,10)
hold on
histogram(allMeasuresReconstructed.llrMean,10)
subtitle('Mean Log Likelihood Ratio Distribution')
ylabel('Count')
xlabel('Mean LLR')
legend('Reverberant (Original)','Dereverberated (Predicted)')
```


Mean Cepstral Distance Distribution
Reverberant (Original)
Dereverberated (Predicted)
count
mean CD
0
2
4
6
8
10
12
2.5
3
3.5
4
4.5
5
5.5


Mean Log Likelihood Ratio Distribution
Reverberant (Original)
Dereverberated (Predicted)
Count
Mean LLR
0
2
4
6
8
10
12
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4

References

[1] Ernst, O., Chazan, S.E., Gannot, S., & Goldberger, J. (2018). Speech Dereverberation Using Fully Convolutional Networks. *2018 26th European Signal Processing Conference (EUSIPCO)*, 390-394.

[2] https://datashare.is.ed.ac.uk/handle/10283/2031

[3] https://datashare.is.ed.ac.uk/handle/10283/2791

[4] https://github.com/MuSAELab/SRMRToolbox

Supporting Functions

Apply Reverberation

```
function yOut = applyReverb(y,preDelay,decayFactor,wetDryMix,fs)
% This function generates reverberant speech data using the reverberator
% object
%
% inputs:
% y                                 - clean speech sample
% preDelay, decayFactor, wetDryMix - reverberation parameters
% fs                                - sampling rate of y
%
% outputs:
% yOut - corresponding reveberated speech sample

    revObj = reverberator('SampleRate',fs, ...
        'DecayFactor',decayFactor, ...
        'WetDryMix',wetDryMix, ...
        'PreDelay',preDelay);
    yOut = revObj(y);
    yOut = yOut(1:length(y),1);
end
```

Extract Features Batch

```
function [featuresClean,featuresReverb,phaseReverb,cleanAudios,reverbAudios] ...
    = helperFeatureExtract(cleanAudio,reverbAudio,isVal,params)
% This function performs the preprocessing and features extraction task on
% the audio files used for dereverberation model training and testing.
%
% inputs:
% cleanAudio  - the clean audio file (reference)
% reverbAudio - corresponding reverberant speech file
% isVal       - Boolean flag indicating if it is the validation set
% params      - a structure containing feature extraction parameters
%
% outputs:
% featuresClean  - log-magnitude STFT features of clean audio
% featuresReverb - log-magnitude STFT features of reverberant audio
% phaseReverb    - phase of STFT of reverberant audio
% cleanAudios    - 2.072s-segments of clean audio file used for feature extraction
% reverbAudios   - 2.072s-segments of corresponding reverberant audio

    assert(length(cleanAudio) == length(reverbAudio));
    nSegments = floor((length(reverbAudio) - (params.samplesPerImage - params.shiftImage))/params
```

```
    featuresClean = {};
    featuresReverb = {};
    phaseReverb = {};
    cleanAudios = {};
    reverbAudios = {};
    nGood = 0;
    nonSilentRegions = detectSpeech(reverbAudio, params.fs);
    nonSilentRegionIdx = 1;
    totalRegions = size(nonSilentRegions, 1);
    for cid = 1:nSegments
        start = (cid - 1)*params.shiftImage + 1;
        en = start + params.samplesPerImage - 1;

        nonSilentSamples = 0;
        while nonSilentRegionIdx < totalRegions && nonSilentRegions(nonSilentRegionIdx, 2) < sta
            nonSilentRegionIdx = nonSilentRegionIdx + 1;
        end

        nonSilentStart = nonSilentRegionIdx;
        while nonSilentStart <= totalRegions && nonSilentRegions(nonSilentStart, 1) <= en
            nonSilentDuration = min(en, nonSilentRegions(nonSilentStart,2)) - max(start,nonSilen
            nonSilentSamples = nonSilentSamples + nonSilentDuration;
            nonSilentStart = nonSilentStart + 1;
        end

        nonSilentPerc = nonSilentSamples * 100 / (en - start + 1);
        silent = nonSilentPerc < 50;

        reverbAudioSegment = reverbAudio(start:en);
        if ~silent
            nGood = nGood + 1;
            cleanAudioSegment = cleanAudio(start:en);
            assert(length(cleanAudioSegment)==length(reverbAudioSegment), 'Lengths do not match a

            % Clean Audio
            [featsUnit, ~] = featureExtract(cleanAudioSegment, params);
            featuresClean{nGood} = featsUnit; %#ok

            % Reverb Audio
            [featsUnit, phaseUnit] = featureExtract(reverbAudioSegment, params);
            featuresReverb{nGood} = featsUnit; %#ok
            if isVal
                phaseReverb{nGood} = phaseUnit; %#ok
                reverbAudios{nGood} = reverbAudioSegment;%#ok
                cleanAudios{nGood} = cleanAudioSegment;%#ok
            end
        end
    end
end
```

Extract Features

```
function [features, phase, lastFBin] = featureExtract(audio, params)
% Function to extract features for a speech file
    audio = single(audio);

    audioSTFT = stft(audio,'Window',params.Window,'OverlapLength',params.OverlapLength, ...
                    'FFTLength', params.FFTLength, 'FrequencyRange', 'onesided');
```

```
    phase = single(angle(audioSTFT(1:end-1,:)));
    features = single(log(abs(audioSTFT(1:end-1,:)) + 10e-30));
    lastFBin = audioSTFT(end,:);

end
```

Normalize and Reshape Features

```
function [featNorm,minMaxPairs] = featureNormalizeAndReshape(feats)
% function to normalize features - range [-1, 1] and reshape to 4
% dimensions
%
% inputs:
% feats - 3-dimensional array of extracted features
%
% outputs:
% featNorm - normalized and reshaped features
% minMaxPairs - array of original min and max pairs used for normalization

    nSamples = length(feats);
    minMaxPairs = zeros(nSamples,2,'single');
    featNorm = zeros([size(feats{1}),nSamples],'single');
    parfor i = 1:nSamples
        feat = feats{i};
        maxFeat = max(feat,[],'all');
        minFeat = min(feat,[],'all');
        featNorm(:,:,i) = 2.*(feat - minFeat)./(maxFeat - minFeat) - 1;
        minMaxPairs(i,:) = [minFeat,maxFeat];
    end
    featNorm = reshape(featNorm,size(featNorm,1),size(featNorm,2),1,size(featNorm,3));
end
```

Reconstruct Predicted Audio

```
function dereverbedAudioAll = helperReconstructPredictedAudios(predictedSTFT4D,minMaxPairs,reverb
% This function will reconstruct the 2.072s long audios predicted by the
% model using the predicted log-magnitude spectrogram and the phase of the
% reverberant audio file
%
% inputs:
% predictedSTFT4D - Predicted 4-dimensional STFT log-magnitude features
% minMaxPairs     - Original minimum/maximum value pairs used in normalization
% reverbPhase     - Array of phases of STFT of reverberant audio files
% reverbAudios    - 2.072s-segments of corresponding reverberant audios
% params          - Structure containing feature extraction parameters

    predictedSTFT = squeeze(predictedSTFT4D);
    denormalizedFeatures = zeros(size(predictedSTFT),'single');
    for i = 1:size(predictedSTFT,3)
        feat = predictedSTFT(:,:,i);
        maxFeat = minMaxPairs(i,2);
        minFeat = minMaxPairs(i,1);
        denormalizedFeatures(:,:,i) = (feat + 1).*(maxFeat-minFeat)./2 + minFeat;
    end

    predictedSTFT = exp(denormalizedFeatures);
```

```
    nCount = size(predictedSTFT,3);
    dereverbedAudioAll = cell(1,nCount);

    nSeg = params.NumSegments;
    win = params.Window;
    ovrlp = params.OverlapLength;
    FFTLength = params.FFTLength;
    parfor ii = 1:nCount
        % Append zeros to the highest frequency bin
        stftUnit = predictedSTFT(:,:,ii);
        stftUnit = cat(1,stftUnit, zeros(1,nSeg));
        phase = reverbPhase{ii};
        phase = cat(1,phase,zeros(1,nSeg));

        oneSidedSTFT = stftUnit.*exp(1j*phase);
        dereverbedAudio= istft(oneSidedSTFT, ...
            'Window', win,'OverlapLength', ovrlp, ...
                                    'FFTLength',FFTLength,'ConjugateSymmetric',true,...
                                    'FrequencyRange','onesided');

        dereverbedAudioAll{ii} = dereverbedAudio./max(max(abs(dereverbedAudio)), max(abs(reverbAu
    end
end
```

Calculate Objective Measures

```
function [summaryMeasures,allMeasures] = calculateObjectiveMeasures(reconstructedAudios,cleanAudi
% This function computes the objective measures on time-domain signals.
%
% inputs:
% reconstructedAudios - An array of audio files to evaluate.
% cleanAudios - An array of reference audio files
% fs - Sampling rate of audio files
%
% outputs:
% summaryMeasures - Global means of CD, LLR individual mean and median values
% allMeasures - Individual mean and median values

    nAudios = length(reconstructedAudios);
    cdMean = zeros(nAudios,1);
    cdMedian = zeros(nAudios,1);
    llrMean = zeros(nAudios,1);
    llrMedian = zeros(nAudios,1);

    parfor k = 1 : nAudios
      y = reconstructedAudios{k};
      x = cleanAudios{k};

      y = y./max(abs(y));
      x = x./max(abs(x));

      [cdMean(k),cdMedian(k)] = cepstralDistance(x,y,fs);
      [llrMean(k),llrMedian(k)] = lpcLogLikelihoodRatio(y,x,fs);
    end

    summaryMeasures.avgCdMean = mean(cdMean);
    summaryMeasures.avgCdMedian = mean(cdMedian);
    summaryMeasures.avgLlrMean = mean(llrMean);
```

```
    summaryMeasures.avgLlrMedian = mean(llrMedian);

    allMeasures.cdMean = cdMean;
    allMeasures.llrMean = llrMean;
end
```

Cepstral Distance

```
function [meanVal, medianVal] = cepstralDistance(x,y,fs)
    x = x / sqrt(sum(x.^2));
    y = y / sqrt(sum(y.^2));

    width = round(0.025*fs);
    shift = round(0.01*fs);

    nSamples = length(x);
    nFrames = floor((nSamples - width + shift)/shift);
    win = window(@hanning,width);

    winIndex = repmat((1:width)',1,nFrames) + repmat((0:nFrames - 1)*shift,width,1);

    xFrames = x(winIndex).*win;
    yFrames = y(winIndex).*win;

    xCeps = cepstralReal(xFrames,width);
    yCeps = cepstralReal(yFrames,width);

    dist = (xCeps - yCeps).^2;
    cepsD = 10 / log(10)*sqrt(2*sum(dist(2:end,:),1) + dist(1,:));
    cepsD = max(min(cepsD,10),0);

    meanVal = mean(cepsD);
    medianVal = median(cepsD);
end
```

Real Cepstrum

```
function realC = cepstralReal(x, width)
    width2p = 2 ^ nextpow2(width);
    powX = abs(fft(x, width2p));

    lowCutoff = max(powX(:)) * 10^-5;
    powX  = max(powX, lowCutoff);

    realC = real(ifft(log(powX)));
    order = 24;
    realC = realC(1 : order + 1, :);
    realC = realC - mean(realC, 2);
end
```

LPC Log-Likelihood Ratio

```
function [meanLlr,medianLlr] = lpcLogLikelihoodRatio(x,y,fs)
    order = 12;
    width = round(0.025*fs);
    shift = round(0.01*fs);

    nSamples = length(x);
    nFrames  = floor((nSamples - width + shift)/shift);
```

```
    win = window(@hanning,width);

    winIndex = repmat((1:width)',1,nFrames) + repmat((0:nFrames - 1)*shift,width,1);

    xFrames = x(winIndex) .* win;
    yFrames = y(winIndex) .* win;

    lpcX = realLpc(xFrames, width, order);
    [lpcY,realY] = realLpc(yFrames, width, order);

    llr = zeros(nFrames, 1);
    for n = 1 : nFrames
      R = toeplitz(realY(1:order+1,n));
      num = lpcX(:,n)'*R*lpcX(:,n);
      den = lpcY(:,n)'*R*lpcY(:,n);
      llr(n) = log(num/den);
    end

    llr = sort(llr);
    llr = llr(1:ceil(nFrames*0.95));
    llr = max(min(llr,2),0);

    meanLlr = mean(llr);
    medianLlr = median(llr);
end
```

Real Linear Prection Coefficients

```
function [lpcCoeffs, realX] = realLpc(xFrames, width, order)
    width2p = 2 ^ nextpow2(width);
    X = fft(xFrames, width2p);

    Rx = ifft(abs(X).^2);
    Rx = Rx./width;
    realX = real(Rx);

    lpcX = levinson(realX, order);
    lpcCoeffs = real(lpcX');
end
```

Speaker Recognition Using x-vectors

Speaker recognition answers the question "Who is speaking?". Speaker recognition is usually divided into two tasks: *speaker identification* and *speaker verification*. In speaker identification, a speaker is recognized by comparing their speech to a closed set of templates. In speaker verification, a speaker is recognized by comparing the likelihood that the speech belongs to a particular speaker against a predetermined threshold. Traditional machine learning methods perform well at these tasks in ideal conditions. For examples of speaker identification using traditional machine learning methods, see “Speaker Identification Using Pitch and MFCC” (Audio Toolbox) and “Speaker Verification Using i-Vectors” (Audio Toolbox). Audio Toolbox™ provides `ivectorSystem` (Audio Toolbox) which encapsulates the ability to train an i-vector system, enroll speakers or other audio labels, evaluate the system for a decision threshold, and identify or verify speakers or other audio labels.

In adverse conditions, the deep learning approach of x-vectors has been shown to achieve state-of-the-art results for many scenarios and applications [1] on page 14-0 . The x-vector system is an evolution of i-vectors originally developed for the task of speaker verification.

In this example, you develop an x-vector system. First, you train a time-delay neural network (TDNN) to perform speaker identification. Then you train the traditional backends for an x-vector-based speaker verification system: an LDA projection matrix and a PLDA model. You then perform speaker verification using the TDNN and the backend dimensionality reduction and scoring. The x-vector system backend, or classifier, is the same as developed for i-vector systems. For details on the backend, see “Speaker Verification Using i-Vectors” (Audio Toolbox) and `ivectorSystem` (Audio Toolbox).

In Speaker Diarization Using x-vectors, you use the x-vector system trained in this example to perform speaker diarization. Speaker diarization answers the question, "Who spoke when?".

Throughout this example, you will find live controls on tunable parameters. Changing the controls does not rerun the example. If you change a control, you must rerun the example.

Data Set Management

This example uses the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [2] on page 14-0 . The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set. Depending on your system, downloading and extracting the data set can take approximately 1.5 hours.

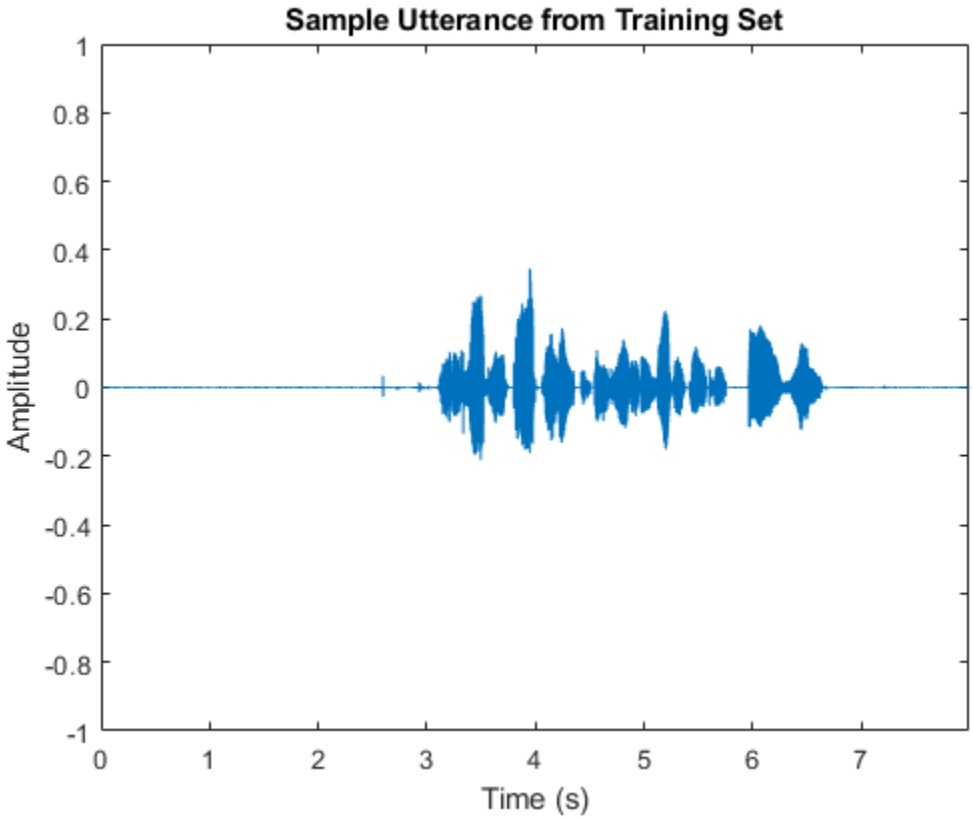
```
url = 'https://www2.spsc.tugraz.at/databases/PTDB-TUG/SPEECH_DATA_ZIPPED.zip';
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'PTDB-TUG');
if ~exist(datasetFolder,'dir')
    disp('Downloading PTDB-TUG (3.9 G) ...')
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` (Audio Toolbox) object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation, and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(datasetFolder,"SPEECH DATA","FEMALE","MIC"),fullfile(datasetFolde
                      'IncludeSubfolders',true, ...
                      'FileExtensions','.wav');
fileNames = ads.Files;
```

Read an audio file from the training data set, listen to it, and then plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;
t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
axis([0 t(end) -1 1])
title('Sample Utterance from Training Set')
```



The file names contain the speaker IDs. Decode the file names to set the labels on the `audioDatastore` object.

```
speakerIDs = extractBetween(fileNames,'mic_','_');
ads.Labels = categorical(speakerIDs);
```

Separate the `audioDatastore` object into five sets:

- `adsTrain` - Contains training set for the TDNN and backend classifier.
- `adsValidation` - Contains validation set to evaluate TDNN training progress.
- `adsTest` - Contains test set to evaluate the TDNN performance for speaker identification.
- `adsEnroll` - Contains enrollment set to evaluate the detection error tradeoff of the x-vector system for speaker verification.

- `adsDET` - Contains evaluation set used to determine the detection error tradeoff of the x-vector system for speaker verification.

```
developmentLabels = categorical(["M01","M02","M03","M04","M06","M07","M08","M09","F01","F02","F0
evaluationLabels = categorical(["M05","M010","F05","F010"]);
adsTrain = subset(ads,ismember(ads.Labels,developmentLabels));
[adsTrain,adsValidation,adsTest] = splitEachLabel(adsTrain,0.8,0.1,0.1);
adsEvaluate = subset(ads,ismember(ads.Labels,evaluationLabels));
[adsEnroll,adsDET] = splitEachLabel(adsEvaluate,3);
```

Display the label distributions of the resulting `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F01      189
     F02      189
     F03      189
     F04      189
     F06      189
     F07      189
     F08      187
     F09      189
     M01      189
     M02      189
     M03      189
     M04      189
     M06      189
     M07      189
     M08      189
     M09      189
```

```
countEachLabel(adsValidation)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F01      23
     F02      23
     F03      23
     F04      23
     F06      23
     F07      23
     F08      24
     F09      23
     M01      23
     M02      23
     M03      23
     M04      23
     M06      23
     M07      23
     M08      23
     M09      23
```

```
countEachLabel(adsTest)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F01      24
     F02      24
     F03      24
     F04      24
     F06      24
     F07      24
     F08      23
     F09      24
     M01      24
     M02      24
     M03      24
     M04      24
     M06      24
     M07      24
     M08      24
     M09      24
```

```
countEachLabel(adsEnroll)
```

```
ans=2×2 table
    Label    Count
    _____    _____

     F05       3
     M05       3
```

```
countEachLabel(adsDET)
```

```
ans=2×2 table
    Label    Count
    _____    _____

     F05      233
     M05      233
```

You can reduce the training and detection error trade-off datasets used in this example to speed up the runtime at the cost of performance. In general, reducing the data set is a good practice for development and debugging.

```
speedUpExample = false;
if speedUpExample
    adsTrain = splitEachLabel(adsTrain,20);
    adsDET = splitEachLabel(adsDET,20);
end
```

Feature Extraction

Create an `audioFeatureExtractor` (Audio Toolbox) object to extract 30 MFCCs from 30 ms Hann windows with a 10 ms hop. The sample rate of the data set is 48 kHz, but you will downsample the data set to 16 kHz. Design the `audioFeatureExtractor` assuming the desired sample rate, 16 kHz.

```
desiredFs = 16e3;

windowDuration = 0.03 ;
hopDuration = 0.005 ;
windowSamples = round(windowDuration*desiredFs);
hopSamples = round(hopDuration*desiredFs);
overlapSamples = windowSamples - hopSamples;
numCoeffs = 30 ;
afe = audioFeatureExtractor( ...
    'SampleRate',desiredFs, ...
    'Window',hann(windowSamples,'periodic'), ...
    'OverlapLength',overlapSamples, ...
    ...
    'mfcc',true, ...
    'pitch',false, ...
    'spectralEntropy',false, ...
    'spectralFlux',false);
setExtractorParams(afe,'mfcc','NumCoeffs',numCoeffs)
```

Downsample the audio data to 16 kHz and extract features from the train and validation data sets. Use the training data set to determine the mean and standard deviation of the features to perform feature standardization. The supporting function, xVectorPreprocessBatch on page 14-0 , uses your default parallel pool if you have Parallel Computing Toolbox™.

```
adsTrain = transform(adsTrain,@(x)resample(x,desiredFs,fs));
[features,YTrain] = xVectorPreprocessBatch(adsTrain,afe);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
featuresMAT = cat(1,features{:});
numFeatures = size(featuresMAT,2);
factors = struct('Mean',mean(featuresMAT,1),'STD',std(featuresMAT,1));

XTrain = cellfun(@(x)(x-factors.Mean)./factors.STD,features,'UniformOutput',false);
XTrain = cellfun(@(x)x-mean(x(:)),XTrain,'UniformOutput',false);

adsValidation = transform(adsValidation,@(x)resample(x,desiredFs,fs));
[XValidation,YValidation] = xVectorPreprocessBatch(adsTrain,afe,'Factors',factors);

classes = unique(YTrain);
numClasses = numel(classes);
```

x-vector Feature Extraction Model

In this example, you implement the x-vector feature extractor model [1] on page 14-0 using the functional programming paradigm provided by Deep Learning Toolbox™. This paradigm enables complete control of the design of your deep learning model. For a tutorial on functional programming in Deep Learning Toolbox, see “Define Model Gradients Function for Custom Training Loop” on page

18-231. The supporting function, `xvecModel`, is placed in your current folder when you open this example. Display the contents of the `xvecModel` function.

```
type('xvecModel')
```

```
function [Y,state] = xvecModel(X,parameters,state,nvargs)
% This function is only for use in this example. It may be changed or
% removed in a future release.
arguments
    X
    parameters
    state
    nvargs.DoTraining = false
    nvargs.OutputLayer = 'final'
    nvargs.Dropout = 0.2;
end


% LAYER 1 ------------------------------------------------------------------
Y = dlconv(X,parameters.conv1.Weights,parameters.conv1.Bias,'DilationFactor',1);
if nvargs.DoTraining
    [Y,state.batchnorm1.TrainedMean,state.batchnorm1.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm1.Offset, ...
        parameters.batchnorm1.Scale, ...
        state.batchnorm1.TrainedMean, ...
        state.batchnorm1.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
    Y = batchnorm(Y, ...
        parameters.batchnorm1.Offset, ...
        parameters.batchnorm1.Scale, ...
        state.batchnorm1.TrainedMean, ...
        state.batchnorm1.TrainedVariance);
end
if nvargs.OutputLayer==1
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------


% LAYER 2 ------------------------------------------------------------------
Y = dlconv(Y,parameters.conv2.Weights,parameters.conv2.Bias,'DilationFactor',2);
if nvargs.DoTraining
    [Y,state.batchnorm2.TrainedMean,state.batchnorm2.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm2.Offset, ...
        parameters.batchnorm2.Scale, ...
        state.batchnorm2.TrainedMean, ...
        state.batchnorm2.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
    Y = batchnorm(Y, ...
        parameters.batchnorm2.Offset, ...
        parameters.batchnorm2.Scale, ...
        state.batchnorm2.TrainedMean, ...
        state.batchnorm2.TrainedVariance);
```

```
end
if nvargs.OutputLayer==2
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------


% LAYER 3 ----------------------------------------------------------------
Y = dlconv(Y,parameters.conv3.Weights,parameters.conv3.Bias,'DilationFactor',3);
if nvargs.DoTraining
    [Y,state.batchnorm3.TrainedMean,state.batchnorm3.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm3.Offset, ...
        parameters.batchnorm3.Scale, ...
        state.batchnorm3.TrainedMean, ...
        state.batchnorm3.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
    Y = batchnorm(Y, ...
        parameters.batchnorm3.Offset, ...
        parameters.batchnorm3.Scale, ...
        state.batchnorm3.TrainedMean, ...
        state.batchnorm3.TrainedVariance);
end
if nvargs.OutputLayer==3
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------


% LAYER 4 ----------------------------------------------------------------
Y = dlconv(Y,parameters.conv4.Weights,parameters.conv4.Bias,'DilationFactor',1);
if nvargs.DoTraining
    [Y,state.batchnorm4.TrainedMean,state.batchnorm4.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm4.Offset, ...
        parameters.batchnorm4.Scale, ...
        state.batchnorm4.TrainedMean, ...
        state.batchnorm4.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
    Y = batchnorm(Y, ...
        parameters.batchnorm4.Offset, ...
        parameters.batchnorm4.Scale, ...
        state.batchnorm4.TrainedMean, ...
        state.batchnorm4.TrainedVariance);
end
if nvargs.OutputLayer==4
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------


% LAYER 5 ----------------------------------------------------------------
Y = dlconv(Y,parameters.conv5.Weights,parameters.conv5.Bias,'DilationFactor',1);
```

```
if nvargs.DoTraining
    [Y,state.batchnorm5.TrainedMean,state.batchnorm5.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm5.Offset, ...
        parameters.batchnorm5.Scale, ...
        state.batchnorm5.TrainedMean, ...
        state.batchnorm5.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
    Y = batchnorm(Y, ...
        parameters.batchnorm5.Offset, ...
        parameters.batchnorm5.Scale, ...
        state.batchnorm5.TrainedMean, ...
        state.batchnorm5.TrainedVariance);
end
if nvargs.OutputLayer==5
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------


% Layer 6: Statistical pooling ------------------------------------------
if nvargs.DoTraining
    Y = Y + 0.0001*rand(size(Y));
end
Y = cat(2,mean(Y,1),std(Y,[],1));
if nvargs.OutputLayer==6
    return
end
% ------------------------------------------------------------------------

% LAYER 7 ---------------------------------------------------------------
Y = fullyconnect(Y,parameters.fc7.Weights,parameters.fc7.Bias);
if nvargs.DoTraining
    [Y,state.batchnorm7.TrainedMean,state.batchnorm6.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm7.Offset, ...
        parameters.batchnorm7.Scale, ...
        state.batchnorm7.TrainedMean, ...
        state.batchnorm7.TrainedVariance);
     Y(rand(size(Y))<nvargs.Dropout) = 0;
else
        Y = batchnorm(Y, ...
            parameters.batchnorm7.Offset, ...
            parameters.batchnorm7.Scale, ...
            state.batchnorm7.TrainedMean, ...
            state.batchnorm7.TrainedVariance);
end
if nvargs.OutputLayer==7
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------

% LAYER 8 ---------------------------------------------------------------
Y = fullyconnect(Y,parameters.fc8.Weights,parameters.fc8.Bias);
if nvargs.DoTraining
```

```
    [Y,state.batchnorm8.TrainedMean,state.batchnorm8.TrainedVariance] = ...
        batchnorm(Y, ...
        parameters.batchnorm8.Offset, ...
        parameters.batchnorm8.Scale, ...
        state.batchnorm8.TrainedMean, ...
        state.batchnorm8.TrainedVariance);
    Y(rand(size(Y))<nvargs.Dropout) = 0;
else
        Y = batchnorm(Y, ...
            parameters.batchnorm8.Offset, ...
            parameters.batchnorm8.Scale, ...
            state.batchnorm8.TrainedMean, ...
            state.batchnorm8.TrainedVariance);
end
if nvargs.OutputLayer==8
    return
end
Y = relu(Y);
% ------------------------------------------------------------------------

% LAYER 9 (softmax)-------------------------------------------------------
Y = fullyconnect(Y,parameters.fc9.Weights,parameters.fc9.Bias);
if nvargs.OutputLayer==9
    return
end
Y = softmax(Y);
% ------------------------------------------------------------------------
end
```

Initialize structs that contain the parameters and state of the TDNN model using the supporting function, initializexVecModelLayers on page 14-0 . [1] on page 14-0 specifies the number of filters between most layers, including the embedding layer, as 512. Because the training set in this example is small, use a representation size of 128.

```
numFilters = 128;
[parameters,state] = initializexVecModelLayers(numFeatures,numFilters,numClasses)

parameters = struct with fields:
         conv1: [1×1 struct]
    batchnorm1: [1×1 struct]
         conv2: [1×1 struct]
    batchnorm2: [1×1 struct]
         conv3: [1×1 struct]
    batchnorm3: [1×1 struct]
         conv4: [1×1 struct]
    batchnorm4: [1×1 struct]
         conv5: [1×1 struct]
    batchnorm5: [1×1 struct]
           fc7: [1×1 struct]
    batchnorm7: [1×1 struct]
           fc8: [1×1 struct]
    batchnorm8: [1×1 struct]
           fc9: [1×1 struct]


state = struct with fields:
    batchnorm1: [1×1 struct]
```

```
    batchnorm2: [1×1 struct]
    batchnorm3: [1×1 struct]
    batchnorm4: [1×1 struct]
    batchnorm5: [1×1 struct]
    batchnorm7: [1×1 struct]
    batchnorm8: [1×1 struct]
```

The table summarizes the architecture of the network described in [1] on page 14-0 and implemented in this example. T is the total number of frames (feature vectors over time) in an audio signal. N is the number of classes (speakers) in the training set.

Layer	Description	Layer Context	Total Context	Input-by-Output
1	**1-d convolutional** batch normalization ReLu activation	$[t-2, t+2]$	5	$(5 \times numFeatures) - by - numFilters$
2	**1-d convolutional** batch normalization ReLu activation	$\{t-2, t, t+2\}$	9	$(3 \times numFilters) - by - numFilters$
3	**1-d convolutional** batch normalization ReLu activation	$\{t-3, t, t+3\}$	15	$(3 \times numFilters) - by - numFilters$
4	**1-d convolutional** batch normalization ReLu activation	$\{t\}$	15	$numFilters - by - numFilters$
5	**1-d convolutional** batch normalization ReLu activation	$\{t\}$	15	$numFilters - by - 1500$
6	**statistics pooling**	$[0, T)$	T	$(1500 \times \mathrm{T}) - by - 3000$
7	**fully-connected** batch normalization ReLu activation	$\{0\}$	T	$3000 - by - numFilters$
8	**fully-connected** batch normalization ReLu activation	$\{0\}$	T	$numFilters - by - numFilters$
9	**fully-connected** softmax	$\{0\}$	T	$numFilters - by - N$

Train Model

Use `arrayDatastore` and `minibatchqueue` to create a mini-batch queue for the training data. If you have access to a compute GPU, set `ExecutionEnvironment` to `gpu`. Otherwise, set `ExecutionEnvironment` to `cpu`.

```
ExecutionEnvironment = gpu;

dsXTrain = arrayDatastore(XTrain,'OutputType','same');
dsYTrain = arrayDatastore(YTrain','OutputType','cell');

dsTrain = combine(dsXTrain,dsYTrain);
```

```
miniBatchSize =  128 ;
numOutputs = 2;
mbq = minibatchqueue(dsTrain,numOutputs, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFormat',{'SCB','CB'}, ...
    'MiniBatchFcn',@preprocessMiniBatch, ...
    'OutputEnvironment',ExecutionEnvironment);
```

Set the number of training epochs, the initial learn rate, the learn rate drop period, the learn rate drop factor, and the validations per epoch.

```
numEpochs =  6 ;

learnRate =  0.001 ;
gradDecay = 0.5;
sqGradDecay = 0.999;
trailingAvg = [];
trailingAvgSq = [];

LearnRateDropPeriod =  2 ;
LearnRateDropFactor =  0.1 ;

ValidationsPerEpoch =  2 ;

iterationsPerEpoch = floor(numel(XTrain)/miniBatchSize);
iterationsPerValidation = round(iterationsPerEpoch/ValidationsPerEpoch);
```

If performing validation while training, preprocess the validation set for faster in-the-loop performance.

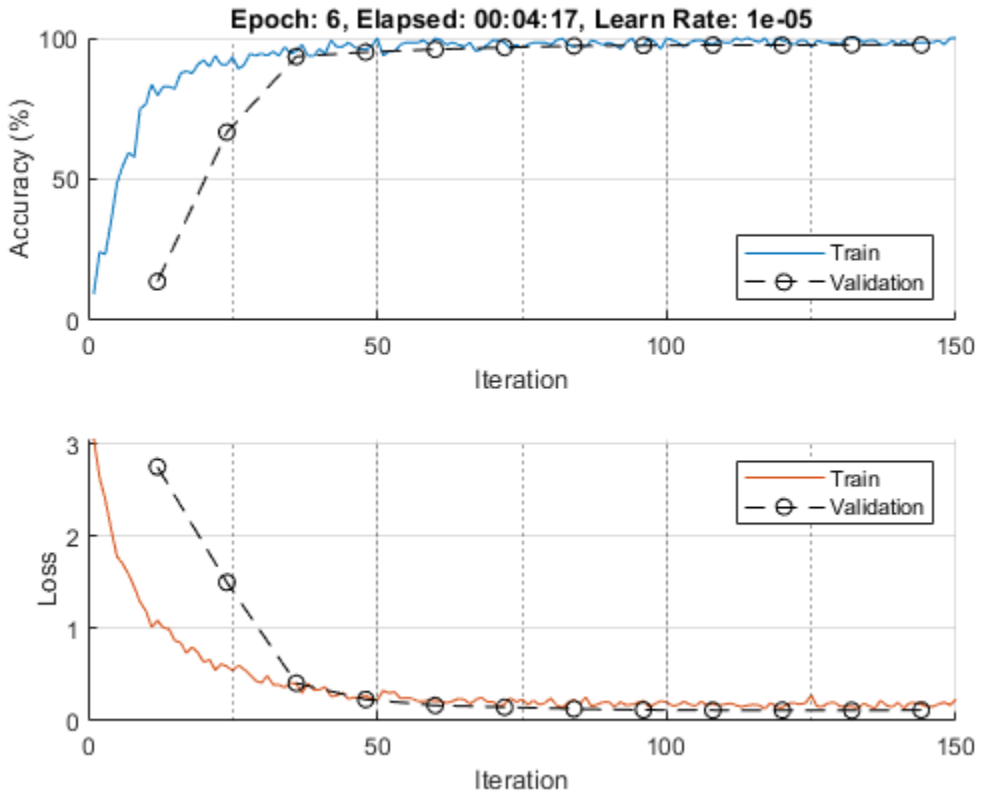
```
if ValidationsPerEpoch ~= 0
    [XValidation,YValidation] = preprocessMiniBatch(XValidation,{YValidation});
    XValidation = dlarray(XValidation,'SCB');
    if strcmp(ExecutionEnvironment,'gpu')
        XValidation = gpuArray(XValidation);
    end
end
```

To display training progress, initialize the supporting object `progressPlotter`. The supporting object, `progressPlotter`, is placed in your current folder when you open this example.

Run the training loop.

```
pp = progressPlotter(categories(classes));

iteration = 0;
for epoch = 1:numEpochs

    % Shuffle mini-batch queue
    shuffle(mbq)

    while hasdata(mbq)
```

```
            % Update iteration counter
            iteration = iteration + 1;

            % Get mini-batch from mini-batch queue
            [dlX,Y] = next(mbq);

            % Evaluate the model gradients, state, and loss using dlfeval and the modelGradients fun
            [gradients,state,loss,predictions] = dlfeval(@modelGradients,dlX,Y,parameters,state);

            % Update the network parameters using the Adam optimizer
            [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
                trailingAvg,trailingAvgSq,iteration,learnRate,gradDecay,sqGradDecay,eps('single'));

            % Update the training progress plot
            updateTrainingProgress(pp,'Epoch',epoch,'Iteration',iteration,'LearnRate',learnRate,'Pre

            % Update the validation plot
            if ~rem(iteration,iterationsPerValidation)

                % Pass validation data through x-vector model
                predictions = xvecModel(XValidation,parameters,state,'DoTraining',false);

                % Update plot
                updateValidation(pp,'Iteration',iteration,'Predictions',predictions,'Targets',YValida
            end
        end

        % Update learn rate
        if rem(epoch,LearnRateDropPeriod)==0
            learnRate = learnRate*LearnRateDropFactor;
        end

    end
```



Evaluate TDNN Model

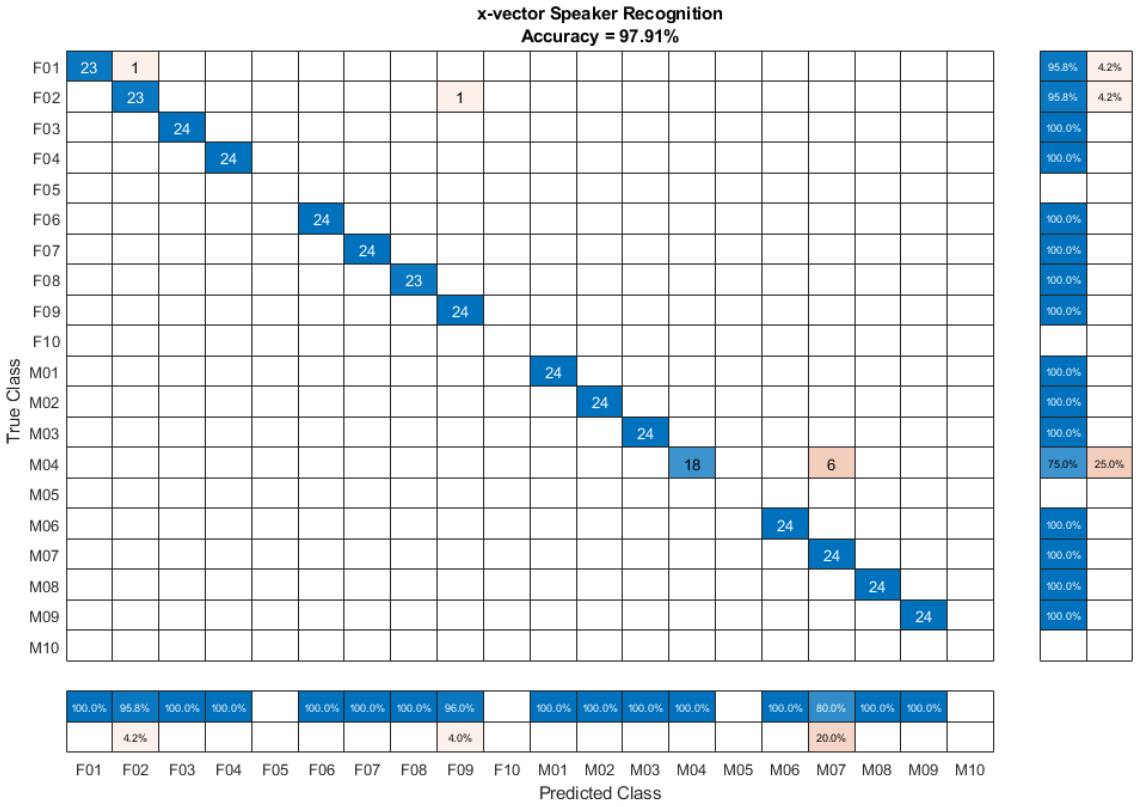
Evaluate the TDNN speaker recognition accuracy using the held-out test set. For each file in the test set:

1. Resample the audio to 16 kHz
2. Extract features using the `xVectorPreprocess` on page 14-0 supporting function. Features are returned in cell arrays, where the number of elements in a cell array is equal to the number of individual speech segments.
3. To get the predicted speaker label, pass each segment through the model.
4. If more than one speech segment was present in the audio signal, average the predictions.
5. Use `onehotdecode` to convert the prediction to a label.

Use `confusionchart` to evaluate the system performance.

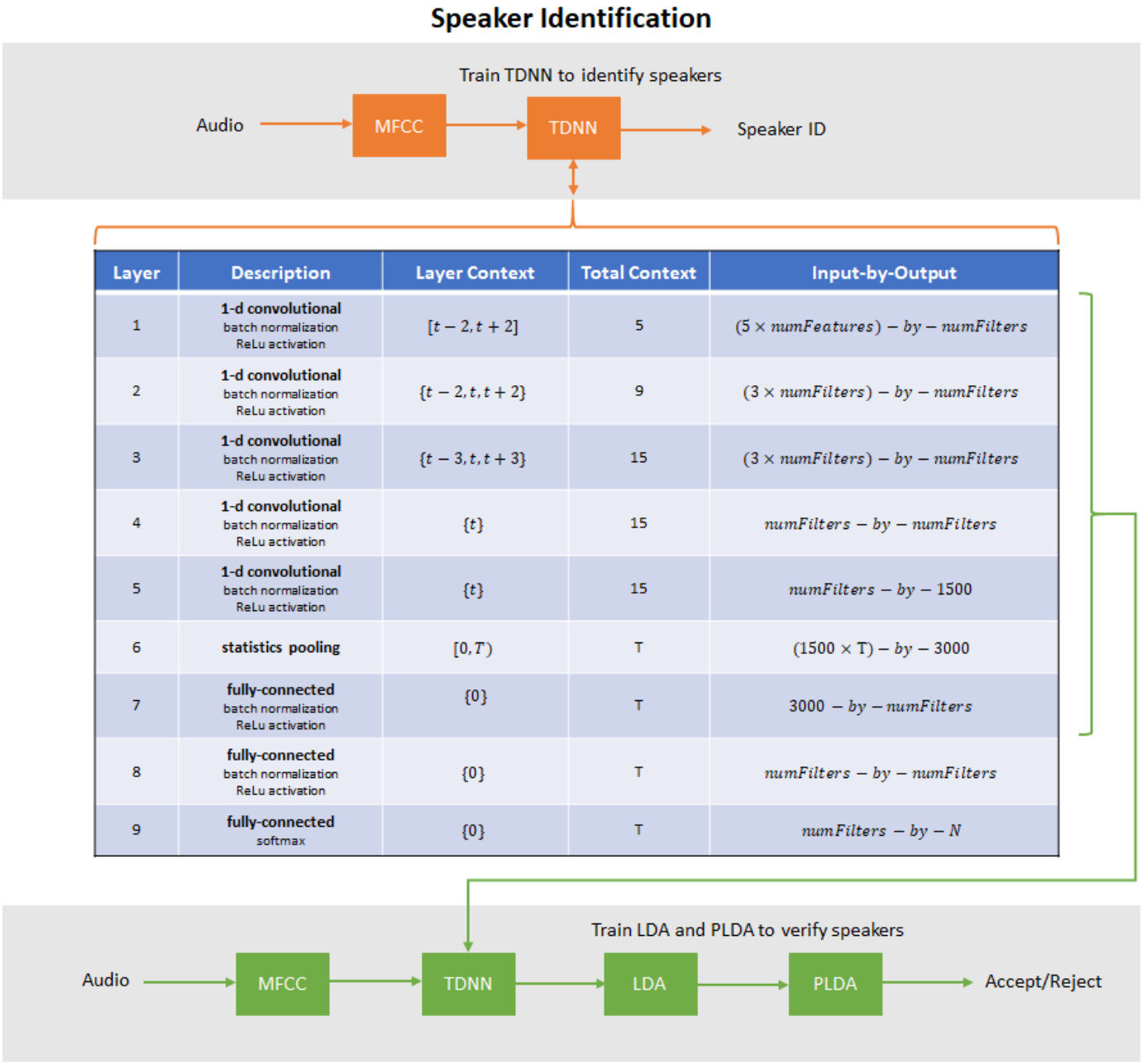
```
predictedLabels = classes;
predictedLabels(:) = [];

for sample = 1:numel(adsTest.Files)
    [audioIn,xInfo] = read(adsTest);
    audioIn = resample(audioIn,desiredFs,fs);
    f = xVectorPreprocess(audioIn,afe,'Factors',factors,'MinimumDuration',0);
    predictions = zeros(numel(classes),numel(f));
    for segment = 1:numel(f)
        dlX = dlarray(f{segment},'SCB');
```

```
        predictions(:,segment) = extractdata(xvecModel(dlX,parameters,state,'DoTraining',false));
    end
    predictedLabels(sample) = onehotdecode(mean(predictions,2),categories(classes),1);
end
trueLabels = adsTest.Labels;
accuracy = mean(trueLabels==predictedLabels');

figure('Units','normalized','Position',[0.2 0.2 0.6 0.6]);
confusionchart(trueLabels,predictedLabels', ...
    'ColumnSummary','column-normalized', ...
    'RowSummary','row-normalized', ...
    'Title',sprintf('x-vector Speaker Recognition\nAccuracy = %0.2f%%',accuracy*100))
```



Train x-vector System Backend

In the x-vector system for speaker verification, the TDNN you just trained is used to output an embedding layer. The output from the embedding layer (layer 7 in this example, after batch normalization and before activation) are the 'x-vectors' in an x-vector system.



Layer	Description	Layer Context	Total Context	Input-by-Output
1	**1-d convolutional** batch normalization ReLu activation	$[t-2, t+2]$	5	$(5 \times numFeatures) - by - numFilters$
2	**1-d convolutional** batch normalization ReLu activation	$\{t-2, t, t+2\}$	9	$(3 \times numFilters) - by - numFilters$
3	**1-d convolutional** batch normalization ReLu activation	$\{t-3, t, t+3\}$	15	$(3 \times numFilters) - by - numFilters$
4	**1-d convolutional** batch normalization ReLu activation	$\{t\}$	15	$numFilters - by - numFilters$
5	**1-d convolutional** batch normalization ReLu activation	$\{t\}$	15	$numFilters - by - 1500$
6	**statistics pooling**	$[0, T)$	T	$(1500 \times T) - by - 3000$
7	**fully-connected** batch normalization ReLu activation	$\{0\}$	T	$3000 - by - numFilters$
8	**fully-connected** batch normalization ReLu activation	$\{0\}$	T	$numFilters - by - numFilters$
9	**fully-connected** softmax	$\{0\}$	T	$numFilters - by - N$

The backend (or classifier) of an x-vector system is the same as the backend of an i-vector system. For details on the algorithms, see `ivectorSystem` (Audio Toolbox) and “Speaker Verification Using i-Vectors” (Audio Toolbox).

Extract x-vectors from the train set. The supporting function, `xvecModel`, has the optional name-value pair `'OutputLayer'`. Set `'OutputLayer'` to 7 to return the output of the seventh layer. In [1] on page 14-0 , the output from either layer 7 or layer 8 are suggested as possible embedding layers.

```
xvecs = zeros(numFilters,numel(YTrain));
for sample = 1:size(YTrain,2)
```

```
    dlX = dlarray(XTrain{sample},'SCB');

    embedding = xvecModel(dlX,parameters,state,'DoTraining',false,'OutputLayer',7);
    xvecs(:,sample) = extractdata(embedding);
end
```

Create a linear discriminant analysis (LDA) projection matrix to reduce the dimensionality of the x-vectors to 32. LDA attempts to minimize the intra-class variance and maximize the variance between speakers.

```
numEigenvectors = 32;
projMat = helperTrainProjectionMatrix(xvecs,YTrain,numEigenvectors);
```

Apply the LDA projection matrix to the x-vectors.

```
xvecs = projMat*xvecs;
```

Train a G-PLDA model to perform scoring.

```
numIterations = 3;
numDimensions = 32;
plda = helperTrainPLDA(xvecs,YTrain,numIterations,numDimensions);
```

Evaluate x-vector System

Speaker verification systems verify that a speaker is who they purport to be. Before a speaker can be verified, they must be enrolled in the system. Enrollment in the system means that the system has a template x-vector representation of the speaker.

Enroll Speakers

Extract x-vectors from the held-out data set, `adsEnroll`. Set the minimum duration of an audio segment to the equivalent of 15 features hops (the minimum number required to calculate x-vectors).

```
minDur = (numel(afe.Window)+14*(numel(afe.Window)-afe.OverlapLength)+1)/desiredFs;

xvecs = zeros(numEigenvectors,numel(adsEnroll.Files));
reset(adsEnroll)
for sample = 1:numel(adsEnroll.Files)
    [audioIn,xInfo] = read(adsEnroll);
    audioIn = resample(audioIn,desiredFs,fs);
    f = xVectorPreprocess(audioIn,afe,'Factors',factors,'MinimumDuration',minDur);
    embeddings = zeros(numFilters,numel(f));
    for segment = 1:numel(f)
        dlX = dlarray(f{segment},'SCB');

        embeddings(:,segment) = extractdata(xvecModel(dlX,parameters,state,'DoTraining',false,'Ou
    end
    xvecs(:,sample) = mean(projMat*embeddings,2);
end
```

Create template x-vectors for each speaker by averaging the x-vectors of individual speakers across enrollment files.

```
labels = adsEnroll.Labels;
uniqueLabels = unique(labels);
```

```
atable = cell2table(cell(0,2),'VariableNames',{'xvector','NumSamples'});
for ii = 1:numel(uniqueLabels)
    idx = uniqueLabels(ii)==labels;
    wLocalMean = mean(xvecs(:,idx),2);
    localTable = table({wLocalMean},(sum(idx)), ...
        'VariableNames',{'xvector','NumSamples'}, ...
        'RowNames',string(uniqueLabels(ii)));
    atable = [atable;localTable]; %#ok<AGROW>
end
enrolledLabels = atable
```

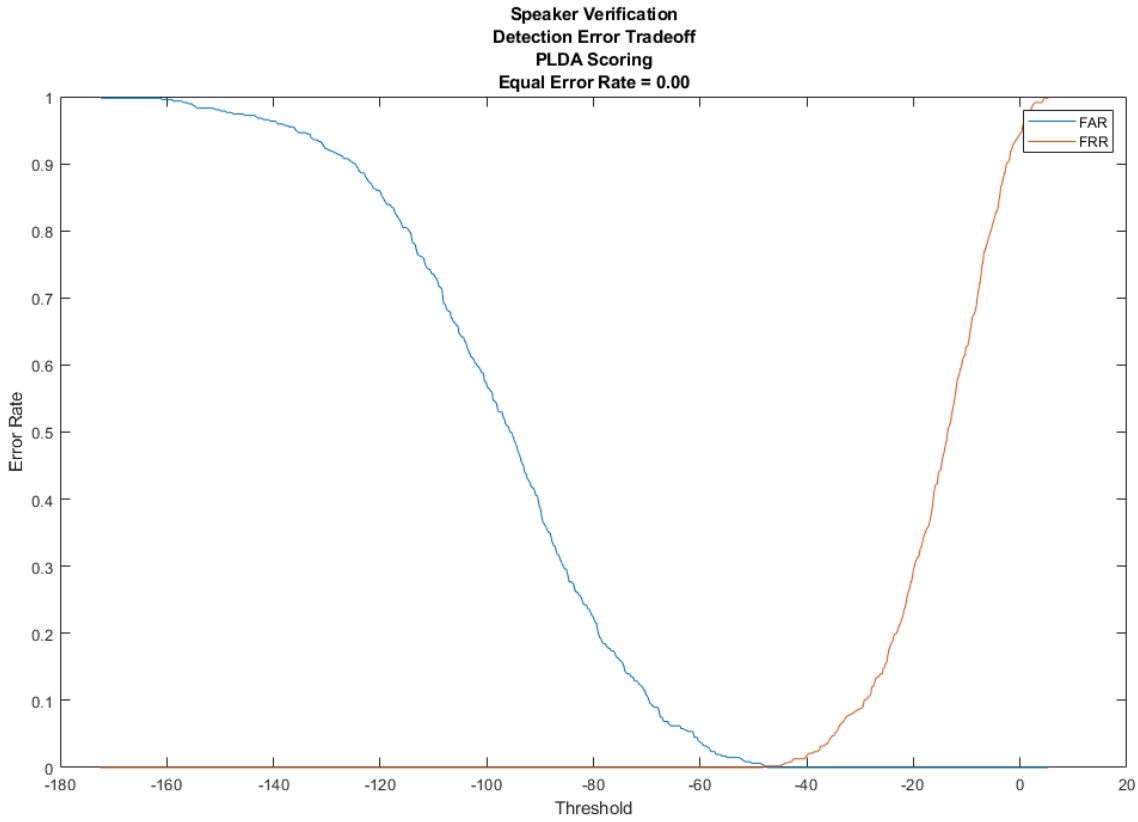
```
enrolledLabels=2×2 table
              xvector       NumSamples
            ____________    __________

    F05     {32×1 double}        3
    M05     {32×1 double}        3
```

Speaker verification systems require you to set a threshold that balances the probability of a false acceptance (FA) and the probability of a false rejection (FR), according to the requirements of your application. To determine the threshold that meets your FA/FR requirements, evaluate the detection error tradeoff of the system.

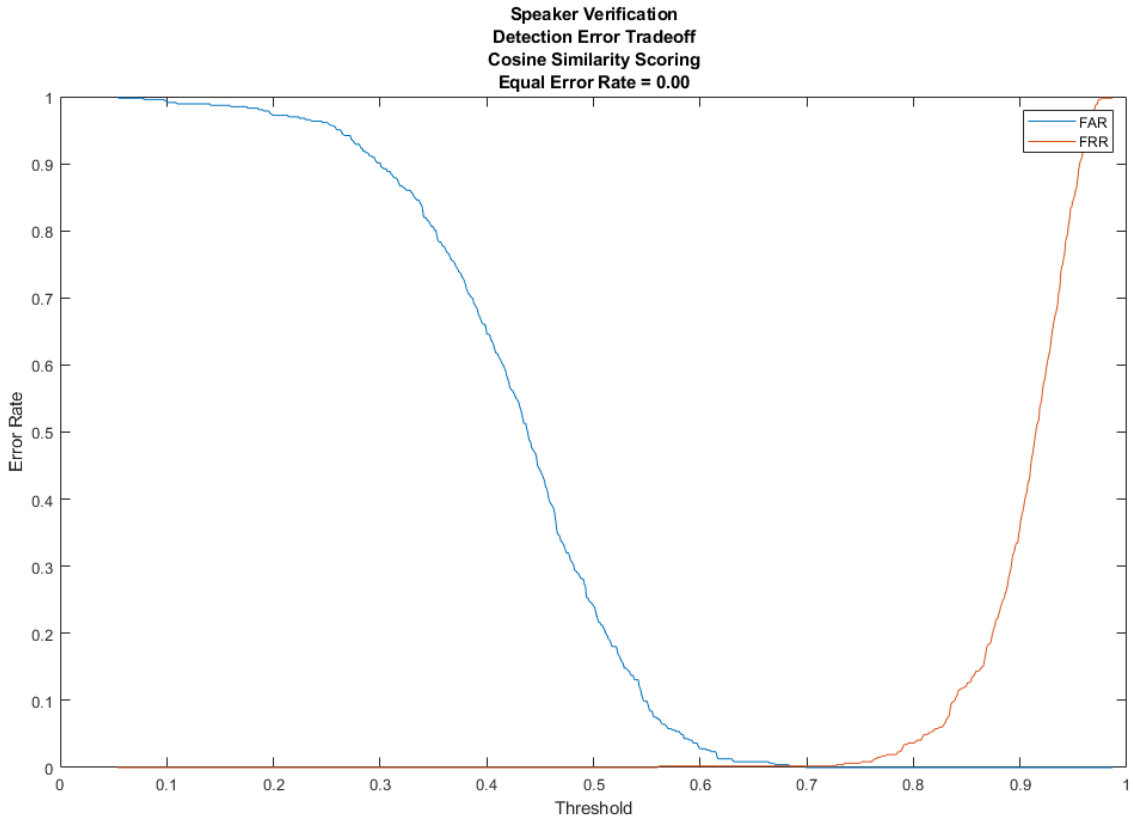
```
xvecs = zeros(numEigenvectors,numel(adsDET.Files));
reset(adsDET)
for sample = 1:numel(adsDET.Files)
    [audioIn,xInfo] = read(adsDET);
    audioIn = resample(audioIn,desiredFs,fs);
    f = xVectorPreprocess(audioIn,afe,'Factors',factors,'MinimumDuration',minDur);
    embeddings = zeros(numFilters,numel(f));
    for segment = 1:numel(f)
        dlX = dlarray(f{segment},'SCB');
        embeddings(:,segment) = extractdata(xvecModel(dlX,parameters,state,'DoTraining',false,'Ou
    end
    xvecs(:,sample) = mean(projMat*embeddings,2);
end
labels = adsDET.Labels;
detTable = helperDetectionErrorTradeoff(xvecs,labels,enrolledLabels,plda);
```

Plot the results of the detection error tradeoff evaluation for both PLDA scoring and cosine similarity scoring (CSS).

```
plot(detTable.PLDA.Threshold,detTable.PLDA.FAR, ...
    detTable.PLDA.Threshold,detTable.PLDA.FRR)
eer = helperEqualErrorRate(detTable.PLDA);
title(sprintf('Speaker Verification\nDetection Error Tradeoff\nPLDA Scoring\nEqual Error Rate = %
xlabel('Threshold')
ylabel('Error Rate')
legend({'FAR','FRR'})
```




```
plot(detTable.CSS.Threshold,detTable.CSS.FAR, ...
    detTable.CSS.Threshold,detTable.CSS.FRR)
eer = helperEqualErrorRate(detTable.CSS);
title(sprintf('Speaker Verification\nDetection Error Tradeoff\nCosine Similarity Scoring\nEqual
xlabel('Threshold')
ylabel('Error Rate')
legend({'FAR','FRR'})
```



References

[1] Snyder, David, et al. “x-vectors: Robust DNN Embeddings for Speaker Recognition.” *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2018, pp. 5329–33. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2018.8461375.

[2] Signal Processing and Speech Communication Laboratory. Accessed December 12, 2019. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html.

Supporting Functions

Initialize Parameters of TDNN Layers

```
function [parameters,state] = initializexVecModelLayers(numFeatures,numFilters,numClasses)
% This function is only for use in this example. It may be changed or
% removed in a future release.

% Initialize Layer 1 (1-D Convolutional)
filterSize1                     = 5;
numChannels1                    = numFeatures;
numFilters1                     = numFilters;

numIn1                          = filterSize1*numFilters1;
numOut1                         = filterSize1*numFilters1;

parameters.conv1.Weights        = initializeGlorot([filterSize1,numChannels1,numFilters1],numOu
```

```
parameters.conv1.Bias            = dlarray(zeros([numFilters1,1],'single'));
parameters.batchnorm1.Offset     = dlarray(zeros([numFilters1,1],'single'));
parameters.batchnorm1.Scale      = dlarray(ones([numFilters1,1],'single'));
state.batchnorm1.TrainedMean     = zeros(numFilters1,1,'single');
state.batchnorm1.TrainedVariance = ones(numFilters1,1,'single');


% Initialize Layer 2 (1-D Convolutional)
filterSize2                      = 3;
numChannels2                     = numFilters1;
numFilters2                      = numFilters;

numIn2                           = filterSize2*numFilters2;
numOut2                          = filterSize2*numFilters2;

parameters.conv2.Weights         = initializeGlorot([filterSize2,numChannels2,numFilters2],numOu
parameters.conv2.Bias            = dlarray(zeros([numFilters2,1],'single'));
parameters.batchnorm2.Offset     = dlarray(zeros([numFilters2,1],'single'));
parameters.batchnorm2.Scale      = dlarray(ones([numFilters2,1],'single'));
state.batchnorm2.TrainedMean     = zeros(numFilters2,1,'single');
state.batchnorm2.TrainedVariance = ones(numFilters2,1,'single');


% Initialize Layer 3 (1-D Convolutional)
filterSize3                      = 3;
numChannels3                     = numFilters2;
numFilters3                      = numFilters;

numIn3                           = filterSize3*numFilters3;
numOut3                          = filterSize3*numFilters3;

parameters.conv3.Weights         = initializeGlorot([filterSize3,numChannels3,numFilters3],numOu
parameters.conv3.Bias            = dlarray(zeros([numFilters3,1],'single'));
parameters.batchnorm3.Offset     = dlarray(zeros([numFilters3,1],'single'));
parameters.batchnorm3.Scale      = dlarray(ones([numFilters3,1],'single'));
state.batchnorm3.TrainedMean     = zeros(numFilters3,1,'single');
state.batchnorm3.TrainedVariance = ones(numFilters3,1,'single');


% Initialize Layer 4 (1-D Convolutional)
filterSize4                      = 1;
numChannels4                     = numFilters3;
numFilters4                      = numFilters;

numIn4                           = filterSize4*numFilters4;
numOut4                          = filterSize4*numFilters4;

parameters.conv4.Weights         = initializeGlorot([filterSize4,numChannels4,numFilters4],numOu
parameters.conv4.Bias            = dlarray(zeros([numFilters4,1],'single'));
parameters.batchnorm4.Offset     = dlarray(zeros([numFilters4,1],'single'));
parameters.batchnorm4.Scale      = dlarray(ones([numFilters4,1],'single'));
state.batchnorm4.TrainedMean     = zeros(numFilters4,1,'single');
state.batchnorm4.TrainedVariance = ones(numFilters4,1,'single');


% Initialize Layer 5 (1-D Convolutional)
filterSize5                      = 1;
numChannels5                     = numFilters4;
```

```
numFilters5                       = 1500;

numOut5                           = filterSize5*numFilters5;
numIn5                            = filterSize5*numFilters5;

parameters.conv5.Weights          = initializeGlorot([filterSize5,numChannels5,numFilters5],numOu
parameters.conv5.Bias             = dlarray(zeros([numFilters5,1],'single'));
parameters.batchnorm5.Offset      = dlarray(zeros([numFilters5,1],'single'));
parameters.batchnorm5.Scale       = dlarray(ones([numFilters5,1],'single'));
state.batchnorm5.TrainedMean      = zeros(numFilters5,1,'single');
state.batchnorm5.TrainedVariance = ones(numFilters5,1,'single');


% Initialize Layer 6 (Statistical Pooling)
numIn6                            = numOut5;
numOut6                           = 2*numIn6;


% Initialize Layer 7 (Fully Connected)
numIn7                            = numOut6;
numOut7                           = numFilters;

parameters.fc7.Weights            = initializeGlorot([numFilters,numIn7],numOut7,numIn7);
parameters.fc7.Bias               = dlarray(zeros([numOut7,1],'single'));
parameters.batchnorm7.Offset      = dlarray(zeros([numOut7,1],'single'));
parameters.batchnorm7.Scale       = dlarray(ones([numOut7,1],'single'));
state.batchnorm7.TrainedMean      = zeros(numOut7,1,'single');
state.batchnorm7.TrainedVariance = ones(numOut7,1,'single');


% Initialize Layer 8 (Fully Connected)
numIn8                            = numOut7;
numOut8                           = numFilters;

parameters.fc8.Weights            = initializeGlorot([numOut8,numIn8],numOut8,numIn8);
parameters.fc8.Bias               = dlarray(zeros([numOut8,1],'single'));
parameters.batchnorm8.Offset      = dlarray(zeros([numOut8,1],'single'));
parameters.batchnorm8.Scale       = dlarray(ones([numOut8,1],'single'));
state.batchnorm8.TrainedMean      = zeros(numOut8,1,'single');
state.batchnorm8.TrainedVariance = ones(numOut8,1,'single');


% Initialize Layer 9 (Fully Connected)
numIn9                            = numOut8;
numOut9                           = numClasses;

parameters.fc9.Weights            = initializeGlorot([numOut9,numIn9],numOut9,numIn9);
parameters.fc9.Bias               = dlarray(zeros([numOut9,1],'single'));
end
```

Initialize Weights Using Glorot Initialization

```
function weights = initializeGlorot(sz,numOut,numIn)
% This function is only for use in this example. It may be changed or
% removed in a future release.
Z = 2*rand(sz,'single') - 1;
bound = sqrt(6 / (numIn + numOut));
weights = bound*Z;
```

```
weights = dlarray(weights);
end
```

Calculate Model Gradients and Updated State

```
function [gradients,state,loss,Y] = modelGradients(X,target,parameters,state)
% This function is only for use in this example. It may be changed or
% removed in a future release.
[Y,state] = xvecModel(X,parameters,state,'DoTraining',true);
loss = crossentropy(Y,target);
gradients = dlgradient(loss,parameters);
end
```

Preprocess Mini-Batch

```
function [sequences,labels] = preprocessMiniBatch(sequences,labels)
% This function is only for use in this example. It may be changed or
% removed in a future release.
lengths = cellfun(@(x)size(x,1),sequences);
minLength = min(lengths);
sequences = cellfun(@(x)randomTruncate(x,1,minLength),sequences,'UniformOutput',false);
sequences = cat(3,sequences{:});

labels = cat(2,labels{:});
labels = onehotencode(labels,1);
labels(isnan(labels)) = 0;
end
```

Randomly Truncate Audio Signals to Specified Length

```
function y = randomTruncate(x,dim,minLength)
% This function is only for use in this example. It may be changed or
% removed in a future release.
N = size(x,dim);
if N > minLength
    start = randperm(N-minLength,1);
    if dim==1
        y = x(start:start+minLength-1,:);
    elseif dim ==2
        y = x(:,start:start+minLength-1);
    end
else
    y = x;
end
end
```

Feature Extraction and Normalization - Datastore

```
function [features,labels] = xVectorPreprocessBatch(ads,afe,nvargs)
% This function is only for use in this example. It may be changed or
% removed in a future release.
    arguments
        ads
        afe
        nvargs.Factors = []
        nvargs.Segment = true;
    end
    if ~isempty(ver('parallel'))
        pool = gcp;
```

```
        numpar = numpartitions(ads,pool);
    else
        numpar = 1;
    end
    labels = [];
    features = [];
    parfor ii = 1:numpar
        adsPart = partition(ads,numpar,ii);
        numFiles = numel(adsPart.UnderlyingDatastores{1}.Files);
        localFeatures = cell(numFiles,1);
        localLabels = [];
        for jj = 1:numFiles
            [audioIn,xInfo] = read(adsPart);
            label = xInfo.Label;
            [f,ns] = xVectorPreprocess(audioIn,afe,'Factors',nvargs.Factors,'Segment',nvargs.Segm
            localFeatures{jj} = f;
            localLabels = [localLabels,repelem(label,ns)];
        end
        features = [features;localFeatures];
        labels = [labels,localLabels];
    end
    features = cat(1,features{:});
    labels = removecats(labels);
end
```

Feature Extraction and Normalization

```
function [features,numSegments] = xVectorPreprocess(audioData,afe,nvargs)
% This function is only for use in this example. It may be changed or
% removed in a future release.
arguments
    audioData
    afe
    nvargs.Factors = []
    nvargs.Segment = true;
    nvargs.MinimumDuration = 1;
end
% Scale
audioData = audioData/max(abs(audioData(:)));

% Protect against NaNs
audioData(isnan(audioData)) = 0;

% Determine regions of speech
mergeDur = 0.5; % seconds
idx = detectSpeech(audioData,afe.SampleRate,'MergeDistance',afe.SampleRate*mergeDur);

% If a region is less than MinimumDuration seconds, drop it.
if nvargs.Segment
    idxToRemove = (idx(:,2)-idx(:,1))<afe.SampleRate*nvargs.MinimumDuration;
    idx(idxToRemove,:) = [];
end

% Extract features
numSegments = size(idx,1);
features = cell(numSegments,1);
for ii = 1:numSegments
    features{ii} = single(extract(afe,audioData(idx(ii,1):idx(ii,2))));
```

```
end

% Standardize features
if ~isempty(nvargs.Factors)
    features = cellfun(@(x)(x-nvargs.Factors.Mean)./nvargs.Factors.STD,features,'UniformOutput',
end

% Cepstral mean subtraction (for channel noise)
if ~isempty(nvargs.Factors)
    fileMean = mean(cat(1,features{:}),'all');
    features = cellfun(@(x)x - fileMean,features,'UniformOutput',false);
end

if ~nvargs.Segment
    features = cat(1,features{:});
end
end
```

Speaker Diarization Using x-vectors

Speaker diarization is the process of partitioning an audio signal into segments according to speaker identity. It answers the question "who spoke when" without prior knowledge of the speakers and, depending on the application, without prior knowledge of the number of speakers.

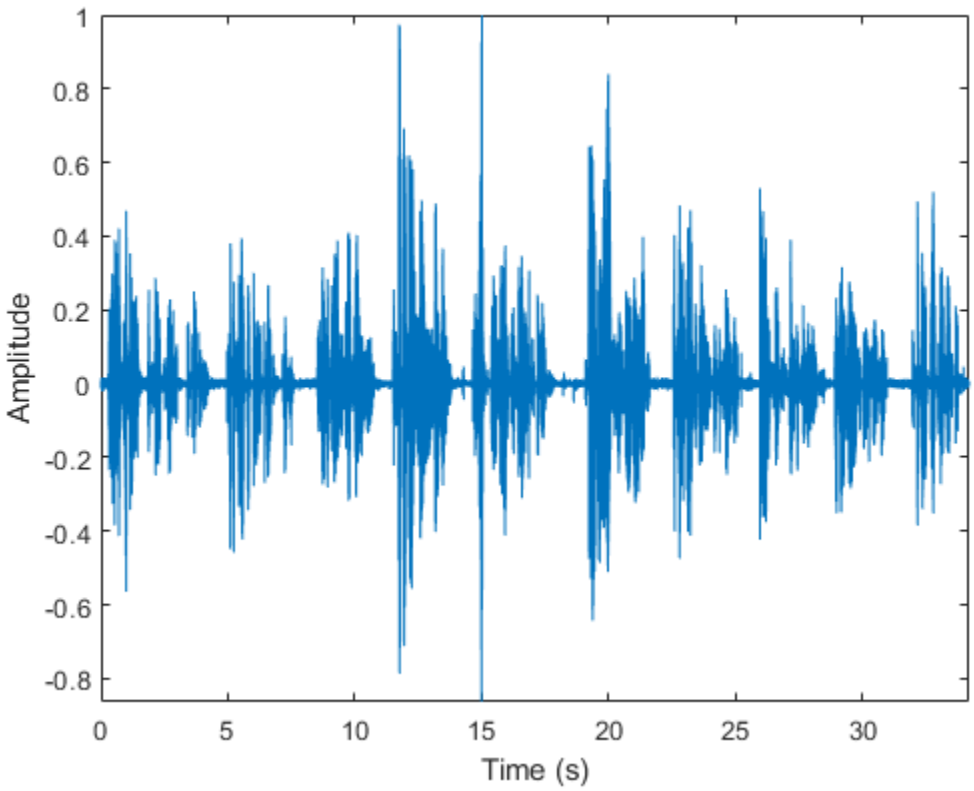
Speaker diarization has many applications, including: enhancing speech transcription by structuring text according to active speaker, video captioning, content retrieval (*what did Jane say?*) and speaker counting (*how many speakers were present in the meeting?*).

In this example, you perform speaker diarization using a pretrained x-vector system [1] on page 14-0 to characterize regions of audio and agglomerative hierarchical clustering (AHC) to group similar regions of audio [2] on page 14-0 . To see how the x-vector system was defined and trained, see Speaker Recognition Using x-vectors.

Load Audio Signal

Load an audio signal and a table containing ground truth annotations. The signal consists of five speakers. Listen to the audio signal and plot its time-domain waveform.

```
[audioIn,fs] = audioread('exampleconversation.flac');
load('exampleconversationlabels.mat')
audioIn = audioIn./max(abs(audioIn));
sound(audioIn,fs)

t = (0:size(audioIn,1)-1)/fs;

figure(1)
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
axis tight
```



Extract x-vectors

In this example, you used a pretrained x-vector system. The x-vector system is a lightweight version of the original x-vector system described in [1] on page 14-0 . To see how the x-vector system was defined and trained, see Speaker Recognition Using x-vectors.

Load Pretrained x-Vector System

Load the lightweight pretrained x-vector system. The x-vector system consists of:

- `afe` - an `audioFeatureExtractor` (Audio Toolbox) object to extract mel frequency cepstral coefficients (MFCCs).
- `factors` - a struct containing the mean and standard deviation of MFCCs determined from a representative data set. These factors are used to standardize the MFCCs.
- `extractor` - a struct containing the parameters and state of the neural network trained to extract x-vectors. The `xvecModel` function performs the actual x-vector extraction. The `xvecModel` function is placed in your current folder when you open this example.
- `classifier` - a struct containing a trained projection matrix to reduce the dimensionality of x-vectors and a trained PLDA model for scoring x-vectors.

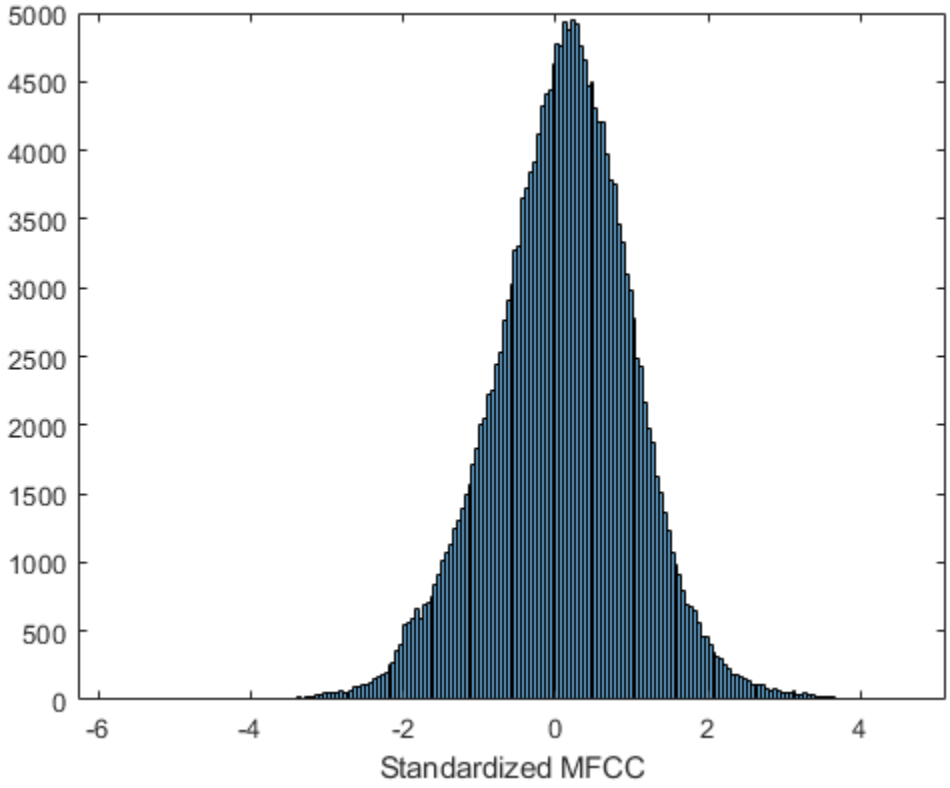
```
load('xvectorSystem.mat')
```

Extract Standardized Acoustic Features

Extract standardized MFCC features from the audio data. View the feature distributions to confirm that the standardization factors learned from a separate data set approximately standardize the

features derived in this example. A standard distribution has a mean of zero and a standard deviation of 1.

```
features = single((extract(afe,audioIn)-factors.Mean)./factors.STD);

figure(2)
histogram(features)
xlabel('Standardized MFCC')
```



Extract x-Vectors

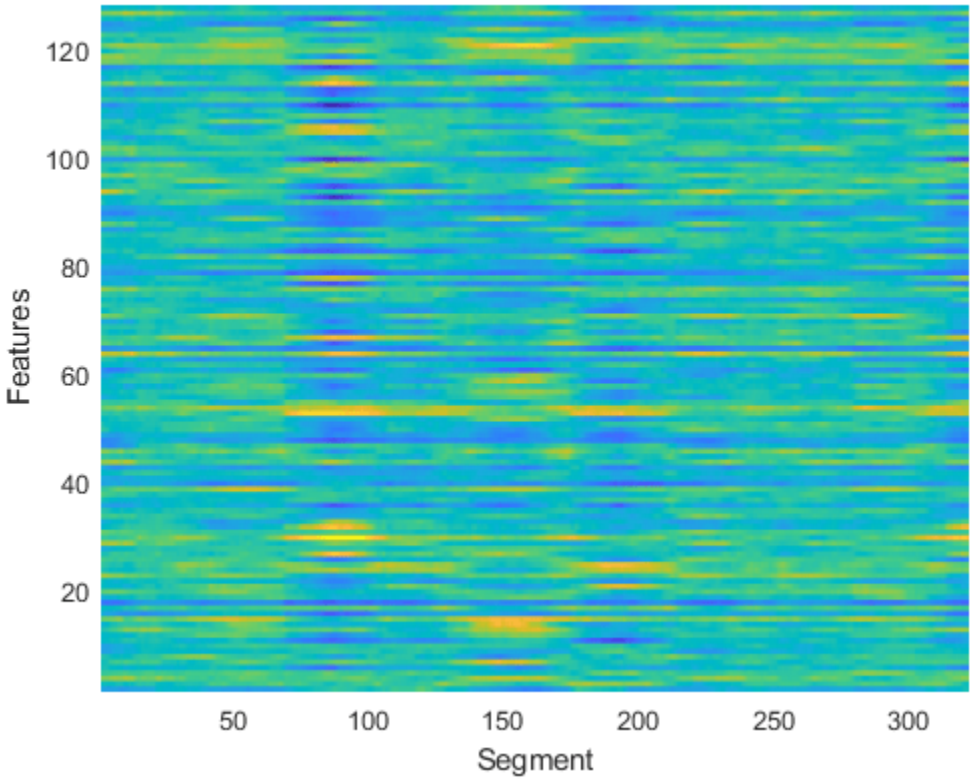
Each acoustic feature vector represents approximately 0.01 seconds of audio data. Group the features into approximately 2 second segments with 0.1 second hops between segments.

```
featureVectorHopDur = (numel(afe.Window) - afe.OverlapLength)/afe.SampleRate;

segmentDur = 2 ;
segmentHopDur = 0.1 ;

segmentLength = round(segmentDur/featureVectorHopDur);
segmentHop = round(segmentHopDur/featureVectorHopDur);

idx = 1:segmentLength;
featuresSegmented = [];
while idx(end) < size(features,1)
    featuresSegmented = cat(3,featuresSegmented,features(idx,:));
```

```
    idx = idx + segmentHop;
end
```

Extract x-vectors from each segment. x-vectors correspond to the output from the 7th layer in the x-vector model trained in Speaker Recognition Using x-vectors. The 7th layer is the first segment-level layer after statistics are calculated for the time-dilated frame-level layers. Visualize the x-vectors over time.

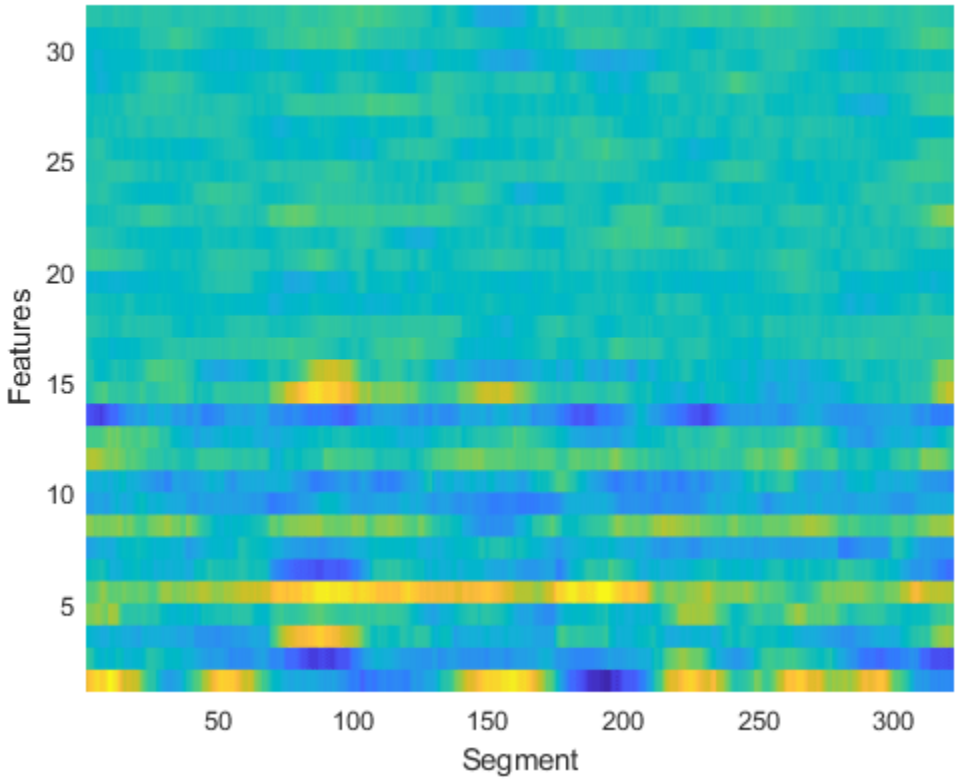
```
outputLayer = 7;
xvecs = zeros(numel(extractor.Parameters.("fc"+outputLayer).Bias),size(featuresSegmented,3));
for sample = 1:size(featuresSegmented,3)
    dlX = dlarray(featuresSegmented(:,:,sample),'SCB');
    xvecs(:,sample) = extractdata(xvecModel(dlX,extractor.Parameters,extractor.State,'DoTraining
end

figure(3)
surf(xvecs','EdgeColor','none')
view([90,-90])
axis([1 size(xvecs,1) 1 size(xvecs,2)])
xlabel('Features')
ylabel('Segment')
```



Apply the pretrained linear discriminant analysis (LDA) projection matrix to reduce the dimensionality of the x-vectors and then visualize the x-vectors over time.

```
x = classifier.projMat*xvecs;
```

```
figure(4)
surf(x','EdgeColor','none')
view([90,-90])
axis([1 size(x,1) 1 size(x,2)])
xlabel('Features')
ylabel('Segment')
```



Cluster x-vectors

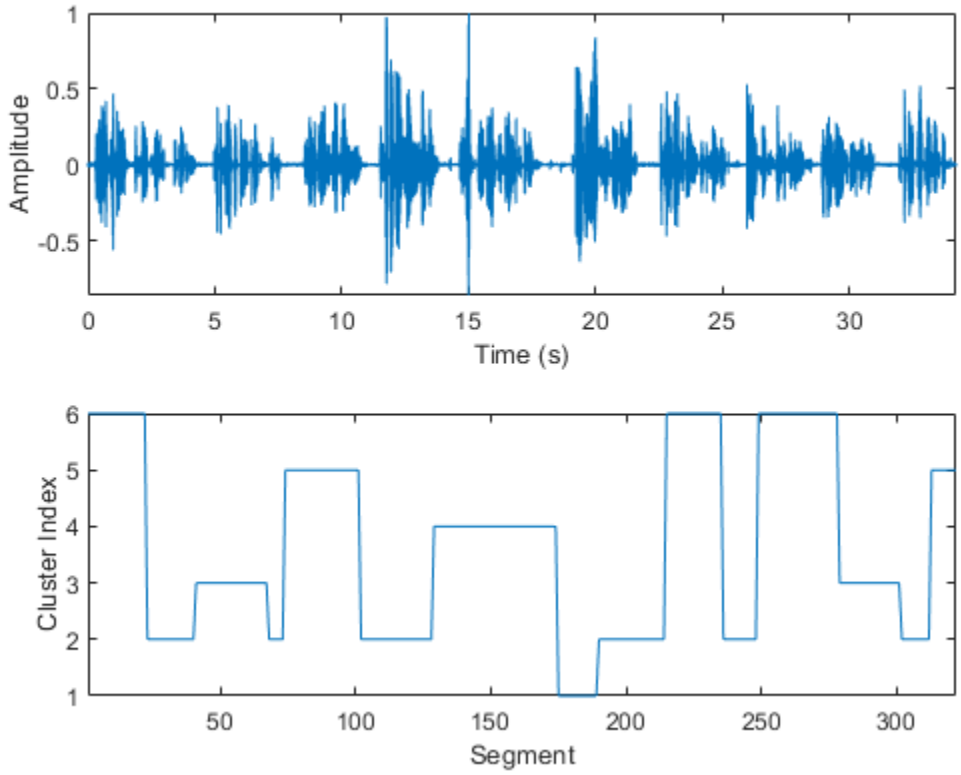
An x-vector system learns to extract compact representations (x-vectors) of speakers. Cluster the x-vectors to group similar regions of audio using either agglomerative hierarchical clustering (`clusterdata` (Statistics and Machine Learning Toolbox)) or k-means clustering (`kmeans` (Statistics and Machine Learning Toolbox)). [2] on page 14-0 suggests using agglomerative heirarchical clustering with PLDA scoring as the distance measurement. K-means clustering using a cosine similarity score is also commonly used. Assume prior knowledge of the the number of speakers in the audio. Set the maximum clusters to the number of known speakers + 1 so that the background is clustered independently.

```
knownNumberOfSpeakers = numel(unique(groundTruth.Label));
maxclusters = knownNumberOfSpeakers + 1;

clusterMethod = 'agglomerative - PL...';
switch clusterMethod
    case 'agglomerative - PLDA scoring'
        T = clusterdata(x','Criterion','distance','distance',@(a,b)helperPLDAScorer(a,b,classifie
    case 'agglomerative - CSS scoring'
        T = clusterdata(x','Criterion','distance','distance','cosine','linkage','average','maxclu
```

```
    case 'kmeans - CSS scoring'
        T = kmeans(x',maxclusters,'Distance','cosine');
end
```

Plot the cluster decisions over time.

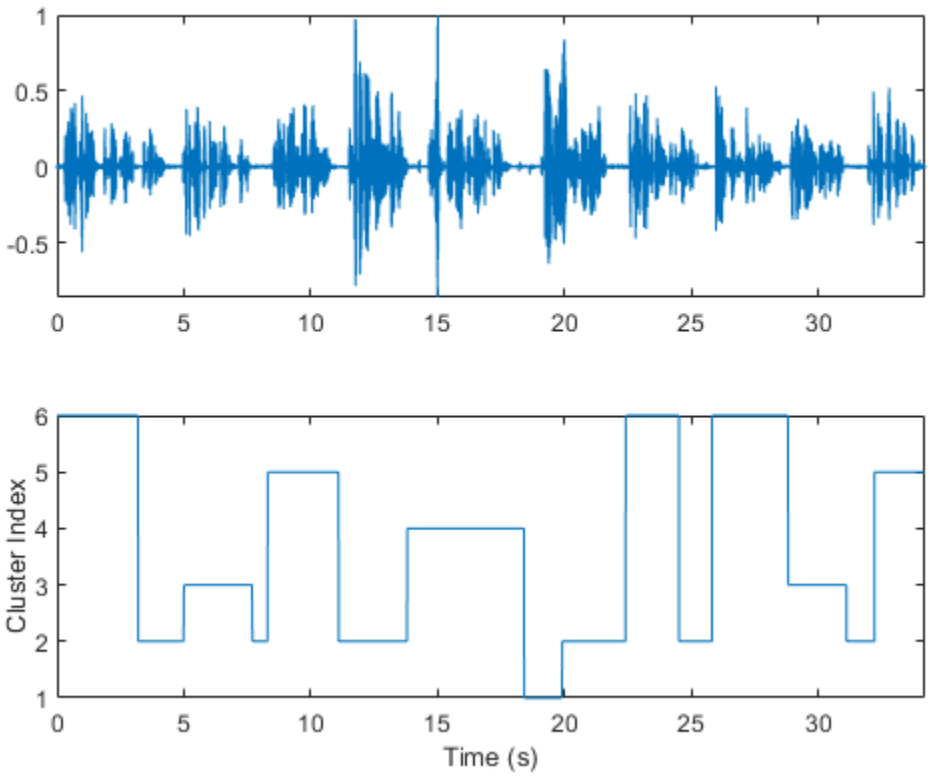
```
figure(5)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight
ylabel('Amplitude')
xlabel('Time (s)')

nexttile
plot(T)
axis tight
ylabel('Cluster Index')
xlabel('Segment')
```



To isolate segments of speech corresponding to clusters, map the segments back to audio samples. Plot the results.

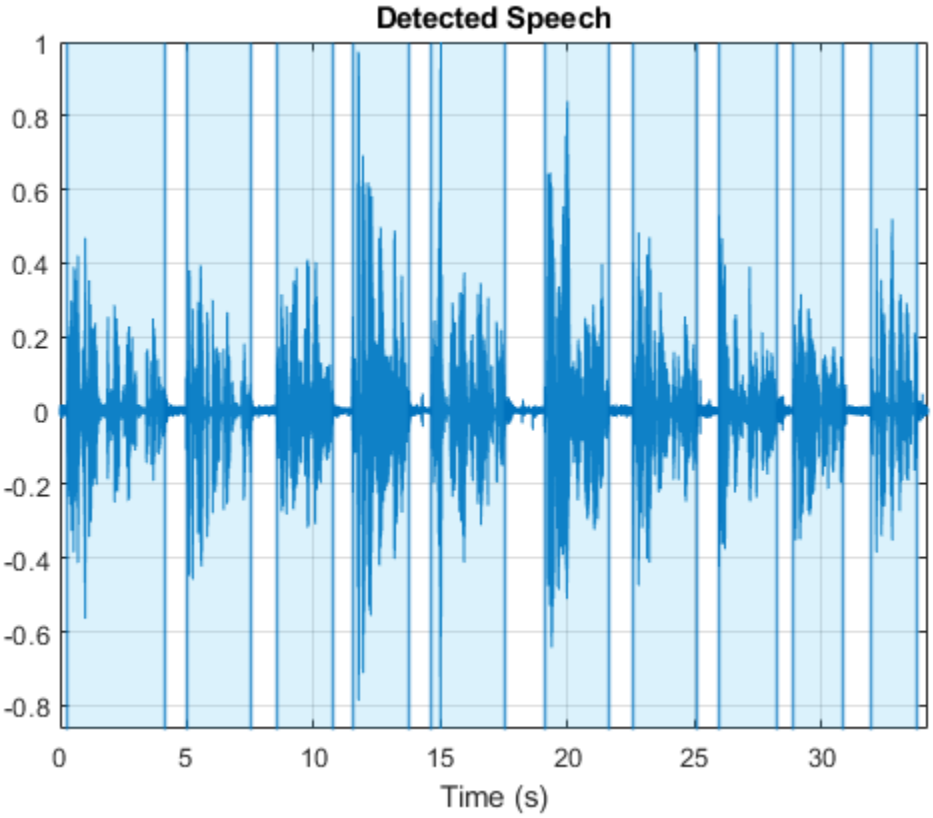
```
mask = zeros(size(audioIn,1),1);
start = round((segmentDur/2)*fs);

segmentHopSamples = round(segmentHopDur*fs);
```

```
mask(1:start) = T(1);
start = start + 1;
for ii = 1:numel(T)
    finish = start + segmentHopSamples;
    mask(start:start + segmentHopSamples) = T(ii);
    start = finish + 1;
end
mask(finish:end) = T(end);

figure(6)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight

nexttile
plot(t,mask)
ylabel('Cluster Index')
axis tight
xlabel('Time (s)')
```



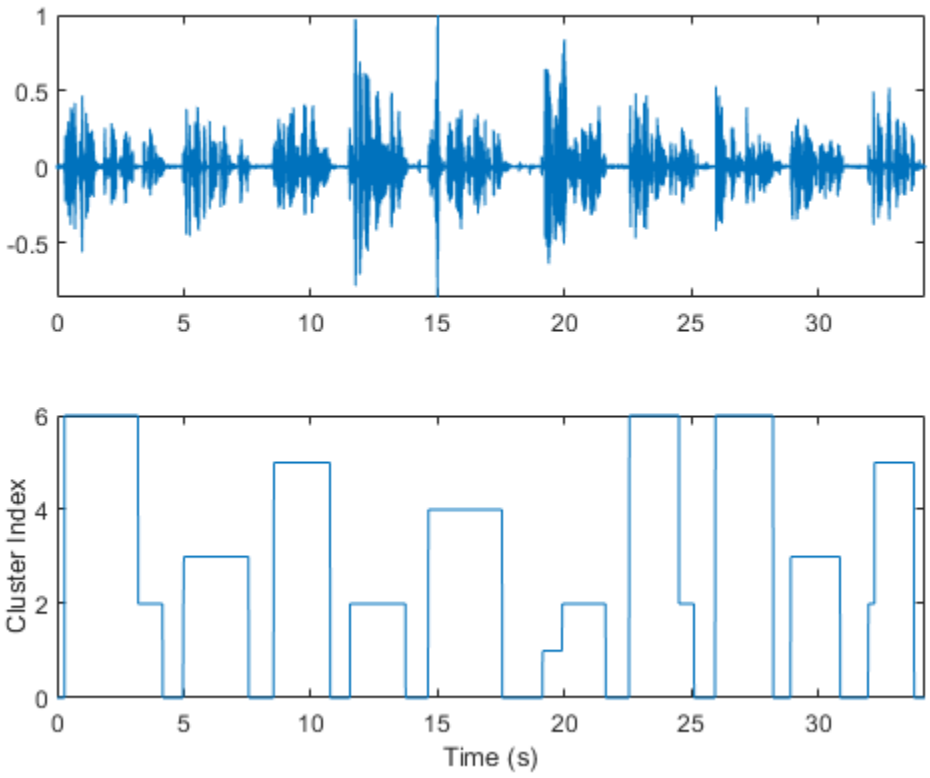
Use `detectSpeech` (Audio Toolbox) to determine speech regions. Use `sigroi2binmask` (Signal Processing Toolbox) to convert speech regions to a binary voice activity detection (VAD) mask. Call `detectSpeech` a second time without any arguments to plot the detected speech regions.

```
mergeDuration = 0.5;
VADidx = detectSpeech(audioIn,fs,'MergeDistance',fs*mergeDuration);

VADmask = sigroi2binmask(VADidx,numel(audioIn));

figure(7)
detectSpeech(audioIn,fs,'MergeDistance',fs*mergeDuration)
```



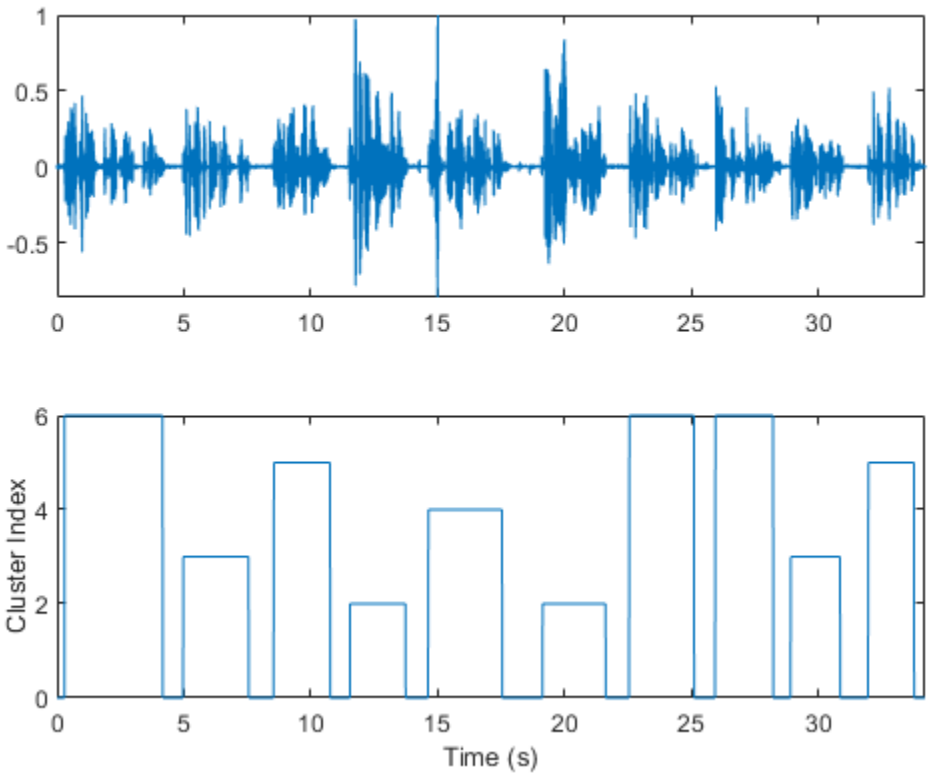
Apply the VAD mask to the speaker mask and plot the results. A cluster index of 0 indicates a region of no speech.

```
mask = mask.*VADmask;

figure(8)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight

nexttile
plot(t,mask)
ylabel('Cluster Index')
axis tight
xlabel('Time (s)')
```



In this example, you assume each detected speech region belongs to a single speaker. If more than two labels are present in a speech region, merge them to the most frequently occuring label.

```
maskLabels = zeros(size(VADidx,1),1);
for ii = 1:size(VADidx,1)
    maskLabels(ii) = mode(mask(VADidx(ii,1):VADidx(ii,2)),'all');
    mask(VADidx(ii,1):VADidx(ii,2)) = maskLabels(ii);
end

figure(9)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight

nexttile
plot(t,mask)
ylabel('Cluster Index')
axis tight
xlabel('Time (s)')
```



Count the number of remaining speaker clusters.

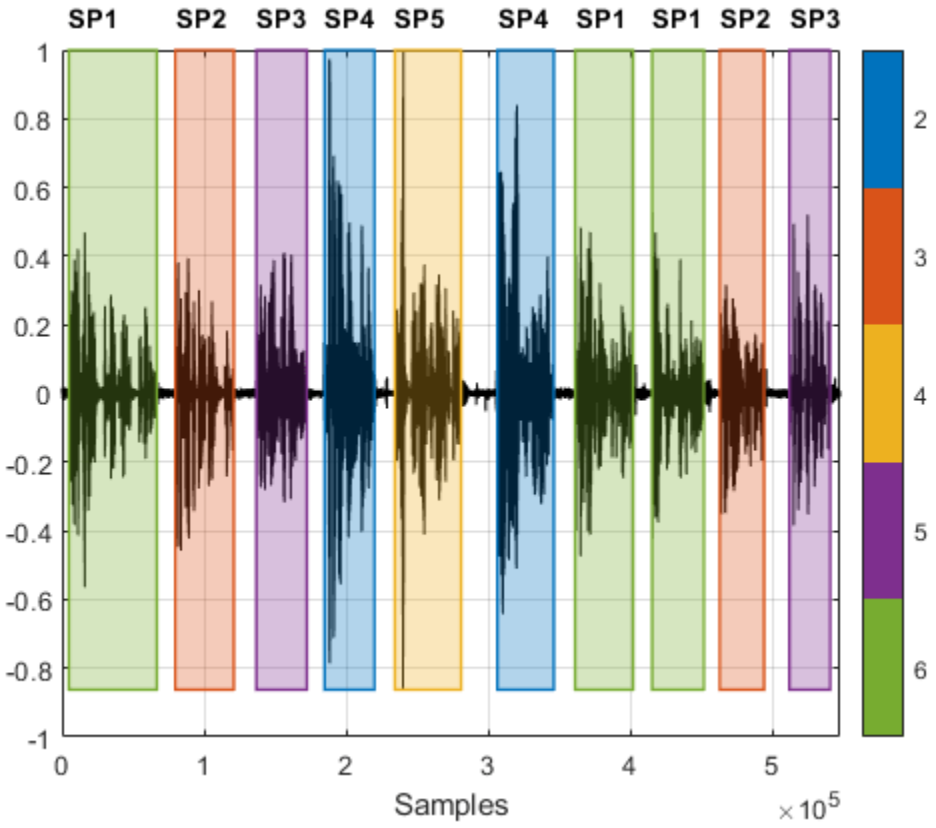
```
uniqueSpeakerClusters = unique(maskLabels);
numSpeakers = numel(uniqueSpeakerClusters)
```

```
numSpeakers = 5
```

Visualize Diarization Results

Create a `signalMask` (Signal Processing Toolbox) object and then plot the speaker clusters. Label the plot with the ground truth labels. The cluster labels are color coded with a key on the right of the plot. The true labels are printed above the plot.

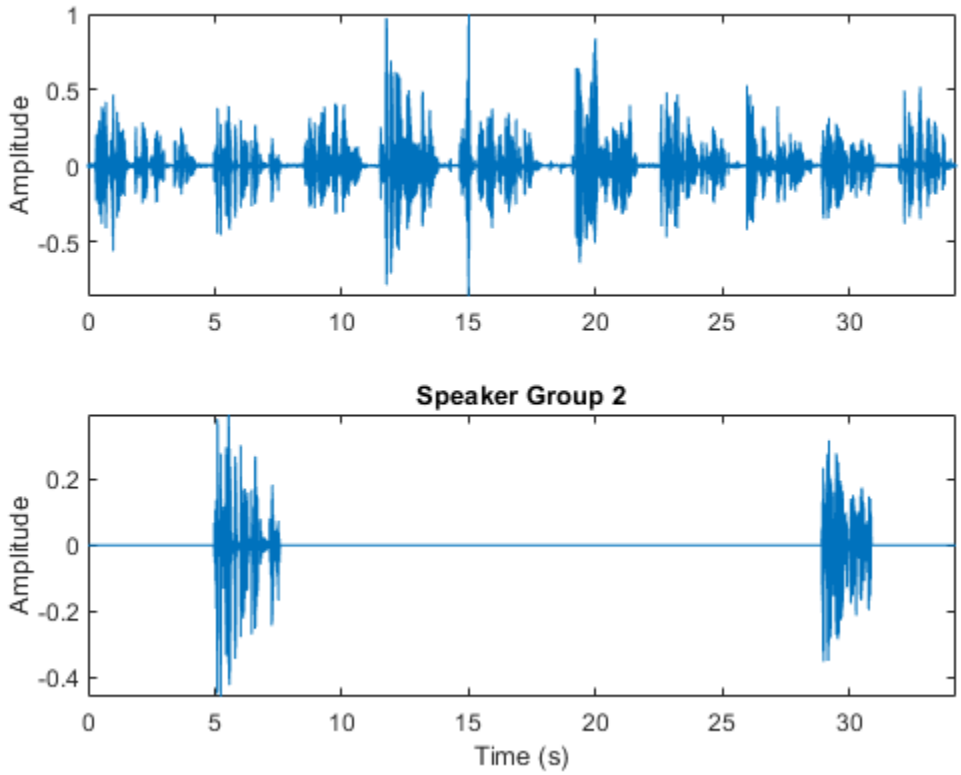
```
msk = signalMask(table(VADidx,categorical(maskLabels)));

figure(10)
plotsigroi(msk,audioIn,true)
axis([0 numel(audioIn) -1 1])

trueLabel = groundTruth.Label;
for ii = 1:numel(trueLabel)
    text(VADidx(ii,1),1.1,trueLabel(ii),'FontWeight','bold')
end
```



Choose a cluster to inspect and then use `binmask` (Signal Processing Toolbox) to isolate the speaker. Plot the isolated speech signal and listen to the speaker cluster.

```
speakerToInspect = 2;
cutOutSilenceFromAudio = true;

bmsk = binmask(msk,numel(audioIn));

audioToPlay = audioIn;
if cutOutSilenceFromAudio
    audioToPlay(~bmsk(:,speakerToInspect)) = [];
end
sound(audioToPlay,fs)

figure(11)
tiledlayout(2,1)

nexttile
plot(t,audioIn)
axis tight
ylabel('Amplitude')

nexttile
plot(t,audioIn.*bmsk(:,speakerToInspect))
axis tight
xlabel('Time (s)')
```

```
ylabel('Amplitude')
title("Speaker Group "+speakerToInspect)
```



Diarization System Evaluation

The common metric for speaker diarization systems is the diarization error rate (DER). The DER is the sum of the miss rate (classifying speech as non-speech), the false alarm rate (classifying non-speech as speech) and the speaker error rate (confusing one speaker's speech for another).

In this simple example, the miss rate and false alarm rate are trivial problems. You evaluate the speaker error rate only.

Map each true speaker to the corresponding best-fitting speaker cluster. To determine the speaker error rate, count the number of mismatches between the true speakers and the best-fitting speaker clusters, and then divide by the number of true speaker regions.

```
uniqueLabels = unique(trueLabel);
guessLabels = maskLabels;
uniqueGuessLabels = unique(guessLabels);

totalNumErrors = 0;
for ii = 1:numel(uniqueLabels)
    isSpeaker = uniqueLabels(ii)==trueLabel;
    minNumErrors = inf;

    for jj = 1:numel(uniqueGuessLabels)
        groupCandidate = uniqueGuessLabels(jj) == guessLabels;
```

```
        numErrors = nnz(isSpeaker-groupCandidate);
        if numErrors < minNumErrors
            minNumErrors = numErrors;
            bestCandidate = jj;
        end
        minNumErrors = min(minNumErrors,numErrors);
    end
    uniqueGuessLabels(bestCandidate) = [];
    totalNumErrors = totalNumErrors + minNumErrors;
end
SpeakerErrorRate = totalNumErrors/numel(trueLabel)
```

```
SpeakerErrorRate = 0
```

References

[1] Snyder, David, et al. “X-Vectors: Robust DNN Embeddings for Speaker Recognition.” *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2018, pp. 5329–33. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2018.8461375.

[2] Sell, G., Snyder, D., McCree, A., Garcia-Romero, D., Villalba, J., Maciejewski, M., Manohar, V., Dehak, N., Povey, D., Watanabe, S., Khudanpur, S. (2018) Diarization is Hard: Some Experiences and Lessons Learned for the JHU Team in the Inaugural DIHARD Challenge. Proc. Interspeech 2018, 2808-2812, DOI: 10.21437/Interspeech.2018-1893.

Train Spoken Digit Recognition Network Using Out-of-Memory Audio Data

This example trains a spoken digit recognition network on out-of-memory audio data using a transformed datastore. In this example, you apply a random pitch shift to audio data used to train a convolutional neural network (CNN). For each training iteration, the audio data is augmented using the audioDataAugmenter object and then features are extracted using the audioFeatureExtractor object. The workflow in this example applies to any random data augmentation used in a training loop. The workflow also applies when the underlying audio data set or training features do not fit in memory.

Data

Download the Free Spoken Digit Data Set (FSDD). FSDD consists of 2000 recordings of four speakers saying the numbers 0 through 9 in English.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/FSDD.zip';

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'FSDD');

if ~exist(datasetFolder,'dir')
    disp('Downloading FSDD...')
    unzip(url,downloadFolder)
end
```

Create an `audioDatastore` that points to the dataset.

```
pathToRecordingsFolder = fullfile(datasetFolder,'recordings');
location = pathToRecordingsFolder;
ads = audioDatastore(location);
```

The helper function, `helperGenerateLabels`, creates a categorical array of labels from the FSDD files. The source code for `helpergenLabels` is listed in the appendix. Display the classes and the number of examples in each class.

```
ads.Labels = helpergenLabels(ads);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

```
summary(ads.Labels)
```

```
     0      200
     1      200
     2      200
     3      200
     4      200
     5      200
     6      200
     7      200
     8      200
     9      200
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. You use the training set to train the model and the test set to validate the trained model.

```
rng default
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       160
      1       160
      2       160
      3       160
      4       160
      5       160
      6       160
      7       160
      8       160
      9       160
```

```
countEachLabel(adsTest)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       40
      1       40
      2       40
      3       40
      4       40
      5       40
      6       40
      7       40
      8       40
      9       40
```

Reduce Training Dataset

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to false. To run this example quickly, set `reduceDataset` to true.

```
reduceDataset = false;
if reduceDataset == "true"
    adsTrain = splitEachLabel(adsTrain,2);
    adsTest = splitEachLabel(adsTest,2);
end
```

Transformed Training Datastore

Data Augmentation

Augment the training data by applying pitch shifting with an `audioDataAugmenter` object.

Create an `audioDataAugmenter`. The augmenter applies pitch shifting on an input audio signal with a 0.5 probability. The augmenter selects a random pitch shifting value in the range [-12 12] semitones.

```
augmenter = audioDataAugmenter('PitchShiftProbability',.5, ...
                               'SemitoneShiftRange',[-12 12], ...
                               'TimeShiftProbability',0, ...
                               'VolumeControlProbability',0, ...
                               'AddNoiseProbability',0, ...
                               'TimeShiftProbability',0);
```

Set custom pitch-shifting parameters. Use identity phase locking and preserve formants using spectral envelope estimation with 30th order cepstral analysis.

```
setAugmenterParams(augmenter,'shiftPitch','LockPhase',true,'PreserveFormants',true,'CepstralOrde
```

Create a transformed datastore that applies data augmentation to the training data.

```
fs = 8000;
adsAugTrain = transform(adsTrain,@(y)deal(augment(augmenter,y,fs).Audio{1}));
```

Mel Spectrogram Feature Extraction

The CNN accepts mel-frequency spectrograms.

Define parameters used to extract mel-frequency spectrograms. Use 220 ms windows with 10 ms hops between windows. Use a 2048-point DFT and 40 frequency bands.

```
frameDuration = 0.22;
hopDuration = 0.01;
params.segmentLength = 8192;
segmentDuration = params.segmentLength*(1/fs);
params.numHops = ceil((segmentDuration-frameDuration)/hopDuration);
params.numBands = 40;
frameLength = round(frameDuration*fs);
hopLength = round(hopDuration*fs);
fftLength = 2048;
```

Create an `audioFeatureExtractor` object to compute mel-frequency spectrograms from input audio signals.

```
afe = audioFeatureExtractor('melSpectrum',true,'SampleRate',fs);
afe.Window = hamming(frameLength,'periodic');
afe.OverlapLength = frameLength-hopLength;
afe.FFTLength = fftLength;
```

Set the parameters for the mel-frequency spectrogram.

```
setExtractorParams(afe,'melSpectrum','NumBands',params.numBands,'FrequencyRange',[50 fs/2],'Windo
```

Create a transformed datastore that computes mel-frequency spectrograms from pitch-shifted audio data. The helper function, `getSpeechSpectrogram` (see appendix), standardizes the recording length and normalizes the amplitude of the audio input. `getSpeechSpectrogram` uses the `audioFeatureExtractor` object (afe) to obtain the log-based mel-frequency spectrograms.

```
adsSpecTrain = transform(adsAugTrain, @(x)getSpeechSpectrogram(x,afe,params));
```

Training Labels

Use an `arrayDatastore` to hold the training labels.

```
labelsTrain = arrayDatastore(adsTrain.Labels);
```

Combined Training Datastore

Create a combined datastore that points to the mel-frequency spectrogram data and the corresponding labels.

```
tdsTrain = combine(adsSpecTrain,labelsTrain);
```

Validation Data

The validation dataset fits into memory and you precompute validation features using the helper function `getValidationSpeechSpectrograms` (see appendix).

```
XTest = getValidationSpeechSpectrograms(adsTest,afe,params);
```

Get the validation labels.

```
YTest = adsTest.Labels;
```

Define CNN Architecture

Construct a small CNN as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps using max pooling layers. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

```
sz = size(XTest);
specSize = sz(1:2);
imageSize = [specSize 1];

numClasses = numel(categories(YTest));

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize,'Normalization','none')

    convolution2dLayer(5,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
```

```
    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2)

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',categories(YTest));
    ];
```

Set the hyperparameters to use in training the network. Use a mini-batch size of 50 and a learning rate of 1e-4. Specify 'adam' optimization. To use the parallel pool to read the transformed datastore, set `DispatchInBackground` to `true`. For more information, see `trainingOptions`.

```
miniBatchSize = 50;
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-4, ...
    'MaxEpochs',30, ...
    'LearnRateSchedule',"piecewise",...
    'LearnRateDropFactor',.1,...
    'LearnRateDropPeriod',15,...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{XTest, YTest},...
    'ValidationFrequency',ceil(numel(adsTrain.Files)/miniBatchSize),...
    'ExecutionEnvironment','gpu',...
    'DispatchInBackground', true);
```

Train the network by passing the transformed training datastore to `trainNetwork`.

```
trainedNet = trainNetwork(tdsTrain,layers,options);
```

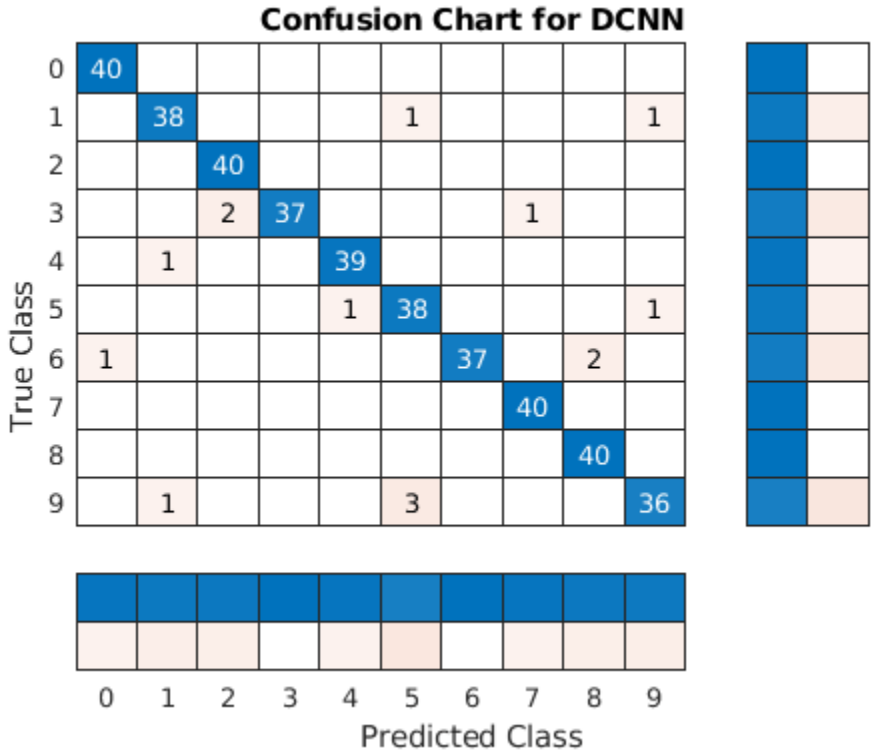
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest);
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 96.2500
```

Summarize the performance of the trained network on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccDCNN = confusionchart(YTest,Ypredicted);
ccDCNN.Title = 'Confusion Chart for DCNN';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```



Appendix: Helper Functions

```
function Labels = helpergenLabels(ads)
% This function is only for use in this example. It may be changed or
% removed in a future release.
files = ads.Files;
tmp = cell(numel(files),1);
expression = "[0-9]+_";
parfor nf = 1:numel(ads.Files)
    idx = regexp(files{nf},expression);
    tmp{nf} = files{nf}(idx);
end
Labels = categorical(tmp);
end

%-----------------------------------------------------------
function X = getValidationSpeechSpectrograms(ads,afe,params)
% This function is only for use in this example. It may changed or be
% removed in a future release.
%
% getValidationSpeechSpectrograms(ads,afe) computes speech spectrograms for
% the files in the datastore ads using the audioFeatureExtractor afe.

numFiles = length(ads.Files);
X = zeros([params.numBands,params.numHops,1,numFiles],'single');

for i = 1:numFiles
    x = read(ads);
    spec = getSpeechSpectrogram(x,afe,params);
    X(:,:,1,i) = spec;
```

```
end

end

%--------------------------------------------------------------------------
function X = getSpeechSpectrogram(x,afe,params)
% This function is only for use in this example. It may changed or be
% removed in a future release.
%
% getSpeechSpectrogram(x,afe) computes a speech spectrogram for the signal
% x using the audioFeatureExtractor afe.

X = zeros([params.numBands,params.numHops],'single');

x = normalizeAndResize(single(x),params);

spec = extract(afe,x).';

% If the spectrogram is less wide than numHops, then put spectrogram in
% the middle of X.
w = size(spec,2);
left = floor((params.numHops-w)/2)+1;
ind = left:left+w-1;
X(:,ind) = log10(spec + 1e-6);

end
%--------------------------------------------------------------------------
function x = normalizeAndResize(x,params)
% This function is only for use in this example. It may change or be
% removed in a future release.

L = params.segmentLength;
N = numel(x);
if N > L
    x = x(1:L);
elseif N < L
    pad = L-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));
end
```

Train Spoken Digit Recognition Network Using Out-of-Memory Features

This example trains a spoken digit recognition network on out-of-memory auditory spectrograms using a transformed datastore. In this example, you extract auditory spectrograms from audio using `audioDatastore` and `audioFeatureExtractor`, and you write them to disk. You then use a `signalDatastore` to access the features during training. The workflow is useful when the training features do not fit in memory. In this workflow, you only extract features once, which speeds up your workflow if you are iterating on the deep learning model design.

Data

Download the Free Spoken Digit Data Set (FSDD). FSDD consists of 2000 recordings of four speakers saying the numbers 0 through 9 in English.

```
url = 'https://ssd.mathworks.com/supportfiles/audio/FSDD.zip';

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'FSDD');

if ~exist(datasetFolder,'dir')
    disp('Downloading FSDD...')
    unzip(url,downloadFolder)
end
```

Create an `audioDatastore` that points to the dataset.

```
pathToRecordingsFolder = fullfile(datasetFolder,'recordings');
location = pathToRecordingsFolder;
ads = audioDatastore(location);
```

The helper function, `helperGenerateLabels`, creates a categorical array of labels from the FSDD files. The source code for `helperGenerateLabels` is listed in the appendix. Display the classes and the number of examples in each class.

```
ads.Labels = helperGenerateLabels(ads);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 8).
```

```
summary(ads.Labels)
```

```
     0      200
     1      200
     2      200
     3      200
     4      200
     5      200
     6      200
     7      200
     8      200
     9      200
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. You use the training set to train the model and the test set to validate the trained model.

```
rng default
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       160
      1       160
      2       160
      3       160
      4       160
      5       160
      6       160
      7       160
      8       160
      9       160
```

```
countEachLabel(adsTest)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       40
      1       40
      2       40
      3       40
      4       40
      5       40
      6       40
      7       40
      8       40
      9       40
```

Reduce Training Dataset

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to false. To run this example quickly, set `reduceDataset` to true.

```
reduceDataset = false;
if reduceDataset == "true"
    adsTrain = splitEachLabel(adsTrain,2);
    adsTest = splitEachLabel(adsTest,2);
end
```

Set up Auditory Spectrogram Extraction

The CNN accepts mel-frequency spectrograms.

Define parameters used to extract mel-frequency spectrograms. Use 220 ms windows with 10 ms hops between windows. Use a 2048-point DFT and 40 frequency bands.

```
fs = 8000;
frameDuration = 0.22;
```

```
hopDuration = 0.01;
params.segmentLength = 8192;
segmentDuration = params.segmentLength*(1/fs);
params.numHops = ceil((segmentDuration-frameDuration)/hopDuration);
params.numBands = 40;
frameLength = round(frameDuration*fs);
hopLength = round(hopDuration*fs);
fftLength = 2048;
```

Create an `audioFeatureExtractor` object to compute mel-frequency spectrograms from input audio signals.

```
afe = audioFeatureExtractor('melSpectrum',true,'SampleRate',fs);
afe.Window = hamming(frameLength,'periodic');
afe.OverlapLength = frameLength-hopLength;
afe.FFTLength = fftLength;
```

Set the parameters for the mel-frequency spectrogram.

```
setExtractorParams(afe,'melSpectrum','NumBands',params.numBands,'FrequencyRange',[50 fs/2],'Windo
```

Create a transformed datastore that computes mel-frequency spectrograms from audio data. The helper function, `getSpeechSpectrogram` (see appendix), standardizes the recording length and normalizes the amplitude of the audio input. `getSpeechSpectrogram` uses the `audioFeatureExtractor` object `afe` to obtain the log-based mel-frequency spectrograms.

```
adsSpecTrain = transform(adsTrain,@(x)getSpeechSpectrogram(x,afe,params));
```

Write Auditory Spectrograms to Disk

Use `writeall` to write auditory spectrograms to disk. Set `UseParallel` to true to perform writing in parallel.

```
writeall(adsSpecTrain,pwd,'WriteFcn',@myCustomWriter,'UseParallel',true);
```

Set up Training Signal Datastore

Create a `signalDatastore` that points to the out-of-memory features. The custom read function returns a spectrogram/label pair.

```
sds = signalDatastore('recordings','ReadFcn',@myCustomRead);
```

Validation Data

The validation dataset fits into memory and you precompute validation features using the helper function `getValidationSpeechSpectrograms` (see appendix).

```
XTest = getValidationSpeechSpectrograms(adsTest,afe,params);
```

Get the validation labels.

```
YTest = adsTest.Labels;
```

Define CNN Architecture

Construct a small CNN as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps using max pooling layers. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

```
sz = size(XTest);
specSize = sz(1:2);
imageSize = [specSize 1];

numClasses = numel(categories(YTest));

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize,'Normalization','none')

    convolution2dLayer(5,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2)

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',categories(YTest));
    ];
```

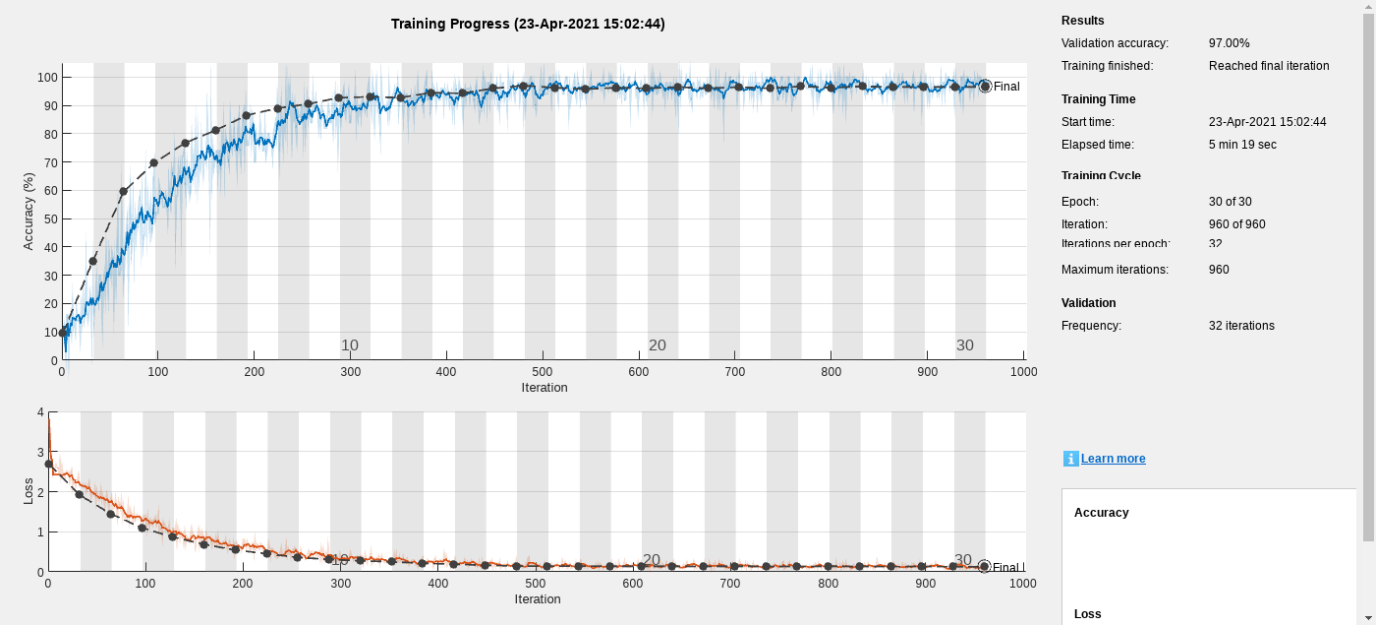
Set the hyperparameters to use in training the network. Use a mini-batch size of 50 and a learning rate of 1e-4. Specify 'adam' optimization. To use the parallel pool to read the transformed datastore, set `DispatchInBackground` to `true`. For more information, see `trainingOptions`.

```
miniBatchSize = 50;
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-4, ...
    'MaxEpochs',30, ...
    'LearnRateSchedule',"piecewise",...
    'LearnRateDropFactor',.1,...
    'LearnRateDropPeriod',15,...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
```

```
    'Verbose',false, ...
    'ValidationData',{XTest, YTest},...
    'ValidationFrequency',ceil(numel(adsTrain.Files)/miniBatchSize),...
    'ExecutionEnvironment','gpu',...
    'DispatchInBackground', true);
```

Train the network by passing the training datastore to `trainNetwork`.

```
trainedNet = trainNetwork(sds,layers,options);
```



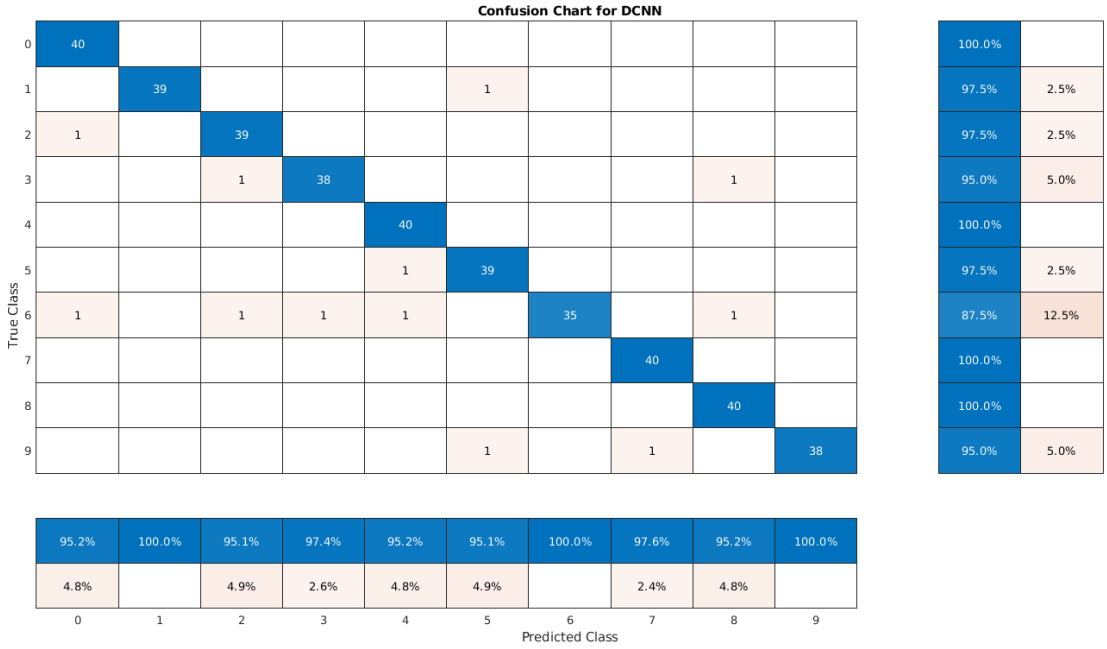
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest);
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 97
```

Summarize the performance of the trained network on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 1.5 1.5]);
ccDCNN = confusionchart(YTest,Ypredicted);
ccDCNN.Title = 'Confusion Chart for DCNN';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```



Appendix: Helper Functions

```
function Labels = helperGenerateLabels(ads)
% This function is only for use in this example. It may be changed or
% removed in a future release.
files = ads.Files;
tmp = cell(numel(files),1);
expression = "[0-9]+_";
parfor nf = 1:numel(ads.Files)
    idx = regexp(files{nf},expression);
    tmp{nf} = files{nf}(idx);
end
Labels = categorical(tmp);
end

%-----------------------------------------------------------------
function X = getValidationSpeechSpectrograms(ads,afe,params)
% This function is only for use in this example. It may changed or be
% removed in a future release.
%
% getValidationSpeechSpectrograms(ads,afe) computes speech spectrograms for
% the files in the datastore ads using the audioFeatureExtractor afe.

numFiles = length(ads.Files);
X = zeros([params.numBands,params.numHops,1,numFiles],'single');

for i = 1:numFiles
    x = read(ads);
    spec = getSpeechSpectrogram(x,afe,params);
    X(:,:,1,i) = spec;

end
```

```
end

%--------------------------------------------------------------------------
function X = getSpeechSpectrogram(x,afe,params)
% This function is only for use in this example. It may changed or be
% removed in a future release.
%
% getSpeechSpectrogram(x,afe) computes a speech spectrogram for the signal
% x using the audioFeatureExtractor afe.

X = zeros([params.numBands,params.numHops],'single');

x = normalizeAndResize(single(x),params);

spec = extract(afe,x).';

% If the spectrogram is less wide than numHops, then put spectrogram in
% the middle of X.
w = size(spec,2);
left = floor((params.numHops-w)/2)+1;
ind = left:left+w-1;
X(:,ind) = log10(spec + 1e-6);

end
%--------------------------------------------------------------------------
function x = normalizeAndResize(x,params)
% This function is only for use in this example. It may change or be
% removed in a future release.

L = params.segmentLength;
N = numel(x);
if N > L
    x = x(1:L);
elseif N < L
    pad = L-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));
end
%--------------------------------------------------------------------------
function myCustomWriter(spec,writeInfo,~)
% This function is only for use in this example. It may change or be
% removed in a future release.
% Define custom writing function to write auditory spectrogram/label pair
% to MAT files.
filename = strrep(writeInfo.SuggestedOutputName, '.wav','.mat');
label = writeInfo.ReadInfo.Label;
save(filename,'label','spec');
end
%--------------------------------------------------------------------------
function [data,info] = myCustomRead(filename)
% This function is only for use in this example. It may change or be
% removed in a future release.
load(filename,'spec','label');
data = {spec,label};
```

```
info.SampleRate = 8000;
end
```

Keyword Spotting in Noise Code Generation with Intel MKL-DNN

This example demonstrates code generation for keyword spotting using a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficient (MFCC) feature extraction. MATLAB® Coder™ with Deep Learning Support enables the generation of a standalone executable (`.exe`) file. Communication between the MATLAB® (.mlx) file and the generated executable file occurs over asynchronous User Datagram Protocol (UDP). The incoming speech signal is displayed using a `timescope`. A mask is shown as a blue rectangle surrounding spotted instances of the keyword, YES. For more details on MFCC feature extraction and deep learning network training, visit "Keyword Spotting in Noise Using MFCC and LSTM Networks" (Audio Toolbox).

Example Requirements

- MATLAB® Coder Interface for Deep Learning Support Package
- Intel® Xeon® processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2)
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment variables for Intel MKL-DNN

For supported versions of libraries and for information about setting up environment variables, see "Prerequisites for Deep Learning with MATLAB Coder" (MATLAB Coder).

Pretrained Network Keyword Spotting Using MATLAB and Streaming Audio from Microphone

The sample rate of the pretrained network is 16 kHz. Set the window length to 512 samples, with an overlap length of 384 samples, and a hop length defined as the difference between the window and overlap lengths. Define the rate at which the mask is estimated. A mask is generated once for every `numHopsPerUpdate` audio frames.

```
fs = 16e3;
windowLength = 512;
overlapLength = 384;
hopLength = windowLength - overlapLength;
numHopsPerUpdate = 16;
maskLength = hopLength*numHopsPerUpdate;
```

Create an `audioFeatureExtractor` (Audio Toolbox) object to perform MFCC feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
                            'Window',hann(windowLength,'periodic'), ...
                            'OverlapLength',overlapLength, ...
                            'mfcc',true, ...
                            'mfccDelta',true, ...
                            'mfccDeltaDelta',true);
```

Download and load the pretrained network, as well as the mean (`M`) and the standard deviation (`S`) vectors used for Feature Standardization.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/KeywordSpotting.zip';
downloadNetFolder = './';
netFolder = fullfile(downloadNetFolder,'KeywordSpotting');
if ~exist(netFolder,'dir')
    disp('Downloading pretrained network and audio files (4 files - 7 MB) ...')
```

```
    unzip(url,downloadNetFolder)
end
load(fullfile(netFolder,'KWSNet.mat'),"KWSNet","M","S");
```

Call `generateMATLABFunction` (Audio Toolbox) on the `audioFeatureExtractor` (Audio Toolbox) object to create the feature extraction function. You will use this function in the processing loop.

```
generateMATLABFunction(afe,'generateKeywordFeatures','IsStreaming',true);
```

Define an Audio Device Reader (Audio Toolbox) that can read audio from your microphone. Set the frame length equal to the hop length. This enables you to compute a new set of features for every new audio frame from the microphone.

```
frameLength = hopLength;
adr = audioDeviceReader('SampleRate',fs, ...
                        'SamplesPerFrame',frameLength);
```

Create a Time Scope (DSP System Toolbox) to visualize the speech signals and estimated mask.

```
scope = timescope('SampleRate',fs, ...
                  'TimeSpanSource','property', ...
                  'TimeSpan',5, ...
                  'TimeSpanOverrunAction','Scroll', ...
                  'BufferLength',fs*5*2, ...
                  'ShowLegend',true, ...
                  'ChannelNames',{'Speech','Keyword Mask'}, ...
                  'YLimits',[-1.2 1.2], ...
                  'Title','Keyword Spotting');
```

Initialize a buffer for the audio data, a buffer for the computed features, and a buffer to plot the input audio and the output speech mask.

```
dataBuff = dsp.AsyncBuffer(windowLength);
featureBuff = dsp.AsyncBuffer(numHopsPerUpdate);
plotBuff = dsp.AsyncBuffer(numHopsPerUpdate*windowLength);
```

Perform keyword spotting on speech received from your microphone. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the simulation, close the `scope`.

```
timeLimit = 20;
show(scope);
tic
while toc < timeLimit && isVisible(scope)

    data = adr();
    write(dataBuff,data);
    write(plotBuff,data);

    frame = read(dataBuff,windowLength,overlapLength);
    features = generateKeywordFeatures(frame,fs);
    write(featureBuff,features.');

    if featureBuff.NumUnreadSamples == numHopsPerUpdate

        featureMatrix = read(featureBuff);
        featureMatrix(~isfinite(featureMatrix)) = 0;
        featureMatrix = (featureMatrix - M)./S;
```

```
        [keywordNet, v] = classifyAndUpdateState(KWSNet,featureMatrix.');

        v = double(v) - 1;
        v = repmat(v,hopLength,1);
        v = v(:);
        v = mode(v);
        predictedMask = repmat(v,numHopsPerUpdate*hopLength,1);

        data = read(plotBuff);
        scope([data,predictedMask]);

        drawnow limitrate;
    end
end

release(adr)
hide(scope)
```

The `helperKeywordSpotting` supporting function encapsulates capturing the audio, feature extraction and network prediction process demonstrated previously. To make feature extraction compatible with code generation, feature extraction is handled by the generated `generateKeywordFeatures` function. To make the network compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) (MATLAB Coder) function to load the network.

The supporting function uses a `dsp.UDPSender` (DSP System Toolbox) System object to send the input data along with the output mask predicted by the network to MATLAB. The MATLAB script uses the `dsp.UDPReceiver` (DSP System Toolbox) System object to receive the input data along with the output mask predicted by the network running in the supporting function.

Generate Executable on Desktop

Create a code generation configuration object to generate an executable. Specify the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the MKL-DNN library. Attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('mkldnn');
cfg.DeepLearningConfig = dlcfg;
```

Generate the C++ main file required to produce the standalone executable.

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Generate `helperKeywordSpotting`, a supporting function that encapsulates the audio capture, feature extraction, and network prediction processes. You get a warning in the code generation logs that you can disregard because `helperKeywordSpotting` has an infinite loop that continously looks for an audio frame from MATLAB.

```
codegen helperKeywordSpotting -config cfg -report
```

```
Warning: Function 'helperKeywordSpotting' does not terminate due to an infinite loop.
```

```
Warning in ==> helperKeywordSpotting Line: 73 Column: 1
Code generation successful (with warnings): View report
```

Prepare Dependencies and Run the Generated Executable

In this section, you generate all the required dependency files and put them into a single folder. During the build process, MATLAB Coder generates `buildInfo.mat`, a file that contains the compilation and run-time dependency information for the standalone executable.

Set the project name to `helperKeywordSpotting`.

```
projName = 'helperKeywordSpotting';
packageName = [projName,'Package'];
if ispc
    exeName = [projName,'.exe'];
else
    exeName = projName;
end
```

Load `buildinfo.mat` and use `packNGo` (MATLAB Coder) to produce a `.zip` package.

```
load(['codegen',filesep,'exe',filesep,projName,filesep,'buildInfo.mat']);
packNGo(buildInfo,'fileName',[packageName,'.zip'],'minimalHeaders',false);
```

Unzip the package and place the executable file in the unzipped directory.

```
unzip([packageName,'.zip'],packageName);
copyfile(exeName, packageName,'f');
```

To invoke a standalone executable that depends on the MKL-DNN Dynamic Link Library, append the path to the MKL-DNN library location to the environment variable `PATH`.

```
setenv('PATH',[getenv('INTEL_MKLDNN'),filesep,'lib',pathsep,getenv('PATH')]);
```

Run the generated executable.

```
if ispc
    system(['start cmd /k "title ',packageName,' && cd ',packageName,' && ',exeName]);
else
    cd(packageName);
    system(['./',exeName,' &']);
    cd ..;
end
```

Perform Keyword Spotting Using Deployed Code

Create a `dsp.UDPReceiver` (DSP System Toolbox) System object to receive speech data and the predicted speech mask from the standalone executable. Each UDP packet received from the executable consists of `maskLength` mask samples and speech samples. The maximum message length for the `dsp.UDPReceiver` (DSP System Toolbox) object is `65507` bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeOfFloatInBytes = 4;
speechDataLength = maskLength;
numElementsPerUDPPacket = maskLength + speechDataLength;

maxUDPMessageLength = floor(65507/sizeOfFloatInBytes);
samplesPerPacket = 1 + numElementsPerUDPPacket;
```

```
numPackets = floor(maxUDPMessageLength/samplesPerPacket);
bufferSize = numPackets*samplesPerPacket*sizeOfFloatInBytes;

UDPReceive = dsp.UDPReceiver('LocalIPPort',20000, ...
    'MessageDataType','single', ...
    'MaximumMessageLength',samplesPerPacket, ...
    'ReceiveBufferSize',bufferSize);
```

To run the keyword spotting indefinitely, set `timelimit` to `Inf`. To stop the simulation, close the `scope`.

```
tic;
timelimit = 20;
show(scope);

while toc < timelimit && isVisible(scope)
    data = UDPReceive();
    if ~isempty(data)
        plotMask = data(1:maskLength);
        plotAudio = data(maskLength+1 : maskLength+speechDataLength);
        scope([plotAudio,plotMask]);
    end
    drawnow limitrate;
end

hide(scope);
```

Release the system objects and terminate the standalone executable.

```
release(UDPReceive);
release(scope);
if ispc
    system(['taskkill /F /FI "WindowTitle eq ',projName,'* " /T']);
else
    system(['killall ',exeName]);
end
```

```
SUCCESS: The process with PID 4644 (child process of PID 21188) has been terminated.
SUCCESS: The process with PID 20052 (child process of PID 21188) has been terminated.
SUCCESS: The process with PID 21188 (child process of PID 22940) has been terminated.
```

Evaluate Execution Time Using Alternative MEX Function Workflow

A similar workflow involves using a MEX file instead of the standalone executable. Perform MEX profiling to measure the computation time for the workflow.

Create a code generation configuration object to generate the MEX function. Specify the target language as C++.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the MKL-DNN library. Attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('mkldnn');
cfg.DeepLearningConfig = dlcfg;
```

Call codegen to generate the MEX function for `profileKeywordSpotting`.

```
inputAudioFrame = ones(hopLength,1,'single');
codegen profileKeywordSpotting -config cfg -args {inputAudioFrame} -report
```

```
Code generation successful: View report
```

Measure the execution time of the MATLAB code.

```
x = pinknoise(hopLength,1,'single');
numPredictCalls = 100;
totalNumCalls = numPredictCalls*numHopsPerUpdate;
exeTimeStart = tic;
for call = 1:totalNumCalls
    [outputMask,inputData,plotFlag] = profileKeywordSpotting(x);
end
exeTime = toc(exeTimeStart);
fprintf('MATLAB execution time per %d ms of audio = %0.4f ms\n',int32(1000*numHopsPerUpdate*hopLe
```

```
MATLAB execution time per 128 ms of audio = 24.9238 ms
```

Measure the execution time of the MEX function.

```
exeTimeMexStart = tic;
for call = 1:totalNumCalls
    [outputMask,inputData,plotFlag] = profileKeywordSpotting_mex(x);
end
exeTimeMex = toc(exeTimeMexStart);
fprintf('MEX execution time per %d ms of audio = %0.4f ms\n',int32(1000*numHopsPerUpdate*hopLengt
```

```
MEX execution time per 128 ms of audio = 5.2710 ms
```

Compare total execution time of the standalone executable approach with the MEX function approach. This performance test is done on a machine using an NVIDIA Quadro® P620 (Version 26) GPU and an Intel Xeon W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = exeTime/exeTimeMex
```

```
PerformanceGain = 4.7285
```

Keyword Spotting in Noise Code Generation on Raspberry Pi

This example demonstrates code generation for keyword spotting using a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficient (MFCC) feature extraction on Raspberry Pi™. MATLAB® Coder™ with Deep Learning Support enables the generation of a standalone executable (.elf) file on Raspberry Pi. Communication between MATLAB® (.mlx) file and the generated executable file occurs over asynchronous User Datagram Protocol (UDP). The incoming speech signal is displayed using a `timescope`. A mask is shown as a blue rectangle surrounding spotted instances of the keyword, YES. For more details on MFCC feature extraction and deep learning network training, visit “Keyword Spotting in Noise Using MFCC and LSTM Networks” (Audio Toolbox).

Example Requirements

- MATLAB® Coder Interface for Deep Learning Support Package
- ARM processor that supports the NEON extension
- ARM Compute Library version 20.02.1 (on the target ARM hardware)
- Environment variables for the compilers and libraries

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

Pretrained Network Keyword Spotting Using MATLAB® and Streaming Audio from Microphone

The sample rate of the pretrained network is 16 kHz. Set the window length to 512 samples, with an overlap length of 384 samples, and a hop length defined as the difference between the window and overlap lengths. Define the rate at which the mask is estimated. A mask is generated once for every `numHopsPerUpdate` audio frames.

```
fs = 16e3;
windowLength = 512;
overlapLength = 384;
hopLength = windowLength - overlapLength;

numHopsPerUpdate = 16;
maskLength = hopLength * numHopsPerUpdate;
```

Create an `audioFeatureExtractor` (Audio Toolbox) object to perform MFCC feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
                            'Window',hann(windowLength,'periodic'), ...
                            'OverlapLength',overlapLength, ...
                            'mfcc',true, ...
                            'mfccDelta',true, ...
                            'mfccDeltaDelta',true);
```

Download and load the pretrained network, as well as the mean (`M`) and the standard deviation (`S`) vectors used for feature standardization.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/KeywordSpotting.zip';
downloadNetFolder = './';
netFolder = fullfile(downloadNetFolder,'KeywordSpotting');
if ~exist(netFolder,'dir')
    disp('Downloading pretrained network and audio files (4 files - 7 MB) ...')
```

```
    unzip(url,downloadNetFolder)
end
load(fullfile(netFolder,'KWSNet.mat'),"KWSNet","M","S");
```

Call `generateMATLABFunction` (Audio Toolbox) on the `audioFeatureExtractor` (Audio Toolbox) object to create the feature extraction function.

```
generateMATLABFunction(afe,'generateKeywordFeatures','IsStreaming',true);
```

Define an Audio Device Reader (Audio Toolbox) System object™ to read audio from your microphone. Set the frame length equal to the hop length. This enables the computation of a new set of features for every new audio frame received from the microphone.

```
frameLength = hopLength;
adr = audioDeviceReader('SampleRate',fs, ...
                        'SamplesPerFrame',frameLength,'OutputDataType','single');
```

Create a Time Scope (DSP System Toolbox) to visualize the speech signals and estimated mask.

```
scope = timescope('SampleRate',fs, ...
                  'TimeSpanSource','property', ...
                  'TimeSpan',5, ...
                  'TimeSpanOverrunAction','Scroll', ...
                  'BufferLength',fs*5*2, ...
                  'ShowLegend',true, ...
                  'ChannelNames',{'Speech','Keyword Mask'}, ...
                  'YLimits',[-1.2 1.2], ...
                  'Title','Keyword Spotting');
```

Initialize a buffer for the audio data, a buffer for the computed features, and a buffer to plot the input audio and the output speech mask.

```
dataBuff = dsp.AsyncBuffer(windowLength);
featureBuff = dsp.AsyncBuffer(numHopsPerUpdate);
plotBuff = dsp.AsyncBuffer(numHopsPerUpdate*windowLength);
```

Perform keyword spotting on speech received from your microphone. To run the loop indefinitely, set `timeLimit` to `Inf`. To stop the simulation, close the `scope`.

```
show(scope);
timeLimit = 20;
tic
while toc < timeLimit && isVisible(scope)

    data = adr();
    write(dataBuff,data);
    write(plotBuff,data);

    frame = read(dataBuff,windowLength,overlapLength);
    features = generateKeywordFeatures(frame,fs);
    write(featureBuff,features.');

    if featureBuff.NumUnreadSamples == numHopsPerUpdate

        featureMatrix = read(featureBuff);
        featureMatrix(~isfinite(featureMatrix)) = 0;
        featureMatrix = (featureMatrix - M)./S;
```

```
        [keywordNet,v] = classifyAndUpdateState(KWSNet,featureMatrix.');

        v = double(v) - 1;
        v = repmat(v,hopLength,1);
        v = v(:);
        v = mode(v);
        v = repmat(v,numHopsPerUpdate * hopLength,1);

        data = read(plotBuff);
        scope([data,v]);

        drawnow limitrate;
    end
end
hide(scope)
```

The `helperKeywordSpottingRaspi` supporting function encapsulates the feature extraction and network prediction process demonstrated previously. To make feature extraction compatible with code generation, feature extraction is handled by the generated `generateKeywordFeatures` function. To make the network compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) function to load the network.

The supporting function uses a `dsp.UDPReceiver` (DSP System Toolbox) System object to receive the captured audio from MATLAB® and uses a `dsp.UDPSender` (DSP System Toolbox) System object to send the input speech signal along with the estimated mask predicted by the network to MATLAB®. Similarly, the MATLAB® live script uses the `dsp.UDPSender` (DSP System Toolbox) System object to send the captured speech signal to the executable running on Raspberry Pi and the `dsp.UDPReceiver` (DSP System Toolbox) System object to receive the speech signal and estimated mask from Raspberry Pi.

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends the input speech signal and estimated mask to the specified IP address.

```
hostIPAddress = coder.Constant('172.18.230.30');
```

Create a code generation configuration object to generate an executable program. Specify the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the ARM compute library that is on your Raspberry Pi. Specify the architecture of the Raspberry Pi and attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '20.02.1';
cfg.DeepLearningConfig = dlcfg;
```

Use the Raspberry Pi Support Package function, `raspi`, to create a connection to your Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi

- `pi` with your user name
- `password` with your password

```
r = raspi('raspiname','pi','password');
```

Create a `coder.hardware` (MATLAB Coder) object for Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Generate the C++ main file required to produce the standalone executable.

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Generate C++ code for `helperKeywordSpottingRaspi` on your Raspberry Pi.

```
codegen -config cfg helperKeywordSpottingRaspi -args {hostIPAddress} -report
```

```
 Deploying code. This may take a few minutes.
Warning: Function 'helperKeywordSpottingRaspi' does not terminate due to an
infinite loop.

Warning in ==> helperKeywordSpottingRaspi Line: 78 Column: 1
Code generation successful (with warnings): View report
```

Perform Keyword Spotting Using Deployed Code

Create a command to open the `helperKeywordSpottingRaspi` application on Raspberry Pi. Use `system` to send the command to your Raspberry Pi.

```
applicationName = 'helperKeywordSpottingRaspi';

applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName',applicationName);
targetDirPath = applicationDirPaths{1}.directory;

exeName = strcat(applicationName,'.elf');
command = ['cd ',targetDirPath,'; ./',exeName,' &> 1 &'];

system(r,command);
```

Create a `dsp.UDPSender` (DSP System Toolbox) System object to send audio captured in MATLAB® to your Raspberry Pi. Update the `targetIPAddress` for your Raspberry Pi. Raspberry Pi receives the captured audio from the same port using the `dsp.UDPReceiver` (DSP System Toolbox) System object.

```
targetIPAddress = '172.18.240.234';
UDPSend = dsp.UDPSender('RemoteIPPort',26000,'RemoteIPAddress',targetIPAddress);
```

Create a `dsp.UDPReceiver` (DSP System Toolbox) System object to receive speech data and the predicted speech mask from your Raspberry Pi. Each UDP packet received from the Raspberry Pi consists of `maskLength` mask and speech samples. The maximum message length for the

`dsp.UDPReceiver` (DSP System Toolbox) object is `65507` bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeOfFloatInBytes = 4;
speechDataLength = maskLength;
numElementsPerUDPPacket = maskLength + speechDataLength;
maxUDPMessageLength = floor(65507/sizeOfFloatInBytes);
numPackets = floor(maxUDPMessageLength/numElementsPerUDPPacket);
bufferSize = numPackets*numElementsPerUDPPacket*sizeOfFloatInBytes;

UDPReceive = dsp.UDPReceiver("LocalIPPort",21000, ...
    "MessageDataType","single", ...
    "MaximumMessageLength",1+numElementsPerUDPPacket, ...
    "ReceiveBufferSize",bufferSize);
```

Spot the keyword as long as time scope is open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope.

```
tic;
show(scope);
timelimit = 20;
while toc < timelimit && isVisible(scope)
    x = adr();
    UDPSend(x);
    data = UDPReceive();
    if ~isempty(data)
        mask = data(1:maskLength);
        dataForPlot = data(maskLength + 1 : numElementsPerUDPPacket);
        scope([dataForPlot,mask]);
    end
    drawnow limitrate;
end
```

Release the system objects and terminate the standalone executable.

```
hide(scope)
release(UDPSend)
release(UDPReceive)
release(scope)
release(adr)
stopExecutable(codertarget.raspi.raspberrypi,exeName)
```

Evaluate Execution Time Using Alternative PIL Function Workflow

To evaluate execution time taken by standalone executable on Raspberry Pi, use a PIL (processor-in-loop) workflow. To perform PIL profiling, generate a PIL function for the supporting function `profileKeywordSpotting`. The `profileKeywordSpotting` is equivalent to `helperKeywordSpottingRaspi`, except that the former returns the speech and predicted speech mask while the latter sends the same parameters using UDP. The time taken by the UDP calls is less than 1 ms, which is relatively small compared to the overall execution time.

Create a code generation configuration object to generate the PIL function.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
```

Set the ARM compute library and architecture.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
cfg.DeepLearningConfig = dlcfg ;
cfg.DeepLearningConfig.ArmArchitecture = 'armv7';
cfg.DeepLearningConfig.ArmComputeVersion = '20.02.1';
```

Set up the connection with your target hardware.

```
if (~exist('r','var'))
  r = raspi('raspiname','pi','password');
end
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Set the build directory and target language.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
cfg.TargetLang = 'C++';
```

Enable profiling and generate the PIL code. A MEX file named `profileKeywordSpotting_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
codegen -config cfg profileKeywordSpotting -args {pinknoise(hopLength,1,'single')} -report
```

```
 Deploying code. This may take a few minutes.
### Connectivity configuration for function 'profileKeywordSpotting': 'Raspberry Pi'
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2021a/E/sandbox/sporwal/Examp
Code generation successful: View report
```

Evaluate Raspberry Pi Execution Time

Call the generated PIL function multiple times to get the average execution time.

```
numPredictCalls = 10;
totalCalls = numHopsPerUpdate * numPredictCalls;

x = pinknoise(hopLength,1,'single');
for k = 1:totalCalls
    [maskReceived,inputSignal,plotFlag] = profileKeywordSpotting_pil(x);
end
```

```
### Starting application: 'codegen\lib\profileKeywordSpotting\pil\profileKeywordSpotting.elf'
    To terminate execution: clear profileKeywordSpotting_pil
### Launching application profileKeywordSpotting.elf...
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.
```

Terminate the PIL execution.

```
clear profileKeywordSpotting_pil
```

```
### Host application produced the following standard output (stdout) and standard error (stderr)

    Execution profiling report: report(getCoderExecutionProfile('profileKeywordSpotting'))
```

Generate an execution profile report to evaluate execution time.

```
executionProfile = getCoderExecutionProfile('profileKeywordSpotting');
report(executionProfile, ...
```

```
        'Units','Seconds', ...
        'ScaleFactor','1e-03', ...
        'NumericFormat','%0.4f')

ans =
'E:\sandbox\sporwal\Examples\ExampleManager\sporwal.Bdoc21a.j1572571\deeplearning_shared-ex18742
```

Code Execution Profiling Report

Code Execution Profiling Report for profileKeywordSpotting

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time	278.7215
Unit of time	ms
Command	report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second)	1e+09
Profiling data created	28-Oct-2020 18:52:30

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
profileKeywordSpotting_initialize	0.0868	0.0868	0.0868	0.0868	1	
profileKeywordSpotting	26.9524	1.7414	26.9524	1.7414	160	
profileKeywordSpotting_terminate	0.0029	0.0029	0.0029	0.0029	1	

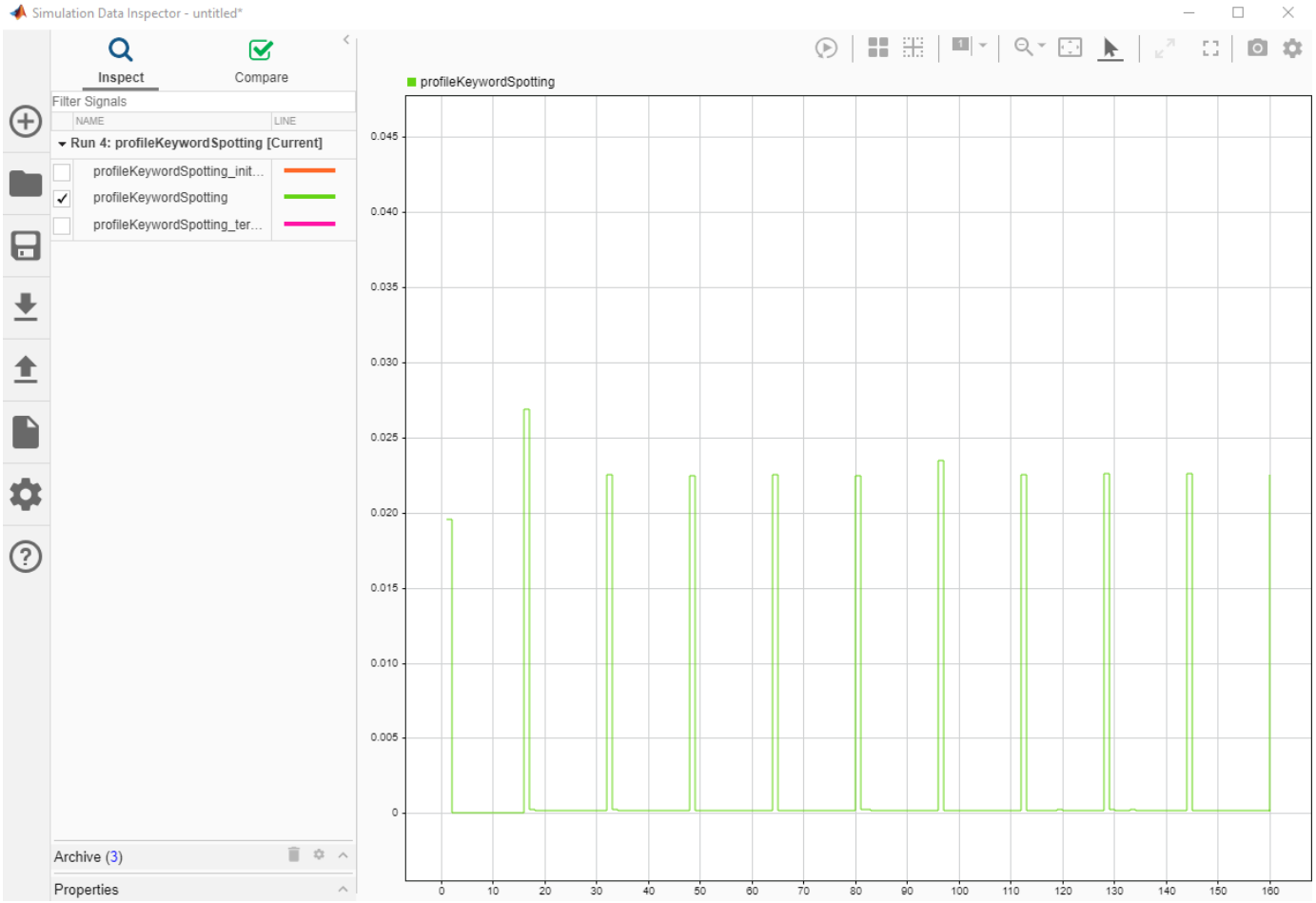
3. Definitions

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

OK Help

Plot the Execution Time of each frame from the generated report.



Processing of the first frame took ~20 ms due to initialization overhead costs. The spikes in the time graph at every 16th frame (`numHopsPerUpdate`) correspond to the computationally intensive predict function called every 16th frame. The maximum execution time is ~30 ms, which is below the 128 ms budget for real-time streaming. The performance is measuerd on Raspberry Pi 4 Model B Rev 1.1.

Speech Command Recognition Code Generation on Raspberry Pi

This example shows how to deploy feature extraction and a convolutional neural network (CNN) for speech command recognition to Raspberry Pi™. To generate the feature extraction and network code, you use MATLAB Coder, MATLAB Support Package for Raspberry Pi Hardware, and the ARM® Compute Library. In this example, the generated code is an executable on your Raspberry Pi, which is called by a MATLAB script that displays the predicted speech command along with the signal and auditory spectrogram. Interaction between the MATLAB script and the executable on your Raspberry Pi is handled using the user datagram protocol (UDP). For details about audio preprocessing and network training, see “Speech Command Recognition Using Deep Learning” (Audio Toolbox).

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library version 19.05 (on the target ARM hardware)
- Environment variables for the compilers and libraries

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

Streaming Demonstration in MATLAB

Use the same parameters for the feature extraction pipeline and classification as developed in “Speech Command Recognition Using Deep Learning” (Audio Toolbox).

Define the same sample rate the network was trained on (16 kHz). Define the classification rate and the number of audio samples input per frame. The feature input to the network is a Bark spectrogram that corresponds to 1 second of audio data. The Bark spectrogram is calculated for 25 ms windows with 10 ms hops. Calculate the number of individual spectrums in each spectrogram.

```
fs = 16000;
classificationRate = 20;
samplesPerCapture = fs/classificationRate;

segmentDuration = 1;
segmentSamples = round(segmentDuration*fs);

frameDuration = 0.025;
frameSamples = round(frameDuration*fs);

hopDuration = 0.010;
hopSamples = round(hopDuration*fs);

numSpectrumPerSpectrogram = floor((segmentSamples-frameSamples)/hopSamples) + 1;
```

Create an `audioFeatureExtractor` (Audio Toolbox) object to extract 50-band Bark spectrograms without window normalization. Calculate the number of elements in each spectrogram.

```
afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',512, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',frameSamples - hopSamples, ...
```

```
    'barkSpectrum',true);

numBands = 50;
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);

numElementsPerSpectrogram = numSpectrumPerSpectrogram*numBands;
```

Load the pretrained CNN and labels.

```
load('commandNet.mat')
labels = trainedNet.Layers(end).Classes;
NumLabels = numel(labels);
BackGroundIdx = find(labels == 'background');
```

Define buffers and decision thresholds to post process network predictions.

```
probBuffer = single(zeros([NumLabels,classificationRate/2]));
YBuffer = single(NumLabels * ones(1, classificationRate/2));

countThreshold = ceil(classificationRate*0.2);
probThreshold = single(0.7);
```

Create an `audioDeviceReader` (Audio Toolbox) object to read audio from your device. Create a `dsp.AsyncBuffer` (DSP System Toolbox) object to buffer the audio into chunks.

```
adr = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',samplesPerCapture,'OutputDataType','si
audioBuffer = dsp.AsyncBuffer(fs);
```

Create a `dsp.MatrixViewer` (DSP System Toolbox) object and a `timescope` (DSP System Toolbox) object to display the results.

```
matrixViewer = dsp.MatrixViewer("ColorBarLabel","Power per band (dB/Band)",...
    "XLabel","Frames",...
    "YLabel","Bark Bands", ...
    "Position",[400 100 600 250], ...
    "ColorLimits",[-4 2.6445], ...
    "AxisOrigin","Lower left corner", ...
    "Name","Speech Command Recognition using Deep Learning");

timeScope = timescope("SampleRate",fs, ...
    "YLimits",[-1 1], ...
    "Position",[400 380 600 250], ...
    "Name","Speech Command Recognition Using Deep Learning", ...
    "TimeSpanSource","Property", ...
    "TimeSpan",1, ...
    "BufferLength",fs, ...
    "YLabel","Amplitude", ...
    "ShowGrid",true);
```

Show the time scope and matrix viewer. Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```
show(timeScope)
show(matrixViewer)

timeLimit = 10;
```

```
tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    % Capture audio
    x = adr();
    write(audioBuffer,x);
    y = read(audioBuffer,fs,fs-samplesPerCapture);

    % Compute auditory features
    features = extract(afe,y);
    auditoryFeatures = log10(features + 1e-6);

    % Perform prediction
    probs = predict(trainedNet, auditoryFeatures);
    [~, YPredicted] = max(probs);

    % Perform statistical post processing
    YBuffer = [YBuffer(2:end),YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

    [YModeIdx, count] = mode(YBuffer);
    maxProb = max(probBuffer(YModeIdx,:));

    if YModeIdx == single(BackGroundIdx) || single(count) < countThreshold || maxProb < probThres
        speechCommandIdx = BackGroundIdx;
    else
        speechCommandIdx = YModeIdx;
    end

    % Update plots
    matrixViewer(auditoryFeatures');
    timeScope(x);

    if (speechCommandIdx == BackGroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow limitrate
end
```

Hide the scopes.

```
hide(matrixViewer)
hide(timeScope)
```

Prepare MATLAB Code for Deployment

To create a function to perform feature extraction compatible with code generation, call `generateMATLABFunction` (Audio Toolbox) on the `audioFeatureExtractor` object. The `generateMATLABFunction` object function creates a standalone function that performs equivalent feature extraction and is compatible with code generation.

```
generateMATLABFunction(afe,'extractSpeechFeatures')
```

The `HelperSpeechCommandRecognitionRasPi` supporting function encapsulates the feature extraction and network prediction process demonstrated previously. So that the feature extraction is compatible with code generation, feature extraction is handled by the generated `extractSpeechFeatures` function. So that the network is compatible with code generation, the

supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) function to load the network. The supporting function uses a `dsp.UDPReceiver` (DSP System Toolbox) system object to send the auditory spectrogram and the index corresponding to the predicted speech command from Raspberry Pi to MATLAB. The supporting function uses the `dsp.UDPReceiver` (DSP System Toolbox) system object to receive the audio captured by your microphone in MATLAB.

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends auditory spectrograms and the predicted speech command to this IP address.

```
hostIPAddress = coder.Constant('172.18.230.30');
```

Create a code generation configuration object to generate an executable program. Specify the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the ARM compute library that is on your Raspberry Pi. Specify the architecture of the Raspberry Pi and attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

Use the Raspberry Pi Support Package function, `raspi`, to create a connection to your Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi
- pi with your user name
- `password` with your password

```
r = raspi('raspiname','pi','password');
```

Create a `coder.hardware` (MATLAB Coder) object for Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Use an auto generated C++ main file for the generation of a standalone executable.

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Call `codegen` (MATLAB Coder) to generate C++ code and the executable on your Raspberry Pi. By default, the Raspberry Pi application name is the same as the MATLAB function.

```
codegen -config cfg HelperSpeechCommandRecognitionRasPi -args {hostIPAddress} -report -v
```

```
 Deploying code. This may take a few minutes.
### Compiling function(s) HelperSpeechCommandRecognitionRasPi ...
-------------------------------------------------------------------------
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2021b/C/ExampleMatlab/ExampleM
### Using toolchain: GNU GCC Embedded Linux
### 'C:\ExampleMatlab\ExampleManager\sporwal.Bdoc21b.j1648568\deeplearning_shared-ex00376115\code
### Building 'HelperSpeechCommandRecognitionRasPi': make  -f HelperSpeechCommandRecognitionRasPi_

-------------------------------------------------------------------------
### Generating compilation report ...
Warning: Function 'HelperSpeechCommandRecognitionRasPi' does not terminate due to an infinite
loop.

Warning in ==> HelperSpeechCommandRecognitionRasPi Line: 86 Column: 1
Code generation successful (with warnings): View report
```

Initialize Application on Raspberry Pi

Create a command to open the `HelperSpeechCommandRasPi application on Raspberry Pi`. Use `system` to send the command to your Raspberry Pi.

```
applicationName = 'HelperSpeechCommandRecognitionRasPi';

applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName',applicationName);
targetDirPath = applicationDirPaths{1}.directory;

exeName = strcat(applicationName,'.elf');
command = ['cd ' targetDirPath '; ./' exeName ' &> 1 &'];

system(r,command);
```

Create a `dsp.UDPReceiver` (DSP System Toolbox) system object to send audio captured in MATLAB to your Raspberry Pi. Update the `targetIPAddress` for your Raspberry Pi. Raspberry Pi receives the captured audio from the same port using the `dsp.UDPReceiver` (DSP System Toolbox) system object.

```
targetIPAddress = '172.18.228.24';
UDPSend = dsp.UDPSender('RemoteIPPort',26000,'RemoteIPAddress',targetIPAddress);
```

Create a `dsp.UDPReceiver` (DSP System Toolbox) system object to receive auditory features and the predicted speech command index from your Raspberry Pi. Each UDP packet received from the Raspberry Pi consists of auditory features in column-major order followed by the predicted speech command index. The maximum message length for the `dsp.UDPReceiver` object is 65507 bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeOfFloatInBytes = 4;
maxUDPMessageLength = floor(65507/sizeOfFloatInBytes);
samplesPerPacket = 1 + numElementsPerSpectrogram;
numPackets = floor(maxUDPMessageLength/samplesPerPacket);
bufferSize = numPackets*samplesPerPacket*sizeOfFloatInBytes;

UDPReceive = dsp.UDPReceiver("LocalIPPort",21000, ...
    "MessageDataType","single", ...
    "MaximumMessageLength",samplesPerPacket, ...
    "ReceiveBufferSize",bufferSize);
```

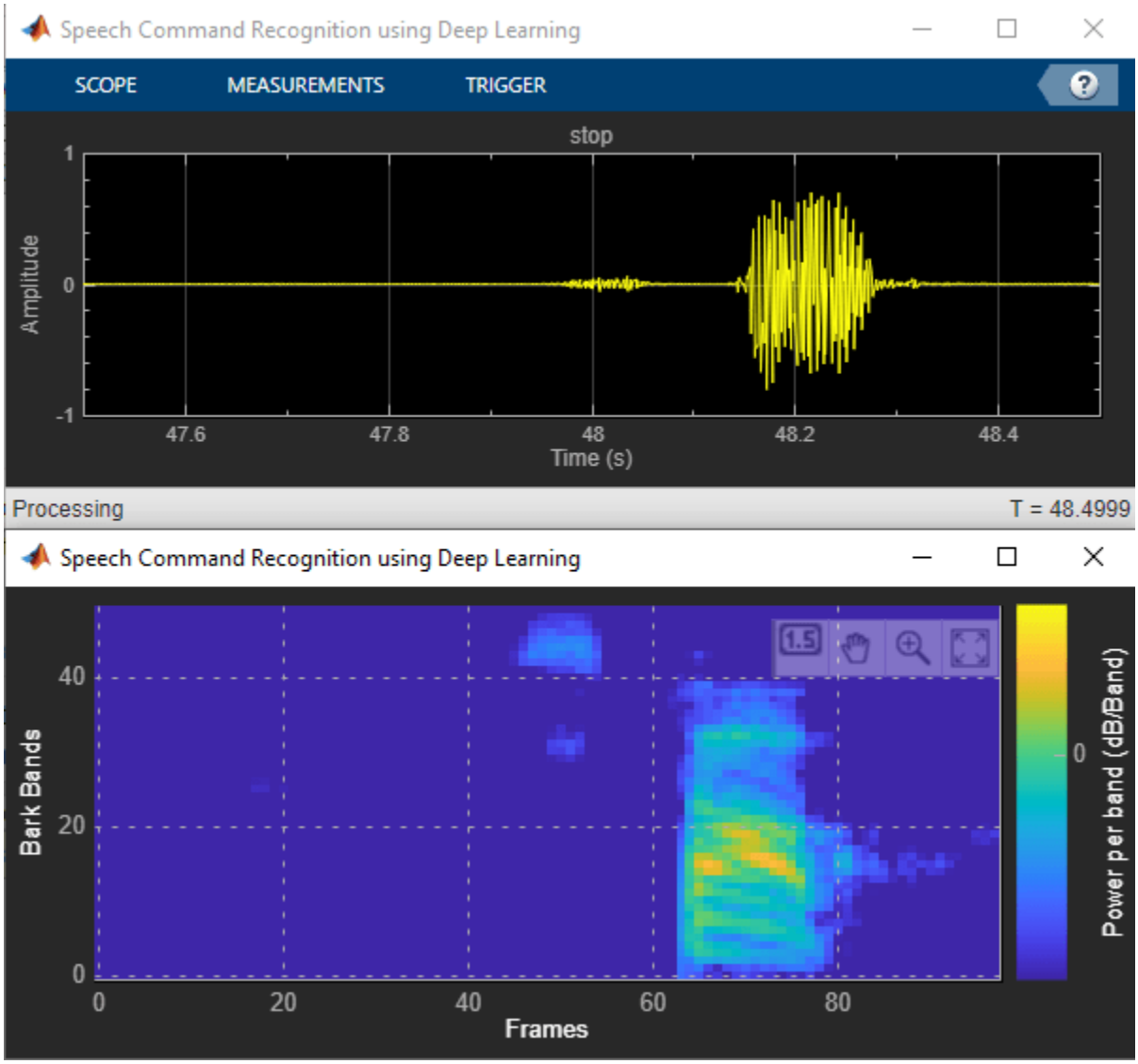
Reduce initialization overhead by sending a frame of zeros to the executable running on your Raspberry Pi.

```
UDPSend(zeros(samplesPerCapture,1,"single"));
```

Perform Speech Command Recognition Using Deployed Code

Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope or matrix viewer window.

```
show(timeScope)
show(matrixViewer)

timeLimit = 20;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    % Capture audio and send that to RasPi
    x = adr();
    UDPSend(x);

    % Receive data packet from RasPi
    udpRec = UDPReceive();

    if ~isempty(udpRec)
        % Extract predicted index, the last sample of received UDP packet
        speechCommandIdx = udpRec(end);

        % Extract auditory spectrogram
        spec = reshape(udpRec(1:numElementsPerSpectrogram), [numBands, numSpectrumPerSpectrogram]

        % Display time domain signal and auditory spectrogram
        timeScope(x)
        matrixViewer(spec)

        if speechCommandIdx == BackGroundIdx
            timeScope.Title = ' ';
        else
            timeScope.Title = char(labels(speechCommandIdx));
        end

        drawnow limitrate
    end
end

hide(matrixViewer)
hide(timeScope)
```



To stop the executable on your Raspberry Pi, use `stopExecutable`. Release the UDP objects.

```
stopExecutable(codertarget.raspi.raspberrypi,exeName)

release(UDPSend)
release(UDPReceive)
```

Profile Using PIL Workflow

You can measure the execution time taken on the Raspberry Pi using a PIL (processor-in-loop) workflow. The `ProfileSpeechCommandRecognitionRaspi` supporting function is the equivalent of the HelperSpeechCommandRecognitionRaspi function, except that the former returns the speech command index and auditory spectrogram while the latter sends the same parameters using UDP. The time taken by the UDP calls is less than 1 ms, which is relatively small compared to the overall execution time.

Create a PIL configuration object.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
```

Set the ARM compute library and architecture.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
cfg.DeepLearningConfig = dlcfg ;
cfg.DeepLearningConfig.ArmArchitecture = 'armv7';
cfg.DeepLearningConfig.ArmComputeVersion = '19.05';
```

Set up the connection with your target hardware.

```
if (~exist('r','var'))
  r = raspi('raspiname','pi','password');
end
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Set the build directory and target language.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
cfg.TargetLang = 'C++';
```

Enable profiling and then generate the PIL code. A MEX file named `ProfileSpeechCommandRecognition_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
codegen -config cfg ProfileSpeechCommandRecognitionRaspi -args {rand(samplesPerCapture, 1, 'sing
```

```
 Deploying code. This may take a few minutes.
### Compiling function(s) ProfileSpeechCommandRecognitionRaspi ...
### Connectivity configuration for function 'ProfileSpeechCommandRecognitionRaspi': 'Raspberry Pi
### Using toolchain: GNU GCC Embedded Linux
### Creating 'C:\ExampleMatlab\ExampleManager\sporwal.Bdoc21b.j1648568\deeplearning_shared-ex003
### Building 'ProfileSpeechCommandRecognitionRaspi_ca': make  -f ProfileSpeechCommandRecognitionR
### Using toolchain: GNU GCC Embedded Linux
### Creating 'C:\ExampleMatlab\ExampleManager\sporwal.Bdoc21b.j1648568\deeplearning_shared-ex003
### Building 'ProfileSpeechCommandRecognitionRaspi': make  -f ProfileSpeechCommandRecognitionRasp
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2021b/C/ExampleMatlab/ExampleM
--------------------------------------------------------------------------
### Using toolchain: GNU GCC Embedded Linux
### 'C:\ExampleMatlab\ExampleManager\sporwal.Bdoc21b.j1648568\deeplearning_shared-ex00376115\code
### Building 'ProfileSpeechCommandRecognitionRaspi': make  -f ProfileSpeechCommandRecognitionRasp

--------------------------------------------------------------------------
### Generating compilation report ...
Code generation successful: View report
```

Evaluate Raspberry Pi Execution Time

Call the generated PIL function multiple times to get the average execution time.

```
testDur = 50e-3;
numCalls = 100;

for k = 1:numCalls
    x = pinknoise(fs*testDur,'single');
```

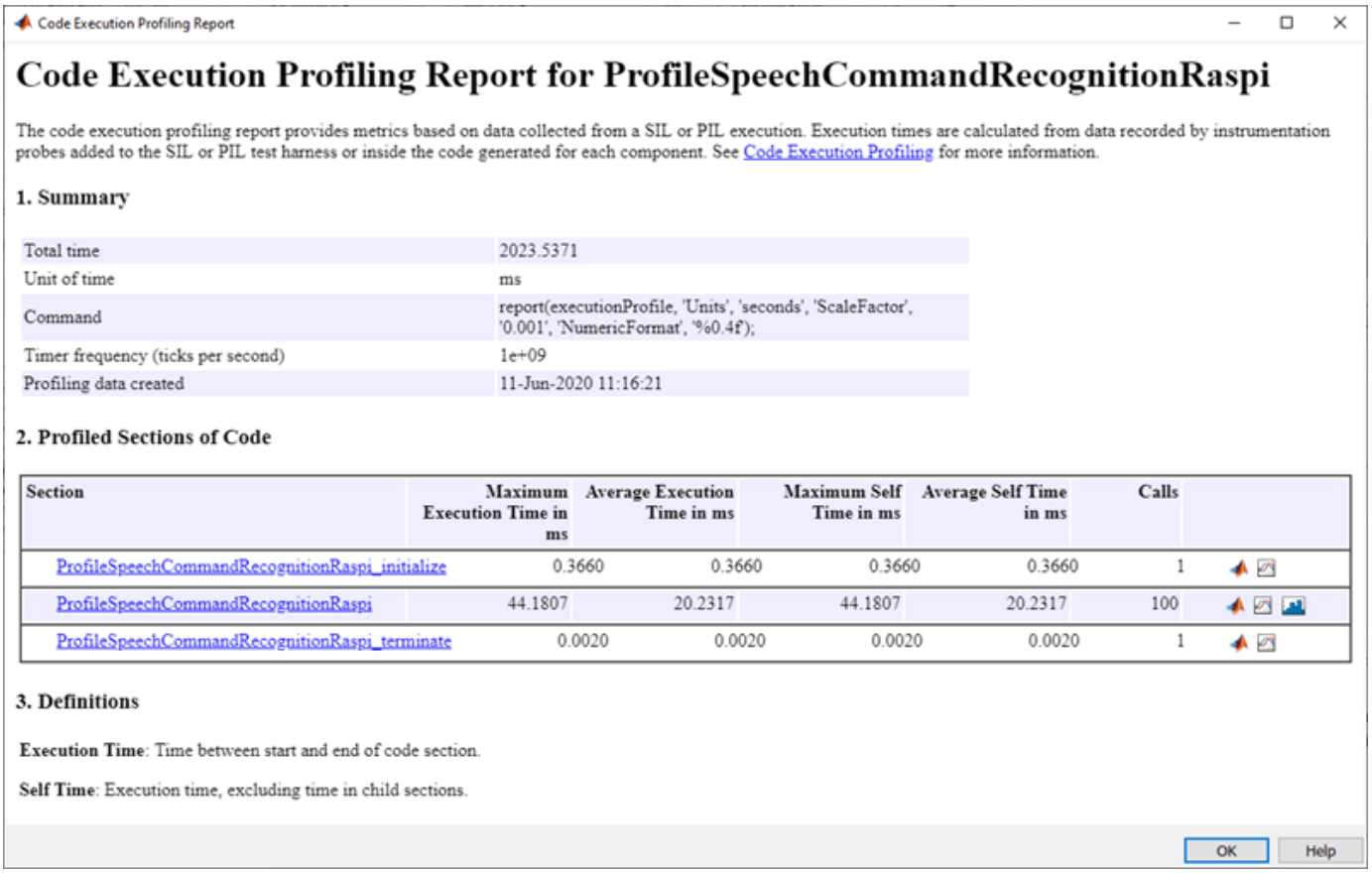
```
    [speechCommandIdx, auditoryFeatures] = ProfileSpeechCommandRecognitionRaspi_pil(x);
end

### Starting application: 'codegen\lib\ProfileSpeechCommandRecognitionRaspi\pil\ProfileSpeechComm
    To terminate execution: clear ProfileSpeechCommandRecognitionRaspi_pil
### Launching application ProfileSpeechCommandRecognitionRaspi.elf...
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.
```

Terminate the PIL execution.

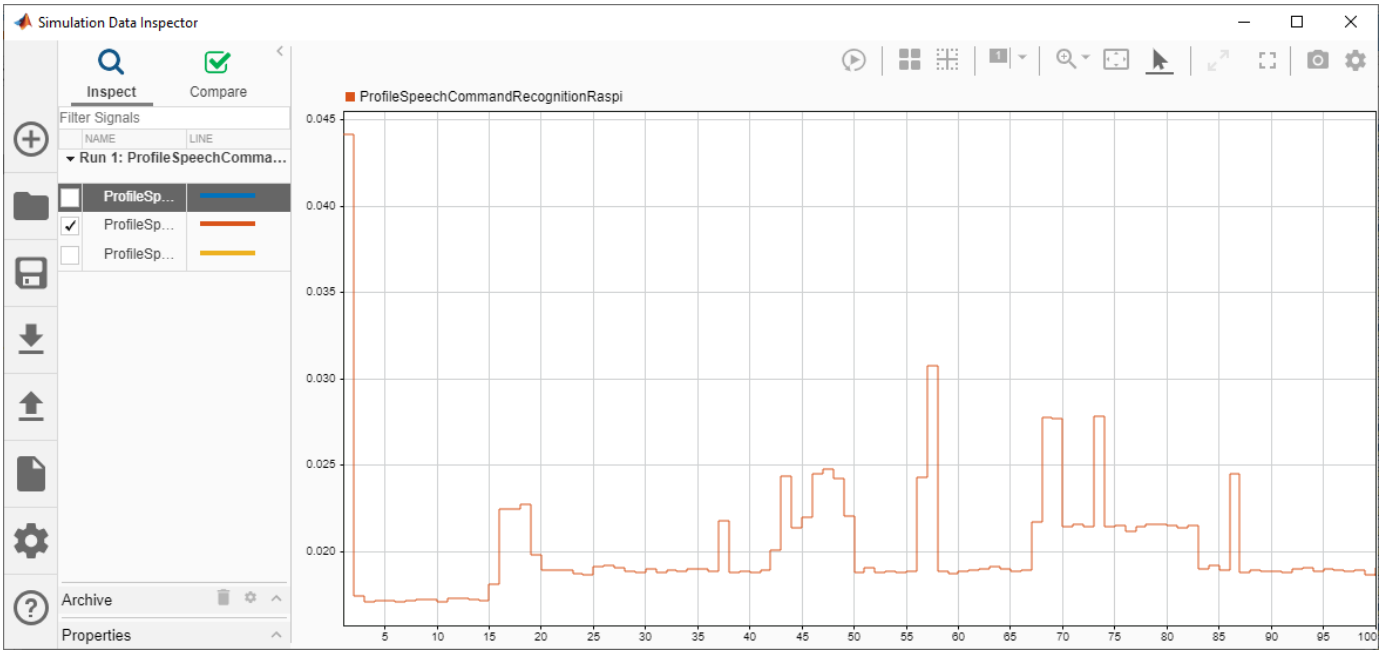
```
clear ProfileSpeechCommandRecognitionRaspi_pil

### Host application produced the following standard output (stdout) and standard error (stderr)

    Execution profiling report: report(getCoderExecutionProfile('ProfileSpeechCommandRecognitionR
```

Generate an execution profile report to evaluate execution time.

```
executionProfile = getCoderExecutionProfile('ProfileSpeechCommandRecognitionRaspi');
report(executionProfile, ...
       'Units','Seconds', ...
       'ScaleFactor','1e-03', ...
       'NumericFormat','%0.4f')

ans =
'C:\ExampleMatlab\ExampleManager\sporwal.Bdoc21b.j1648568\deeplearning_shared-ex00376115\codegen\
```



The maximum execution time taken by the `ProfileSpeechCommandRecognitionRaspi` function is nearly twice the average execution time. You can notice that the execution time is maximum for the first call of the PIL function and it is due to the initialization happening in the first call. The average execution time is approximately 20 ms, which is below the 50 ms budget (audio capture time). The performance is measured on Raspberry Pi 4 Model B Rev 1.1.



Speech Command Recognition Code Generation with Intel MKL-DNN

This example shows how to deploy feature extraction and a convolutional neural network (CNN) for speech command recognition on Intel® processors. To generate the feature extraction and network code, you use MATLAB Coder and the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). In this example, the generated code is a MATLAB executable (MEX) function, which is called by a MATLAB script that displays the predicted speech command along with the time domain signal and auditory spectrogram. For details about audio preprocessing and network training, see “Speech Command Recognition Using Deep Learning” (Audio Toolbox).

Prerequisites

- The MATLAB Coder Interface for Deep Learning Support Package
- Xeon processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2)
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment variables for Intel MKL-DNN

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

Streaming Demonstration in MATLAB

Use the same parameters for the feature extraction pipeline and classification as developed in “Speech Command Recognition Using Deep Learning” (Audio Toolbox).

Define the same sample rate the network was trained on (16 kHz). Define the classification rate and the number of audio samples input per frame. The feature input to the network is a Bark spectrogram that corresponds to 1 second of audio data. The Bark spectrogram is calculated for 25 ms windows with 10 ms hops.

```
fs = 16000;
classificationRate = 20;
samplesPerCapture = fs/classificationRate;

segmentDuration = 1;
segmentSamples = round(segmentDuration*fs);

frameDuration = 0.025;
frameSamples = round(frameDuration*fs);

hopDuration = 0.010;
hopSamples = round(hopDuration*fs);
```

Create an `audioFeatureExtractor` (Audio Toolbox) object to extract 50-band Bark spectrograms without window normalization.

```
afe = audioFeatureExtractor( ...
    'SampleRate',fs, ...
    'FFTLength',512, ...
    'Window',hann(frameSamples,'periodic'), ...
    'OverlapLength',frameSamples - hopSamples, ...
    'barkSpectrum',true);
```

```
numBands = 50;
setExtractorParams(afe,'barkSpectrum','NumBands',numBands,'WindowNormalization',false);
```

Load the pretrained convolutional neural network and labels.

```
load('commandNet.mat')
labels = trainedNet.Layers(end).Classes;
numLabels = numel(labels);
backgroundIdx = find(labels == 'background');
```

Define buffers and decision thresholds to post process network predictions.

```
probBuffer = single(zeros([numLabels,classificationRate/2]));
YBuffer = single(numLabels * ones(1, classificationRate/2));

countThreshold = ceil(classificationRate*0.2);
probThreshold = single(0.7);
```

Create an `audioDeviceReader` (Audio Toolbox) object to read audio from your device. Create a `dsp.AsyncBuffer` (DSP System Toolbox) object to buffer the audio into chunks.

```
adr = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',samplesPerCapture,'OutputDataType','si
audioBuffer = dsp.AsyncBuffer(fs);
```

Create a `dsp.MatrixViewer` (DSP System Toolbox) object and a `timescope` (DSP System Toolbox) object to display the results.

```
matrixViewer = dsp.MatrixViewer("ColorBarLabel","Power per band (dB/Band)", ...
    "XLabel","Frames", ...
    "YLabel","Bark Bands", ...
    "Position",[400 100 600 250], ...
    "ColorLimits",[-4 2.6445], ...
    "AxisOrigin",'Lower left corner', ...
    "Name","Speech Command Recognition Using Deep Learning");

timeScope = timescope('SampleRate', fs, ...
    'YLimits',[-1 1], 'Position', [400 380 600 250], ...
    'Name','Speech Command Recognition Using Deep Learning', ...
    'TimeSpanSource','Property', ...
    'TimeSpan',1, ...
    'BufferLength',fs);

timeScope.YLabel = 'Amplitude';
timeScope.ShowGrid = true;
```

Show the time scope and matrix viewer. Detect commands as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```
show(timeScope)
show(matrixViewer)
timeLimit = 10;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    %% Capture Audio
    x = adr();
    write(audioBuffer,x);
```

```
    y = read(audioBuffer,fs,fs-samplesPerCapture);

    % Compute auditory features
    features = extract(afe,y);
    auditory_features = log10(features + 1e-6);

    % Transpose to get the auditory spectrum
    auditorySpectrum = auditory_features';

    % Perform prediction
    probs = predict(trainedNet, auditory_features);
    [~, YPredicted] = max(probs);

    % Perform statistical post processing
    YBuffer = [YBuffer(2:end),YPredicted];
    probBuffer = [probBuffer(:,2:end),probs(:)];

    [YMode_idx, count] = mode(YBuffer);
    count = single(count);
    maxProb = max(probBuffer(YMode_idx,:));

    if (YMode_idx == single(backgroundIdx) || count < countThreshold || maxProb < probThreshold)
        speechCommandIdx = backgroundIdx;
    else
        speechCommandIdx = YMode_idx;
    end

    % Update plots
    matrixViewer(auditorySpectrum);
    timeScope(x);

    if (speechCommandIdx == backgroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow
end
```

Hide the scopes.

```
hide(matrixViewer)
hide(timeScope)
```

Prepare MATLAB Code for Deployment

To create a function to perform feature extraction compatible with code generation, call `generateMATLABFunction` (Audio Toolbox) on the `audioFeatureExtractor` object. The `generateMATLABFunction` object function creates a standalone function that performs equivalent feature extraction and is compatible with code generation.

```
generateMATLABFunction(afe,'extractSpeechFeatures')
```

The HelperSpeechCommandRecognition supporting function encapsulates the feature extraction and network prediction process demonstrated previously. So that the feature extraction is compatible with code generation, feature extraction is handled by the generated `extractSpeechFeatures` function. So that the network is compatible with code generation, the supporting function uses the `coder.loadDeepLearningNetwork` (MATLAB Coder) function to load the network.

Use the HelperSpeechCommandRecognition function to perform live detection of speech commands.

```
show(timeScope)
show(matrixViewer)
timeLimit = 10;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    x = adr();

    [speechCommandIdx, auditorySpectrum] = HelperSpeechCommandRecognition(x);

    matrixViewer(auditorySpectrum);
    timeScope(x);

    if (speechCommandIdx == backgroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow
end
```

Hide the scopes.

```
hide(timeScope)
hide(matrixViewer)
```

Generate MATLAB Executable

Create a code generation configuration object for generation of an executable program. Specify the target language as C++.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the MKL-DNN library. Attach the configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('mkldnn');
cfg.DeepLearningConfig = dlcfg;
```

Call `codegen` (MATLAB Coder) to generate C++ code for the `HelperSpeechCommandRecognition` function. Specify the configuration object and prototype arguments. A MEX file named `HelperSpeechCommandRecognition_mex` is generated to your current folder.

```
codegen HelperSpeechCommandRecognition -config cfg -args {rand(samplesPerCapture, 1, 'single')} -
```

```
### Compiling function(s) HelperSpeechCommandRecognition ...
-------------------------------------------------------------------------
[1/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWTensorBase.cpp
[2/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWElementwiseAffineLayer.cpp
[3/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWMaxPoolingLayer.cpp
[4/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWInputLayerImpl.cpp
```

```
[5/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWInputLayer.cpp
[6/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWOutputLayer.cpp
[7/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWFCLayer.cpp
[8/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWCNNLayer.cpp
[9/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWOutputLayerImpl.cpp
[10/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWFusedConvReLULayer.cpp
[11/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWMaxPoolingLayerImpl.cpp
[12/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
HelperSpeechCommandRecognition_data.cpp
[13/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
HelperSpeechCommandRecognition_terminate.cpp
[14/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
colon.cpp
[15/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
HelperSpeechCommandRecognition_initialize.cpp
[16/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWElementwiseAffineLayerImpl.cpp
[17/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
rt_nonfinite.cpp
[18/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWFCLayerImpl.cpp
[19/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWFusedConvReLULayerImpl.cpp
[20/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
eml_int_forloop_overflow_check.cpp
[21/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWSoftmaxLayerImpl.cpp
[22/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
stft.cpp
[23/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
sort.cpp
[24/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWSoftmaxLayer.cpp
[25/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
extractSpeechFeatures.cpp
[26/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
HelperSpeechCommandRecognition.cpp
[27/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
DeepLearningNetwork.cpp
[28/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
sortIdx.cpp
[29/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
_coder_HelperSpeechCommandRecognition_api.cpp
[30/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
MWCNNLayerImpl.cpp
[31/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
permute.cpp
[32/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
predict.cpp
[33/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATI
_coder_HelperSpeechCommandRecognition_info.cpp
```

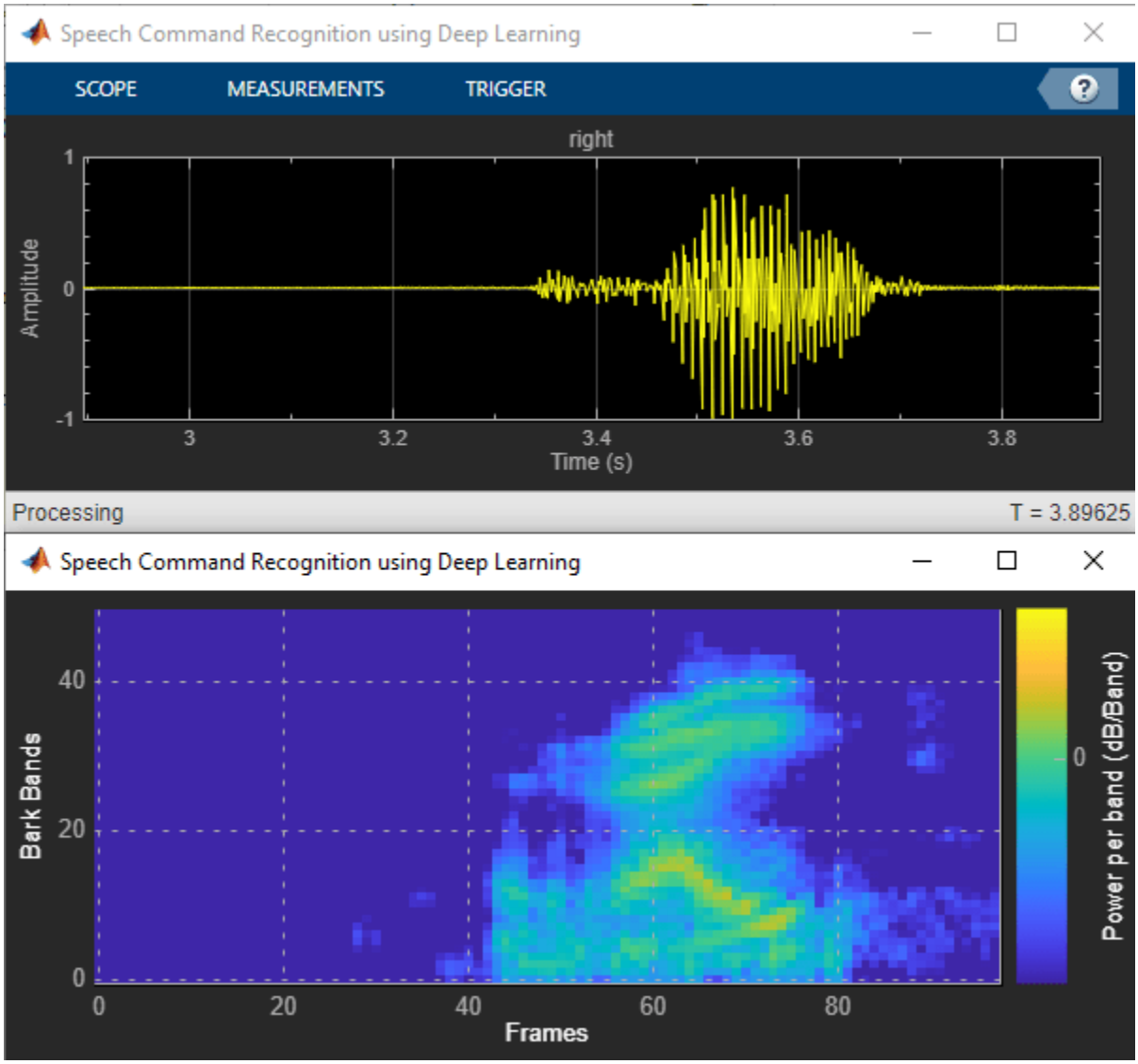
```
[34/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
HelperSpeechCommandRecognition_mexutil.cpp
[35/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWTargetNetworkImpl.cpp
[36/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
_coder_HelperSpeechCommandRecognition_mex.cpp
[37/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
indexShapeCheck.cpp
[38/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
cpp_mexapi_version.cpp
[39/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWCustomLayerForMKLDNN.cpp
[40/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
MWMkldnnUtils.cpp
[41/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
mtimes.cpp
[42/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
computeDFT.cpp
[43/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
formatSTFTOutput.cpp
[44/45] cl /c /Zp8 /GR /W3 /EHs /nologo /MD /D_CRT_SECURE_NO_DEPRECATE /D_SCL_SECURE_NO_DEPRECATE
AsyncBuffer.cpp
[45/45] link build\win64\MWCNNLayer.obj build\win64\MWElementwiseAffineLayer.obj build\win64\MWFC
   Creating library HelperSpeechCommandRecognition_mex.lib and object HelperSpeechCommandRecogni

------------------------------------------------------------------------
### Generating compilation report ...
Code generation successful: View report
```

Perform Speech Command Recognition Using Deployed Code

Show the time scope and matrix viewer. Detect commands using the generated MEX for as long as both the time scope and matrix viewer are open or until the time limit is reached. To stop the live detection before the time limit is reached, close the time scope window or matrix viewer window.

```
show(timeScope)
show(matrixViewer)

timeLimit = 20;

tic
while isVisible(timeScope) && isVisible(matrixViewer) && toc < timeLimit
    x = adr();

    [speechCommandIdx, auditorySpectrum] = HelperSpeechCommandRecognition_mex(x);

    matrixViewer(auditorySpectrum);
    timeScope(x);

    if (speechCommandIdx == backgroundIdx)
        timeScope.Title = ' ';
    else
        timeScope.Title = char(labels(speechCommandIdx));
    end
    drawnow
end
```

```
hide(matrixViewer)
hide(timeScope)
```



Evaluate MEX Execution Time

Use `tic` and `toc` to compare the execution time to run the simulation completely in MATLAB with the execution time of the MEX function.

Measure the performance of the simulation code.

```
testDur = 50e-3;
x = pinknoise(fs*testDur,'single');
numLoops = 100;
tic
for k = 1:numLoops
    [speechCommandIdx, auditory_features] = HelperSpeechCommandRecognition(x);
end
```

```
exeTime = toc;
fprintf('SIM execution time per 50 ms of audio = %0.4f ms\n',(exeTime/numLoops)*1000);
```

```
SIM execution time per 50 ms of audio = 6.8212 ms
```

Measure the performance of the MEX code.

```
tic
for k = 1:numLoops
    [speechCommandIdx, auditory_features] = HelperSpeechCommandRecognition_mex(x);
end
exeTimeMex = toc;
fprintf('MEX execution time per 50 ms of audio = %0.4f ms\n',(exeTimeMex/numLoops)*1000);
```

```
MEX execution time per 50 ms of audio = 1.3347 ms
```

Evaluate the performance gained from using the MEX function. This performance test is performed on a machine using NVIDIA Quadro P620 (Version 26) GPU and Intel(R) Xeon(R) W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = exeTime/exeTimeMex
```

```
PerformanceGain = 5.1107
```

Train Generative Adversarial Network (GAN) for Sound Synthesis

This example shows how to train and use a generative adversarial network (GAN) to generate sounds.

Introduction

In generative adversarial networks, a generator and a discriminator compete against each other to improve the generation quality.

GANs have generated significant interest in the field of audio and speech processing. Applications include text-to-speech synthesis, voice conversion, and speech enhancement.

This example trains a GAN for unsupervised synthesis of audio waveforms. The GAN in this example generates drumbeat sounds. The same approach can be followed to generate other types of sound, including speech.

Synthesize Audio with Pre-Trained GAN

Before you train a GAN from scratch, you will use a pretrained GAN generator to synthesize drum beats.

Download the pretrained generator.

```
matFileName = 'drumGeneratorWeights.mat';
if ~exist(matFileName,'file')
    websave(matFileName,'https://www.mathworks.com/supportfiles/audio/GanAudioSynthesis/drumGene
end
```

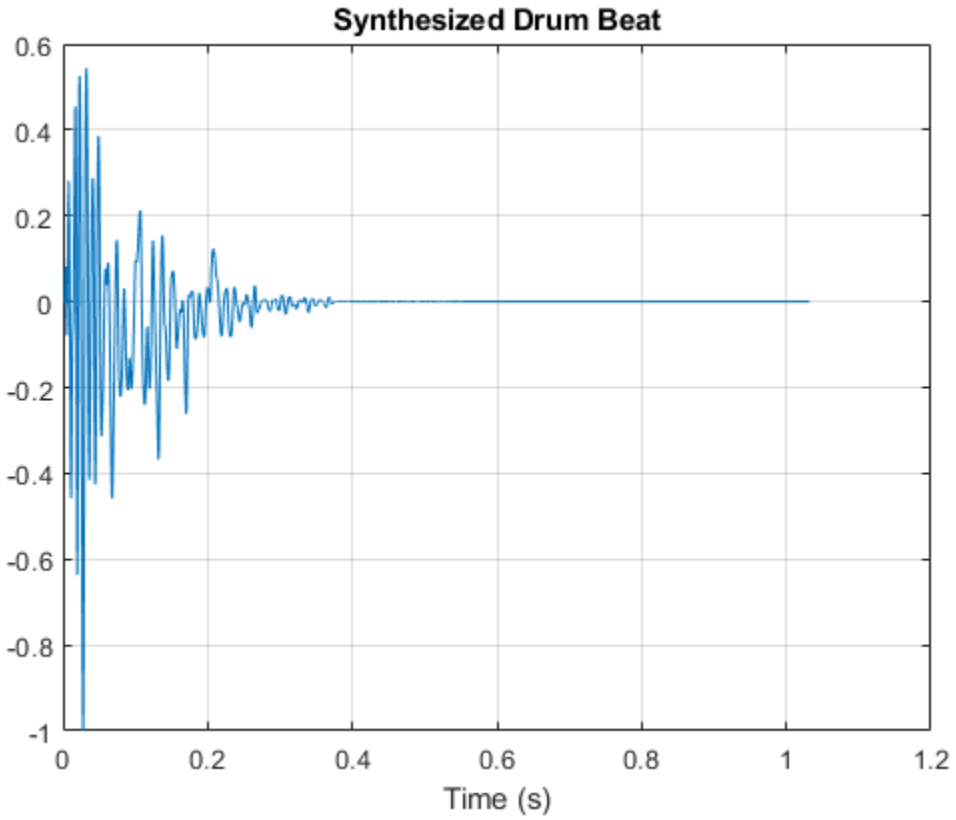
The function `synthesizeDrumBeat` calls a pretrained network to synthesize a drumbeat sampled at 16 kHz. The `synthesizeDrumBeat` function is included at the end of this example.

Synthesize a drumbeat and listen to it.

```
drum = synthesizeDrumBeat;

fs = 16e3;
sound(drum,fs)
```

Plot the synthesized drumbeat.

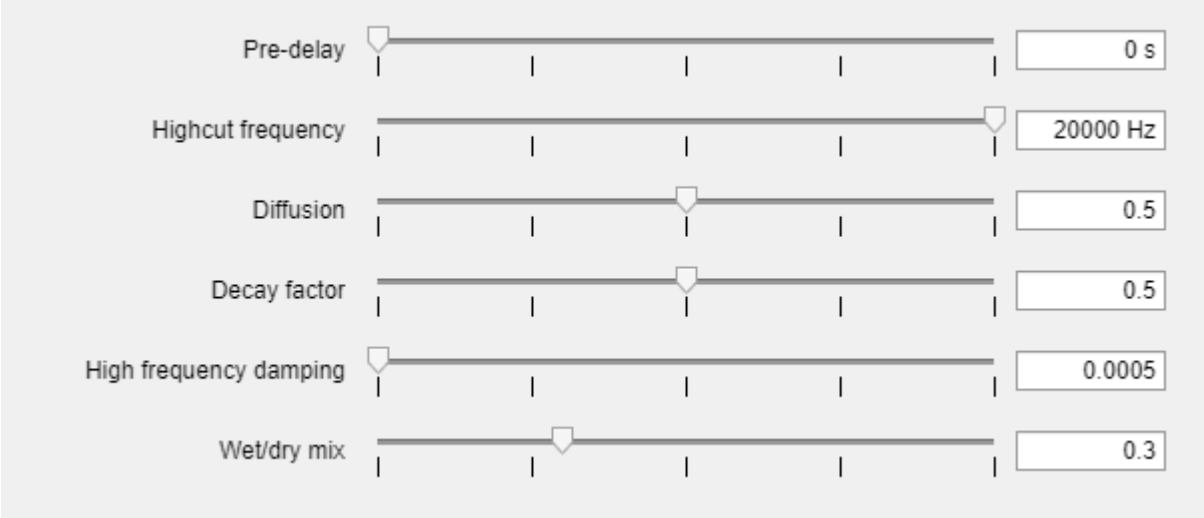
```
t = (0:length(drum)-1)/fs;
plot(t,drum)
grid on
xlabel('Time (s)')
title('Synthesized Drum Beat')
```



You can use the drumbeat synthesizer with other audio effects to create more complex applications. For example, you can apply reverberation to the synthesized drum beats.

Create a `reverberator` (Audio Toolbox) object and open its parameter tuner UI. This UI enables you to tune the `reverberator` parameters as the simulation runs.

```
reverb = reverberator('SampleRate',fs);
parameterTuner(reverb);
```

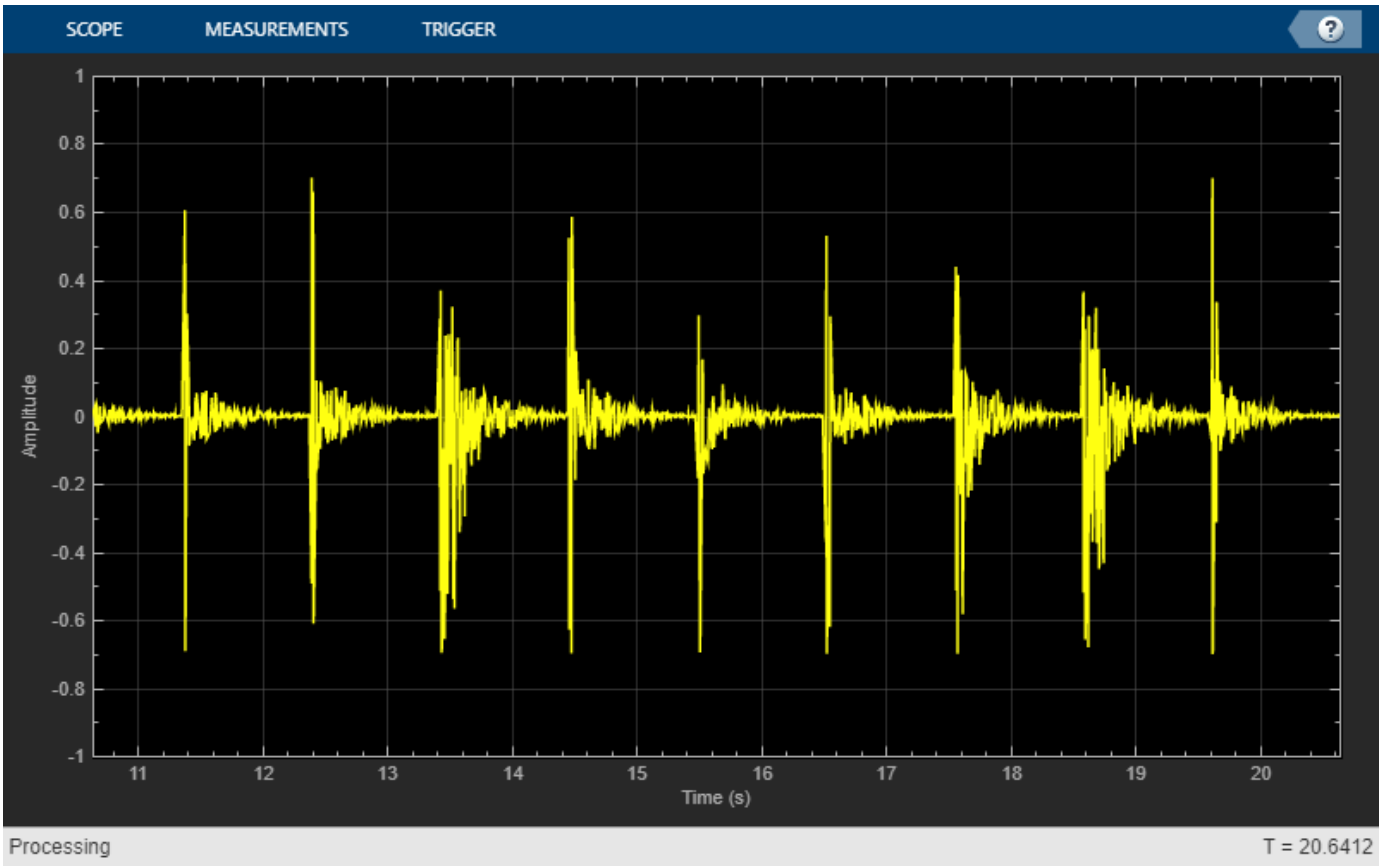


Create a time scope object to visualize the drum beats.

```
ts = timescope('SampleRate',fs, ...
    'TimeSpanSource','Property', ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',10, ...
    'BufferLength',10*256*64, ...
    'ShowGrid',true, ...
    'YLimits',[-1 1]);
```

In a loop, synthesize the drum beats and apply reverberation. Use the parameter tuner UI to tune reverberation. If you want to run the simulation for a longer time, increase the value of the `loopCount` parameter.

```
loopCount = 20;
for ii = 1:loopCount
    drum = synthesizeDrumBeat;
    drum = reverb(drum);
    ts(drum(:,1));
    soundsc(drum,fs)
    pause(0.5)
end
```



Train the GAN

Now that you have seen the pretrained drumbeat generator in action, you can investigate the training process in detail.

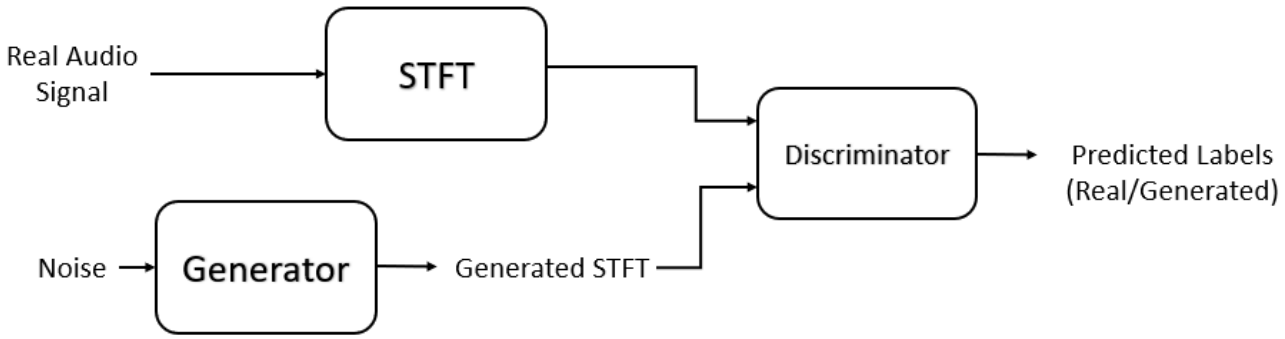
A GAN is a type of deep learning network that generates data with characteristics similar to the training data.

A GAN consists of two networks that train together, a *generator* and a *discriminator*:

- Generator - Given a vector or random values as input, this network generates data with the same structure as the training data. It is the generator's job to fool the discriminator.
- Discriminator - Given batches of data containing observations from both the training data and the generated data, this network attempts to classify the observations as real or generated.

To maximize the performance of the generator, maximize the loss of the discriminator when given generated data. That is, the objective of the generator is to generate data that the discriminator classifies as real. To maximize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. Ideally, these strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.

In this example, you train the generator to create fake time-frequency short-time Fourier transform (STFT) representations of drum beats. You train the discriminator to identify real STFTs. You create the real STFTs by computing the STFT of short recordings of real drum beats.



Load Training Data

Train a GAN using the Drum Sound Effects dataset [1]. Download and extract the dataset.

```
url = 'http://deepyeti.ucsd.edu/cdonahue/wavegan/data/drums.tar.gz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'drums_dataset.tgz');

drumsFolder = fullfile(downloadFolder,'drums');
if ~exist(drumsFolder,'dir')
    disp('Downloading Drum Sound Effects Dataset (218 MB)...')
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

Create an `audioDatastore` (Audio Toolbox) object that points to the drums dataset.

```
ads = audioDatastore(drumsFolder,'IncludeSubfolders',true);
```

Define Generator Network

Define a network that generates STFTs from 1-by-1-by-100 arrays of random values. Create a network that upscales 1-by-1-by-100 arrays to 128-by-128-by-1 arrays using a fully connected layer followed by a reshape layer and a series of transposed convolution layers with ReLU layers.

This figure shows the dimensions of the signal as it travels through the generator. The generator architecture is defined in Table 4 of [1].


Random
1.1.100
fc
fullyConnected...
256*64.1
reshape
4,4,1024
relu
reluLayer
4.4.1024
transposed-co...
transposedCon...
8.8.512
relu
reluLayer
8.8.512
transposed-co...
transposedCon...
16.16.256
relu
reluLayer
16,16,256
transposed-co...
transposedCon...
32,32,128
relu
reluLayer
32.32.128
transposed-co...
transposedCon...
64.64.64
relu
reluLayer
64.64.64
transposed-co...
transposedCon...
128.128
tanh
tanhLayer

The generator network is defined in `modelGenerator`, which is included at the end of this example.

Define Discriminator Network

Define a network that classifies real and generated 128-by-128 STFTs.

Create a network that takes 128-by-128 images and outputs a scalar prediction score using a series of convolution layers with leaky ReLU layers followed by a fully connected layer.

This figure shows the dimensions of the signal as it travels through the discriminator. The discriminator architecture is defined in Table 5 of [1].


imageinput
imageInputLayer
128.128.1
conv
convolution2dL...
64.64.64
leakyrelu
leakyReluLayer
64,64,64
conv
convolution2dL...
32.32.128
leakyrelu
leakyReluLayer
32.32.128
conv
convolution2dL...
16.16.256
leakyrelu
leakyReluLayer
16,16,256
conv
convolution2dL...
8,8,512
leakyrelu
leakyReluLayer
8,8,512
conv
convolution2dL...
4.4.1024
leakyrelu
leakyReluLayer
4,4,1024
Reshape
64*256.1
fc
fullyConnected...
1,1

The discriminator network is defined in `modelDiscriminator`, which is included at the end of this example.

Generate Real Drumbeat Training Data

Generate STFT data from the drumbeat signals in the datastore.

Define the STFT parameters.

```
fftLength = 256;
win = hann(fftLength,'periodic');
overlapLength = 128;
```

To speed up processing, distribute the feature extraction across multiple workers using `parfor`.

First, determine the number of partitions for the dataset. If you do not have Parallel Computing Toolbox™, use a single partition.

```
if ~isempty(ver('parallel'))
    pool = gcp;
    numPar = numpartitions(ads,pool);
else
    numPar = 1;
end
```

For each partition, read from the datastore and compute the STFT.

```
parfor ii = 1:numPar

    subds = partition(ads,numPar,ii);
    STrain = zeros(fftLength/2+1,128,1,numel(subds.Files));

    for idx = 1:numel(subds.Files)

        x = read(subds);

        if length(x) > fftLength*64
            % Lengthen the signal if it is too short
            x = x(1:fftLength*64);
        end

        % Convert from double-precision to single-precision
        x = single(x);

        % Scale the signal
        x = x ./ max(abs(x));

        % Zero-pad to ensure stft returns 128 windows.
        x = [x ; zeros(overlapLength,1,'like',x)];

        S0 = stft(x,'Window',win,'OverlapLength',overlapLength,'Centered',false);

        % Convert from two-sided to one-sided.
        S = S0(1:129,:);
        S = abs(S);
        STrain(:,:,:,idx) = S;
    end
    STrainC{ii} = STrain;
end
```

Convert the output to a four-dimensional array with STFTs along the fourth dimension.

```
STrain = cat(4,STrainC{:});
```

Convert the data to the log scale to better align with human perception.

```
STrain = log(STrain + 1e-6);
```

Normalize training data to have zero mean and unit standard deviation.

Compute the STFT mean and standard deviation of each frequency bin.

```
SMean = mean(STrain,[2 3 4]);
SStd = std(STrain,1,[2 3 4]);
```

Normalize each frequency bin.

```
STrain = (STrain-SMean)./SStd;
```

The computed STFTs have unbounded values. Following the approach in [1], make the data bounded by clipping the spectra to 3 standard deviations and rescaling to [-1 1].

```
STrain = STrain/3;
Y = reshape(STrain,numel(STrain),1);
Y(Y<-1) = -1;
Y(Y>1) = 1;
STrain = reshape(Y,size(STrain));
```

Discard the last frequency bin to force the number of STFT bins to a power of two (which works well with convolutional layers).

```
STrain = STrain(1:end-1,:,:,:);
```

Permute the dimensions in preparation for feeding to the discriminator.

```
STrain = permute(STrain,[2 1 3 4]);
```

Specify Training Options

Train with a mini-batch size of 64 for 1000 epochs.

```
maxEpochs = 1000;
miniBatchSize = 64;
```

Compute the number of iterations required to consume the data.

```
numIterationsPerEpoch = floor(size(STrain,4)/miniBatchSize);
```

Specify the options for Adam optimization. Set the learn rate of the generator and discriminator to `0.0002`. For both networks, use a gradient decay factor of 0.5 and a squared gradient decay factor of 0.999.

```
learnRateGenerator = 0.0002;
learnRateDiscriminator = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™.

```
executionEnvironment = "auto";
```

Initialize the generator and discriminator weights. The `initializeGeneratorWeights` and `initializeDiscriminatorWeights` functions return random weights obtained using Glorot uniform initialization. The functions are included at the end of this example.

```
generatorParameters = initializeGeneratorWeights;
discriminatorParameters = initializeDiscriminatorWeights;
```

Train Model

Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

For each epoch, shuffle the training data and loop over mini-batches of data.

For each mini-batch:

- Generate a `dlarray` object containing an array of random values for the generator network.
- For GPU training, convert the data to a `gpuArray` (Parallel Computing Toolbox) object.
- Evaluate the model gradients using `dlfeval` and the helper functions, `modelDiscriminatorGradients` and `modelGeneratorGradients`.
- Update the network parameters using the `adamupdate` function.

Initialize the parameters for Adam.

```
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

You can set `saveCheckpoints` to `true` to save the updated weights and states to a MAT file every ten epochs. You can then use this MAT file to resume training if it is interrupted. For the purpose of this example, set `saveCheckpoints` to `false`.

```
saveCheckpoints = false;
```

Specify the length of the generator input.

```
numLatentInputs = 100;
```

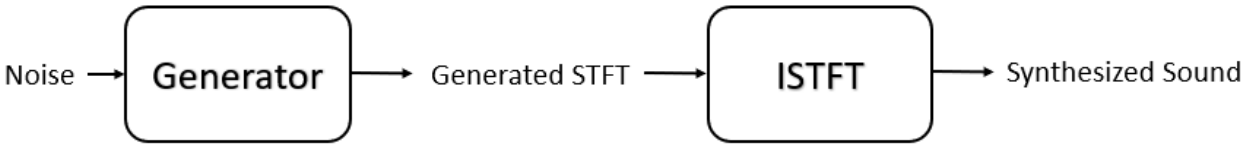
Train the GAN. This can take multiple hours to run.

```
iteration = 0;

for epoch = 1:maxEpochs

    % Shuffle the data.
    idx = randperm(size(STrain,4));
    STrain = STrain(:,:,:,idx);

    % Loop over mini-batches.
    for index = 1:numIterationsPerEpoch

        iteration = iteration + 1;

        % Read mini-batch of data.
```

```
            dlX = STrain(:,:,:,(index-1)*miniBatchSize+1:index*miniBatchSize);
            dlX = dlarray(dlX,'SSCB');

            % Generate latent inputs for the generator network.
            Z = 2 * ( rand(1,1,numLatentInputs,miniBatchSize,'single') - 0.5 ) ;
            dlZ = dlarray(Z);

            % If training on a GPU, then convert data to gpuArray.
            if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
                dlZ = gpuArray(dlZ);
                dlX = gpuArray(dlX);
            end

            % Evaluate the discriminator gradients using dlfeval and the
            % |modelDiscriminatorGradients| helper function.
            gradientsDiscriminator = ...
                dlfeval(@modelDiscriminatorGradients,discriminatorParameters,generatorParameters,dlX

            % Update the discriminator network parameters.
            [discriminatorParameters,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
                adamupdate(discriminatorParameters,gradientsDiscriminator, ...
                trailingAvgDiscriminator,trailingAvgSqDiscriminator,iteration, ...
                learnRateDiscriminator,gradientDecayFactor,squaredGradientDecayFactor);

            % Generate latent inputs for the generator network.
            Z = 2 * ( rand(1,1,numLatentInputs,miniBatchSize,'single') - 0.5 ) ;
            dlZ = dlarray(Z);

            % If training on a GPU, then convert data to gpuArray.
            if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
                dlZ = gpuArray(dlZ);
            end

            % Evaluate the generator gradients using dlfeval and the
            % |modelGeneratorGradients| helper function.
            gradientsGenerator  = ...
                dlfeval(@modelGeneratorGradients,discriminatorParameters,generatorParameters,dlZ);

            % Update the generator network parameters.
            [generatorParameters,trailingAvgGenerator,trailingAvgSqGenerator] = ...
                adamupdate(generatorParameters,gradientsGenerator, ...
                trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
                learnRateGenerator,gradientDecayFactor,squaredGradientDecayFactor);
        end

        % Every 10 iterations, save a training snapshot to a MAT file.
        if saveCheckpoints && mod(epoch,10)==0
            fprintf('Epoch %d out of %d complete\n',epoch,maxEpochs);
            % Save checkpoint in case training is interrupted.
            save('audioganncheckpoint.mat',...
                'generatorParameters','discriminatorParameters',...
                'trailingAvgDiscriminator','trailingAvgSqDiscriminator',...
                'trailingAvgGenerator','trailingAvgSqGenerator','iteration');
        end
    end
```

Synthesize Sounds

Now that you have trained the network, you can investigate the synthesis process in more detail.



The trained drumbeat generator synthesizes short-time Fourier transform (STFT) matrices from input arrays of random values. An inverse STFT (ISTFT) operation converts the time-frequency STFT to a synthesized time-domain audio signal.

Load the weights of a pretrained generator. These weights were obtained by running the training highlighted in the previous section for 1000 epochs.

```
load(matFileName,'generatorParameters','SMean','SStd');
```

The generator takes 1-by-1-by-100 vectors of random values as an input. Generate a sample input vector.

```
numLatentInputs = 100;
dlZ = dlarray(2 * ( rand(1,1,numLatentInputs,1,'single') - 0.5 ));
```

Pass the random vector to the generator to create an STFT image. `generatorParameters` is a structure containing the weights of the pretrained generator.

```
dlXGenerated = modelGenerator(dlZ,generatorParameters);
```

Convert the STFT `dlarray` to a single-precision matrix.

```
S = dlXGenerated.extractdata;
```

Transpose the STFT to align its dimensions with the `istft` function.

```
S = S.';
```

The STFT is a 128-by-128 matrix, where the first dimension represents 128 frequency bins linearly spaced from 0 to 8 kHz. The generator was trained to generate a one-sided STFT from an FFT length of 256, with the last bin omitted. Reintroduce that bin by inserting a row of zeros into the STFT.

```
S = [S ; zeros(1,128)];
```

Revert the normalization and scaling steps used when you generated the STFTs for training.

```
S = S * 3;
S = (S.*SStd) + SMean;
```

Convert the STFT from the log domain to the linear domain.

```
S = exp(S);
```

Convert the STFT from one-sided to two-sided.

```
S = [S; S(end-1:-1:2,:)];
```

Pad with zeros to remove window edge-effects.

```
S = [zeros(256,100) S zeros(256,100)];
```

The STFT matrix does not contain any phase information. Use a fast version of the Griffin-Lim algorithm with 20 iterations to estimate the signal phase and produce audio samples.

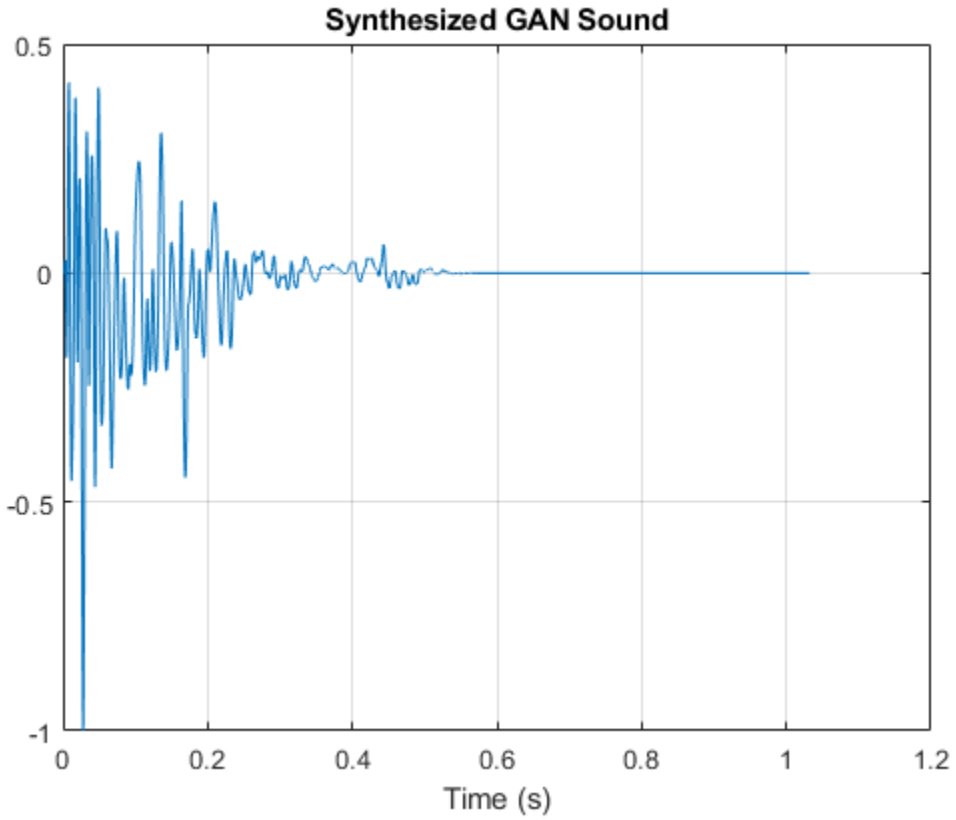
```
myAudio = stftmag2sig(S,256, ...
    'FrequencyRange','twosided', ...
    'Window',hann(256,'periodic'), ...
    'OverlapLength',128, ...
    'MaxIterations',20, ...
    'Method','fgla');
myAudio = myAudio./max(abs(myAudio),[],'all');
myAudio = myAudio(128*100:end-128*100);
```

Listen to the synthesized drumbeat.

```
sound(myAudio,fs)
```

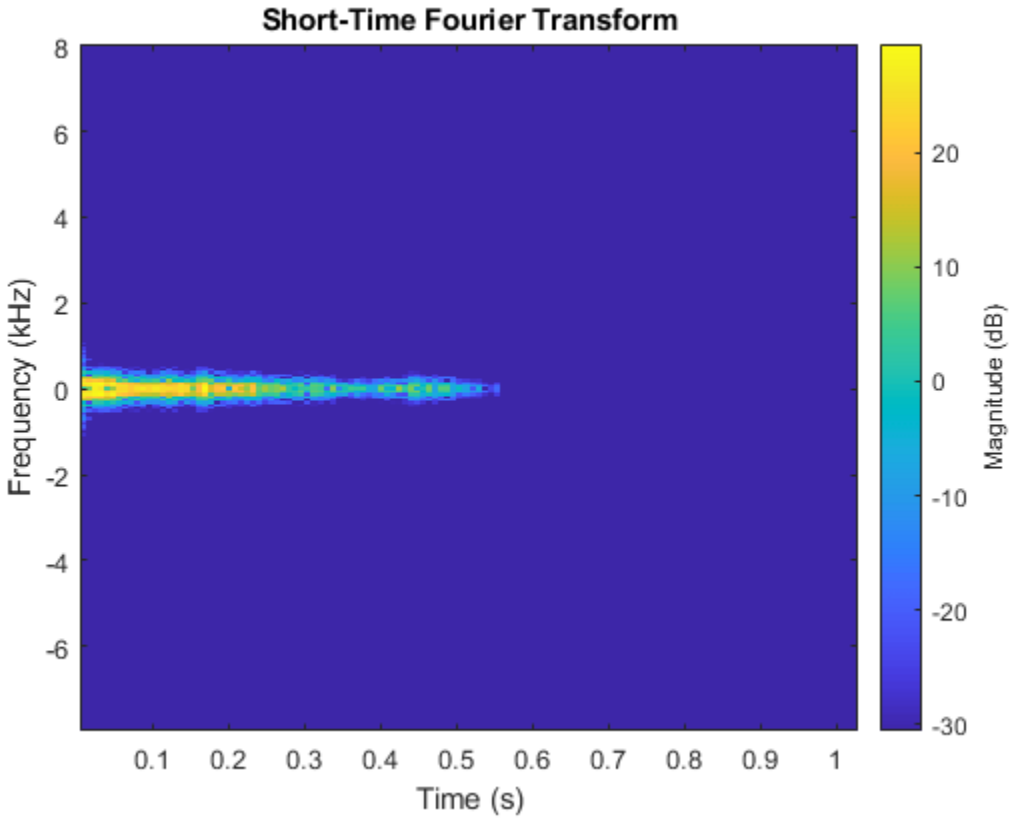
Plot the synthesized drumbeat.

```
t = (0:length(myAudio)-1)/fs;
plot(t,myAudio)
grid on
xlabel('Time (s)')
title('Synthesized GAN Sound')
```



Plot the STFT of the synthesized drumbeat.

```
figure
stft(myAudio,fs,'Window',hann(256,'periodic'),'OverlapLength',128);
```



Model Generator Function

The `modelGenerator` function upscales 1-by-1-by-100 arrays (dlX) to 128-by-128-by-1 arrays (dlY). `parameters` is a structure holding the weights of the generator layers. The generator architecture is defined in Table 4 of [1].

```
function dlY = modelGenerator(dlX,parameters)

dlY = fullyconnect(dlX,parameters.FC.Weights,parameters.FC.Bias,'Dataformat','SSCB');

dlY = reshape(dlY,[1024 4 4 size(dlY,2)]);
dlY = permute(dlY,[3 2 1 4]);
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv1.Weights,parameters.Conv1.Bias,'Stride' ,2 ,'Cropping','sa
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv2.Weights,parameters.Conv2.Bias,'Stride' ,2 ,'Cropping','sa
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv3.Weights,parameters.Conv3.Bias,'Stride' ,2 ,'Cropping','sa
dlY = relu(dlY);
```

```
dlY = dltranspconv(dlY,parameters.Conv4.Weights,parameters.Conv4.Bias,'Stride' ,2 ,'Cropping','sa
dlY = relu(dlY);

dlY = dltranspconv(dlY,parameters.Conv5.Weights,parameters.Conv5.Bias,'Stride' ,2 ,'Cropping','sa
dlY = tanh(dlY);
end
```

Model Discriminator Function

The `modelDiscriminator` function takes 128-by-128 images and outputs a scalar prediction score. The discriminator architecture is defined in Table 5 of [1].

```
function dlY = modelDiscriminator(dlX,parameters)

dlY = dlconv(dlX,parameters.Conv1.Weights,parameters.Conv1.Bias,'Stride' ,2 ,'Padding','same');
dlY = leakyrelu(dlY,0.2);

dlY = dlconv(dlY,parameters.Conv2.Weights,parameters.Conv2.Bias,'Stride' ,2 ,'Padding','same');
dlY = leakyrelu(dlY,0.2);

dlY = dlconv(dlY,parameters.Conv3.Weights,parameters.Conv3.Bias,'Stride' ,2 ,'Padding','same');
dlY = leakyrelu(dlY,0.2);

dlY = dlconv(dlY,parameters.Conv4.Weights,parameters.Conv4.Bias,'Stride' ,2 ,'Padding','same');
dlY = leakyrelu(dlY,0.2);

dlY = dlconv(dlY,parameters.Conv5.Weights,parameters.Conv5.Bias,'Stride' ,2 ,'Padding','same');
dlY = leakyrelu(dlY,0.2);

dlY = stripdims(dlY);
dlY = permute(dlY,[3 2 1 4]);
dlY = reshape(dlY,4*4*64*16,numel(dlY)/(4*4*64*16));

weights = parameters.FC.Weights;
bias = parameters.FC.Bias;
dlY = fullyconnect(dlY,weights,bias,'Dataformat','CB');

end
```

Model Discriminator Gradients Function

The `modelDiscriminatorGradients` functions takes as input the generator and discriminator parameters `generatorParameters` and `discriminatorParameters`, a mini-batch of input data `dlX`, and an array of random values `dlZ`, and returns the gradients of the discriminator loss with respect to the learnable parameters in the networks.

```
function gradientsDiscriminator = modelDiscriminatorGradients(discriminatorParameters , generator

% Calculate the predictions for real data with the discriminator network.
dlYPred = modelDiscriminator(dlX,discriminatorParameters);

% Calculate the predictions for generated data with the discriminator network.
dlXGenerated     = modelGenerator(dlZ,generatorParameters);
dlYPredGenerated = modelDiscriminator(dlarray(dlXGenerated,'SSCB'),discriminatorParameters);

% Calculate the GAN loss
lossDiscriminator = ganDiscriminatorLoss(dlYPred,dlYPredGenerated);
```

```
% For each network, calculate the gradients with respect to the loss.
gradientsDiscriminator = dlgradient(lossDiscriminator,discriminatorParameters);

end
```

Model Generator Gradients Function

The `modelGeneratorGradients` function takes as input the discriminator and generator learnable parameters and an array of random values `dlZ`, and returns the gradients of the generator loss with respect to the learnable parameters in the networks.

```
function gradientsGenerator = modelGeneratorGradients(discriminatorParameters, generatorParamete

% Calculate the predictions for generated data with the discriminator network.
dlXGenerated = modelGenerator(dlZ,generatorParameters);
dlYPredGenerated = modelDiscriminator(dlarray(dlXGenerated,'SSCB'),discriminatorParameters);

% Calculate the GAN loss
lossGenerator = ganGeneratorLoss(dlYPredGenerated);

% For each network, calculate the gradients with respect to the loss.
gradientsGenerator = dlgradient(lossGenerator, generatorParameters);

end
```

Discriminator Loss Function

The objective of the discriminator is to not be fooled by the generator. To maximize the probability that the discriminator successfully discriminates between the real and generated images, minimize the discriminator loss function. The loss function for the generator follows the DCGAN approach highlighted in [1].

```
function  lossDiscriminator = ganDiscriminatorLoss(dlYPred,dlYPredGenerated)

fake = dlarray(zeros(1,size(dlYPred,2)));
real = dlarray(ones(1,size(dlYPred,2)));

D_loss = mean(sigmoid_cross_entropy_with_logits(dlYPredGenerated,fake));
D_loss = D_loss + mean(sigmoid_cross_entropy_with_logits(dlYPred,real));
lossDiscriminator  = D_loss / 2;
end
```

Generator Loss Function

The objective of the generator is to generate data that the discriminator classifies as "real". To maximize the probability that images from the generator are classified as real by the discriminator, minimize the generator loss function. The loss function for the generator follows the deep convolutional generative adverarial network (DCGAN) approach highlighted in [1].

```
function lossGenerator = ganGeneratorLoss(dlYPredGenerated)
real = dlarray(ones(1,size(dlYPredGenerated,2)));
lossGenerator = mean(sigmoid_cross_entropy_with_logits(dlYPredGenerated,real));
end
```

Discriminator Weights Initializer

`initializeDiscriminatorWeights` initializes discriminator weights using the Glorot algorithm.

```
function discriminatorParameters = initializeDiscriminatorWeights

filterSize = [5 5];
dim = 64;

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 1 dim]);
bias = zeros(1,1,dim,'single');
discriminatorParameters.Conv1.Weights = dlarray(weights);
discriminatorParameters.Conv1.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) dim 2*dim]);
bias = zeros(1,1,2*dim,'single');
discriminatorParameters.Conv2.Weights = dlarray(weights);
discriminatorParameters.Conv2.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 2*dim 4*dim]);
bias = zeros(1,1,4*dim,'single');
discriminatorParameters.Conv3.Weights = dlarray(weights);
discriminatorParameters.Conv3.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 4*dim 8*dim]);
bias = zeros(1,1,8*dim,'single');
discriminatorParameters.Conv4.Weights = dlarray(weights);
discriminatorParameters.Conv4.Bias = dlarray(bias);

% Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 8*dim 16*dim]);
bias = zeros(1,1,16*dim,'single');
discriminatorParameters.Conv5.Weights = dlarray(weights);
discriminatorParameters.Conv5.Bias = dlarray(bias);

% fully connected
weights = iGlorotInitialize([1,4 * 4 * dim * 16]);
bias = zeros(1,1,'single');
discriminatorParameters.FC.Weights = dlarray(weights);
discriminatorParameters.FC.Bias = dlarray(bias);
end
```

Generator Weights Initializer

`initializeGeneratorWeights` initializes generator weights using the Glorot algorithm.

```
function generatorParameters = initializeGeneratorWeights

dim = 64;

% Dense 1
weights = iGlorotInitialize([dim*256,100]);
bias = zeros(dim*256,1,'single');
generatorParameters.FC.Weights = dlarray(weights);
generatorParameters.FC.Bias = dlarray(bias);

filterSize = [5 5];
```

```
% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 8*dim 16*dim]);
bias = zeros(1,1,dim*8,'single');
generatorParameters.Conv1.Weights = dlarray(weights);
generatorParameters.Conv1.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 4*dim 8*dim]);
bias = zeros(1,1,dim*4,'single');
generatorParameters.Conv2.Weights = dlarray(weights);
generatorParameters.Conv2.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 2*dim 4*dim]);
bias = zeros(1,1,dim*2,'single');
generatorParameters.Conv3.Weights = dlarray(weights);
generatorParameters.Conv3.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) dim 2*dim]);
bias = zeros(1,1,dim,'single');
generatorParameters.Conv4.Weights = dlarray(weights);
generatorParameters.Conv4.Bias = dlarray(bias);

% Trans Conv2D
weights = iGlorotInitialize([filterSize(1) filterSize(2) 1 dim]);
bias = zeros(1,1,1,'single');
generatorParameters.Conv5.Weights = dlarray(weights);
generatorParameters.Conv5.Bias = dlarray(bias);
end
```

Synthesize Drumbeat

`synthesizeDrumBeat` uses a pretrained network to synthesize drum beats.

```
function y = synthesizeDrumBeat

persistent pGeneratorParameters pMean pSTD
if isempty(pGeneratorParameters)
    % If the MAT file does not exist, download it
    filename = 'drumGeneratorWeights.mat';
    load(filename,'SMean','SStd','generatorParameters');
    pMean = SMean;
    pSTD  = SStd;
    pGeneratorParameters = generatorParameters;
end

% Generate random vector
dlZ = dlarray(2 * ( rand(1,1,100,1,'single') - 0.5 ));

% Generate spectrograms
dlXGenerated = modelGenerator(dlZ,pGeneratorParameters);

% Convert from dlarray to single
S = dlXGenerated.extractdata;

S = S.';
% Zero-pad to remove edge effects
```

```
S = [S ; zeros(1,128)];

% Reverse steps from training
S = S * 3;
S = (S.*pSTD) + pMean;
S = exp(S);

% Make it two-sided
S = [S ; S(end-1:-1:2,:)];
% Pad with zeros at end and start
S = [zeros(256,100) S zeros(256,100)];

% Reconstruct the signal using a fast Griffin-Lim algorithm.
myAudio = stftmag2sig(gather(S),256, ...
    'FrequencyRange','twosided', ...
    'Window',hann(256,'periodic'), ...
    'OverlapLength',128, ...
    'MaxIterations',20, ...
    'Method','fgla');
myAudio = myAudio./max(abs(myAudio),[],'all');
y = myAudio(128*100:end-128*100);
end
```

Utility Functions

```
function out = sigmoid_cross_entropy_with_logits(x,z)
out = max(x, 0) - x .* z + log(1 + exp(-abs(x)));
end

function w = iGlorotInitialize(sz)
if numel(sz) == 2
    numInputs = sz(2);
    numOutputs = sz(1);
else
    numInputs = prod(sz(1:3));
    numOutputs = prod(sz([1 2 4]));
end
multiplier = sqrt(2 / (numInputs + numOutputs));
w = multiplier * sqrt(3) * (2 * rand(sz,'single') - 1);
end
```

Reference

[1] Donahue, C., J. McAuley, and M. Puckette. 2019. "Adversarial Audio Synthesis." ICLR.

Sequential Feature Selection for Audio Features

This example shows a typical workflow for feature selection applied to the task of spoken digit recognition.

In sequential feature selection, you train a network on a given feature set and then incrementally add or remove features until the highest accuracy is reached [1] on page 14-0 . In this example, you apply sequential forward selection to the task of spoken digit recognition using the Free Spoken Digit Dataset [2] on page 14-0 .

Streaming Spoken Digit Recognition

To motivate the example, begin by loading a pretrained network, the `audioFeatureExtractor` (Audio Toolbox) object used to train the network, and normalization factors for the features.

```
load('network_Audio_SequentialFeatureSelection.mat','bestNet','afe','normalizers');
```

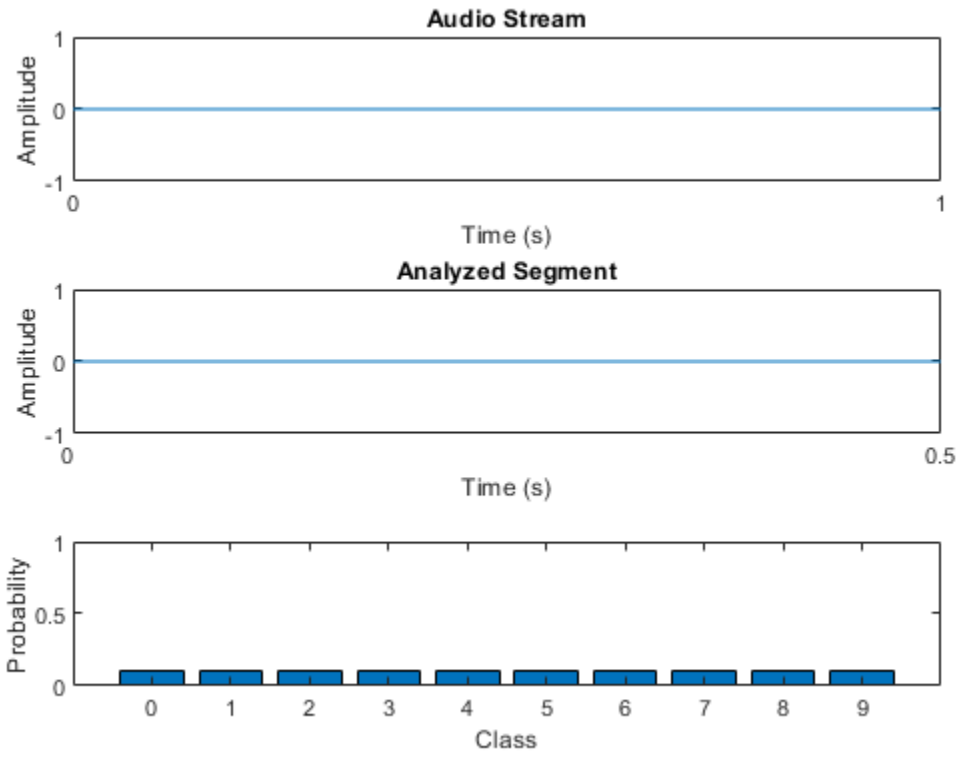
Create an `audioDeviceReader` (Audio Toolbox) to read audio from a microphone. Create three `dsp.AsyncBuffer` (DSP System Toolbox) objects: one to buffer audio read from your microphone, one to buffer short-term energy of the input audio for speech detection, and one to buffer predictions.

```
fs = afe.SampleRate;

deviceReader = audioDeviceReader('SampleRate',fs,'SamplesPerFrame',256);

audioBuffer = dsp.AsyncBuffer(fs*3);
steBuffer = dsp.AsyncBuffer(1000);
predictionBuffer = dsp.AsyncBuffer(5);
```

Create a plot to display the streaming audio, the probability the network outputs during inference, and the prediction.

```
fig = figure;

streamAxes = subplot(3,1,1);
streamPlot = plot(zeros(fs,1));
ylabel('Amplitude')
xlabel('Time (s)')
title('Audio Stream')
streamAxes.XTick = [0,fs];
streamAxes.XTickLabel = [0,1];
streamAxes.YLim = [-1,1];

analyzedAxes = subplot(3,1,2);
analyzedPlot = plot(zeros(fs/2,1));
title('Analyzed Segment')
ylabel('Amplitude')
xlabel('Time (s)')
set(gca,'XTickLabel',[])
analyzedAxes.XTick = [0,fs/2];
analyzedAxes.XTickLabel = [0,0.5];
analyzedAxes.YLim = [-1,1];

probabilityAxes = subplot(3,1,3);
probabilityPlot = bar(0:9,0.1*ones(1,10));
axis([-1,10,0,1])
```

```
ylabel('Probability')
xlabel('Class')
```



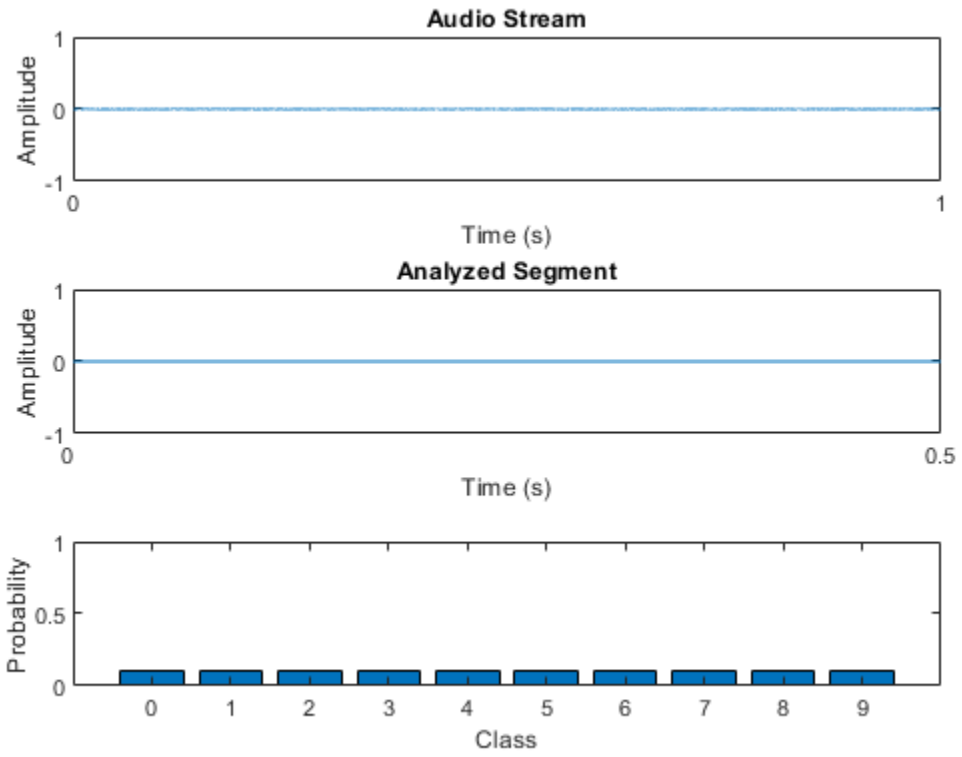
Perform streaming digit recognition (digits 0 through 9) for 20 seconds. While the loop runs, speak one of the digits and test its accuracy.

First, define a short-term energy threshold under which to assume a signal contains no speech.

```
steThreshold = 0.015;
idxVec = 1:fs;
tic
while toc < 20

    % Read in a frame of audio from your device.
    audioIn = deviceReader();

    % Write the audio into a the buffer.
    write(audioBuffer,audioIn);

    % While 200 ms of data is unused, continue this loop.
    while audioBuffer.NumUnreadSamples > 0.2*fs

        % Read 1 second from the audio buffer. Of that 1 second, 800 ms
        % is rereading old data and 200 ms is new data.
        audioToAnalyze = read(audioBuffer,fs,0.8*fs);

        % Update the figure to plot the current audio data.
        streamPlot.YData = audioToAnalyze;
```

```
ste = mean(abs(audioToAnalyze));
write(steBuffer,ste);
if steBuffer.NumUnreadSamples > 5
    abc = sort(peek(steBuffer));
    steThreshold = abc(round(0.4*numel(abc)));
end
if ste > steThreshold

    % Use the detectSpeeech function to determine if a region of speech
    % is present.
    idx = detectSpeech(audioToAnalyze,fs);

    % If a region of speech is present, perform the following.
    if ~isempty(idx)
        % Zero out all parts of the signal except the speech
        % region, and trim to 0.5 seconds.
        audioToAnalyze = HelperTrimOrPad(audioToAnalyze(idx(1,1):idx(1,2)),fs/2);

        % Normalize the audio.
        audioToAnalyze = audioToAnalyze/max(abs(audioToAnalyze));

        % Update the analyzed segment plot
        analyzedPlot.YData = audioToAnalyze;

        % Extract the features and transpose them so that time is
        % across columns.
        features = (extract(afe,audioToAnalyze))';

        % Normalize the features.
        features = (features - normalizers.Mean) ./ normalizers.StandardDeviation;

        % Call classify to determine the probabilities and the
        % winning label.
        features(isnan(features)) = 0;
        [label,probs] = classify(bestNet,features);

        % Update the plot with the probabilities and the winning
        % label.
        probabilityPlot.YData = probs;
        write(predictionBuffer,probs);

        if predictionBuffer.NumUnreadSamples == predictionBuffer.Capacity
            lastTen = peek(predictionBuffer);
            [~,decision] = max(mean(lastTen.*hann(size(lastTen,1)),1));
            probabilityAxes.Title.String = num2str(decision-1);
        end
    end
else
    % If the signal energy is below the threshold, assume no speech
    % detected.
     probabilityAxes.Title.String = '';
     probabilityPlot.YData = 0.1*ones(10,1);
     analyzedPlot.YData = zeros(fs/2,1);
     reset(predictionBuffer)
end

drawnow limitrate
```

```
    end
end
```



The remainder of the example illustrates how the network used in the streaming detection was trained and how the features fed into the network were chosen.

Create Train and Validation Data Sets

Download the Free Spoken Digit Dataset (FSDD) [2] on page 14-0 . FSDD consists of short audio files with spoken digits (0-9).

```
url = "https://zenodo.org/record/1342401/files/Jakobovski/free-spoken-digit-dataset-v1.0.8.zip";
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'FSDD');

if ~exist(datasetFolder,'dir')
    fprintf('Downloading Free Spoken Digit Dataset ...\n')
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` (Audio Toolbox) to point to the recordings. Get the sample rate of the data set.

```
ads = audioDatastore(datasetFolder,'IncludeSubfolders',true);
[~,adsInfo] = read(ads);
fs = adsInfo.SampleRate;
```

The first element of the file names is the digit spoken in the file. Get the first element of the file names, convert them to categorical, and then set the `Labels` property of the `audioDatastore`.

```
[~,filenames] = cellfun(@(x)fileparts(x),ads.Files,'UniformOutput',false);
ads.Labels = categorical(string(cellfun(@(x)x(1),filenames)));
```

To split the datastore into a development set and a validation set, use `splitEachLabel` (Audio Toolbox). Allocate 80% of the data for development and the remaining 20% for validation.

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

Set Up Audio Feature Extractor

Create an `audioFeatureExtractor` (Audio Toolbox) object to extract audio features over 30 ms windows with an update rate of 10 ms. Set all features you would like to test in this example to `true`.

```
win = hamming(round(0.03*fs),"periodic");
overlapLength = round(0.02*fs);

afe = audioFeatureExtractor( ...
    'Window',       win, ...
    'OverlapLength',overlapLength, ...
    'SampleRate',   fs, ...
    ...
    'linearSpectrum',      false, ...
    'melSpectrum',         false, ...
    'barkSpectrum',        false, ...
    'erbSpectrum',         false, ...
    ...
    'mfcc',                true, ...
    'mfccDelta',           true, ...
    'mfccDeltaDelta',      true, ...
    'gtcc',                true, ...
    'gtccDelta',           true, ...
    'gtccDeltaDelta',      true, ...
    ...
    'spectralCentroid',    true, ...
    'spectralCrest',       true, ...
    'spectralDecrease',    true, ...
    'spectralEntropy',     true, ...
    'spectralFlatness',    true, ...
    'spectralFlux',        true, ...
    'spectralKurtosis',    true, ...
    'spectralRolloffPoint',true, ...
    'spectralSkewness',    true, ...
    'spectralSlope',       true, ...
    'spectralSpread',      true, ...
    ...
    'pitch',               false, ...
    'harmonicRatio',       false);
```

Define Layers and Training Options

Define the “List of Deep Learning Layers” on page 1-21 and `trainingOptions` used in this example. The first layer, `sequenceInputLayer`, is just a placeholder. Depending on which features you test during sequential feature selection, the first layer is replaced with a `sequenceInputLayer` of the appropriate size.

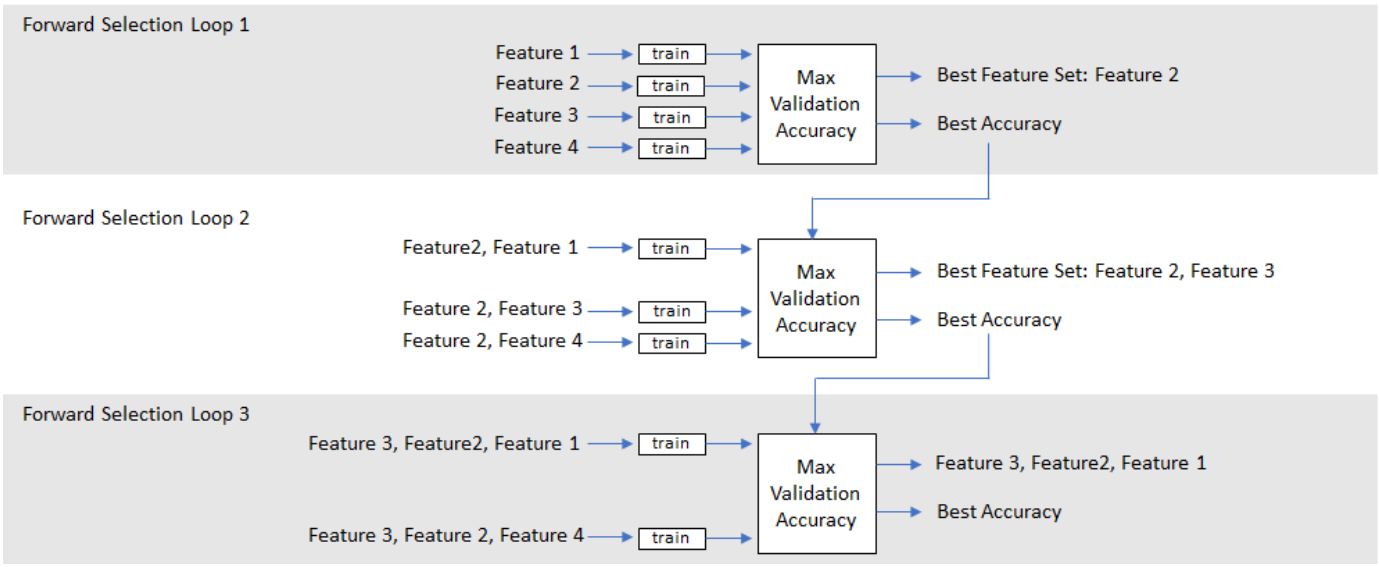
```
numUnits = 100 ;
layers = [ ...
    sequenceInputLayer(1)
    bilstmLayer(numUnits,"OutputMode","last")
    fullyConnectedLayer(numel(categories(adsTrain.Labels)))
    softmaxLayer
    classificationLayer];

options = trainingOptions("adam", ...
    "LearnRateSchedule","piecewise", ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "MaxEpochs",20);
```

Sequential Feature Selection

In the basic form of sequential feature selection, you train a network on a given feature set and then incrementally add or remove features until the accuracy no longer improves [1] on page 14-0 .

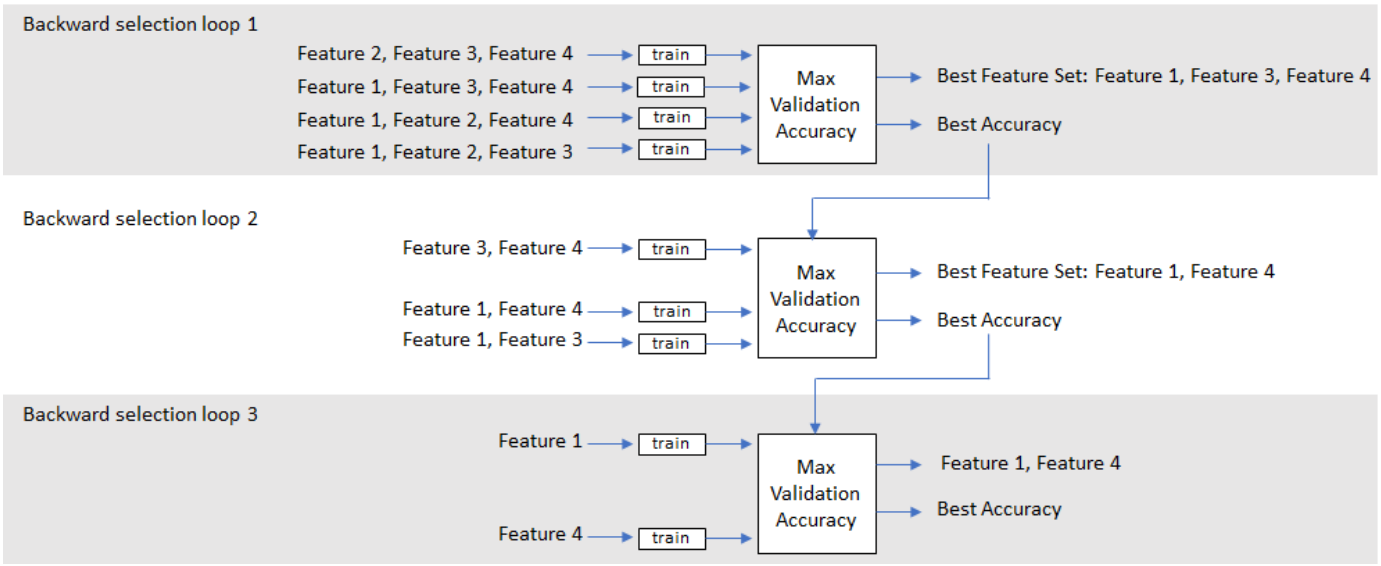
Forward Selection

Consider a simple case of forward selection on a set of four features. In the first forward selection loop, each of the four features are tested independently by training a network and comparing their validation accuracy. The feature that resulted in the highest validation accuracy is noted. In the second forward selection loop, the best feature from the first loop is combined with each of the remaining features. Now each pair of features is used for training. If the accuracy in the second loop did not improve over the accuracy in the first loop, the selection process ends. Otherwise, a new best feature set is selected. The forward selection loop continues until the accuracy no longer improves.



Backward Selection

In backward feature selection, you begin by training on a feature set that consists of all features and test whether or not accuracy improves as you remove features.



Run Sequential Feature Selection

The helper functions (HelperSFS on page 14-0 , HelperTrainAndValidateNetwork on page 14-0 , and HelperTrimOrPad on page 14-0) implement forward or backward sequential feature selection. Specify the training datastore, validation datastore, audio feature extractor, network layers, network options, and direction. As a general rule, choose forward if you anticipate a small feature set or backward if you anticipate a large feature set.

```
direction = forward;
[logbook,bestFeatures,bestNet,normalizers] = HelperSFS(adsTrain,adsValidation,afe,layers,options,
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

The `logbook` output from `HelperFeatureExtractor` is a table containing all feature configurations tested and the corresponding validation accuracy.

```
logbook
```

```
logbook=48×2 table
                Features                    Accuracy
    ____________________________________    ________

    "mfcc, gtcc"                             97.333
    "mfcc, mfccDelta, gtcc"                      97
    "mfcc, gtcc, spectralEntropy"                97
    "mfcc, gtcc, spectralFlatness"               97
    "mfcc, gtcc, spectralFlux"                   97
    "mfcc, gtcc, spectralSpread"                 97
    "gtcc"                                   96.667
    "gtcc, spectralCentroid"                 96.667
    "gtcc, spectralFlux"                     96.667
    "mfcc, gtcc, spectralRolloffPoint"       96.667
    "mfcc, gtcc, spectralSkewness"           96.667
    "gtcc, spectralEntropy"                  96.333
    "mfcc, gtcc, gtccDeltaDelta"             96.333
```

```
    "mfcc, gtcc, spectralKurtosis"         96.333
    "mfccDelta, gtcc"                          96
    "gtcc, gtccDelta"                          96
      ⋮
```

The `bestFeatures` output from `HelperSFS` contains a struct with the optimal features set to `true`.

```
bestFeatures
```

```
bestFeatures = struct with fields:
                    mfcc: 1
                mfccDelta: 0
           mfccDeltaDelta: 0
                    gtcc: 1
                gtccDelta: 0
           gtccDeltaDelta: 0
          spectralCentroid: 0
             spectralCrest: 0
          spectralDecrease: 0
           spectralEntropy: 0
          spectralFlatness: 0
              spectralFlux: 0
          spectralKurtosis: 0
      spectralRolloffPoint: 0
          spectralSkewness: 0
             spectralSlope: 0
           spectralSpread: 0
```

You can set your `audioFeatureExtractor` using the struct.

```
set(afe,bestFeatures)
afe
```

```
afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [240×1 double]
              OverlapLength: 160
                 SampleRate: 8000
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'

   Enabled Features
     mfcc, gtcc

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfccDelta, mfccDeltaDelta
     gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease, spectralEntrop
     spectralFlatness, spectralFlux, spectralKurtosis, spectralRolloffPoint, spectralSkewness, sp
     spectralSpread, pitch, harmonicRatio


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

`HelperSFS` also outputs the best performing network and the normalization factors that correspond to the chosen features. To save the network, configured `audioFeatureExtractor`, and normalization factors, uncomment this line:

```
% save('network_Audio_SequentialFeatureSelection.mat','bestNet','afe','normalizers')
```

Conclusion

This example illustrates a workflow for sequential feature selection for a Recurrent Neural Network (LSTM or BiLSTM). It could easily be adapted for CNN and RNN-CNN workflows.

Supporting Functions

HelperTrainAndValidateNetwork

```
function [trueLabels,predictedLabels,net,normalizers] = HelperTrainAndValidateNetwork(adsTrain,ad
% Train and validate a network.
%
%   INPUTS:
%   adsTrain      - audioDatastore object that points to training set
%   adsValidation - audioDatastore object that points to validation set
%   afe           - audioFeatureExtractor object.
%   layers        - Layers of LSTM or BiLSTM network
%   options       - trainingOptions object
%
%   OUTPUTS:
%   trueLabels      - true labels of validation set
%   predictedLabels - predicted labels of validation set
%   net             - trained network
%   normalizers     - normalization factors for features under test

% Copyright 2019 The MathWorks, Inc.

% Convert the data to tall arrays.
tallTrain      = tall(adsTrain);
tallValidation = tall(adsValidation);

% Extract features from the training set. Reorient the features so that
% time is along rows to be compatible with sequenceInputLayer.
fs = afe.SampleRate;
tallTrain         = cellfun(@(x)HelperTrimOrPad(x,fs/2),tallTrain,"UniformOutput",false);
tallTrain         = cellfun(@(x)x/max(abs(x),[],'all'),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)extract(afe,x),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)x',tallFeaturesTrain,"UniformOutput",false);  %#ok<NASGU>
[~,featuresTrain] = evalc('gather(tallFeaturesTrain)'); % Use evalc to suppress command-line outp

tallValidation         = cellfun(@(x)HelperTrimOrPad(x,fs/2),tallValidation,"UniformOutput",false
tallValidation         = cellfun(@(x)x/max(abs(x),[],'all'),tallValidation,"UniformOutput",false)
tallFeaturesValidation = cellfun(@(x)extract(afe,x),tallValidation,"UniformOutput",false);
tallFeaturesValidation = cellfun(@(x)x',tallFeaturesValidation,"UniformOutput",false); %#ok<NASGU
[~,featuresValidation] = evalc('gather(tallFeaturesValidation)'); % Use evalc to suppress command

% Use the training set to determine the mean and standard deviation of each
% feature. Normalize the training and validation sets.
allFeatures = cat(2,featuresTrain{:});
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,'UniformOutput',false);
```

```
for ii = 1:numel(featuresTrain)
    idx = find(isnan(featuresTrain{ii}));
    if ~isempty(idx)
        featuresTrain{ii}(idx) = 0;
    end
end
featuresValidation = cellfun(@(x)(x-M)./S,featuresValidation,'UniformOutput',false);
for ii = 1:numel(featuresValidation)
    idx = find(isnan(featuresValidation{ii}));
    if ~isempty(idx)
        featuresValidation{ii}(idx) = 0;
    end
end

% Replicate the labels of the train and validation sets so that they are in
% one-to-one correspondence with the sequences.
labelsTrain = adsTrain.Labels;

% Update input layer for the number of features under test.
layers(1) = sequenceInputLayer(size(featuresTrain{1},1));

% Train the network.
net = trainNetwork(featuresTrain,labelsTrain,layers,options);

% Evaluate the network. Call classify to get the predicted labels for each
% sequence.
predictedLabels = classify(net,featuresValidation);
trueLabels = adsValidation.Labels;

% Save the normalization factors as a struct.
normalizers.Mean = M;
normalizers.StandardDeviation = S;
end
```

HelperSFS

```
function [logbook,bestFeatures,bestNet,bestNormalizers] = HelperSFS(adsTrain,adsValidate,afeThis
%
%   INPUTS:
%   adsTrain    - audioDatastore object that points to training set
%   adsValidate - audioDatastore object that points to validation set
%   afe         - audioFeatureExtractor object. Set all features to test to true
%   layers      - Layers of LSTM or BiLSTM network
%   options     - trainingOptions object
%   direction   - SFS direction, specify as 'forward' or 'backward'
%
%   OUTPUTS:
%   logbook         - table containing feature configurations tested and corresponding validatio
%   bestFeatures    - struct containg best feature configuration
%   bestNet         - Trained network with highest validation accuracy
%   bestNormalizers - Feature normalization factors for best features

% Copyright 2019 The MathWorks, Inc.

afe = copy(afeThis);
featuresToTest = fieldnames(info(afe));
N = numel(featuresToTest);
bestValidationAccuracy = 0;
```

```
% Set the initial feature configuration: all on for backward selection
% or all off for forward selection.
featureConfig = info(afe);
for i = 1:N
    if strcmpi(direction,"backward")
        featureConfig.(featuresToTest{i}) = true;
    else
        featureConfig.(featuresToTest{i}) = false;
    end
end

% Initialize logbook to track feature configuration and accuracy.
logbook = table(featureConfig,0,'VariableNames',["Feature Configuration","Accuracy"]);

% Perform sequential feature evaluation.
wrapperIdx = 1;
bestAccuracy = 0;
while wrapperIdx <= N
    % Create a cell array containing all feature configurations to test
    % in the current loop.
    featureConfigsToTest = cell(numel(featuresToTest),1);
    for ii = 1:numel(featuresToTest)
        if strcmpi(direction,"backward")
            featureConfig.(featuresToTest{ii}) = false;
        else
            featureConfig.(featuresToTest{ii}) = true;
        end
        featureConfigsToTest{ii} = featureConfig;
        if strcmpi(direction,"backward")
            featureConfig.(featuresToTest{ii}) = true;
        else
            featureConfig.(featuresToTest{ii}) = false;
        end
    end

    % Loop over every feature set.
    for ii = 1:numel(featureConfigsToTest)

        % Determine the current feature configuration to test. Update
        % the feature afe.
        currentConfig = featureConfigsToTest{ii};
        set(afe,currentConfig)

        % Train and get k-fold cross-validation accuracy for current
        % feature configuration.
        [trueLabels,predictedLabels,net,normalizers] = HelperTrainAndValidateNetwork(adsTrain,ads
        valAccuracy = mean(trueLabels==predictedLabels)*100;
        if valAccuracy > bestValidationAccuracy
            bestValidationAccuracy = valAccuracy;
            bestNet = net;
            bestNormalizers = normalizers;
        end

        % Update Logbook
        result = table(currentConfig,valAccuracy,'VariableNames',["Feature Configuration","Accura
        logbook = [logbook;result]; %#ok<AGROW>
```

```
    end

    % Determine and print the setting with the best accuracy. If accuracy
    % did not improve, end the run.
    [a,b] = max(logbook{:,'Accuracy'});
    if a <= bestAccuracy
        wrapperIdx = inf;
    else
        wrapperIdx = wrapperIdx + 1;
    end
    bestAccuracy = a;

    % Update the features-to-test based on the most recent winner.
    winner = logbook{b,'Feature Configuration'};
    fn = fieldnames(winner);
    tf = structfun(@(x)(x),winner);
    if strcmpi(direction,"backward")
        featuresToRemove = fn(~tf);
    else
        featuresToRemove = fn(tf);
    end
    for ii = 1:numel(featuresToRemove)
        loc =  strcmp(featuresToTest,featuresToRemove{ii});
        featuresToTest(loc) = [];
        if strcmpi(direction,"backward")
            featureConfig.(featuresToRemove{ii}) = false;
        else
            featureConfig.(featuresToRemove{ii}) = true;
        end
    end

end

% Sort the logbook and make it more readable.
logbook(1,:) = []; % Delete placeholder first row.
logbook = sortrows(logbook,{'Accuracy'},{'descend'});
bestFeatures = logbook{1,'Feature Configuration'};
m = logbook{:,'Feature Configuration'};
fn = fieldnames(m);
myString = strings(numel(m),1);
for wrapperIdx = 1:numel(m)
    tf = structfun(@(x)(x),logbook{wrapperIdx,'Feature Configuration'});
    myString(wrapperIdx) = strjoin(fn(tf),", ");
end
logbook = table(myString,logbook{:,'Accuracy'},'VariableNames',["Features","Accuracy"]);
end
```

HelperTrimOrPad

```
function y = HelperTrimOrPad(x,n)
% y = HelperTrimOrPad(x,n) trims or pads the input x to n samples. If x is
% trimmed, it is trimmed equally on the front and back. If x is padded, it is
% padded equally on the front and back with zeros. For odd-length trimming or
% padding, the extra sample is trimmed or padded from the back.

% Copyright 2019 The MathWorks, Inc.
a = size(x,1);
if a < n
```

```
    frontPad = floor((n-a)/2);
    backPad = n - a - frontPad;
    y = [zeros(frontPad,1);x;zeros(backPad,1)];
elseif a > n
    frontTrim = floor((a-n)/2)+1;
    backTrim = a - n - frontTrim;
    y = x(frontTrim:end-backTrim);
else
    y = x;
end
end
```

References

[1] Jain, A., and D. Zongker. "Feature Selection: Evaluation, Application, and Small Sample Performance." IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol. 19, Issue 2, 1997, pp. 153-158.

[2] Jakobovski. “Jakobovski/Free-Spoken-Digit-Dataset.” GitHub, May 30, 2019. https://github.com/Jakobovski/free-spoken-digit-dataset.

Acoustic Scene Recognition Using Late Fusion

This example shows how to create a multi-model late fusion system for acoustic scene recognition. The example trains a convolutional neural network (CNN) using mel spectrograms and an ensemble classifier using wavelet scattering. The example uses the TUT dataset for training and evaluation [1].

Introduction

Acoustic scene classification (ASC) is the task of classifying environments from the sounds they produce. ASC is a generic classification problem that is foundational for context awareness in devices, robots, and many other applications [1]. Early attempts at ASC used mel-frequency cepstral coefficients (`mfcc` (Audio Toolbox)) and Gaussian mixture models (GMMs) to describe their statistical distribution. Other popular features used for ASC include zero crossing rate, spectral centroid (`spectralCentroid` (Audio Toolbox)), spectral rolloff (`spectralRolloffPoint` (Audio Toolbox)), spectral flux (`spectralFlux` (Audio Toolbox)), and linear prediction coefficients (`lpc` (Signal Processing Toolbox)) [5]. Hidden Markov models (HMMs) were trained to describe the temporal evolution of the GMMs. More recently, the best performing systems have used deep learning, usually CNNs, and a fusion of multiple models. The most popular feature for top-ranked systems in the DCASE 2017 contest was the mel spectrogram (`melSpectrogram` (Audio Toolbox)). The top-ranked systems in the challenge used late fusion and data augmentation to help their systems generalize.

To illustrate a simple approach that produces reasonable results, this example trains a CNN using mel spectrograms and an ensemble classifier using wavelet scattering. The CNN and ensemble classifier produce roughly equivalent overall accuracy, but perform better at distinguishing different acoustic scenes. To increase overall accuracy, you merge the CNN and ensemble classifier results using late fusion.

Load Acoustic Scene Recognition Data Set

To run the example, you must first download the data set [1]. The full data set is approximately 15.5 GB. Depending on your machine and internet connection, downloading the data can take about 4 hours.

```
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'TUT-acoustic-scenes-2017');

if ~exist(datasetFolder,'dir')
    disp('Downloading TUT-acoustic-scenes-2017 (15.5 GB)...')
    HelperDownload_TUT_acoustic_scenes_2017(datasetFolder);
end
```

Read in the development set metadata as a table. Name the table variables `FileName`, `AcousticScene`, and `SpecificLocation`.

```
metadata_train = readtable(fullfile(datasetFolder,'TUT-acoustic-scenes-2017-development','meta.tx
    'Delimiter',{'\t'}, ...
    'ReadVariableNames',false);
metadata_train.Properties.VariableNames = {'FileName','AcousticScene','SpecificLocation'};
head(metadata_train)
```

```
ans =

  8×3 table
```

```
            FileName              AcousticScene    SpecificLocation
    ______________________________    _____________    ________________

    {'audio/b020_90_100.wav' }      {'beach'}          {'b020'}
    {'audio/b020_110_120.wav'}      {'beach'}          {'b020'}
    {'audio/b020_100_110.wav'}      {'beach'}          {'b020'}
    {'audio/b020_40_50.wav'  }      {'beach'}          {'b020'}
    {'audio/b020_50_60.wav'  }      {'beach'}          {'b020'}
    {'audio/b020_30_40.wav'  }      {'beach'}          {'b020'}
    {'audio/b020_160_170.wav'}      {'beach'}          {'b020'}
    {'audio/b020_170_180.wav'}      {'beach'}          {'b020'}
```

```
metadata_test = readtable(fullfile(datasetFolder,'TUT-acoustic-scenes-2017-evaluation','meta.txt
    'Delimiter',{'\t'}, ...
    'ReadVariableNames',false);
metadata_test.Properties.VariableNames = {'FileName','AcousticScene','SpecificLocation'};
head(metadata_test)
```

```
ans =

  8×3 table

        FileName          AcousticScene    SpecificLocation
    ____________________    _____________    ________________

    {'audio/1245.wav'}      {'beach'}          {'b174'}
    {'audio/1456.wav'}      {'beach'}          {'b174'}
    {'audio/1318.wav'}      {'beach'}          {'b174'}
    {'audio/967.wav' }      {'beach'}          {'b174'}
    {'audio/203.wav' }      {'beach'}          {'b174'}
    {'audio/777.wav' }      {'beach'}          {'b174'}
    {'audio/231.wav' }      {'beach'}          {'b174'}
    {'audio/768.wav' }      {'beach'}          {'b174'}
```

Note that the specific recording locations in the test set do not intersect with the specific recording locations in the development set. This makes it easier to validate that the trained models can generalize to real-world scenarios.

```
sharedRecordingLocations = intersect(metadata_test.SpecificLocation,metadata_train.SpecificLocati
fprintf('Number of specific recording locations in both train and test sets = %d\n',numel(sharedR
```

```
Number of specific recording locations in both train and test sets = 0
```

The first variable of the metadata tables contains the file names. Concatenate the file names with the file paths.

```
train_filePaths = fullfile(datasetFolder,'TUT-acoustic-scenes-2017-development',metadata_train.Fi

test_filePaths = fullfile(datasetFolder,'TUT-acoustic-scenes-2017-evaluation',metadata_test.FileN
```

Create audio datastores for the train and test sets. Set the `Labels` property of the `audioDatastore` (Audio Toolbox) to the acoustic scene. Call `countEachLabel` (Audio Toolbox) to verify an even distribution of labels in both the train and test sets.

```
adsTrain = audioDatastore(train_filePaths, ...
    'Labels',categorical(metadata_train.AcousticScene), ...
```

```
    'IncludeSubfolders',true);
display(countEachLabel(adsTrain))

adsTest = audioDatastore(test_filePaths, ...
    'Labels',categorical(metadata_test.AcousticScene), ...
    'IncludeSubfolders',true);
display(countEachLabel(adsTest))
```

```
  15×2 table

        Label          Count
    _________________    _____

    beach                312
    bus                  312
    cafe/restaurant      312
    car                  312
    city_center          312
    forest_path          312
    grocery_store        312
    home                 312
    library              312
    metro_station        312
    office               312
    park                 312
    residential_area     312
    train                312
    tram                 312

  15×2 table

        Label          Count
    _________________    _____

    beach                108
    bus                  108
    cafe/restaurant      108
    car                  108
    city_center          108
    forest_path          108
    grocery_store        108
    home                 108
    library              108
    metro_station        108
    office               108
    park                 108
    residential_area     108
    train                108
    tram                 108
```

You can reduce the data set used in this example to speed up the run time at the cost of performance. In general, reducing the data set is a good practice for development and debugging. Set `reduceDataset` to `true` to reduce the data set.

```
reduceDataset = false;
if reduceDataset
    adsTrain = splitEachLabel(adsTrain,20);
```

```
    adsTest = splitEachLabel(adsTest,10);
end
```

Call `read` (Audio Toolbox) to get the data and sample rate of a file from the train set. Audio in the database has consistent sample rate and duration. Normalize the audio and listen to it. Display the corresponding label.

```
[data,adsInfo] = read(adsTrain);
data = data./max(data,[],'all');

fs = adsInfo.SampleRate;
sound(data,fs)

fprintf('Acoustic scene = %s\n',adsTrain.Labels(1))
```

```
Acoustic scene = beach
```

Call `reset` (Audio Toolbox) to return the datastore to its initial condition.

```
reset(adsTrain)
```

Feature Extraction for CNN

Each audio clip in the dataset consists of 10 seconds of stereo (left-right) audio. The feature extraction pipeline and the CNN architecture in this example are based on [3]. Hyperparameters for the feature extraction, the CNN architecture, and the training options were modified from the original paper using a systematic hyperparameter optimization workflow.

First, convert the audio to mid-side encoding. [3] suggests that mid-side encoded data provides better spatial information that the CNN can use to identify moving sources (such as a train moving across an acoustic scene).

```
dataMidSide = [sum(data,2),data(:,1)-data(:,2)];
```

Divide the signal into one-second segments with overlap. The final system uses a probability-weighted average on the one-second segments to predict the scene for each 10-second audio clip in the test set. Dividing the audio clips into one-second segments makes the network easier to train and helps prevent overfitting to specific acoustic events in the training set. The overlap helps to ensure all combinations of features relative to one another are captured by the training data. It also provides the system with additional data that can be mixed uniquely during augmentation.

```
segmentLength = 1;
segmentOverlap = 0.5;

[dataBufferedMid,~] = buffer(dataMidSide(:,1),round(segmentLength*fs),round(segmentOverlap*fs),'
[dataBufferedSide,~] = buffer(dataMidSide(:,2),round(segmentLength*fs),round(segmentOverlap*fs),
dataBuffered = zeros(size(dataBufferedMid,1),size(dataBufferedMid,2)+size(dataBufferedSide,2));
dataBuffered(:,1:2:end) = dataBufferedMid;
dataBuffered(:,2:2:end) = dataBufferedSide;
```

Use `melSpectrogram` (Audio Toolbox) to transform the data into a compact frequency-domain representation. Define parameters for the mel spectrogram as suggested by [3].

```
windowLength = 2048;
samplesPerHop = 1024;
samplesOverlap = windowLength - samplesPerHop;
```

```
fftLength = 2*windowLength;
numBands = 128;
```

`melSpectrogram` operates along channels independently. To optimize processing time, call `melSpectrogram` with the entire buffered signal.

```
spec = melSpectrogram(dataBuffered,fs, ...
    'Window',hamming(windowLength,'periodic'), ...
    'OverlapLength',samplesOverlap, ...
    'FFTLength',fftLength, ...
    'NumBands',numBands);
```

Convert the mel spectrogram into the logarithmic scale.

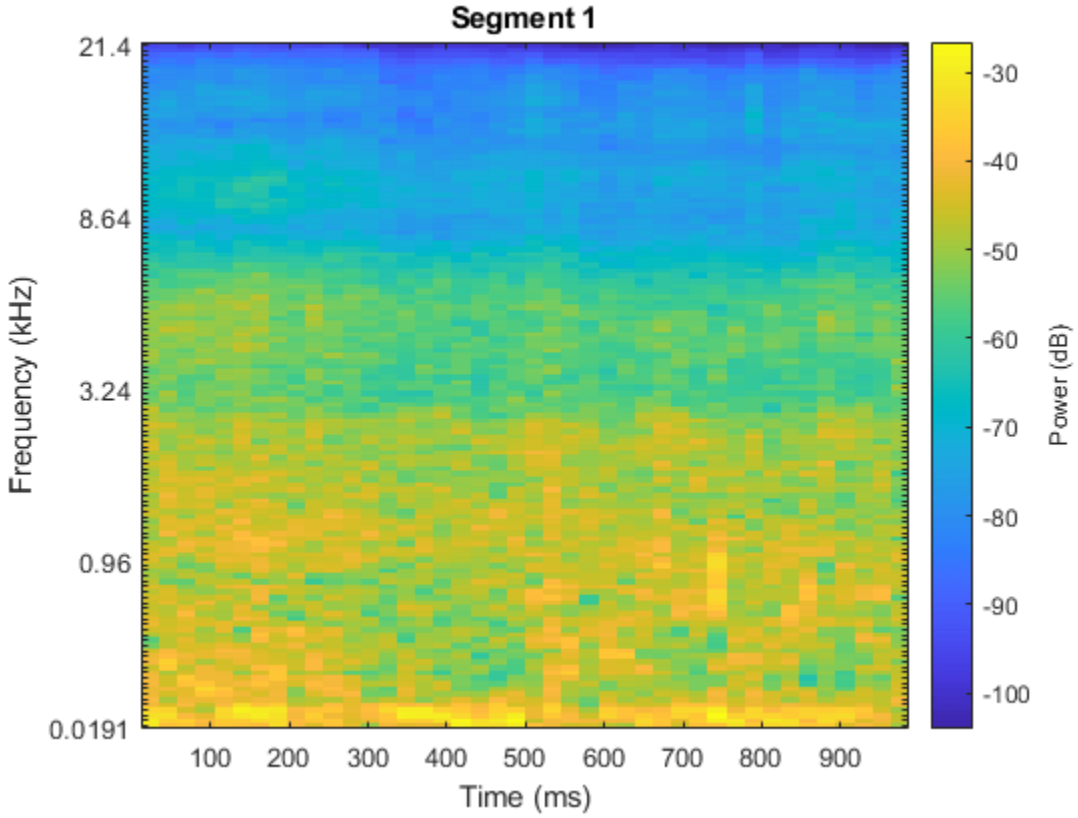
```
spec = log10(spec+eps);
```

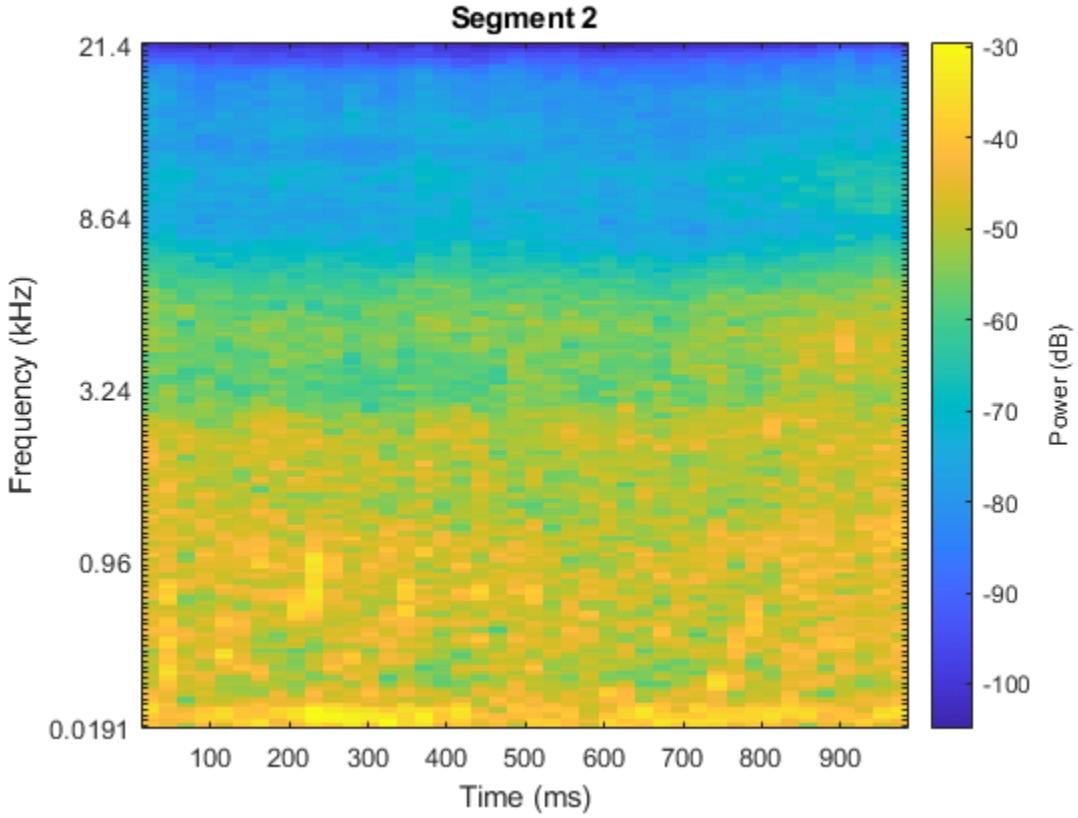
Reshape the array to dimensions (Number of bands)-by-(Number of hops)-by-(Number of channels)-by-(Number of segments). When you feed an image into a neural network, the first two dimensions are the height and width of the image, the third dimension is the channels, and the fourth dimension separates the individual images.

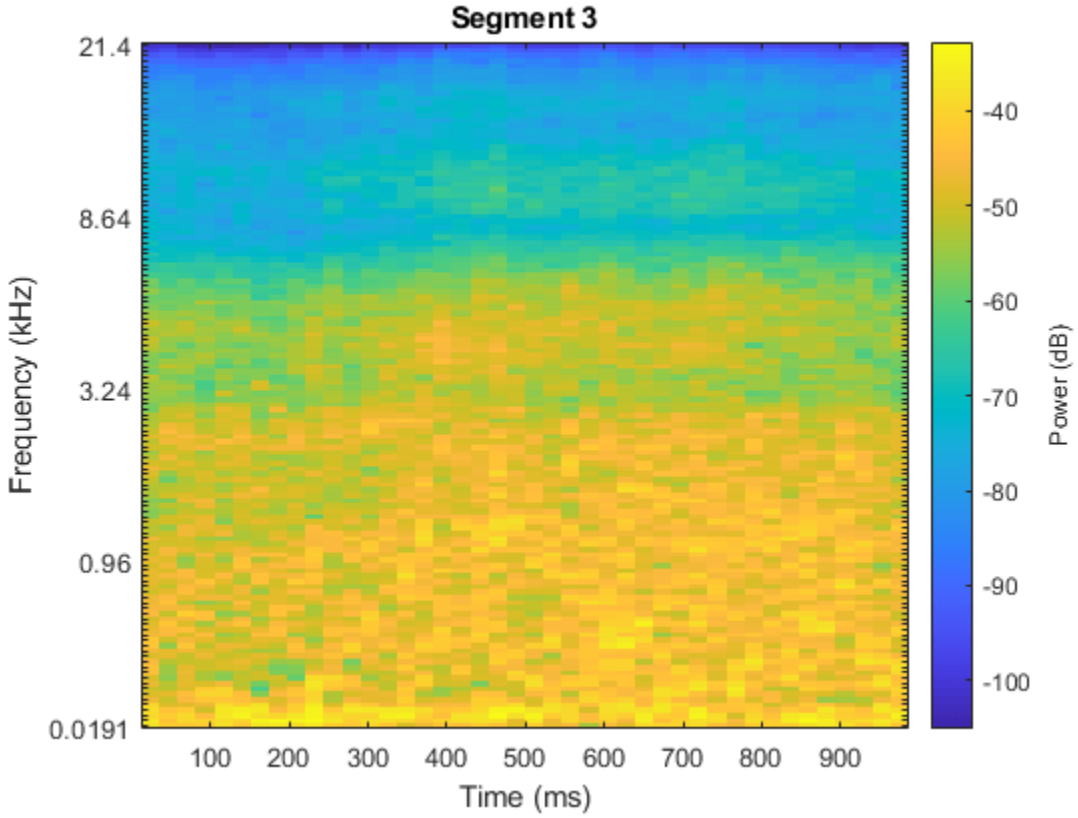
```
X = reshape(spec,size(spec,1),size(spec,2),size(data,2),[]);
```

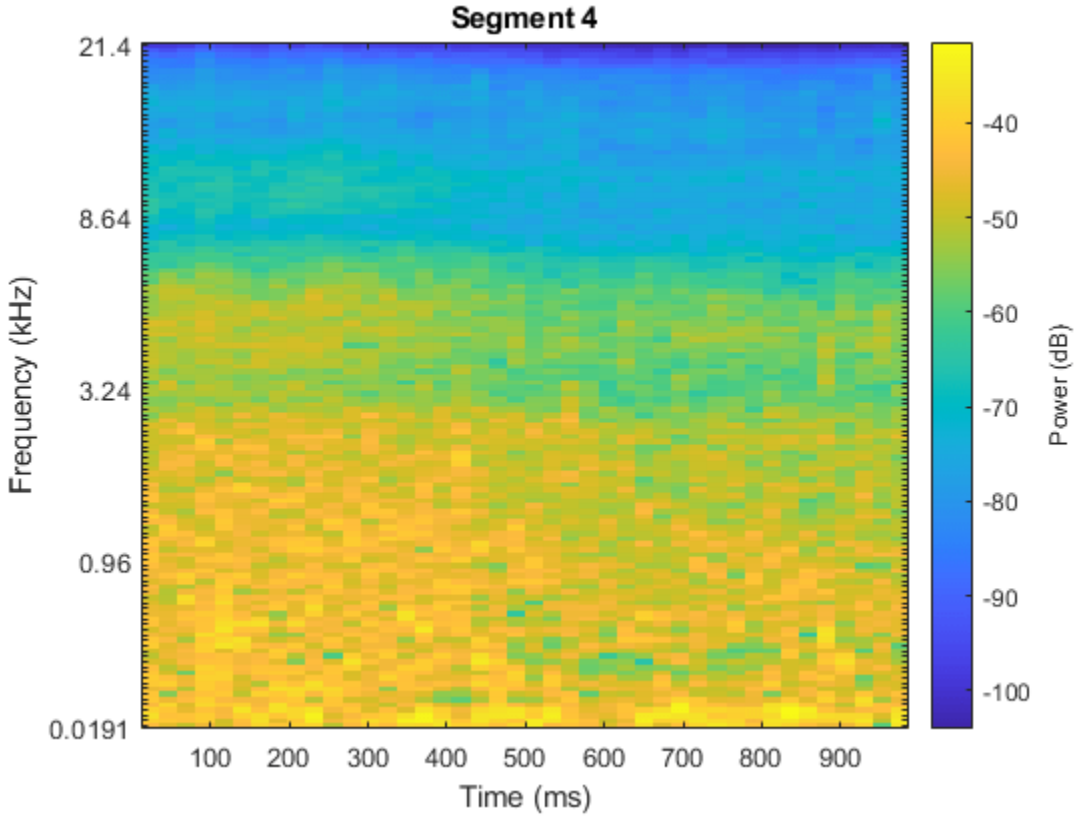
Call `melSpectrogram` without output arguments to plot the mel spectrogram of the mid channel for the first six of the one-second increments.

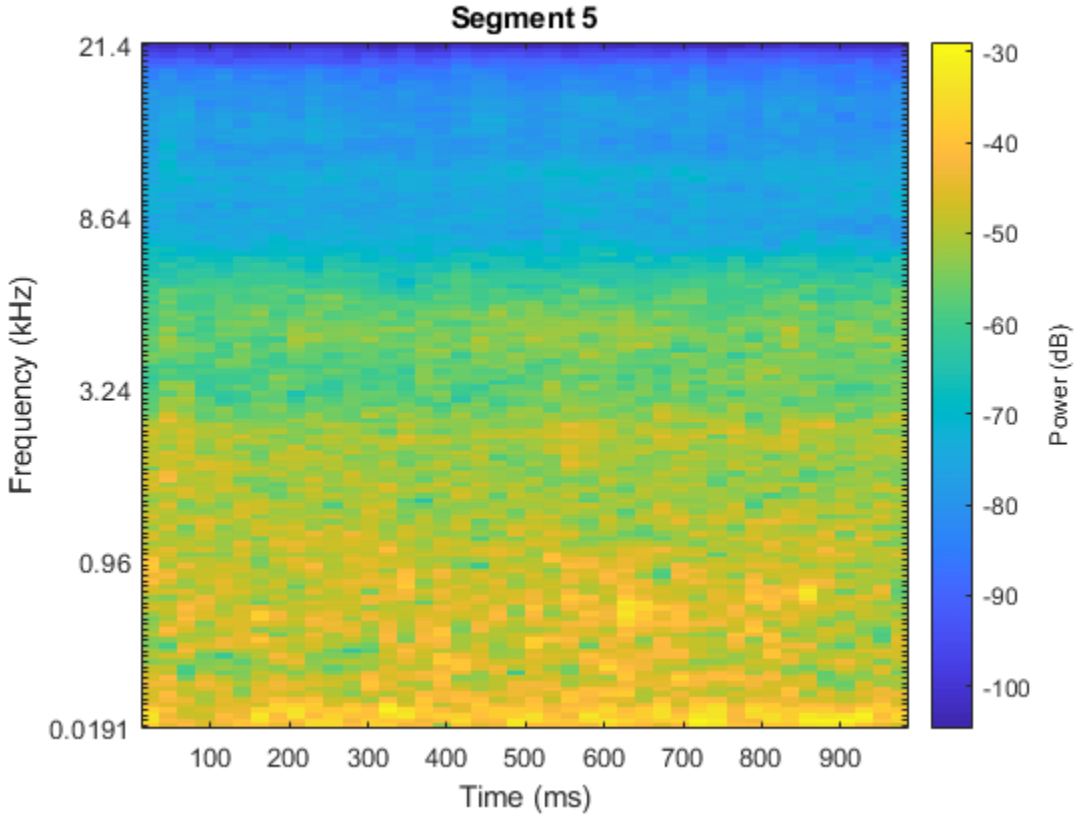
```
for channel = 1:2:11
    figure
    melSpectrogram(dataBuffered(:,channel),fs, ...
        'Window',hamming(windowLength,'periodic'), ...
        'OverlapLength',samplesOverlap, ...
        'FFTLength',fftLength, ...
        'NumBands',numBands);
    title(sprintf('Segment %d',ceil(channel/2)))
end
```

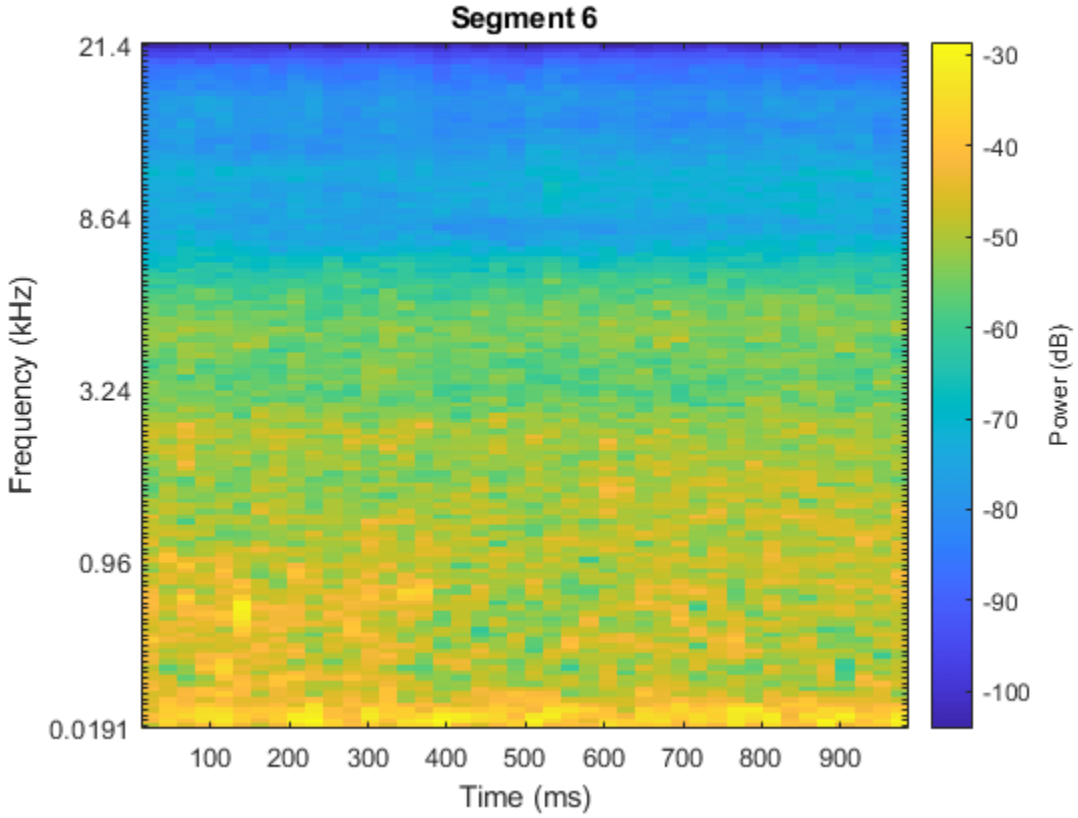

Segment 1
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-30
-40
-50
-60
-70
-80
-90
-100
Power (dB)


Segment 2
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-30
-40
-50
-60
-70
-80
-90
-100
Power (dB)


Segment 3
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-40
-50
-60
-70
-80
-90
-100
Power (dB)


Segment 4
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-40
-50
-60
-70
-80
-90
-100
Power (dB)


Segment 5
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-30
-40
-50
-60
-70
-80
-90
-100
Power (dB)



The helper function `HelperSegmentedMelSpectrograms` performs the feature extraction steps outlined above.

To speed up processing, extract mel spectrograms of all audio files in the datastores using `tall` arrays. Unlike in-memory arrays, tall arrays remain unevaluated until you request that the calculations be performed using the `gather` function. This deferred evaluation enables you to work quickly with large data sets. When you eventually request the output using `gather`, MATLAB combines the queued calculations where possible and takes the minimum number of passes through the data. If you have Parallel Computing Toolbox™, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB® Parallel Server™ installed.

If you do not have Parallel Computing Toolbox™, the code in this example still runs.

```
pp = parpool('IdleTimeout',inf);

train_set_tall = tall(adsTrain);
xTrain = cellfun(@(x)HelperSegmentedMelSpectrograms(x,fs, ...
    'SegmentLength',segmentLength, ...
    'SegmentOverlap',segmentOverlap, ...
    'WindowLength',windowLength, ...
    'HopLength',samplesPerHop, ...
    'NumBands',numBands, ...
    'FFTLength',fftLength), ...
    train_set_tall, ...
    'UniformOutput',false);
xTrain = gather(xTrain);
```

```
xTrain = cat(4,xTrain{:});

test_set_tall = tall(adsTest);
xTest = cellfun(@(x)HelperSegmentedMelSpectrograms(x,fs, ...
    'SegmentLength',segmentLength, ...
    'SegmentOverlap',segmentOverlap, ...
    'WindowLength',windowLength, ...
    'HopLength',samplesPerHop, ...
    'NumBands',numBands, ...
    'FFTLength',fftLength), ...
    test_set_tall, ...
    'UniformOutput',false);
xTest = gather(xTest);
xTest = cat(4,xTest{:});
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 3 min 45 sec
Evaluation completed in 3 min 45 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 22 sec
Evaluation completed in 1 min 22 sec
```

Replicate the labels of the training set so that they are in one-to-one correspondence with the segments.

```
numSegmentsPer10seconds = size(dataBuffered,2)/2;
yTrain = repmat(adsTrain.Labels,1,numSegmentsPer10seconds)';
yTrain = yTrain(:);
```

Data Augmentation for CNN

The DCASE 2017 dataset contains a relatively small number of acoustic recordings for the task, and the development set and evaluation set were recorded at different specific locations. As a result, it is easy to overfit to the data during training. One popular method to reduce overfitting is *mixup*. In mixup, you augment your dataset by mixing the features of two different classes. When you mix the features, you mix the labels in equal proportion. That is:

$$\tilde{\mathrm{x}} = \lambda \mathrm{x}_i + (1-\lambda)\,\mathrm{x}_j$$
$$\tilde{y} = \lambda y_i + (1-\lambda)\,y_j$$

Mixup was reformulated by [2] as labels drawn from a probability distribution instead of mixed labels. The implementation of mixup in this example is a simplified version of mixup: each spectrogram is mixed with a spectrogram of a different label with lambda set to 0.5. The original and mixed datasets are combined for training.

```
xTrainExtra = xTrain;
yTrainExtra = yTrain;
lambda = 0.5;
for i = 1:size(xTrain,4)

    % Find all available spectrograms with different labels.
    availableSpectrograms = find(yTrain~=yTrain(i));

    % Randomly choose one of the available spectrograms with a different label.
    numAvailableSpectrograms = numel(availableSpectrograms);
```

```
    idx = randi([1,numAvailableSpectrograms]);

    % Mix.
    xTrainExtra(:,:,:,i) = lambda*xTrain(:,:,:,i) + (1-lambda)*xTrain(:,:,:,availableSpectrograms

    % Specify the label as randomly set by lambda.
    if rand > lambda
        yTrainExtra(i) = yTrain(availableSpectrograms(idx));
    end
end
xTrain = cat(4,xTrain,xTrainExtra);
yTrain = [yTrain;yTrainExtra];
```

Call `summary` to display the distribution of labels for the augmented training set.

```
summary(yTrain)
```

```
     beach                11749
     bus                  11870
     cafe/restaurant      11860
     car                  11873
     city_center          11789
     forest_path          12023
     grocery_store        11850
     home                 11877
     library              11756
     metro_station        11912
     office               11940
     park                 11895
     residential_area     11875
     train                11795
     tram                 11776
```

Define and Train CNN

Define the CNN architecture. This architecture is based on [1] and modified through trial and error. See “List of Deep Learning Layers” on page 1-21 to learn more about deep learning layers available in MATLAB®.

```
imgSize = [size(xTrain,1),size(xTrain,2),size(xTrain,3)];
numF = 32;
layers = [ ...
    imageInputLayer(imgSize)

    batchNormalizationLayer

    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
```

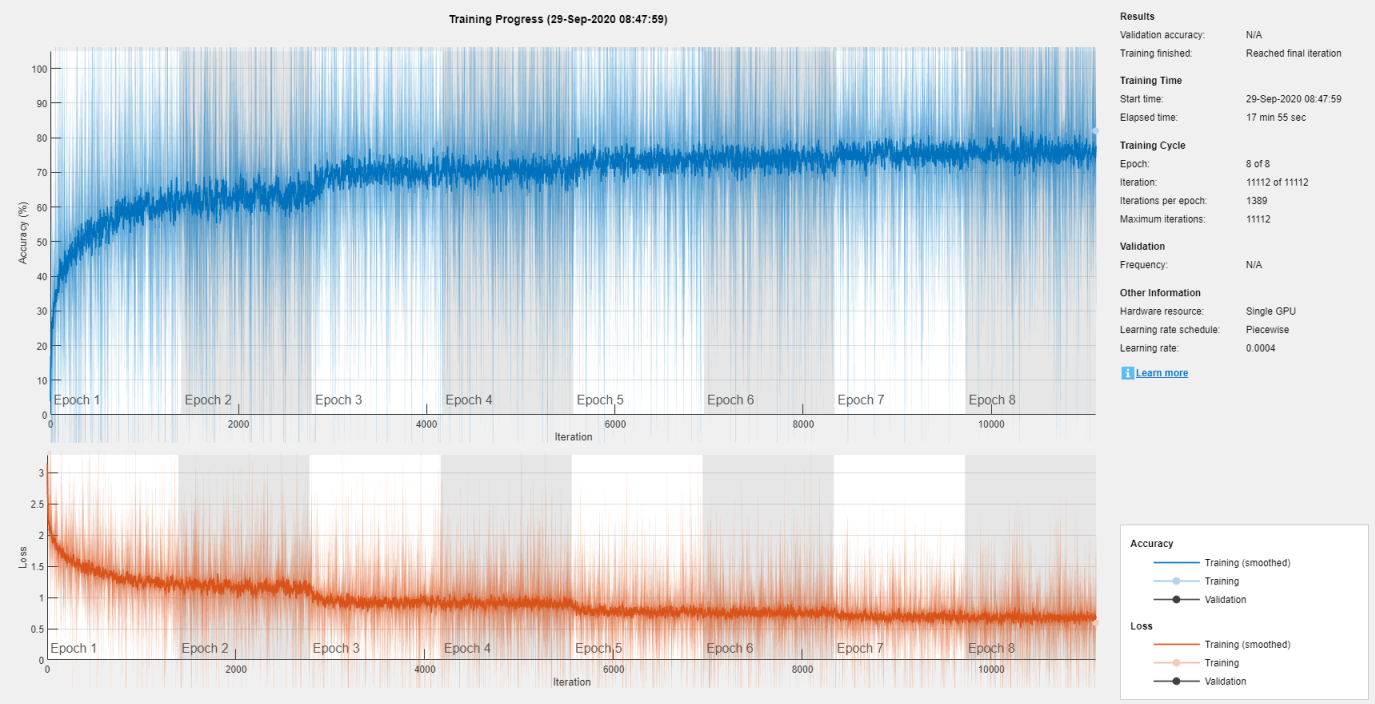
```
    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,8*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,8*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    globalAveragePooling2dLayer

    dropoutLayer(0.5)

    fullyConnectedLayer(15)
    softmaxLayer
    classificationLayer];
```

Define `trainingOptions` for the CNN. These options are based on [3] and modified through a systematic hyperparameter optimization workflow.

```
miniBatchSize = 128;
tuneme = 128;
lr = 0.05*miniBatchSize/tuneme;
options = trainingOptions('sgdm', ...
    'InitialLearnRate',lr, ...
    'MiniBatchSize',miniBatchSize, ...
    'Momentum',0.9, ...
    'L2Regularization',0.005, ...
    'MaxEpochs',8, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',2, ...
    'LearnRateDropFactor',0.2);
```

Call `trainNetwork` to train the network.

```
trainedNet = trainNetwork(xTrain,yTrain,layers,options);
```



Evaluate CNN

Call `predict` to predict responses from the trained network using the held-out test set.

```
cnnResponsesPerSegment = predict(trainedNet,xTest);
```

Average the responses over each 10-second audio clip.

```
classes = trainedNet.Layers(end).Classes;
numFiles = numel(adsTest.Files);

counter = 1;
cnnResponses = zeros(numFiles,numel(classes));
for channel = 1:numFiles
    cnnResponses(channel,:) = sum(cnnResponsesPerSegment(counter:counter+numSegmentsPer10seconds
    counter = counter + numSegmentsPer10seconds;
end
```

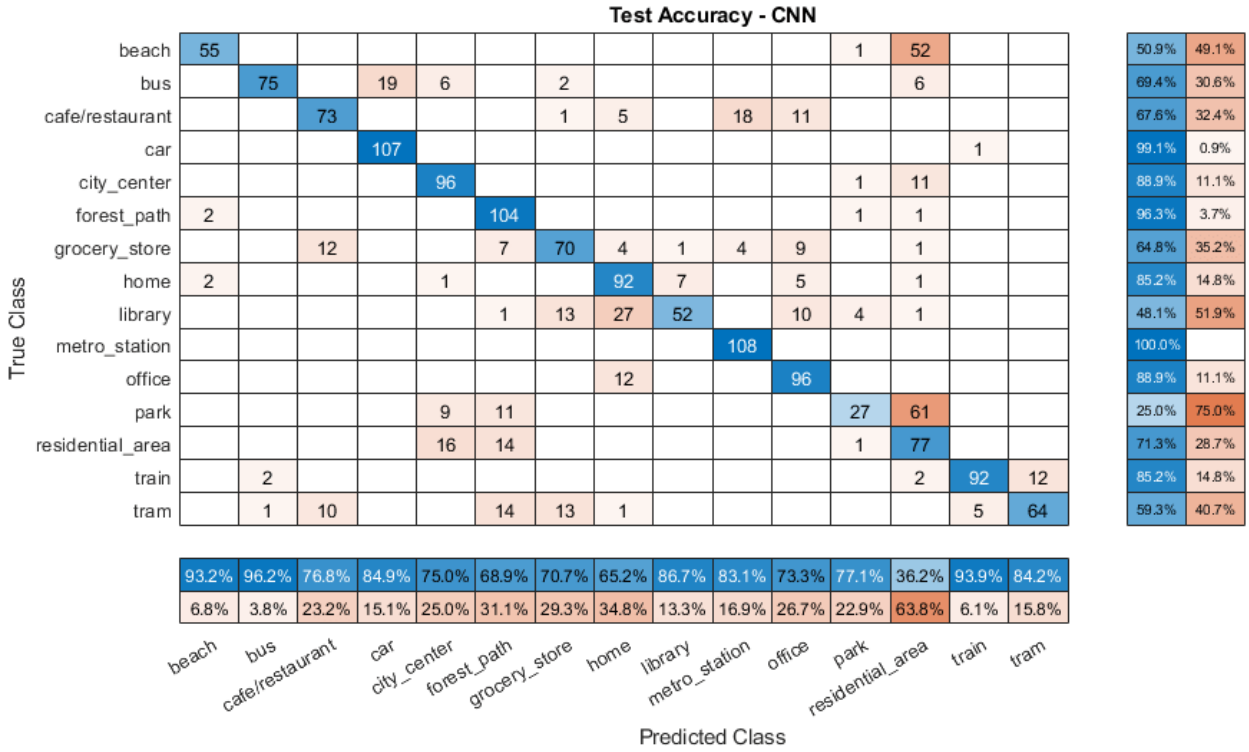
For each 10-second audio clip, choose the maximum of the predictions, then map it to the corresponding predicted location.

```
[~,classIdx] = max(cnnResponses,[],2);
cnnPredictedLabels = classes(classIdx);
```

Call `confusionchart` to visualize the accuracy on the test set. Return the average accuracy to the Command Window.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5])
cm = confusionchart(adsTest.Labels,cnnPredictedLabels,'title','Test Accuracy - CNN');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

fprintf('Average accuracy of CNN = %0.2f\n',mean(adsTest.Labels==cnnPredictedLabels)*100)
```

```
Average accuracy of CNN = 73.33
```



Feature Extraction for Ensemble Classifier

Wavelet scattering has been shown in [4] to provide a good representation of acoustic scenes. Define a `waveletScattering` (Wavelet Toolbox) object. The invariance scale and quality factors were determined through trial and error.

```
sf = waveletScattering('SignalLength',size(data,1), ...
                       'SamplingFrequency',fs, ...
                       'InvarianceScale',0.75, ...
                       'QualityFactors',[4 1]);
```

Convert the audio signal to mono, and then call `featureMatrix` (Wavelet Toolbox) to return the scattering coefficients for the scattering decomposition framework, `sf`.

```
dataMono = mean(data,2);
scatteringCoeffients = featureMatrix(sf,dataMono,'Transform','log');
```

Average the scattering coefficients over the 10-second audio clip.

```
featureVector = mean(scatteringCoeffients,2);
fprintf('Number of wavelet features per 10-second clip = %d\n',numel(featureVector))
```

```
Number of wavelet features per 10-second clip = 290
```

The helper function `HelperWaveletFeatureVector` performs the above steps. Use a `tall` array with `cellfun` and `HelperWaveletFeatureVector` to parallelize the feature extraction. Extract wavelet feature vectors for the train and test sets.

```
scatteringTrain = cellfun(@(x)HelperWaveletFeatureVector(x,sf),train_set_tall,'UniformOutput',fa
xTrain = gather(scatteringTrain);
xTrain = cell2mat(xTrain')';

scatteringTest = cellfun(@(x)HelperWaveletFeatureVector(x,sf),test_set_tall,'UniformOutput',false
xTest = gather(scatteringTest);
xTest = cell2mat(xTest')';
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 25 min 15 sec
Evaluation completed in 25 min 15 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 8 min 2 sec
Evaluation completed in 8 min 2 sec
```

Define and Train Ensemble Classifier

Use `fitcensemble` to create a trained classification ensemble model (`ClassificationEnsemble`).

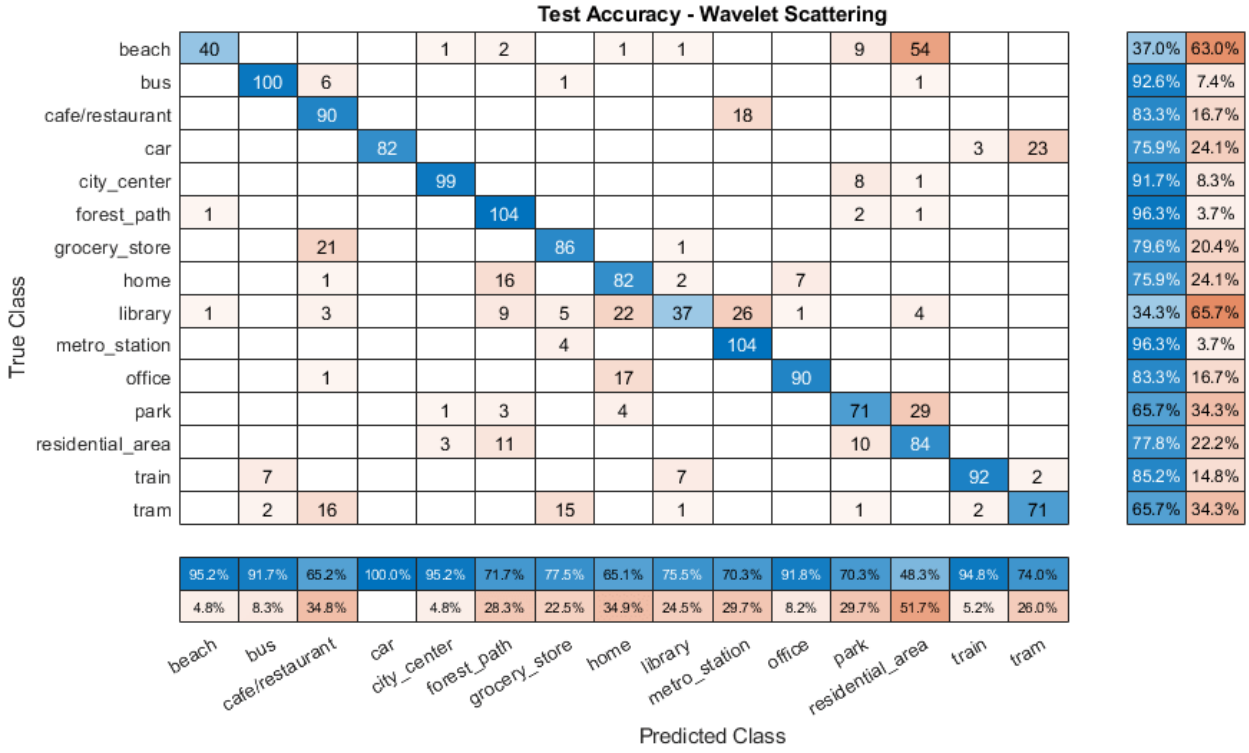
```
subspaceDimension = min(150,size(xTrain,2) - 1);
numLearningCycles = 30;
classificationEnsemble = fitcensemble(xTrain,adsTrain.Labels, ...
    'Method','Subspace', ...
    'NumLearningCycles',numLearningCycles, ...
    'Learners','discriminant', ...
    'NPredToSample',subspaceDimension, ...
    'ClassNames',removecats(unique(adsTrain.Labels)));
```

Evaluate Ensemble Classifier

For each 10-second audio clip, call `predict` to return the labels and the weights, then map it to the corresponding predicted location. Call `confusionchart` to visualize the accuracy on the test set. Print the average.

```
[waveletPredictedLabels,waveletResponses] = predict(classificationEnsemble,xTest);

figure('Units','normalized','Position',[0.2 0.2 0.5 0.5])
cm = confusionchart(adsTest.Labels,waveletPredictedLabels,'title','Test Accuracy - Wavelet Scatte
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

fprintf('Average accuracy of classifier = %0.2f\n',mean(adsTest.Labels==waveletPredictedLabels)*1
```

```
Average accuracy of classifier = 76.05
```



Apply Late Fusion

For each 10-second clip, calling predict on the wavelet classifier and the CNN returns a vector indicating the relative confidence in their decision. Multiply the `waveletResponses` with the `cnnResponses` to create a late fusion system.

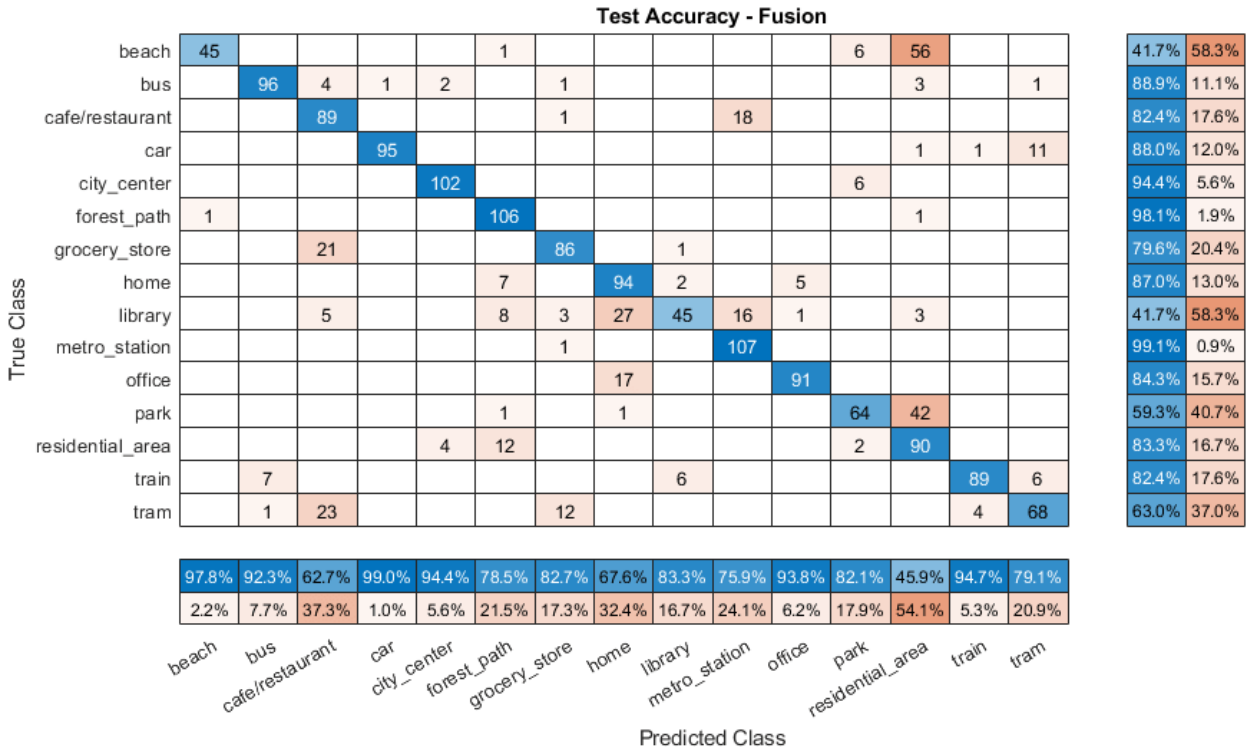
```
fused = waveletResponses .* cnnResponses;
[~,classIdx] = max(fused,[],2);

predictedLabels = classes(classIdx);
```

Evaluate Late Fusion

Call `confusionchart` to visualize the fused classification accuracy. Print the average accuracy to the Command Window.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5])
cm = confusionchart(adsTest.Labels,predictedLabels,'title','Test Accuracy - Fusion');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';

fprintf('Average accuracy of fused models = %0.2f\n',mean(adsTest.Labels==predictedLabels)*100)
```

```
Average accuracy of fused models = 78.21
```



Close the parallel pool.

```
delete(pp)
```

```
Parallel pool using the 'local' profile is shutting down.
```

References

[1] A. Mesaros, T. Heittola, and T. Virtanen. Acoustic Scene Classification: an Overview of DCASE 2017 Challenge Entries. In proc. International Workshop on Acoustic Signal Enhancement, 2018.

[2] Huszar, Ferenc. "Mixup: Data-Dependent Data Augmentation." InFERENCe. November 03, 2017. Accessed January 15, 2019. https://www.inference.vc/mixup-data-dependent-data-augmentation/.

[3] Han, Yoonchang, Jeongsoo Park, and Kyogu Lee. "Convolutional neural networks with binaural representations and background subtraction for acoustic scene classification." the Detection and Classification of Acoustic Scenes and Events (DCASE) (2017): 1-5.

[4] Lostanlen, Vincent, and Joakim Anden. Binaural scene classification with wavelet scattering. Technical Report, DCASE2016 Challenge, 2016.

[5] A. J. Eronen, V. T. Peltonen, J. T. Tuomi, A. P. Klapuri, S. Fagerlund, T. Sorsa, G. Lorho, and J. Huopaniemi, "Audio-based context recognition," IEEE Trans. on Audio, Speech, and Language Processing, vol 14, no. 1, pp. 321-329, Jan 2006.

[6] TUT Acoustic scenes 2017, Development dataset

[7] TUT Acoustic scenes 2017, Evaluation dataset

Appendix -- Supporting Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%HelperSegmentedMelSpectrograms
function X = HelperSegmentedMelSpectrograms(x,fs,varargin)

p = inputParser;
addParameter(p,'WindowLength',1024);
addParameter(p,'HopLength',512);
addParameter(p,'NumBands',128);
addParameter(p,'SegmentLength',1);
addParameter(p,'SegmentOverlap',0);
addParameter(p,'FFTLength',1024);
parse(p,varargin{:})
params = p.Results;

x = [sum(x,2),x(:,1)-x(:,2)];
x = x./max(max(x));

[xb_m,~] = buffer(x(:,1),round(params.SegmentLength*fs),round(params.SegmentOverlap*fs),'nodelay
[xb_s,~] = buffer(x(:,2),round(params.SegmentLength*fs),round(params.SegmentOverlap*fs),'nodelay
xb = zeros(size(xb_m,1),size(xb_m,2)+size(xb_s,2));
xb(:,1:2:end) = xb_m;
xb(:,2:2:end) = xb_s;

spec = melSpectrogram(xb,fs, ...
    'Window',hamming(params.WindowLength,'periodic'), ...
    'OverlapLength',params.WindowLength - params.HopLength, ...
    'FFTLength',params.FFTLength, ...
    'NumBands',params.NumBands, ...
    'FrequencyRange',[0,floor(fs/2)]);
spec = log10(spec+eps);

X = reshape(spec,size(spec,1),size(spec,2),size(x,2),[]);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%HelperWaveletFeatureVector
function features = HelperWaveletFeatureVector(x,sf)
x = mean(x,2);
features = featureMatrix(sf,x,'Transform','log');
features = mean(features,2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

See Also

`trainNetwork` | `trainingOptions` | `classify` | `layerGraph` | `batchNormalizationLayer` | `convolution2dLayer`

Related Examples

- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Deep Learning in MATLAB” on page 1-2

Keyword Spotting in Noise Using MFCC and LSTM Networks

This example shows how to identify a keyword in noisy speech using a deep learning network. In particular, the example uses a Bidirectional Long Short-Term Memory (BiLSTM) network and mel frequency cepstral coefficients (MFCC).

Introduction

Keyword spotting (KWS) is an essential component of voice-assist technologies, where the user speaks a predefined keyword to wake-up a system before speaking a complete command or query to the device.

This example trains a KWS deep network with feature sequences of mel-frequency cepstral coefficients (MFCC). The example also demonstrates how network accuracy in a noisy environment can be improved using data augmentation.

This example uses long short-term memory (LSTM) networks, which are a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. An LSTM layer (`lstmLayer`) can look at the time sequence in the forward direction, while a bidirectional LSTM layer (`bilstmLayer`) can look at the time sequence in both forward and backward directions. This example uses a bidirectional LSTM layer.

The example uses the google Speech Commands Dataset to train the deep learning model. To run the example, you must first download the data set. If you do not want to download the data set or train the network, then you can download and use a pretrained network by opening this example in MATLAB® and running lines 3-10 of the example.

Spot Keyword with Pretrained Network

Before going into the training process in detail, you will download and use a pretrained keyword spotting network to identify a keyword.

In this example, the keyword to spot is **YES**.

Read a test signal where the keyword is uttered.

```
[audioIn, fs] = audioread('keywordTestSignal.wav');
sound(audioIn,fs)
```

Download and load the pretrained network, the mean (M) and standard deviation (S) vectors used for feature normalization, as well as 2 audio files used for validating the network later in the example.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/KeywordSpotting.zip';
downloadNetFolder = tempdir;
netFolder = fullfile(downloadNetFolder,'KeywordSpotting');
if ~exist(netFolder,'dir')
    disp('Downloading pretrained network and audio files (4 files - 7 MB) ...')
    unzip(url,downloadNetFolder)
end
load(fullfile(netFolder,'KWSNet.mat'));
```

Create an `audioFeatureExtractor` (Audio Toolbox) object to perform feature extraction.

```
WindowLength = 512;
OverlapLength = 384;
```

```
afe = audioFeatureExtractor('SampleRate',fs, ...
                            'Window',hann(WindowLength,'periodic'), ...
                            'OverlapLength',OverlapLength, ...
                            'mfcc',true, ...
                            'mfccDelta', true, ...
                            'mfccDeltaDelta',true);
```

Extract features from the test signal and normalize them.

```
features = extract(afe,audioIn);

features = (features - M)./S;
```

Compute the keyword spotting binary mask. A mask value of one corresponds to a segment where the keyword was spotted.

```
mask = classify(KWSNet,features.');
```

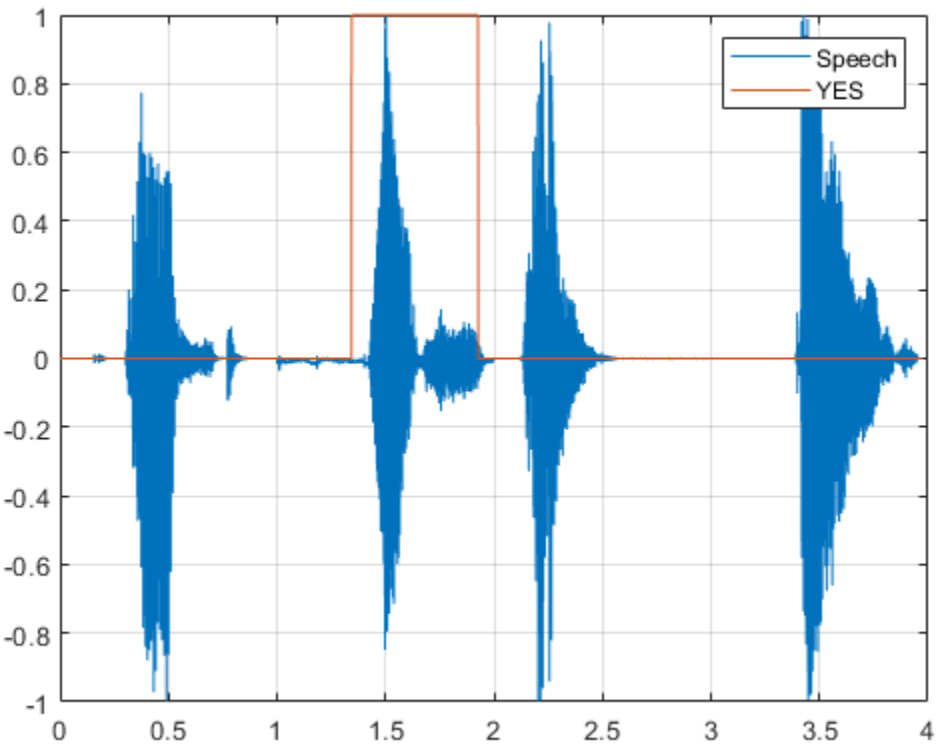
Each sample in the mask corresponds to 128 samples from the speech signal (`WindowLength - OverlapLength`).

Expand the mask to the length of the signal.

```
mask = repmat(mask, WindowLength-OverlapLength, 1);
mask = double(mask) - 1;
mask = mask(:);
```

Plot the test signal and the mask.

```
figure
audioIn = audioIn(1:length(mask));
t = (0:length(audioIn)-1)/fs;
plot(t, audioIn)
grid on
hold on
plot(t, mask)
legend('Speech', 'YES')
```



Listen to the spotted keyword.

```
sound(audioIn(mask==1),fs)
```

Detect Commands Using Streaming Audio from Microphone

Test your pre-trained command detection network on streaming audio from your microphone. Try saying random words, including the keyword (**YES**).

Call `generateMATLABFunction` (Audio Toolbox) on the `audioFeatureExtractor` object to create the feature extraction function. You will use this function in the processing loop.

```
generateMATLABFunction(afe,'generateKeywordFeatures','IsStreaming',true);
```

Define an audio device reader that can read audio from your microphone. Set the frame length to the hop length. This enables you to compute a new set of features for every new audio frame from the microphone.

```
HopLength = WindowLength - OverlapLength;
FrameLength = HopLength;
adr = audioDeviceReader('SampleRate',fs, ...
                        'SamplesPerFrame',FrameLength);
```

Create a scope for visualizing the speech signal and the estimated mask.

```
scope = timescope('SampleRate',fs, ...
                  'TimeSpanSource','property', ...
                  'TimeSpan',5, ...
```

```
                'TimeSpanOverrunAction','Scroll', ...
                'BufferLength',fs*5*2, ...
                'ShowLegend',true, ...
                'ChannelNames',{'Speech','Keyword Mask'}, ...
                'YLimits',[-1.2 1.2], ...
                'Title','Keyword Spotting');
```

Define the rate at which you estimate the mask. You will generate a mask once every `NumHopsPerUpdate` audio frames.

```
NumHopsPerUpdate = 16;
```

Initialize a buffer for the audio.

```
dataBuff = dsp.AsyncBuffer(WindowLength);
```

Initialize a buffer for the computed features.

```
featureBuff = dsp.AsyncBuffer(NumHopsPerUpdate);
```

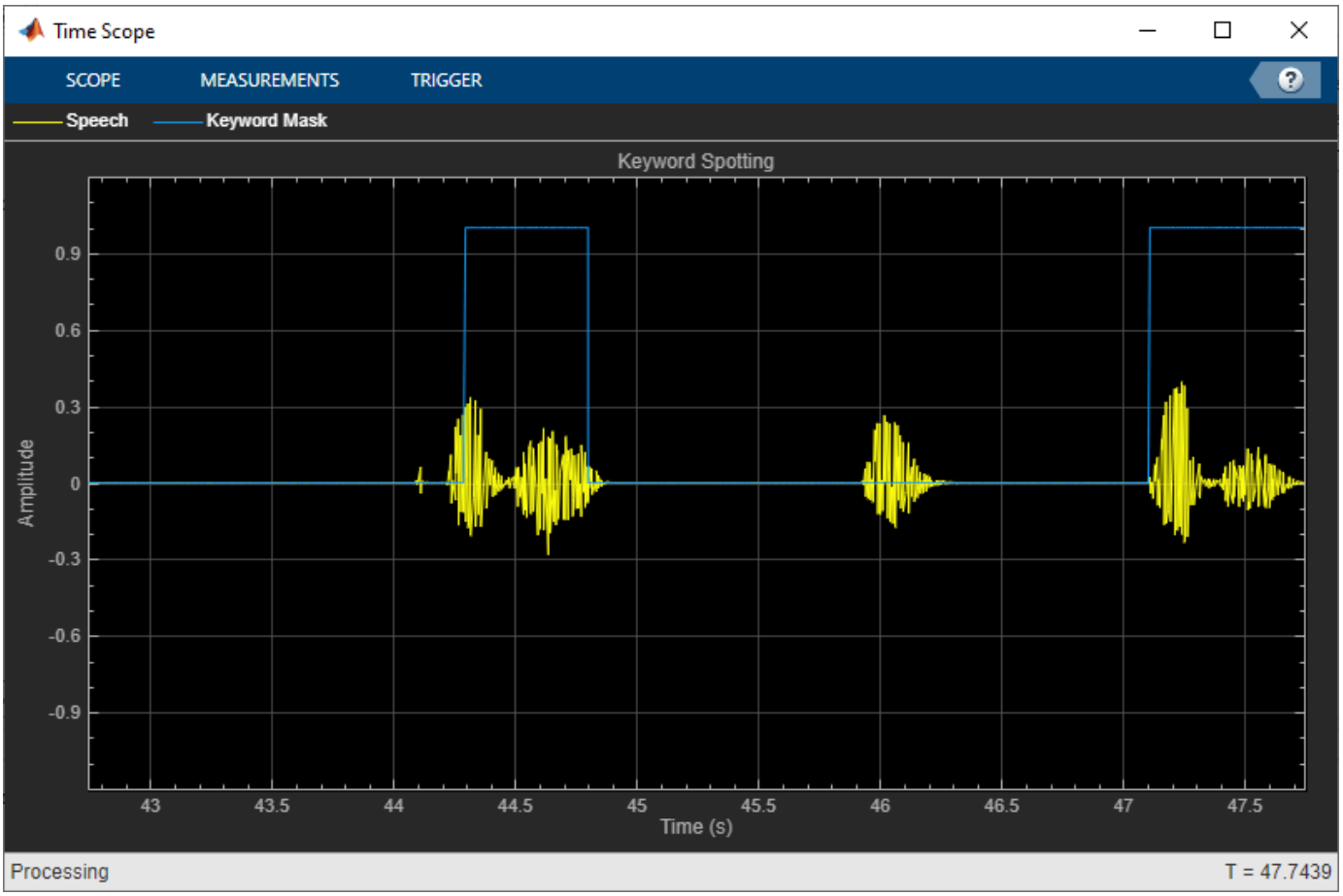
Initialize a buffer to manage plotting the audio and the mask.

```
plotBuff = dsp.AsyncBuffer(NumHopsPerUpdate*WindowLength);
```

To run the loop indefinitely, set timeLimit to `Inf`. To stop the simulation, close the scope.

```
timeLimit = 20;

tic
while toc < timeLimit

    data = adr();
    write(dataBuff, data);
    write(plotBuff, data);

    frame = read(dataBuff,WindowLength,OverlapLength);
    features = generateKeywordFeatures(frame,fs);
    write(featureBuff,features.');

    if featureBuff.NumUnreadSamples == NumHopsPerUpdate
        featureMatrix = read(featureBuff);
        featureMatrix(~isfinite(featureMatrix)) = 0;
        featureMatrix = (featureMatrix - M)./S;

        [keywordNet, v] = classifyAndUpdateState(KWSNet,featureMatrix.');
        v = double(v) - 1;
        v = repmat(v, HopLength, 1);
        v = v(:);
        v = mode(v);
        v = repmat(v, NumHopsPerUpdate * HopLength,1);

        data = read(plotBuff);
        scope([data, v]);

        if ~isVisible(scope)
            break;
        end
    end
```

```
    end
    hide(scope)
```



In the rest of the example, you will learn how to train the keyword spotting network.

Training Process Summary

The training process goes through the following steps:

1. Inspect a "gold standard" keyword spotting baseline on a validation signal.
2. Create training utterances from a noise-free dataset.
3. Train a keyword spotting LSTM network using MFCC sequences extracted from those utterances.
4. Check the network accuracy by comparing the validation baseline to the output of the network when applied to the validation signal.
5. Check the network accuracy for a validation signal corrupted by noise.
6. Augment the training dataset by injecting noise to the speech data using `audioDataAugmenter` (Audio Toolbox).
7. Retrain the network with the augmented dataset.
8. Verify that the retrained network now yields higher accuracy when applied to the noisy validation signal.

Inspect the Validation Signal

You use a sample speech signal to validate the KWS network. The validation signal consists 34 seconds of speech with the keyword **YES** appearing intermittently.

Load the validation signal.

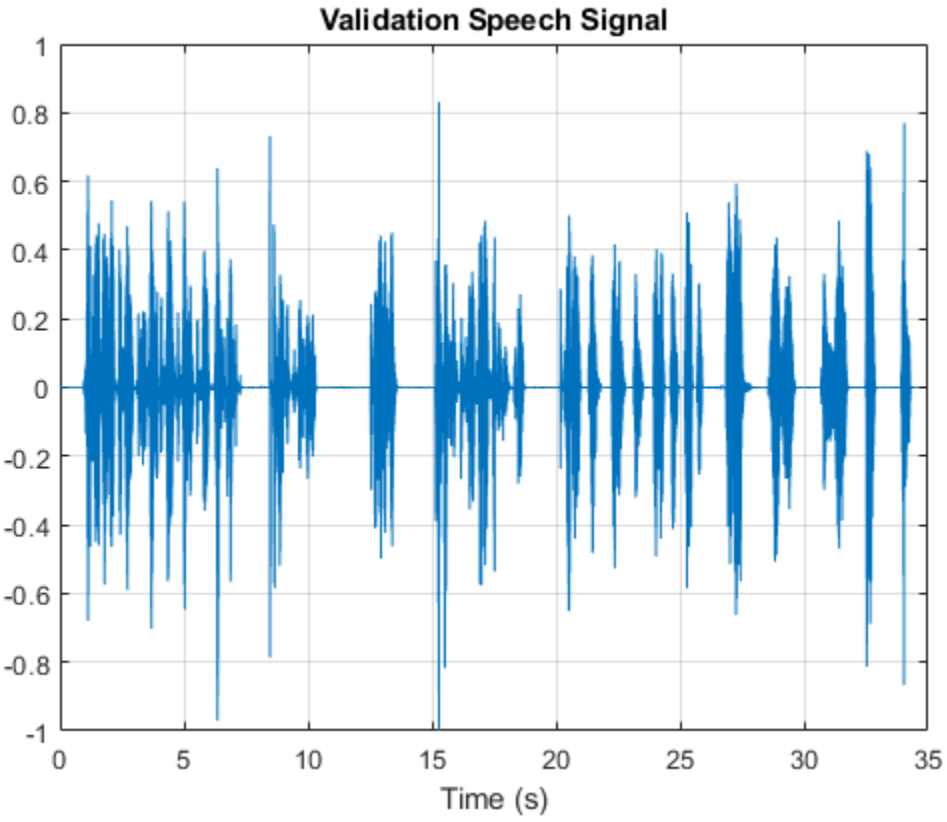
```
[audioIn,fs] = audioread(fullfile(netFolder,'KeywordSpeech-16-16-mono-34secs.flac'));
```

Listen to the signal.

```
sound(audioIn,fs)
```

Visualize the signal.

```
figure
t = (1/fs) * (0:length(audioIn)-1);
plot(t,audioIn);
grid on
xlabel('Time (s)')
title('Validation Speech Signal')
```



Inspect the KWS Baseline

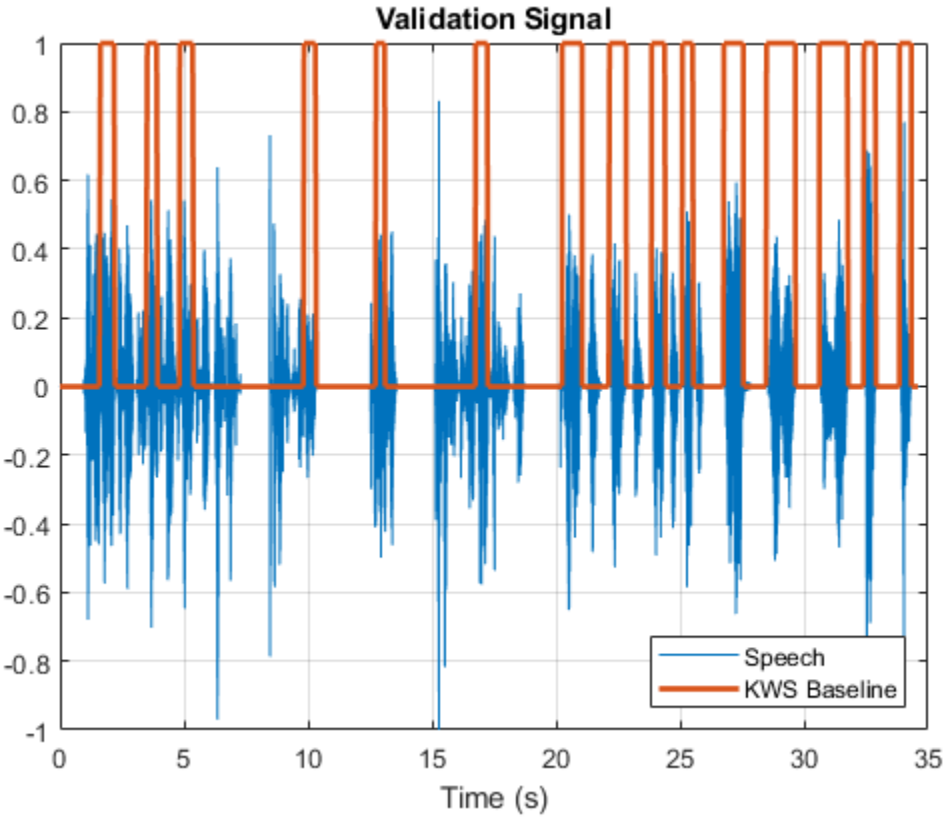
Load the KWS baseline. This baseline was obtained using `speech2text`: “Create Keyword Spotting Mask Using Audio Labeler” (Audio Toolbox).

```
load('KWSBaseline.mat','KWSBaseline')
```

The baseline is a logical vector of the same length as the validation audio signal. Segments in `audioIn` where the keyword is uttered are set to one in `KWSBaseline`.

Visualize the speech signal along with the KWS baseline.

```
fig = figure;
plot(t,[audioIn,KWSBaseline'])
grid on
xlabel('Time (s)')
legend('Speech','KWS Baseline','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
title("Validation Signal")
```



Listen to the speech segments identified as keywords.

```
sound(audioIn(KWSBaseline),fs)
```

The objective of the network that you train is to output a KWS mask of zeros and ones like this baseline.

Load Speech Commands Data Set

Download and extract the Google Speech Commands Dataset [1] on page 14-0 .

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';

downloadFolder = tempdir;
```

```
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading Google speech commands data set (1.5 GB)...')
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` (Audio Toolbox) that points to the data set.

```
ads = audioDatastore(datasetFolder,'LabelSource','foldername','Includesubfolders',true);
ads = shuffle(ads);
```

The dataset contains background noise files that are not used in this example. Use `subset` (Audio Toolbox) to create a new datastore that does not have the background noise files.

```
isBackNoise = ismember(ads.Labels,"background");
ads = subset(ads,~isBackNoise);
```

The dataset has approximately 65,000 one-second long utterances of 30 short words (including the keyword YES). Get a breakdown of the word distribution in the datastore.

```
countEachLabel(ads)
```

```
ans=30×2 table
    Label     Count
    ______    _____

    bed       1713
    bird      1731
    cat       1733
    dog       1746
    down      2359
    eight     2352
    five      2357
    four      2372
    go        2372
    happy     1742
    house     1750
    left      2353
    marvin    1746
    nine      2364
    no        2375
    off       2357
      ⋮
```

Split `ads` into two datastores: The first datastore contains files corresponding to the keyword. The second datastore contains all the other words.

```
keyword = 'yes';
isKeyword = ismember(ads.Labels,keyword);
ads_keyword = subset(ads,isKeyword);
ads_other = subset(ads,~isKeyword);
```

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to `false`. To run this example quickly, set `reduceDataset` to `true`.

```
reduceDataset = false;
if reduceDataset
```

```
    % Reduce the dataset by a factor of 20
    ads_keyword = splitEachLabel(ads_keyword,round(numel(ads_keyword.Files)  / 20));
    numUniqueLabels = numel(unique(ads_other.Labels));
    ads_other = splitEachLabel(ads_other,round(numel(ads_other.Files) / numUniqueLabels / 20));
end
```

Get a breakdown of the word distribution in each datastore. Shuffle the `ads_other` datastore so that consecutive reads return different words.

```
countEachLabel(ads_keyword)
```

```
ans=1×2 table
    Label    Count
    _____    _____

     yes     2377
```

```
countEachLabel(ads_other)
```

```
ans=29×2 table
    Label     Count
    ______    _____

    bed       1713
    bird      1731
    cat       1733
    dog       1746
    down      2359
    eight     2352
    five      2357
    four      2372
    go        2372
    happy     1742
    house     1750
    left      2353
    marvin    1746
    nine      2364
    no        2375
    off       2357
      ⋮
```

```
ads_other = shuffle(ads_other);
```

Create Training Sentences and Labels

The training datastores contain one-second speech signals where one word is uttered. You will create more complex training speech utterances that contain a mixture of the keyword along with other words.

Here is an example of a constructed utterance. Read one keyword from the keyword datastore and normalize it to have a maximum value of one.

```
yes = read(ads_keyword);
yes = yes / max(abs(yes));
```

The signal has non-speech portions (silence, background noise, etc.) that do not contain useful speech information. This example removes silence using `detectSpeech` (Audio Toolbox).

Get the start and end indices of the useful portion of the signal.

```
speechIndices = detectSpeech(yes,fs);
```

Randomly select the number of words to use in the synthesized training sentence. Use a maximum of 10 words.

```
numWords = randi([0 10]);
```

Randomly pick the location at which the keyword occurs.

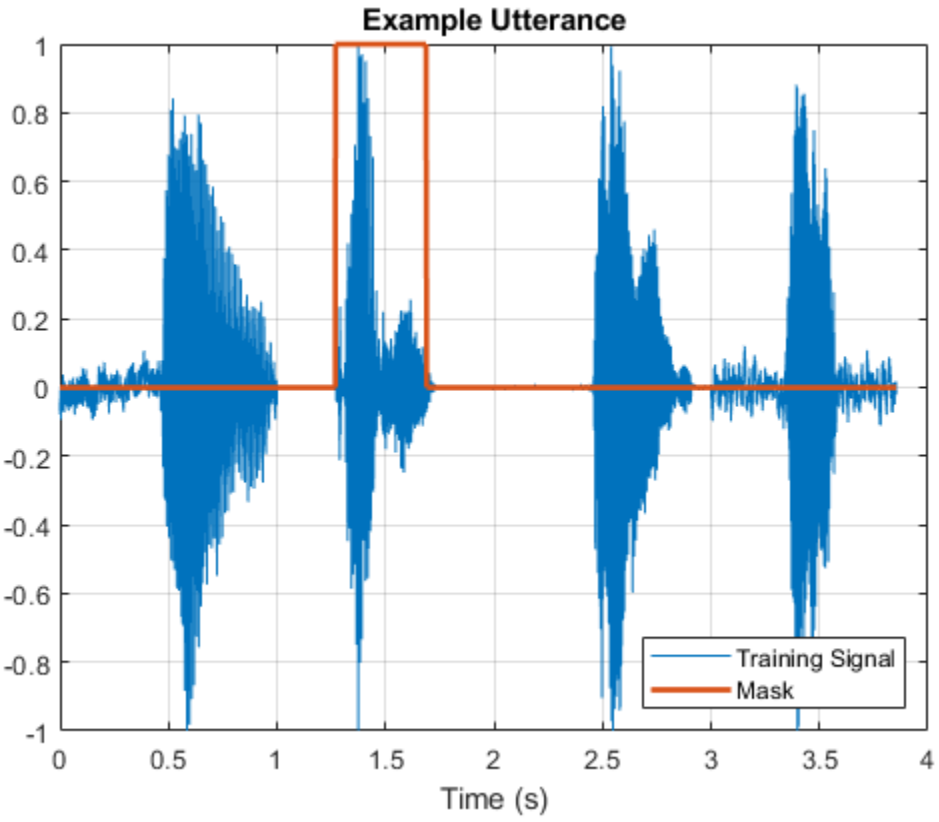
```
keywordLocation = randi([1 numWords+1]);
```

Read the desired number of non-keyword utterances, and construct the training sentence and mask.

```
sentence = [];
mask = [];
for index = 1:numWords+1
    if index == keywordLocation
        sentence = [sentence;yes]; %#ok
        newMask = zeros(size(yes));
        newMask(speechIndices(1,1):speechIndices(1,2)) = 1;
        mask = [mask;newMask]; %#ok
    else
        other = read(ads_other);
        other = other ./ max(abs(other));
        sentence = [sentence;other]; %#ok
        mask = [mask;zeros(size(other))]; %#ok
    end
end
```

Plot the training sentence along with the mask.

```
figure
t = (1/fs) * (0:length(sentence)-1);
fig = figure;
plot(t,[sentence,mask])
grid on
xlabel('Time (s)')
legend('Training Signal','Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
title("Example Utterance")
```



Listen to the training sentence.

```
sound(sentence,fs)
```

Extract Features

This example trains a deep learning network using 39 MFCC coefficients (13 MFCC, 13 delta and 13 delta-delta coefficients).

Define parameters required for MFCC extraction.

```
WindowLength = 512;
OverlapLength = 384;
```

Create an audioFeatureExtractor object to perform the feature extraction.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
                            'Window',hann(WindowLength,'periodic'), ...
                            'OverlapLength',OverlapLength, ...
                            'mfcc',true, ...
                            'mfccDelta',true, ...
                            'mfccDeltaDelta',true);
```

Extract the features.

```
featureMatrix = extract(afe,sentence);
size(featureMatrix)
```

```
ans = 1×2

   478    39
```

Note that you compute MFCC by sliding a window through the input, so the feature matrix is shorter than the input speech signal. Each row in `featureMatrix` corresponds to 128 samples from the speech signal (`WindowLength - OverlapLength`).

Compute a mask of the same length as `featureMatrix`.

```
HopLength = WindowLength - OverlapLength;
range = HopLength * (1:size(featureMatrix,1)) + HopLength;
featureMask = zeros(size(range));
for index = 1:numel(range)
    featureMask(index) = mode(mask( (index-1)*HopLength+1:(index-1)*HopLength+WindowLength ));
end
```

Extract Features from Training Dataset

Sentence synthesis and feature extraction for the whole training dataset can be quite time-consuming. To speed up processing, if you have Parallel Computing Toolbox™, partition the training datastore, and process each partition on a separate worker.

Select a number of datastore partitions.

```
numPartitions = 6;
```

Initialize cell arrays for the feature matrices and masks.

```
TrainingFeatures = {};
TrainingMasks= {};
```

Perform sentence synthesis, feature extraction, and mask creation using `parfor`.

```
emptyCategories = categorical([1 0]);
emptyCategories(:) = [];

tic
parfor ii = 1:numPartitions

    subads_keyword = partition(ads_keyword,numPartitions,ii);
    subads_other = partition(ads_other,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subads_keyword.Files),1);
    localMasks = cell(length(subads_keyword.Files),1);

    while hasdata(subads_keyword)

        % Create a training sentence
        [sentence,mask] = HelperSynthesizeSentence(subads_keyword,subads_other,fs,WindowLength);

        % Compute mfcc features
        featureMatrix = extract(afe, sentence);
        featureMatrix(~isfinite(featureMatrix)) = 0;

        % Create mask
```

```
        hopLength = WindowLength - OverlapLength;
        range = (hopLength) * (1:size(featureMatrix,1)) + hopLength;
        featureMask = zeros(size(range));
        for index = 1:numel(range)
            featureMask(index) = mode(mask( (index-1)*hopLength+1:(index-1)*hopLength+WindowLengt
        end

        localFeatures{count} = featureMatrix;
        localMasks{count} = [emptyCategories,categorical(featureMask)];

        count = count + 1;
    end

    TrainingFeatures = [TrainingFeatures;localFeatures];
    TrainingMasks = [TrainingMasks;localMasks];
end
fprintf('Training feature extraction took %f seconds.\n',toc)
```

```
Training feature extraction took 33.656404 seconds.
```

It is good practice to normalize all features to have zero mean and unity standard deviation. Compute the mean and standard deviation for each coefficient and use them to normalize the data.

```
sampleFeature = TrainingFeatures{1};
numFeatures = size(sampleFeature,2);
featuresMatrix = cat(1,TrainingFeatures{:});
if reduceDataset
    load(fullfile(netFolder,'keywordNetNoAugmentation.mat'),'keywordNetNoAugmentation','M','S');
else
    M = mean(featuresMatrix);
    S = std(featuresMatrix);
end
for index = 1:length(TrainingFeatures)
    f = TrainingFeatures{index};
    f = (f - M) ./ S;
    TrainingFeatures{index} = f.'; %#ok
end
```

Extract Validation Features

Extract MFCC features from the validation signal.

```
featureMatrix = extract(afe, audioIn);
featureMatrix(~isfinite(featureMatrix)) = 0;
```

Normalize the validation features.

```
FeaturesValidationClean = (featureMatrix - M)./S;
range = HopLength * (1:size(FeaturesValidationClean,1)) + HopLength;
```

Construct the validation KWS mask.

```
featureMask = zeros(size(range));
for index = 1:numel(range)
    featureMask(index) = mode(KWSBaseline( (index-1)*HopLength+1:(index-1)*HopLength+WindowLengt
end
BaselineV = categorical(featureMask);
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer` to look at the sequence in both forward and backward directions.

Specify the input size to be sequences of size `numFeatures`. Specify two hidden bidirectional LSTM layers with an output size of 150 and output a sequence. This command instructs the bidirectional LSTM layer to map the input time series into 150 features that are passed to the next layer. Specify two classes by including a fully connected layer of size 2, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(150,"OutputMode","sequence")
    bilstmLayer(150,"OutputMode","sequence")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ];
```

Define Training Options

Specify the training options for the classifier. Set `MaxEpochs` to 10 so that the network makes 10 passes through the training data. Set `MiniBatchSize` to `64` so that the network looks at 64 training signals at a time. Set `Plots` to `"training-progress"` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot. Set `Shuffle` to `"every-epoch"` to shuffle the training sequence at the beginning of each epoch. Set `LearnRateSchedule` to `"piecewise"` to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (5) has passed. Set `ValidationData` to the validation predictors and targets.

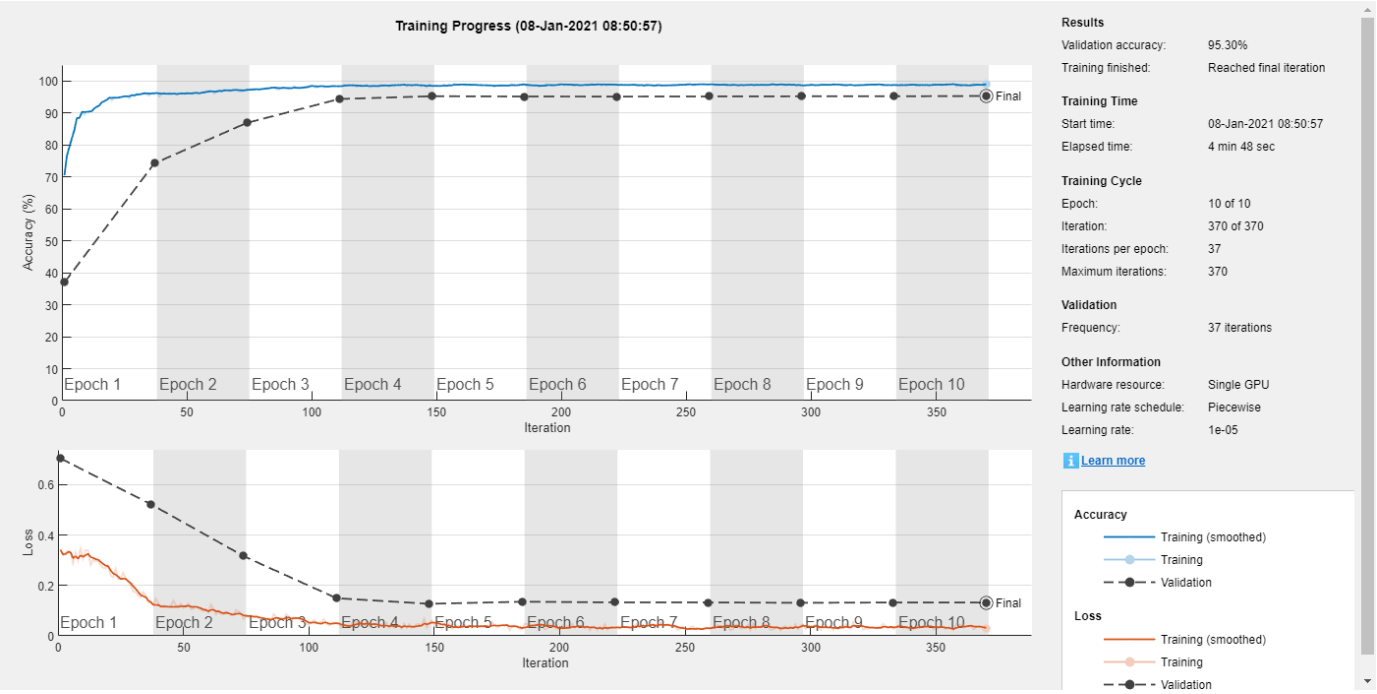
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
maxEpochs = 10;
miniBatchSize = 64;
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-4, ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(numel(TrainingFeatures)/miniBatchSize), ...
    "ValidationData",{FeaturesValidationClean.',BaselineV}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
[keywordNetNoAugmentation,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,options)
```




```
if reduceDataset
    load(fullfile(netFolder,'keywordNetNoAugmentation.mat'),'keywordNetNoAugmentation','M','S');
end
```

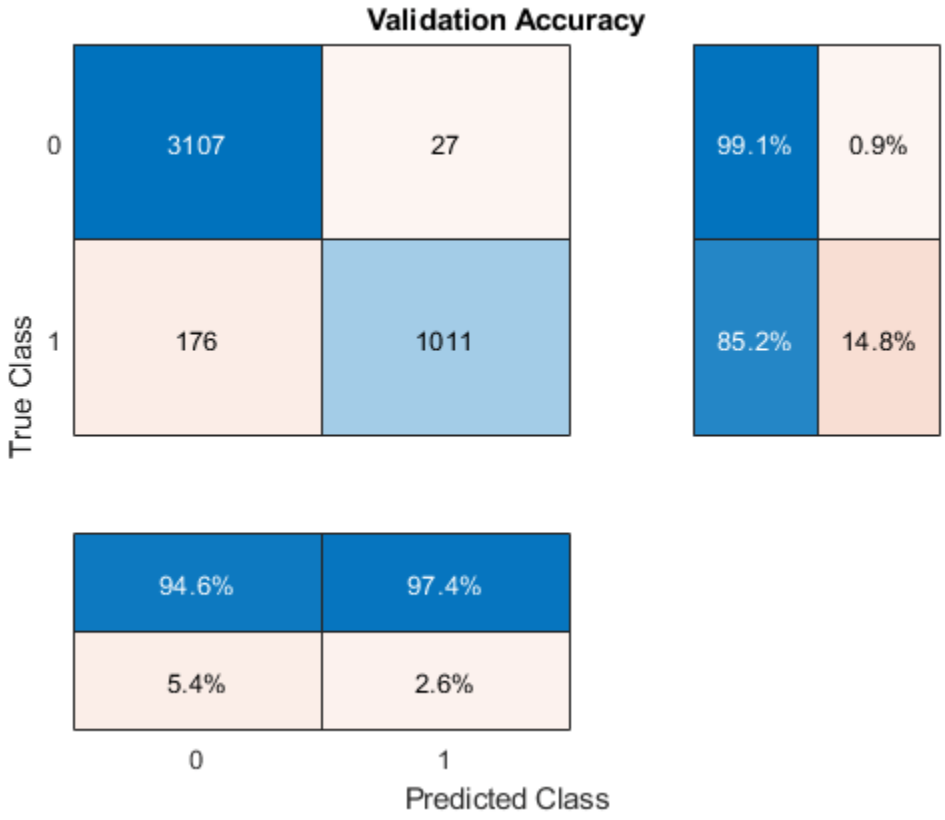
Check Network Accuracy for Noise-Free Validation Signal

Estimate the KWS mask for the validation signal using the trained network.

```
v = classify(keywordNetNoAugmentation,FeaturesValidationClean.');
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels.

```
figure
cm = confusionchart(BaselineV,v,"title","Validation Accuracy");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



Convert the network output from categorical to double.

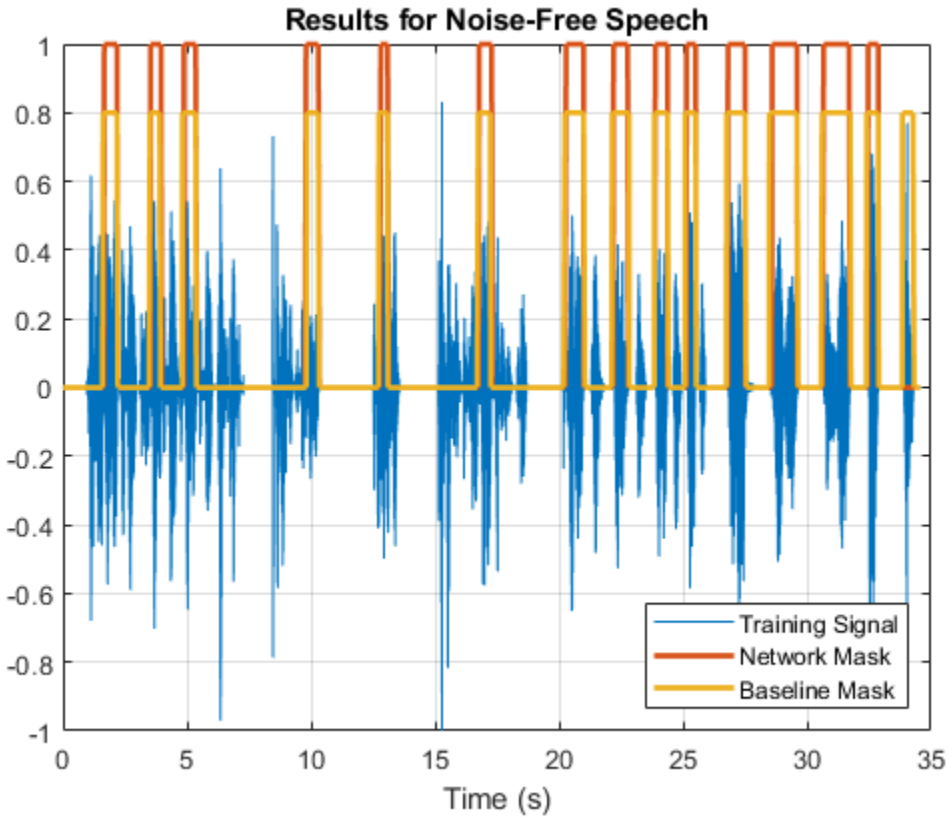
```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

Listen to the keyword areas identified by the network.

```
sound(audioIn(logical(v)),fs)
```

Visualize the estimated and expected KWS masks.

```
baseline = double(BaselineV) - 1;
baseline = repmat(baseline,HopLength,1);
baseline = baseline(:);

t = (1/fs) * (0:length(v)-1);
fig = figure;
plot(t,[audioIn(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noise-Free Speech')
```



Check Network Accuracy for a Noisy Validation Signal

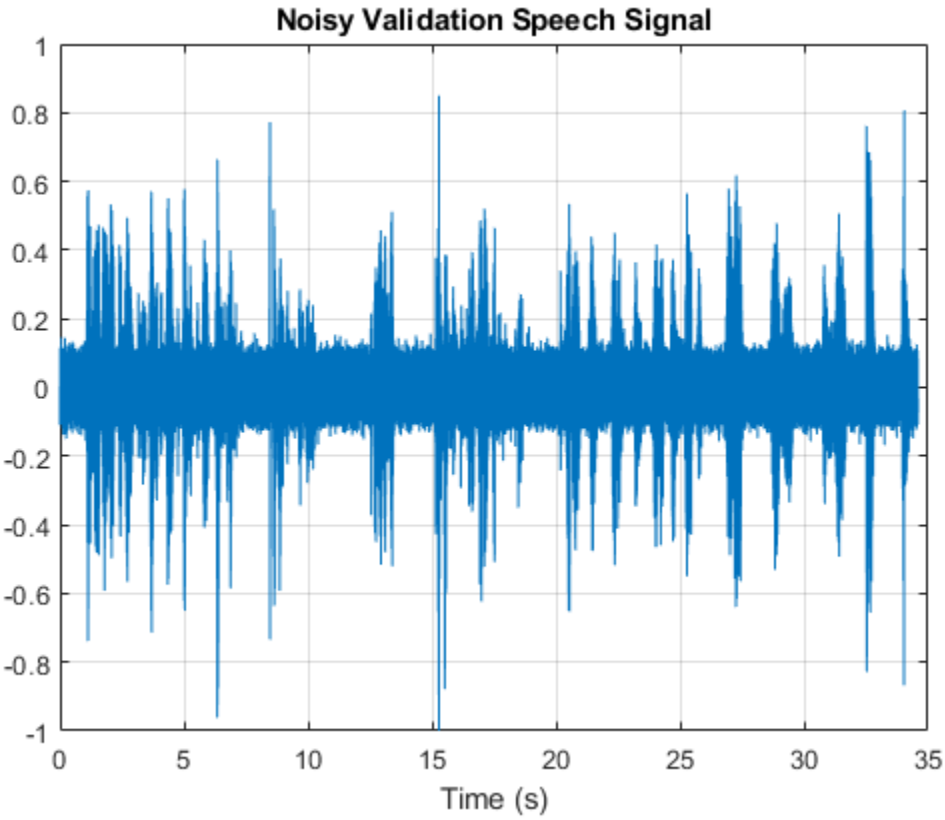
You will now check the network accuracy for a noisy speech signal. The noisy signal was obtained by corrupting the clean validation signal by additive white Gaussian noise.

Load the noisy signal.

```
[audioInNoisy,fs] = audioread(fullfile(netFolder,'NoisyKeywordSpeech-16-16-mono-34secs.flac'));
sound(audioInNoisy,fs)
```

Visualize the signal.

```
figure
t = (1/fs) * (0:length(audioInNoisy)-1);
plot(t,audioInNoisy)
grid on
xlabel('Time (s)')
title('Noisy Validation Speech Signal')
```



Extract the feature matrix from the noisy signal.

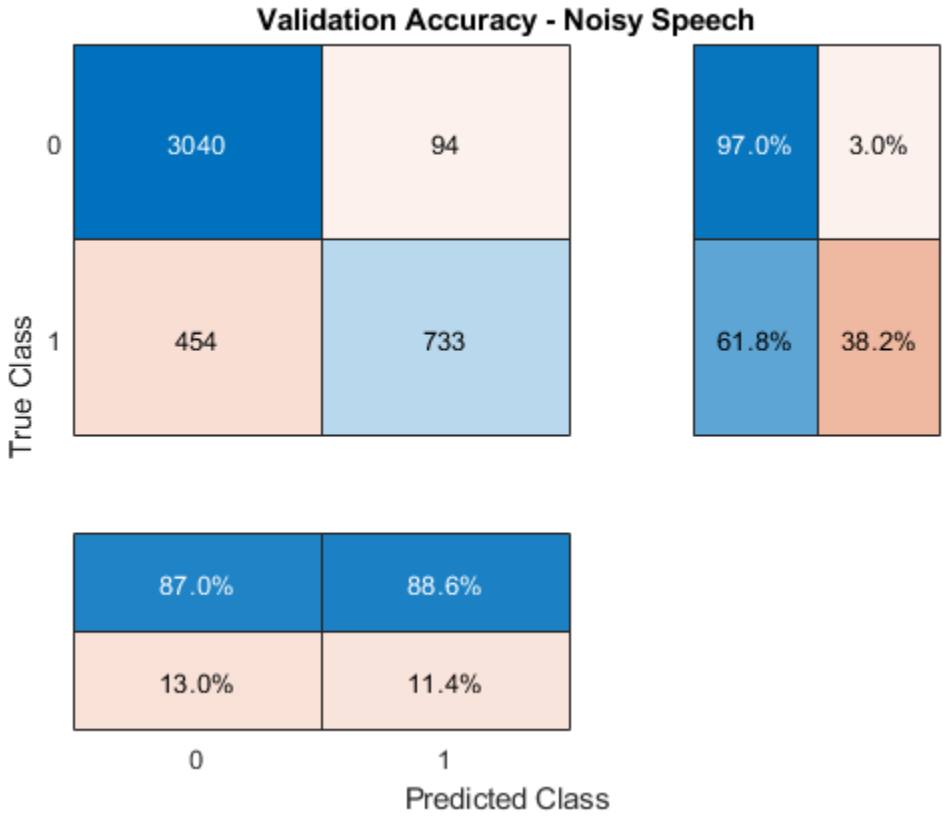
```
featureMatrixV = extract(afe, audioInNoisy);
featureMatrixV(~isfinite(featureMatrixV)) = 0;
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

Pass the feature matrix to the network.

```
v = classify(keywordNetNoAugmentation,FeaturesValidationNoisy.');
```

Compare the network output to the baseline. Note that the accuracy is lower than the one you got for a clean signal.

```
figure
cm = confusionchart(BaselineV,v,"title","Validation Accuracy - Noisy Speech");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



Convert the network output from categorical to double.

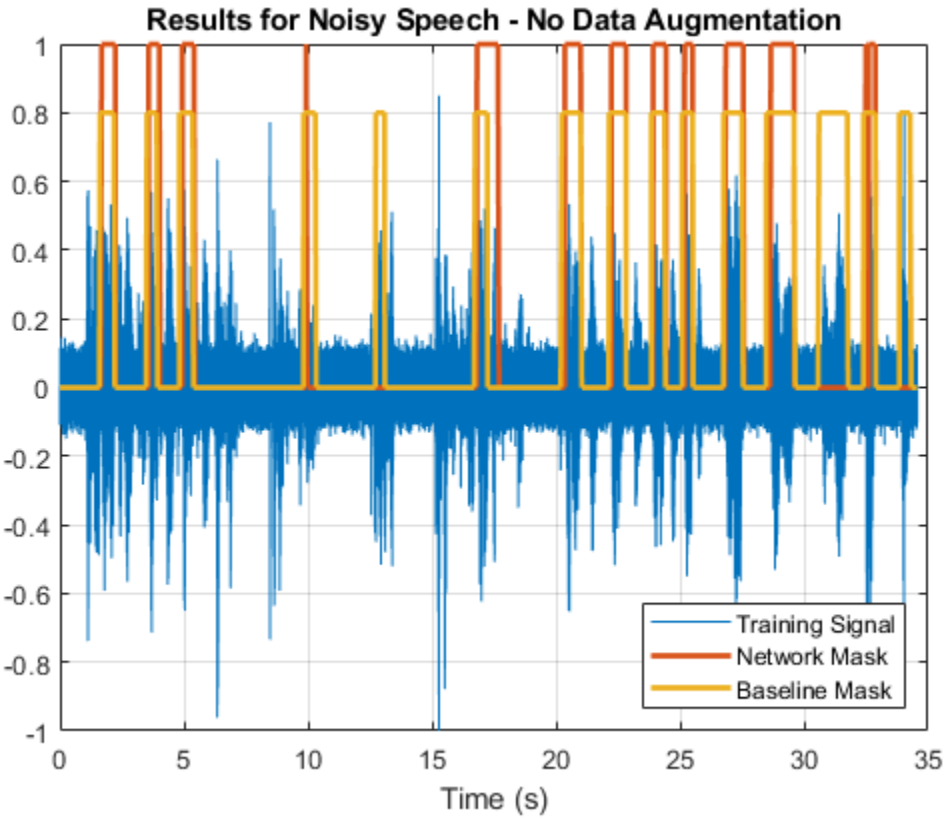
```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

Listen to the keyword areas identified by the network.

```
sound(audioIn(logical(v)),fs)
```

Visualize the estimated and baseline masks.

```
t = (1/fs)*(0:length(v)-1);
fig = figure;
plot(t,[audioInNoisy(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noisy Speech - No Data Augmentation')
```



Perform Data Augmentation

The trained network did not perform well on a noisy signal because the trained dataset contained only noise-free sentences. You will rectify this by augmenting your dataset to include noisy sentences.

Use `audioDataAugmenter` (Audio Toolbox) to augment your dataset.

```
ada = audioDataAugmenter('TimeStretchProbability',0, ...
                         'PitchShiftProbability',0, ...
                         'VolumeControlProbability',0, ...
                         'TimeShiftProbability',0, ...
                         'SNRRange',[-1, 1], ...
                         'AddNoiseProbability',0.85);
```

With these settings, the `audioDataAugmenter` object corrupts an input audio signal with white Gaussian noise with a probability of 85%. The SNR is randomly selected from the range [-1 1] (in dB). There is a 15% probability that the augmenter does not modify your input signal.

As an example, pass an audio signal to the augmenter.

```
reset(ads_keyword)
x = read(ads_keyword);
data = augment(ada,x,fs)
```

```
data=1×2 table
        Audio          AugmentationInfo
    _______________    _______________
```

```
    {16000×1 double}      [1×1 struct]
```

Inspect the `AugmentationInfo` variable in `data` to verify how the signal was modified.

```
data.AugmentationInfo
```

```
ans = struct with fields:
    SNR: 0.3410
```

Reset the datastores.

```
reset(ads_keyword)
reset(ads_other)
```

Initialize the feature and mask cells.

```
TrainingFeatures = {};
TrainingMasks = {};
```

Perform feature extraction again. Each signal is corrupted by noise with a probability of 85%, so your augmented dataset has approximately 85% noisy data and 15% noise-free data.

```
tic
parfor ii = 1:numPartitions

    subads_keyword = partition(ads_keyword,numPartitions,ii);
    subads_other = partition(ads_other,numPartitions,ii);

    count = 1;
    localFeatures = cell(length(subads_keyword.Files),1);
    localMasks = cell(length(subads_keyword.Files),1);

    while hasdata(subads_keyword)

        [sentence,mask] = HelperSynthesizeSentence(subads_keyword,subads_other,fs,WindowLength);

        % Corrupt with noise
        augmentedData = augment(ada,sentence,fs);
        sentence = augmentedData.Audio{1};

        % Compute mfcc features
        featureMatrix = extract(afe, sentence);
        featureMatrix(~isfinite(featureMatrix)) = 0;

        hopLength = WindowLength - OverlapLength;
        range = hopLength * (1:size(featureMatrix,1)) + hopLength;
        featureMask = zeros(size(range));
        for index = 1:numel(range)
            featureMask(index) = mode(mask( (index-1)*hopLength+1:(index-1)*hopLength+WindowLeng
        end

        localFeatures{count} = featureMatrix;
        localMasks{count} = [emptyCategories,categorical(featureMask)];

        count = count + 1;
    end
```

```
    TrainingFeatures = [TrainingFeatures;localFeatures];
    TrainingMasks = [TrainingMasks;localMasks];
end
fprintf('Training feature extraction took %f seconds.\n',toc)
```

```
Training feature extraction took 36.090923 seconds.
```

Compute the mean and standard deviation for each coefficient; use them to normalize the data.

```
sampleFeature = TrainingFeatures{1};
numFeatures = size(sampleFeature,2);
featuresMatrix = cat(1,TrainingFeatures{:});
if reduceDataset
    load(fullfile(netFolder,'KWSNet.mat'),'KWSNet','M','S');
else
    M = mean(featuresMatrix);
    S = std(featuresMatrix);
end
for index = 1:length(TrainingFeatures)
    f = TrainingFeatures{index};
    f = (f - M) ./ S;
    TrainingFeatures{index} = f.'; %#ok
end
```

Normalize the validation features with the new mean and standard deviation values.

```
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

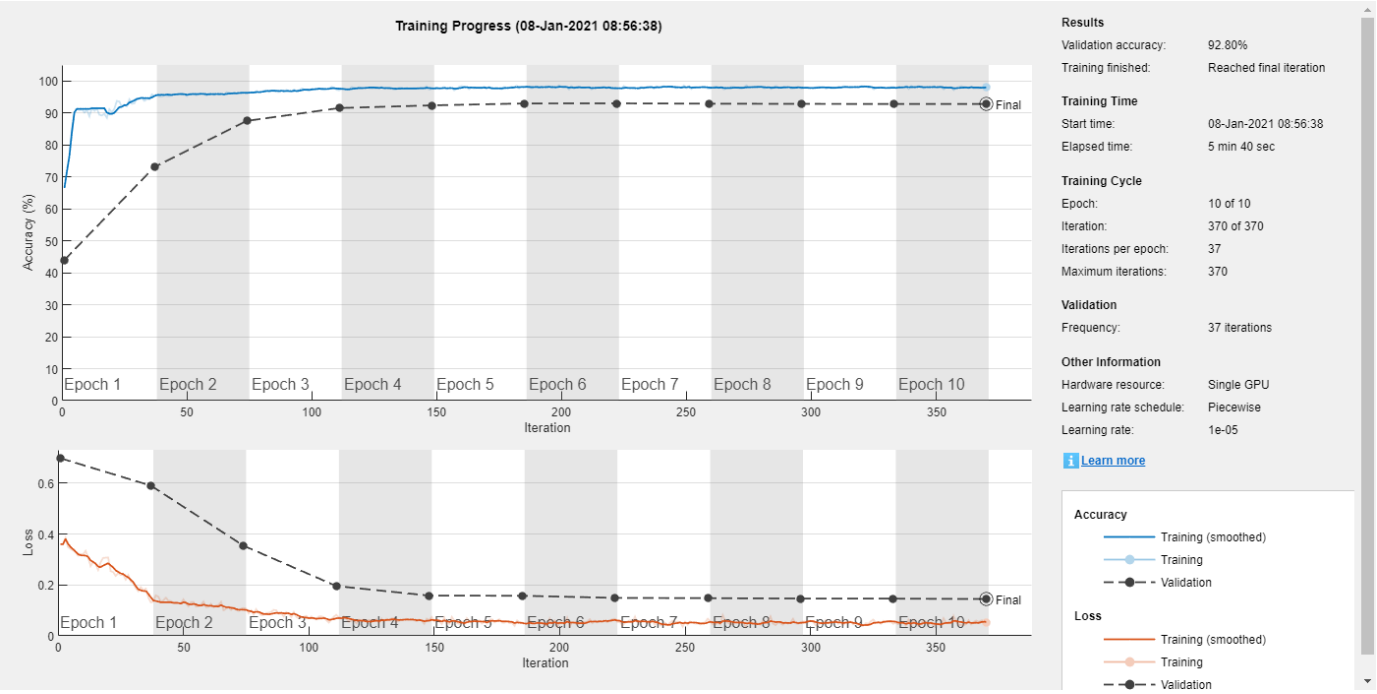
Retrain Network with Augmented Dataset

Recreate the training options. Use the noisy baseline features and mask for validation.

```
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-4, ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(numel(TrainingFeatures)/miniBatchSize), ...
    "ValidationData",{FeaturesValidationNoisy.',BaselineV}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the network.

```
[KWSNet,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,options);
```



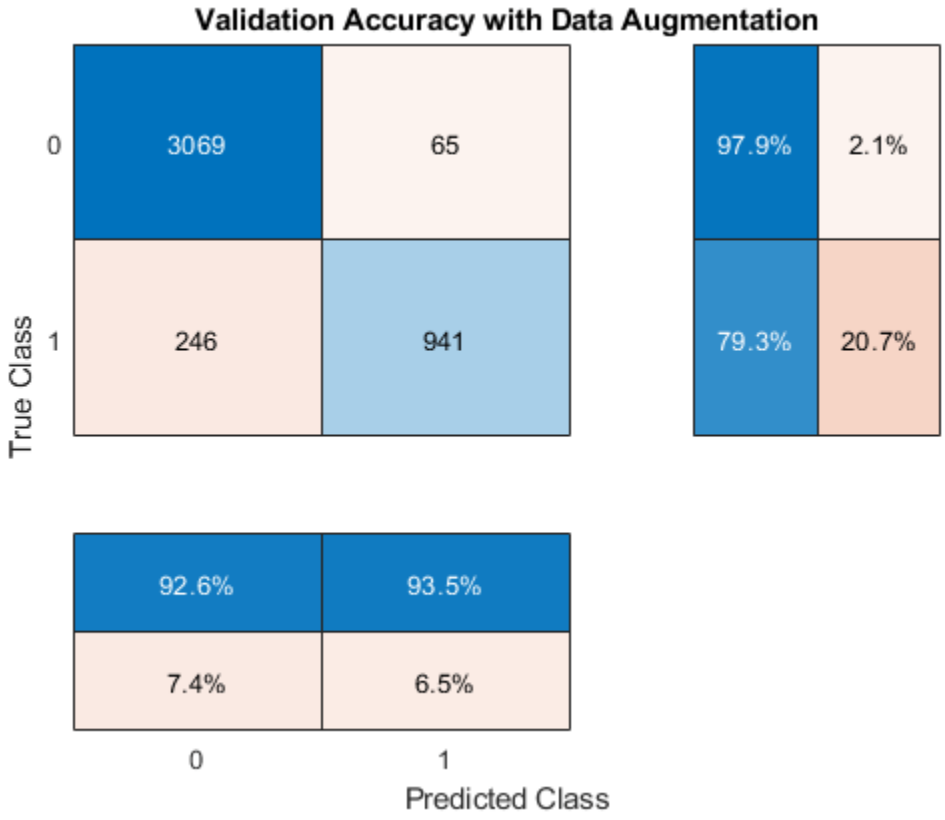

```
if reduceDataset
    load(fullfile(netFolder,'KWSNet.mat'));
end
```

Verify the network accuracy on the validation signal.

```
v = classify(KWSNet,FeaturesValidationNoisy.');
```

Compare the estimated and expected KWS masks.

```
figure
cm = confusionchart(BaselineV,v,"title","Validation Accuracy with Data Augmentation");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```

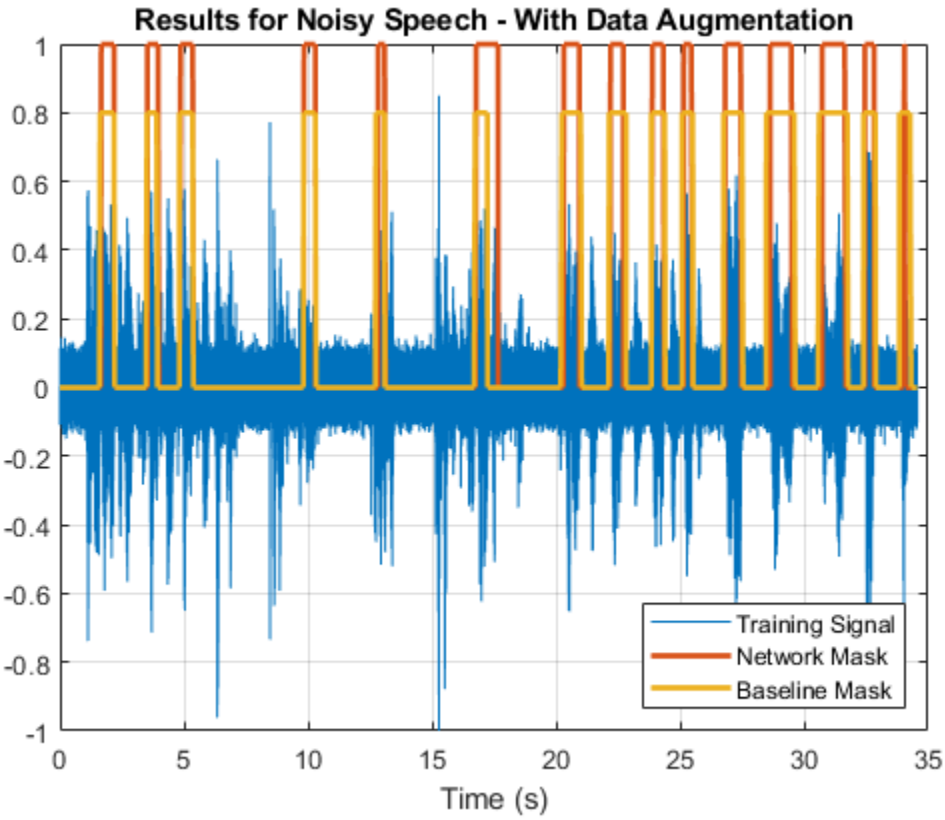


Listen to the identified keyword regions.

```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);

sound(audioIn(logical(v)),fs)
```

Visualize the estimated and expected masks.

```
fig = figure;
plot(t,[audioInNoisy(1:length(v)),v,0.8*baseline])
grid on
xlabel('Time (s)')
legend('Training Signal','Network Mask','Baseline Mask','Location','southeast')
l = findall(fig,'type','line');
l(1).LineWidth = 2;
l(2).LineWidth = 2;
title('Results for Noisy Speech - With Data Augmentation')
```



References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license.

Appendix - Helper Functions

```
function [sentence,mask] = HelperSynthesizeSentence(ads_keyword,ads_other,fs,minlength)

% Read one keyword
keyword = read(ads_keyword);
keyword = keyword ./ max(abs(keyword));

% Identify region of interest
speechIndices = detectSpeech(keyword,fs);
if isempty(speechIndices) || diff(speechIndices(1,:)) <= minlength
    speechIndices = [1,length(keyword)];
end
keyword = keyword(speechIndices(1,1):speechIndices(1,2));

% Pick a random number of other words (between 0 and 10)
numWords = randi([0 10]);
% Pick where to insert keyword
loc = randi([1 numWords+1]);
sentence = [];
```

```
mask = [];
for index = 1:numWords+1
    if index==loc
        sentence = [sentence;keyword];
        newMask = ones(size(keyword));
        mask = [mask ;newMask];
    else

        other = read(ads_other);
        other = other ./ max(abs(other));
        sentence = [sentence;other];
        mask = [mask;zeros(size(other))];
    end
end
end
```

See Also

`bilstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Speech Emotion Recognition

This example illustrates a simple speech emotion recognition (SER) system using a BiLSTM network. You begin by downloading the data set and then testing the trained network on individual files. The network was trained on a small German-language database [1] on page 14-0 .

The example walks you through training the network, which includes downloading, augmenting, and training the dataset. Finally, you perform leave-one-speaker-out (LOSO) 10-fold cross validation to evaluate the network architecture.

The features used in this example were chosen using sequential feature selection, similar to the method described in “Sequential Feature Selection for Audio Features” (Audio Toolbox).

Download Data Set

Download the Berlin Database of Emotional Speech [1] on page 14-0 . The database contains 535 utterances spoken by 10 actors intended to convey one of the following emotions: anger, boredom, disgust, anxiety/fear, happiness, sadness, or neutral. The emotions are text independent.

```
url = "http://emodb.bilderbar.info/download/download.zip";
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,"Emo-DB");

if ~exist(datasetFolder,'dir')
    disp('Downloading Emo-DB (40.5 MB) ...')
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` (Audio Toolbox) that points to the audio files.

```
ads = audioDatastore(fullfile(datasetFolder,"wav"));
```

The file names are codes indicating the speaker ID, text spoken, emotion, and version. The website contains a key for interpreting the code and additional information about the speakers such as gender and age. Create a table with the variables `Speaker` and `Emotion`. Decode the file names into the table.

```
filepaths = ads.Files;
emotionCodes = cellfun(@(x)x(end-5),filepaths,'UniformOutput',false);
emotions = replace(emotionCodes,{'W','L','E','A','F','T','N'}, ...
    {'Anger','Boredom','Disgust','Anxiety/Fear','Happiness','Sadness','Neutral'});

speakerCodes = cellfun(@(x)x(end-10:end-9),filepaths,'UniformOutput',false);
labelTable = cell2table([speakerCodes,emotions],'VariableNames',{'Speaker','Emotion'});
labelTable.Emotion = categorical(labelTable.Emotion);
labelTable.Speaker = categorical(labelTable.Speaker);
summary(labelTable)
```

```
Variables:

    Speaker: 535×1 categorical

        Values:

            03       49
            08       58
            09       43
```

```
            10       38
            11       55
            12       35
            13       61
            14       69
            15       56
            16       71

    Emotion: 535×1 categorical

        Values:

            Anger             127
            Anxiety/Fear       69
            Boredom            81
            Disgust            46
            Happiness          71
            Neutral            79
            Sadness            62
```

`labelTable` is in the same order as the files in `audioDatastore`. Set the `Labels` property of the `audioDatastore` to the `labelTable`.

```
ads.Labels = labelTable;
```

Perform Speech Emotion Recognition

Download and load the pretrained network, the `audioFeatureExtractor` (Audio Toolbox) object used to train the network, and normalization factors for the features. This network was trained using all speakers in the data set except speaker `03`.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/SpeechEmotionRecognition.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'SpeechEmotionRecognition');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 1.5 MB) ...')
        unzip(url,downloadNetFolder)
    end
load(fullfile(netFolder,'network_Audio_SER.mat'));
```

The sample rate set on the `audioFeatureExtractor` corresponds to the sample rate of the data set.

```
fs = afe.SampleRate;
```

Select a speaker and emotion, then subset the datastore to only include the chosen speaker and emotion. Read from the datastore and listen to the file.

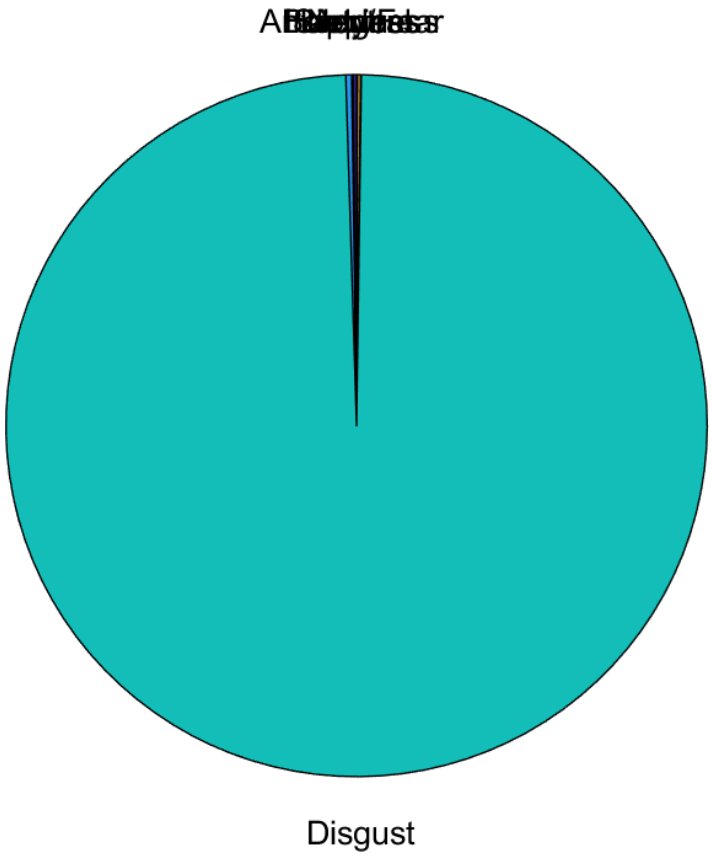
```
speaker = 03 - male, 31 ;
emotion = Disgust ;

adsSubset = subset(ads,ads.Labels.Speaker==speaker & ads.Labels.Emotion == emotion);

audio = read(adsSubset);
sound(audio,fs)
```

Use the `audioFeatureExtractor` object to extract the features and then transpose them so that time is along rows. Normalize the features and then convert them to 20-element sequences with 10-element overlap, which corresponds to approximately 600 ms windows with 300 ms overlap. Use the supporting function, HelperFeatureVector2Sequence on page 14-0 , to convert the array of feature vectors to sequences.

```
features = (extract(afe,audio))';

featuresNormalized = (features - normalizers.Mean)./normalizers.StandardDeviation;

numOverlap = 10 ;
featureSequences = HelperFeatureVector2Sequence(featuresNormalized,20,numOverlap);
```

Feed the feature sequences into the network for prediction. Compute the mean prediction and plot the probability distribution of the chosen emotions as a pie chart. You can try different speakers, emotions, sequence overlap, and prediction average to test the network's performance. To get a realistic approximation of the network's performance, use speaker `03`, which the network was not trained on.

```
YPred = double(predict(net,featureSequences));

average = mode;
switch average
    case 'mean'
        probs = mean(YPred,1);
    case 'median'
        probs = median(YPred,1);
    case 'mode'
        probs = mode(YPred,1);
end

pie(probs./sum(probs),string(net.Layers(end).Classes))
```



The remainder of the example illustrates how the network was trained and validated.

Train Network

The 10-fold cross validation accuracy of a first attempt at training was about 60% because of insufficient training data. A model trained on the insufficient data overfits some folds and underfits others. To improve overall fit, increase the size of the dataset using `audioDataAugmenter` (Audio Toolbox). 50 augmentations per file was chosen empirically as a good tradeoff between processing time and accuracy improvement. You can decrease the number of augmentations to speed up the example.

Create an `audioDataAugmenter` object. Set the probability of applying pitch shifting to `0.5` and use the default range. Set the probability of applying time shifting to `1` and use a range of `[-0.3,0.3]` seconds. Set the probability of adding noise to `1` and specify the SNR range as `[-20,40]` dB.

```
numAugmentations = 50;
augmenter = audioDataAugmenter('NumAugmentations',numAugmentations, ...
    'TimeStretchProbability',0, ...
    'VolumeControlProbability',0, ...
```

```
    ...
    'PitchShiftProbability',0.5, ...
    ...
    'TimeShiftProbability',1, ...
    'TimeShiftRange',[-0.3,0.3], ...
    ...
    'AddNoiseProbability',1, ...
    'SNRRange', [-20,40]);
```

Create a new folder in your current folder to hold the augmented data set.

```
currentDir = pwd;
writeDirectory = fullfile(currentDir,'augmentedData');
mkdir(writeDirectory)
```

For each file in the audio datastore:

1. Create 50 augmentations.
2. Normalize the audio to have a max absolute value of 1.
3. Write the augmented audio data as a WAV file. Append `_augK` to each of the file names, where *K* is the augmentation number. To speed up processing, use `parfor` and partition the datastore.

This method of augmenting the database is time consuming (approximately 1 hour) and space consuming (approximately 26 GB). However, when iterating on choosing a network architecture or feature extraction pipeline, this upfront cost is generally advantageous.

```
N = numel(ads.Files)*numAugmentations;
myWaitBar = HelperPoolWaitbar(N,"Augmenting Dataset...");

reset(ads)

numPartitions = 18;

tic
parfor ii = 1:numPartitions
    adsPart = partition(ads,numPartitions,ii);
    while hasdata(adsPart)
        [x,adsInfo] = read(adsPart);
        data = augment(augmenter,x,fs);

        [~,fn] = fileparts(adsInfo.FileName);
        for i = 1:size(data,1)
            augmentedAudio = data.Audio{i};
            augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
            augNum = num2str(i);
            if numel(augNum)==1
                iString = ['0',augNum];
            else
                iString = augNum;
            end
            audiowrite(fullfile(writeDirectory,sprintf('%s_aug%s.wav',fn,iString)),augmentedAudio
            increment(myWaitBar)
        end
    end
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
delete(myWaitBar)
fprintf('Augmentation complete (%0.2f minutes).\n',toc/60)
```

```
Augmentation complete (6.28 minutes).
```

Create an audio datastore that points to the augmented data set. Replicate the rows of the label table of the original datastore `NumAugmentations` times to determine the labels of the augmented datastore.

```
adsAug = audioDatastore(writeDirectory);
adsAug.Labels = repelem(ads.Labels,augmenter.NumAugmentations,1);
```

Create an `audioFeatureExtractor` (Audio Toolbox) object. Set `Window` to a periodic 30 ms Hamming window, `OverlapLength` to `0`, and `SampleRate` to the sample rate of the database. Set `gtcc`, `gtccDelta`, `mfccDelta`, and `spectralCrest` to `true` to extract them. Set `SpectralDescriptorInput` to `melSpectrum` so that the `spectralCrest` is calculated for the mel spectrum.

```
win = hamming(round(0.03*fs),"periodic");
overlapLength = 0;

afe = audioFeatureExtractor( ...
    'Window',win, ...
    'OverlapLength',overlapLength, ...
    'SampleRate',fs, ...
    ...
    'gtcc',true, ...
    'gtccDelta',true, ...
    'mfccDelta',true, ...
    ...
    'SpectralDescriptorInput','melSpectrum', ...
    'spectralCrest',true);
```

Train for Deployment

When you train for deployment, use all available speakers in the data set. Set the training datastore to the augmented datastore.

```
adsTrain = adsAug;
```

Convert the training audio datastore to a tall array. If you have Parallel Computing Toolbox™, the extraction is automatically parallelized. If you do not have Parallel Computing Toolbox™, the code continues to run.

```
tallTrain = tall(adsTrain);
```

Extract the training features and reorient the features so that time is along rows to be compatible with `sequenceInputLayer`.

```
featuresTallTrain = cellfun(@(x)extract(afe,x),tallTrain,"UniformOutput",false);
featuresTallTrain = cellfun(@(x)x',featuresTallTrain,"UniformOutput",false);
featuresTrain = gather(featuresTallTrain);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 7 sec
Evaluation completed in 1 min 7 sec
```

Use the training set to determine the mean and standard deviation of each feature.

```
allFeatures = cat(2,featuresTrain{:});
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');

featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,'UniformOutput',false);
```

Buffer the feature vectors into sequences so that each sequence consists of 20 feature vectors with overlaps of 10 feature vectors.

```
featureVectorsPerSequence = 20;
featureVectorOverlap = 10;
[sequencesTrain,sequencePerFileTrain] = HelperFeatureVector2Sequence(featuresTrain,featureVectors
```

Replicate the labels of the training and validation sets so that they are in one-to-one correspondence with the sequences. Not all speakers have utterances for all emotions. Create an empty `categorical` array that contains all the emotional categories and append it to the validation labels so that the categorical array contains all emotions.

```
labelsTrain = repelem(adsTrain.Labels.Emotion,[sequencePerFileTrain{:}]);

emptyEmotions = ads.Labels.Emotion;
emptyEmotions(:) = [];
```

Define a BiLSTM network using `bilstmLayer`. Place a `dropoutLayer` before and after the `bilstmLayer` to help prevent overfitting.

```
dropoutProb1 = 0.3;
numUnits = 200;
dropoutProb2 = 0.6;
layers = [ ...
    sequenceInputLayer(size(sequencesTrain{1},1))
    dropoutLayer(dropoutProb1)
    bilstmLayer(numUnits,"OutputMode","last")
    dropoutLayer(dropoutProb2)
    fullyConnectedLayer(numel(categories(emptyEmotions)))
    softmaxLayer
    classificationLayer];
```

Define training options using `trainingOptions`.

```
miniBatchSize = 512;
initialLearnRate = 0.005;
learnRateDropPeriod = 2;
maxEpochs = 3;
options = trainingOptions("adam", ...
    "MiniBatchSize",miniBatchSize, ...
    "InitialLearnRate",initialLearnRate, ...
    "LearnRateDropPeriod",learnRateDropPeriod, ...
    "LearnRateSchedule","piecewise", ...
    "MaxEpochs",maxEpochs, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "Plots","Training-Progress");
```

Train the network using `trainNetwork`.

```
net = trainNetwork(sequencesTrain,labelsTrain,layers,options);
```

To save the network, configured `audioFeatureExtractor`, and normalization factors, set `saveSERSystem` to `true`.

```
saveSERSystem = false;
if saveSERSystem
    normalizers.Mean = M;
    normalizers.StandardDeviation = S;
    save('network_Audio_SER.mat','net','afe','normalizers')
end
```

Training for System Validation

To provide an accurate assessment of the model you created in this example, train and validate using leave-one-speaker-out (LOSO) k-fold cross validation. In this method, you train using $k - 1$ speakers and then validate on the left-out speaker. You repeat this procedure k times. The final validation accuracy is the average of the k folds.

Create a variable that contains the speaker IDs. Determine the number of folds: 1 for each speaker. The database contains utterances from 10 unique speakers. Use `summary` to display the speaker IDs (left column) and the number of utterances they contribute to the database (right column).

```
speaker = ads.Labels.Speaker;
numFolds = numel(speaker);
summary(speaker)
```

```
     03      49
     08      58
     09      43
     10      38
     11      55
     12      35
     13      61
     14      69
     15      56
     16      71
```

The helper function HelperTrainAndValidateNetwork on page 14-0 performs the steps outlined above for all 10 folds and returns the true and predicted labels for each fold. Call `HelperTrainAndValidateNetwork` with the `audioDatastore`, the augmented `audioDatastore`, and the `audioFeatureExtractor`.

```
[labelsTrue,labelsPred] = HelperTrainAndValidateNetwork(ads,adsAug,afe);
```

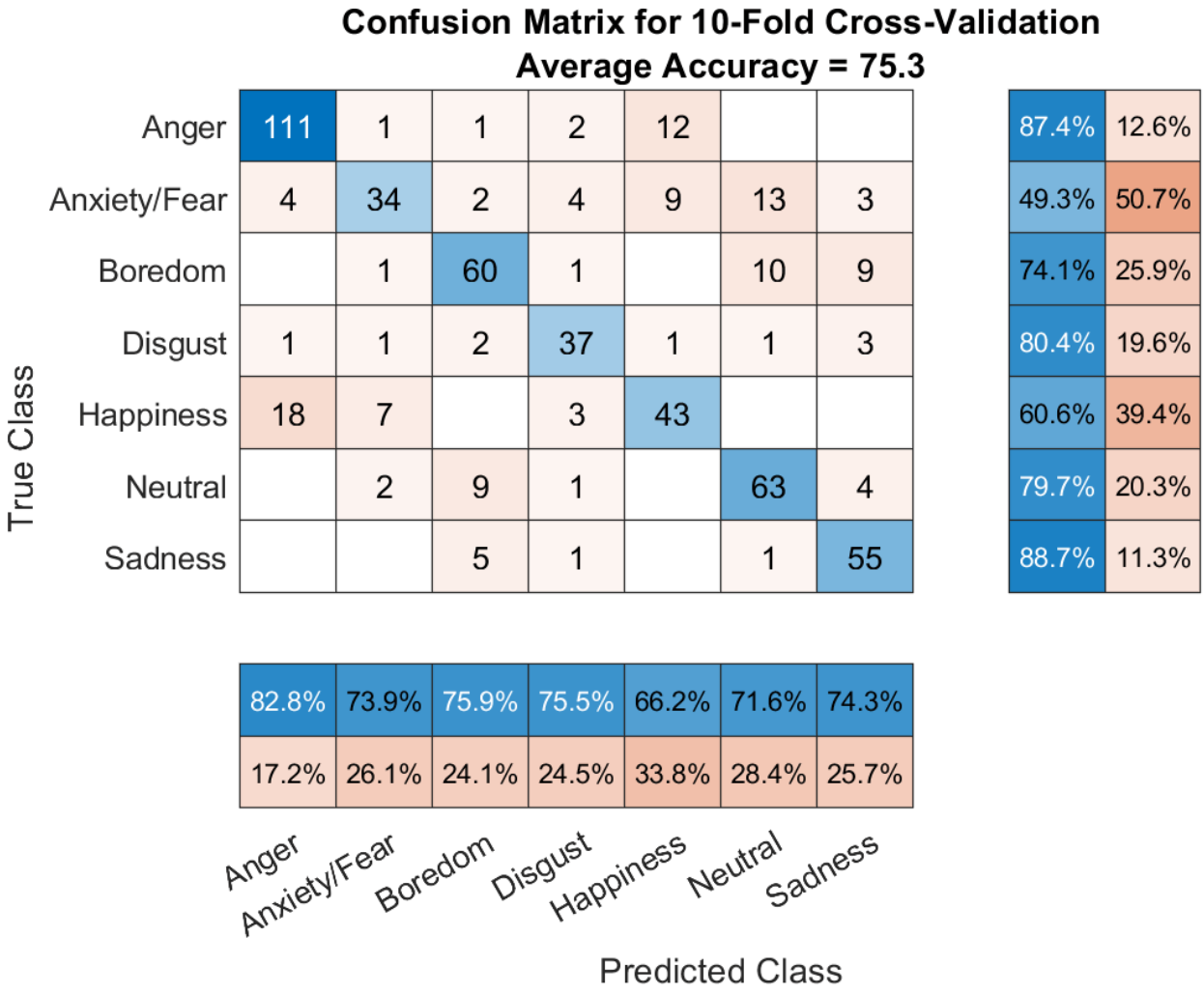
Print the accuracy per fold and plot the 10-fold confusion chart.

```
for ii = 1:numel(labelsTrue)
    foldAcc = mean(labelsTrue{ii}==labelsPred{ii})*100;
    fprintf('Fold %1.0f, Accuracy = %0.1f\n',ii,foldAcc);
end
```

```
Fold 1, Accuracy = 73.5
Fold 2, Accuracy = 77.6
Fold 3, Accuracy = 74.4
Fold 4, Accuracy = 68.4
Fold 5, Accuracy = 76.4
Fold 6, Accuracy = 80.0
Fold 7, Accuracy = 73.8
```

```
Fold 8, Accuracy = 87.0
Fold 9, Accuracy = 69.6
Fold 10, Accuracy = 70.4
```

```
labelsTrueMat = cat(1,labelsTrue{:});
labelsPredMat = cat(1,labelsPred{:});
figure
cm = confusionchart(labelsTrueMat,labelsPredMat);
valAccuracy = mean(labelsTrueMat==labelsPredMat)*100;
cm.Title = sprintf('Confusion Matrix for 10-Fold Cross-Validation\nAverage Accuracy = %0.1f',valA
sortClasses(cm,categories(emptyEmotions))
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



Supporting Functions

Convert Array of Feature Vectors to Sequences

```
function [sequences,sequencePerFile] = HelperFeatureVector2Sequence(features,featureVectorsPerSe
    % Copyright 2019 MathWorks, Inc.
```

```
    if featureVectorsPerSequence <= featureVectorOverlap
        error('The number of overlapping feature vectors must be less than the number of feature
    end

    if ~iscell(features)
        features = {features};
    end
    hopLength = featureVectorsPerSequence - featureVectorOverlap;
    idx1 = 1;
    sequences = {};
    sequencePerFile = cell(numel(features),1);
    for ii = 1:numel(features)
        sequencePerFile{ii} = floor((size(features{ii},2) - featureVectorsPerSequence)/hopLength)
        idx2 = 1;
        for j = 1:sequencePerFile{ii}
            sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1); %#ok<A
            idx1 = idx1 + 1;
            idx2 = idx2 + hopLength;
        end
    end
end
```

Train and Validate Network

```
function [trueLabelsCrossFold,predictedLabelsCrossFold] = HelperTrainAndValidateNetwork(varargin)
    % Copyright 2019 The MathWorks, Inc.
    if nargin == 3
        ads = varargin{1};
        augads = varargin{2};
        extractor = varargin{3};
    elseif nargin == 2
        ads = varargin{1};
        augads = varargin{1};
        extractor = varargin{2};
    end
    speaker = categories(ads.Labels.Speaker);
    numFolds = numel(speaker);
    emptyEmotions = (ads.Labels.Emotion);
    emptyEmotions(:) = [];

    % Loop over each fold.
    trueLabelsCrossFold = {};
    predictedLabelsCrossFold = {};

    for i = 1:numFolds

        % 1. Divide the audio datastore into training and validation sets.
        % Convert the data to tall arrays.
        idxTrain           = augads.Labels.Speaker~=speaker(i);
        augadsTrain        = subset(augads,idxTrain);
        augadsTrain.Labels = augadsTrain.Labels.Emotion;
        tallTrain          = tall(augadsTrain);
        idxValidation        = ads.Labels.Speaker==speaker(i);
        adsValidation        = subset(ads,idxValidation);
        adsValidation.Labels = adsValidation.Labels.Emotion;
        tallValidation       = tall(adsValidation);

        % 2. Extract features from the training set. Reorient the features
```

```
% so that time is along rows to be compatible with
% sequenceInputLayer.
tallTrain         = cellfun(@(x)x/max(abs(x),[],'all'),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)extract(extractor,x),tallTrain,"UniformOutput",false);
tallFeaturesTrain = cellfun(@(x)x',tallFeaturesTrain,"UniformOutput",false);  %#ok<NASGU>
[~,featuresTrain] = evalc('gather(tallFeaturesTrain)'); % Use evalc to suppress command-
tallValidation         = cellfun(@(x)x/max(abs(x),[],'all'),tallValidation,"UniformOutput
tallFeaturesValidation = cellfun(@(x)extract(extractor,x),tallValidation,"UniformOutput",
tallFeaturesValidation = cellfun(@(x)x',tallFeaturesValidation,"UniformOutput",false); %#
[~,featuresValidation] = evalc('gather(tallFeaturesValidation)'); % Use evalc to suppress

% 3. Use the training set to determine the mean and standard
% deviation of each feature. Normalize the training and validation
% sets.
allFeatures = cat(2,featuresTrain{:});
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
featuresTrain = cellfun(@(x)(x-M)./S,featuresTrain,'UniformOutput',false);
for ii = 1:numel(featuresTrain)
    idx = find(isnan(featuresTrain{ii}));
    if ~isempty(idx)
        featuresTrain{ii}(idx) = 0;
    end
end
featuresValidation = cellfun(@(x)(x-M)./S,featuresValidation,'UniformOutput',false);
for ii = 1:numel(featuresValidation)
    idx = find(isnan(featuresValidation{ii}));
    if ~isempty(idx)
        featuresValidation{ii}(idx) = 0;
    end
end

% 4. Buffer the sequences so that each sequence consists of twenty
% feature vectors with overlaps of 10 feature vectors.
featureVectorsPerSequence = 20;
featureVectorOverlap = 10;
[sequencesTrain,sequencePerFileTrain] = HelperFeatureVector2Sequence(featuresTrain,featur
[sequencesValidation,sequencePerFileValidation] = HelperFeatureVector2Sequence(featuresVa

% 5. Replicate the labels of the train and validation sets so that
% they are in one-to-one correspondence with the sequences.
labelsTrain = [emptyEmotions;augadsTrain.Labels];
labelsTrain = labelsTrain(:);
labelsTrain = repelem(labelsTrain,[sequencePerFileTrain{:}]);

% 6. Define a BiLSTM network.
dropoutProb1 = 0.3;
numUnits     = 200;
dropoutProb2 = 0.6;
layers = [ ...
    sequenceInputLayer(size(sequencesTrain{1},1))
    dropoutLayer(dropoutProb1)
    bilstmLayer(numUnits,"OutputMode","last")
    dropoutLayer(dropoutProb2)
    fullyConnectedLayer(numel(categories(emptyEmotions)))
    softmaxLayer
    classificationLayer];
```

```
        % 7. Define training options.
        miniBatchSize       = 512;
        initialLearnRate    = 0.005;
        learnRateDropPeriod = 2;
        maxEpochs           = 3;
        options = trainingOptions("adam", ...
            "MiniBatchSize",miniBatchSize, ...
            "InitialLearnRate",initialLearnRate, ...
            "LearnRateDropPeriod",learnRateDropPeriod, ...
            "LearnRateSchedule","piecewise", ...
            "MaxEpochs",maxEpochs, ...
            "Shuffle","every-epoch", ...
            "Verbose",false);

        % 8. Train the network.
        net = trainNetwork(sequencesTrain,labelsTrain,layers,options);

        % 9. Evaluate the network. Call classify to get the predicted labels
        % for each sequence. Get the mode of the predicted labels of each
        % sequence to get the predicted labels of each file.
        predictedLabelsPerSequence = classify(net,sequencesValidation);
        trueLabels = categorical(adsValidation.Labels);
        predictedLabels = trueLabels;
        idx1 = 1;
        for ii = 1:numel(trueLabels)
            predictedLabels(ii,:) = mode(predictedLabelsPerSequence(idx1:idx1 + sequencePerFileVa
            idx1 = idx1 + sequencePerFileValidation{ii};
        end
        trueLabelsCrossFold{i} = trueLabels; %#ok<AGROW>
        predictedLabelsCrossFold{i} = predictedLabels; %#ok<AGROW>
    end
end
```

References

[1] Burkhardt, F., A. Paeschke, M. Rolfes, W.F. Sendlmeier, and B. Weiss, "A Database of German Emotional Speech." In *Proceedings Interspeech 2005*. Lisbon, Portugal: International Speech Communication Association, 2005.

See Also

`bilstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Spoken Digit Recognition with Wavelet Scattering and Deep Learning

This example shows how to classify spoken digits using both machine and deep learning techniques. In the example, you perform classification using wavelet time scattering with a support vector machine (SVM) and with a long short-term memory (LSTM) network. You also apply Bayesian optimization to determine suitable hyperparameters to improve the accuracy of the LSTM network. In addition, the example illustrates an approach using a deep convolutional neural network (CNN) and mel-frequency spectrograms.

Data

Clone or download the Free Spoken Digit Dataset (FSDD), available at https://github.com/Jakobovski/free-spoken-digit-dataset. FSDD is an open data set, which means that it can grow over time. This example uses the version committed on January 29, 2019, which consists of 2000 recordings in English of the digits 0 through 9 obtained from four speakers. In this version, two of the speakers are native speakers of American English, one speaker is a nonnative speaker of English with a Belgian French accent, and one speaker is a nonnative speaker of English with a German accent. The data is sampled at 8000 Hz.

Use `audioDatastore` to manage data access and ensure the random division of the recordings into training and test sets. Set the `location` property to the location of the FSDD recordings folder on your computer, for example:

```
pathToRecordingsFolder = fullfile(tempdir,'free-spoken-digit-dataset-master','recordings');
location = pathToRecordingsFolder;
```

Point `audioDatastore` to that location.

```
ads = audioDatastore(location);
```

The helper function `helpergenLabels` creates a categorical array of labels from the FSDD files. The source code for `helpergenLabels` is listed in the appendix. List the classes and the number of examples in each class.

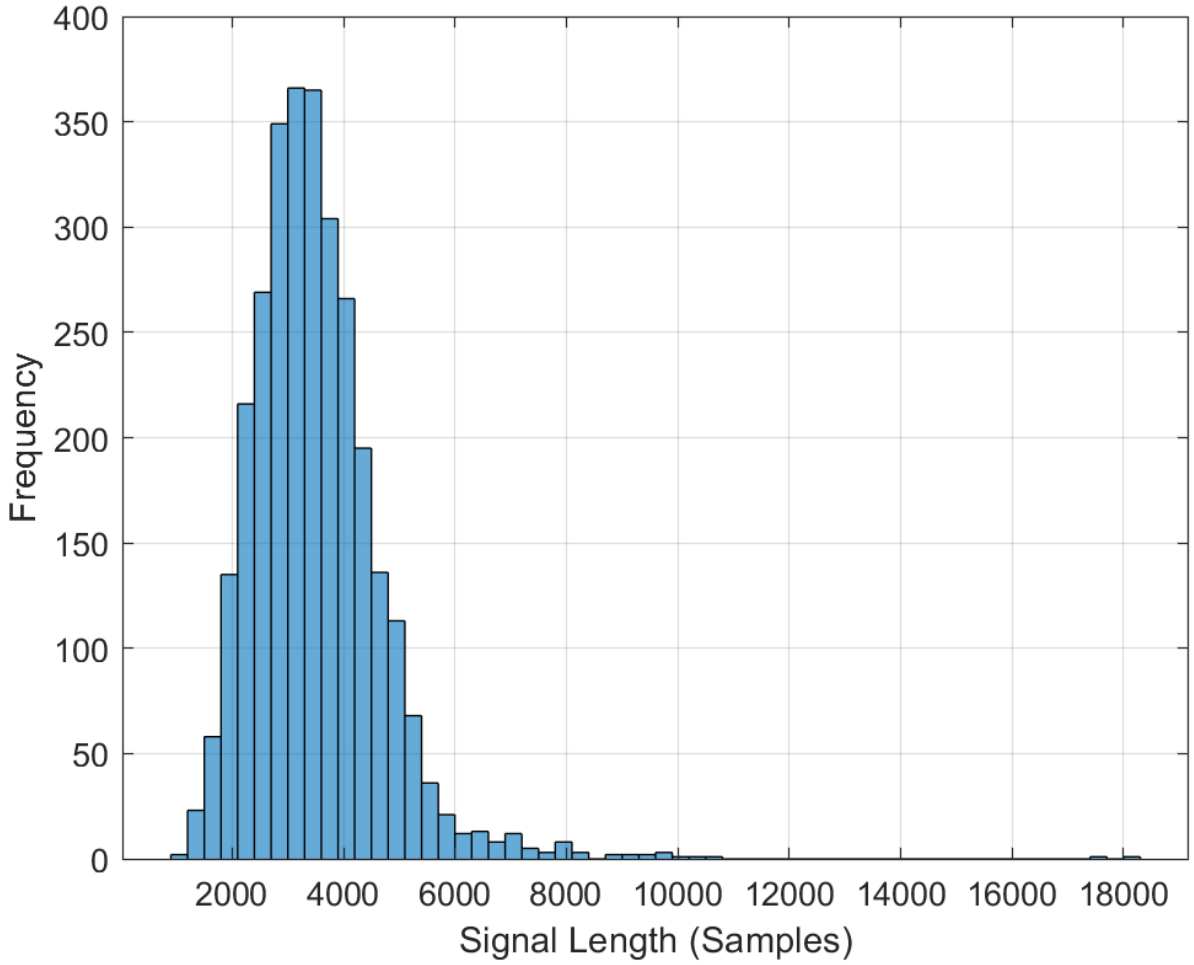
```
ads.Labels = helpergenLabels(ads);
summary(ads.Labels)
```

```
     0      300
     1      300
     2      300
     3      300
     4      300
     5      300
     6      300
     7      300
     8      300
     9      300
```

The FSDD data set consists of 10 balanced classes with 200 recordings each. The recordings in the FSDD are not of equal duration. The FSDD is not prohibitively large, so read through the FSDD files and construct a histogram of the signal lengths.

```
LenSig = zeros(numel(ads.Files),1);
nr = 1;
```

```
while hasdata(ads)
    digit = read(ads);
    LenSig(nr) = numel(digit);
    nr = nr+1;
end
reset(ads)
histogram(LenSig)
grid on
xlabel('Signal Length (Samples)')
ylabel('Frequency')
```



The histogram shows that the distribution of recording lengths is positively skewed. For classification, this example uses a common signal length of 8192 samples, a conservative value that ensures that truncating longer recordings does not cut off speech content. If the signal is greater than 8192 samples (1.024 seconds) in length, the recording is truncated to 8192 samples. If the signal is less than 8192 samples in length, the signal is prepadded and postpadded symmetrically with zeros out to a length of 8192 samples.

Wavelet Time Scattering

Use `waveletScattering` (Wavelet Toolbox) to create a wavelet time scattering framework using an invariant scale of 0.22 seconds. In this example, you create feature vectors by averaging the scattering transform over all time samples. To have a sufficient number of scattering coefficients per time window to average, set `OversamplingFactor` to 2 to produce a four-fold increase in the number of scattering coefficients for each path with respect to the critically downsampled value.

```
sf = waveletScattering('SignalLength',8192,'InvarianceScale',0.22,...
    'SamplingFrequency',8000,'OversamplingFactor',2);
```

Split the FSDD into training and test sets. Allocate 80% of the data to the training set and retain 20% for the test set. The training data is for training the classifier based on the scattering transform. The test data is for validating the model.

```
rng default;
ads = shuffle(ads);
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
countEachLabel(adsTrain)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       240
      1       240
      2       240
      3       240
      4       240
      5       240
      6       240
      7       240
      8       240
      9       240
```

```
countEachLabel(adsTest)
```

```
ans=10×2 table
    Label    Count
    _____    _____

      0       60
      1       60
      2       60
      3       60
      4       60
      5       60
      6       60
      7       60
      8       60
      9       60
```

The helper function `helperReadSPData` truncates or pads the data to a length of 8192 and normalizes each recording by its maximum value. The source code for `helperReadSPData` is listed in the appendix. Create an 8192-by-1600 matrix where each column is a spoken-digit recording.

```
Xtrain = [];
scatds_Train = transform(adsTrain,@(x)helperReadSPData(x));
while hasdata(scatds_Train)
    smat = read(scatds_Train);
    Xtrain = cat(2,Xtrain,smat);

end
```

Repeat the process for the test set. The resulting matrix is 8192-by-400.

```
Xtest = [];
scatds_Test = transform(adsTest,@(x)helperReadSPData(x));
while hasdata(scatds_Test)
    smat = read(scatds_Test);
    Xtest = cat(2,Xtest,smat);

end
```

Apply the wavelet scattering transform to the training and test sets.

```
Strain = sf.featureMatrix(Xtrain);
Stest = sf.featureMatrix(Xtest);
```

Obtain the mean scattering features for the training and test sets. Exclude the zeroth-order scattering coefficients.

```
TrainFeatures = Strain(2:end,:,:);
TrainFeatures = squeeze(mean(TrainFeatures,2))';
TestFeatures = Stest(2:end,:,:);
TestFeatures = squeeze(mean(TestFeatures,2))';
```

SVM Classifier

Now that the data has been reduced to a feature vector for each recording, the next step is to use these features for classifying the recordings. Create an SVM learner template with a quadratic polynomial kernel. Fit the SVM to the training data.

```
template = templateSVM(...
    'KernelFunction', 'polynomial', ...
    'PolynomialOrder', 2, ...
    'KernelScale', 'auto', ...
    'BoxConstraint', 1, ...
    'Standardize', true);
classificationSVM = fitcecoc(...
    TrainFeatures, ...
    adsTrain.Labels, ...
    'Learners', template, ...
    'Coding', 'onevsone', ...
    'ClassNames', categorical({'0'; '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'}));
```

Use k-fold cross-validation to predict the generalization accuracy of the model based on the training data. Split the training set into five groups.

```
partitionedModel = crossval(classificationSVM, 'KFold', 5);
[validationPredictions, validationScores] = kfoldPredict(partitionedModel);
validationAccuracy = (1 - kfoldLoss(partitionedModel, 'LossFun', 'ClassifError'))*100
```

```
validationAccuracy = 97.4167
```

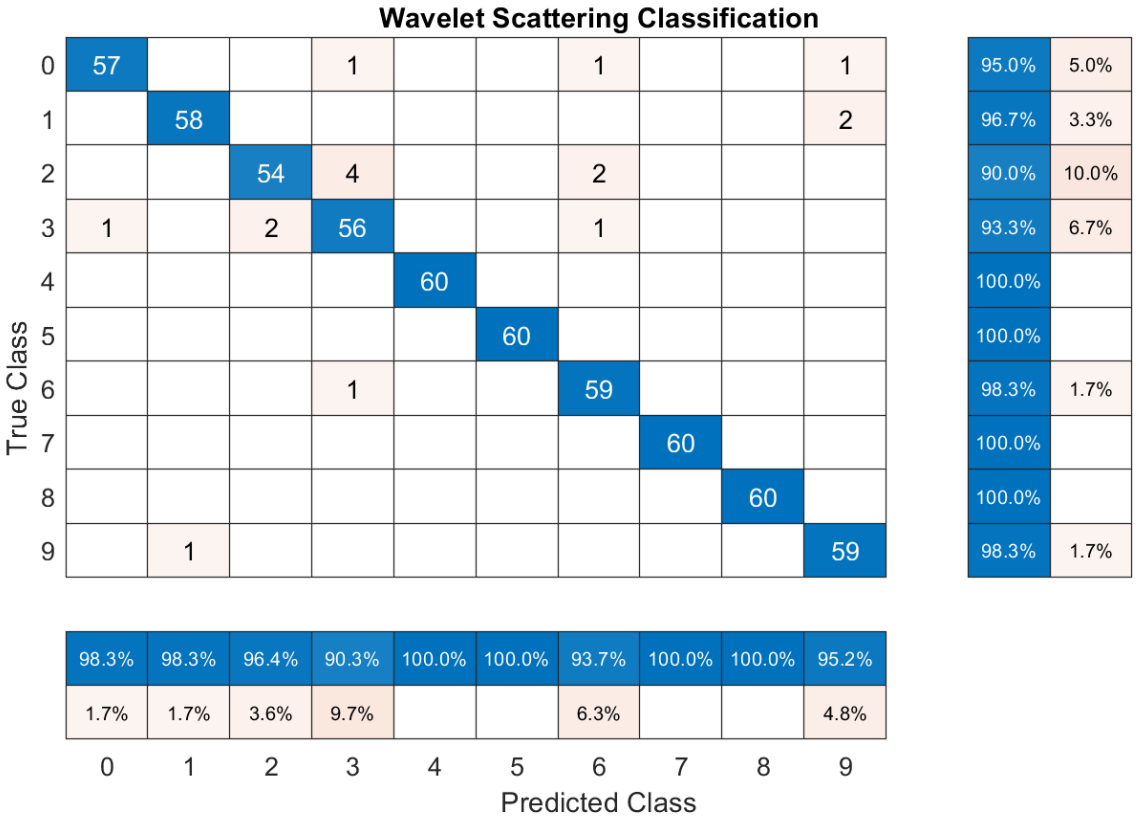
The estimated generalization accuracy is approximately 97%. Use the trained SVM to predict the spoken-digit classes in the test set.

```
predLabels = predict(classificationSVM,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

```
testAccuracy = 97.1667
```

Summarize the performance of the model on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values for each class. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccscat = confusionchart(adsTest.Labels,predLabels);
ccscat.Title = 'Wavelet Scattering Classification';
ccscat.ColumnSummary = 'column-normalized';
ccscat.RowSummary = 'row-normalized';
```



The scattering transform coupled with a SVM classifier classifies the spoken digits in the test set with an accuracy of 98% (or an error rate of 2%).

Long Short-Term Memory (LSTM) Networks

An LSTM network is a type of recurrent neural network (RNN). RNNs are neural networks that are specialized for working with sequential or temporal data such as speech data. Because the wavelet

scattering coefficients are sequences, they can be used as inputs to an LSTM. By using scattering features as opposed to the raw data, you can reduce the variability that your network needs to learn.

Modify the training and testing scattering features to be used with the LSTM network. Exclude the zeroth-order scattering coefficients and convert the features to cell arrays.

```
TrainFeatures = Strain(2:end,:,:);
TrainFeatures = squeeze(num2cell(TrainFeatures,[1 2]));
TestFeatures = Stest(2:end,:,:);
TestFeatures = squeeze(num2cell(TestFeatures, [1 2]));
```

Construct a simple LSTM network with 512 hidden layers.

```
[inputSize, ~] = size(TrainFeatures{1});
YTrain = adsTrain.Labels;

numHiddenUnits = 512;
numClasses = numel(unique(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Set the hyperparameters. Use Adam optimization and a mini-batch size of 50. Set the maximum number of epochs to 300. Use a learning rate of 1e-4. You can turn off the training progress plot if you do not want to track the progress using plots. The training uses a GPU by default if one is available. Otherwise, it uses a CPU. For more information, see `trainingOptions`.

```
maxEpochs = 300;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'InitialLearnRate',0.0001,...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','shortest', ...
    'Shuffle','every-epoch',...
    'Verbose', false, ...
    'Plots','training-progress');
```

Train the network.

```
net = trainNetwork(TrainFeatures,YTrain,layers,options);

predLabels = classify(net,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

```
testAccuracy = 96.3333
```

Bayesian Optimization

Determining suitable hyperparameter settings is often one of the most difficult parts of training a deep network. To mitigate this, you can use Bayesian optimization. In this example, you optimize the number of hidden layers and the initial learning rate by using Bayesian techniques. Create a new

directory to store the MAT-files containing information about hyperparameter settings and the network along with the corresponding error rates.

```
YTrain = adsTrain.Labels;
YTest = adsTest.Labels;

if ~exist("results/",'dir')
    mkdir results
end
```

Initialize the variables to be optimized and their value ranges. Because the number of hidden layers must be an integer, set `'type'` to `'integer'`.

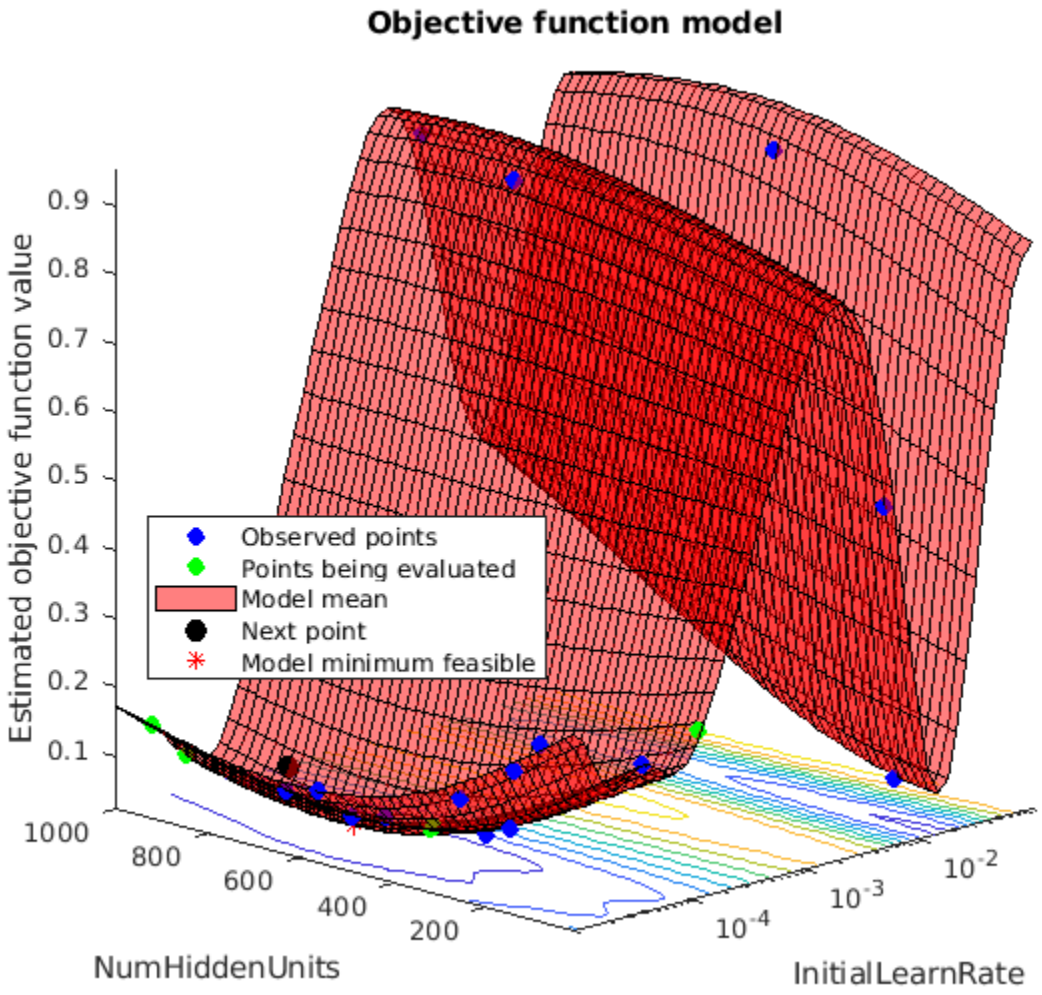
```
optVars = [
    optimizableVariable('InitialLearnRate',[1e-5, 1e-1],'Transform','log')
    optimizableVariable('NumHiddenUnits',[10, 1000],'Type','integer')
    ];
```

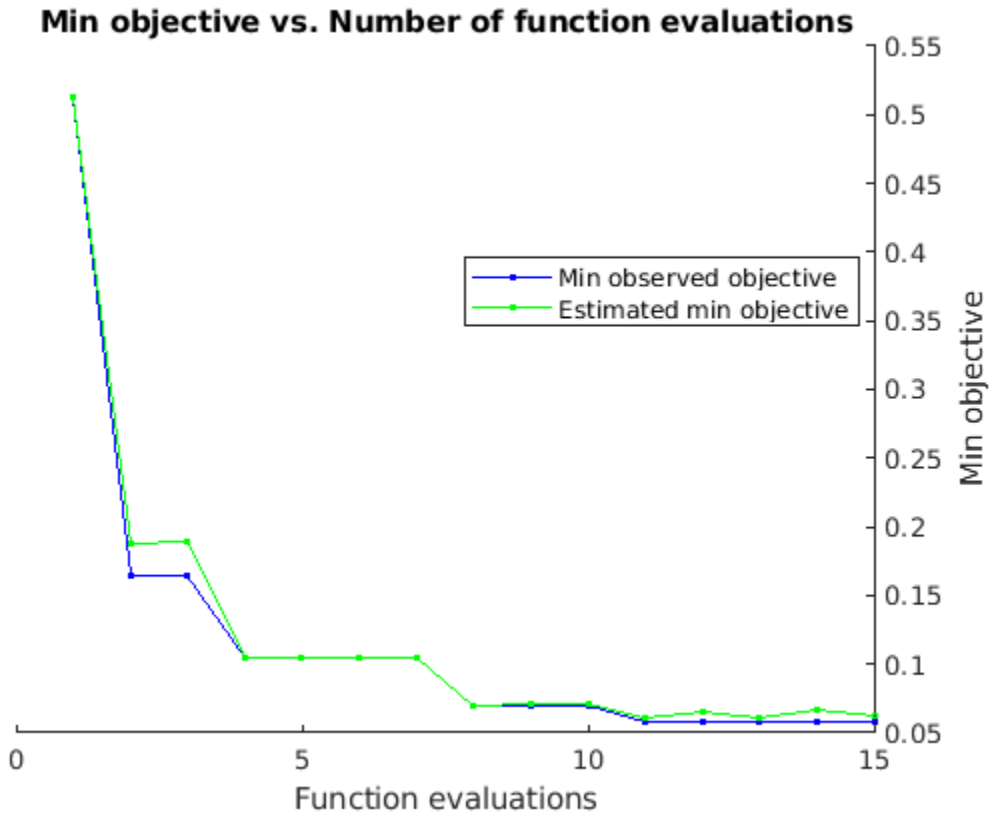
Bayesian optimization is computationally intensive and can take several hours to finish. For the purposes of this example, set `optimizeCondition` to `false` to download and use predetermined optimized hyperparameter settings. If you set `optimizeCondition` to `true`, the objective function `helperBayesOptLSTM` is minimized using Bayesian optimization. The objective function, listed in the appendix, is the error rate of the network given specific hyperparameter settings. The loaded settings are for the objective function minimum of 0.02 (2% error rate).

```
ObjFcn = helperBayesOptLSTM(TrainFeatures,YTrain,TestFeatures,YTest);

optimizeCondition = false;
if optimizeCondition
    BayesObject = bayesopt(ObjFcn,optVars,...
            'MaxObjectiveEvaluations',15,...
            'IsObjectiveDeterministic',false,...
            'UseParallel',true);
else
    url = 'http://ssd.mathworks.com/supportfiles/audio/SpokenDigitRecognition.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'SpokenDigitRecognition');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 12 MB) ...')
        unzip(url,downloadNetFolder)
    end
    load(fullfile(netFolder,'0.02.mat'));
end
```

```
Downloading pretrained network (1 file - 12 MB) ...
```

If you perform Bayesian optimization, figures similar to the following are generated to track the objective function values with the corresponding hyperparameter values and the number of iterations. You can increase the number of Bayesian optimization iterations to ensure that the global minimum of the objective function is reached.


Objective function model
Estimated objective function value
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Observed points
Points being evaluated
Model mean
Next point
Model minimum feasible
1000
800
600
400
200
NumHiddenUnits
10^{-2}
10^{-3}
10^{-4}
InitialLearnRate



Use the optimized values for the number of hidden units and initial learning rate and retrain the network.

```
numHiddenUnits = 768;
numClasses = numel(unique(YTrain));

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

maxEpochs = 300;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'InitialLearnRate',2.198827960269379e-04,...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'SequenceLength','shortest', ...
    'Shuffle','every-epoch',...
    'Verbose', false, ...
    'Plots','training-progress');

net = trainNetwork(TrainFeatures,YTrain,layers,options);
```

```
predLabels = classify(net,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100

testAccuracy = 97.5000
```

As the plot shows, using Bayesian optimization yields an LSTM with a higher accuracy.

Deep Convolutional Network Using Mel-Frequency Spectrograms

As another approach to the task of spoken digit recognition, use a deep convolutional neural network (DCNN) based on mel-frequency spectrograms to classify the FSDD data set. Use the same signal truncation/padding procedure as in the scattering transform. Similarly, normalize each recording by dividing each signal sample by the maximum absolute value. For consistency, use the same training and test sets as for the scattering transform.

Set the parameters for the mel-frequency spectrograms. Use the same window, or frame, duration as in the scattering transform, 0.22 seconds. Set the hop between windows to 10 ms. Use 40 frequency bands.

```
segmentDuration = 8192*(1/8000);
frameDuration = 0.22;
hopDuration = 0.01;
numBands = 40;
```

Reset the training and test datastores.

```
reset(adsTrain);
reset(adsTest);
```

The helper function `helperspeechSpectrograms`, defined at the end of this example, uses `melSpectrogram` to obtain the mel-frequency spectrogram after standardizing the recording length and normalizing the amplitude. Use the logarithm of the mel-frequency spectrograms as the inputs to the DCNN. To avoid taking the logarithm of zero, add a small epsilon to each element.

```
epsil = 1e-6;
XTrain = helperspeechSpectrograms(adsTrain,segmentDuration,frameDuration,hopDuration,numBands);

Computing speech spectrograms...
Processed 500 files out of 2400
Processed 1000 files out of 2400
Processed 1500 files out of 2400
Processed 2000 files out of 2400
...done

XTrain = log10(XTrain + epsil);

XTest = helperspeechSpectrograms(adsTest,segmentDuration,frameDuration,hopDuration,numBands);

Computing speech spectrograms...
Processed 500 files out of 600
...done

XTest = log10(XTest + epsil);

YTrain = adsTrain.Labels;
YTest = adsTest.Labels;
```

Define DCNN Architecture

Construct a small DCNN as an array of layers. Use convolutional and batch normalization layers, and downsample the feature maps using max pooling layers. To reduce the possibility of the network memorizing specific features of the training data, add a small amount of dropout to the input to the last fully connected layer.

```
sz = size(XTrain);
specSize = sz(1:2);
imageSize = [specSize 1];

numClasses = numel(categories(YTrain));

dropoutProb = 0.2;
numF = 12;
layers = [
    imageInputLayer(imageSize)

    convolution2dLayer(5,numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(3,'Stride',2,'Padding','same')

    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer
    convolution2dLayer(3,4*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2)

    dropoutLayer(dropoutProb)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',categories(YTrain));
    ];
```

Set the hyperparameters to use in training the network. Use a mini-batch size of 50 and a learning rate of 1e-4. Specify Adam optimization. Because the amount of data in this example is relatively small, set the execution environment to `'cpu'` for reproducibility. You can also train the network on an available GPU by setting the execution environment to either `'gpu'` or `'auto'`. For more information, see `trainingOptions`.

```
miniBatchSize = 50;
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-4, ...
    'MaxEpochs',30, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ExecutionEnvironment','cpu');
```

Train the network.

```
trainedNet = trainNetwork(XTrain,YTrain,layers,options);
```

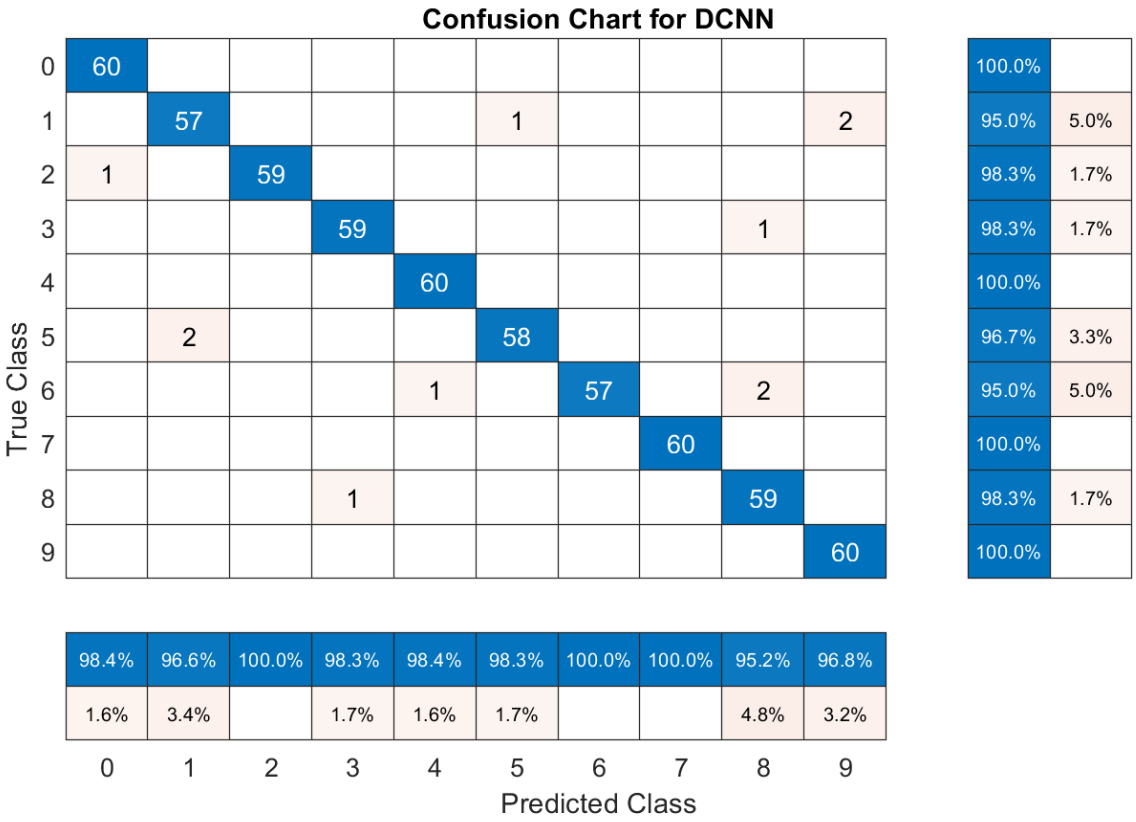
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest,'ExecutionEnvironment','CPU');
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 98.1667
```

Summarize the performance of the trained network on the test set with a confusion chart. Display the precision and recall for each class by using column and row summaries. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccDCNN = confusionchart(YTest,Ypredicted);
ccDCNN.Title = 'Confusion Chart for DCNN';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```



The DCNN using mel-frequency spectrograms as inputs classifies the spoken digits in the test set with an accuracy rate of approximately 98% as well.

Summary

This example shows how to use different machine and deep learning approaches for classifying spoken digits in the FSDD. The example illustrated wavelet scattering paired with both an SVM and a LSTM. Bayesian techniques were used to optimize LSTM hyperparameters. Finally, the example shows how to use a CNN with mel-frequency spectrograms.

The goal of the example is to demonstrate how to use MathWorks® tools to approach the problem in fundamentally different but complementary ways. All workflows use `audioDatastore` to manage flow of data from disk and ensure proper randomization.

All approaches used in this example performed equally well on the test set. This example is not intended as a direct comparison between the various approaches. For example, you can also use Bayesian optimization for hyperparameter selection in the CNN. An additional strategy that is useful in deep learning with small training sets like this version of the FSDD is to use data augmentation. How manipulations affect class is not always known, so data augmentation is not always feasible. However, for speech, established data augmentation strategies are available through `audioDataAugmenter`.

In the case of wavelet time scattering, there are also a number of modifications you can try. For example, you can change the invariant scale of the transform, vary the number of wavelet filters per filter bank, and try different classifiers.

Appendix: Helper Functions

```
function Labels = helpergenLabels(ads)
% This function is only for use in Wavelet Toolbox examples. It may be
% changed or removed in a future release.
tmp = cell(numel(ads.Files),1);
expression = "[0-9]+_";
for nf = 1:numel(ads.Files)
    idx = regexp(ads.Files{nf},expression);
    tmp{nf} = ads.Files{nf}(idx);
end
Labels = categorical(tmp);
end

function x = helperReadSPData(x)
% This function is only for use Wavelet Toolbox examples. It may change or
% be removed in a future release.

N = numel(x);
if N > 8192
    x = x(1:8192);
elseif N < 8192
    pad = 8192-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));

end

function x = helperBayesOptLSTM(X_train, Y_train, X_val, Y_val)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.
x = @valErrorFun;

    function [valError,cons, fileName] = valErrorFun(optVars)
        %% LSTM Architecture
        [inputSize,~] = size(X_train{1});
        numClasses = numel(unique(Y_train));

        layers = [ ...
            sequenceInputLayer(inputSize)
            bilstmLayer(optVars.NumHiddenUnits,'OutputMode','last') % Using number of hidden laye
            fullyConnectedLayer(numClasses)
            softmaxLayer
            classificationLayer];

        % Plots not displayed during training
        options = trainingOptions('adam', ...
            'InitialLearnRate',optVars.InitialLearnRate, ... % Using initial learning rate value
            'MaxEpochs',300, ...
            'MiniBatchSize',30, ...
            'SequenceLength','shortest', ...
            'Shuffle','never', ...
            'Verbose', false);
```

```
        %% Train the network
        net = trainNetwork(X_train, Y_train, layers, options);
        %% Training accuracy
        X_val_P = net.classify(X_val);
        accuracy_training  = sum(X_val_P == Y_val)./numel(Y_val);
        valError = 1 - accuracy_training;
        %% save results of network and options in a MAT file in the results folder along with the
        fileName = fullfile('results', num2str(valError) + ".mat");
        save(fileName,'net','valError','options')
        cons = [];
    end % end for inner function
end % end for outer function


function X = helperspeechSpectrograms(ads,segmentDuration,frameDuration,hopDuration,numBands)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.
%
% helperspeechSpectrograms(ads,segmentDuration,frameDuration,hopDuration,numBands)
% computes speech spectrograms for the files in the datastore ads.
% segmentDuration is the total duration of the speech clips (in seconds),
% frameDuration the duration of each spectrogram frame, hopDuration the
% time shift between each spectrogram frame, and numBands the number of
% frequency bands.
disp("Computing speech spectrograms...");

numHops = ceil((segmentDuration - frameDuration)/hopDuration);
numFiles = length(ads.Files);
X = zeros([numBands,numHops,1,numFiles],'single');

for i = 1:numFiles

    [x,info] = read(ads);
    x = normalizeAndResize(x);
    fs = info.SampleRate;
    frameLength = round(frameDuration*fs);
    hopLength = round(hopDuration*fs);

    spec = melSpectrogram(x,fs, ...
        'Window',hamming(frameLength,'periodic'), ...
        'OverlapLength',frameLength - hopLength, ...
        'FFTLength',2048, ...
        'NumBands',numBands, ...
        'FrequencyRange',[50,4000]);

    % If the spectrogram is less wide than numHops, then put spectrogram in
    % the middle of X.
    w = size(spec,2);
    left = floor((numHops-w)/2)+1;
    ind = left:left+w-1;
    X(:,ind,1,i) = spec;

    if mod(i,500) == 0
        disp("Processed " + i + " files out of " + numFiles)
    end

end
```

```
disp("...done");

end

%--------------------------------------------------------------------------
function x = normalizeAndResize(x)
% This function is only for use in the
% "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
% example. It may change or be removed in a future release.

N = numel(x);
if N > 8192
    x = x(1:8192);
elseif N < 8192
    pad = 8192-N;
    prepad = floor(pad/2);
    postpad = ceil(pad/2);
    x = [zeros(prepad,1) ; x ; zeros(postpad,1)];
end
x = x./max(abs(x));
end
```


See Also

`trainingOptions` | `trainNetwork`

More About

- “Deep Learning in MATLAB” on page 1-2

Cocktail Party Source Separation Using Deep Learning Networks

This example shows how to isolate a speech signal using a deep learning network.

Introduction

The cocktail party effect refers to the ability of the brain to focus on a single speaker while filtering out other voices and background noise. Humans perform very well at the cocktail party problem. This example shows how to use a deep learning network to separate individual speakers from a speech mix where one male and one female are speaking simultaneously.

Download Required Files

Before going into the example in detail, you will download a pre-trained network and 4 audio files.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/CocktailPartySourceSeparation.zip';

downloadNetFolder = tempdir;
netFolder = fullfile(downloadNetFolder,'CocktailPartySourceSeparation');

if ~exist(netFolder,'dir')
    disp('Downloading pretrained network and audio files (5 files - 24.5 MB) ...')
    unzip(url,downloadNetFolder)
end
```

Problem Summary

Load audio files containing male and female speech sampled at 4 kHz. Listen to the audio files individually for reference.

```
[mSpeech,Fs] = audioread(fullfile(netFolder,"MaleSpeech-16-4-mono-20secs.wav"));
sound(mSpeech,Fs)

[fSpeech] = audioread(fullfile(netFolder,"FemaleSpeech-16-4-mono-20secs.wav"));
sound(fSpeech,Fs)
```

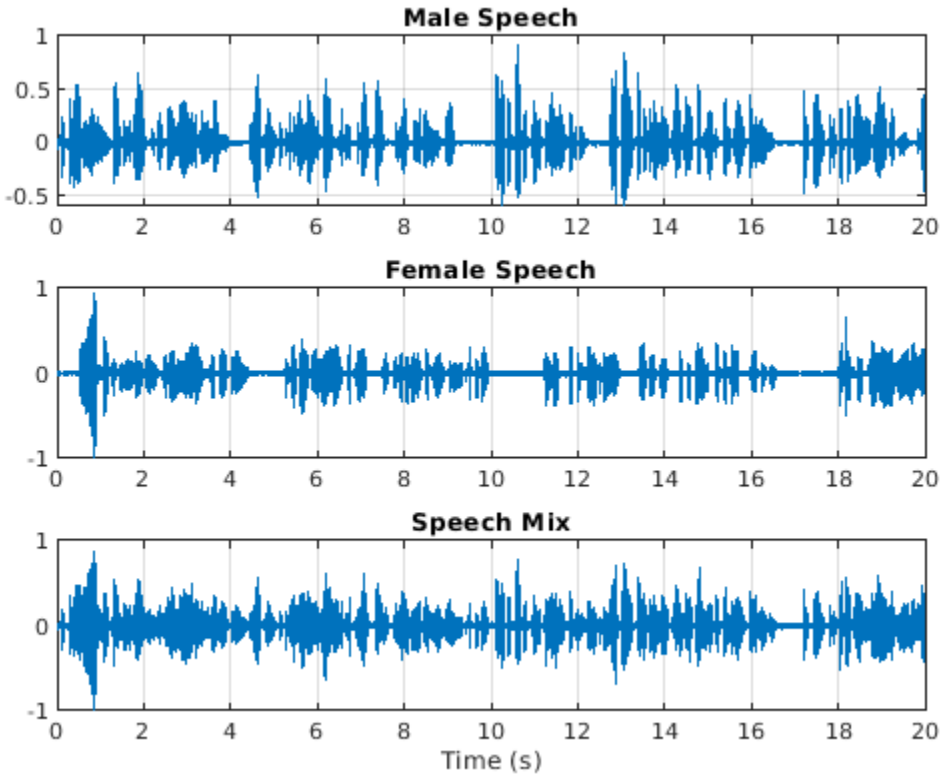
Combine the two speech sources. Ensure the sources have equal power in the mix. Normalize the mix so that its max amplitude is one.

```
mSpeech = mSpeech/norm(mSpeech);
fSpeech = fSpeech/norm(fSpeech);
ampAdj  = max(abs([mSpeech;fSpeech]));
mSpeech = mSpeech/ampAdj;
fSpeech = fSpeech/ampAdj;
mix     = mSpeech + fSpeech;
mix     = mix ./ max(abs(mix));
```

Visualize the original and mix signals. Listen to the mixed speech signal. This example shows a source separation scheme that extracts the male and female sources from the speech mix.

```
t = (0:numel(mix)-1)*(1/Fs);

figure(1)
subplot(3,1,1)
plot(t,mSpeech)
title("Male Speech")
```

```
grid on
subplot(3,1,2)
plot(t,fSpeech)
title("Female Speech")
grid on
subplot(3,1,3)
plot(t,mix)
title("Speech Mix")
xlabel("Time (s)")
grid on
```



Listen to the mix audio.

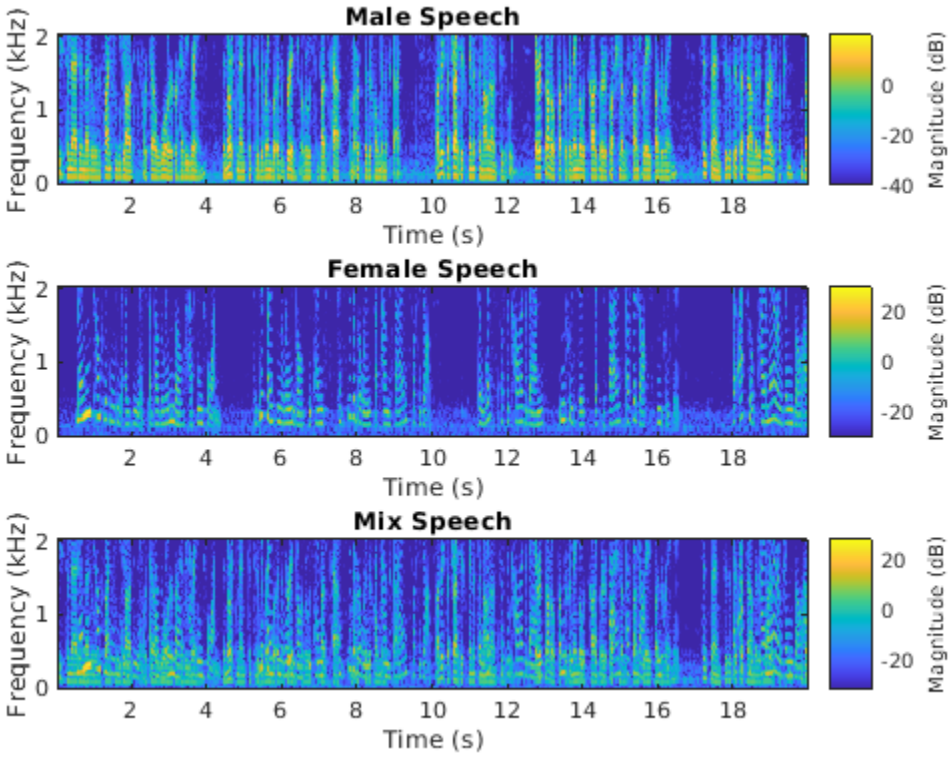
```
sound(mix,Fs)
```

Time-Frequency Representation

Use `stft` to visualize the time-frequency (TF) representation of the male, female, and mix speech signals. Use a Hann window of length 128, an FFT length of 128, and an overlap length of 96.

```
WindowLength  = 128;
FFTLength     = 128;
OverlapLength = 96;
win           = hann(WindowLength,"periodic");

figure(2)
subplot(3,1,1)
stft(mSpeech, Fs, 'Window', win,'OverlapLength',OverlapLength,...
```

```
     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Male Speech")
subplot(3,1,2)
stft(fSpeech, Fs, 'Window', win, 'OverlapLength', OverlapLength,...
     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Female Speech")
subplot(3,1,3)
stft(mix, Fs, 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Mix Speech")
```



Source Separation Using Ideal Time-Frequency Masks

The application of a TF mask has been shown to be an effective method for separating desired audio signals from competing sounds. A TF mask is a matrix of the same size as the underlying STFT. The mask is multiplied element-by-element with the underlying STFT to isolate the desired source. The TF mask can be binary or soft.

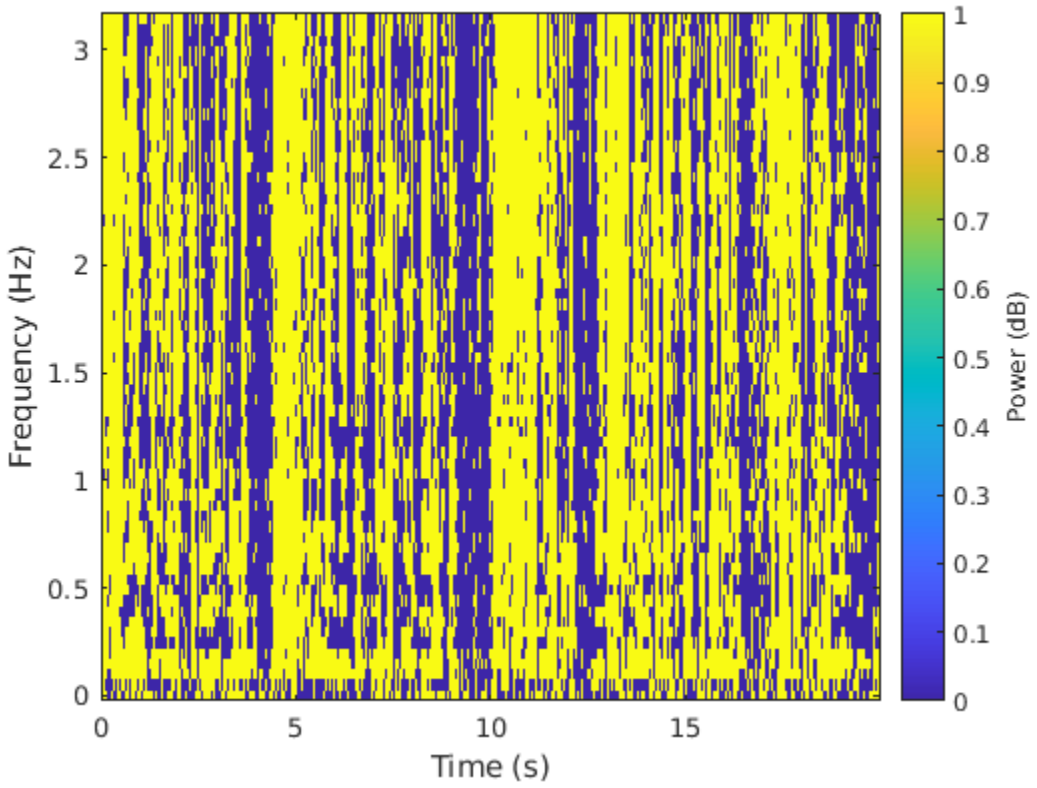
Source Separation Using Ideal Binary Masks

In an ideal binary mask, the mask cell values are either 0 or 1. If the power of the desired source is greater than the combined power of other sources at a particular TF cell, then that cell is set to 1. Otherwise, the cell is set to 0.

Compute the ideal binary mask for the male speaker and then visualize it.

```
P_M        = stft(mSpeech, 'Window', win, 'OverlapLength', OverlapLength,...
                  'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
```

```
P_F        = stft(fSpeech, 'Window', win, 'OverlapLength', OverlapLength,...
                  'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
[P_mix,F]  = stft(mix, 'Window', win, 'OverlapLength', OverlapLength,...
                   'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
binaryMask = abs(P_M) >= abs(P_F);

figure(3)
plotMask(binaryMask, WindowLength - OverlapLength, F, Fs)
```



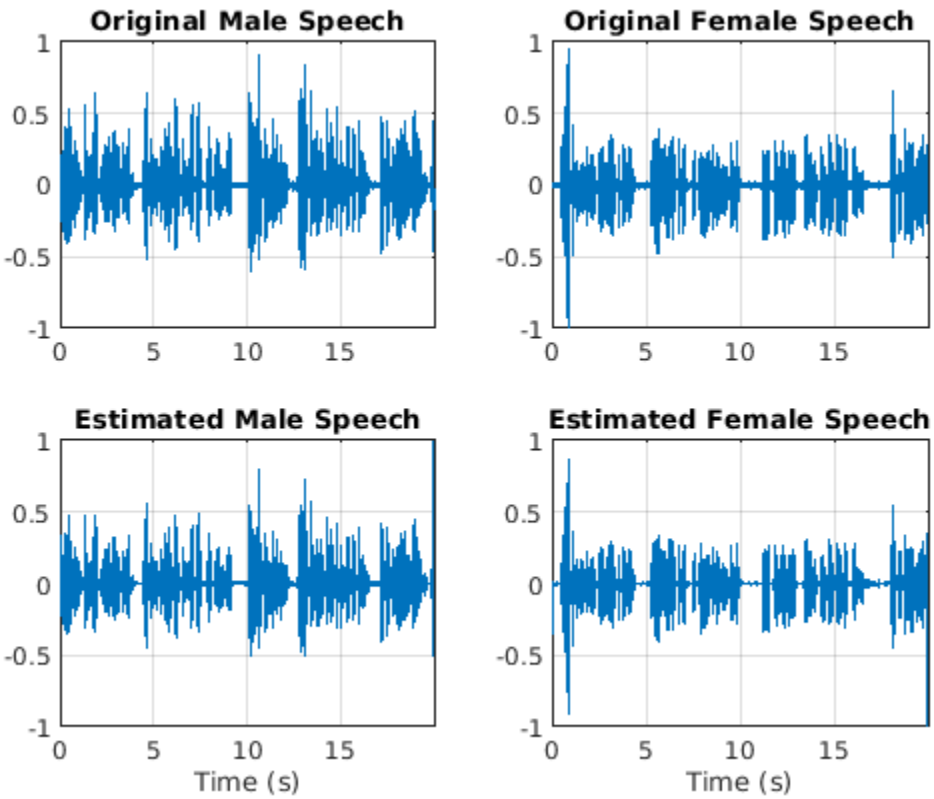
Estimate the male speech STFT by multiplying the mix STFT by the male speaker's binary mask. Estimate the female speech STFT by multiplying the mix STFT by the inverse of the male speaker's binary mask.

```
P_M_Hard = P_mix .* binaryMask;
P_F_Hard = P_mix .* (1-binaryMask);
```

Estimate the male and female audio signals using the inverse short-time FFT (ISTFT). Visualize the estimated and original signals. Listen to the estimated male and female speech signals.

```
mSpeech_Hard = istft(P_M_Hard , 'Window', win, 'OverlapLength', OverlapLength,...
                     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
fSpeech_Hard = istft(P_F_Hard , 'Window', win, 'OverlapLength', OverlapLength,...
                     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
figure(4)
subplot(2,2,1)
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
```

```
grid on

subplot(2,2,3)
plot(t,mSpeech_Hard)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Male Speech")
grid on

subplot(2,2,2)
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
title("Original Female Speech")
grid on

subplot(2,2,4)
plot(t,fSpeech_Hard)
axis([t(1) t(end) -1 1])
title("Estimated Female Speech")
xlabel("Time (s)")
grid on
```




```
sound(mSpeech_Hard,Fs)

sound(fSpeech_Hard,Fs)
```

Source Separation Using Ideal Soft Masks

In a soft mask, the TF mask cell value is equal to the ratio of the desired source power to the total mix power. TF cells have values in the range [0,1].

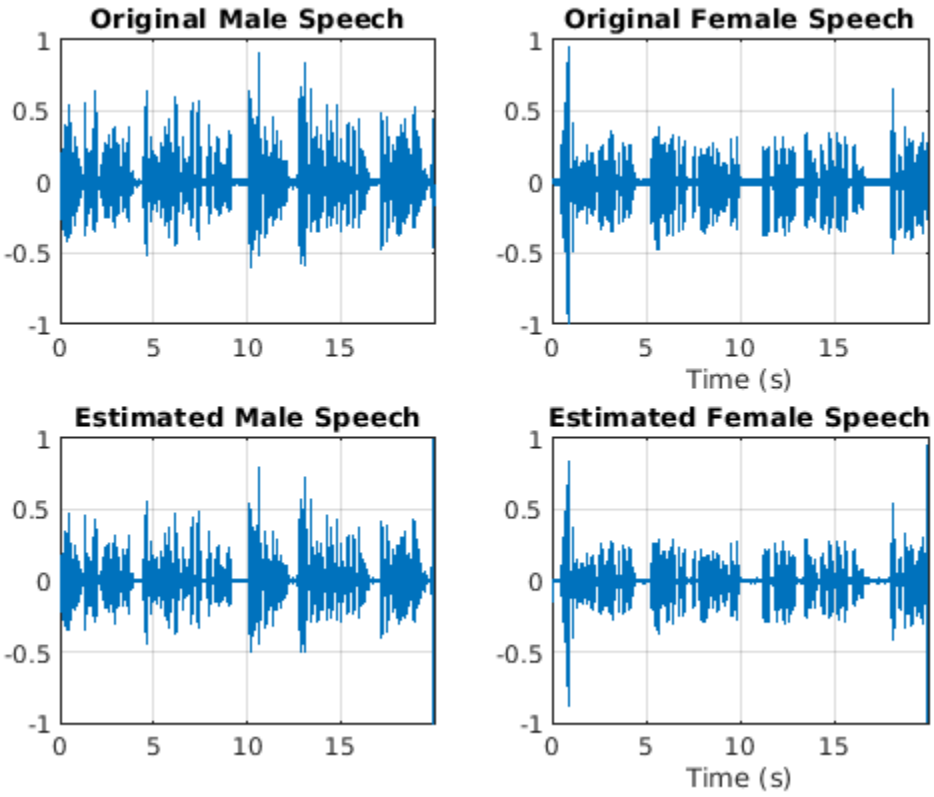
Compute the soft mask for the male speaker. Estimate the STFT of the male speaker by multiplying the mix STFT by the male speaker's soft mask. Estimate the STFT of the female speaker by multiplying the mix STFT by the female speaker's soft mask.

Estimate the male and female audio signals using the ISTFT.

```
softMask = abs(P_M) ./ (abs(P_F) + abs(P_M) + eps);

P_M_Soft = P_mix .* softMask;
P_F_Soft = P_mix .* (1-softMask);

mSpeech_Soft = istft(P_M_Soft, 'Window', win, 'OverlapLength', OverlapLength,...
                     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
fSpeech_Soft = istft(P_F_Soft, 'Window', win, 'OverlapLength', OverlapLength,...
                     'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
```

Visualize the estimated and original signals. Listen to the estimated male and female speech signals. Note that the results are very good because the mask is created with full knowledge of the separated male and female signals.

```
figure(5)
subplot(2,2,1)
plot(t,mSpeech)
axis([t(1) t(end) -1 1])
title("Original Male Speech")
grid on

subplot(2,2,3)
plot(t,mSpeech_Soft)
axis([t(1) t(end) -1 1])
title("Estimated Male Speech")
grid on

subplot(2,2,2)
plot(t,fSpeech)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Original Female Speech")
grid on

subplot(2,2,4)
plot(t,fSpeech_Soft)
axis([t(1) t(end) -1 1])
xlabel("Time (s)")
title("Estimated Female Speech")
grid on
```



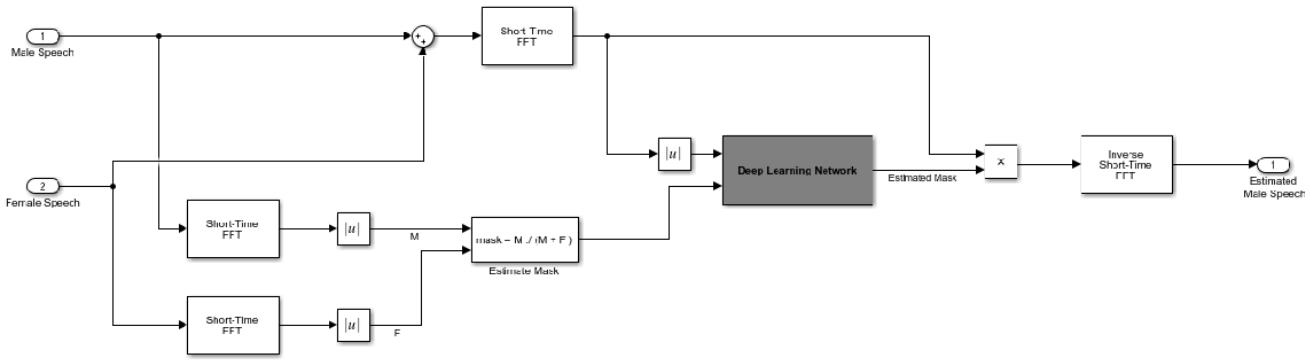

```
sound(mSpeech_Soft,Fs)
```

```
sound(fSpeech_Soft,Fs)
```

Mask Estimation Using Deep Learning

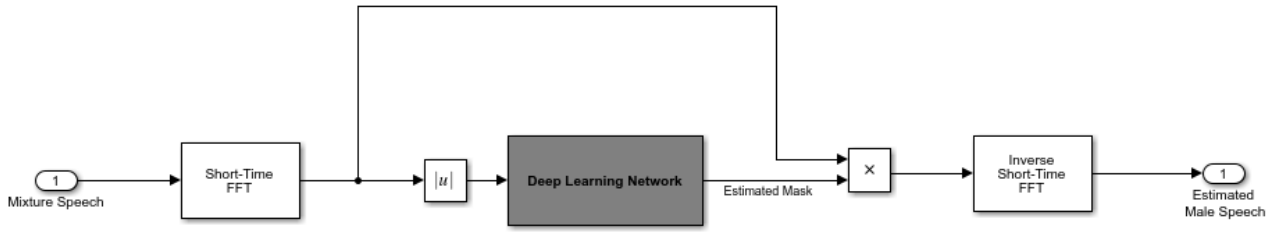
The goal of the deep learning network in this example is to estimate the ideal soft mask described above. The network estimates the mask corresponding to the male speaker. The female speaker mask is derived directly from the male mask.

The basic deep learning training scheme is shown below. The predictor is the magnitude spectra of the mixed (male + female) audio. The target is the ideal soft masks corresponding to the male speaker. The regression network uses the predictor input to minimize the mean square error between its output and the input target. At the output, the audio STFT is converted back to the time domain using the output magnitude spectrum and the phase of the mix signal.



You transform the audio to the frequency domain using the Short-Time Fourier transform (STFT), with a window length of 128 samples, an overlap of 127, and a Hann window. You reduce the size of the spectral vector to 65 by dropping the frequency samples corresponding to negative frequencies (because the time-domain speech signal is real, this does not lead to any information loss). The predictor input consists of 20 consecutive STFT vectors. The output is a 65-by-20 soft mask.

You use the trained network to estimate the male speech. The input to the trained network is the mixture (male + female) speech audio.



STFT Targets and Predictors

This section illustrates how to generate the target and predictor signals from the training dataset.

Read in training signals consisting of around 400 seconds of speech from male and female speakers, respectively, sampled at 4 kHz. The low sample rate is used to speed up training. Trim the training signals so that they are the same length.

```
maleTrainingAudioFile   = "MaleSpeech-16-4-mono-405secs.wav";
femaleTrainingAudioFile = "FemaleSpeech-16-4-mono-405secs.wav";

maleSpeechTrain   = audioread(fullfile(netFolder,maleTrainingAudioFile));
femaleSpeechTrain = audioread(fullfile(netFolder,femaleTrainingAudioFile));

L = min(length(maleSpeechTrain),length(femaleSpeechTrain));
maleSpeechTrain   = maleSpeechTrain(1:L);
femaleSpeechTrain = femaleSpeechTrain(1:L);
```

Read in validation signals consisting of around 20 seconds of speech from male and female speakers, respectively, sampled at 4 kHz. Trim the validation signals so that they are the same length

```
maleValidationAudioFile   = "MaleSpeech-16-4-mono-20secs.wav";
femaleValidationAudioFile = "FemaleSpeech-16-4-mono-20secs.wav";

maleSpeechValidate   = audioread(fullfile(netFolder,maleValidationAudioFile));
femaleSpeechValidate = audioread(fullfile(netFolder,femaleValidationAudioFile));

L = min(length(maleSpeechValidate),length(femaleSpeechValidate));
maleSpeechValidate   = maleSpeechValidate(1:L);
femaleSpeechValidate = femaleSpeechValidate(1:L);
```

Scale the training signals to the same power. Scale the validation signals to the same power.

```
maleSpeechTrain   = maleSpeechTrain/norm(maleSpeechTrain);
femaleSpeechTrain = femaleSpeechTrain/norm(femaleSpeechTrain);
ampAdj            = max(abs([maleSpeechTrain;femaleSpeechTrain]));
maleSpeechTrain   = maleSpeechTrain/ampAdj;
femaleSpeechTrain = femaleSpeechTrain/ampAdj;

maleSpeechValidate   = maleSpeechValidate/norm(maleSpeechValidate);
femaleSpeechValidate = femaleSpeechValidate/norm(femaleSpeechValidate);
ampAdj               = max(abs([maleSpeechValidate;femaleSpeechValidate]));
maleSpeechValidate   = maleSpeechValidate/ampAdj;
femaleSpeechValidate = femaleSpeechValidate/ampAdj;
```

Create the training and validation "cocktail party" mixes.

```
mixTrain = maleSpeechTrain + femaleSpeechTrain;
mixTrain = mixTrain / max(mixTrain);

mixValidate = maleSpeechValidate + femaleSpeechValidate;
mixValidate = mixValidate / max(mixValidate);
```

Generate training STFTs.

```
WindowLength  = 128;
FFTLength     = 128;
OverlapLength = 128-1;
Fs            = 4000;
win           = hann(WindowLength,"periodic");

P_mix0 = stft(mixTrain, 'Window', win, 'OverlapLength', OverlapLength,...
              'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
P_M    = abs(stft(maleSpeechTrain, 'Window', win, 'OverlapLength', OverlapLength,...
              'FFTLength', FFTLength, 'FrequencyRange', 'onesided'));
P_F    = abs(stft(femaleSpeechTrain, 'Window', win, 'OverlapLength', OverlapLength,...
              'FFTLength', FFTLength, 'FrequencyRange', 'onesided'));
```

Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_mix = log(abs(P_mix0) + eps);
MP    = mean(P_mix(:));
SP    = std(P_mix(:));
P_mix = (P_mix - MP) / SP;
```

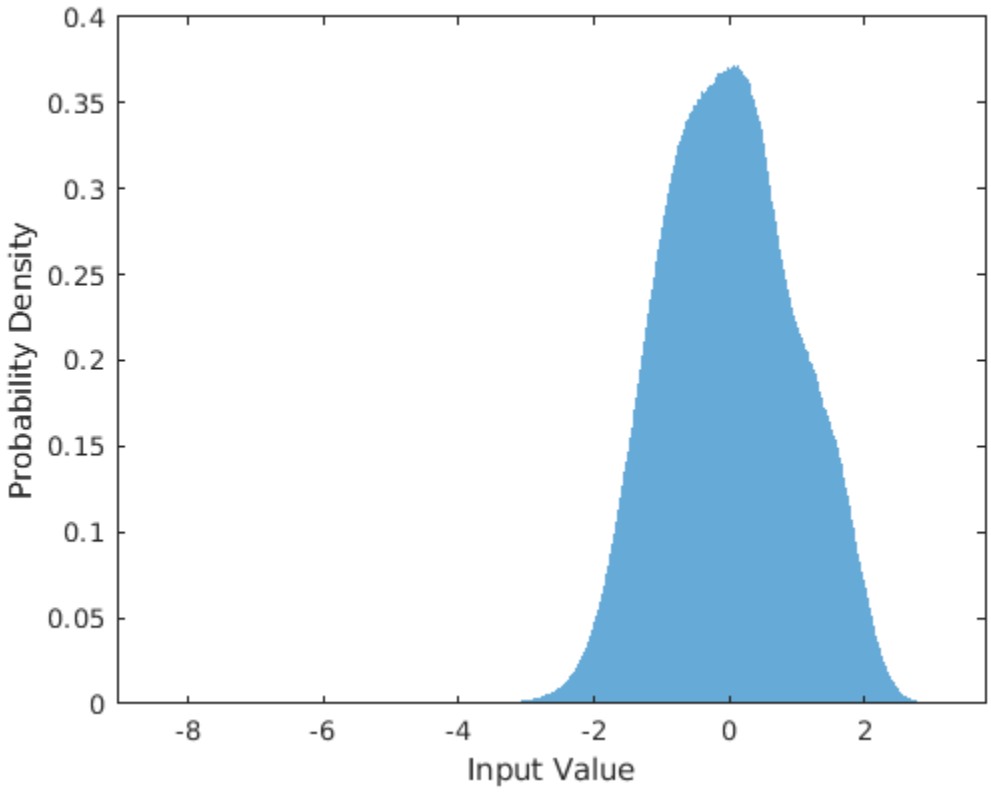
Generate validation STFTs. Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

```
P_Val_mix0 = stft(mixValidate, 'Window', win, 'OverlapLength', OverlapLength,...
                  'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
```

```
P_Val_M    = abs(stft(maleSpeechValidate, 'Window', win, 'OverlapLength', OverlapLength,...
                 'FFTLength', FFTLength, 'FrequencyRange', 'onesided'));
P_Val_F    = abs(stft(femaleSpeechValidate, 'Window', win, 'OverlapLength', OverlapLength,...
                 'FFTLength', FFTLength, 'FrequencyRange', 'onesided'));

P_Val_mix = log(abs(P_Val_mix0) + eps);
MP        = mean(P_Val_mix(:));
SP        = std(P_Val_mix(:));
P_Val_mix = (P_Val_mix - MP) / SP;
```

Training neural networks is easiest when the inputs to the network have a reasonably smooth distribution and are normalized. To check that the data distribution is smooth, plot a histogram of the STFT values of the training data.

```
figure(6)
histogram(P_mix,"EdgeColor","none","Normalization","pdf")
xlabel("Input Value")
ylabel("Probability Density")
```



Compute the training soft mask. Use this mask as the target signal while training the network.

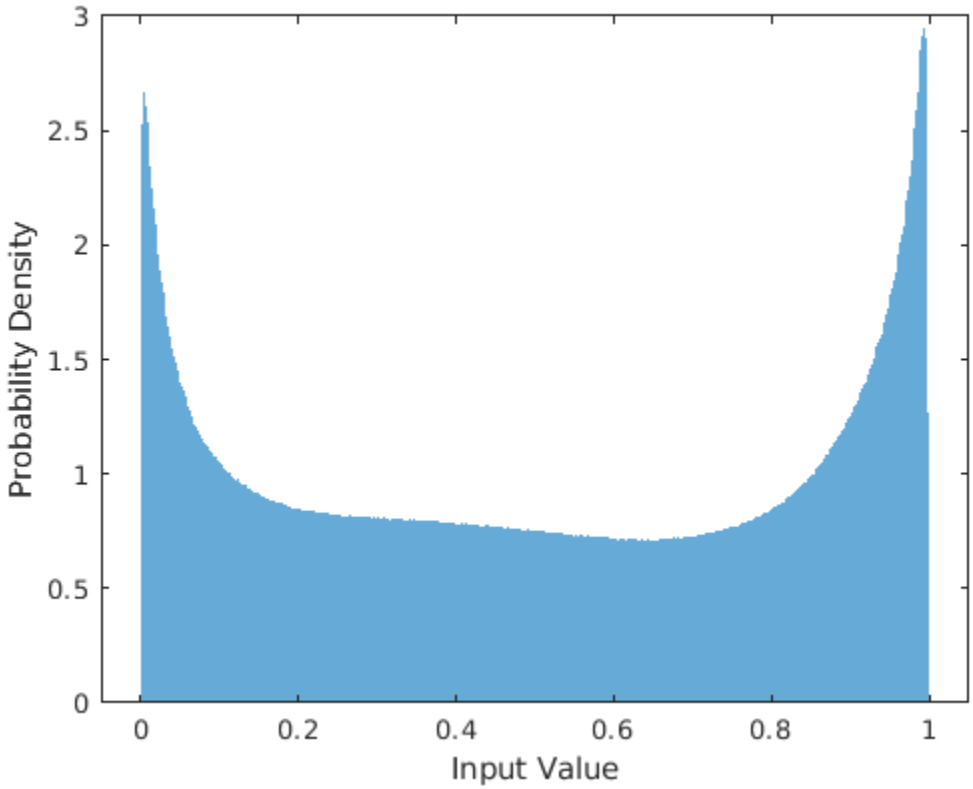
```
maskTrain = P_M ./ (P_M + P_F + eps);
```

Compute the validation soft mask. Use this mask to evaluate the mask emitted by the trained network.

```
maskValidate = P_Val_M ./ (P_Val_M + P_Val_F + eps);
```

To check that the target data distribution is smooth, plot a histogram of the mask values of the training data.

```
figure(7)
histogram(maskTrain,"EdgeColor","none","Normalization","pdf")
xlabel("Input Value")
ylabel("Probability Density")
```



Create chunks of size (65, 20) from the predictor and target signals. In order to get more training samples, use an overlap of 10 segments between consecutive chunks.

```
seqLen        = 20;
seqOverlap    = 10;
mixSequences  = zeros(1 + FFTLength/2,seqLen,1,0);
maskSequences = zeros(1 + FFTLength/2,seqLen,1,0);

loc = 1;
while loc < size(P_mix,2) - seqLen
    mixSequences(:,:,:,end+1)  = P_mix(:,loc:loc+seqLen-1); %#ok
    maskSequences(:,:,:,end+1) = maskTrain(:,loc:loc+seqLen-1); %#ok
    loc                        = loc + seqOverlap;
end
```

Create chunks of size (65,20) from the validation predictor and target signals.

```
mixValSequences  = zeros(1 + FFTLength/2,seqLen,1,0);
maskValSequences = zeros(1 + FFTLength/2,seqLen,1,0);
seqOverlap       = seqLen;
```

```
loc = 1;
while loc < size(P_Val_mix,2) - seqLen
    mixValSequences(:,:,:,end+1)  = P_Val_mix(:,loc:loc+seqLen-1); %#ok
    maskValSequences(:,:,:,end+1) = maskValidate(:,loc:loc+seqLen-1); %#ok
    loc                           = loc + seqOverlap;
end
```

Reshape the training and validation signals.

```
mixSequencesT  = reshape(mixSequences,    [1 1 (1 + FFTLength/2) * seqLen size(mixSequences,4)]);
mixSequencesV  = reshape(mixValSequences, [1 1 (1 + FFTLength/2) * seqLen size(mixValSequences,4
maskSequencesT = reshape(maskSequences,   [1 1 (1 + FFTLength/2) * seqLen size(maskSequences,4)]
maskSequencesV = reshape(maskValSequences,[1 1 (1 + FFTLength/2) * seqLen size(maskValSequences,4
```

Define Deep Learning Network

Define the layers of the network. Specify the input size to be images of size 1-by-1-by-1300. Define two hidden fully connected layers, each with 1300 neurons. Follow each hidden fully connected layer with a sigmoid layer. The batch normalization layers normalize the means and standard deviations of the outputs. Add a fully connected layer with 1300 neurons, followed by a regression layer.

```
numNodes = (1 + FFTLength/2) * seqLen;

layers = [ ...

    imageInputLayer([1 1 (1 + FFTLength/2)*seqLen],"Normalization","None")

    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(6)
    batchNormalizationLayer
    dropoutLayer(0.1)

    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(6)
    batchNormalizationLayer
    dropoutLayer(0.1)

    fullyConnectedLayer(numNodes)
    BiasedSigmoidLayer(0)

    regressionLayer

    ];
```

Specify the training options for the network. Set `MaxEpochs` to `3` so that the network makes three passes through the training data. Set `MiniBatchSize` to `64` so that the network looks at `64` training signals at a time. Set `Plots` to `training-progress` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot into the command line window. Set `Shuffle` to `every-epoch` to shuffle the training sequences at the beginning of each epoch. Set `LearnRateSchedule` to `piecewise` to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (1) has passed. Set `ValidationData` to the validation predictors and targets. Set `ValidationFrequency` such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (ADAM) solver.

```
maxEpochs     = 3;
miniBatchSize = 64;
```

```
options = trainingOptions("adam", ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "SequenceLength","longest", ...
    "Shuffle","every-epoch",...
    "Verbose",0, ...
    "Plots","training-progress",...
    "ValidationFrequency",floor(size(mixSequencesT,4)/miniBatchSize),...
    "ValidationData",{mixSequencesV,maskSequencesV},...
    "LearnRateSchedule","piecewise",...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1);
```

Train Deep Learning Network

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To load a pre-trained network, set `doTraining` to `false`.

```
doTraining = true;
if doTraining
    CocktailPartyNet = trainNetwork(mixSequencesT,maskSequencesT,layers,options);
else
    s = load("CocktailPartyNet.mat");
    CocktailPartyNet = s.CocktailPartyNet;
end
```

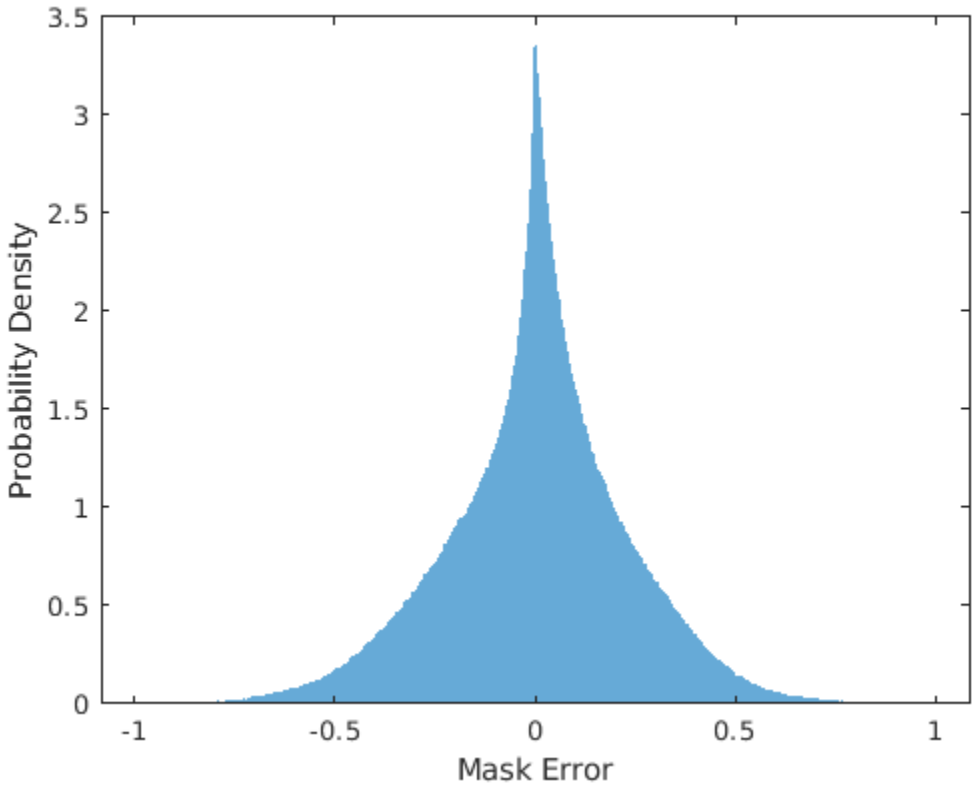
Pass the validation predictors to the network. The output is the estimated mask. Reshape the estimated mask.

```
estimatedMasks0 = predict(CocktailPartyNet,mixSequencesV);

estimatedMasks0 = estimatedMasks0.';
estimatedMasks0 = reshape(estimatedMasks0,1 + FFTLength/2,numel(estimatedMasks0)/(1 + FFTLength/2
```

Evaluate Deep Learning Network

Plot a histogram of the error between the actual and expected mask.

```
figure(8)
histogram(maskValSequences(:) - estimatedMasks0(:),"EdgeColor","none","Normalization","pdf")
xlabel("Mask Error")
ylabel("Probability Density")
```



Evaluate Soft Mask Estimation

Estimate male and female soft masks. Estimate male and female binary masks by thresholding the soft masks.

```
SoftMaleMask   = estimatedMasks0;
SoftFemaleMask = 1 - SoftMaleMask;
```

Shorten the mix STFT to match the size of the estimated mask.

```
P_Val_mix0 = P_Val_mix0(:,1:size(SoftMaleMask,2));
```

Multiply the mix STFT by the male soft mask to get the estimated male speech STFT.

```
P_Male = P_Val_mix0 .* SoftMaleMask;
```

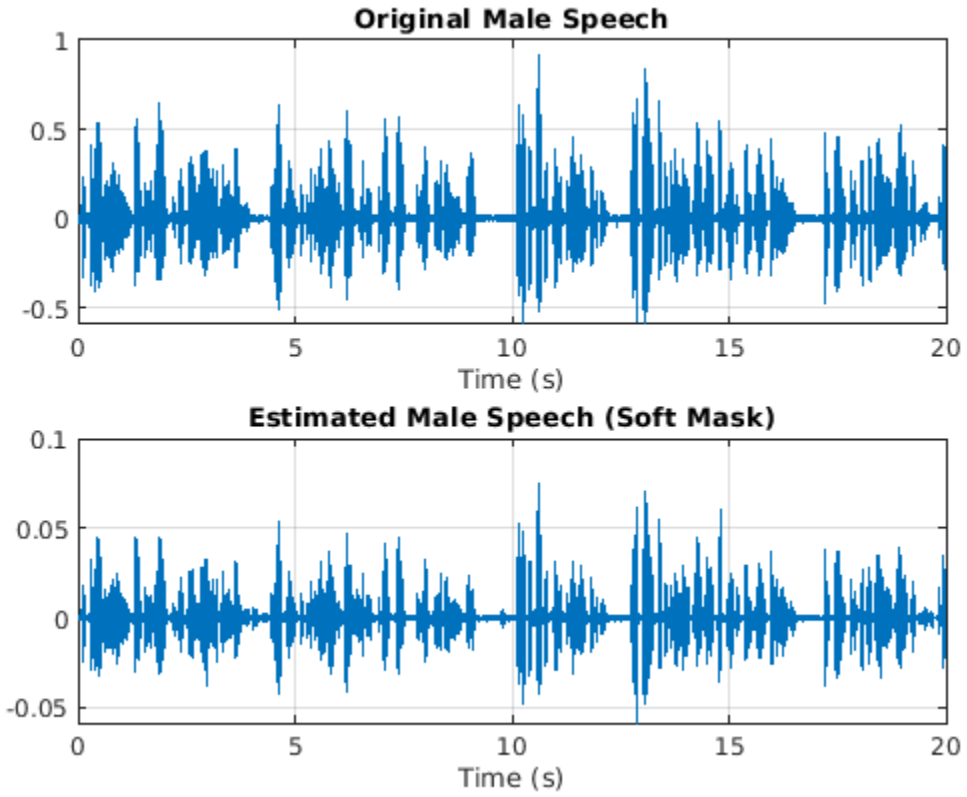
Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
maleSpeech_est_soft = istft(P_Male, 'Window', win, 'OverlapLength', OverlapLength,...
                            'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
                            'FrequencyRange', 'onesided');
maleSpeech_est_soft = maleSpeech_est_soft / max(abs(maleSpeech_est_soft));
```

Visualize the estimated and original male speech signals. Listen to the estimated soft mask male speech.

```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t     = range * (1/Fs);
```

```
figure(9)
subplot(2,1,1)
plot(t,maleSpeechValidate(range))
title("Original Male Speech")
xlabel("Time (s)")
grid on

subplot(2,1,2)
plot(t,maleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Male Speech (Soft Mask)")
grid on
```



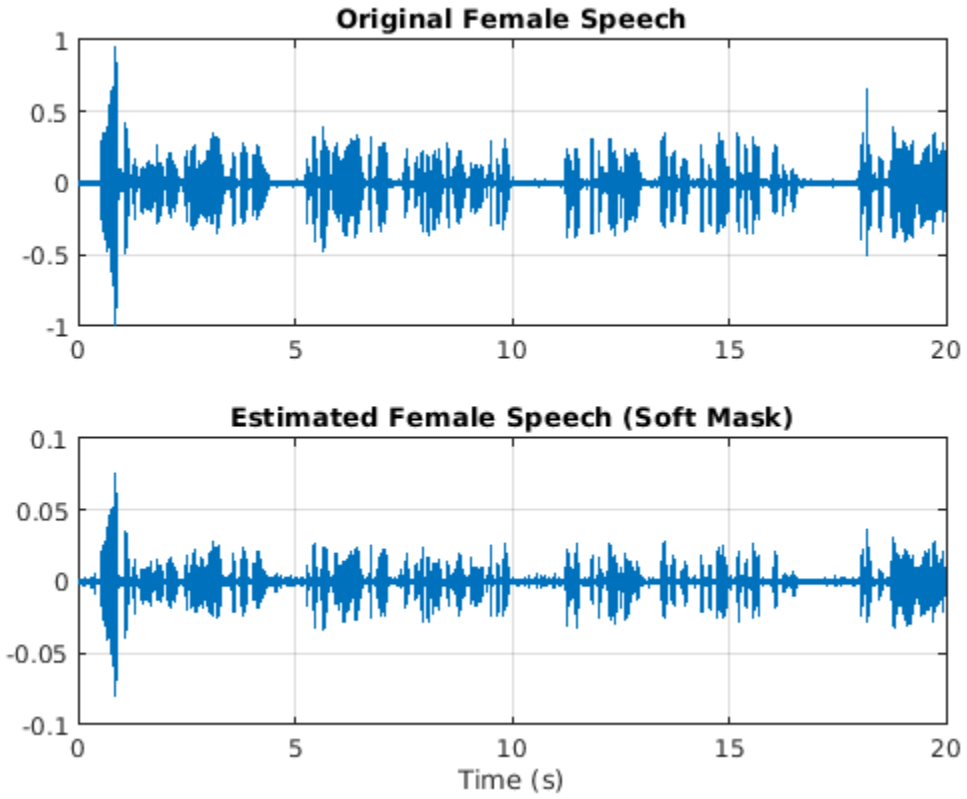

```
sound(maleSpeech_est_soft(range),Fs)
```

Multiply the mix STFT by the female soft mask to get the estimated female speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
P_Female = P_Val_mix0 .* SoftFemaleMask;

femaleSpeech_est_soft = istft(P_Female, 'Window', win, 'OverlapLength', OverlapLength,...
                              'FFTLength',FFTLength, 'ConjugateSymmetric',true,...
                              'FrequencyRange', 'onesided');
femaleSpeech_est_soft = femaleSpeech_est_soft / max(femaleSpeech_est_soft);
```

Visualize the estimated and original female signals. Listen to the estimated female speech.

```
range = (numel(win):numel(maleSpeech_est_soft) - numel(win));
t     = range * (1/Fs);

figure(10)
subplot(2,1,1)
plot(t,femaleSpeechValidate(range))
title("Original Female Speech")
grid on

subplot(2,1,2)
plot(t,femaleSpeech_est_soft(range))
xlabel("Time (s)")
title("Estimated Female Speech (Soft Mask)")
grid on
```




```
sound(femaleSpeech_est_soft(range),Fs)
```

Evaluate Binary Mask Estimation

Estimate male and female binary masks by thresholding the soft masks.

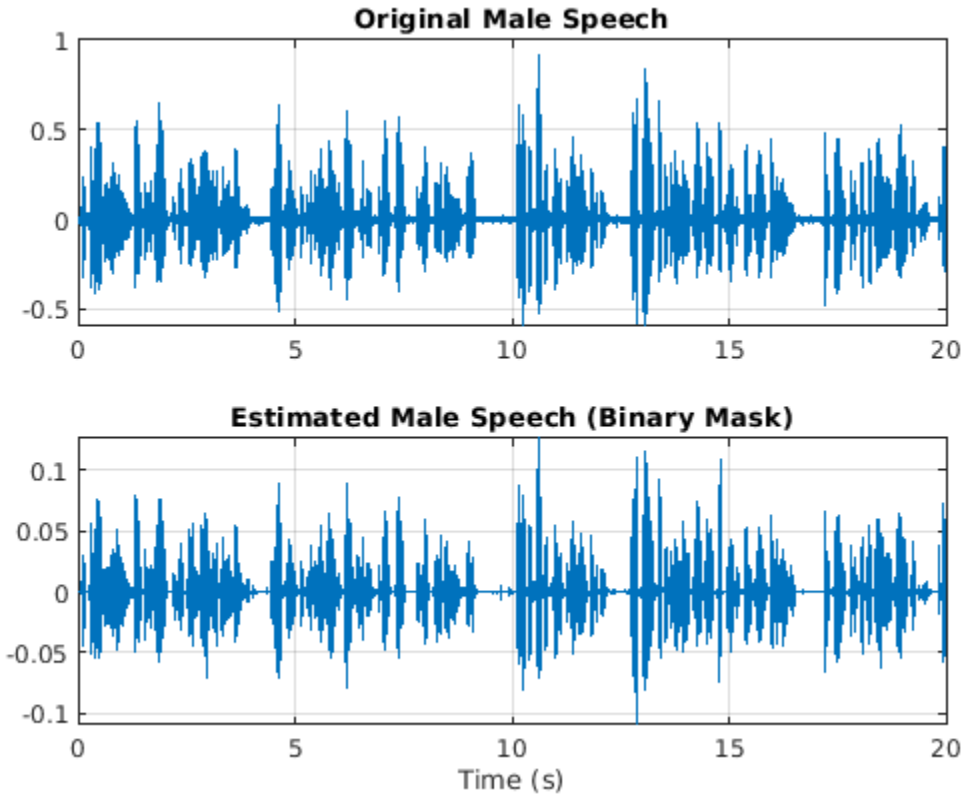
```
HardMaleMask   = SoftMaleMask >= 0.5;
HardFemaleMask = SoftMaleMask < 0.5;
```

Multiply the mix STFT by the male binary mask to get the estimated male speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
P_Male = P_Val_mix0 .* HardMaleMask;

maleSpeech_est_hard = istft(P_Male, 'Window', win, 'OverlapLength', OverlapLength,...
                            'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
                            'FrequencyRange', 'onesided');
maleSpeech_est_hard = maleSpeech_est_hard / max(maleSpeech_est_hard);
```

Visualize the estimated and original male speech signals. Listen to the estimated binary mask male speech.

```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t = range * (1/Fs);

figure(11)
subplot(2,1,1)
plot(t,maleSpeechValidate(range))
title("Original Male Speech")
grid on

subplot(2,1,2)
plot(t,maleSpeech_est_hard(range))
xlabel("Time (s)")
title("Estimated Male Speech (Binary Mask)")
grid on
```



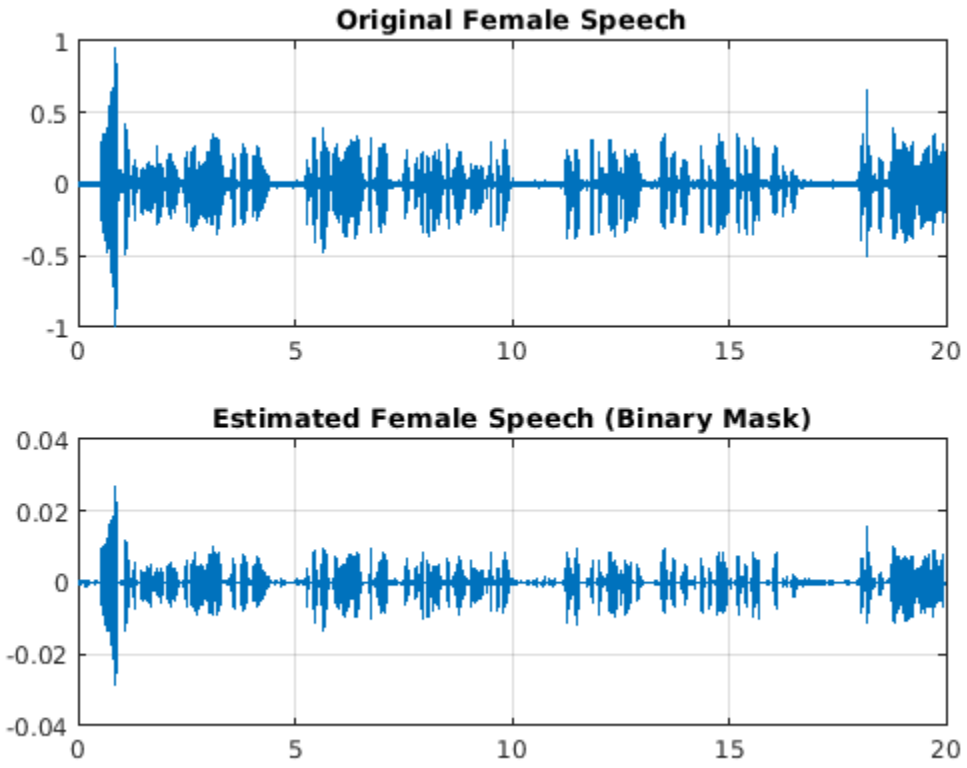

```
sound(maleSpeech_est_hard(range),Fs)
```

Multiply the mix STFT by the female binary mask to get the estimated male speech STFT. Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
P_Female = P_Val_mix0 .* HardFemaleMask;

femaleSpeech_est_hard = istft(P_Female, 'Window', win, 'OverlapLength', OverlapLength,...
                              'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
                              'FrequencyRange', 'onesided');
femaleSpeech_est_hard = femaleSpeech_est_hard / max(femaleSpeech_est_hard);
```

Visualize the estimated and original female speech signals. Listen to the estimated female speech.

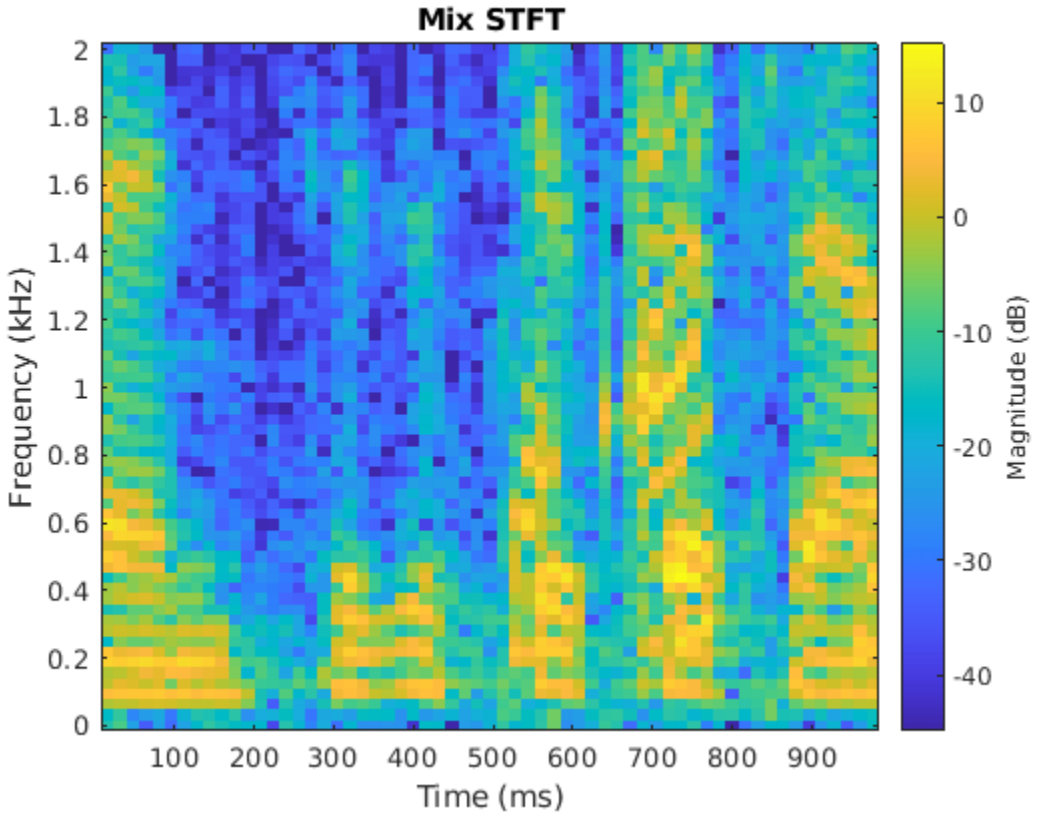
```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t = range * (1/Fs);

figure(12)
subplot(2,1,1)
plot(t,femaleSpeechValidate(range))
title("Original Female Speech")
grid on

subplot(2,1,2)
plot(t,femaleSpeech_est_hard(range))
title("Estimated Female Speech (Binary Mask)")
grid on
```



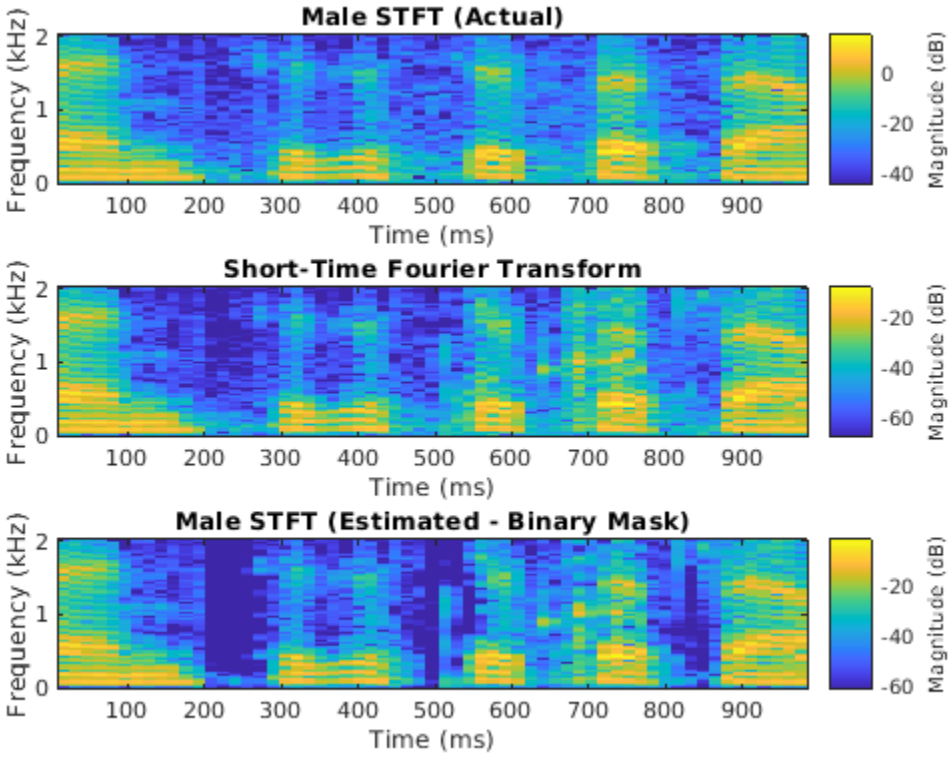

```
sound(femaleSpeech_est_hard(range),Fs)
```

Compare STFTs of a one-second segment for mix, original female and male, and estimated female and male, respectively.

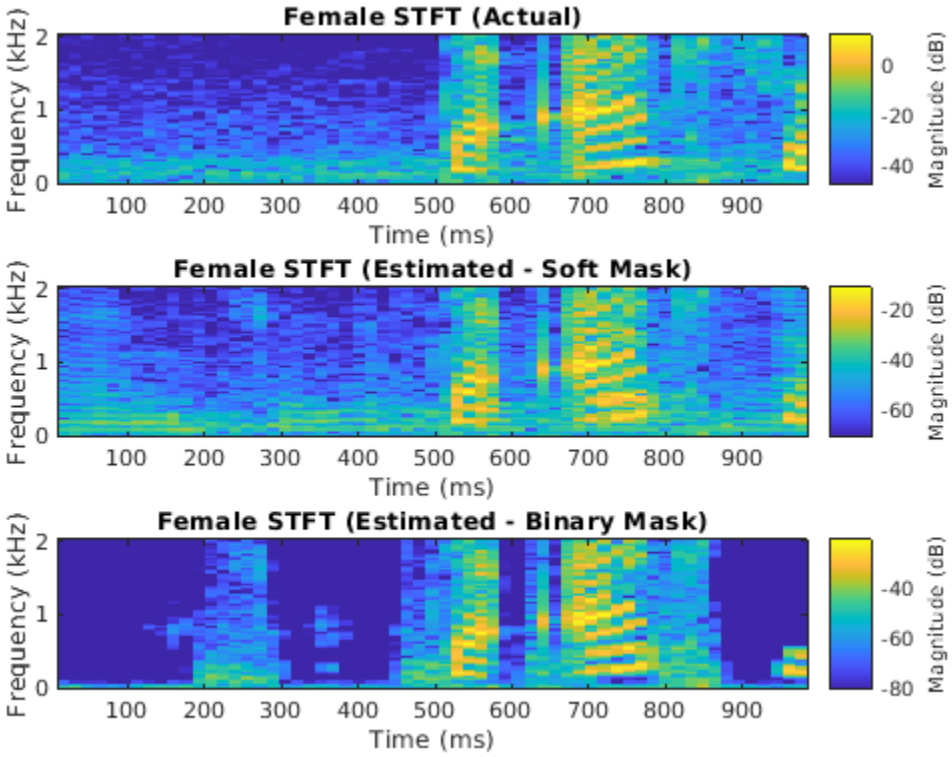
```
range = 7e4:7.4e4;

figure(13)
stft(mixValidate(range), Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Mix STFT")
```




```
figure(14)
subplot(3,1,1)
stft(maleSpeechValidate(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Male STFT (Actual)")
subplot(3,1,2)
stft(maleSpeech_est_soft(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
subplot(3,1,3)
stft(maleSpeech_est_hard(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Male STFT (Estimated - Binary Mask)");
```




```
figure(15)
subplot(3,1,1)
stft(femaleSpeechValidate(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Actual)")
subplot(3,1,2)
stft(femaleSpeech_est_soft(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Estimated - Soft Mask)")
subplot(3,1,3)
stft(femaleSpeech_est_hard(range),Fs, 'Window', win, 'OverlapLength', 64,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
title("Female STFT (Estimated - Binary Mask)")
```



References

[1] "Probabilistic Binary-Mask Cocktail-Party Source Separation in a Convolutional Deep Neural Network", Andrew J.R. Simpson, 2015.

See Also

`trainingOptions` | `trainNetwork`

More About

- “Deep Learning in MATLAB” on page 1-2

Voice Activity Detection in Noise Using Deep Learning

This example shows how to detect regions of speech in a low signal-to-noise environment using deep learning. The example uses the Speech Commands Dataset to train a Bidirectional Long Short-Term Memory (BiLSTM) network to detect voice activity.

Introduction

Voice activity detection is an essential component of many audio systems, such as automatic speech recognition and speaker recognition. Voice activity detection can be especially challenging in low signal-to-noise (SNR) situations, where speech is obstructed by noise.

This example uses long short-term memory (LSTM) networks, which are a type of recurrent neural network (RNN) well-suited to study sequence and time-series data. An LSTM network can learn long-term dependencies between time steps of a sequence. An LSTM layer (`lstmLayer`) can look at the time sequence in the forward direction, while a bidirectional LSTM layer (`bilstmLayer`) can look at the time sequence in both forward and backward directions. This example uses a bidirectional LSTM layer.

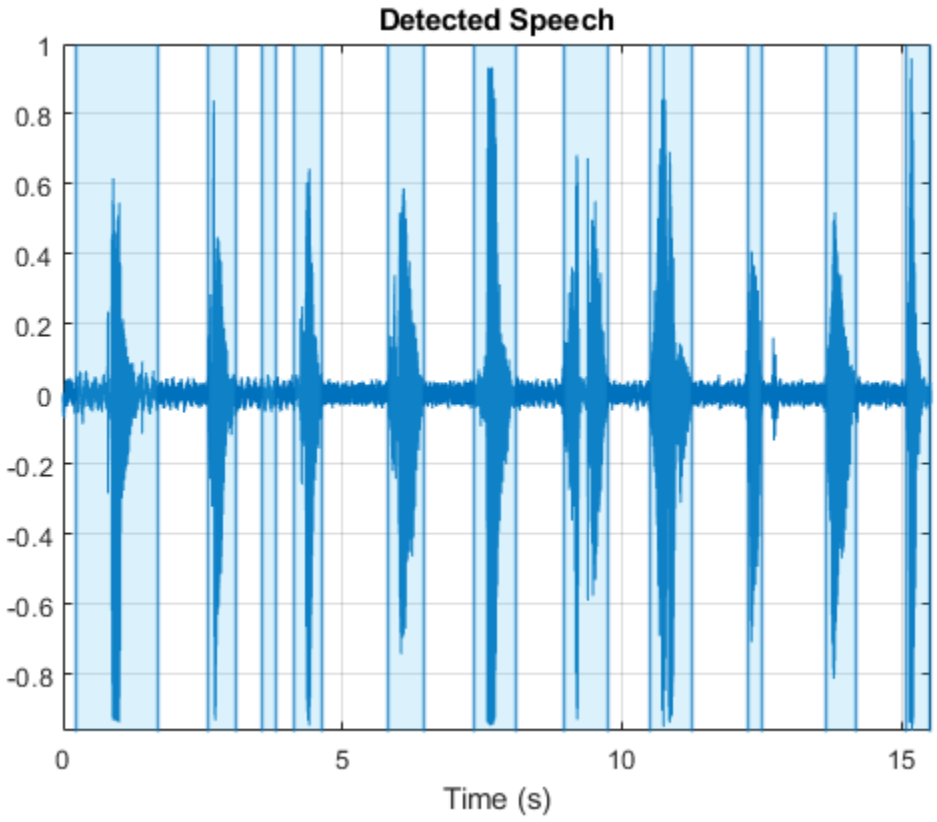
This example trains a voice activity detection bidirectional LSTM network with feature sequences of spectral characteristics and a harmonic ratio metric.

In high SNR scenarios, traditional speech detection algorithms perform adequately. Read in an audio file that consists of words spoken with pauses between. Resample the audio to 16 kHz. Listen to the audio.

```
fs = 16e3;
[speech,fileFs] = audioread('Counting-16-44p1-mono-15secs.wav');
speech = resample(speech,fs,fileFs);
speech = speech/max(abs(speech));
sound(speech,fs)
```

Use the `detectSpeech` (Audio Toolbox) function to locate regions of speech. The `detectSpeech` function correctly identifies all regions of speech.

```
win = hamming(50e-3 * fs,'periodic');
detectSpeech(speech,fs,'Window',win)
```

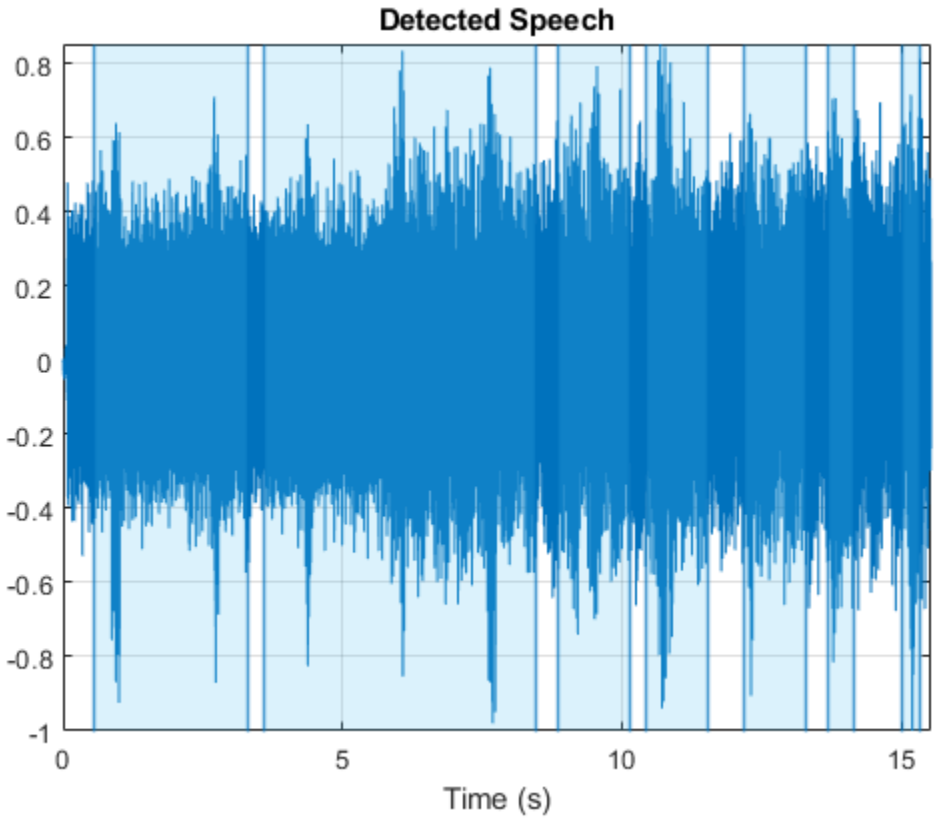


Corrupt the audio signal with washing machine noise at a -20 dB SNR. Listen to the corrupted audio.

```
[noise,fileFs] = audioread('WashingMachine-16-8-mono-200secs.mp3');
noise = resample(noise,fs,fileFs);

SNR = -20;
noiseGain = 10^(-SNR/20) * norm(speech) / norm(noise);

noisySpeech = speech + noiseGain*noise(1:numel(speech));
noisySpeech = noisySpeech./max(abs(noisySpeech));

sound(noisySpeech,fs)
```

Call `detectSpeech` on the noisy audio signal. The function fails to detect the speech regions given the very low SNR.

```
detectSpeech(noisySpeech,fs,'Window',win)
```



Download and load a pretrained network and a configured `audioFeatureExtractor` (Audio Toolbox) object. The network was trained to detect speech in a low SNR environments given features output from the `audioFeatureExtractor` object.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/VoiceActivityDetection.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'VoiceActivityDetection');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 8 MB) ...')
        unzip(url,downloadNetFolder)
    end
load(fullfile(netFolder,'voiceActivityDetectionExample.mat'));

speechDetectNet
```

```
speechDetectNet =
  SeriesNetwork with properties:

         Layers: [6×1 nnet.cnn.layer.Layer]
     InputNames: {'sequenceinput'}
    OutputNames: {'classoutput'}
```

```
afe
```

```
afe =
  audioFeatureExtractor with properties:
```

```
  Properties
                    Window: [256×1 double]
              OverlapLength: 128
                 SampleRate: 16000
                  FFTLength: []
   SpectralDescriptorInput: 'linearSpectrum'

  Enabled Features
    spectralCentroid, spectralCrest, spectralEntropy, spectralFlux, spectralKurtosis, spectralRo
    spectralSkewness, spectralSlope, harmonicRatio

  Disabled Features
    linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
    mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralDecrease, spectralFlatness
    spectralSpread, pitch


  To extract a feature, set the corresponding property to true.
  For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Extract features from the speech data and then normalize them. Orient the features so that time is across columns.

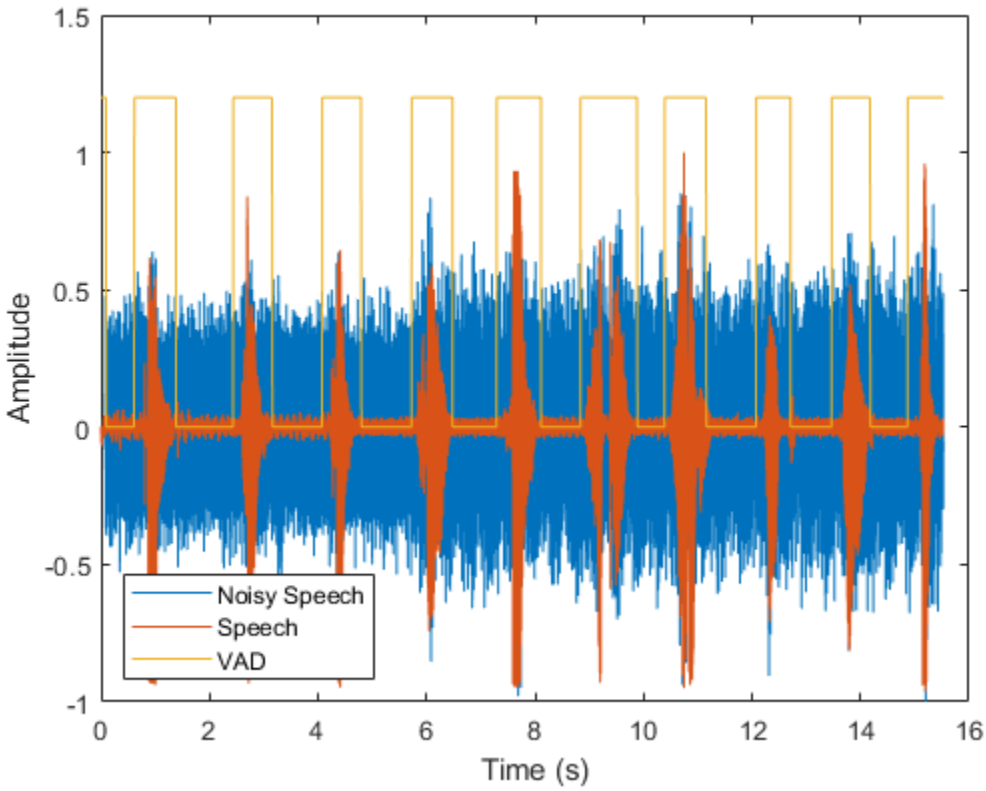
```
features = extract(afe,noisySpeech);
features = (features - mean(features,1)) ./ std(features,[],1);
features = features';
```

Pass the features through the speech detection network to classify each feature vector as belonging to a frame of speech or not.

```
decisionsCategorical = classify(speechDetectNet,features);
```

Each decision corresponds to an analysis window analyzed by the `audioFeatureExtractor`. Replicate the decisions so that they are in one-to-one correspondence with the audio samples. Plot the speech, the noisy speech, and the VAD decisions.

```
decisionsWindow = 1.2*(double(decisionsCategorical)-1);
decisionsSample = [repelem(decisionsWindow(1),numel(afe.Window)), ...
                   repelem(decisionsWindow(2:end),numel(afe.Window)-afe.OverlapLength)];


t = (0:numel(decisionsSample)-1)/afe.SampleRate;
plot(t,noisySpeech(1:numel(t)), ...
     t,speech(1:numel(t)), ...
     t,decisionsSample);
xlabel('Time (s)')
ylabel('Amplitude')
legend('Noisy Speech','Speech','VAD','Location','southwest')
```



You can also use the trained VAD network in a streaming context. To simulate a streaming environment, first save the speech and noise signals as WAV files. To simulate streaming input, you will read frames from the files and mix them at a desired SNR.

```
audiowrite('Speech.wav',speech,fs)
audiowrite('Noise.wav',noise,fs)
```

To apply the VAD network to streaming audio, you have to trade off between delay and accuracy. Define parameters for the streaming voice activity detection in noise demonstration. You can set the duration of the test, the sequence length fed into the network, the sequence hop length, and the SNR to test. Generally, increasing the sequence length increases the accuracy but also increases the lag. You can also choose the signal output to your device as the original signal or the noisy signal.

```
testDuration = 20 ;

sequenceLength = 400 ;

sequenceHop = 20 ;

SNR = -20 ;
noiseGain = 10^(-SNR/20) * norm(speech) / norm(noise);

signalToListenTo = "noisy" ;
```

Call the streaming demo helper function to observe the performance of the VAD network on streaming audio. The parameters you set using the live controls do not interrupt the streaming example. After the streaming demo is complete, you can modify parameters of the demonstration, then run the streaming demo again. You can find the code for the streaming demo in the Supporting Functions on page 14-0 .

```
helperStreamingDemo(speechDetectNet,afe, ...
                    'Speech.wav','Noise.wav', ...
                    testDuration,sequenceLength,sequenceHop,signalToListenTo,noiseGain);
```

The remainder of the example walks through training and evaluating the VAD network.

Train and Evaluate VAD Network

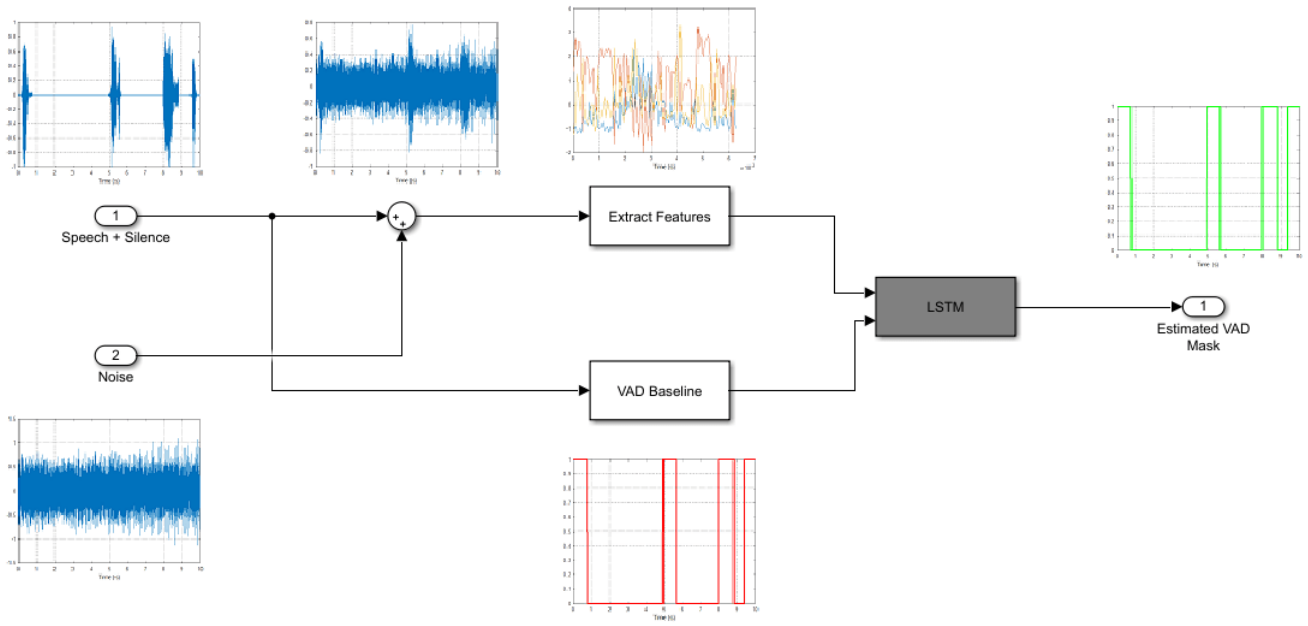
Training:

1. Create an `audioDatastore` (Audio Toolbox) that points to the audio speech files used to train the LSTM network.
2. Create a training signal consisting of speech segments separated by segments of silence of varying durations.
3. Corrupt the speech-plus-silence signal with washing machine noise (SNR = -10 dB).
4. Extract feature sequences consisting of spectral characteristics and harmonic ratio from the noisy signal.
5. Train the LSTM network using the feature sequences to identify regions of voice activity.

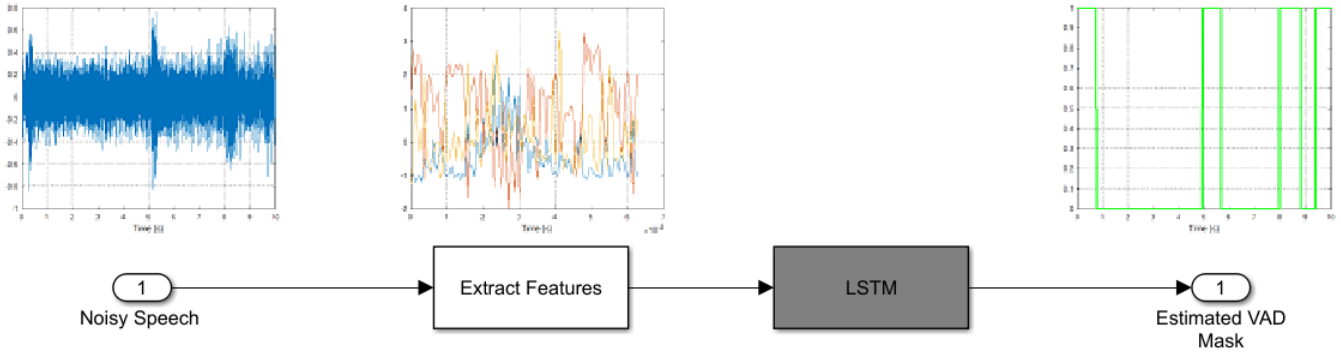
Prediction:

1. Create an `audioDatastore` of speech files used to test the trained network, and create a test signal consisting of speech separated by segments of silence.
2. Corrupt the test signal with washing machine noise (SNR = -10 dB).
3. Extract feature sequences from the noisy test signal.
4. Identify regions of voice activity by passing the test features through the trained network.
5. Compare the network's accuracy to the voice activity baseline from the signal-plus-silence test signal.

Here is a sketch of the training process.



Here is a sketch of the prediction process. You use the trained network to make predictions.



Load Speech Commands Data Set

Download and extract the Google Speech Commands Dataset [1] on page 14-0 .

```
url = 'https://ssd.mathworks.com/supportfiles/audio/google_speech.zip';

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading Google speech commands data set (1.9 GB)...')
    unzip(url,downloadFolder)
end
```

```
Downloading Google speech commands data set (1.9 GB)...
```

Create an `audioDatastore` (Audio Toolbox) that points to the training data set.

```
adsTrain = audioDatastore(fullfile(datasetFolder, 'train'), "Includesubfolders",true);
```

Create an `audioDatastore` (Audio Toolbox) that points to the validation data set.

```
adsValidation = audioDatastore(fullfile(datasetFolder, 'validation'), "Includesubfolders",true);
```

Create Speech-Plus-Silence Training Signal

Read the contents of an audio file using `read` (Audio Toolbox). Get the sample rate from the `adsInfo` struct.

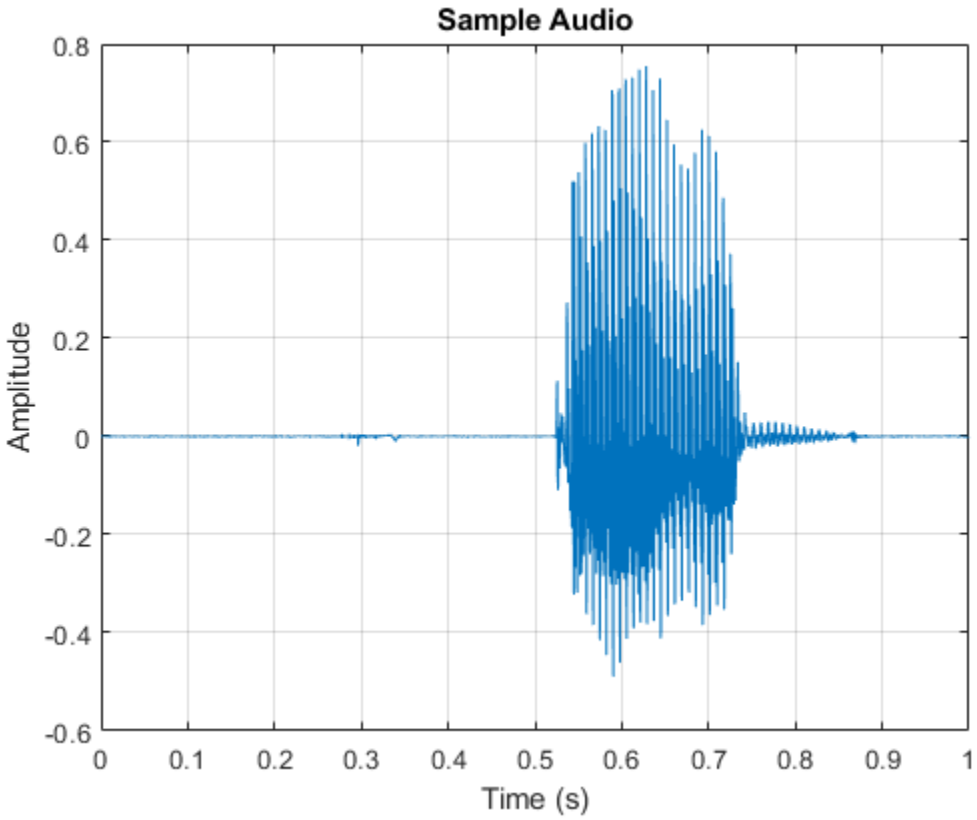
```
[data,adsInfo] = read(adsTrain);
Fs = adsInfo.SampleRate;
```

Listen to the audio signal using the sound command.

```
sound(data,Fs)
```

Plot the audio signal.

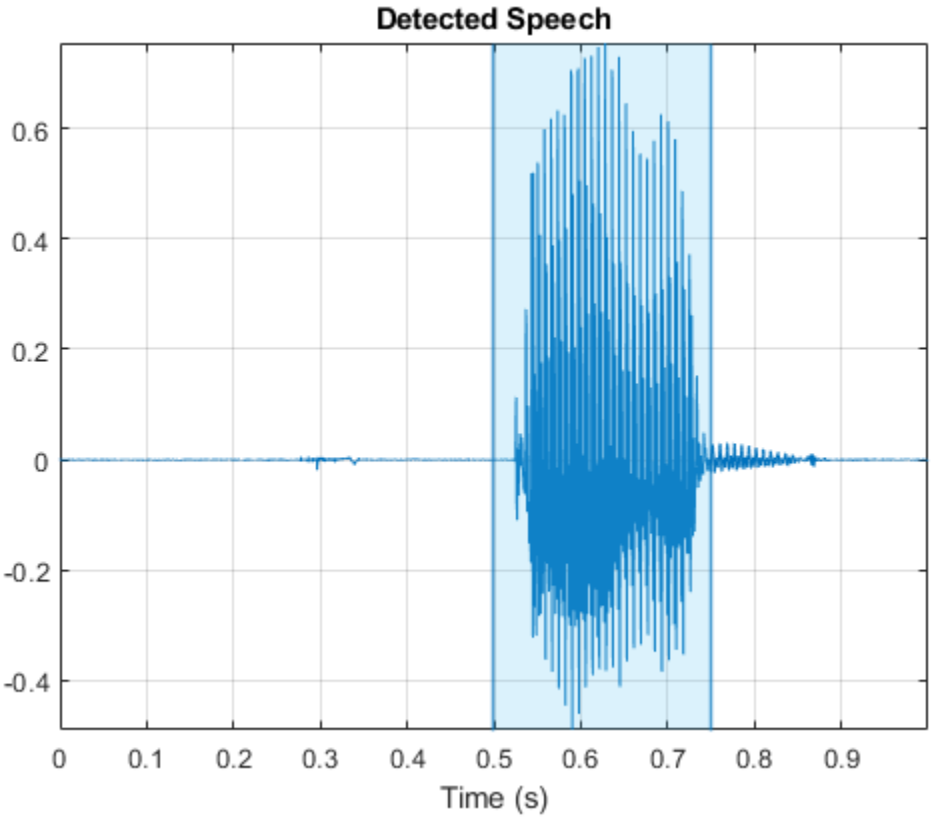
```
timeVector = (1/Fs) * (0:numel(data)-1);
plot(timeVector,data)
ylabel("Amplitude")
xlabel("Time (s)")
title("Sample Audio")
grid on
```



The signal has non-speech portions (silence, background noise, etc) that do not contain useful speech information. This example removes silence using the `detectSpeech` (Audio Toolbox) function.

Extract the useful portion of data. Define a 50 ms periodic Hamming window for analysis. Call `detectSpeech` with no output arguments to plot the detected speech regions. Call `detectSpeech` again to return the indices of the detected speech. Isolate the detected speech regions and then use the `sound` command to listen to the audio.

```
win = hamming(50e-3 * Fs,'periodic');

detectSpeech(data,Fs,'Window',win);
```




```
speechIndices = detectSpeech(data,Fs,'Window',win);

sound(data(speechIndices(1,1):speechIndices(1,2)),Fs)
```

The `detectSpeech` function returns indices that tightly surround the detected speech region. It was determined empirically that, for this example, extending the indices of the detected speech by five frames on either side increased the final model's performance. Extend the speech indices by five frames and then listen to the speech.

```
speechIndices(1,1) = max(speechIndices(1,1) - 5*numel(win),1);
speechIndices(1,2) = min(speechIndices(1,2) + 5*numel(win),numel(data));

sound(data(speechIndices(1,1):speechIndices(1,2)),Fs)
```

Reset the training datastore and shuffle the order of files in the datastores.

```
reset(adsTrain)
adsTrain = shuffle(adsTrain);
adsValidation = shuffle(adsValidation);
```

The `detectSpeech` function calculates statistics-based thresholds to determine the speech regions. You can skip the threshold calculation and speed up the `detectSpeech` function by specifying the thresholds directly. To determine thresholds for a data set, call `detectSpeech` on a sampling of files and get the thresholds it calculates. Take the mean of the thresholds.

```
TM = [];
for index1 = 1:500
     data = read(adsTrain);
     [~,T] = detectSpeech(data,Fs,'Window',win);
     TM = [TM;T];
end

T = mean(TM);

reset(adsTrain)
```

Create a 1000-second training signal by combining multiple speech files from the training data set. Use `detectSpeech` to remove unwanted portions of each file. Insert a random period of silence between speech segments.

Preallocate the training signal.

```
duration = 2000*Fs;
audioTraining = zeros(duration,1);
```

Preallocate the voice activity training mask. Values of 1 in the mask correspond to samples located in areas with voice activity. Values of 0 correspond to areas with no voice activity.

```
maskTraining = zeros(duration,1);
```

Specify a maximum silence segment duration of 2 seconds.

```
maxSilenceSegment = 2;
```

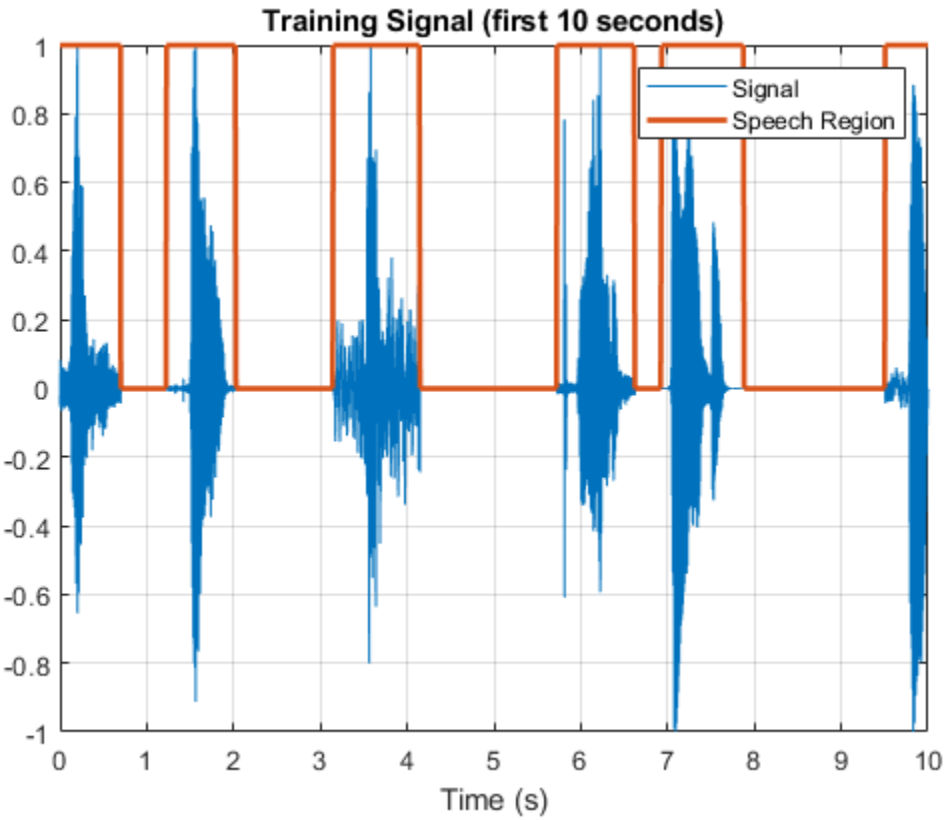
Construct the training signal by calling `read` on the datastore in a loop.

```
numSamples = 1;

while numSamples < duration
    data = read(adsTrain);
    data = data ./ max(abs(data)); % Normalize amplitude

    % Determine regions of speech
    idx = detectSpeech(data,Fs,'Window',win,'Thresholds',T);

    % If a region of speech is detected
    if ~isempty(idx)

        % Extend the indices by five frames
        idx(1,1) = max(1,idx(1,1) - 5*numel(win));
        idx(1,2) = min(length(data),idx(1,2) + 5*numel(win));

        % Isolate the speech
        data = data(idx(1,1):idx(1,2));
```

```
            % Write speech segment to training signal
            audioTraining(numSamples:numSamples+numel(data)-1) = data;

            % Set VAD baseline
            maskTraining(numSamples:numSamples+numel(data)-1) = true;

            % Random silence period
            numSilenceSamples = randi(maxSilenceSegment*Fs,1,1);
            numSamples = numSamples + numel(data) + numSilenceSamples;
        end
    end
```

Visualize a 10-second portion of the training signal. Plot the baseline voice activity mask.

```
figure
range = 1:10*Fs;
plot((1/Fs)*(range-1),audioTraining(range));
hold on
plot((1/Fs)*(range-1),maskTraining(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
legend("Signal","Speech Region")
title("Training Signal (first 10 seconds)");
```



Listen to the first 10 seconds of the training signal.

```
sound(audioTraining(range),Fs);
```

Add Noise to the Training Signal

Corrupt the training signal with washing machine noise by adding washing machine noise to the speech signal such that the signal-to-noise ratio is -10 dB.

Read 8 kHz noise and convert it to 16 kHz.

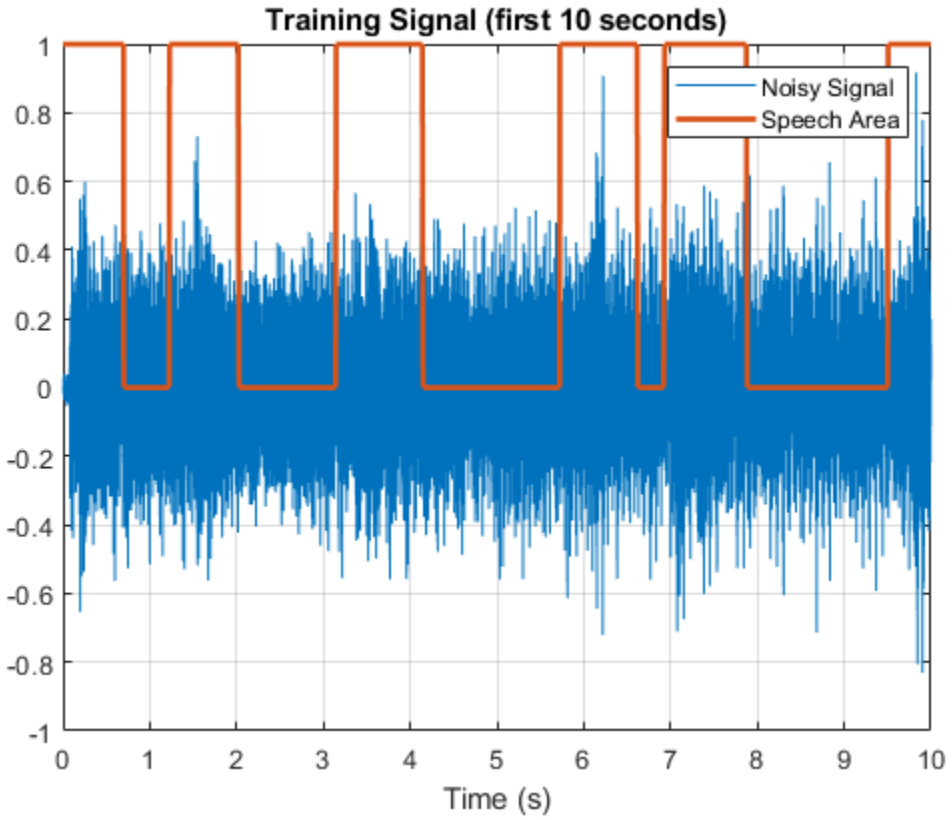
```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");
noise = resample(noise,2,1);
```

Corrupt training signal with noise.

```
audioTraining = audioTraining(1:numel(noise));
SNR = -10;
noise = 10^(-SNR/20) * noise * norm(audioTraining) / norm(noise);
audioTrainingNoisy = audioTraining + noise;
audioTrainingNoisy = audioTrainingNoisy / max(abs(audioTrainingNoisy));
```

Visualize a 10-second portion of the noisy training signal. Plot the baseline voice activity mask.

```
figure
plot((1/Fs)*(range-1),audioTrainingNoisy(range));
hold on
plot((1/Fs)*(range-1),maskTraining(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
legend("Noisy Signal","Speech Area")
title("Training Signal (first 10 seconds)");
```



Listen to the first 10 seconds of the noisy training signal.

```
sound(audioTrainingNoisy(range),Fs)
```

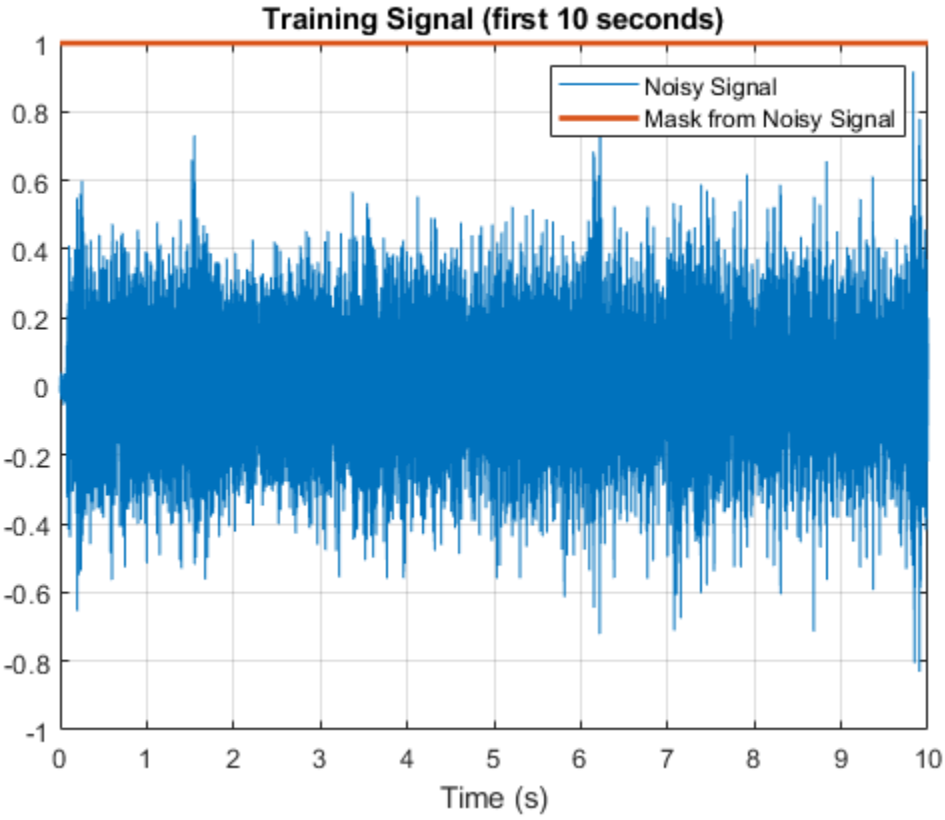
Note that you obtained the baseline voice activity mask using the noiseless speech-plus-silence signal. Verify that using `detectSpeech` on the noise-corrupted signal does not yield good results.

```
speechIndices = detectSpeech(audioTrainingNoisy,Fs,'Window',win);

speechIndices(:,1) = max(1,speechIndices(:,1) - 5*numel(win));
speechIndices(:,2) = min(numel(audioTrainingNoisy),speechIndices(:,2) + 5*numel(win));

noisyMask = zeros(size(audioTrainingNoisy));
for ii = 1:size(speechIndices)
    noisyMask(speechIndices(ii,1):speechIndices(ii,2)) = 1;
end
```

Visualize a 10-second portion of the noisy training signal. Plot the voice activity mask obtained by analyzing the noisy signal.

```
figure
plot((1/Fs)*(range-1),audioTrainingNoisy(range));
hold on
plot((1/Fs)*(range-1),noisyMask(range));
grid on
lines = findall(gcf,"Type","Line");
lines(1).LineWidth = 2;
xlabel("Time (s)")
```

```
legend("Noisy Signal","Mask from Noisy Signal")
title("Training Signal (first 10 seconds)");
```



Create Speech-Plus-Silence Validation Signal

Create a 200-second noisy speech signal to validate the trained network. Use the validation datastore. Note that the validation and training datastores have different speakers.

Preallocate the validation signal and the validation mask. You will use this mask to assess the accuracy of the trained network.

```
duration = 200*Fs;
audioValidation = zeros(duration,1);
maskValidation = zeros(duration,1);
```

Construct the validation signal by calling `read` on the datastore in a loop.

```
numSamples = 1;
while numSamples < duration
    data = read(adsValidation);
    data = data ./ max(abs(data)); % Normalize amplitude

    % Determine regions of speech
    idx = detectSpeech(data,Fs,'Window',win,'Thresholds',T);

    % If a region of speech is detected
    if ~isempty(idx)
```

```
            % Extend the indices by five frames
            idx(1,1) = max(1,idx(1,1) - 5*numel(win));
            idx(1,2) = min(length(data),idx(1,2) + 5*numel(win));

            % Isolate the speech
            data = data(idx(1,1):idx(1,2));

            % Write speech segment to training signal
            audioValidation(numSamples:numSamples+numel(data)-1) = data;

            % Set VAD Baseline
            maskValidation(numSamples:numSamples+numel(data)-1) = true;

            % Random silence period
            numSilenceSamples = randi(maxSilenceSegment*Fs,1,1);
            numSamples = numSamples + numel(data) + numSilenceSamples;
        end
    end
```

Corrupt the validation signal with washing machine noise by adding washing machine noise to the speech signal such that the signal-to-noise ratio is -10 dB. Use a different noise file for the validation signal than you did for the training signal.

```
noise = audioread("WashingMachine-16-8-mono-200secs.mp3");
noise = resample(noise,2,1);
noise = noise(1:duration);
audioValidation = audioValidation(1:numel(noise));

noise = 10^(-SNR/20) * noise * norm(audioValidation) / norm(noise);
audioValidationNoisy = audioValidation + noise;
audioValidationNoisy = audioValidationNoisy / max(abs(audioValidationNoisy));
```

Extract Training Features

This example trains the LSTM network using the following features:

1. `spectralCentroid` (Audio Toolbox)
2. `spectralCrest` (Audio Toolbox)
3. `spectralEntropy` (Audio Toolbox)
4. `spectralFlux` (Audio Toolbox)
5. `spectralKurtosis` (Audio Toolbox)
6. `spectralRolloffPoint` (Audio Toolbox)
7. `spectralSkewness` (Audio Toolbox)
8. `spectralSlope` (Audio Toolbox)
9. `harmonicRatio` (Audio Toolbox)

This example uses `audioFeatureExtractor` (Audio Toolbox) to create an optimal feature extraction pipeline for the feature set. Create an `audioFeatureExtractor` object to extract the feature set. Use a 256-point Hann window with 50% overlap.

```
afe = audioFeatureExtractor('SampleRate',Fs, ...
    'Window',hann(256,"Periodic"), ...
    'OverlapLength',128, ...
    ...
```

```
    'spectralCentroid',true, ...
    'spectralCrest',true, ...
    'spectralEntropy',true, ...
    'spectralFlux',true, ...
    'spectralKurtosis',true, ...
    'spectralRolloffPoint',true, ...
    'spectralSkewness',true, ...
    'spectralSlope',true, ...
    'harmonicRatio',true);

featuresTraining = extract(afe,audioTrainingNoisy);
```

Display the dimensions of the features matrix. The first dimension corresponds to the number of windows the signal was broken into (it depends on the window length and the overlap length). The second dimension is the number of features used in this example.

```
[numWindows,numFeatures] = size(featuresTraining)
```

```
numWindows = 125009
```

```
numFeatures = 9
```

In classification applications, it is a good practice to normalize all features to have zero mean and unity standard deviation.

Compute the mean and standard deviation for each coefficient, and use them to normalize the data.

```
M = mean(featuresTraining,1);
S = std(featuresTraining,[],1);
featuresTraining = (featuresTraining - M) ./ S;
```

Extract the features from the validation signal using the same process.

```
featuresValidation = extract(afe,audioValidationNoisy);
featuresValidation = (featuresValidation - mean(featuresValidation,1)) ./ std(featuresValidation
```

Each feature corresponds to 128 samples of data (the hop length). For each hop, set the expected voice/no voice value to the mode of the baseline mask values corresponding to those 128 samples. Convert the voice/no voice mask to categorical.

```
windowLength = numel(afe.Window);
hopLength = windowLength - afe.OverlapLength;
range = (hopLength) * (1:size(featuresTraining,1)) + hopLength;
maskMode = zeros(size(range));
for index = 1:numel(range)
    maskMode(index) = mode(maskTraining( (index-1)*hopLength+1:(index-1)*hopLength+windowLength
end
maskTraining = maskMode.';

maskTrainingCat = categorical(maskTraining);
```

Do the same for the validation mask.

```
range = (hopLength) * (1:size(featuresValidation,1)) + hopLength;
maskMode = zeros(size(range));
for index = 1:numel(range)
    maskMode(index) = mode(maskValidation( (index-1)*hopLength+1:(index-1)*hopLength+windowLengt
end
```

```
maskValidation = maskMode.';

maskValidationCat = categorical(maskValidation);
```

Split the training features and the mask into sequences of length 800, with 75% overlap between consecutive sequences.

```
sequenceLength = 800;
sequenceOverlap = round(0.75*sequenceLength);

trainFeatureCell = helperFeatureVector2Sequence(featuresTraining',sequenceLength,sequenceOverlap)
trainLabelCell = helperFeatureVector2Sequence(maskTrainingCat',sequenceLength,sequenceOverlap);
```

Define the LSTM Network Architecture

LSTM networks can learn long-term dependencies between time steps of sequence data. This example uses the bidirectional LSTM layer `bilstmLayer` to look at the sequence in both forward and backward directions.

Specify the input size to be sequences of length `9` (the number of features). Specify a hidden bidirectional LSTM layer with an output size of 200 and output a sequence. This command instructs the bidirectional LSTM layer to map the input time series into 200 features that are passed to the next layer. Then, specify a bidirectional LSTM layer with an output size of 200 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map its input into 200 features and then prepares the output for the fully connected layer. Finally, specify two classes by including a fully connected layer of size `2`, followed by a softmax layer and a classification layer.

```
layers = [ ...
    sequenceInputLayer( size(featuresValidation,2) )
    bilstmLayer(200,"OutputMode","sequence")
    bilstmLayer(200,"OutputMode","sequence")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ];
```

Next, specify the training options for the classifier. Set `MaxEpochs` to `20` so that the network makes 20 passes through the training data. Set `MiniBatchSize` to `64` so that the network looks at 64 training signals at a time. Set `Plots` to `"training-progress"` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot. Set `Shuffle` to `"every-epoch"` to shuffle the training sequence at the beginning of each epoch. Set `LearnRateSchedule` to `"piecewise"` to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (10) has passed. Set `ValidationData` to the validation predictors and targets.

This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
maxEpochs = 20;
miniBatchSize = 64;
options = trainingOptions("adam", ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",0, ...
```

```
    "SequenceLength",sequenceLength, ...
    "ValidationFrequency",floor(numel(trainFeatureCell)/miniBatchSize), ...
    "ValidationData",{featuresValidation.',maskValidationCat.'}, ...
    "Plots","training-progress", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",5);
```

Train the LSTM Network

Train the LSTM network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes.

```
doTraining = true;
if doTraining
   [speechDetectNet,netInfo] = trainNetwork(trainFeatureCell,trainLabelCell,layers,options);
    fprintf("Validation accuracy: %f percent.\n", netInfo.FinalValidationAccuracy);
else
    load speechDetectNet
end
```

```
Validation accuracy: 91.320312 percent.
```

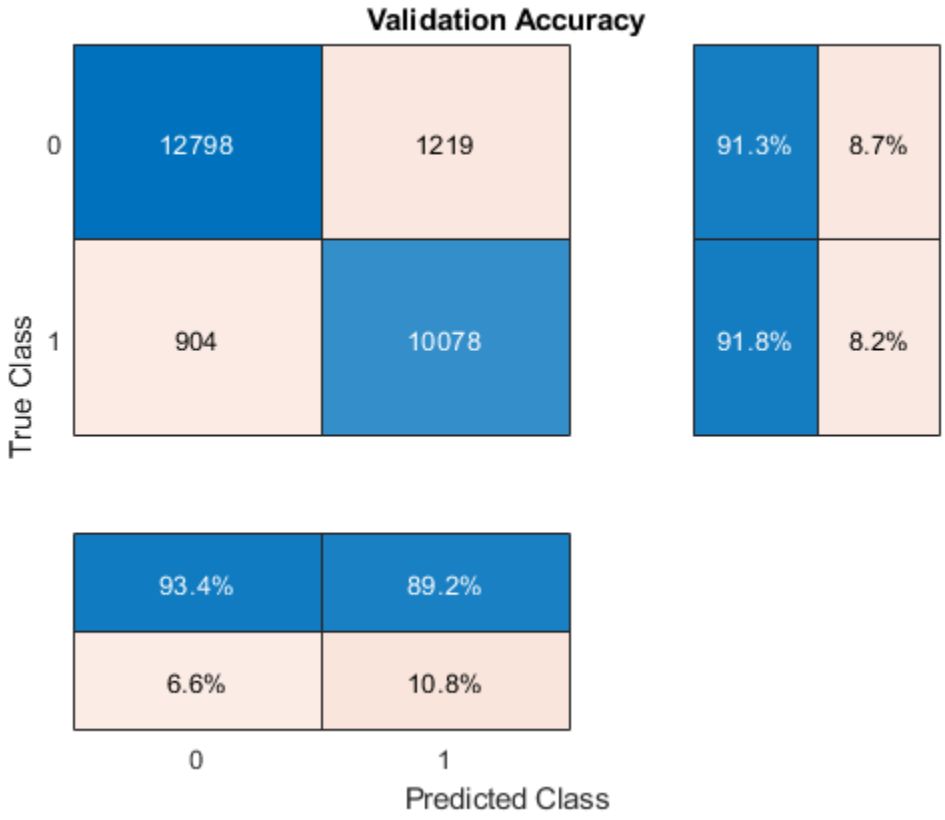
Use Trained Network to Detect Voice Activity

Estimate voice activity in the validation signal using the trained network. Convert the estimated VAD mask from categorical to double.

```
EstimatedVADMask = classify(speechDetectNet,featuresValidation.');
EstimatedVADMask = double(EstimatedVADMask);
EstimatedVADMask = EstimatedVADMask.' - 1;
```

Calculate and plot the validation confusion matrix from the vectors of actual and estimated labels.

```
figure
cm = confusionchart(maskValidation,EstimatedVADMask,"title","Validation Accuracy");
cm.ColumnSummary = "column-normalized";
cm.RowSummary = "row-normalized";
```



If you changed parameters of your network or feature extraction pipeline, consider resaving the MAT file with the new network and `audioFeatureExtractor` object.

```
resaveNetwork = false;
if resaveNetwork
     save('Audio_VoiceActivityDetectionExample.mat','speechDetectNet','afe');
end
```

Supporting Functions

Convert Feature Vectors to Sequences

```
function [sequences,sequencePerFile] = helperFeatureVector2Sequence(features,featureVectorsPerSe
    if featureVectorsPerSequence <= featureVectorOverlap
        error('The number of overlapping feature vectors must be less than the number of feature
    end

    if ~iscell(features)
        features = {features};
    end
    hopLength = featureVectorsPerSequence - featureVectorOverlap;
    idx1 = 1;
    sequences = {};
    sequencePerFile = cell(numel(features),1);
    for ii = 1:numel(features)
        sequencePerFile{ii} = floor((size(features{ii},2) - featureVectorsPerSequence)/hopLength
        idx2 = 1;
```

```
        for j = 1:sequencePerFile{ii}
            sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1); %#ok<A
            idx1 = idx1 + 1;
            idx2 = idx2 + hopLength;
        end
    end
end
```

Streaming Demo

```
function helperStreamingDemo(speechDetectNet,afe,cleanSpeech,noise,testDuration,sequenceLength,se
```

Create `dsp.AudioFileReader` (DSP System Toolbox) objects to read from the speech and noise files frame by frame.

```
speechReader = dsp.AudioFileReader(cleanSpeech,'PlayCount',inf);
noiseReader = dsp.AudioFileReader(noise,'PlayCount',inf);
fs = speechReader.SampleRate;
```

Create a `dsp.MovingStandardDeviation` (DSP System Toolbox) object and a `dsp.MovingAverage` (DSP System Toolbox) object. You will use these to determine the standard deviation and mean of the audio features for normalization. The statistics should improve over time.

```
movSTD = dsp.MovingStandardDeviation('Method','Exponential weighting','ForgettingFactor',1);
movMean = dsp.MovingAverage('Method','Exponential weighting','ForgettingFactor',1);
```

Create three `dsp.AsyncBuffer` (DSP System Toolbox) objects. One to buffer the input audio, one to buffer the extracted features, and one to buffer the output buffer. The output buffer is only necessary for visualizing the decisions in real time.

```
audioInBuffer = dsp.AsyncBuffer;
featureBuffer = dsp.AsyncBuffer;
audioOutBuffer = dsp.AsyncBuffer;
```

For the audio buffers, you will buffer both the original clean speech signal, and the noisy signal. You will play back only the specified `signalToListenTo`. Convert the `signalToListenTo` variable to the channel you want to listen to.

```
channelToListenTo = 1;
if strcmp(signalToListenTo,"clean")
    channelToListenTo = 2;
end
```

Create a time scope to visualize the original speech signal, the noisy signal that the network is applied to, and the decision output from the network.

```
scope = timescope('SampleRate',fs, ...
    'TimeSpanSource','property', ...
    'TimeSpan',3, ...
    'BufferLength',fs*3*3, ...
    'YLimits',[-1 1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'NumInputPorts',3, ...
    'LayoutDimensions',[3,1], ...
    'Title','Noisy Speech');
scope.ActiveDisplay = 2;
scope.Title = 'Clean Speech (Original)';
```

```
scope.YLimits = [-1 1];
scope.ActiveDisplay = 3;
scope.Title = 'Detected Speech';
scope.YLimits = [-1 1];
```

Create an `audioDeviceWriter` (Audio Toolbox) object to play either the original or noisy audio from your speakers.

```
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Initialize variables used in the loop.

```
windowLength = numel(afe.Window);
hopLength = windowLength - afe.OverlapLength;
myMax = 0;
audioBufferInitialized = false;
featureBufferInitialized = false;
```

Run the streaming demonstration.

```
tic
while toc < testDuration

    % Read a frame of the speech signal and a frame of the noise signal
    speechIn = speechReader();
    noiseIn = noiseReader();

    % Mix the speech and noise at the specified SNR
    noisyAudio = speechIn + noiseGain*noiseIn;

    % Update a running max for normalization
    myMax = max(myMax,max(abs(noisyAudio)));

    % Write the noisy audio and speech to buffers
    write(audioInBuffer,[noisyAudio,speechIn]);

    % If enough samples are buffered,
    % mark the audio buffer as initialized and push the read pointer
    % for the audio buffer up a window length.
    if audioInBuffer.NumUnreadSamples >= windowLength && ~audioBufferInitialized
        audioBufferInitialized = true;
        read(audioInBuffer,windowLength);
    end

    % If enough samples are in the audio buffer to calculate a feature
    % vector, read the samples, normalize them, extract the feature vectors, and write
    % the latest feature vector to the features buffer.
    while (audioInBuffer.NumUnreadSamples >= hopLength) && audioBufferInitialized
        x = read(audioInBuffer,windowLength + hopLength,windowLength);
        write(audioOutBuffer,x(end-hopLength+1:end,:));
        noisyAudio = x(:,1);
        noisyAudio = noisyAudio/myMax;
        features = extract(afe,noisyAudio);
        write(featureBuffer,features(2,:));
    end

    % If enough feature vectors are buffered, mark the feature buffer
    % as initialized and push the read pointer for the feature buffer
```

```
            % and the audio output buffer (so that they are in sync).
            if featureBuffer.NumUnreadSamples >= (sequenceLength + sequenceHop) && ~featureBufferIni
                featureBufferInitialized = true;
                read(featureBuffer,sequenceLength - sequenceHop);
                read(audioOutBuffer,(sequenceLength - sequenceHop)*windowLength);
            end

            while featureBuffer.NumUnreadSamples >= sequenceHop && featureBufferInitialized
                features = read(featureBuffer,sequenceLength,sequenceLength - sequenceHop);
                features(isnan(features)) = 0;

                % Use only the new features to update the
                % standard deviation and mean. Normalize the features.
                localSTD = movSTD(features(end-sequenceHop+1:end,:));
                localMean = movMean(features(end-sequenceHop+1:end,:));
                features = (features - localMean(end,:)) ./ localSTD(end,:);

                decision = classify(speechDetectNet,features');
                decision = decision(end-sequenceHop+1:end);
                decision = double(decision)' - 1;
                decision = repelem(decision,hopLength);

                audioHop = read(audioOutBuffer,sequenceHop*hopLength);

                % Listen to the speech or speech+noise
                deviceWriter(audioHop(:,channelToListenTo));

                % Visualize the speech+noise, the original speech, and the
                % voice activity detection.
                scope(audioHop(:,1),audioHop(:,2),audioHop(:,1).*decision)
            end
        end
        release(deviceWriter)
        release(audioInBuffer)
        release(audioOutBuffer)
        release(featureBuffer)
        release(movSTD)
        release(movMean)
        release(scope)
    end
```

References

[1] Warden P. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license

See Also

`trainingOptions` | `trainNetwork`

More About

- “Deep Learning in MATLAB” on page 1-2

Denoise Speech Using Deep Learning Networks

This example shows how to denoise speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

Introduction

The aim of speech denoising is to remove noise from speech signals while enhancing the quality and intelligibility of speech. This example showcases the removal of washing machine noise from speech signals using deep learning networks. The example compares two types of networks applied to the same task: fully connected, and convolutional.

Problem Summary

Consider the following speech signal sampled at 8 kHz.

```
[cleanAudio,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");
sound(cleanAudio,fs)
```

Add washing machine noise to the speech signal. Set the noise power such that the signal-to-noise ratio (SNR) is zero dB.

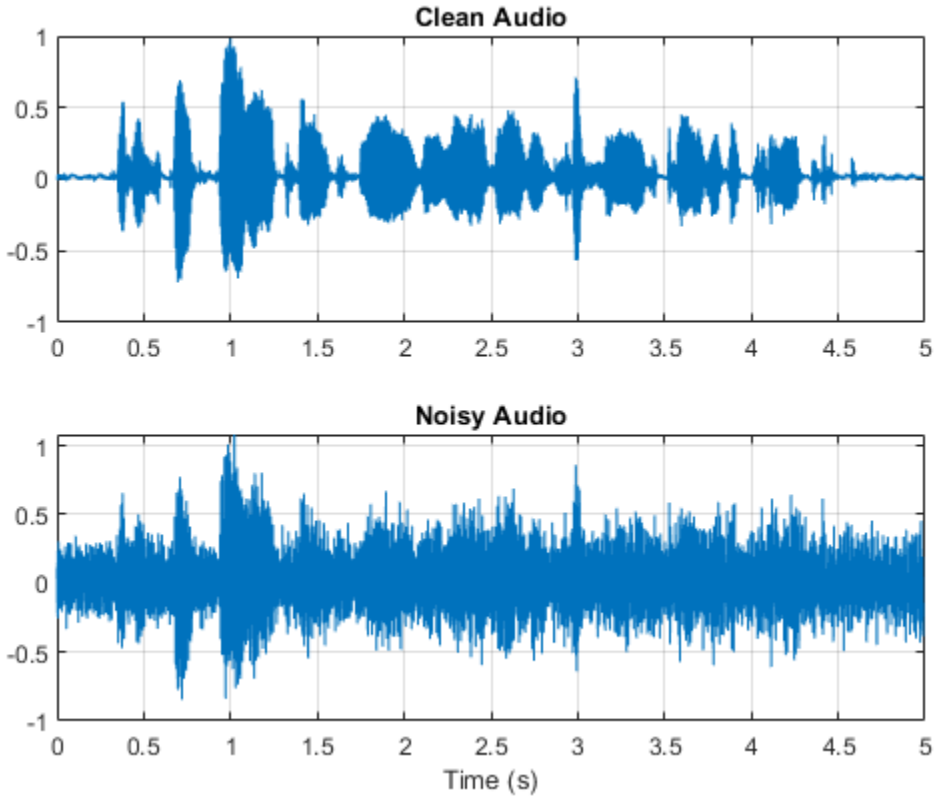
```
noise = audioread("WashingMachine-16-8-mono-1000secs.mp3");

% Extract a noise segment from a random location in the noise file
ind = randi(numel(noise) - numel(cleanAudio) + 1, 1, 1);
noiseSegment = noise(ind:ind + numel(cleanAudio) - 1);

speechPower = sum(cleanAudio.^2);
noisePower = sum(noiseSegment.^2);
noisyAudio = cleanAudio + sqrt(speechPower/noisePower) * noiseSegment;
```

Listen to the noisy speech signal.

```
sound(noisyAudio,fs)
```

Visualize the original and noisy signals.

```
t = (1/fs) * (0:numel(cleanAudio)-1);

subplot(2,1,1)
plot(t,cleanAudio)
title("Clean Audio")
grid on

subplot(2,1,2)
plot(t,noisyAudio)
title("Noisy Audio")
xlabel("Time (s)")
grid on
```



The objective of speech denoising is to remove the washing machine noise from the speech signal while minimizing undesired artifacts in the output speech.

Examine the Dataset

This example uses a subset of the Mozilla Common Voice dataset [1 on page 14-0] to train and test the deep learning networks. The data set contains 48 kHz recordings of subjects speaking short sentences. Download the data set and unzip the downloaded file.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/commonvoice.zip';
downloadFolder = tempdir;
dataFolder = fullfile(downloadFolder,'commonvoice');

if ~exist(dataFolder,'dir')
    disp('Downloading data set (956 MB) ...')
    unzip(url,downloadFolder)
end
```

Use `audioDatastore` to create a datastore for the training set. To speed up the runtime of the example at the cost of performance, set `reduceDataset` to `true`.

```
adsTrain = audioDatastore(fullfile(dataFolder,'train'),'IncludeSubfolders',true);

reduceDataset = true;
if reduceDataset
    adsTrain = shuffle(adsTrain);
```

```
    adsTrain = subset(adsTrain,1:1000);
end
```

Use `read` to get the contents of the first file in the datastore.

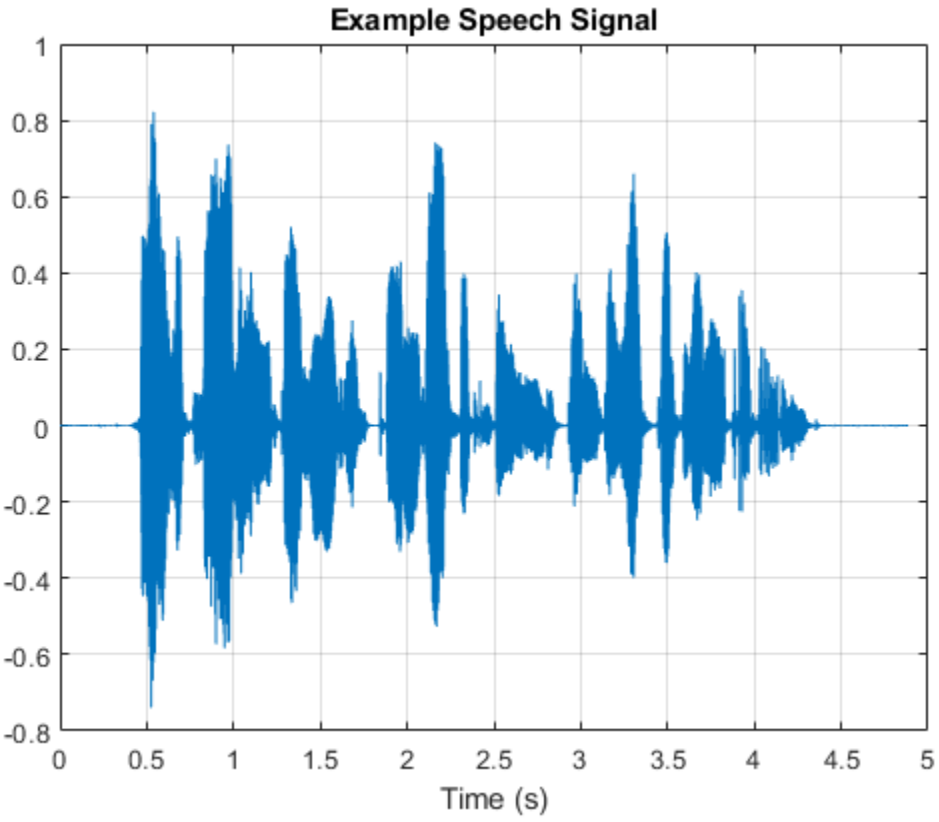
```
[audio,adsTrainInfo] = read(adsTrain);
```

Listen to the speech signal.

```
sound(audio,adsTrainInfo.SampleRate)
```

Plot the speech signal.

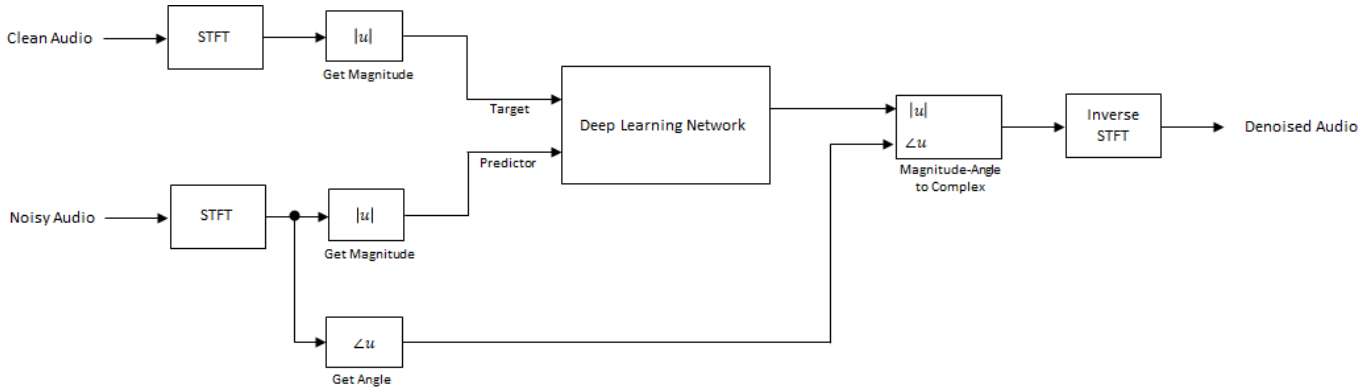
```
figure
t = (1/adsTrainInfo.SampleRate) * (0:numel(audio)-1);
plot(t,audio)
title("Example Speech Signal")
xlabel("Time (s)")
grid on
```



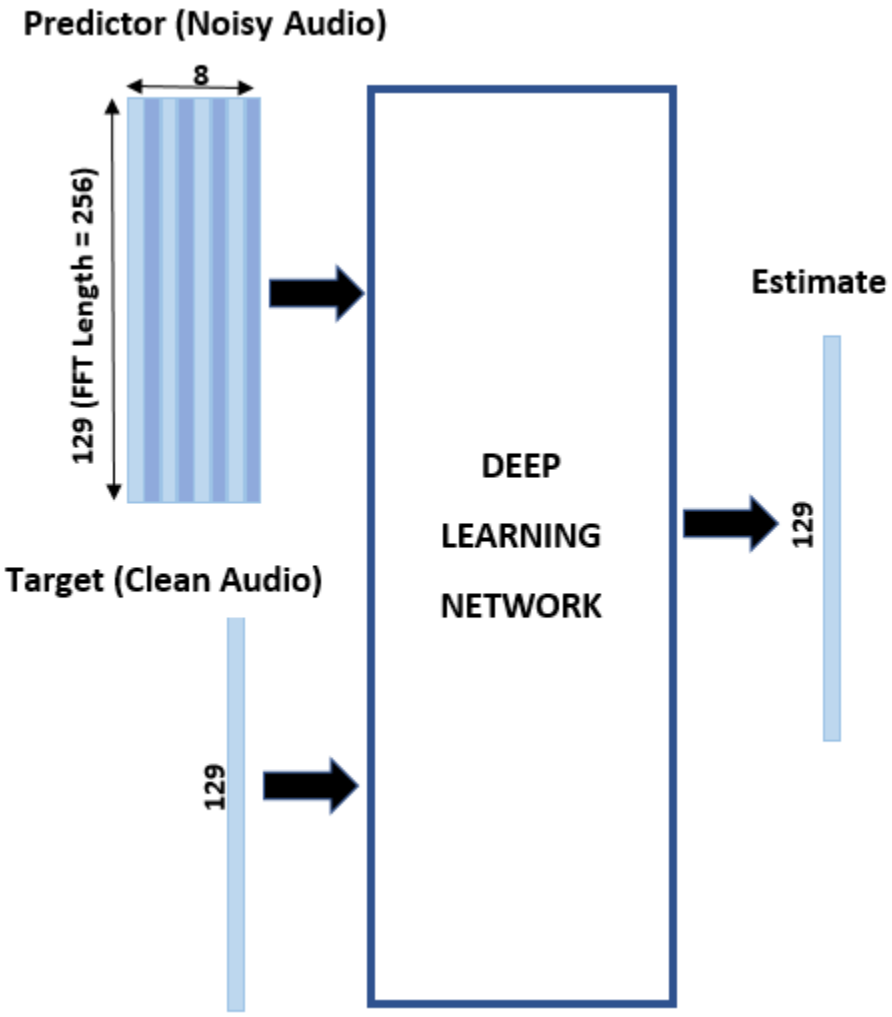
Deep Learning System Overview

The basic deep learning training scheme is shown below. Note that, since speech generally falls below 4 kHz, you first downsample the clean and noisy audio signals to 8 kHz to reduce the computational load of the network. The predictor and target network signals are the magnitude spectra of the noisy and clean audio signals, respectively. The network's output is the magnitude spectrum of the denoised signal. The regression network uses the predictor input to minimize the mean square error between its output and the input target. The denoised audio is converted back to

the time domain using the output magnitude spectrum and the phase of the noisy signal [2 on page 14-0].



You transform the audio to the frequency domain using the Short-Time Fourier transform (STFT), with a window length of 256 samples, an overlap of 75%, and a Hamming window. You reduce the size of the spectral vector to 129 by dropping the frequency samples corresponding to negative frequencies (because the time-domain speech signal is real, this does not lead to any information loss). The predictor input consists of 8 consecutive noisy STFT vectors, so that each STFT output estimate is computed based on the current noisy STFT and the 7 previous noisy STFT vectors.



STFT Targets and Predictors

This section illustrates how to generate the target and predictor signals from one training file.

First, define system parameters:

```
windowLength = 256;
win = hamming(windowLength,"periodic");
overlap = round(0.75 * windowLength);
ffTLength = windowLength;
inputFs = 48e3;
fs = 8e3;
numFeatures = ffTLength/2 + 1;
numSegments = 8;
```

Create a `dsp.SampleRateConverter` (DSP System Toolbox) object to convert the 48 kHz audio to 8 kHz.

```
src = dsp.SampleRateConverter("InputSampleRate",inputFs, ...
                              "OutputSampleRate",fs, ...
                              "Bandwidth",7920);
```

Use `read` to get the contents of an audio file from the datastore.

```
audio = read(adsTrain);
```

Make sure the audio length is a multiple of the sample rate converter decimation factor.

```
decimationFactor = inputFs/fs;
L = floor(numel(audio)/decimationFactor);
audio = audio(1:decimationFactor*L);
```

Convert the audio signal to 8 kHz.

```
audio = src(audio);
reset(src)
```

Create a random noise segment from the washing machine noise vector.

```
randind = randi(numel(noise) - numel(audio),[1 1]);
noiseSegment = noise(randind : randind + numel(audio) - 1);
```

Add noise to the speech signal such that the SNR is 0 dB.

```
noisePower = sum(noiseSegment.^2);
cleanPower = sum(audio.^2);
noiseSegment = noiseSegment .* sqrt(cleanPower/noisePower);
noisyAudio = audio + noiseSegment;
```

Use `stft` (Signal Processing Toolbox) to generate magnitude STFT vectors from the original and noisy audio signals.

```
cleanSTFT = stft(audio,'Window',win,'OverlapLength',overlap,'FFTLength',ffTLength);
cleanSTFT = abs(cleanSTFT(numFeatures-1:end,:));
noisySTFT = stft(noisyAudio,'Window',win,'OverlapLength',overlap,'FFTLength',ffTLength);
noisySTFT = abs(noisySTFT(numFeatures-1:end,:));
```

Generate the 8-segment training predictor signals from the noisy STFT. The overlap between consecutive predictors is 7 segments.

```
noisySTFT = [noisySTFT(:,1:numSegments - 1), noisySTFT];
stftSegments = zeros(numFeatures, numSegments , size(noisySTFT,2) - numSegments + 1);
for index = 1:size(noisySTFT,2) - numSegments + 1
    stftSegments(:,:,index) = (noisySTFT(:,index:index + numSegments - 1));
end
```

Set the targets and predictors. The last dimension of both variables corresponds to the number of distinct predictor/target pairs generated by the audio file. Each predictor is 129-by-8, and each target is 129-by-1.

```
targets = cleanSTFT;
size(targets)
```

```
ans = 1×2

   129   544
```

```
predictors = stftSegments;
size(predictors)
```

```
ans = 1×3

   129     8   544
```

Extract Features Using Tall Arrays

To speed up processing, extract feature sequences from the speech segments of all audio files in the datastore using tall arrays. Unlike in-memory arrays, tall arrays typically remain unevaluated until you call the `gather` function. This deferred evaluation enables you to work quickly with large data sets. When you eventually request output using `gather`, MATLAB combines the queued calculations where possible and takes the minimum number of passes through the data. If you have Parallel Computing Toolbox™, you can use tall arrays in your local MATLAB session, or on a local parallel pool. You can also run tall array calculations on a cluster if you have MATLAB® Parallel Server™ installed.

First, convert the datastore to a tall array.

```
reset(adsTrain)
T = tall(adsTrain)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

T =

  M×1 tall cell array

    {234480×1 double}
    {210288×1 double}
    {282864×1 double}
    {292080×1 double}
    {410736×1 double}
    {303600×1 double}
    {326640×1 double}
    {233328×1 double}
        :        :
        :        :
```

The display indicates that the number of rows (corresponding to the number of files in the datastore), M, is not yet known. M is a placeholder until the calculation completes.

Extract the target and predictor magnitude STFT from the tall table. This action creates new tall array variables to use in subsequent calculations. The function `HelperGenerateSpeechDenoisingFeatures` performs the steps already highlighted in the STFT Targets and Predictors on page 14-0 section. The `cellfun` command applies `HelperGenerateSpeechDenoisingFeatures` to the contents of each audio file in the datastore.

```
[targets,predictors] = cellfun(@(x)HelperGenerateSpeechDenoisingFeatures(x,noise,src),T,"Uniform(
```

Use `gather` to evaluate the targets and predictors.

```
[targets,predictors] = gather(targets,predictors);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 42 sec
Evaluation completed in 1 min 36 sec
```

It is good practice to normalize all features to zero mean and unity standard deviation.

Compute the mean and standard deviation of the predictors and targets, respectively, and use them to normalize the data.

```
predictors = cat(3,predictors{:});
noisyMean = mean(predictors(:));
noisyStd = std(predictors(:));
predictors(:) = (predictors(:) - noisyMean)/noisyStd;

targets = cat(2,targets{:});
cleanMean = mean(targets(:));
cleanStd = std(targets(:));
targets(:) = (targets(:) - cleanMean)/cleanStd;
```

Reshape predictors and targets to the dimensions expected by the deep learning networks.

```
predictors = reshape(predictors,size(predictors,1),size(predictors,2),1,size(predictors,3));
targets = reshape(targets,1,1,size(targets,1),size(targets,2));
```

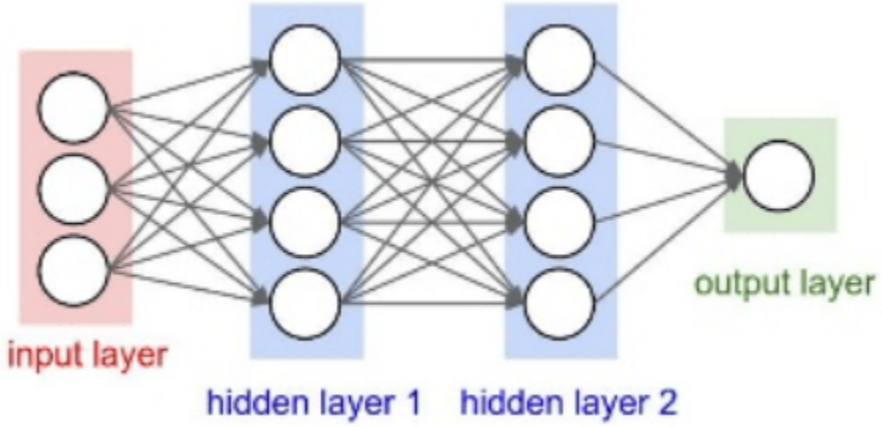
You will use 1% of the data for validation during training. Validation is useful to detect scenarios where the network is overfitting the training data.

Randomly split the data into training and validation sets.

```
inds = randperm(size(predictors,4));
L = round(0.99 * size(predictors,4));

trainPredictors = predictors(:,:,:,inds(1:L));
trainTargets = targets(:,:,:,inds(1:L));

validatePredictors = predictors(:,:,:,inds(L+1:end));
validateTargets = targets(:,:,:,inds(L+1:end));
```

Speech Denoising with Fully Connected Layers

You first consider a denoising network comprised of fully connected layers. Each neuron in a fully connected layer is connected to all activations from the previous layer. A fully connected layer multiplies the input by a weight matrix and then adds a bias vector. The dimensions of the weight matrix and bias vector are determined by the number of neurons in the layer and the number of activations from the previous layer.



Define the layers of the network. Specify the input size to be images of size `NumFeatures`-by-`NumSegments` (129-by-8 in this example). Define two hidden fully connected layers, each with 1024 neurons. Since purely linear systems, follow each hidden fully connected layer with a Rectified Linear Unit (ReLU) layer. The batch normalization layers normalize the means and standard deviations of the outputs. Add a fully connected layer with 129 neurons, followed by a regression layer.

```
layers = [
    imageInputLayer([numFeatures,numSegments])
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(1024)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(numFeatures)
    regressionLayer
    ];
```

Next, specify the training options for the network. Set `MaxEpochs` to `3` so that the network makes 3 passes through the training data. Set `MiniBatchSize` of `128` so that the network looks at 128 training signals at a time. Specify `Plots` as `"training-progress"` to generate plots that show the training progress as the number of iterations increases. Set `Verbose` to `false` to disable printing the table output that corresponds to the data shown in the plot into the command line window. Specify `Shuffle` as `"every-epoch"` to shuffle the training sequences at the beginning of each epoch. Specify `LearnRateSchedule` to `"piecewise"` to decrease the learning rate by a specified factor (0.9) every time a certain number of epochs (1) has passed. Set `ValidationData` to the validation predictors and targets. Set `ValidationFrequency` such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (Adam) solver.

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5,...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,validateTargets});
```

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To download and load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = true;
if doTraining
    denoiseNetFullyConnected = trainNetwork(trainPredictors,trainTargets,layers,options);
else
    url = 'http://ssd.mathworks.com/supportfiles/audio/SpeechDenoising.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'SpeechDenoising');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 8 MB) ...')
```

```
        unzip(url,downloadNetFolder)
    end
    s = load(fullfile(netFolder,"denoisenet.mat"));
    denoiseNetFullyConnected = s.denoiseNetFullyConnected;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
    noisyStd = s.noisyStd;
end
```

Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConnected.Layers)
    if isa(denoiseNetFullyConnected.Layers(index),"nnet.cnn.layer.FullyConnectedLayer")
        numWeights = numWeights + numel(denoiseNetFullyConnected.Layers(index).Weights);
    end
end
fprintf("The number of weights is %d.\n",numWeights);
```

```
The number of weights is 2237440.
```

Speech Denoising with Convolutional Layers

Consider a network that uses convolutional layers instead of fully connected layers [3 on page 14-0]. A 2-D convolutional layer applies sliding filters to the input. The layer convolves the input by moving the filters along the input vertically and horizontally and computing the dot product of the weights and the input, and then adding a bias term. Convolutional layers typically consist of fewer parameters than fully connected layers.

Define the layers of the fully convolutional network described in [3 on page 14-0], comprising 16 convolutional layers. The first 15 convolutional layers are groups of 3 layers, repeated 5 times, with filter widths of 9, 5, and 9, and number of filters of 18, 30 and 8, respectively. The last convolutional layer has a filter width of 129 and 1 filter. In this network, convolutions are performed in only one direction (along the frequency dimension), and the filter width along the time dimension is set to 1 for all layers except the first one. Similar to the fully connected network, convolutional layers are followed by ReLu and batch normalization layers.

```
layers = [imageInputLayer([numFeatures,numSegments])
          convolution2dLayer([9 8],18,"Stride",[1 100],"Padding","same")
          batchNormalizationLayer
          reluLayer

          repmat( ...
          [convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
          batchNormalizationLayer
          reluLayer
          convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
          batchNormalizationLayer
          reluLayer
          convolution2dLayer([9 1],18,"Stride",[1 100],"Padding","same")
          batchNormalizationLayer
          reluLayer],4,1)

          convolution2dLayer([5 1],30,"Stride",[1 100],"Padding","same")
          batchNormalizationLayer
          reluLayer
```

```
    convolution2dLayer([9 1],8,"Stride",[1 100],"Padding","same")
    batchNormalizationLayer
    reluLayer

    convolution2dLayer([129 1],1,"Stride",[1 100],"Padding","same")

    regressionLayer
    ];
```

The training options are identical to the options for the fully connected network, except that the dimensions of the validation target signals are permuted to be consistent with the dimensions expected by the regression layer.

```
options = trainingOptions("adam", ...
    "MaxEpochs",3, ...
    "InitialLearnRate",1e-5, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(size(trainPredictors,4)/miniBatchSize), ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.9, ...
    "LearnRateDropPeriod",1, ...
    "ValidationData",{validatePredictors,permute(validateTargets,[3 1 2 4])});
```

Train the network with the specified training options and layer architecture using `trainNetwork`. Because the training set is large, the training process can take several minutes. To download and load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = true;
if doTraining
    denoiseNetFullyConvolutional = trainNetwork(trainPredictors,permute(trainTargets,[3 1 2 4]),
else
    url = 'http://ssd.mathworks.com/supportfiles/audio/SpeechDenoising.zip';
    downloadNetFolder = tempdir;
    netFolder = fullfile(downloadNetFolder,'SpeechDenoising');
    if ~exist(netFolder,'dir')
        disp('Downloading pretrained network (1 file - 8 MB) ...')
        unzip(url,downloadNetFolder)
    end
    s = load(fullfile(netFolder,"denoisenet.mat"));
    denoiseNetFullyConvolutional = s.denoiseNetFullyConvolutional;
    cleanMean = s.cleanMean;
    cleanStd = s.cleanStd;
    noisyMean = s.noisyMean;
    noisyStd = s.noisyStd;
end
```

Count the number of weights in the fully connected layers of the network.

```
numWeights = 0;
for index = 1:numel(denoiseNetFullyConvolutional.Layers)
    if isa(denoiseNetFullyConvolutional.Layers(index),"nnet.cnn.layer.Convolution2DLayer")
        numWeights = numWeights + numel(denoiseNetFullyConvolutional.Layers(index).Weights);
    end
end
fprintf("The number of weights in convolutional layers is %d\n",numWeights);
```

```
The number of weights in convolutional layers is 31812
```

Test the Denoising Networks

Read in the test data set.

```
adsTest = audioDatastore(fullfile(dataFolder,'test'),'IncludeSubfolders',true);
```

Read the contents of a file from the datastore.

```
[cleanAudio,adsTestInfo] = read(adsTest);
```

Make sure the audio length is a multiple of the sample rate converter decimation factor.

```
L = floor(numel(cleanAudio)/decimationFactor);
cleanAudio = cleanAudio(1:decimationFactor*L);
```

Convert the audio signal to 8 kHz.

```
cleanAudio = src(cleanAudio);
reset(src)
```

In this testing stage, you corrupt speech with washing machine noise not used in the training stage.

```
noise = audioread("WashingMachine-16-8-mono-200secs.mp3");
```

Create a random noise segment from the washing machine noise vector.

```
randind = randi(numel(noise) - numel(cleanAudio), [1 1]);
noiseSegment = noise(randind : randind + numel(cleanAudio) - 1);
```

Add noise to the speech signal such that the SNR is 0 dB.

```
noisePower = sum(noiseSegment.^2);
cleanPower = sum(cleanAudio.^2);
noiseSegment = noiseSegment .* sqrt(cleanPower/noisePower);
noisyAudio = cleanAudio + noiseSegment;
```

Use `stft` to generate magnitude STFT vectors from the noisy audio signals.

```
noisySTFT = stft(noisyAudio,'Window',win,'OverlapLength',overlap,'FFTLength',ffTLength);
noisyPhase = angle(noisySTFT(numFeatures-1:end,:));
noisySTFT = abs(noisySTFT(numFeatures-1:end,:));
```

Generate the 8-segment training predictor signals from the noisy STFT. The overlap between consecutive predictors is 7 segments.

```
noisySTFT = [noisySTFT(:,1:numSegments-1) noisySTFT];
predictors = zeros( numFeatures, numSegments , size(noisySTFT,2) - numSegments + 1);
for index = 1:(size(noisySTFT,2) - numSegments + 1)
    predictors(:,:,index) = noisySTFT(:,index:index + numSegments - 1);
end
```

Normalize the predictors by the mean and standard deviation computed in the training stage.

```
predictors(:) = (predictors(:) - noisyMean) / noisyStd;
```

Compute the denoised magnitude STFT by using `predict` with the two trained networks.

```
predictors = reshape(predictors, [numFeatures,numSegments,1,size(predictors,3)]);
STFTFullyConnected = predict(denoiseNetFullyConnected, predictors);
STFTFullyConvolutional = predict(denoiseNetFullyConvolutional, predictors);
```

Scale the outputs by the mean and standard deviation used in the training stage.

```
STFTFullyConnected(:) = cleanStd * STFTFullyConnected(:) + cleanMean;
STFTFullyConvolutional(:) = cleanStd * STFTFullyConvolutional(:) + cleanMean;
```

Convert the one-sided STFT to a centered STFT.

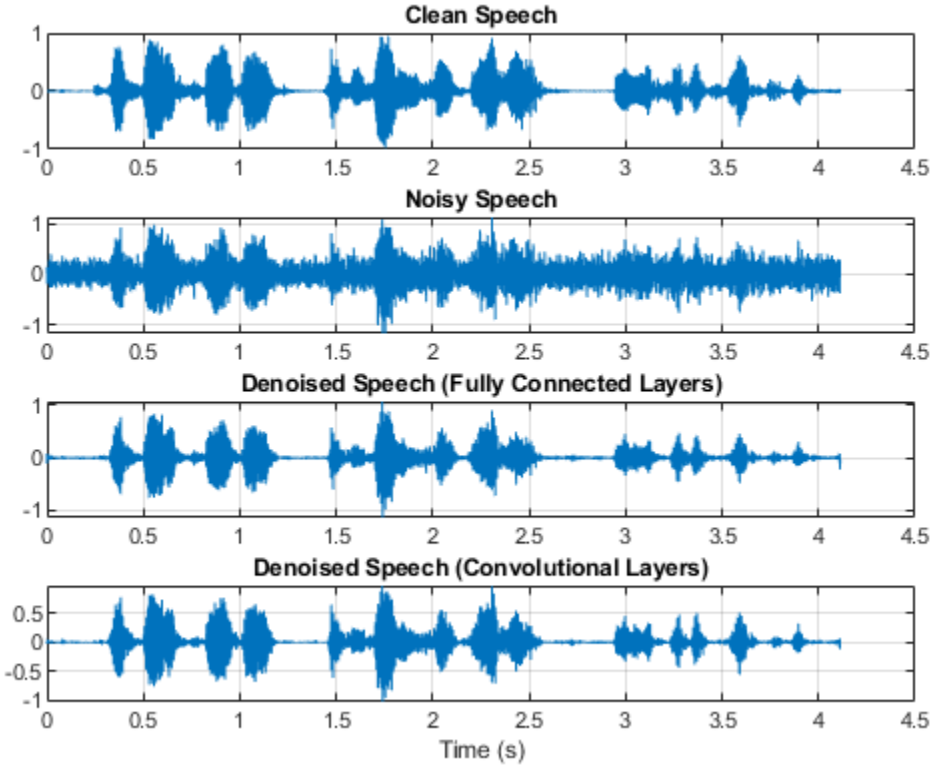
```
STFTFullyConnected = STFTFullyConnected.' .* exp(1j*noisyPhase);
STFTFullyConnected = [conj(STFTFullyConnected(end-1:-1:2,:)); STFTFullyConnected];
STFTFullyConvolutional = squeeze(STFTFullyConvolutional) .* exp(1j*noisyPhase);
STFTFullyConvolutional = [conj(STFTFullyConvolutional(end-1:-1:2,:)) ; STFTFullyConvolutional];
```

Compute the denoised speech signals. `istft` performs the inverse STFT. Use the phase of the noisy STFT vectors to reconstruct the time-domain signal.

```
denoisedAudioFullyConnected = istft(STFTFullyConnected,  ...
                                    'Window',win,'OverlapLength',overlap, ...
                                    'FFTLength',ffTLength,'ConjugateSymmetric',true);

denoisedAudioFullyConvolutional = istft(STFTFullyConvolutional,  ...
                                        'Window',win,'OverlapLength',overlap, ...
                                        'FFTLength',ffTLength,'ConjugateSymmetric',true);
```

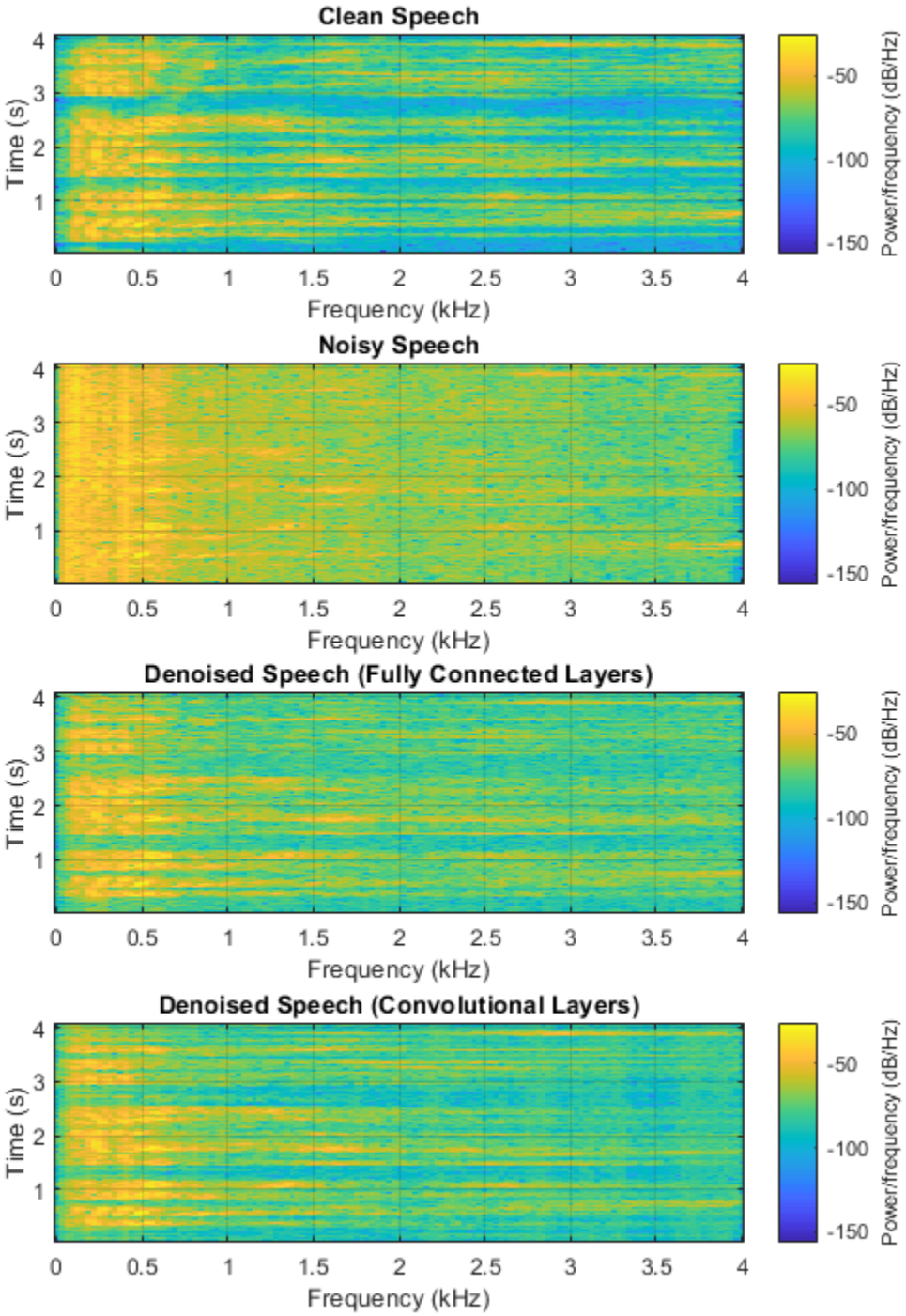
Plot the clean, noisy and denoised audio signals.

```
t = (1/fs) * (0:numel(denoisedAudioFullyConnected)-1);

figure

subplot(4,1,1)
plot(t,cleanAudio(1:numel(denoisedAudioFullyConnected)))
title("Clean Speech")
grid on

subplot(4,1,2)
plot(t,noisyAudio(1:numel(denoisedAudioFullyConnected)))
title("Noisy Speech")
grid on

subplot(4,1,3)
plot(t,denoisedAudioFullyConnected)
title("Denoised Speech (Fully Connected Layers)")
grid on

subplot(4,1,4)
plot(t,denoisedAudioFullyConvolutional)
title("Denoised Speech (Convolutional Layers)")
grid on
xlabel("Time (s)")
```



Plot the clean, noisy, and denoised spectrograms.

```
h = figure;

subplot(4,1,1)
spectrogram(cleanAudio,win,overlap,ffTLength,fs);
title("Clean Speech")
grid on

subplot(4,1,2)
spectrogram(noisyAudio,win,overlap,ffTLength,fs);
title("Noisy Speech")
grid on

subplot(4,1,3)
spectrogram(denoisedAudioFullyConnected,win,overlap,ffTLength,fs);
title("Denoised Speech (Fully Connected Layers)")
grid on

subplot(4,1,4)
spectrogram(denoisedAudioFullyConvolutional,win,overlap,ffTLength,fs);
title("Denoised Speech (Convolutional Layers)")
grid on

p = get(h,'Position');
set(h,'Position',[p(1) 65 p(3) 800]);
```



Listen to the noisy speech.

```
sound(noisyAudio,fs)
```

Listen to the denoised speech from the network with fully connected layers.

```
sound(denoisedAudioFullyConnected,fs)
```

Listen to the denoised speech from the network with convolutional layers.

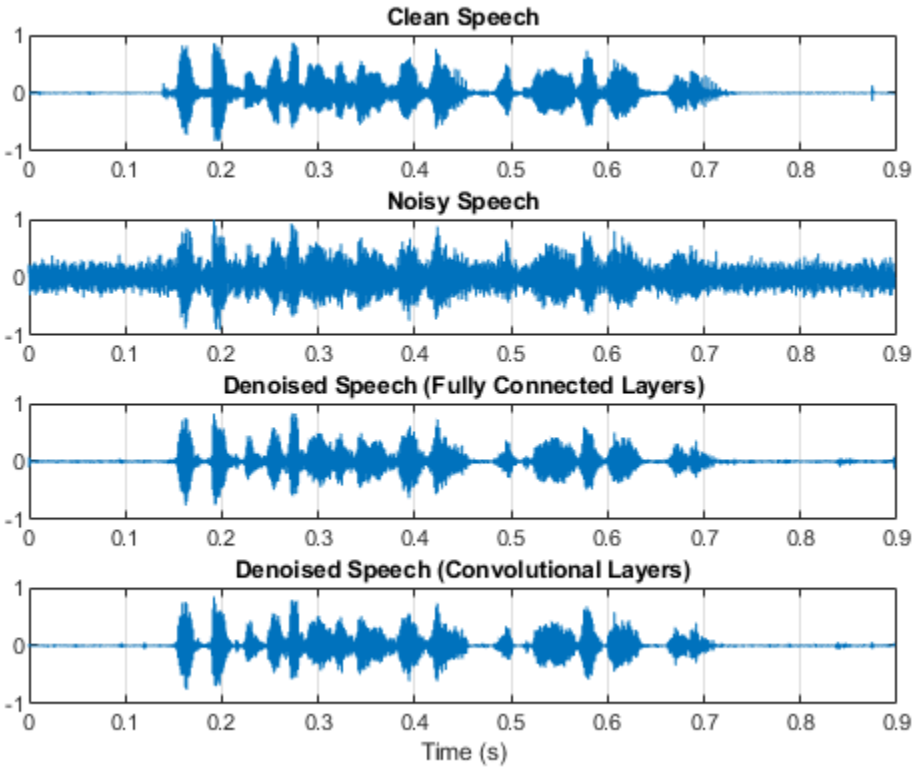
```
sound(denoisedAudioFullyConvolutional,fs)
```

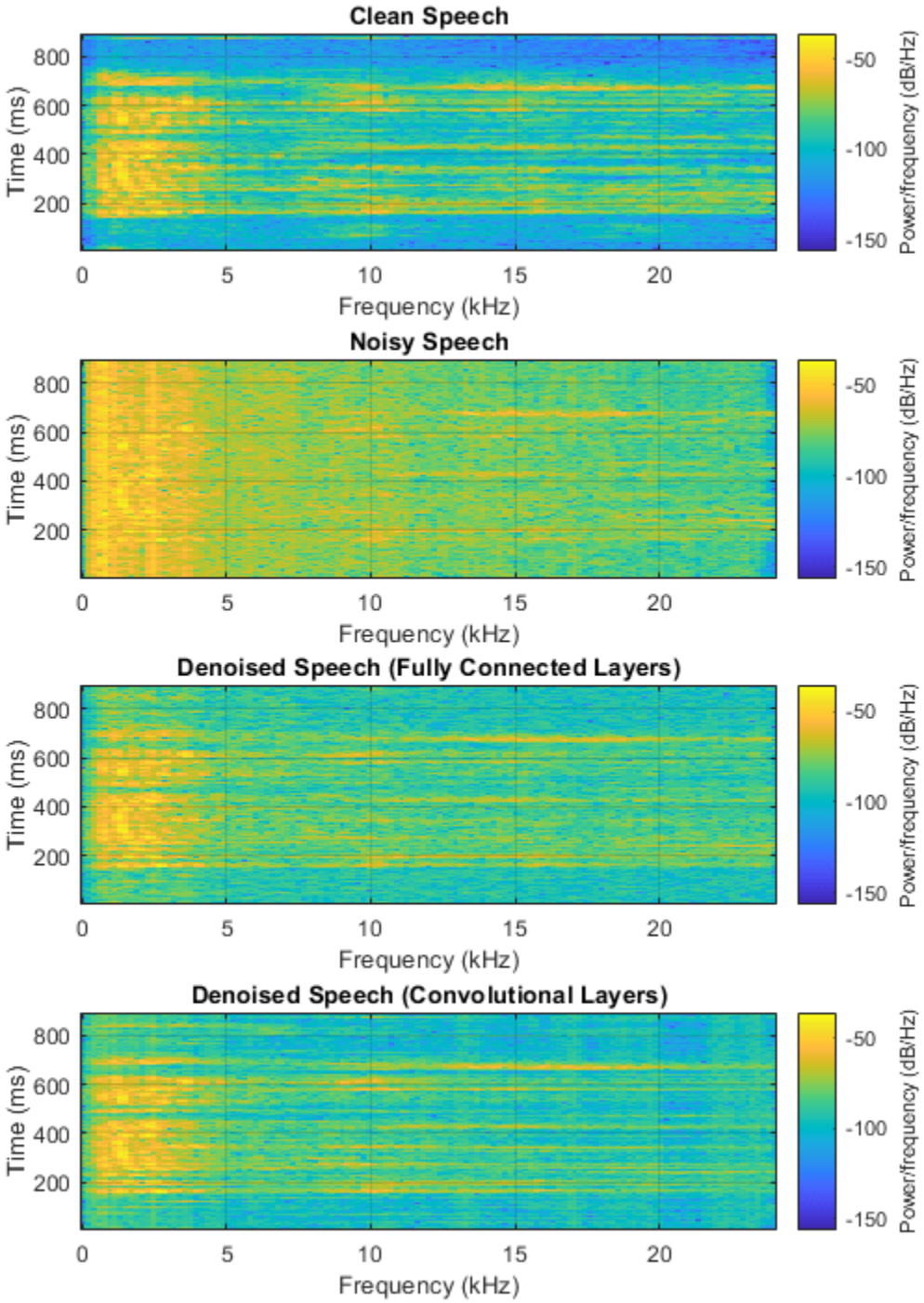
Listen to clean speech.

```
sound(cleanAudio,fs)
```

You can test more files from the datastore by calling `testDenoisingNets`. The function produces the time-domain and frequency-domain plots highlighted above, and also returns the clean, noisy, and denoised audio signals.

```
[cleanAudio,noisyAudio,denoisedAudioFullyConnected,denoisedAudioFullyConvolutional] = testDenois
```



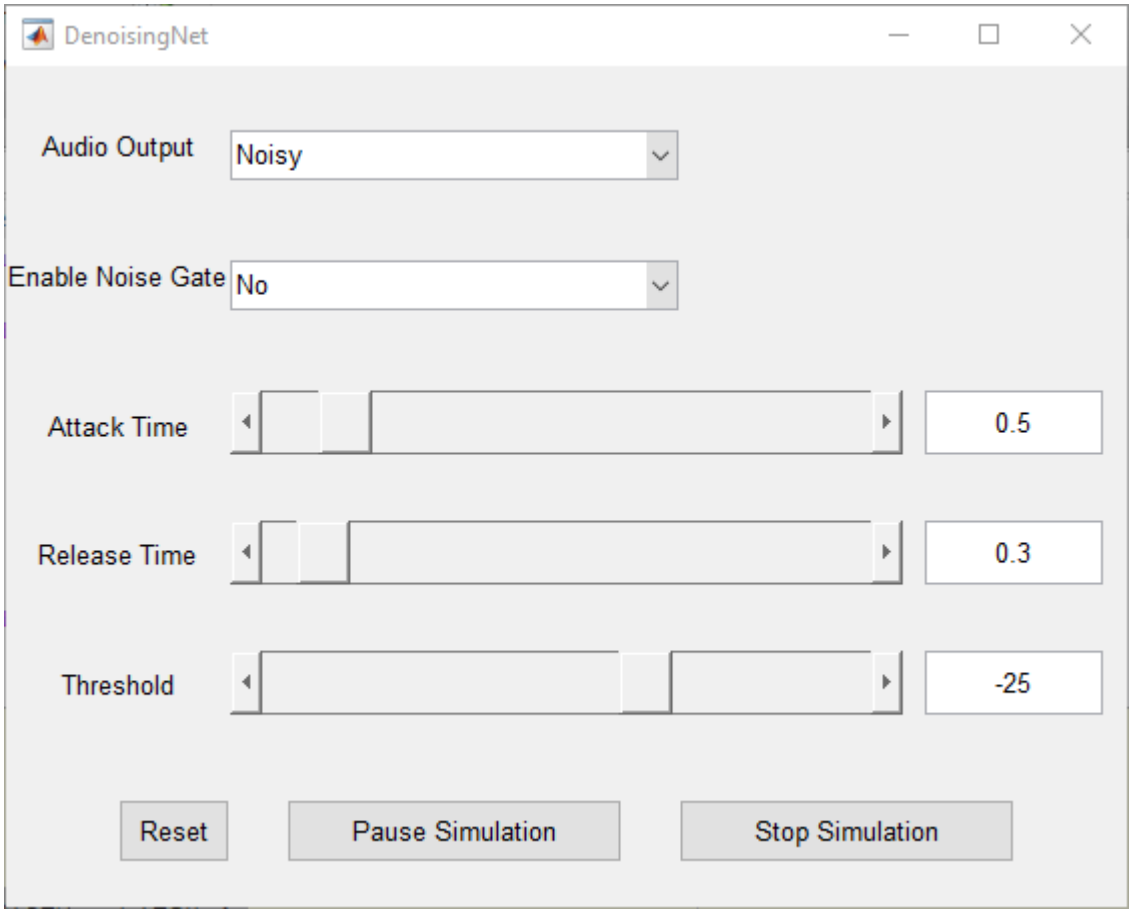

Clean Speech
Noisy Speech
Denoised Speech (Fully Connected Layers)
Denoised Speech (Convolutional Layers)
Time (ms)
800
600
400
200
0
5
10
15
20
Frequency (kHz)
-50
-100
-150
Power/frequency (dB/Hz)

Real-Time Application

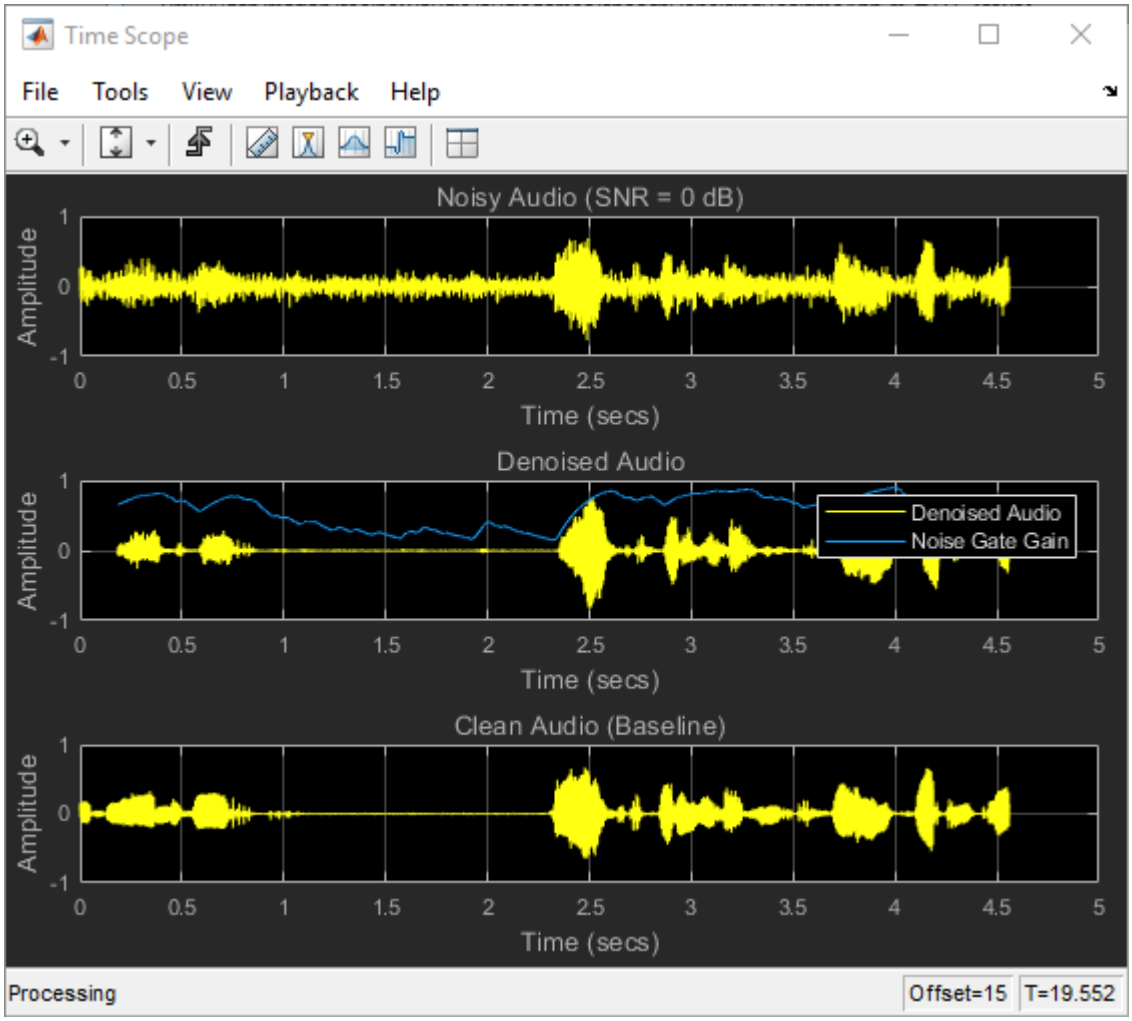
The procedure in the previous section passes the entire spectrum of the noisy signal to `predict`. This is not suitable for real-time applications where low latency is a requirement.

Run `speechDenoisingRealtimeApp` for an example of how to simulate a streaming, real-time version of the denoising network. The app uses the network with fully connected layers. The audio frame length is equal to the STFT hop size, which is 0.25 * 256 = 64 samples.

`speechDenoisingRealtimeApp` launches a User Interface (UI) designed to interact with the simulation. The UI enables you to tune parameters and the results are reflected in the simulation instantly. You can also enable/disable a noise gate that operates on the denoised output to further reduce the noise, as well as tune the attack time, release time, and threshold of the noise gate. You can listen to the noisy, clean or denoised audio from the UI.



The scope plots the clean, noisy and denoised signals, as well as the gain of the noise gate.



References

[1] https://voice.mozilla.org/en

[2] "Experiments on Deep Learning for Speech Denoising", Ding Liu, Paris Smaragdis, Minje Kim, INTERSPEECH, 2014.

[3] "A Fully Convolutional Neural Network for Speech Enhancement", Se Rim Park, Jin Won Lee, INTERSPEECH, 2017.

See Also

Functions

`trainingOptions` | `trainNetwork`

More About

- “Deep Learning in MATLAB” on page 1-2

Accelerate Audio Deep Learning Using GPU-Based Feature Extraction

In this example, you leverage GPUs for feature extraction and augmentation to decrease the time required to train a deep learning model. The model you train is a convolutional neural network (CNN) for acoustic fault recognition.

Audio Toolbox™ includes `gpuArray` (Parallel Computing Toolbox) support for most feature extractors, including popular ones such as `melSpectrogram` (Audio Toolbox) and `mfcc` (Audio Toolbox). For an overview of GPU support, see “Code Generation and GPU Support” (Audio Toolbox).

Load Training Data

Download and unzip the air compressor data set [1] on page 14-0 . This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

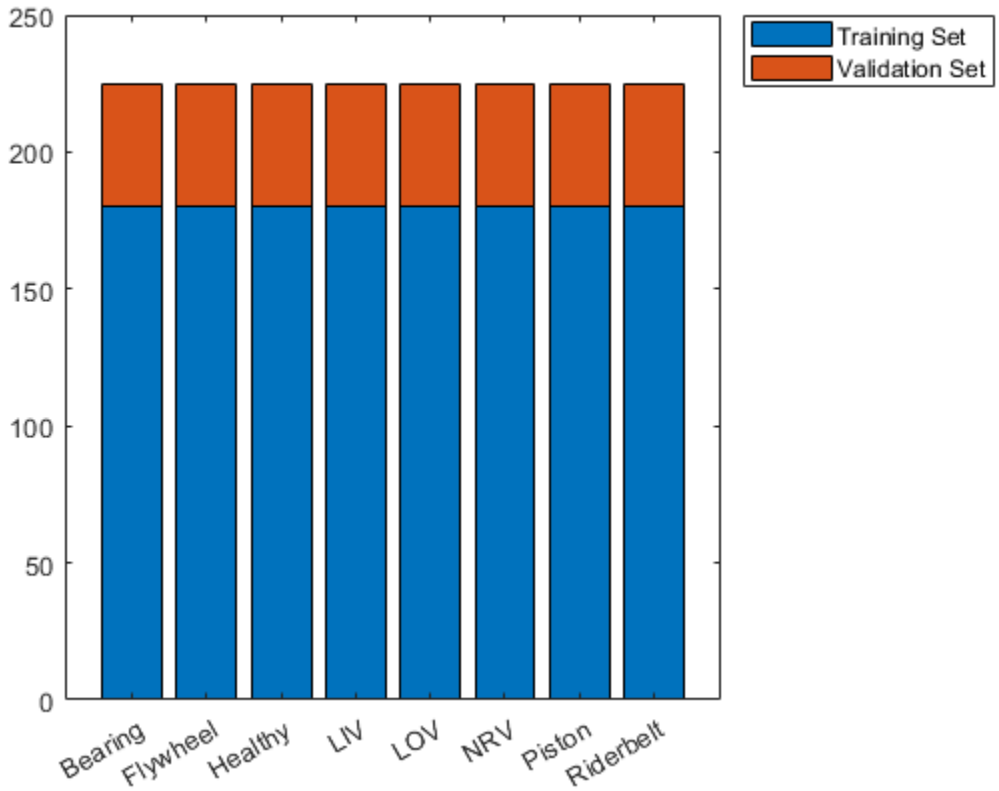
```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'AirCompressorDataset');
datasetLocation = tempdir;

if ~isfolder(fullfile(datasetLocation,'AirCompressorDataset'))
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(datasetLocation,'AirCompressorDataset'))
end
```

Create an `audioDatastore` (Audio Toolbox) object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');
rng default
[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

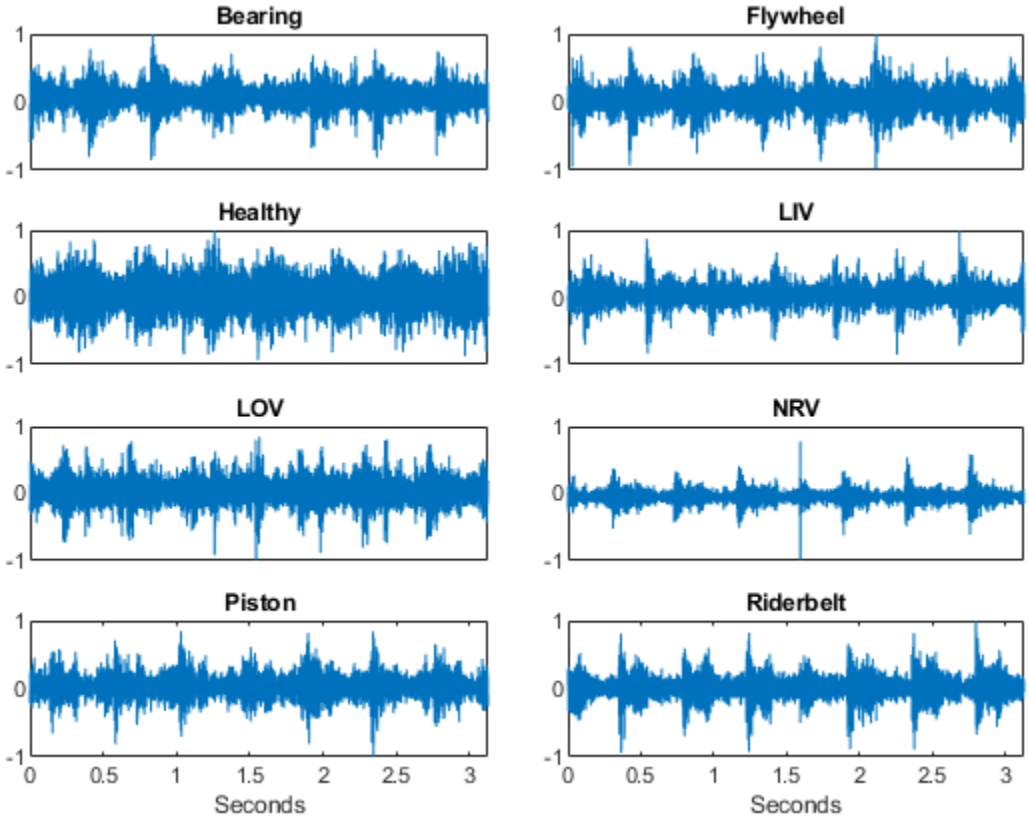
Visualize the number of files in the training and validation sets.

```
uniqueLabels = unique(adsTrain.Labels);
tblTrain = countEachLabel(adsTrain);
tblValidation = countEachLabel(adsValidation);
H = bar(uniqueLabels,[tblTrain.Count, tblValidation.Count],'stacked');
legend(H,["Training Set","Validation Set"],'Location','NorthEastOutside')
```



Select random examples from the training set for plotting. Each recording has 50,000 samples sampled at 16 kHz.

```
t = (0:5e4-1)/16e3;
tiledlayout(4,2,'TileSpacing','compact','Padding','compact')
for n = 1:numel(uniqueLabels)
    idx = find(adsTrain.Labels==uniqueLabels(n));
    [x,fs] = audioread(adsTrain.Files{idx(randperm(numel(idx),1))});

    nexttile
    plotHandle = plot(t,x);
    if n == 7 || n == 8
        xlabel('Seconds');
    else
        set(gca,'xtick',[])
    end
    title(string(uniqueLabels(n)));
end
```



Preprocess Data on CPU and GPU

In this example, you perform feature extraction and data augmentation while training the network. In this section, you define the feature extraction and augmentation pipeline and compare the speed of the pipeline executed on a CPU against the speed of the pipeline executed on a GPU. The output of this pipeline is the input to the CNN you train.

Create an `audioFeatureExtractor` (Audio Toolbox) object to extract mel spectrums using 200 ms mel windows with a 5 ms hop. The output from `extract` is a `numHops`-by-128-by-1 array.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
    'FFTLength',4096, ...
    'Window',hann(round(fs*0.2),'periodic'), ...
    'OverlapLength',round(fs*0.195), ...
    'melSpectrum',true);
setExtractorParams(afe,"melSpectrum",'NumBands',128);

featureVector = extract(afe,x);
[numHops,numFeatures,numChannels] = size(featureVector)
```

```
numHops = 586
```

```
numFeatures = 128
```

```
numChannels = 1
```

Deep learning methods are data-hungry, and the training dataset in this example is relatively small. Use the mixup [2] on page 14-0 augmentation technique to effectively enlarge the training set. In

mixup, you merge the features extracted from two audio signals as a weighted sum. The two signals have different labels, and the label assigned to the merged feature matrix is probabilistically assigned based on the mixing coefficient. The mixup augmentation is implemented in the supporting object, `Mixup` on page 14-0 .

Create the pipeline to perform the following steps:

1. Extract the log-mel spectrogram.
2. Apply mixup to the feature matrices. The `Mixup` supporting object outputs a cell array containing the features and the label.

Create two versions of the pipeline for comparison: one that executes the pipeline on your CPU, and one that converts the raw audio signal to a `gpuArray` so that the pipeline is executed on your GPU.

```
offset = eps;

adsTrainCPU = transform(adsTrain,@(x)log10(extract(afe,x)+offset));
mixerCPU = Mixup(adsTrainCPU);
adsTrainCPU = transform(adsTrainCPU,@(x,info)mix(mixerCPU,x,info),'IncludeInfo',true);

adsTrainGPU = transform(adsTrain,@gpuArray);
adsTrainGPU = transform(adsTrainGPU,@(x)log10(extract(afe,x)+offset));
mixerGPU = Mixup(adsTrainGPU);
adsTrainGPU = transform(adsTrainGPU,@(x,info)mix(mixerGPU,x,info),'IncludeInfo',true);
```

For the validation set, apply the feature extraction pipeline but not the augmentation. Because you are not applying mixup, create a combined datastore to output a cell array containing the features and the label. Again, create one validation pipeline that executes on your GPU and one validation pipeline that executes on your CPU.

```
adsValidationGPU = transform(adsValidation,@gpuArray);
adsValidationGPU = transform(adsValidationGPU,@(x){log10(extract(afe,x)+offset)});
adsValidationGPU = combine(adsValidationGPU,arrayDatastore(adsValidation.Labels));

adsValidationCPU = transform(adsValidation,@(x){log10(extract(afe,x)+offset)});
adsValidationCPU = combine(adsValidationCPU,arrayDatastore(adsValidation.Labels));
```

Compare the time it takes for the CPU and a single GPU to extract features and perform data augmentation.

```
tic
for ii = 1:numel(adsTrain.Files)
    x = read(adsTrainCPU);
end
cpuPipeline = toc;
reset(adsTrainCPU)

tic
for ii = 1:numel(adsTrain.Files)
    x = read(adsTrainGPU);
end
wait(gpuDevice) % Ensure all calculations are completed
gpuPipeline = toc;
reset(adsTrainGPU)

fprintf(['Read, extract, and augment train set (CPU): %0.2f seconds\n' ...
    'Read, extract, and augment train set (GPU): %0.2f seconds\n' ...
```

```
    'Speedup (CPU time)/(GPU time): %0.3f\n\n'], ...
    cpuPipeline,gpuPipeline,cpuPipeline/gpuPipeline)
```

```
Read, extract, and augment train set (CPU): 110.80 seconds
Read, extract, and augment train set (GPU): 34.65 seconds
Speedup (CPU time)/(GPU time): 3.198
```

Reading from the datastore contributes a significant amount of the overall time to the pipeline. A comparison of just extraction and augmentation shows an even greater speedup. Compare just feature extraction on the GPU versus on the CPU.

```
x = read(ads);

extract(afe,x); % Incur initialization cost outside timing loop
tic
for ii = 1:numel(adsTrain.Files)
    features = log10(extract(afe,x)+offset);
end
cpuFeatureExtraction = toc;

x = gpuArray(x); % Incur initialization cost outside timing loop
extract(afe,x);
tic
for ii = 1:numel(adsTrain.Files)
    features = log10(extract(afe,x)+offset);
end
wait(gpuDevice) % Ensure all calculations are completed
gpuFeatureExtraction = toc;

fprintf(['Extract features from train set (CPU): %0.2f seconds\n' ...
    'Extract features from train set (GPU): %0.2f seconds\n' ...
    'Speedup (CPU time)/(GPU time): %0.3f\n\n'], ...
    cpuFeatureExtraction,gpuFeatureExtraction,cpuFeatureExtraction/gpuFeatureExtraction)
```

```
Extract features from train set (CPU): 50.57 seconds
Extract features from train set (GPU): 4.48 seconds
Speedup (CPU time)/(GPU time): 11.299
```

Define Network

Define a convolutional neural network that takes the augmented mel spectrogram as input. This network applies a single convolutional layer consisting of 48 filters with 3-by-3 kernels, followed by a batch normalization layer and a ReLU activation layer. The time dimension is then collapsed using a max pooling layer. Finally, the output of the pooling layer is reduced using a fully connected layer followed by softmax and classification layers. See “List of Deep Learning Layers” on page 1-21 for more information.

```
numClasses = numel(categories(adsTrain.Labels));
imageSize = [numHops,afe.FeatureVectorLength];

layers = [
    imageInputLayer(imageSize,'Normalization','none')

    convolution2dLayer(3,48,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer([numHops,1])
```

```
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer('Classes',categories(adsTrain.Labels));
    ];
```

To define the training options, use `trainingOptions`. Set the `ExecutionEnvironment` to `multi-gpu` to leverage multiple GPUs, if available. Otherwise, you can set `ExecutionEnvironment` to `gpu`. The computer used in this example has access to four Titan V GPU devices. In this example, the network training always leverages GPUs.

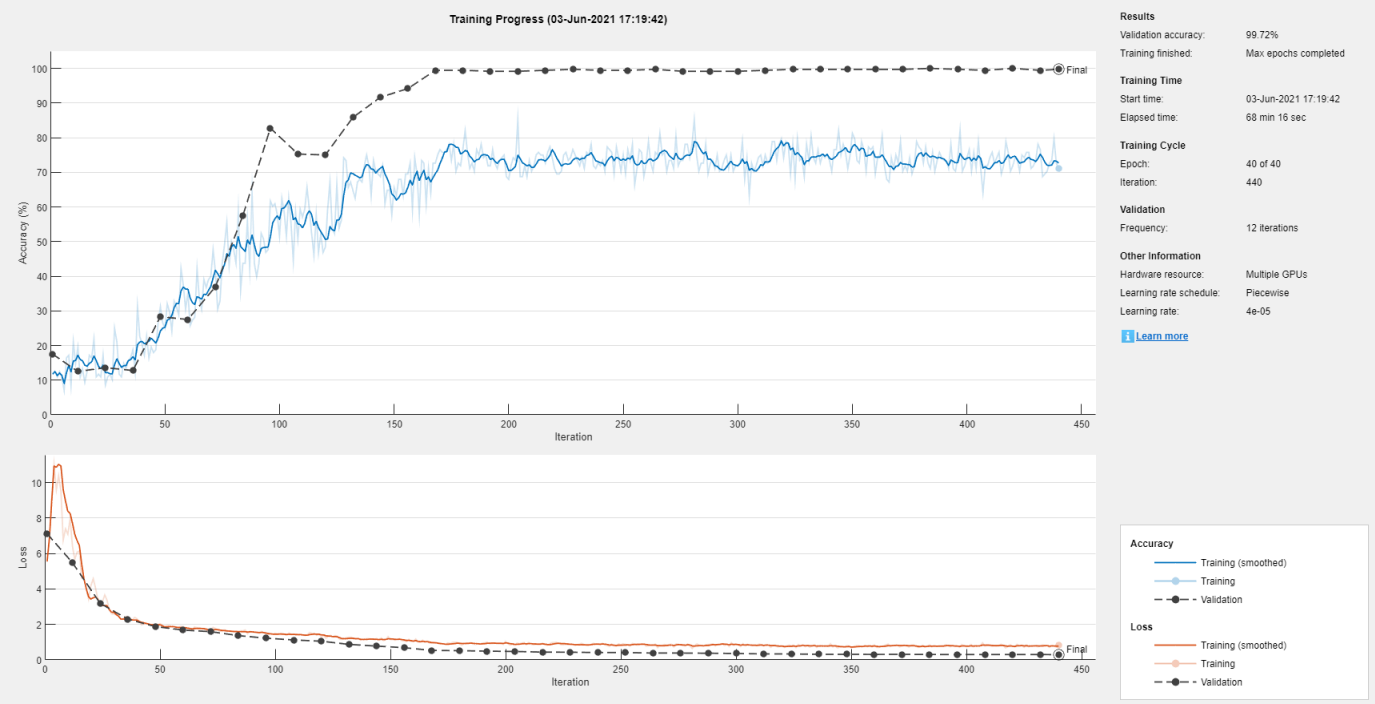
```
miniBatchSize = 128;
options = trainingOptions('adam', ...
    'Shuffle','every-epoch', ...
    'MaxEpochs',40, ...
    'LearnRateSchedule',"piecewise", ...
    'LearnRateDropPeriod',15, ...
    'LearnRateDropFactor',0.2, ...
    'MiniBatchSize',miniBatchSize, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',adsValidationCPU, ...
    'ValidationFrequency',ceil(numel(adsTrain.Files)/miniBatchSize), ...
    'ExecutionEnvironment','multi-gpu');
```

Train Network

Train Network Using CPU-Based Preprocessing

Call `trainNetwork` to train the network using your CPU for the feature extraction pipeline. The execution environment for the network training is your GPU(s).

```
tic
net = trainNetwork(adsTrainCPU,layers,options);
```

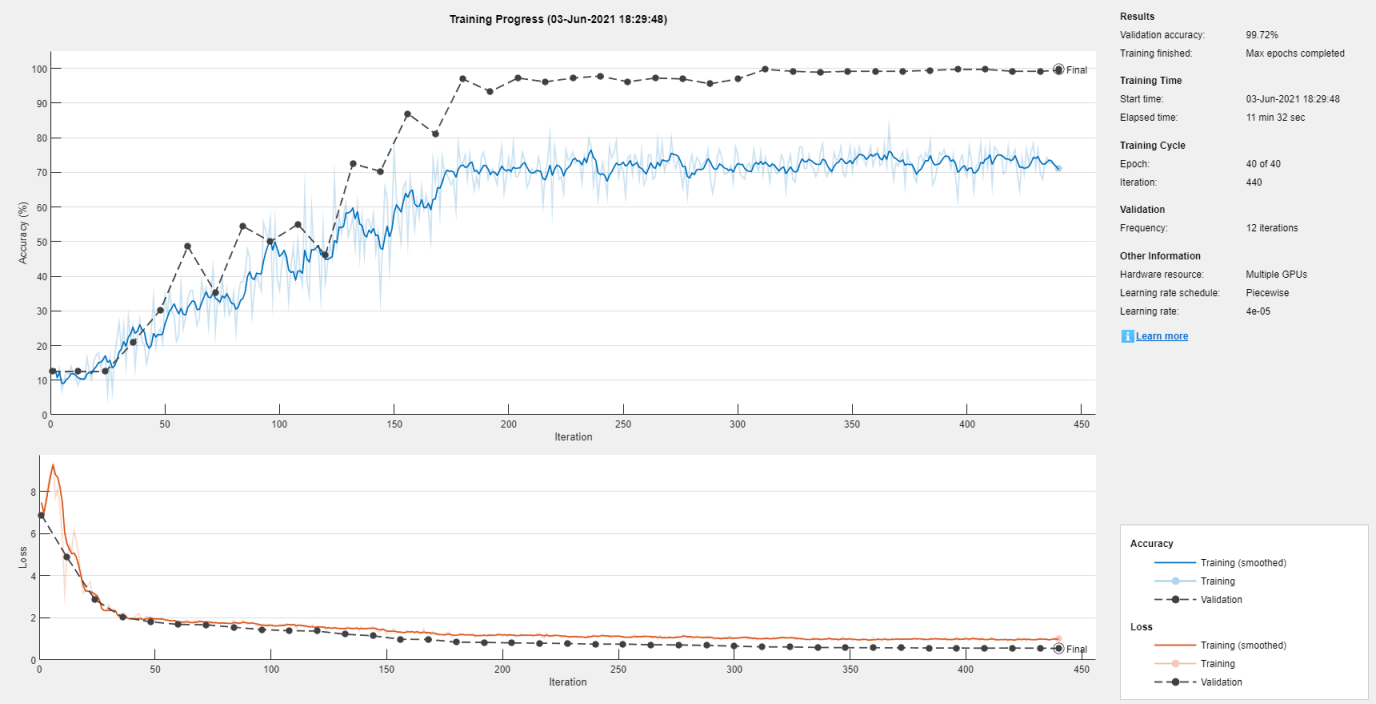



```
cpuTrainTime = toc;
```

Train Network Using GPU-Based Preprocessing

Replace the validation data in the training options with the GPU-based pipeline. Train the network using your GPU(s) for the feature extraction pipeline. The execution environment for the network training is your GPU(s).

```
options.ValidationData = adsValidationGPU;
tic
net = trainNetwork(adsTrainGPU,layers,options);
```




```
gpuTrainTime = toc;
```

Compare CPU- and GPU-based Preprocessing

Print the timing results for training using a CPU for feature extraction and augmentation, and training using GPU(s) for feature extraction and augmentation.

```
fprintf(['Training time (CPU): %0.2f seconds\n' ...
    'Training time (GPU): %0.2f seconds\n' ...
    'Speedup (CPU time)/(GPU time): %0.3f\n\n'], ...
    cpuTrainTime,gpuTrainTime,cpuTrainTime/gpuTrainTime)
```

```
Training time (CPU): 4242.82 seconds
Training time (GPU): 731.77 seconds
Speedup (CPU time)/(GPU time): 5.798
```

Compare CPU and GPU Inference Performance

Compare the time it takes to perform prediction on a single 3-second clip when feature extraction is performed on the GPU versus the CPU. In both cases, the network prediction happens on your GPU.

```
signalToClassify = read(ads);

gpuFeatureExtraction = gputimeit(@()predict(net,log10(extract(afe,gpuArray(signalToClassify))+of
cpuFeatureExtraction = gputimeit(@()predict(net,log10(extract(afe,(signalToClassify))+offset)));

fprintf(['Prediction time for 3 s of data (feature extraction on CPU): %0.2f ms\n' ...
    'Prediction time for 3 s of data (feature extraction on GPU): %0.2f ms\n' ...
    'Speedup (CPU time)/(GPU time): %0.3f\n\n'], ...
    cpuFeatureExtraction*1e3,gpuFeatureExtraction*1e3,cpuFeatureExtraction/gpuFeatureExtraction)
```

```
Prediction time for 3 s of data (feature extraction on CPU): 41.62 ms
Prediction time for 3 s of data (feature extraction on GPU): 7.71 ms
Speedup (CPU time)/(GPU time): 5.397
```

Compare the time it takes to perform prediction on a set of 3-second clips when feature extraction is performed on the GPU(s) versus the CPU. In both cases, the network prediction happens on your GPU(s).

```
adsValidationGPU = transform(adsValidation,@(x)gpuArray(x));
adsValidationGPU = transform(adsValidationGPU,@(x){log10(extract(afe,x)+offset)});
adsValidationCPU = transform(adsValidation,@(x){log10(extract(afe,x)+offset)});

gpuFeatureExtraction = gputimeit(@()predict(net,adsValidationGPU,'ExecutionEnvironment','multi-g
cpuFeatureExtraction = gputimeit(@()predict(net,adsValidationCPU,'ExecutionEnvironment','multi-g

fprintf(['Prediction time for validation set (feature extraction on CPU): %0.2f s\n' ...
    'Prediction time for validation set (feature extraction on GPU): %0.2f s\n' ...
    'Speedup (CPU time)/(GPU time): %0.3f\n\n'], ...
    cpuFeatureExtraction,gpuFeatureExtraction,cpuFeatureExtraction/gpuFeatureExtraction)
```

```
Prediction time for validation set (feature extraction on CPU): 34.11 s
Prediction time for validation set (feature extraction on GPU): 5.53 s
Speedup (CPU time)/(GPU time): 6.173
```

Conclusion

It is well known that you can decrease the time it takes to train a network by leveraging GPU devices. This enables you to more quickly iterate and develop your final system. In many training setups, you can achieve additional performance gains by leveraging GPU devices for feature extraction and data augmentation. This example shows a significant decrease in the overall time it takes to train a CNN when leveraging GPU devices for feature extraction and data augmentation. Additionally, leveraging GPU devices for feature extraction at inference time, for both single-observations and data sets, achieves significant performance gains.

Supporting Functions

Mixup

The supporting object, `Mixup`, is placed in your current folder when you open this example.

```
type Mixup
```

```
classdef Mixup < handle
    %MIXUP Mixup data augmentation
    %   mixer = Mixup(augDatastore) creates an object that can mix features
    %   at a randomly set ratio and then probabilistically set the output
    %   label as one of the two original signals.
    %
    %   Mixup Properties:
    %   MixProbability - Mix probability
    %   AugDatastore   - Augmentation datastore
    %
    %   Mixup Methods:
    %   mix            - Apply mixup
    %

    % Copyright 2021 The MathWorks, Inc.
```

```
properties (SetAccess=public,GetAccess=public)
    %MixProbability Mix probability
    % Specify the probability that mixing is applied as a scalar in the
    % range [0,1]. If unspecified, MixProbability defaults to 1/3.
    MixProbability (1,1) {mustBeNumeric} = 1/3;
end
properties (SetAccess=immutable,GetAccess=public)
    %AUGDATASTORE Augmentation datastore
    % Specify a datastore from which to get the mixing signals. The
    % datastore must contain a label in the info returned from reading.
    % This property is immutable, meaning it cannot be changed after
    % construction.
    AugDatastore
end

methods
    function obj = Mixup(augDatastore)
        obj.AugDatastore = augDatastore;
    end

    function [dataOut,infoOut] = mix(obj,x,infoIn)
        %MIX Apply mixup
        % [dataOut,infoOut] = mix(mixer,x,infoIn) probabilistically mix
        % the input, x, and its associated label contained in infoIn
        % with a signal randomly drawn from the augmentation datastore.
        % The output, dataOut, is a cell array with two columns. The
        % first column contains the features and the second column
        % contains the label.

        if rand > obj.MixProbability % Only mix ~1/3 the dataset

            % Randomly set mixing coefficient. Draw from a normal
            % distribution with mean 0.5 and contained within [0,1].
            lambda = max(min((randn./10)+0.5,1),0);

            % Read one file from the augmentation datastore.
            subDS = subset(obj.AugDatastore,randi([1,numel(obj.AugDatastore.UnderlyingDatasto
            [y,yInfo] = read(subDS);

            % Mix the features element-by-element according to lambda.
            dataOut = lambda*x + (1-lambda)*y;

            % Set the output label probabilistically based on the mixing coefficient.
            if lambda < rand
                labelOut = yInfo.Label;
                infoOut.Label = labelOut;
            else
                labelOut = infoIn.Label;
            end
            infoOut.Label = labelOut;

            % Combine the output data and labels.
            dataOut = [{dataOut},{labelOut}];

        else % Do not apply mixing

            dataOut = [{x},{infoIn.Label}];
            infoOut = infoIn;
```

```
            end
        end

    end
end
```

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." IEEE Transactions on Reliability, vol. 65, no. 1, Mar. 2016, pp. 291–309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

[2] Huszar, Ferenc. "Mixup: Data-Dependent Data Augmentation." InFERENCe. November 03, 2017. Accessed January 15, 2019. https://www.inference.vc/mixup-data-dependent-data-augmentation/.

Acoustics-Based Machine Fault Recognition

In this example, you develop a deep learning model to detect faults in an air compressor using acoustic measurements. After developing the model, you package the system so that you can recognize faults based on streaming input data.

Data Preparation

Download and unzip the air compressor data set [1] on page 14-0 . This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

```
loc = matlab.internal.examples.downloadSupportFile('audio','AirCompressorDataset/AirCompressorDa
unzip(loc,pwd)
```

Create an `audioDatastore` (Audio Toolbox) object to manage the data and split it into training and validation sets. Call `countEachLabel` (Audio Toolbox) to inspect the distribution of labels in the train and validation sets.

```
ads = audioDatastore(pwd,'IncludeSubfolders',true,'LabelSource','foldernames');

[adsTrain,adsValidation] = splitEachLabel(ads,0.9,0.1);

countEachLabel(adsTrain)
```

```
ans=8×2 table
      Label      Count
    _________    _____

    Bearing       203
    Flywheel      203
    Healthy       203
    LIV           203
    LOV           203
    NRV           203
    Piston        203
    Riderbelt     203
```

```
countEachLabel(adsValidation)
```

```
ans=8×2 table
      Label      Count
    _________    _____

    Bearing       22
    Flywheel      22
    Healthy       22
    LIV           22
    LOV           22
    NRV           22
    Piston        22
    Riderbelt     22
```

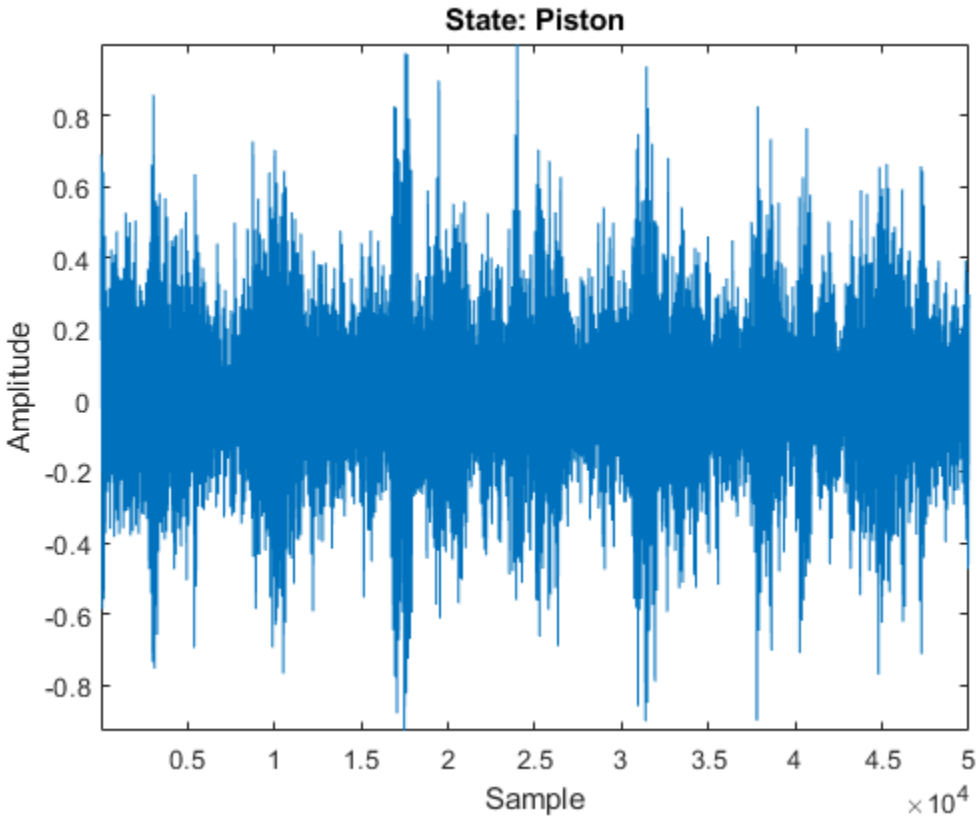
```
adsTrain = shuffle(adsTrain);
adsValidation = shuffle(adsValidation);
```

You can reduce the training data set used in this example to speed up the runtime at the cost of performance. In general, reducing the data set is a good practice for development and debugging.

```
reduceDataset = false;
if reduceDataset
    adsTrain = splitEachLabel(adsTrain,20);
end
```

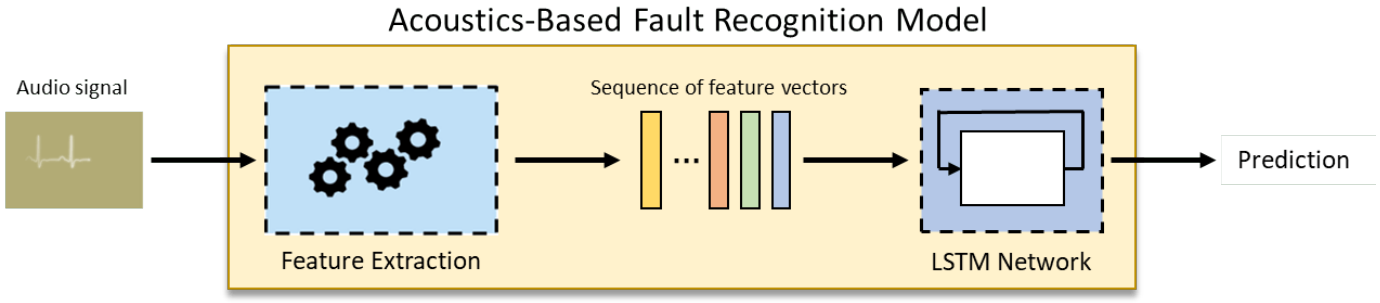
The data consists of time-series recordings of acoustics from faulty or healthy air compressors. As such, there are strong relationships between samples in time. Listen to a recording and plot the waveform.

```
[sampleData,sampleDataInfo] = read(adsTrain);
fs = sampleDataInfo.SampleRate;

soundsc(sampleData,fs)
plot(sampleData)
xlabel("Sample")
ylabel("Amplitude")
title("State: " + string(sampleDataInfo.Label))
axis tight
```



Because the samples are related in time, you can use a recurrent neural network (RNN) to model the data. A long short-term memory (LSTM) network is a popular choice of RNN because it is designed to avoid vanishing and exploding gradients. Before you can train the network, it's important to prepare

the data adequately. Often, it is best to transform or extract features from 1-dimensional signal data in order to provide a richer set of features for the model to learn from.



Feature Engineering

The next step is to extract a set of acoustic features used as inputs to the network. Audio Toolbox™ enables you to extract spectral descriptors that are commonly used as inputs in machine learning tasks. You can extract the features using individual functions, or you can use `audioFeatureExtractor` (Audio Toolbox) to simplify the workflow and do it all at once.

```
trainFeatures = cell(1,numel(adsTrain.Files));
windowLength = 512;
overlapLength = 0;

aFE = audioFeatureExtractor('SampleRate',fs, ...
    'Window',hamming(windowLength,'periodic'),...
    'OverlapLength',overlapLength,...
    'spectralCentroid',true, ...
    'spectralCrest',true, ...
    'spectralDecrease',true, ...
    'spectralEntropy',true, ...
    'spectralFlatness',true, ...
    'spectralFlux',false, ...
    'spectralKurtosis',true, ...
    'spectralRolloffPoint',true, ...
    'spectralSkewness',true, ...
    'spectralSlope',true, ...
    'spectralSpread',true);

reset(adsTrain)
tic
for index = 1:numel(adsTrain.Files)
    data = read(adsTrain);
    trainFeatures{index} = (extract(aFE,data))';
end
fprintf('Feature extraction of training set took %f seconds.\n',toc);
```

```
Feature extraction of training set took 13.648354 seconds.
```

Data Augmentation

The training set contains a relatively small number of acoustic recordings for training a deep learning model. A popular method to enlarge the dataset is to use mixup. In mixup, you augment your dataset by mixing the features and labels from two different class instances. Mixup was reformulated by [2] on page 14-0 as labels drawn from a probability distribution instead of mixed labels. The supporting function, mixup on page 14-0 , takes the training features, associated labels, and the number of mixes per observation and then outputs the mixes and associated labels.

```
trainLabels = adsTrain.Labels;

numMixesPerInstance = 2 ;
tic
[augData,augLabels] = mixup(trainFeatures,trainLabels,numMixesPerInstance);

trainLabels = cat(1,trainLabels,augLabels);
trainFeatures = cat(2,trainFeatures,augData);
fprintf('Feature augmentation of train set took %f seconds.\n',toc);
```

```
Feature augmentation of train set took 0.250037 seconds.
```

Generate Validation Features

Repeat the feature extraction for the validation features.

```
validationFeatures = cell(1,numel(adsValidation.Files));

reset(adsValidation)
tic
for index = 1:numel(adsValidation.Files)
    data = read(adsValidation);
    validationFeatures{index} = (extract(aFE,data))';
end
fprintf('Feature extraction of validation set took %f seconds.\n',toc);
```

```
Feature extraction of validation set took 1.388853 seconds.
```

Train Model

Next, you define and train a network. To skip training the network, set `downloadPretrainedSystem` to `true`, then continue to the next section on page 14-0 .

```
downloadPretrainedSystem = false ;
if downloadPretrainedSystem
    loc = matlab.internal.examples.downloadSupportFile('audio','AcousticsBasedMachineFaultRecogn
    unzip(loc,pwd)
    addpath(fullfile(pwd,'AcousticsBasedMachineFaultRecognition'))
end
```

Define Network

An LSTM layer learns long-term dependencies between time steps of time series or sequence data. The first `lstmLayer` has 100 hidden units and outputs sequence data. Then a dropout layer is used to reduce overfitting. The second `lstmLayer` outputs the last step of the time sequence.

```
numHiddenUnits = 100 ;
dropProb = 0.2 ;
```

```
layers = [ ...
    sequenceInputLayer(aFE.FeatureVectorLength,'Normalization','zscore')
    lstmLayer(numHiddenUnits,"OutputMode","sequence")
    dropoutLayer(dropProb)
    lstmLayer(numHiddenUnits,"OutputMode","last")
    fullyConnectedLayer(numel(unique(adsTrain.Labels)))
    softmaxLayer
    classificationLayer];
```

Define Network Hyperparameters

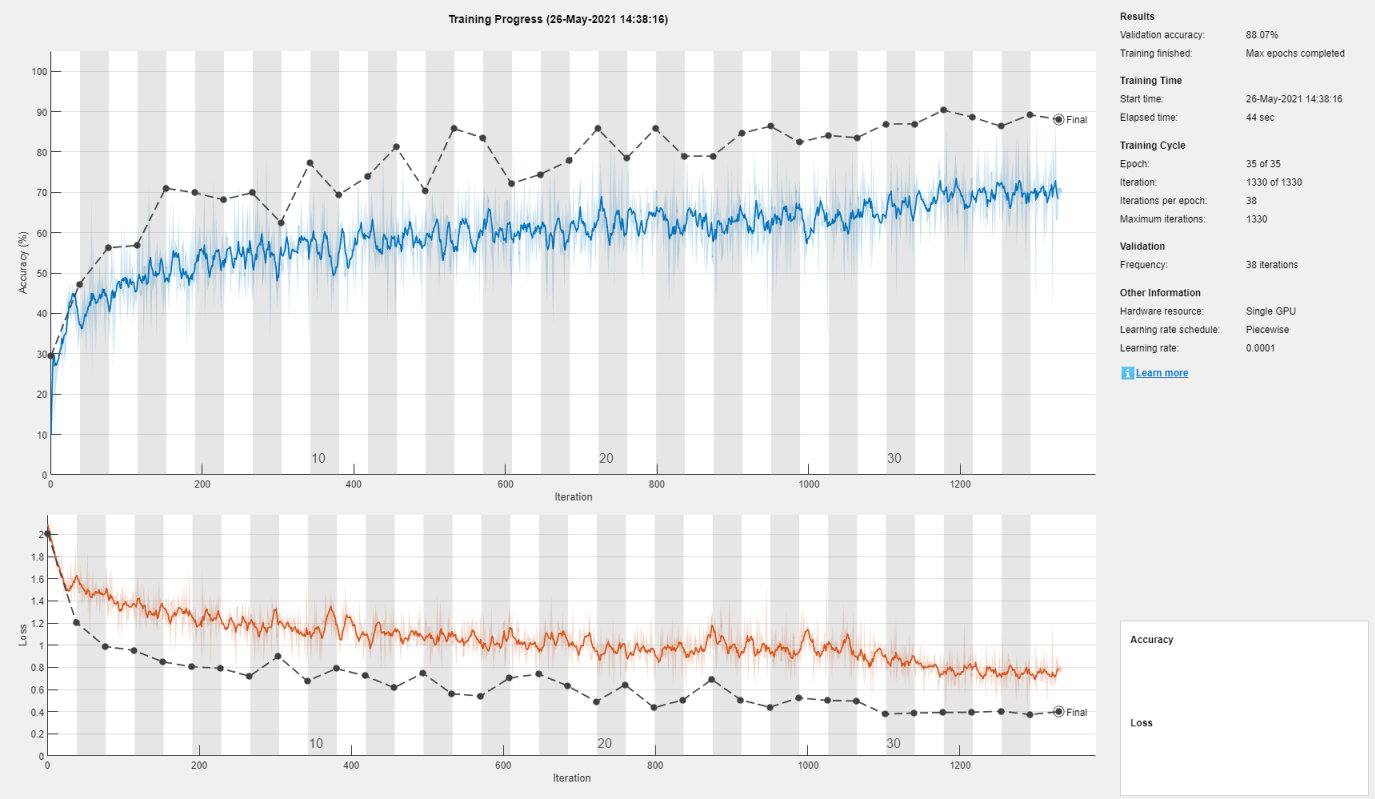
To define hyperparameters for the network, use `trainingOptions`.

```
miniBatchSize = 128 ;
validationFrequency = floor(numel(trainFeatures)/miniBatchSize);
options = trainingOptions("adam", ...
    "MiniBatchSize",miniBatchSize, ...
    "MaxEpochs",35, ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "Shuffle","every-epoch", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropPeriod",30, ...
    "LearnRateDropFactor",0.1, ...
    "ValidationData",{validationFeatures,adsValidation.Labels}, ...
    "ValidationFrequency",validationFrequency);
```

Train Network

To train the network, use `trainNetwork`.

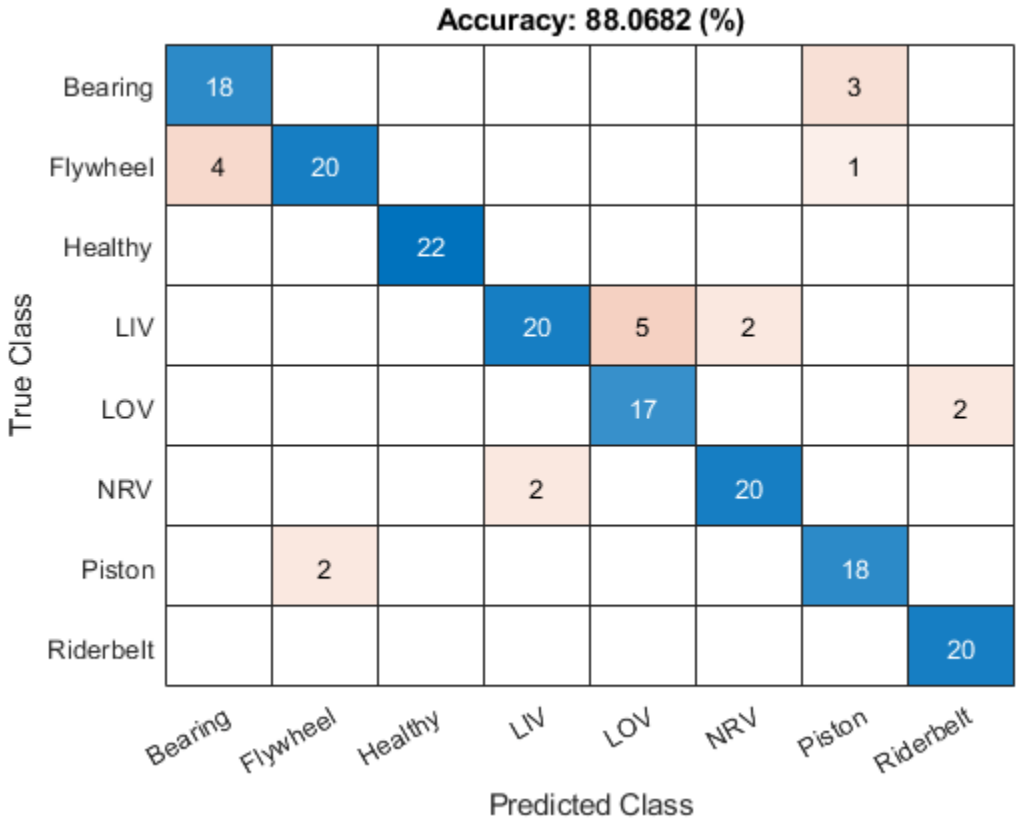
```
airCompNet = trainNetwork(trainFeatures,trainLabels,layers,options);
```



Evaluate Network

View the confusion chart for the validation data.

```
validationResults = classify(airCompNet,validationFeatures);
cm = confusionchart(validationResults,adsValidation.Labels);
accuracy = mean(validationResults == adsValidation.Labels)*100;
cm.title("Accuracy: " + accuracy + " (%)")
```



Model Streaming Detection

Create Functions to Process Data in a Streaming Loop

Once you have a trained network with satisfactory performance, you can apply the network to test data in a streaming fashion.

There are many additional considerations to take into account to make the system work in a real-world embedded system.

For example,

- The rate or interval at which classification can be performed with accurate results
- The size of the network in terms of generated code (program memory) and weights (data memory)
- The efficiency of the network in terms of computation speed

In MATLAB, you can mimic how the network is deployed and used in hardware on a real embedded system and begin to answer these important questions.

Create MATLAB Function Compatible with C/C++ Code Generation

Once you train your deep learning model, you will deploy it to an embedded target. That means you also need to deploy the code used to perform the feature extraction. Use the `generateMATLABFunction` method of `audioFeatureExtractor` to generate a MATLAB function compatible with C/C++ code generation. Specify `IsStreaming` as `true` so that the generated function is optimized for stream processing.

```
filename = fullfile(pwd,"extractAudioFeatures");
generateMATLABFunction(aFE,filename,'IsStreaming',true);
```

Combine Streaming Feature Extraction and Classification

Save the trained network as a MAT file.

```
save('AirCompressorFaultRecognitionModel.mat','airCompNet')
```

Create a function that combines the feature extraction and deep learning classification.

```
type recognizeAirCompressorFault.m
```

```
function scores = recognizeAirCompressorFault(audioIn,rs)
% This is a streaming classifier function

persistent airCompNet

if isempty(airCompNet)
    airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
end
if rs
    airCompNet = resetState(airCompNet);
end

% Extract features using function
features = extractAudioFeatures(audioIn);

% Classify
[airCompNet,scores] = predictAndUpdateState(airCompNet,features);

end
```

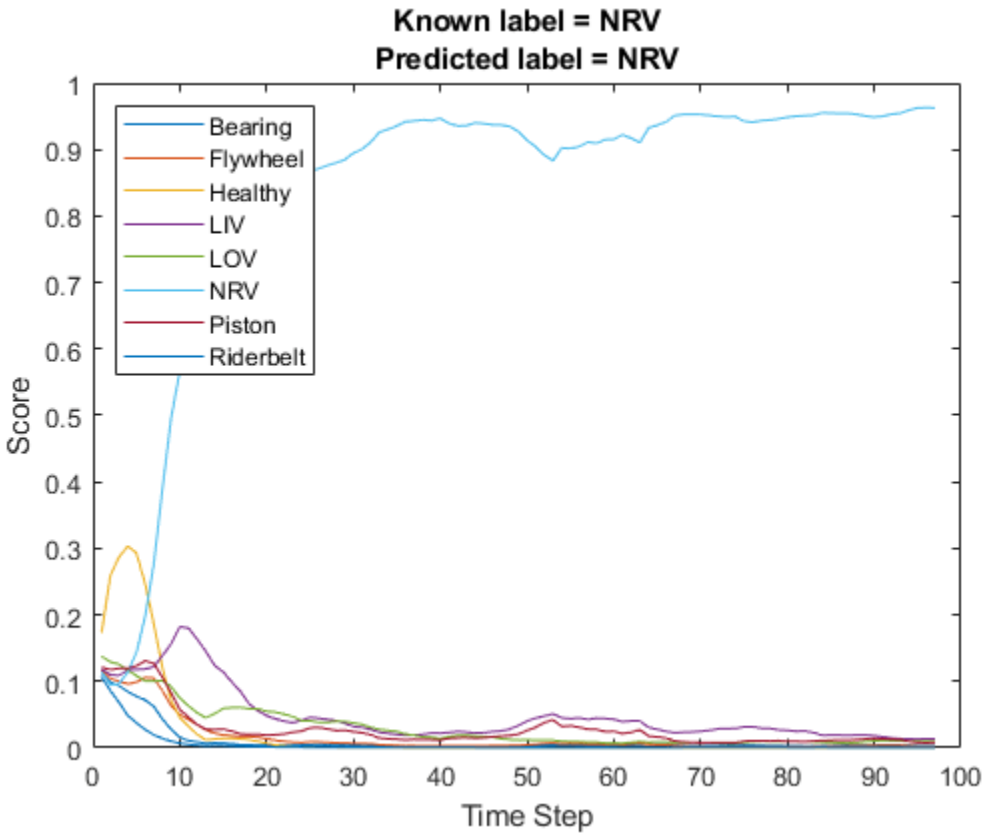
Test Streaming Loop

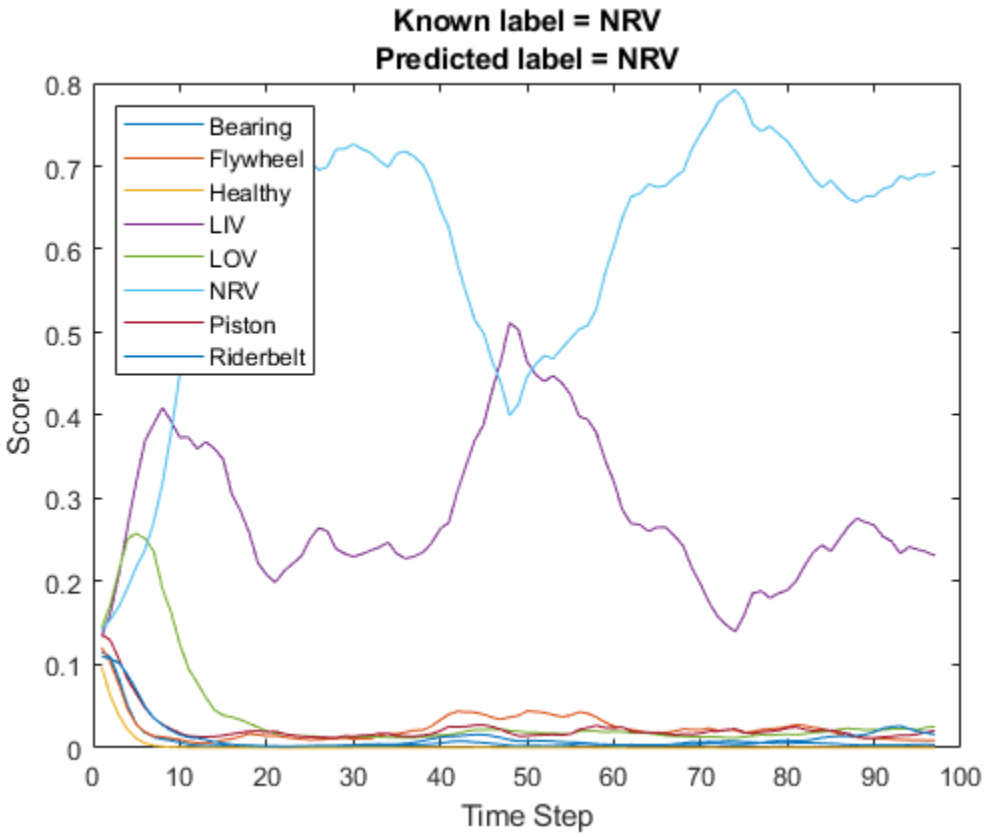
Next, you test the streaming classifier in MATLAB. Stream audio one frame at a time to represent a system as it would be deployed in a real-time embedded system. This enables you to measure and visualize the timing and accuracy of the streaming implementation.

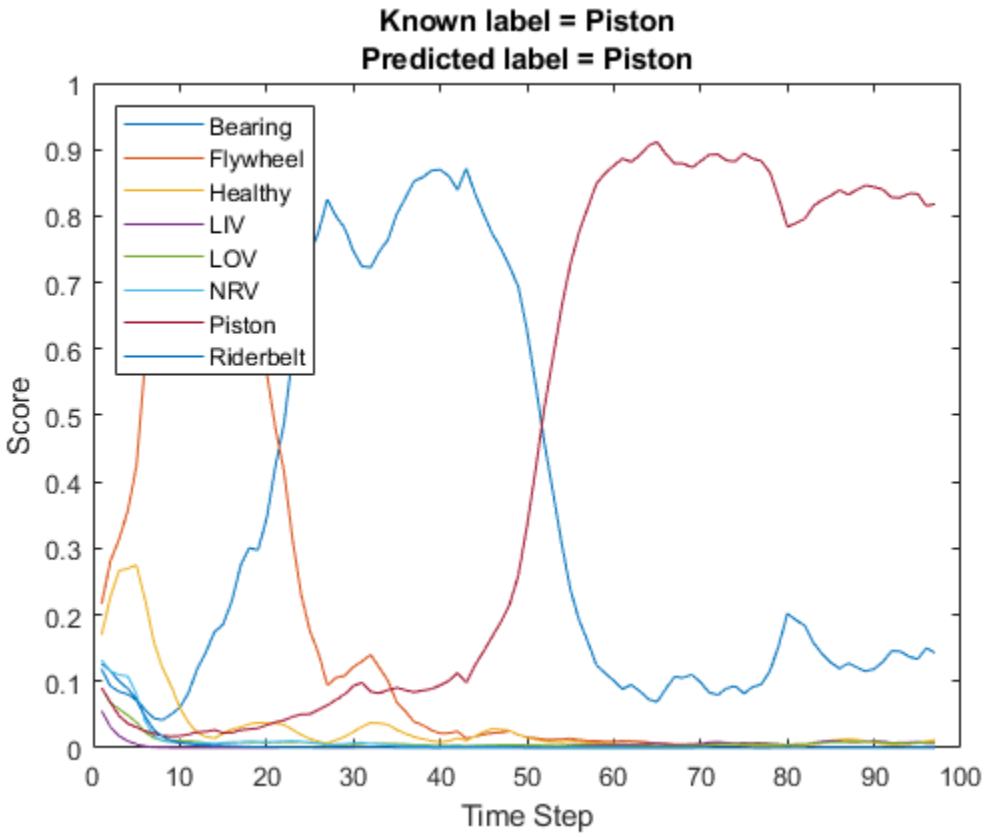
Stream in several audio files and plot the output classification results for each frame of data. At a time interval equal to the length of each file, evaluate the output of the classifier.

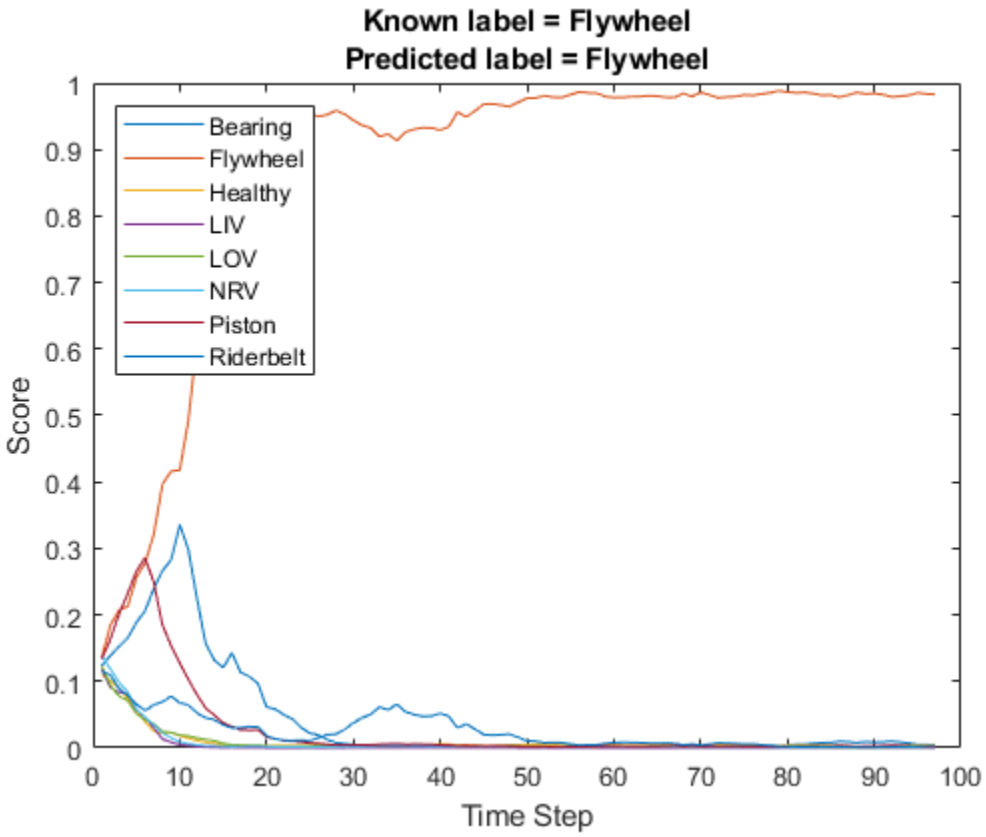
```
reset(adsValidation)

N = 10;
labels = categories(ads.Labels);
numLabels = numel(labels);

% Create a dsp.AsyncBuffer to read audio in a streaming fashion
audioSource = dsp.AsyncBuffer;

% Create a dsp.AsyncBuffer to accumulate scores
scoreBuffer = dsp.AsyncBuffer;

% Create a dsp.AsyncBuffer to record execution time.
timingBuffer = dsp.AsyncBuffer;

% Pre-allocate array to store results
```

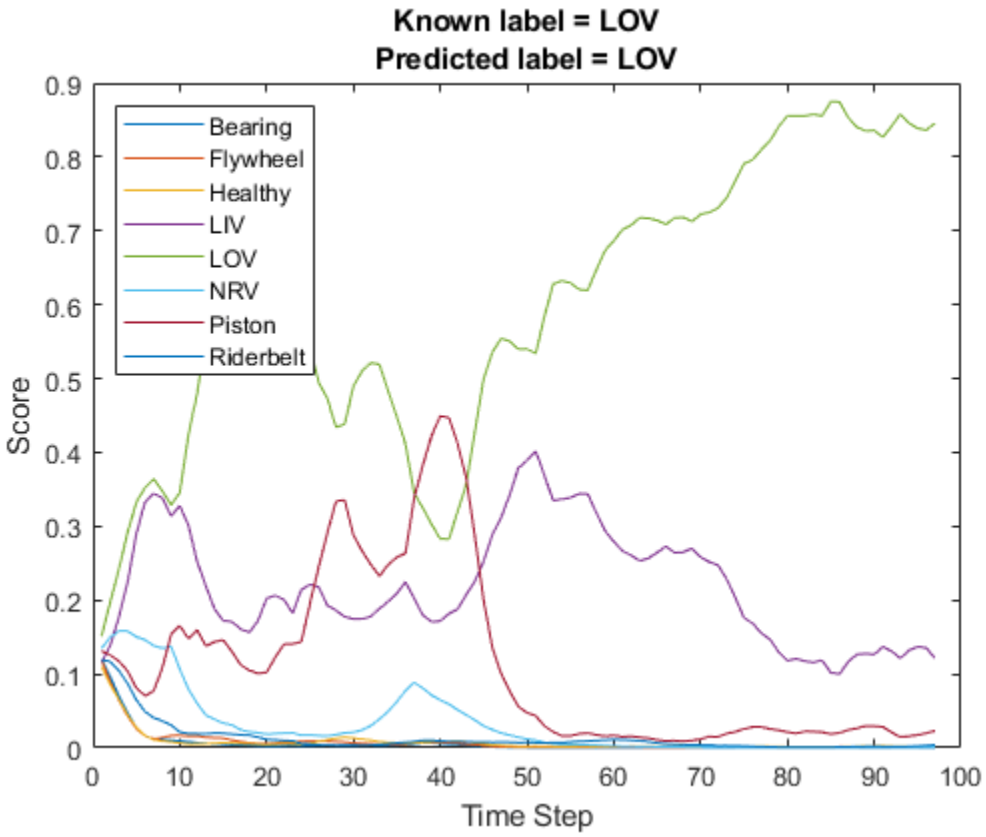
```
streamingResults = categorical(zeros(N,1));

% Loop over files
for fileIdx = 1:N

    % Read one audio file and put it in the source buffer
    [data,dataInfo] = read(adsValidation);
    write(audioSource,data);

    % Inner loop over frames
    rs = true;
    while audioSource.NumUnreadSamples >= windowLength

        % Get a frame of audio data
        x = read(audioSource,windowLength);

        % Apply streaming classifier function
        tic
        score = recognizeAirCompressorFault(x,rs);
        write(timingBuffer,toc);

        % Store score for analysis
        write(scoreBuffer,score);

        rs = false;
    end
    reset(audioSource)

    % Store class result for that file
    scores = read(scoreBuffer);
    [~,result] = max(scores(end,:),[],2);
    streamingResults(fileIdx) = categorical(labels(result));

    % Plot scores to compare over time
    figure
    plot(scores) %#ok<*NASGU>
    legend(string(airCompNet.Layers(end).Classes),'Location','northwest')
    xlabel("Time Step")
    ylabel("Score")
    str = sprintf('Known label = %s\nPredicted label = %s',string(dataInfo.Label),string(streami
    title(str)
end
```

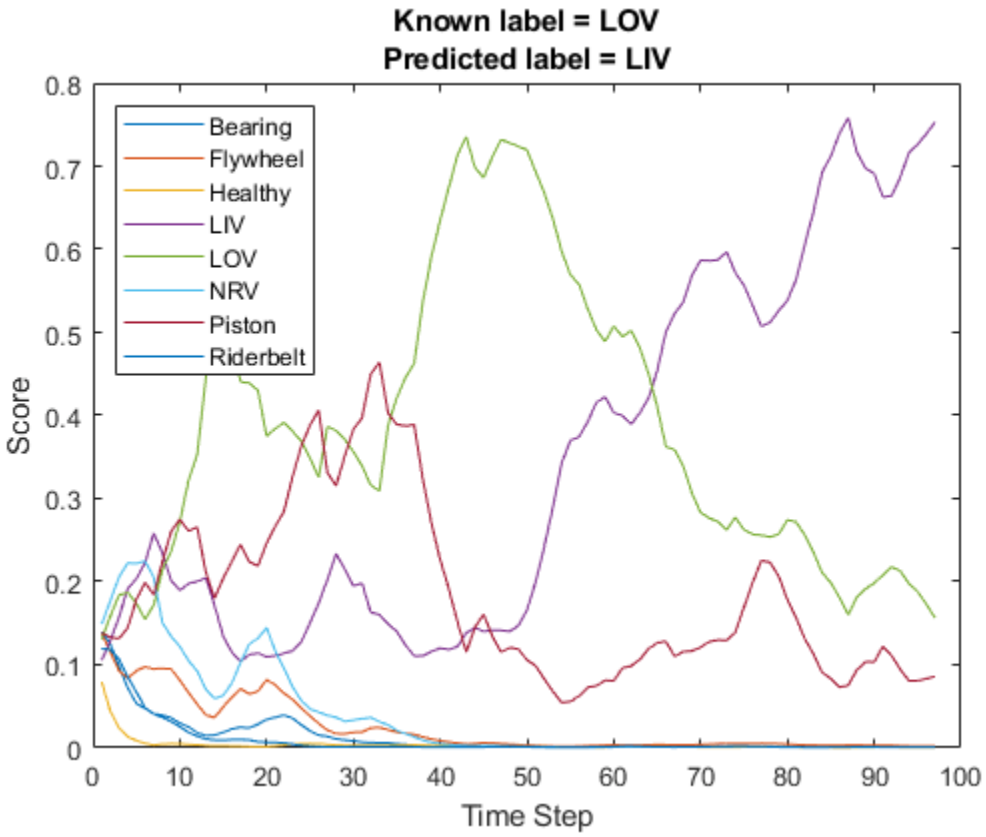

Known label = NRV
Predicted label = NRV
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100

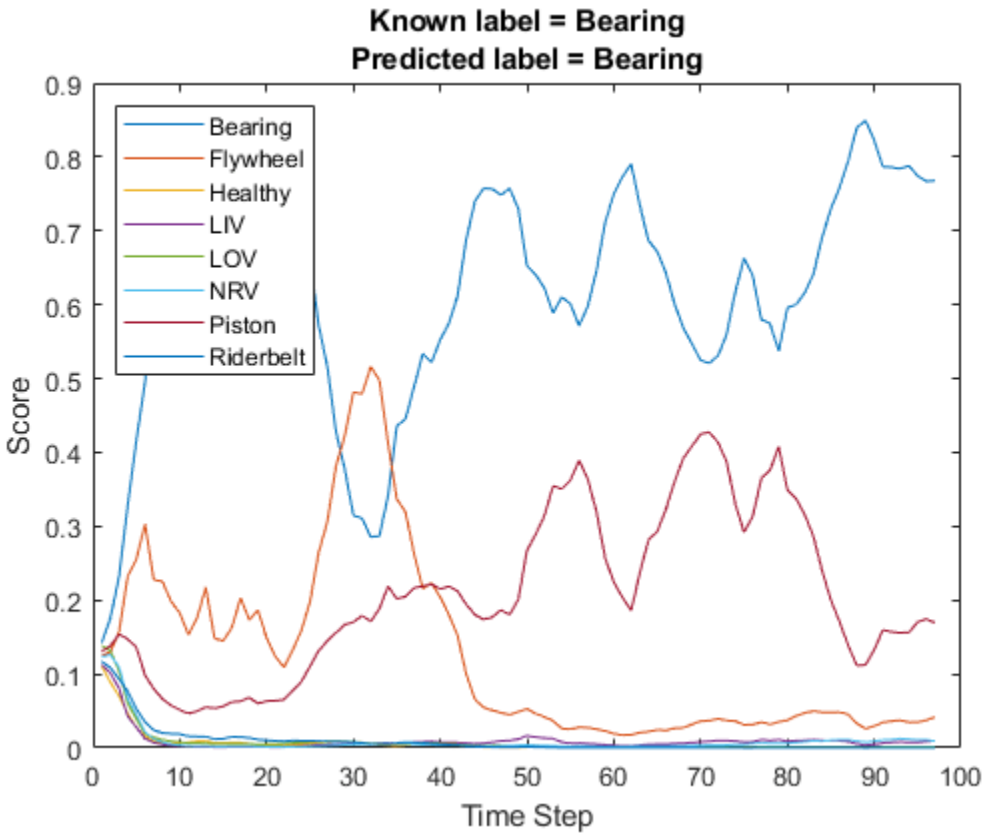

Known label = NRV
Predicted label = NRV
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50
60
70
80
90
100

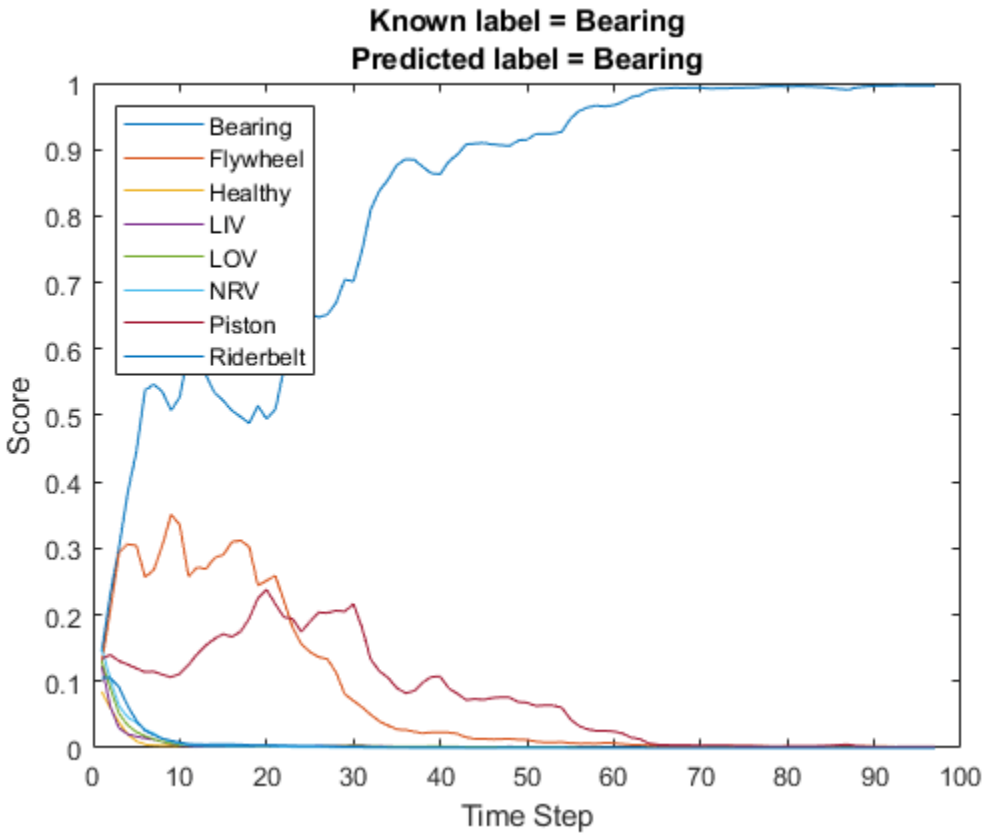

Known label = Piston
Predicted label = Piston
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100

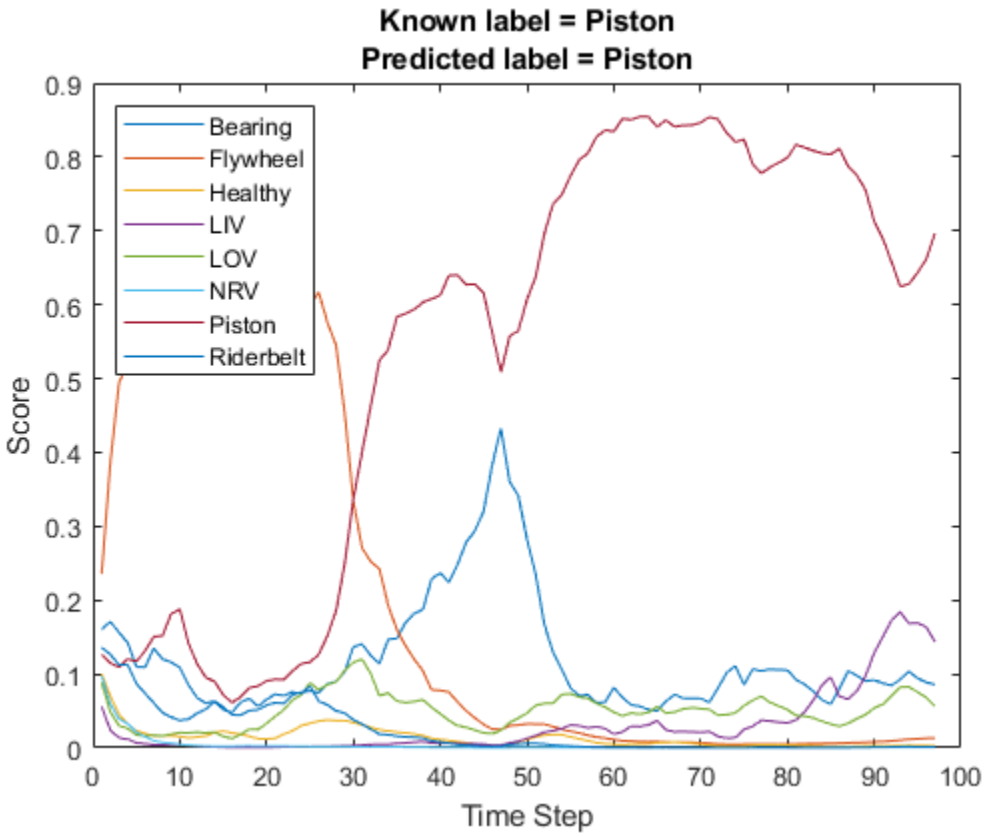

Known label = Flywheel
Predicted label = Flywheel
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100

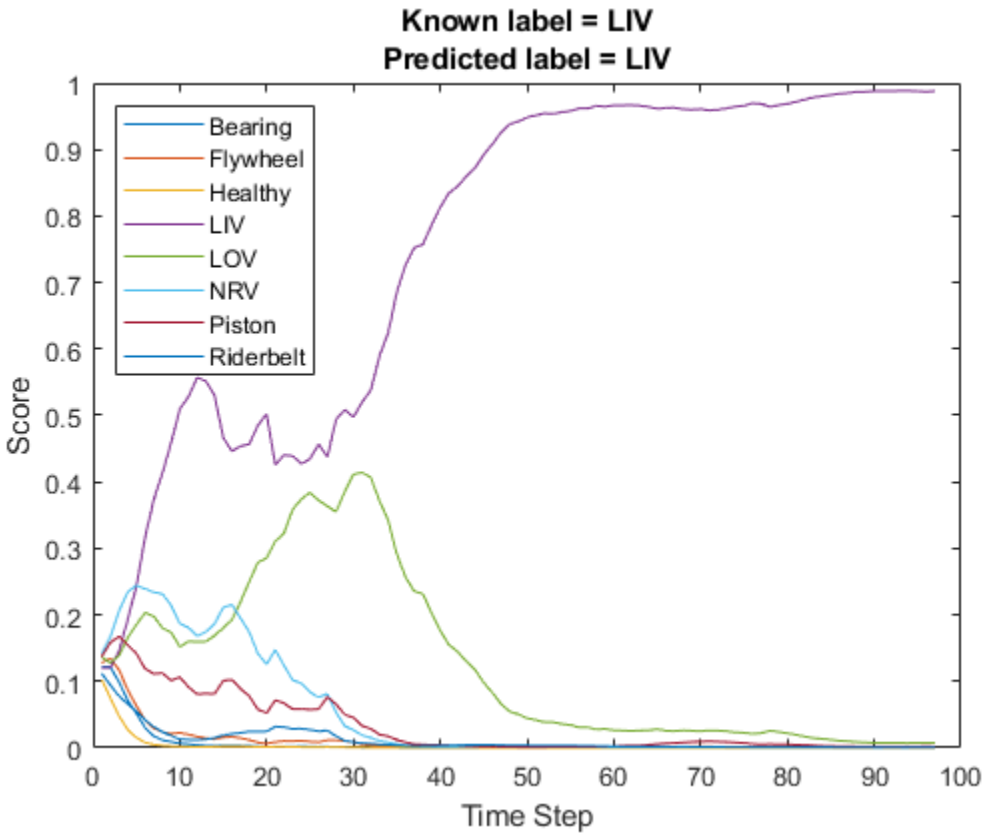

Known label = LOV
Predicted label = LOV
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50
60
70
80
90
100


Known label = LOV
Predicted label = LIV
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50
60
70
80
90
100


Known label = Bearing
Predicted label = Bearing
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50
60
70
80
90
100


Known label = Bearing
Predicted label = Bearing
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
10
20
30
40
50
60
70
80
90
100


Known label = Piston
Predicted label = Piston
Bearing
Flywheel
Healthy
LIV
LOV
NRV
Piston
Riderbelt
Score
Time Step
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50
60
70
80
90
100



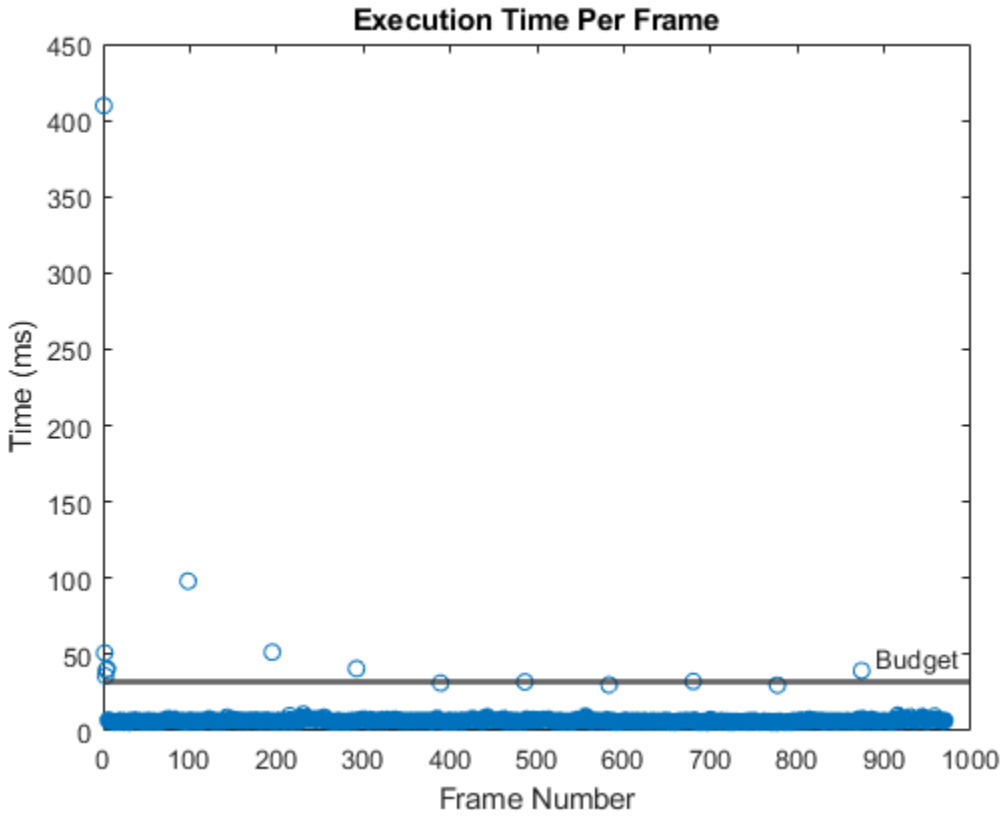
Compare the test results for the streaming version of the classifier and the non-streaming.

```
testError = mean(validationResults(1:N) ~= streamingResults);
disp("Error between streaming classifier and non-streaming: " + testError*100 + " (%)")
```

```
Error between streaming classifier and non-streaming: 0 (%)
```

Analyze the execution time. The execution time when state is reset is often above the 32 ms budget. However, in a real, deployed system, that initialization time will only be incurred once. The execution time of the main loop is around 10 ms, which is well below the 32 ms budget for real-time performance.

```
executionTime = read(timingBuffer)*1000;
budget = (windowLength/aFE.SampleRate)*1000;
plot(executionTime,'o')
title('Execution Time Per Frame')
xlabel('Frame Number')
ylabel('Time (ms)')
yline(budget,'',{'Budget'},'LineWidth',2)
```



Supporting Functions

```
function [augData,augLabels] = mixup(data,labels,numMixesPerInstance)
augData = cell(1,numel(data)*numMixesPerInstance);
augLabels = repelem(labels,numMixesPerInstance);

kk = 1;
for ii = 1:numel(data)
    for jj = 1:numMixesPerInstance
        lambda = max(min((randn./10)+0.5,1),0);

        % Find all available data with different labels.
        availableData = find(labels~=labels(ii));

        % Randomly choose one of the available data with a different label.
        numAvailableData = numel(availableData);
        idx = randi([1,numAvailableData]);

        % Mix.
        augData{kk} = lambda*data{ii} + (1-lambda)*data{availableData(idx)};

        % Specify the label as randomly set by lambda.
        if lambda < rand
            augLabels(kk) = labels(availableData(idx));
        else
            augLabels(kk) = labels(ii);
        end
```

```
            kk = kk + 1;
        end
    end

    end
```

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." IEEE Transactions on Reliability, vol. 65, no. 1, Mar. 2016, pp. 291–309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

[2] Huszar, Ferenc. "Mixup: Data-Dependent Data Augmentation." InFERENCe. November 03, 2017. Accessed January 15, 2019. https://www.inference.vc/mixup-data-dependent-data-augmentation/.

Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN

This example demonstrates code generation for “Acoustics-Based Machine Fault Recognition” (Audio Toolbox) using a long short-term memory (LSTM) network and spectral descriptors. This example uses MATLAB® Coder™ with deep learning support to generate a MEX (MATLAB executable) function that leverages performance of Intel® MKL-DNN library. The input data consists of acoustics time-series recordings from faulty or healthy air compressors and the output is the state of the mechanical machine predicted by the LSTM network. For details on audio preprocessing and network training, see “Acoustics-Based Machine Fault Recognition” (Audio Toolbox).

Example Requirements

- The MATLAB Coder Interface f or Deep Learning Libraries Support Package
- Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2)
- Intel Deep Neural Networks Library (MKL-DNN)
- Environment variables for Intel MKL-DNN

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)

Prepare Input Dataset

Specify a sample rate `fs` of 16 kHz and a `windowLength` of 512 samples, as defined in “Acoustics-Based Machine Fault Recognition” (Audio Toolbox). Set `numFrames` to `100`.

```
fs = 16000;
windowLength = 512;
numFrames = 100;
```

To run the Example on a test signal, generate a pink noise signal. To test the performance of the system on a real dataset, download the air compressor dataset [1] on page 14-0 .

```
downloadDataset = false

if ~downloadDataset
    pinkNoiseSignal = pinknoise(windowLength*numFrames);
else
    % Download AirCompressorDataset.zip
    component = 'audio';
    filename = 'AirCompressorDataset/AirCompressorDataset.zip';
    localfile = matlab.internal.examples.downloadSupportFile(component,filename);

    % Unzip the downloaded zip file to the downloadFolder
    downloadFolder = fileparts(localfile);
    if ~exist(fullfile(downloadFolder,'AirCompressorDataset'),'dir')
        unzip(localfile, downloadFolder)
    end

    % Create an audioDatastore object dataStore, to manage, the data.
    dataStore = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames

    % Use countEachLabel to get the number of samples of each category in the dataset.
```

```
    countEachLabel(dataStore)
end
```

Recognize Machine Fault in MATLAB

To run the streaming classifier in MATLAB, download and unzip the system developed in "Acoustics-Based Machine Fault Recognition" (Audio Toolbox).

```
component = 'audio';
filename = 'AcousticsBasedMachineFaultRecognition/AcousticsBasedMachineFaultRecognition.zip';
localfile = matlab.internal.examples.downloadSupportFile(component,filename);

downloadFolder = fullfile(fileparts(localfile),'system');
if ~exist(downloadFolder,'dir')
    unzip(localfile,downloadFolder)
end
```

To access the `recognizeAirCompressorFault` function of the system, add `downloadFolder` to the search path.

```
addpath(downloadFolder)
```

Create a `dsp.AsyncBuffer` (DSP System Toolbox) object to read audio in a streaming fashion and a `dsp.AsyncBuffer` (DSP System Toolbox) object to accumulate scores.

```
audioSource = dsp.AsyncBuffer;
scoreBuffer = dsp.AsyncBuffer;
```

Load the pretrained network and extract labels from the network.

```
airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
labels = string(airCompNet.Layers(end).Classes);
```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles,~] = splitEachLabel(dataStore,1);
    allData = readall(allFiles);
    signalToBeTested = NRV ;
    signalToBeTested = cell2mat(signalToBeTested);
end
```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use `recognizeAirCompressorFault` developed in "Acoustics-Based Machine Fault Recognition" (Audio Toolbox) to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);
resetNetworkState = true;

while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource,windowLength);
```

```
    % Apply streaming classifier function
    score = recognizeAirCompressorFault(x,resetNetworkState);

    % Store score for analysis
    write(scoreBuffer,score);

    resetNetworkState = false;
end
```

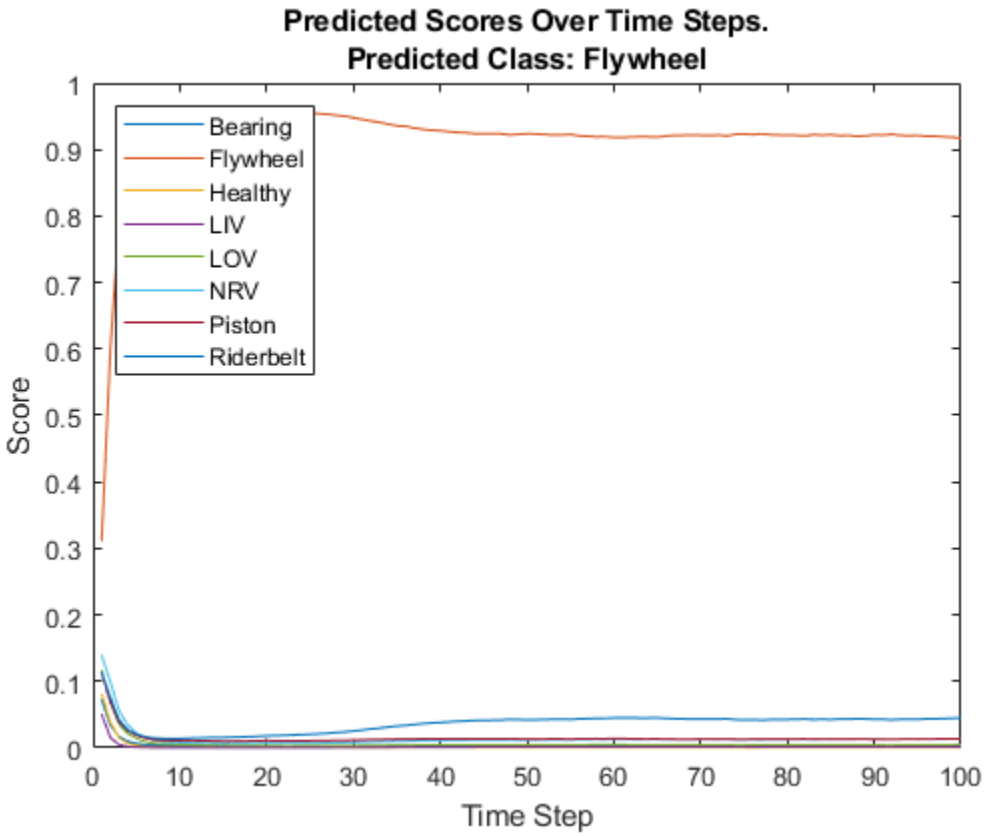
Compute the recognized fault from scores and display it.

```
scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)
```

```
detectedFault =
"Flywheel"
```

Plot the scores of each label for each frame.

```
plot(scores)
legend("" + labels,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)
```



Generate MATLAB Executable

Create a code generation configuration object to generate an executable. Specify the target language as C++.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the MKL-DNN library. Attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('mkldnn');
cfg.DeepLearningConfig = dlcfg;
```

Create an audio data frame of length `windowLength`.

```
audioFrame = ones(windowLength,1);
```

Call the `codegen` (MATLAB Coder) function from MATLAB Coder to generate C++ code for the `recognizeAirCompressorFault` function. Specify the configuration object and prototype arguments. A MEX-file named `recognizeAirCompressorFault_mex` is generated to your current folder.

```
codegen -config cfg recognizeAirCompressorFault -args {audioFrame,resetNetworkState} -report
```

```
Code generation successful: View report
```

Perform Machine Fault Recognition Using MATLAB Executable

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles,~] = splitEachLabel(dataStore,1);
    allData = readall(allFiles);
    signalToBeTested = Riderbelt ;
    signalToBeTested = cell2mat(signalToBeTested);
end
```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use generated `recognizeAirCompressorFault_mex` to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);
resetNetworkState = true;

while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource,windowLength);

    % Apply streaming classifier function
    score = recognizeAirCompressorFault_mex(x,resetNetworkState);

    % Store score for analysis
```

```
    write(scoreBuffer,score);

    resetNetworkState = false;
end
```

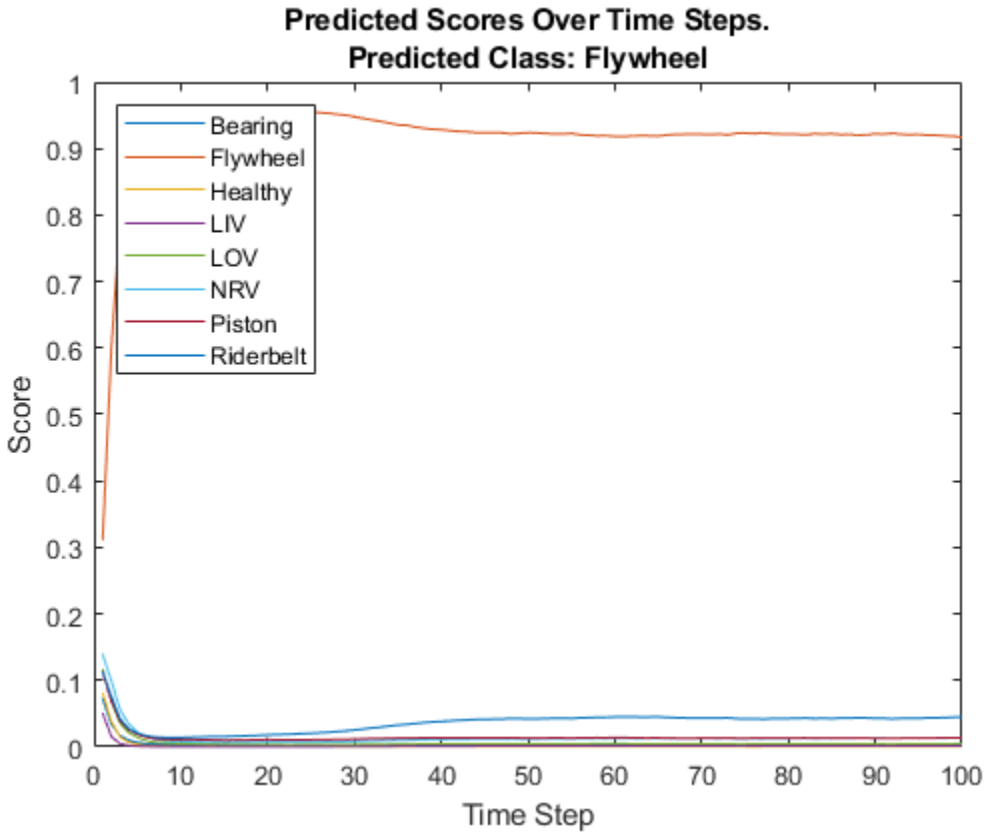
Compute the recognized fault from scores and display it.

```
scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)
```

```
detectedFault =
"Flywheel"
```

Plot the scores of each label for each frame.

```
plot(scores)
legend("" + labels,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)
```



Evaluate Execution Time of Alternative MEX Function Workflow

Use `tic` and `toc` to measure the execution time of MATLAB function `recognizeAirCompressorFault` and MATLAB executable (MEX) `recognizeAirCompressorFault_mex`.

Create a `dsp.AsyncBuffer` (DSP System Toolbox) object to record execution time.

```
timingBufferMATLAB = dsp.AsyncBuffer;
timingBufferMEX = dsp.AsyncBuffer;
```

Use same recording that you chose in previous section as input to `recognizeAirCompressorFault` function and its MEX equivalent `recognizeAirCompressorFault_mex`.

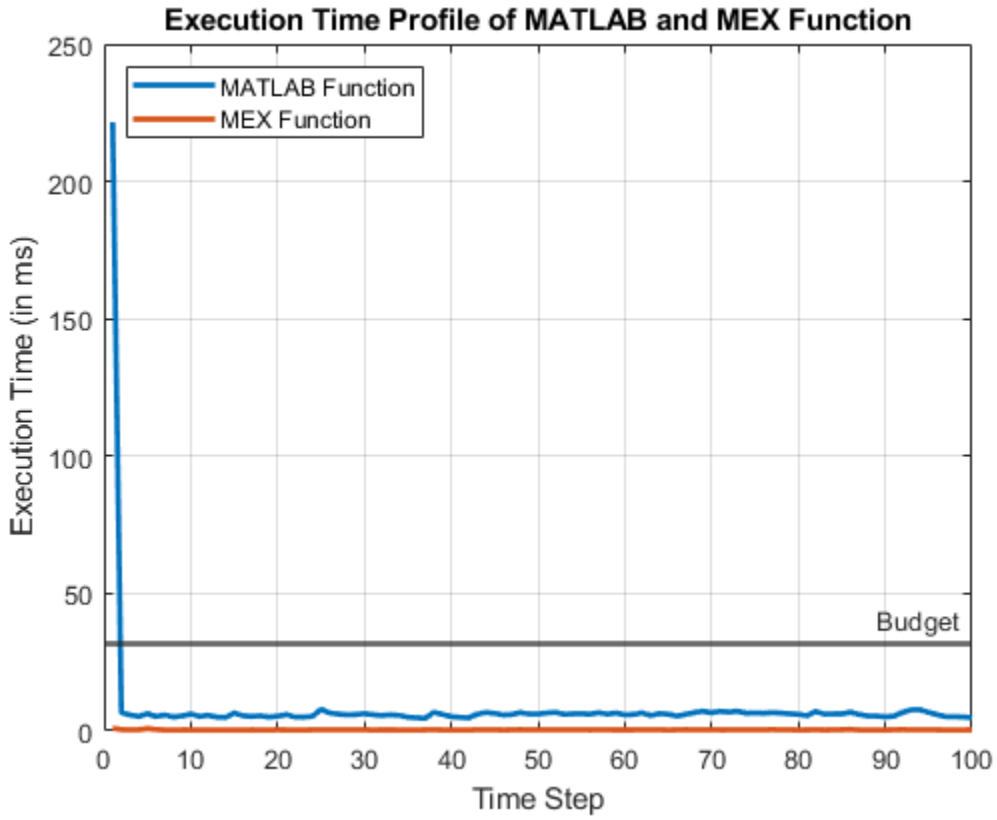
```
write(audioSource,signalToBeTested);
```

Measure the execution time of the MATLAB code.

```
resetNetworkState = true;
while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource,windowLength);

    % Apply streaming classifier function
    tic
    scoreMATLAB = recognizeAirCompressorFault(x,resetNetworkState);
    write(timingBufferMATLAB,toc);

    % Apply streaming classifier MEX function
    tic
    scoreMEX = recognizeAirCompressorFault_mex(x,resetNetworkState);
    write(timingBufferMEX,toc);

    resetNetworkState = false;

end
```

Plot the execution time for each frame and analyze the profile. The first call of `recognizeAirCompressorFault_mex` consumes around four times of the budget as it includes loading of network and resetting of the states. However, in a real, deployed system, that initialization time is only incurred once. The execution time of the MATLAB function is around 10 ms and that of MEX function is ~1 ms, which is well below the 32 ms budget for real-time performance.

```
budget = (windowLength/fs)*1000;
timingMATLAB = read(timingBufferMATLAB)*1000;
timingMEX = read(timingBufferMEX)*1000;
frameNumber = 1:numel(timingMATLAB);
perfGain = timingMATLAB./timingMEX;
plot(frameNumber,timingMATLAB,frameNumber,timingMEX,'LineWidth',2)
grid on
yline(budget,'',{'Budget'},'LineWidth',2)
legend('MATLAB Function','MEX Function','Location','northwest')
xlabel("Time Step")
ylabel("Execution Time (in ms)")
title("Execution Time Profile of MATLAB and MEX Function")
```



Compute the performance gain of MEX over MATLAB function excluding the first call. This performance test is done on a machine using an NVIDIA Quadro P620 (Version 26) GPU and an Intel® Xeon® W-2133 CPU running at 3.60 GHz.

```
PerformanceGain = sum(timingMATLAB(2:end))/sum(timingMEX(2:end))
```

```
PerformanceGain = 15.5501
```

This example ends here. For deploying machine fault recognition on Raspberry Pi, see “Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi” (Audio Toolbox).

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi

This example demonstrates code generation for “Acoustics-Based Machine Fault Recognition” (Audio Toolbox) using a long short-term memory (LSTM) network and spectral descriptors. This example uses MATLAB® Coder™, MATLAB Coder Interface for Deep Learning Libraries, MATLAB Support Package for Raspberry Pi Hardware to generate a standalone executable (`.elf`) file on a Raspberry Pi™ that leverages performance of the Arm Compute Library. The input data consists of acoustics time-series recordings from faulty or healthy air compressors and the output is the state of the mechanical machine predicted by the LSTM network. This standalone executable on Raspberry Pi runs the streaming classifier on the input data received from MATLAB and sends the computed scores for each label to MATLAB. Interaction between MATLAB script and the executable on your Raspberry Pi is handled using the user datagram protocol (UDP). For more details on audio preprocessing and network training, see “Acoustics-Based Machine Fault Recognition” (Audio Toolbox).

Example Requirements

- The MATLAB Coder Interface for Deep Learning Libraries Support Package
- ARM processor that supports the NEON extension
- ARM Compute Library version 20.02.1 (on the target ARM hardware)
- Environment variables for the compilers and libraries

For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)

Prepare Input Dataset

Specify a sample rate `fs` of 16 kHz and a `windowLength` of 512 samples, as defined in “Acoustics-Based Machine Fault Recognition” (Audio Toolbox). Set `numFrames` to `100`.

```
fs = 16000;
windowLength = 512;
numFrames = 100;
```

To run the Example on a test signal, generate a pink noise signal. To test the performance of the system on a real dataset, download the air compressor dataset [1] on page 14-0 .

```
downloadDataset = false

if ~downloadDataset
    pinkNoiseSignal = pinknoise(windowLength*numFrames);
else
    % Download AirCompressorDataset.zip
    component = 'audio';
    filename = 'AirCompressorDataset/AirCompressorDataset.zip';
    localfile = matlab.internal.examples.downloadSupportFile(component,filename);

    % Unzip the downloaded zip file to the downloadFolder
    downloadFolder = fileparts(localfile);
    if ~exist(fullfile(downloadFolder,'AirCompressorDataset'),'dir')
        unzip(localfile, downloadFolder)
    end
```

```
    % Create an audioDatastore object dataStore, to manage, the data.
    dataStore = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames

    % Use countEachLabel to get the number of samples of each category in the dataset.
    countEachLabel(dataStore)
end
```

Recognize Machine Fault in MATLAB

To run the streaming classifier in MATLAB, download and unzip the system developed in "Acoustics-Based Machine Fault Recognition" (Audio Toolbox).

```
component = 'audio';
filename = 'AcousticsBasedMachineFaultRecognition/AcousticsBasedMachineFaultRecognition.zip';
localfile = matlab.internal.examples.downloadSupportFile(component,filename);

downloadFolder = fullfile(fileparts(localfile),'system');
if ~exist(downloadFolder,'dir')
    unzip(localfile,downloadFolder)
end
```

To access the `recognizeAirCompressorFault` function of the system, add `downloadFolder` to the search path.

```
addpath(downloadFolder)
```

Create a `dsp.AsyncBuffer` (DSP System Toolbox) object to read audio in a streaming fashion and a `dsp.AsyncBuffer` (DSP System Toolbox) object to accumulate scores.

```
audioSource = dsp.AsyncBuffer;
scoreBuffer = dsp.AsyncBuffer;
```

Load the pretrained network and extract labels from the network.

```
airCompNet = coder.loadDeepLearningNetwork('AirCompressorFaultRecognitionModel.mat');
labels = string(airCompNet.Layers(end).Classes);
```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles,~] = splitEachLabel(dataStore,1);
    allData = readall(allFiles);
    signalToBeTested = Bearing ;
    signalToBeTested = cell2mat(signalToBeTested);
end
```

Stream one audio frame at a time to represent the system as it would be deployed in a real-time embedded system. Use `recognizeAirCompressorFault` developed in "Acoustics-Based Machine Fault Recognition" (Audio Toolbox) to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);
resetNetworkState = true;
```

```
while audioSource.NumUnreadSamples >= windowLength

    % Get a frame of audio data
    x = read(audioSource,windowLength);

    % Apply streaming classifier function
    score = recognizeAirCompressorFault(x,resetNetworkState);

    % Store score for analysis
    write(scoreBuffer,score);

    resetNetworkState = false;
end
```

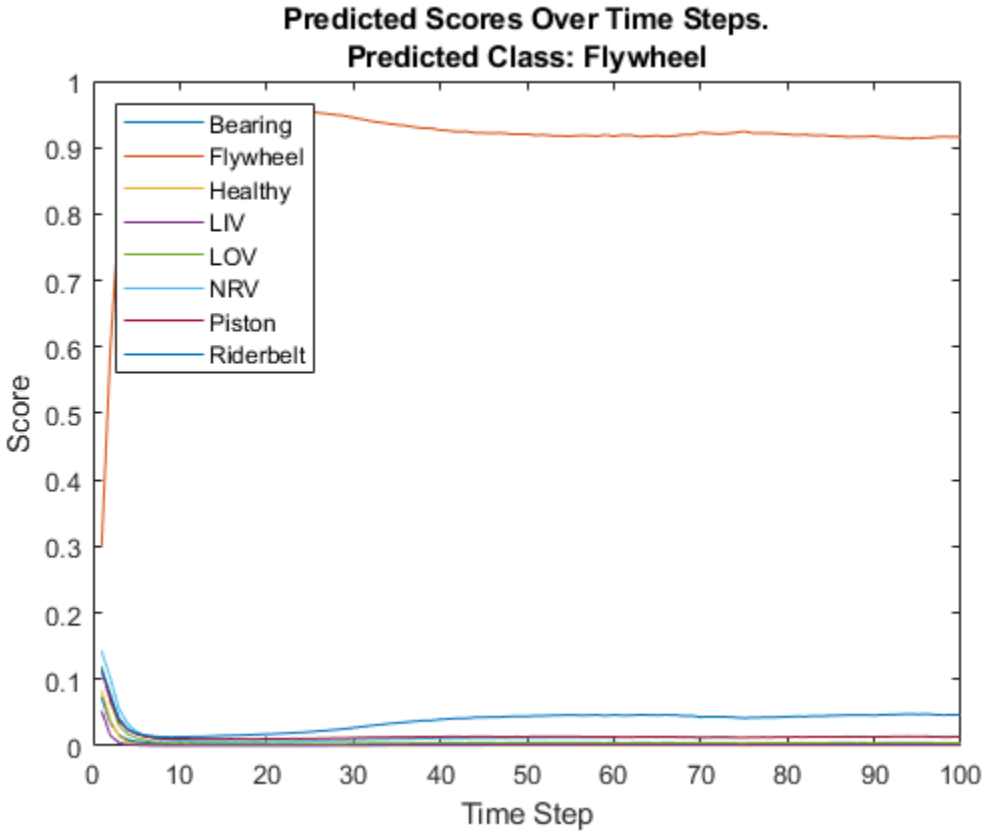
Compute the recognized fault from scores and display it.

```
scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)
```

```
detectedFault =
"Flywheel"
```

Plot the scores of each label for each frame.

```
plot(scores)
legend("" + labels,'Location','northwest')
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)
```



Reset the asynchronous buffer `audioSource`.

```
reset(audioSource)
```

Prepare MATLAB Code For Deployment

This example uses the `dsp.UDPSender` (DSP System Toolbox) System object to send the audio frame to the executable running on Raspberry Pi and the `dsp.UDPReceiver` (DSP System Toolbox) System object to receive the score vector from the Raspberry Pi. Create a `dsp.UDPSender` (DSP System Toolbox) system object to send audio captured in MATLAB to your Raspberry Pi. Set the `targetIPAddress` to the IP address of your Raspberry Pi. Set the `RemoteIPPort` to `25000`. Raspberry Pi receives the input audio frame from the same port using the `dsp.UDPReceiver` (DSP System Toolbox) system object.

```
targetIPAddress = '172.31.164.247';
UDPSend = dsp.UDPSender('RemoteIPPort',25000,'RemoteIPAddress',targetIPAddress);
```

Create a `dsp.UDPReceiver` (DSP System Toolbox) system object to receive predicted scores from your Raspberry Pi. Each UDP packet received from the Raspberry Pi is a vector of scores and each vector element is a score for a state of the air compressor. The maximum message length for the `dsp.UDPReceiver` (DSP System Toolbox) object is 65507 bytes. Calculate the buffer size to accommodate the maximum number of UDP packets.

```
sizeOfDoubleInBytes = 8;
numScores = 8;
maxUDPMessageLength = floor(65507/sizeOfDoubleInBytes);
numPackets = floor(maxUDPMessageLength/numScores);
```

```
bufferSize = numPackets*numScores*sizeOfDoubleInBytes;

UDPReceive = dsp.UDPReceiver("LocalIPPort",21000, ...
    "MessageDataType","single", ...
    "MaximumMessageLength",numScores, ...
    "ReceiveBufferSize",bufferSize);
```

Create a supporting function, `recognizeAirCompressorFaultRaspi`, that receives an audio frame using `dsp.UDPReceiver` (DSP System Toolbox) and applies the streaming classifier and sends the predicted score vector to MATLAB using `dsp.UDPSender` (DSP System Toolbox).

```
type recognizeAirCompressorFaultRaspi
```

```
function recognizeAirCompressorFaultRaspi(hostIPAddress)
% This function receives acoustic input using dsp.UDPReceiver and runs a
% streaming classifier by calling recognizeAirCompressorFault, developed in
% the Acoustics-Based Machine Fault Recognition - MATLAB Example.
% Computed scores are sent to MATLAB using dsp.UDPSender.
%#codegen

%   Copyright 2021 The MathWorks, Inc.

frameLength = 512;

% Configure UDP Sender System Object
UDPSend = dsp.UDPSender('RemoteIPPort',21000,'RemoteIPAddress',hostIPAddress);

% Configure UDP Receiver system object
sizeOfDoubleInBytes = 8;
maxUDPMessageLength = floor(65507/sizeOfDoubleInBytes);
numPackets = floor(maxUDPMessageLength/frameLength);
bufferSize = numPackets*frameLength*sizeOfDoubleInBytes;
UDPReceiveRaspi = dsp.UDPReceiver('LocalIPPort',25000, ...
    'MaximumMessageLength',frameLength, ...
    'ReceiveBufferSize',bufferSize, ...
    'MessageDataType','double');

% Reset network state for first call
resetNetworkState = true;

while true
    % Receive audio frame of size frameLength x 1
    x = UDPReceiveRaspi();

    if(~isempty(x))

        x = x(1:frameLength,1);

        % Apply streaming classifier function
        scores = recognizeAirCompressorFault(x,resetNetworkState);

        %Send output to the host machine
        UDPSend(scores);

        resetNetworkState = false;
    end
end
```

Generate Executable on Raspberry Pi

Replace the `hostIPAddress` with your machine's address. Your Raspberry Pi sends the predicted scores to the IP address you specify.

```
hostIPAddress = coder.Constant('172.18.230.30');
```

Create a code generation configuration object to generate an executable program. Specify the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Create a configuration object for deep learning code generation with the ARM compute library that is on your Raspberry Pi. Specify the architecture of the Raspberry Pi and attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '20.02.1';
cfg.DeepLearningConfig = dlcfg;
```

Use the Raspberry Pi Support Package function `raspi` to create a connection to your Raspberry Pi. In the next block of code, replace:

- `raspiname` with the name of your Raspberry Pi
- `pi` with your user name
- `password` with your password

```
if (~exist('r','var'))
  r = raspi('raspiname','pi','password');
end
```

Create a `coder.hardware` (MATLAB Coder) object for Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify the build folder on the Raspberry Pi.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Use an autogenerated C++ main file to generate a standalone executable.

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
```

Call the `codegen` (MATLAB Coder) function from MATLAB Coder to generate C++ code and the executable on your Raspberry Pi. By default, the Raspberry Pi executable has the same name as the MATLAB function. You get a warning in the code generation logs `that` you can disregard because `recognizeAirCompressorFaultRaspi` has an infinite loop that looks for an audio frame from MATLAB.

```
codegen -config cfg recognizeAirCompressorFaultRaspi -args {hostIPAddress} -report
```

```
 Deploying code. This may take a few minutes.
Warning: Function 'recognizeAirCompressorFaultRaspi' does not terminate due to an infinite loop.
```

```
Warning in ==> recognizeAirCompressorFaultRaspi Line: 1 Column: 1
Code generation successful (with warnings): View report
```

Perform Machine Fault Recognition Using Deployed Code

Create a command to open the `recognizeAirCompressorFaultRaspi` application on a Raspberry Pi. Use `system` to send the command to your Raspberry Pi.

```
applicationName = 'recognizeAirCompressorFaultRaspi';

applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName',applicationName);
targetDirPath = applicationDirPaths{1}.directory;

exeName = strcat(applicationName,'.elf');
command = ['cd ',targetDirPath,'; ./',exeName,' &> 1 &'];

system(r,command);
```

Initialize `signalToBeTested` to `pinkNoiseSignal` or select a signal from the drop-down list to test the file of your choice from the dataset.

```
if ~downloadDataset
    signalToBeTested = pinkNoiseSignal;
else
    [allFiles,~] = splitEachLabel(dataStore,1);
    allData = readall(allFiles);
    signalToBeTested = Bearing ;
    signalToBeTested = cell2mat(signalToBeTested);
end
```

Stream one audio frame at a time to represent a system as it would be deployed in a real-time embedded system. Use the generated MEX file `recognizeAirCompressorFault_mex` to compute audio features and perform deep learning classification.

```
write(audioSource,signalToBeTested);

while audioSource.NumUnreadSamples >= windowLength
    x = read(audioSource,windowLength);
    UDPSend(x);
    score = UDPReceive();
    if ~isempty(score)
        write(scoreBuffer,score');
    end
end
```

Compute the recognized fault from scores and display it.

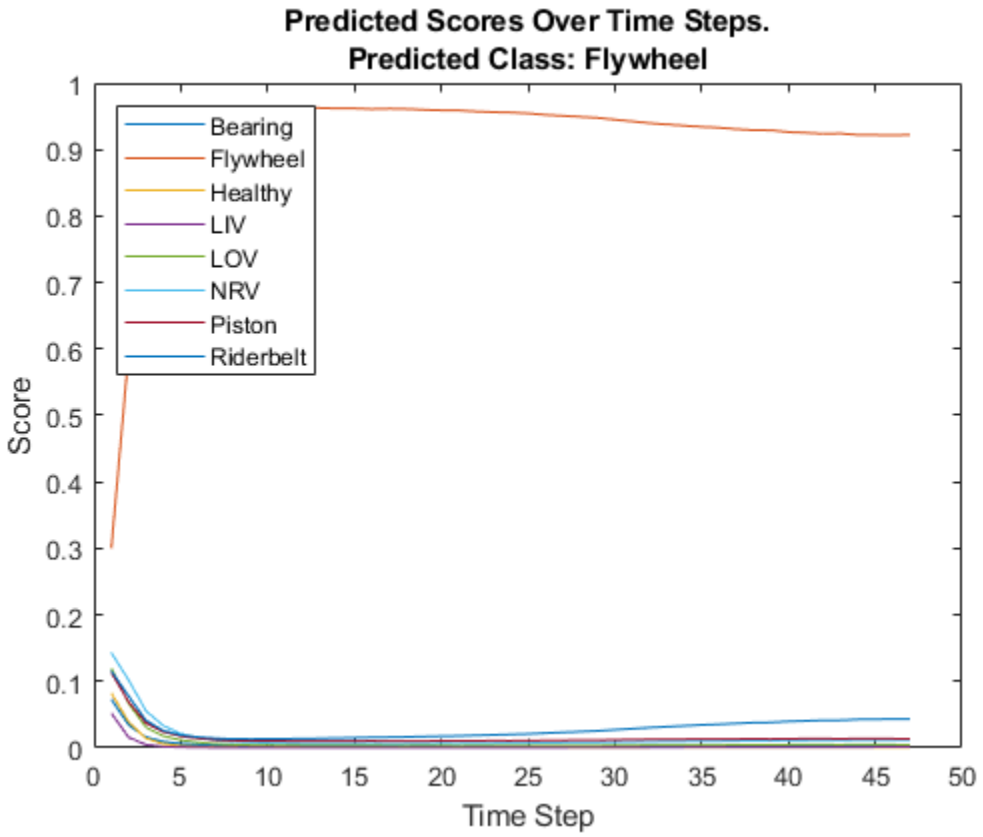
```
scores = read(scoreBuffer);
[~,labelIndex] = max(scores(end,:),[],2);
detectedFault = labels(labelIndex)
```

```
detectedFault =
"Flywheel"
```

Plot the scores of each label for each frame.

```
plot(scores)
legend("" + labels,'Location','northwest')
```

```
xlabel("Time Step")
ylabel("Score")
str = sprintf("Predicted Scores Over Time Steps.\nPredicted Class: %s",detectedFault);
title(str)
```



Terminate the standalone executable running on Raspberry Pi.

```
stopExecutable(codertarget.raspi.raspberrypi,exeName)
```

Evaluate Execution Time Using Alternative PIL Function Workflow

To evaluate execution time taken by standalone executable on Raspberry Pi, use a PIL (processor-in-loop) workflow. To perform PIL profiling, generate a PIL function for the supporting function `recognizeAirCompressorFault`.

Create a code generation configuration object to generate the PIL function.

```
cfg = coder.config('lib','ecoder',true);
cfg.VerificationMode = 'PIL';
```

Set the ARM compute library and architecture.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
cfg.DeepLearningConfig = dlcfg ;
cfg.DeepLearningConfig.ArmArchitecture = 'armv7';
cfg.DeepLearningConfig.ArmComputeVersion = '20.02.1';
```

Set up the connection with your target hardware.

```
if (~exist('r','var'))
  r = raspi('raspiname','pi','password');
end
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Set the build directory and target language.

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
cfg.TargetLang = 'C++';
```

Enable profiling and generate the PIL code. A MEX file named `recognizeAirCompressorFault_pil` is generated in your current folder.

```
cfg.CodeExecutionProfiling = true;
audioFrame = ones(windowLength,1);
resetNetworkStateFlag = true;
codegen -config cfg recognizeAirCompressorFault -args {audioFrame,resetNetworkStateFlag}
```

```
 Deploying code. This may take a few minutes.
### Connectivity configuration for function 'recognizeAirCompressorFault': 'Raspberry Pi'
Location of the generated elf : /home/pi/remoteBuildDir/MATLAB_ws/R2021b/S/MATLAB/Examples/Examp
Code generation successful.
```

Call the generated PIL function 50 times to get the average execution time.

```
totalCalls = 50;

for k = 1:totalCalls
    x = pinknoise(windowLength,1);
    score = recognizeAirCompressorFault_pil(x,resetNetworkStateFlag);
    resetNetworkStateFlag = false;
end
```

```
### Starting application: 'codegen\lib\recognizeAirCompressorFault\pil\recognizeAirCompressorFau
    To terminate execution: clear recognizeAirCompressorFault_pil
### Launching application recognizeAirCompressorFault.elf...
    Execution profiling data is available for viewing. Open Simulation Data Inspector.
    Execution profiling report available after termination.
```

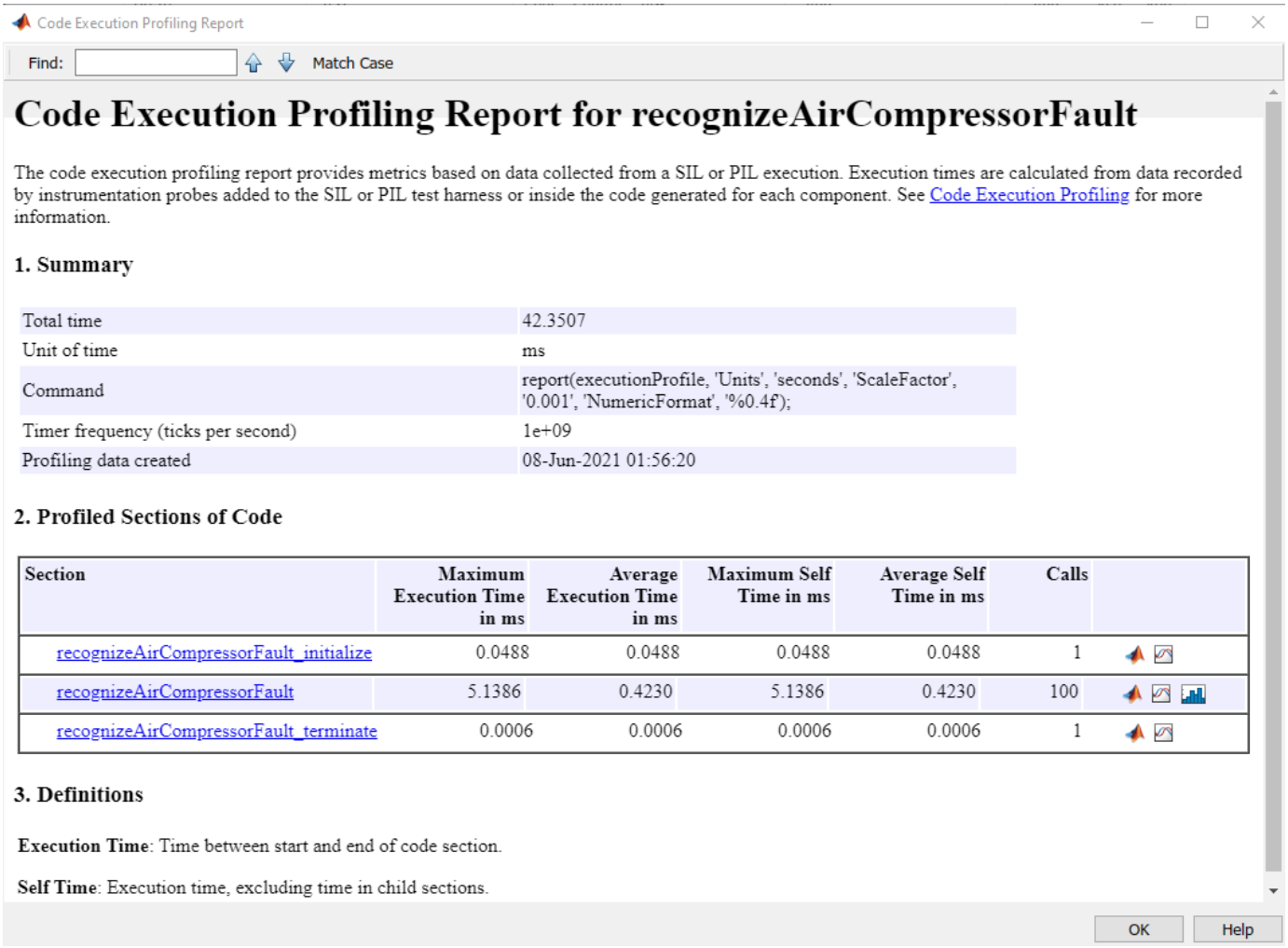
Terminate the PIL execution.

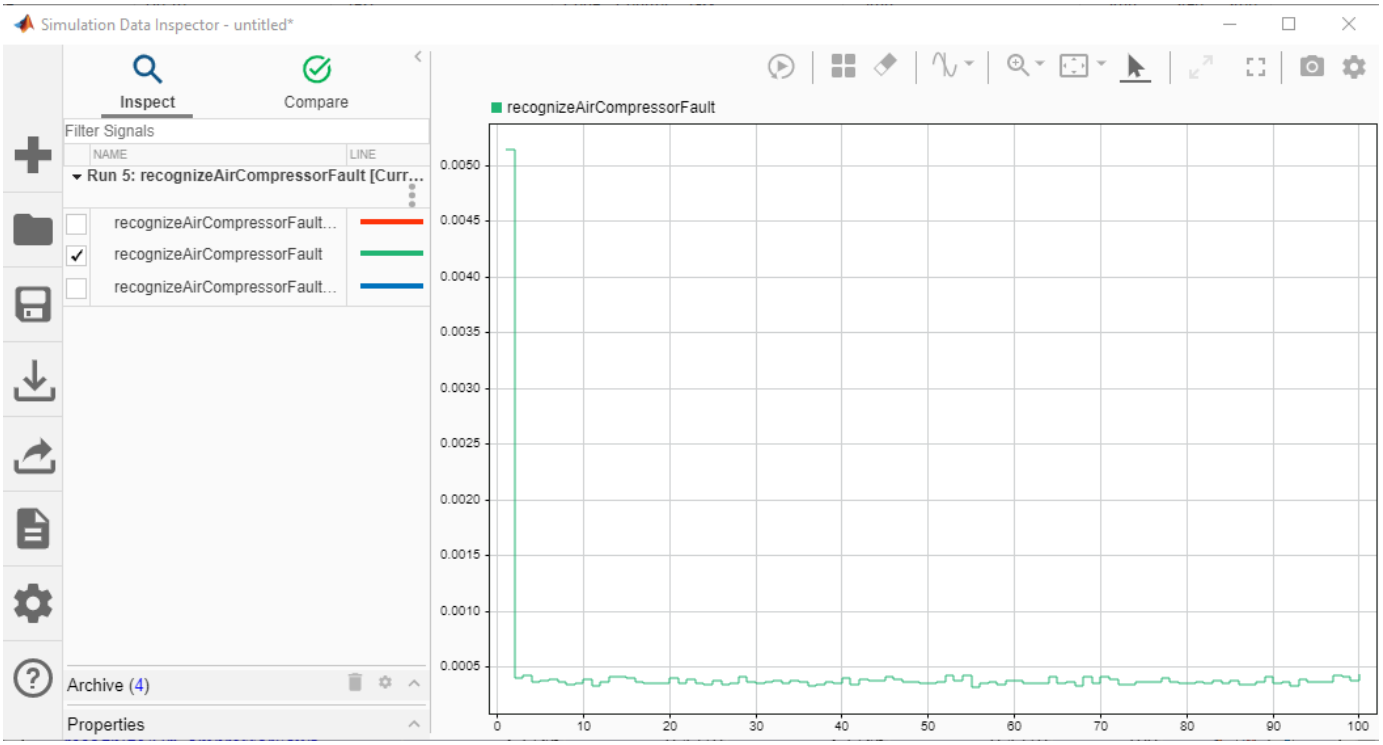
```
clear recognizeAirCompressorFault_pil
```

```
### Host application produced the following standard output (stdout) and standard error (stderr)

### Connectivity configuration for function 'recognizeAirCompressorFault': 'Raspberry Pi'
    Execution profiling report: report(getCoderExecutionProfile('recognizeAirCompressorFault'))
```

Generate an execution profile report to evaluate execution time.

```
executionProfile = getCoderExecutionProfile('recognizeAirCompressorFault');
report(executionProfile, ...
       'Units','Seconds', ...
       'ScaleFactor','1e-03', ...
       'NumericFormat','%0.4f');
```


Code Execution Profiling Report
Find:
Match Case
Code Execution Profiling Report for recognizeAirCompressorFault
The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.
1. Summary
Total time 42.3507
Unit of time ms
Command report(executionProfile, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%0.4f');
Timer frequency (ticks per second) 1e+09
Profiling data created 08-Jun-2021 01:56:20
2. Profiled Sections of Code
Section
Maximum Execution Time in ms
Average Execution Time in ms
Maximum Self Time in ms
Average Self Time in ms
Calls
recognizeAirCompressorFault_initialize 0.0488 0.0488 0.0488 0.0488 1
recognizeAirCompressorFault 5.1386 0.4230 5.1386 0.4230 100
recognizeAirCompressorFault_terminate 0.0006 0.0006 0.0006 0.0006 1
3. Definitions
Execution Time: Time between start and end of code section.
Self Time: Execution time, excluding time in child sections.
OK
Help



The average execution time of `recognizeAirCompressorFault_pil` function is `0.423` `ms`, which is well below the `32` `ms` budget for real-time performance. The first call of `recognizeAirCompressorFault_pil` consumes around `12` times of the average execution time as it includes loading of network and resetting of the states. However, in a real, deployed system, that initialization time is incurred only once. This example ends here. For deploying machine fault recognition on desktops, see “Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN” (Audio Toolbox).

References

[1] Verma, Nishchal K., et al. "Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors." *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. DOI.org (Crossref), doi:10.1109/TR.2015.2459684.

End-to-End Deep Speech Separation

This example showcases an end-to-end deep learning network for speaker-independent speech separation.

Introduction

Speech separation is a challenging and critical speech processing task. A number of speech separation methods based on deep learning have been proposed recently, most of which rely on time-frequency transformations of the time-domain audio mixture (See “Cocktail Party Source Separation Using Deep Learning Networks” (Audio Toolbox) for an implementation of such a deep learning system).

Solutions based on time-frequency methods suffer from two main drawbacks:

- The conversion of the time-frequency representations back to the time domain requires phase estimation, which introduces errors and leads to imperfect reconstruction.
- Relatively long windows are required to yield high resolution frequency representations, which leads to high computational complexity and unacceptable latency for real-time scenarios.

In this example, you explore a deep learning speech separation network (based on [1]) which acts directly on the audio signal and bypasses the issues arising from time-frequency transformations.

Separate Speech using the Pretrained Network

Download the Pretrained Network

Before training the deep learning network from scratch, you will use a pretrained version of the network to separate two speakers from an example mixture signal.

First, download the pretrained network and example audio files.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/speechSeparation.zip';
downloadNetFolder = tempdir;
netFolder = fullfile(downloadNetFolder,'speechSeparation');

if ~exist(netFolder,'dir')
    disp('Downloading pretrained network and audio files ...')
    unzip(url,downloadNetFolder)
end
```

Prepare Test Signal

Load two audio signals corresponding to two different speakers. Both signals are sampled at 8 kHz.

```
Fs = 8000;
s1 = audioread(fullfile(netFolder,'speaker1.wav'));
s2 = audioread(fullfile(netFolder,'speaker2.wav'));
```

Normalize the signals.

```
s1 = s1/max(abs(s1));
s2 = s2/max(abs(s2));
```

Listen to a few seconds of each signal.

```
T = 5;
sound(s1(1:T*Fs))
pause(T)

sound(s2(1:T*Fs))
pause(T)
```

Combine the two signals into a mixture signal.

```
mix = s1+s2;
mix = mix/max(abs(mix));
```

Listen to the first few seconds of the mixture signal.

```
sound(mix(1:T*Fs))
pause(T)
```

Separate Speakers

Load the parameters of the pretrained speech separation network.

```
load(fullfile(netFolder,'paramsBest.mat'),'learnables','states')
```

Separate the two speakers in the mixture signals by calling the `separateSpeakers` function.

```
[z1,z2] = separateSpeakers(mix,learnables,states,false);
```

Listen to the first few seconds of the first estimated speech signal.

```
sound(z1(1:T*Fs))
pause(T)
```

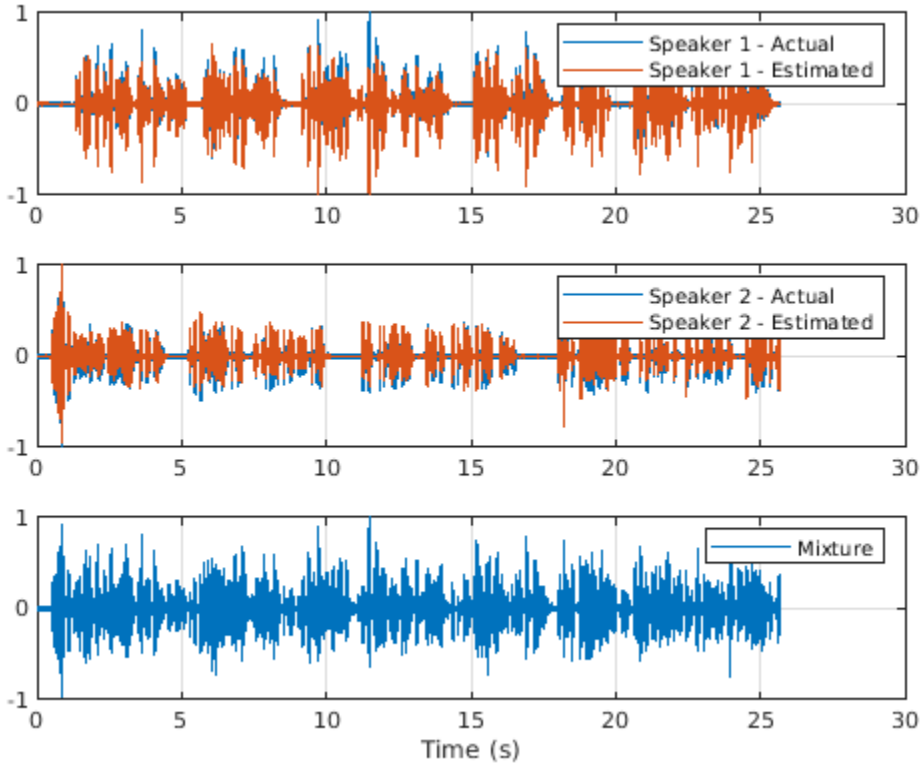
Listen to the second estimated signal.

```
sound(z2(1:T*Fs))
pause(T)
```

To illustrate the effect of speech separation, plot the estimated and original separated signals along with the mixture signal.

```
s1 = s1(1:length(z1));
s2 = s2(1:length(z2));
mix = mix(1:length(s1));

t  = (0:length(s1)-1)/Fs;

figure;
subplot(311)
plot(t,s1)
hold on
plot(t,z1)
grid on
legend('Speaker 1 - Actual','Speaker 1 - Estimated')
subplot(312)
plot(t,s2)
hold on
plot(t,z2)
grid on
legend('Speaker 2 - Actual','Speaker 2 - Estimated')
```

```
subplot(313)
plot(t,mix)
grid on
legend('Mixture')
xlabel('Time (s)')
```



Compare to a Time-Frequency Transformation Deep Learning Network

Next, you compare the performance of the network to the network developed in the “Cocktail Party Source Separation Using Deep Learning Networks” (Audio Toolbox) example. This speech separation network is based on traditional time-frequency representations of the audio mixture (using the short-time Fourier transform, STFT, and the inverse short-time Fourier transform, ISTFT).

Download the pretrained network.

```
url = 'http://ssd.mathworks.com/supportfiles/audio/CocktailPartySourceSeparation.zip';

downloadNetFolder = tempdir;
cocktailNetFolder = fullfile(downloadNetFolder,'CocktailPartySourceSeparation');

if ~exist(cocktailNetFolder,'dir')
    disp('Downloading pretrained network and audio files (5 files - 24.5 MB) ...')
    unzip(url,downloadNetFolder)
end
```

The function `separateSpeakersTimeFrequency` encapsulates the steps required to separate speech using this network. The function performs the following steps:

- Compute the magnitude STFT of the input time-domain mixture.
- Compute a soft time-frequency mask by passing the STFT to the network.
- Compute the STFT of the separated signals by multiplying the mixture STFT by the mask.
- Reconstruct the time-domain separated signals using ISTFT. The phase of the mixture STFT is used.

Refer to the “Cocktail Party Source Separation Using Deep Learning Networks” (Audio Toolbox) example for more details about this network.

Separate the two speakers.

```
[y1,y2] = separateSpeakersTimeFrequency(mix,cocktailNetFolder);
```

Listen to the first separated signal.

```
sound(y1(1:Fs*T))
pause(T)
```

Listen to the second separated signal.

```
sound(y2(1:Fs*T))
pause(T)
```

Evaluate Network Performance using SI-SNR

You will compare the two networks using the scale-invariant source-to-noise ratio (SI-SNR) objective measure [1].

Compute the SISNR for the first speaker with the end-to-end network.

First, normalize the actual and estimated signals.

```
s10 = s1 - mean(s1);
z10 = z1 - mean(z1);
```

Compute the "signal" component of the SNR.

```
t = sum(s10.*z10) .* z10 ./ (sum(z10.^2)+eps);
```

Compute the "noise" component of the SNR.

```
n = s1 - t;
```

Now compute the SI-SNR (in dB).

```
v1 = 20*log((sqrt(sum(t.^2))+eps)./sqrt((sum(n.^2))+eps))/log(10);
fprintf('End-to-end network - Speaker 1 SISNR: %f dB\n',v1)
```

```
End-to-end network - Speaker 1 SISNR: 14.316869 dB
```

The SI-SNR computation steps are encapsulated in the function `SISNR`. Use the function to compute the SI-SNR of the second speaker with the end-to-end network.

```
v2 = SISNR(z2,s2);
fprintf('End-to-end network - Speaker 2 SISNR: %f dB\n',v2)
```

```
End-to-end network - Speaker 2 SISNR: 13.706421 dB
```

Next, compute the SI-SNR for each speaker for the STFT-based network.

```
w1 = SISNR(y1,s1(1:length(y1)));
w2 = SISNR(y2,s2(1:length(y2)));
fprintf('STFT network - Speaker 1 SISNR: %f dB\n',w1)
```

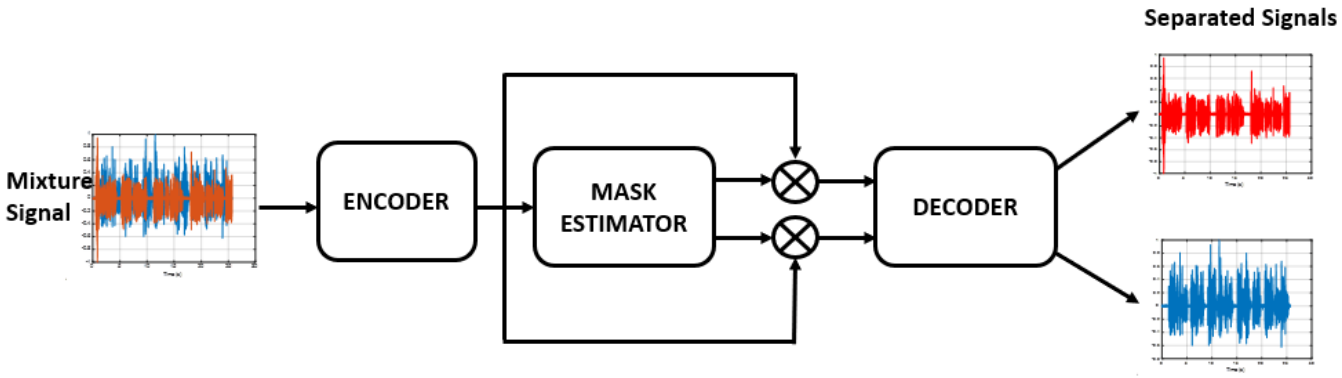
```
STFT network - Speaker 1 SISNR: 7.003789 dB
```

```
fprintf('STFT network - Speaker 2 SISNR: %f dB\n',w2)
```

```
STFT network - Speaker 2 SISNR: 7.382209 dB
```

Training the Speech Separation Network

Examine the Network Architecture



The network is based on [1] and consists of three stages: Encoding, mask estimation or separation, and decoding.

- The encoder transforms the time-domain input mixture signals into an intermediate representation using convolutional layers.
- The mask estimator computes one mask per speaker. The intermediate representation of each speaker is obtained by multiplying the encoder's output by its respective mask. The mask estimator is comprised of 32 blocks of convolutional and normalization layers with skip connections between blocks.
- The decoder transforms the intermediate representations to time-domain separated speech signals using transposed convolutional layers.

The operation of the network is encapsulated in `separateSpeakers`.

Optionally Reduce the Dataset Size

To train the network with the entire dataset and achieve the highest possible accuracy, set `reduceDataset` to false. To run this example quickly, set `reduceDataset` to true. This will run the rest of the example on only a handful of files.

```
reduceDataset = true;
```

Download the Training Dataset

You use a subset of the LibriSpeech Dataset [2] to train the network. The LibriSpeech Dataset is a large corpus of read English speech sampled at 16 kHz. The data is derived from audiobooks read from the LibriVox project.

Download the LibriSpeech dataset. If `reduceDataset` is true, this steo is skipped.

```
downloadDatasetFolder = tempdir;
datasetFolder = fullfile(downloadDatasetFolder,"LibriSpeech","train-clean-360");
if ~reduceDataset
    filename = "train-clean-360.tar.gz";
    url = "http://www.openSLR.org/resources/12/" + filename;
    if ~isfolder(datasetFolder)
        gunzip(url,downloadDatasetFolder);
        unzippedFile = fullfile(downloadDatasetFolder,filename);
        untar(unzippedFile{1}(1:end-3),downloadDatasetFolder);
    end
end
```

Preprocess the Dataset

The LibriSpeech dataset is comprised of a large number of audio files with a single speaker. It does not contain mixture signals where 2 or more persons are speaking simultaneously.

You will process the original dataset to create a new dataset that is suitable for training the speech separation network.

The steps for creating the training dataset are encapsulated in `createTrainingDataset`. The function creates mixture signals comprised of utterances of two random speakers. The function returns three audio datastores:

- `mixDatastore` points to mixture files (where two speakers are talking simultaneously).
- `speaker1Datastore` points to files containing the isolated speech of the first speaker in the mixture.
- `speaker2Datastore` points to files containing the isolated speech of the second speaker in the mixture.

```
miniBatchSize = 4;
[mixDatastore,speaker1Datastore,speaker2Datastore] = createTrainingDataset(netFolder,datasetFolde
```

Combine the datastores. This ensures that the files stay in the correct order when you shuffle them at the start of each new epoch in the training loop.

```
ds = combine(mixDatastore,speaker1Datastore,speaker2Datastore);
```

Create a minibatch queue from the datastore.

```
mqueue = minibatchqueue(ds,'MiniBatchSize',miniBatchSize,'OutputEnvironment','cpu','OutputAsDlar
```

Specify Training Options

Define training parameters.

Train for 10 epochs.

```
if reduceDataset
    numEpochs = 1;
```

```
else
    numEpochs = 10; %#ok
end
```

Specify the options for Adam optimization. Set the initial learning rate to 1e-3. Use a gradient decay factor of 0.9 and a squared gradient decay factor of 0.999.

```
learnRate = 1e-3;
averageGrad = [];
averageSqGrad = [];

gradDecay = 0.9;
sqGradDecay = 0.999;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™.

```
executionEnvironment = "auto"; % Change to "gpu" to train on a GPU.

duration = 4 * 8000;
```

Set Up Validation Data

You will use the test signal you previously employed to test the pretrained network to compute a validation SI-SNR periodically during training.

If a GPU is available, move the validation signal to the GPU.

```
mix = dlarray(mix,'SCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    mix = gpuArray(mix);
end
```

Define the number of iterations between validation SI-SNR computations.

```
numIterPerValidation = 50;
```

Define a vector to hold the validation SI-SNR from each iteration.

```
valSNR = [];
```

Define a variable to hold the best validation SI-SNR.

```
bestSNR = -Inf;
```

Define a variable to hold the epoch in which the best validation score occurred.

```
bestEpoch = 1;
```

Initialize Network

Initialize the network parameters. `learnables` is a structure containing the learnable parameters from the network layers. `states` is a structure containing the states from the normalization layers.

```
[learnables,states] = initializeNetworkParams;
```

Train the Network

Execute the training loop. This can take many hours to run.

Note that there is no a priori way to associate the estimated output speaker signals with the expected speaker signals. This is resolved by using Utterance-level permutation invariant training (uPIT) [1]. The loss is based on computing the SI-SNR. uPIT minimizes the loss over all permutations between outputs and targets. It is defined in the function `uPIT`.

The validation SI-SNR is computed periodically. If the SI-SNR is the best value to-date, the network parameters are saved to `params.mat`.

```
iteration = 0;

% Loop over epochs.
for jj =1:numEpochs

    % Shuffle the data
    shuffle(mqueue);

    while hasdata(mqueue)

        % Compute validation loss/SNR periodically
        if mod(iteration,numIterPerValidation)==0

            [z1,z2] = separateSpeakers(mix, learnables,states,false);

            l = uPIT(z1,s1,z2,s2);
            valSNR(end+1) = l; %#ok

            if l > bestSNR
                bestSNR = l;
                bestEpoch = jj;
                filename = 'params.mat';
                save(filename,'learnables','states');
            end
        end

        iteration = iteration + 1;

        % Get a new batch of training data
        [x1Batch,x2Batch,mixBatch] = next(mqueue);
        x1Batch = reshape(x1Batch,[duration 1 miniBatchSize]);
        x2Batch = reshape(x2Batch,[duration 1 miniBatchSize]);
        mixBatch = reshape(mixBatch,[duration 1 miniBatchSize]);

        x1Batch = dlarray(x1Batch,'SCB');
        x2Batch = dlarray(x2Batch,'SCB');
        mixBatch = dlarray(mixBatch,'SCB');

        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            x1Batch = gpuArray(x1Batch);
            x2Batch = gpuArray(x2Batch);
            mixBatch = gpuArray(mixBatch);
        end

        % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
        [gradients,states] = dlfeval( @modelGradients,mixBatch,x1Batch,x2Batch,learnables,states

        % Update the network parameters using the ADAM optimizer.
        [learnables,averageGrad,averageSqGrad] = adamupdate(learnables,gradients,averageGrad,aver
```

```
    end

    % Reduce the learning rate if the validation accuracy did not improve
    % during the epoch
    if bestEpoch ~= jj
        learnRate = learnRate/2;
    end
end
```

Plot the validation SNR values.

```
if ~reduceDataset
    valIterNum = 0:length(valSNR)-1;
    figure
    semilogx(numIterPerValidation*(valIterNum-1),valSNR,'b*-')
    grid on
    xlabel('Iteration #')
    ylabel('Validation SINR (dB)')
    valFig.Visible = 'on';
end
```

References

[1] Yi Luo, Nima Mesgarani, "Conv-tasnet: Surpassing ideal time–frequency magnitude masking for speech separation," 2019 IEEE/ACM transactions on audio, speech, and language processing, vol. 29, issue 8, pp. 1256-1266.

[2] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, QLD, 2015, pp. 5206-5210, doi: 10.1109/ICASSP.2015.7178964

Supporting Functions

```
function [mixDatastore,speaker1Datastore,speaker2Datastore] = createTrainingDataset(netFolder,da
% createTrainingDataset Create training dataset

newDatasetPath = fullfile(downloadDatasetFolder,'speech-sep-dataset');

%Create the new dataset folders if they do not exist already.
processDataset = ~isfolder(newDatasetPath);
if processDataset
    mkdir(newDatasetPath);
    mkdir([newDatasetPath '/sp1']);
    mkdir([newDatasetPath '/sp2']);
    mkdir([newDatasetPath '/mix']);
end

%Create an audioDatastore that points to the LibriSpeech dataset.
if reduceDataset
    ads = audioDatastore([repmat({fullfile(netFolder,'speaker1.wav')},1,4),...
                          repmat({fullfile(netFolder,'speaker2.wav')},1,4)]);
else
    ads = audioDatastore(datasetFolder,'IncludeSubfolders',true);
end

% The LibriSpeech dataset is comprised of signals from different speakers.
% The unique speaker ID is encoded in the audio file names.
```

```
% Extract the speaker IDs from the file names.
if reduceDataset
    ads.Labels = categorical([repmat({'1'},1,4),repmat({'2'},1,4)]);
else
    ads.Labels = categorical(extractBetween(ads.Files,fullfile(datasetFolder,filesep),filesep));
end

% You will create mixture signals comprised of utterances of two random speakers.
% Randomize the IDs of all the speakers.
names = unique(ads.Labels);
names = names(randperm(length(names)));

% In this example, you create training signals based on 400 speakers. You
% generate mixture signals based on combining utterances from 200 pairs of
% speakers.

% Define the two groups of speakers.
numPairs = min(200,floor(numel(names)/2));
n1 = names(1:numPairs);
n2 = names(numPairs+1:2*numPairs);

% Create the new dataset. For each pair of speakers:
% * Use subset to create two audio datastores, each containing files
%   corresponding to their respective speaker.
% * Adjust the datastores so that they have the same number of files.
% * Combine the two datastores using combine.
% * Use writeall to preprocess the files of the combined datastore and write
%   the new resulting signals to disk.

% The preprocessing steps performed to create the signals before writing
% them to disk are encapsulated in the function createTrainingFiles. For
% each pair of signals:
% * You downsample the signals from 16 kHz to 8 kHz.
% * You randomly select 4 seconds from each downsampled signal.
% * You create the mixture by adding the 2 signal chunks.
% * You adjust the signal power to achieve a randomly selected
%   signal-to-noise value in the range [-5,5] dB.
% * You write the 3 signals (corresponding to the first speaker, the second
%   speaker, and the mixture, respectively) to disk.
parfor index=1:length(n1)
    spkInd1 = n1(index);
    spkInd2 = n2(index);
    spk1ds = subset(ads,ads.Labels==spkInd1);
    spk2ds = subset(ads,ads.Labels==spkInd2);
    L = min(length(spk1ds.Files),length(spk2ds.Files));
    L = floor(L/miniBatchSize) * miniBatchSize;
    spk1ds = subset(spk1ds,1:L);
    spk2ds = subset(spk2ds,1:L);
    pairds = combine(spk1ds,spk2ds);
    writeall(pairds,newDatasetPath,'FolderLayout','flatten','WriteFcn',@(data,writeInfo,outputFm
end

% Create audio datastores pointing to the files corresponding to the individual speakers and the
mixDatastore = audioDatastore(fullfile(newDatasetPath,'mix'));
speaker1Datastore = audioDatastore(fullfile(newDatasetPath,'sp1'));
speaker2Datastore = audioDatastore(fullfile(newDatasetPath,'sp2'));
end
```

```
function mix = createTrainingFiles(data,writeInfo,~,varargin)
% createTrainingFiles - Preprocess the training signals and write them to disk

reduceDataset = varargin{1};

duration = 4*8000;

x1 = data{1};
x2 = data{2};

% Resample from 16 kHz to 8 kHz
if ~reduceDataset
    x1 = resample(x1,1,2);
    x2 = resample(x2,1,2);
end

% Read a chunk from the first speaker signal
if length(x1)<=duration
    x1 = [x1;zeros(duration-length(x1),1)];
else
    startInd = randi([1 length(x1)-duration],1);
    endInd = startInd + duration - 1;
    x1 = x1(startInd:endInd);
end

% Read a chunk from the second speaker signal
if length(x2)<=duration
    x2 = [x2;zeros(duration-length(x2),1)];
else
    startInd = randi([1 length(x2)-duration],1);
    endInd = startInd + duration - 1;
    x2 = x2(startInd:endInd);
end

x1 = x1./max(abs(x1));
x2 = x2./max(abs(x2));

% SNR [-5 5] dB
s = snr(x1,x2);
targetSNR = 10 * (rand - 0.5);
x1b = 10^((targetSNR-s)/20) * x1;
mix = x1b + x2;
mix = mix./max(abs(mix));

if reduceDataset
    [~,n] = fileparts(tempname);
    name = sprintf('%s.wav',n);
else
    [~,s1] = fileparts(writeInfo.ReadInfo{1}.FileName);
    [~,s2] = fileparts(writeInfo.ReadInfo{2}.FileName);
    name = sprintf('%s-%s.wav',s1,s2);
end

audiowrite(sprintf('%s',fullfile(writeInfo.Location,'sp1',name)),x1,8000);
audiowrite(sprintf('%s',fullfile(writeInfo.Location,'sp2',name)),x2,8000);
audiowrite(sprintf('%s',fullfile(writeInfo.Location,'mix',name)),mix,8000);
```

```
end

function [grad, states] = modelGradients(mix,x1,x2,learnables,states,miniBatchSize)
% modelGradients Compute the model gradients

[y1,y2,states] = separateSpeakers(mix,learnables,states,true);

m = uPIT(x1,y1,x2,y2);
l = sum(m);
loss = -l./miniBatchSize;

grad = dlgradient(loss,learnables);

end

function m = uPIT(x1,y1,x2,y2)
% uPIT - Compute utterance-level permutation invariant training
v1 = SISNR(y1,x1);
v2 = SISNR(y2,x2);
m1 = mean([v1;v2]);

v1 = SISNR(y2,x1);
v2 = SISNR(y1,x2);
m2 = mean([v1;v2]);

m = max(m1,m2);
end

function z = SISNR(x,y)
% SISNR - Compute SI-SNR
x = x - mean(x);
y = y - mean(y);

t = sum(x.*y) .* y ./ (sum(y.^2)+eps);
n = x - t;

z = 20*log((sqrt(sum(t.^2))+eps)./sqrt((sum(n.^2))+eps))/log(10);

end

function [learnables,states] = initializeNetworkParams
% initializeNetworkParams - Initialize the learnables and states of the
% network
learnables.Conv1W = initializeGlorot(20,1,256);
learnables.Conv1B = dlarray(zeros(256,1,'single'));

learnables.ln_weight = dlarray(ones(1,256,'single'));
learnables.ln_bias = dlarray(zeros(1,256,'single'));

learnables.Conv2W = initializeGlorot(1,256,256);
learnables.Conv2B = dlarray(zeros(256,1,'single'));

for index=1:32
    blk = [];
    blk.Conv1W = initializeGlorot(1,256,512);
    blk.Conv1B = dlarray(zeros(512,1,'single'));
    blk.Prelu1 = dlarray(single(0.25));
    blk.BN1Offset = dlarray(zeros(512,1,'single'));
```

```
    blk.BN1Scale = dlarray(ones(512,1,'single'));
    blk.Conv2W = initializeGlorot(3,1,512);
    blk.Conv2W =  reshape(blk.Conv2W,[3 1 1 512]);
    blk.Conv2B = dlarray(zeros(512,1,'single'));
    blk.Prelu2 = dlarray(single(0.25));
    blk.BN2Offset= dlarray(zeros(512,1,'single'));
    blk.BN2Scale= dlarray(ones(512,1,'single'));
    blk.Conv3W = initializeGlorot(1,512,256);
    blk.Conv3B = dlarray(ones(256,1,'single'));

    learnables.Blocks(index) = blk;

    s = [];
    s.BN1Mean= dlarray(zeros(512,1,'single'));
    s.BN1Var= dlarray(ones(512,1,'single'));
    s.BN2Mean = dlarray(zeros(512,1,'single'));
    s.BN2Var = dlarray(ones(512,1,'single'));

    states(index) = s; %#ok
end

learnables.Conv3W = initializeGlorot(1,256,512);
learnables.Conv3B = dlarray(zeros(512,1,'single'));

learnables.TransConv1W = initializeGlorot(20,1,256);
learnables.TransConv1B = dlarray(zeros(1,1, 'single'));

end

function weights = initializeGlorot(filterSize,numChannels,numFilters)
% initializeGlorot - Perform Glorot initialization
sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

Z = 2*rand(sz,'single') - 1;
bound = sqrt(6 / (numIn + numOut));

weights = bound * Z;
weights = dlarray(weights);

end

function [output1, output2, states] = separateSpeakers(input, learnables, states, training)
% separateSpeakers - Separate two speaker signals from a mixture input
if ~isa(input,'dlarray')
    input = dlarray(input,'SCB');
end

weights = learnables.Conv1W;
bias = learnables.Conv1B;
x = dlconv(input, weights,bias, 'Stride', 10);

x = relu(x);
x0 = x;

x = x-mean(x, 2);
x = x./sqrt(mean(x.^2, 2) + 1e-5);
```

```
x = x.*learnables.ln_weight + learnables.ln_bias;

weights = learnables.Conv2W;
bias = learnables.Conv2B;
encoderOut = dlconv(x, weights, bias);

for index = 1:32
    [encoderOut,s] = convBlock(encoderOut, index-1,learnables.Blocks(index),states(index),trainin
    states(index) = s;
end

weights = learnables.Conv3W;
bias = learnables.Conv3B;
masks = dlconv(encoderOut, weights, bias);
masks = relu(masks);

mask1 = masks(:,1:256,:);
mask2 = masks(:,257:512,:);

out1 = x0 .* mask1;
out2 = x0 .* mask2;

weights = learnables.TransConv1W;
bias = learnables.TransConv1B;
output2 = dltranspconv(out1, weights, bias, 'Stride', 10);
output1 = dltranspconv(out2, weights, bias, 'Stride', 10);

if ~training
    output1 = gather(extractdata(output1));
    output2 = gather(extractdata(output2));

    output1 = output1./max(abs(output1));
    output2 = output2./max(abs(output2));
end

end

function [output,state] = convBlock(input, count,learnables,state,training)

% Conv:
weights = learnables.Conv1W;
bias = learnables.Conv1B;
conv1Out = dlconv(input, weights, bias);

% PRelu:
conv1Out = relu(conv1Out) - learnables.Prelu1.*relu(-conv1Out);

% BatchNormalization:
offset = learnables.BN1Offset;
scale = learnables.BN1Scale;
datasetMean = state.BN1Mean;
datasetVariance = state.BN1Var;
if training
    [batchOut, dsmean, dsvar] = batchnorm(conv1Out, offset, scale, datasetMean, datasetVariance)
    state.BN1Mean = dsmean;
    state.BN1Var = dsvar;
else
    batchOut = batchnorm(conv1Out, offset, scale, datasetMean, datasetVariance);
```

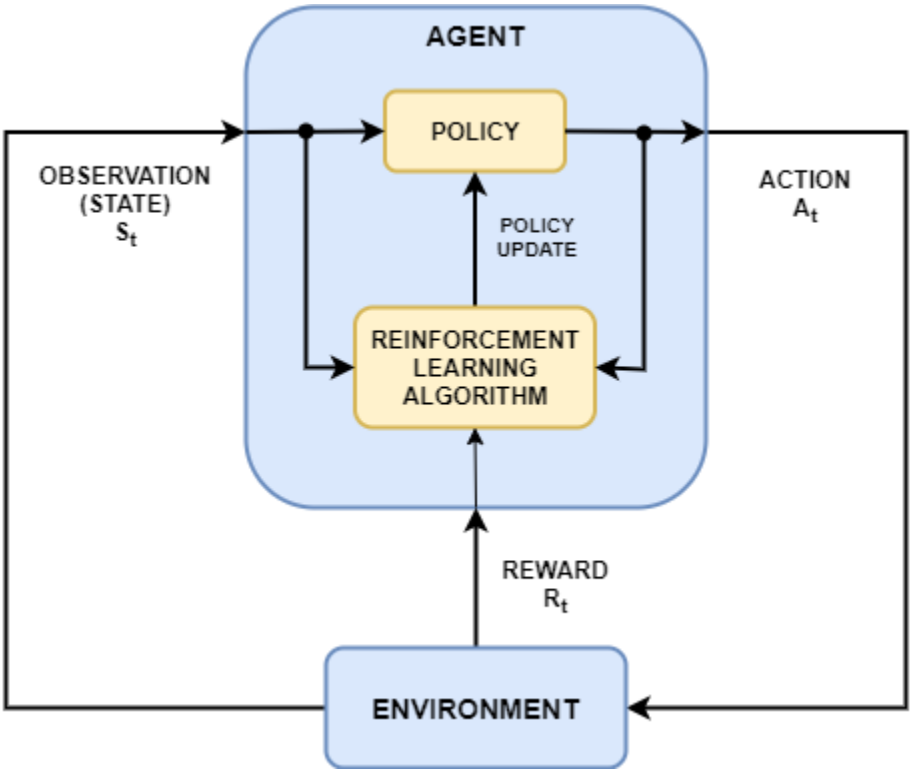
```
end

% Conv:
weights = learnables.Conv2W;
bias = learnables.Conv2B;
padding = [1 1] * 2^(mod(count,8));
dilationFactor = 2^(mod(count,8));
convOut = dlconv(batchOut, weights, bias,'DilationFactor', dilationFactor, 'Padding', padding);

% PRelu:
convOut = relu(convOut) - learnables.Prelu2.*relu(-convOut);

% BatchNormalization:
offset = learnables.BN2Offset;
scale = learnables.BN2Scale;
datasetMean = state.BN2Mean;
datasetVariance = state.BN2Var;
if training
    [batchOut, dsmean, dsvar] = batchnorm(convOut, offset, scale, datasetMean, datasetVariance);
    state.BN2Mean = dsmean;
    state.BN2Var = dsvar;
else
    batchOut = batchnorm(convOut, offset, scale, datasetMean, datasetVariance);
end

% Conv:
weights = learnables.Conv3W;
bias = learnables.Conv3B;
output = dlconv(batchOut, weights, bias);

% Skip connection
output = output + input;

end

function [speaker1,speaker2] = separateSpeakersTimeFrequency(mix,pathToNet)
% separateSpeakersTimeFrequency - STFT-based speaker separation function
WindowLength  = 128;
FFTLength     = 128;
OverlapLength = 128-1;
win           = hann(WindowLength,"periodic");

% Downsample to 4 kHz
mix = resample(mix,1,2);

P0 = stft(mix, 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLength', FFTLength, 'FrequencyRange', 'onesided');
P = log(abs(P0) + eps);
MP = mean(P(:));
SP = std(P(:));
P = (P-MP)/SP;

seqLen = 20;
PSeq  = zeros(1 + FFTLength/2,seqLen,1,0);
seqOverlap = seqLen;

loc = 1;
while loc < size(P,2)-seqLen
```

```
    PSeq(:,:,:,end+1) = P(:,loc:loc+seqLen-1); %#ok
    loc = loc + seqOverlap;
end

PSeq  = reshape(PSeq, [1 1 (1 + FFTLength/2) * seqLen size(PSeq,4)]);

s = load(fullfile(pathToNet,"CocktailPartyNet.mat"));
CocktailPartyNet = s.CocktailPartyNet;
estimatedMasks = predict(CocktailPartyNet,PSeq);

estimatedMasks = estimatedMasks.';
estimatedMasks = reshape(estimatedMasks,1 + FFTLength/2,numel(estimatedMasks)/(1 + FFTLength/2));

mask1   = estimatedMasks;
mask2 = 1 - mask1;

P0 = P0(:,1:size(mask1,2));

P_speaker1 = P0 .* mask1;

speaker1 = istft(P_speaker1, 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLength', FFTLength, 'ConjugateSymmetric', true,...
    'FrequencyRange', 'onesided');
speaker1 = speaker1 / max(abs(speaker1));

P_speaker2 = P0 .* mask2;

speaker2 = istft(P_speaker2, 'Window', win, 'OverlapLength', OverlapLength,...
    'FFTLength',FFTLength, 'ConjugateSymmetric',true,...
    'FrequencyRange', 'onesided');
speaker2 = speaker2 / max(speaker2);

speaker1 = resample(double(speaker1),2,1);
speaker2 = resample(double(speaker2),2,1);
end
```

Reinforcement Learning Examples

Reinforcement Learning Using Deep Neural Networks

Reinforcement learning is a goal-directed computational approach where a computer learns to perform a task by interacting with an unknown dynamic environment. This learning approach enables the computer to make a series of decisions to maximize the cumulative reward for the task without human intervention and without being explicitly programmed to achieve the task. The following diagram shows a general representation of a reinforcement learning scenario.



The goal of reinforcement learning is to train the *policy* of an *agent* to complete a task within an unknown *environment*. The agent receives *observations* and a *reward* from the environment and sends *actions* to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

To create and train reinforcement learning agents, you can use Reinforcement Learning Toolbox™ software. Typically, agent policies are implemented using deep neural networks, which you can create using Deep Learning Toolbox software.

Reinforcement learning is useful for many control and planning applications. The following examples show how to train reinforcement learning agents for robotics and automated driving tasks.

- “Train DDPG Agent to Control Flying Robot” on page 15-45
- “Train Biped Robot to Walk Using Reinforcement Learning Agents” on page 15-51
- “Train DDPG Agent for Adaptive Cruise Control” on page 15-70
- “Train DDPG Agent for Path-Following Control” on page 15-87
- “Train PPO Agent for Automatic Parking Valet” on page 15-95

Reinforcement Learning Workflow

The general workflow for training an agent using reinforcement learning includes the following steps.

1. **Formulate problem** — Define the task for the agent to learn, including how the agent interacts with the environment and any primary and secondary goals the agent must achieve.
2. **Create environment** — Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model.
3. **Define reward** — Specify the reward signal that the agent uses to measure its performance against the task goals and how to calculate this signal from the environment.
4. **Create agent** — Create the agent, which includes defining a policy representation and configuring the agent learning algorithm.
5. **Train agent** — Train the agent policy representation using the defined environment, reward, and agent learning algorithm.
6. **Validate agent** — Evaluate the performance of the trained agent by simulating the agent and environment together.
7. **Deploy policy** — Deploy the trained policy representation using, for example, generated GPU code.

Training an agent using reinforcement learning is an iterative process. Decisions and results in later stages can require you to return to an earlier stage in the learning workflow. For example, if the training process does not converge to an optimal policy within a reasonable amount of time, you might have to update any of the following before retraining the agent:

- Training settings
- Learning algorithm configuration
- Policy representation
- Reward signal definition
- Action and observation signals
- Environment dynamics

Reinforcement Learning Environments

In a reinforcement learning scenario, where you train an agent to complete a task, the environment models the dynamics with which the agent interacts. The environment:

1. Receives actions from the agent.

2 Outputs observations in response to the actions.

3 Generates a reward measuring how well the action contributes to achieving the task.

Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment.
- Reward signal that the agent uses to measure its success. For more information, see “Define Reward Signals” (Reinforcement Learning Toolbox).
- Environment dynamic behavior.

You can create an environment in either MATLAB or Simulink. For more information, see “Create MATLAB Reinforcement Learning Environments” (Reinforcement Learning Toolbox) and “Create Simulink Reinforcement Learning Environments” (Reinforcement Learning Toolbox).

Reinforcement Learning Agents

A reinforcement learning agent contains two components: a *policy* and a *learning algorithm*.

- The policy is a mapping that selects actions based on observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the learning algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.

Agents are distinguished by their learning algorithms and policy representations. Agents can operate in discrete action spaces, continuous action spaces, or both. In a discrete action space, the agent selects actions from a finite set of possible actions. In a continuous action space, the agent selects an action from a continuous range of possible action values. Reinforcement Learning Toolbox software supports the following types of agents.

Agent	Action Space
“Q-Learning Agents” (Reinforcement Learning Toolbox)	Discrete
“Deep Q-Network Agents” (Reinforcement Learning Toolbox)	Discrete
“SARSA Agents” (Reinforcement Learning Toolbox)	Discrete
“Policy Gradient Agents” (Reinforcement Learning Toolbox)	Discrete or continuous
“Actor-Critic Agents” (Reinforcement Learning Toolbox)	Discrete or continuous
“Proximal Policy Optimization Agents” (Reinforcement Learning Toolbox)	Discrete or continuous
“Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox)	Continuous
“Twin-Delayed Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox)	Continuous

Agent	Action Space
“Soft Actor-Critic Agents” (Reinforcement Learning Toolbox)	Continuous

For more information, see “Reinforcement Learning Agents” (Reinforcement Learning Toolbox).

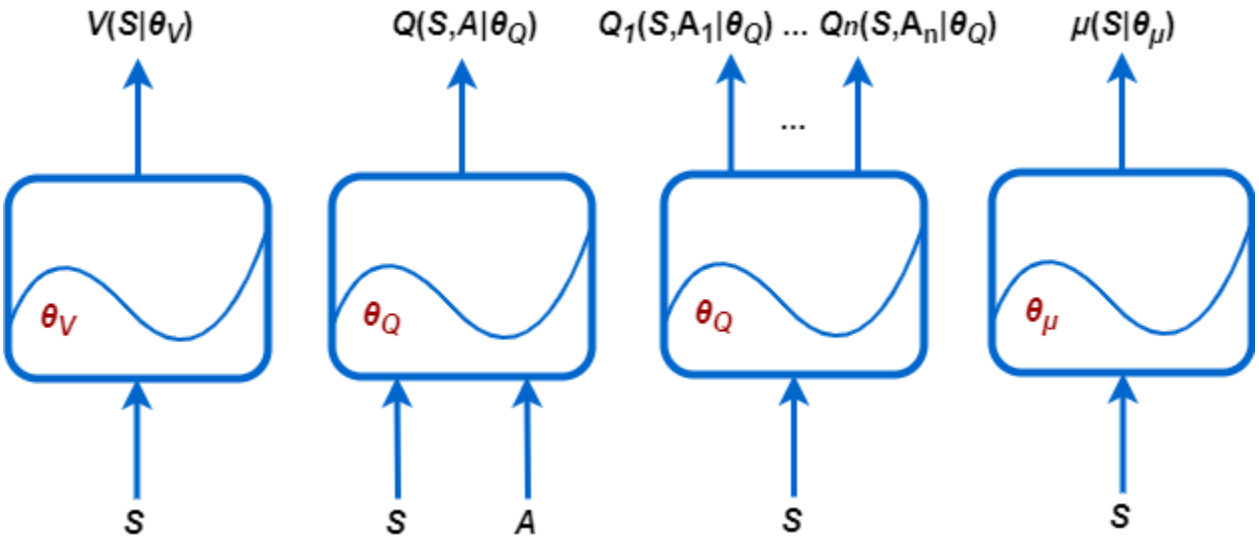
Create Deep Neural Network Policies and Value Functions

Depending on the type of agent you use, its policy and learning algorithm require one or more policy and value function representations, which you can implement using deep neural networks.

Reinforcement Learning Toolbox supports the following types of value function and policy representations.

- $V(S|\theta_V)$ — Critics that estimate the expected cumulative long-term reward (value function) based on a given observation S.
- $Q(S,A|\theta_Q)$ — Critics that estimate the value function for a given discrete action A and a given observation S.
- $Q_i(S,A_i|\theta_Q)$ — Multi-output critics that estimate the value function for all possible discrete actions A_i and a given observation S.
- $\mu(S|\theta_\mu)$ — Actors that select an action based on a given observation S. Actors can select actions using either deterministic or stochastic methods.

During training, the agent updates the parameters of these representations (θ_V, θ_Q, and θ_μ).



You can create most Reinforcement Learning Toolbox agents with default policy and value function representations. The agents define the input and output layers of these deep neural networks based on the action and observation specifications from the environment.

Alternatively, you can create actor and critic representations for your agent using Deep Learning Toolbox functionality, such as the **Deep Network Designer** app. In this case, ensure that the input and output dimensions of the actor and critic representations match the corresponding action and observation specifications of the environment. For an example that creates a critic representation

using **Deep Network Designer**, see “Create Agent Using Deep Network Designer and Train Using Image Observations” on page 15-24.

Deep neural networks consist of a series of interconnected layers. For a full list of available layers, see “List of Deep Learning Layers” on page 1-21.

All agents, except Q-learning and SARSA agents, support recurrent neural networks (RNNs). These networks have an input `sequenceInputLayer` and at least one layer that has hidden state information, such as an `lstmLayer`. These networks can be especially useful when the environment has states that are not in the observation vector.

For more information on creating agents and their associated value function and policy representations, see the corresponding agent pages in the previous table.

Reinforcement Learning Toolbox software provides additional layers that you can use when creating deep neural network representations.

Layer	Description
`scalingLayer`	Applies a linear scale and bias to an input array. This layer is useful for scaling and shifting the outputs of nonlinear layers, such as `tanhLayer` and `sigmoidLayer`.
`quadraticLayer`	Creates a vector of quadratic monomials constructed from the elements of the input array. This layer is useful when you need an output that is some quadratic function of its inputs, such as for an LQR controller.
`softplusLayer`	Implements the softplus activation $Y = \log(1 + e^X)$, which ensures that the output is always positive. This is a smoothed version of the rectified linear unit (ReLU).

For more information on creating policy and value function representations, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

You can also import pretrained deep neural networks or deep neural network layer architectures using the Deep Learning Toolbox network import functionality. For more information, see “Import Policy and Value Function Representations” (Reinforcement Learning Toolbox).

Train Reinforcement Learning Agents

Once you create an environment and reinforcement learning agent, you can train the agent in the environment using the `train` function. To configure your training, use an `rlTrainingOptions` object. For more information, see “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)

If you have Parallel Computing Toolbox software, you can accelerate training and simulation by using multicore processors or GPUs. For more information, see “Train Agents Using Parallel Computing and GPUs” (Reinforcement Learning Toolbox).

Deploy Trained Policies

Once you train a reinforcement learning agent, you can generate code to deploy the optimal policy. You can generate:

- CUDA® code using GPU Coder™
- C/C++ code using MATLAB Coder™

To create a policy evaluation function that selects an action based on a given observation, use the `generatePolicyFunction` command. This command generates a MATLAB script, which contains the policy evaluation function, and a MAT-file, which contains the optimal policy data.

You can generate code to deploy this policy function using GPU Coder or MATLAB Coder.

For more information, see “Deploy Trained Reinforcement Learning Policies” (Reinforcement Learning Toolbox).

See Also

Related Examples

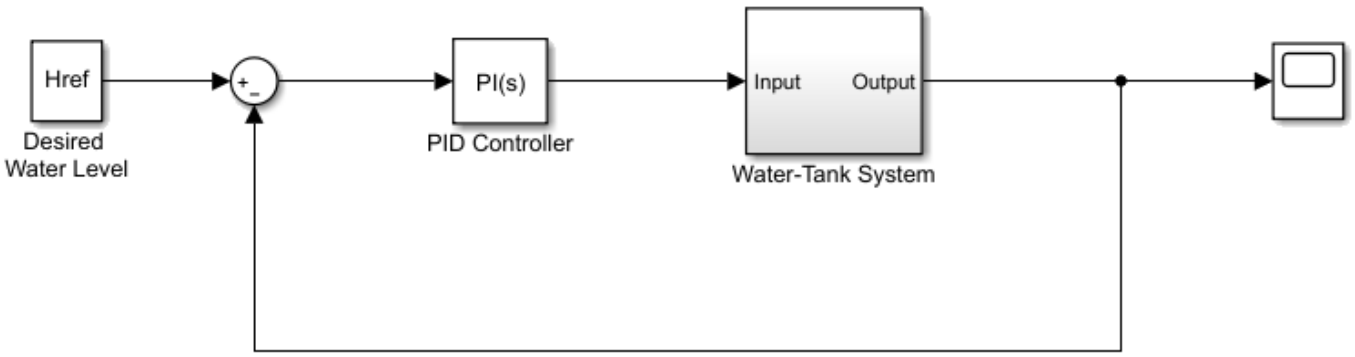
- “What Is Reinforcement Learning?” (Reinforcement Learning Toolbox)
- “Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)
- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)

Create Simulink Environment and Train Agent

This example shows how to convert the PI controller in the `watertank` Simulink® model to a reinforcement learning deep deterministic policy gradient (DDPG) agent. For an example that trains a DDPG agent in MATLAB®, see “Train DDPG Agent to Control Double Integrator System” (Reinforcement Learning Toolbox).

Water Tank Model

The original model for this example is the water tank model. The goal is to control the level of the water in the tank. For more information about the water tank model, see “watertank Simulink Model” (Simulink Control Design).

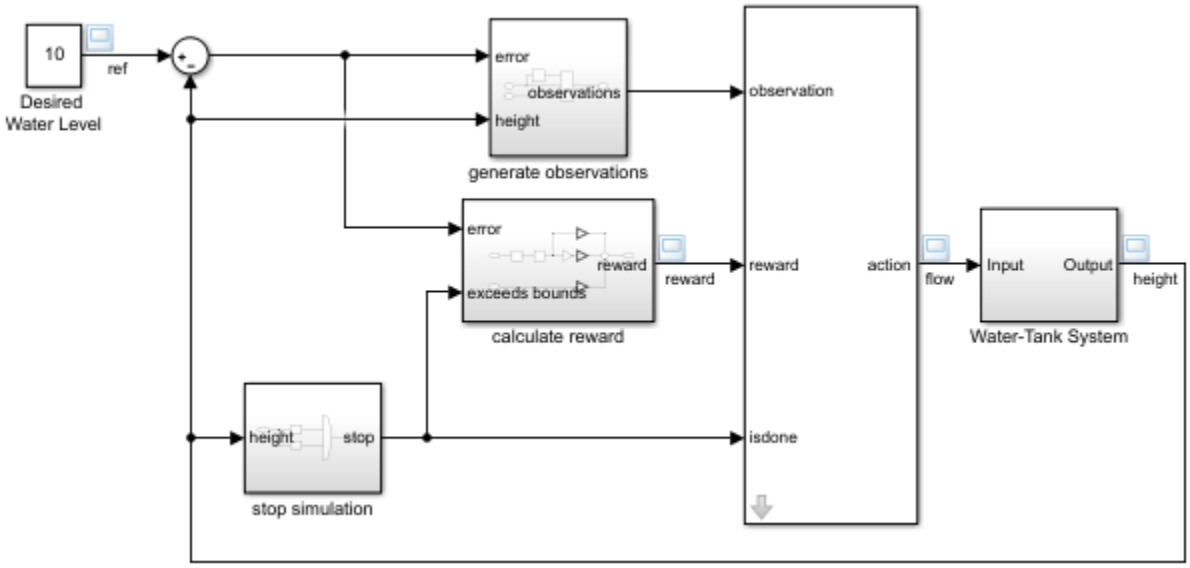


Modify the original model by making the following changes:

1. Delete the PID Controller.
2. Insert the RL Agent block.
3. Connect the observation vector $[\int e\,\mathrm{dt}\ \ e\ \ h]^T$, where h is the height of the tank, $e = r - h$, and r is the reference height.
4. Set up the reward reward $= 10(|e| < 0.1) - 1(|e| \geq 0.1) - 100(h \leq 0 \,||\, h \geq 20)$.
5. Configure the termination signal such that the simulation stops if $h \leq 0$ or $h \geq 20$.

The resulting model is `rlwatertank.slx`. For more information on this model and the changes, see “Create Simulink Reinforcement Learning Environments” (Reinforcement Learning Toolbox).

```
open_system('rlwatertank')
```



Create Environment Interface

Creating an environment model includes defining the following:

- Action and observation signals that the agent uses to interact with the environment. For more information, see `rlNumericSpec` (Reinforcement Learning Toolbox) and `rlFiniteSetSpec` (Reinforcement Learning Toolbox).
- Reward signal that the agent uses to measure its success. For more information, see “Define Reward Signals” (Reinforcement Learning Toolbox).

Define the observation specification `obsInfo` and action specification `actInfo`.

```
obsInfo = rlNumericSpec([3 1],...
    'LowerLimit',[-inf -inf 0  ]',...
    'UpperLimit',[ inf  inf inf]');
obsInfo.Name = 'observations';
obsInfo.Description = 'integrated error, error, and measured height';
numObservations = obsInfo.Dimension(1);

actInfo = rlNumericSpec([1 1]);
actInfo.Name = 'flow';
numActions = actInfo.Dimension(1);
```

Build the environment interface object.

```
env = rlSimulinkEnv('rlwatertank','rlwatertank/RL Agent',...
    obsInfo,actInfo);
```

Set a custom reset function that randomizes the reference values for the model.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Specify the simulation time `Tf` and the agent sample time `Ts` in seconds.

```
Ts = 1.0;
Tf = 200;
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

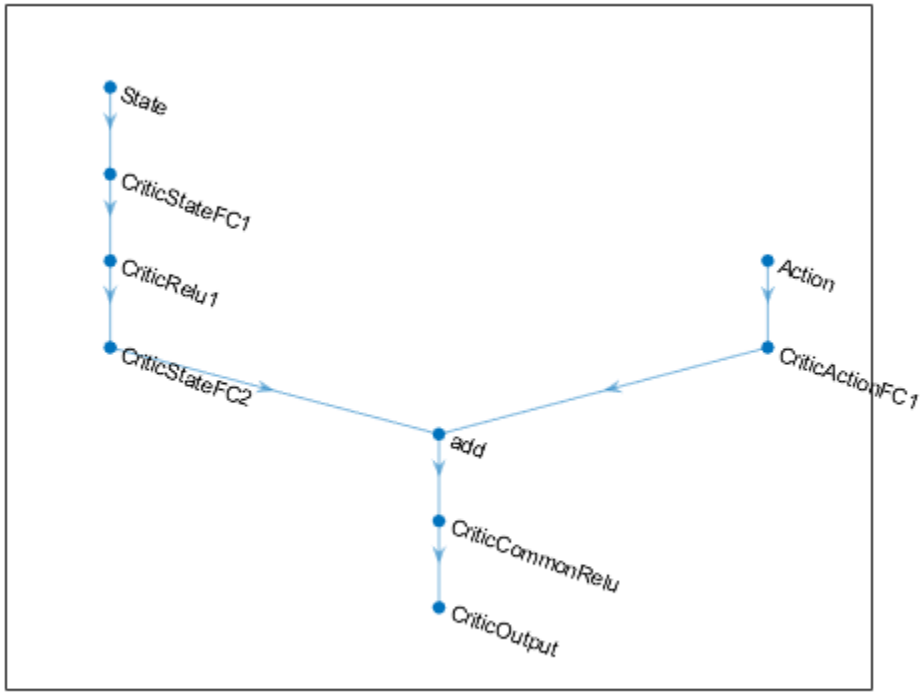
Create DDPG Agent

Given observations and actions, a DDPG agent approximates the long-term reward using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the observation and action, and one output. For more information on creating a deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
statePath = [
    featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(50,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(25,'Name','CriticStateFC2')];
actionPath = [
    featureInputLayer(numActions,'Normalization','none','Name','Action')
    fullyConnectedLayer(25,'Name','CriticActionFC1')];
commonPath = [
    additionLayer(2,'Name','add')
    reluLayer('Name','CriticCommonRelu')
    fullyConnectedLayer(1,'Name','CriticOutput')];

criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,'CriticStateFC2','add/in1');
criticNetwork = connectLayers(criticNetwork,'CriticActionFC1','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation specifications for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,'Observation',{'State'},'Action',{
```

Given observations, a DDPG agent decides which action to take using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor in a similar manner to the critic. For more information, see `rlDeterministicActorRepresentation` (Reinforcement Learning Toolbox).

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(3, 'Name','actorFC')
    tanhLayer('Name','actorTanh')
    fullyConnectedLayer(numActions,'Name','Action')
    ];

actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);
```

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'State'},
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

```
agentOpts = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',1.0, ...
    'MiniBatchSize',64, ...
    'ExperienceBufferLength',1e6);
agentOpts.NoiseOptions.Variance = 0.3;
agentOpts.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent` (Reinforcement Learning Toolbox).

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training for at most `5000` episodes. Specify that each episode lasts for at most `ceil(Tf/Ts)` (that is `200`) time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than `800` over `20` consecutive episodes. At this point, the agent can control the level of water in the tank.

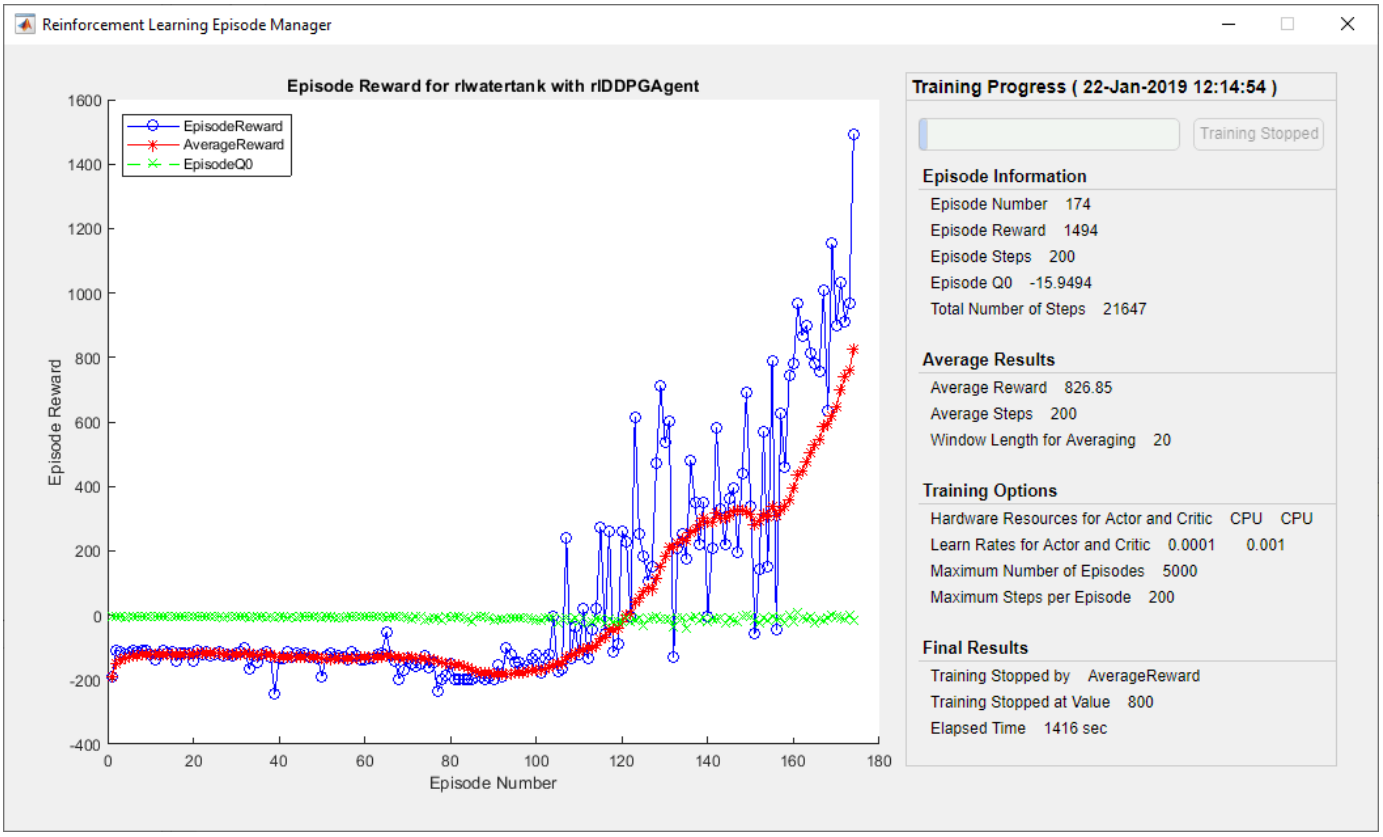
For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'ScoreAveragingWindowLength',20, ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',800);
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
```

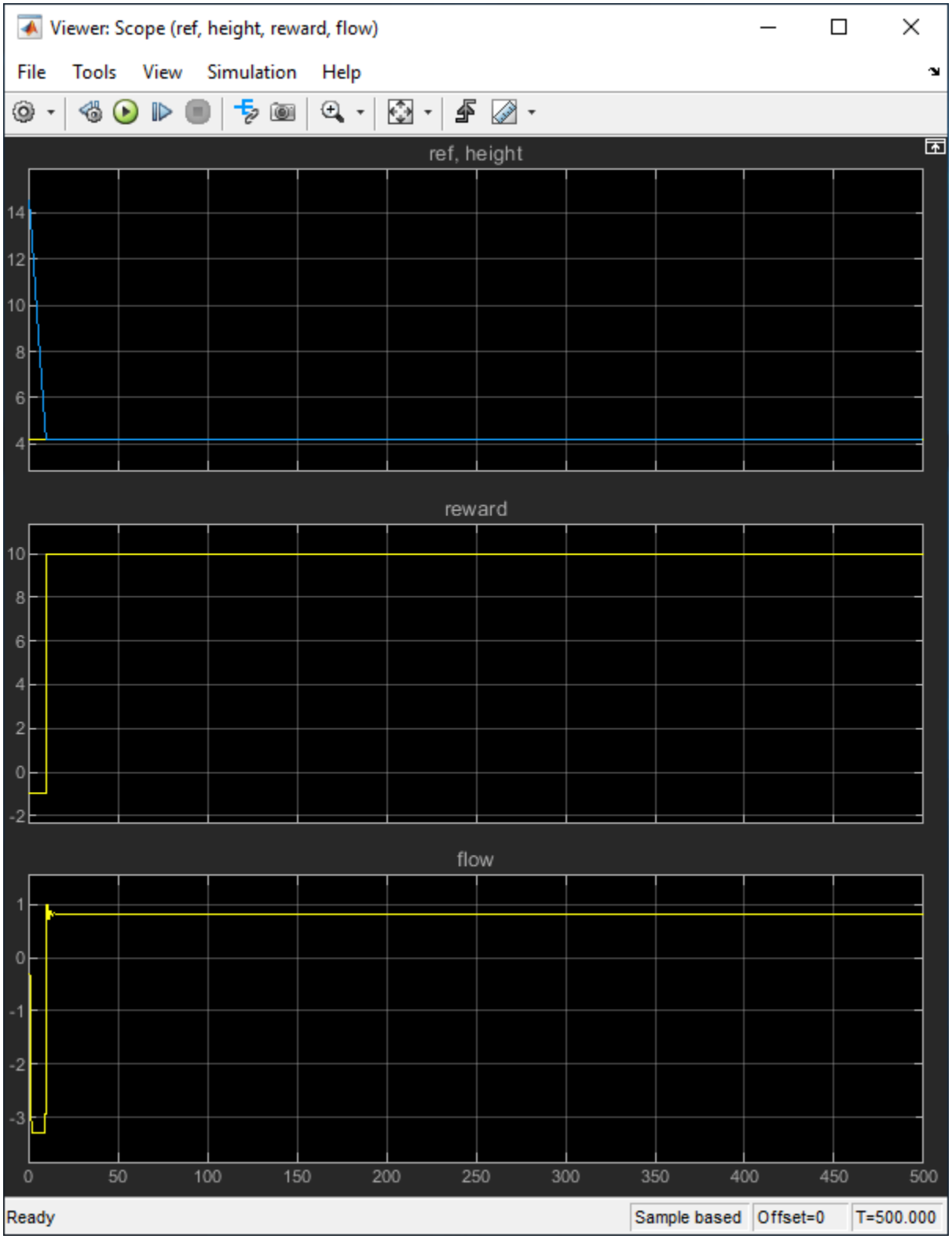
```
    load('WaterTankDDPG.mat','agent')
end
```



Validate Trained Agent

Validate the learned agent against the model by simulation.

```
simOpts = rlSimulationOptions('MaxSteps',maxsteps,'StopOnError','on');
experiences = sim(env,agent,simOpts);
```


Viewer: Scope (ref, height, reward, flow)
File Tools View Simulation Help
ref, height
14
12
10
8
6
4
reward
10
8
6
4
2
0
-2
flow
1
0
-1
-2
-3
0 50 100 150 200 250 300 350 400 450 500
Ready
Sample based
Offset=0
T=500.000

Local Function

```
function in = localResetFcn(in)

% randomize reference signal
blk = sprintf('rlwatertank/Desired \nWater Level');
h = 3*randn + 10;
while h <= 0 || h >= 20
    h = 3*randn + 10;
end
in = setBlockParameter(in,blk,'Value',num2str(h));

% randomize initial height
h = 3*randn + 10;
while h <= 0 || h >= 20
    h = 3*randn + 10;
end
blk = 'rlwatertank/Water-Tank System/H';
in = setBlockParameter(in,blk,'InitialCondition',num2str(h));

end
```

See Also

`train`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Simulink Reinforcement Learning Environments” (Reinforcement Learning Toolbox)

Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation

This example shows how to train a deep deterministic policy gradient (DDPG) agent to swing up and balance a pendulum with an image observation modeled in MATLAB®.

For more information on DDPG agents, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

Simple Pendulum with Image MATLAB Environment

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.

For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are an image indicating the location of the pendulum mass and the pendulum angular velocity.
- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

For more information on this model, see “Load Predefined Control System Environments” (Reinforcement Learning Toolbox).

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumWithImage-Continuous')
```

```
env =
  SimplePendlumWithImageContinuousAction with properties:

            Mass: 1
       RodLength: 1
      RodInertia: 0
         Gravity: 9.8100
    DampingRatio: 0
   MaximumTorque: 2
              Ts: 0.0500
           State: [2x1 double]
```

```
    Q: [2x2 double]
    R: 1.0000e-03
```

The interface has a continuous action space where the agent can apply a torque between –2 to 2 N·m.

Obtain the observation and action specification from the environment interface.

```
obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

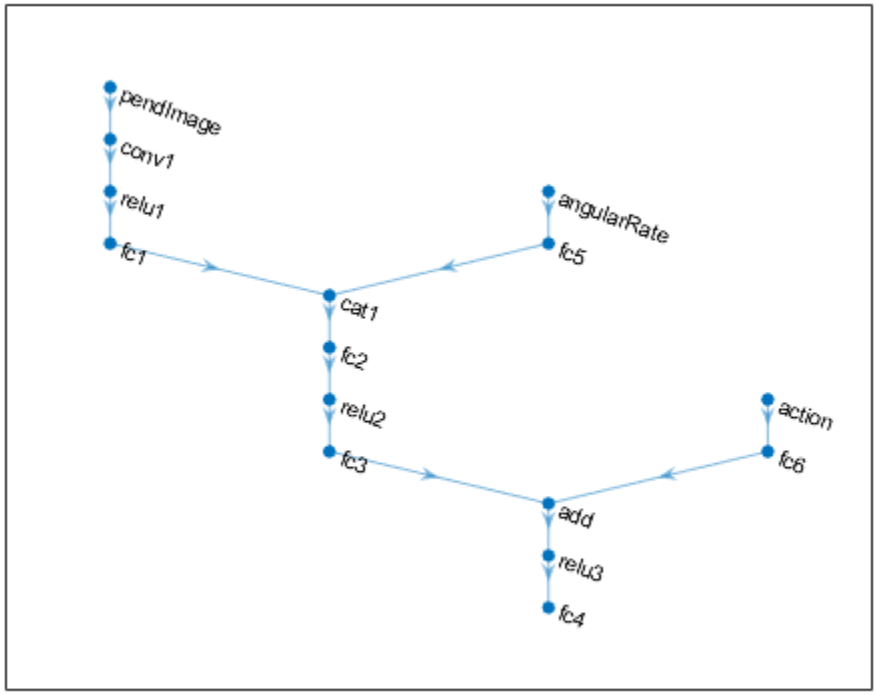
Create DDPG Agent

A DDPG agent approximates the long-term reward, given observations and actions, using a critic value function representation. To create the critic, first create a deep convolutional neural network (CNN) with three inputs (the image, angular velocity, and action) and one output. For more information on creating representations, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
hiddenLayerSize1 = 400;
hiddenLayerSize2 = 300;

imgPath = [
    imageInputLayer(obsInfo(1).Dimension,'Normalization','none','Name',obsInfo(1).Name)
    convolution2dLayer(10,2,'Name','conv1','Stride',5,'Padding',0)
    reluLayer('Name','relu1')
    fullyConnectedLayer(2,'Name','fc1')
    concatenationLayer(3,2,'Name','cat1')
    fullyConnectedLayer(hiddenLayerSize1,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc3')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')
    ];
dthetaPath = [
    imageInputLayer(obsInfo(2).Dimension,'Normalization','none','Name',obsInfo(2).Name)
    fullyConnectedLayer(1,'Name','fc5','BiasLearnRateFactor',0,'Bias',0)
    ];
actPath =[
    imageInputLayer(actInfo(1).Dimension,'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc6','BiasLearnRateFactor',0,'Bias',zeros(hidde
    ];

criticNetwork = layerGraph(imgPath);
criticNetwork = addLayers(criticNetwork,dthetaPath);
criticNetwork = addLayers(criticNetwork,actPath);
criticNetwork = connectLayers(criticNetwork,'fc5','cat1/in2');
criticNetwork = connectLayers(criticNetwork,'fc6','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Uncomment the following line to use the GPU to accelerate training of the critic CNN. For more information on supported GPUs, see “GPU Support by Release” (Parallel Computing Toolbox).

```
% criticOptions.UseDevice = 'gpu';
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'pendImage','angularRate'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations using an actor representation. To create the actor, first create a deep convolutional neural network (CNN) with two inputs (the image and angular velocity) and one output (the action).

Construct the actor in a similar manner to the critic.

```
imgPath = [
    imageInputLayer(obsInfo(1).Dimension,'Normalization','none','Name',obsInfo(1).Name)
    convolution2dLayer(10,2,'Name','conv1','Stride',5,'Padding',0)
    reluLayer('Name','relu1')
```

```
    fullyConnectedLayer(2,'Name','fc1')
    concatenationLayer(3,2,'Name','cat1')
    fullyConnectedLayer(hiddenLayerSize1,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize2,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')
    tanhLayer('Name','tanh1')
    scalingLayer('Name','scale1','Scale',max(actInfo.UpperLimit))
    ];
dthetaPath = [
    imageInputLayer(obsInfo(2).Dimension,'Normalization','none','Name',obsInfo(2).Name)
    fullyConnectedLayer(1,'Name','fc5','BiasLearnRateFactor',0,'Bias',0)
    ];

actorNetwork = layerGraph(imgPath);
actorNetwork = addLayers(actorNetwork,dthetaPath);
actorNetwork = connectLayers(actorNetwork,'fc5','cat1/in2');

actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);
```

Uncomment the following line to use the GPU to accelerate training of the actor CNN.

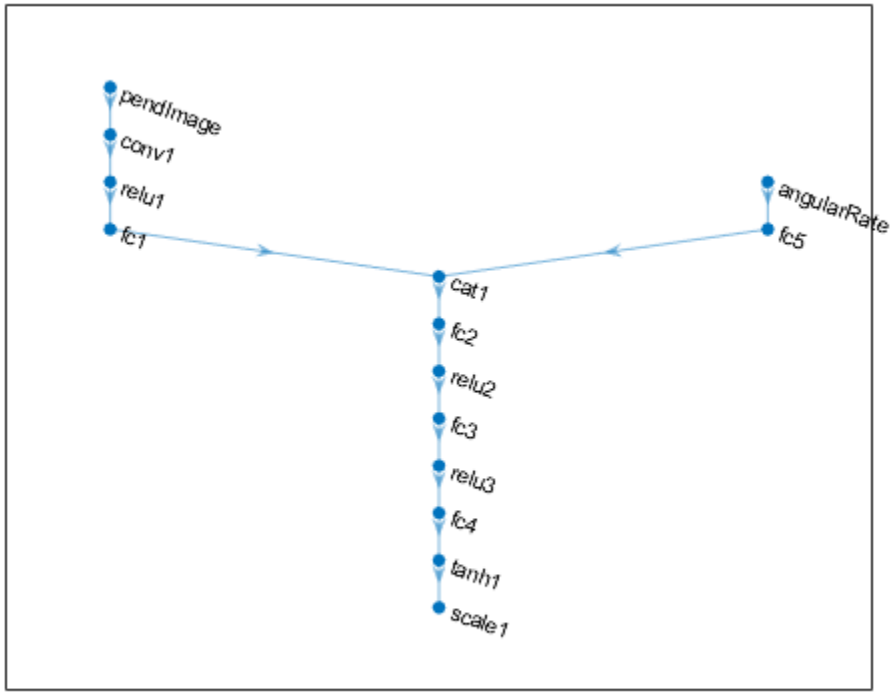
```
% actorOptions.UseDevice = 'gpu';
```

Create the actor representation using the specified neural network and options. For more information, see `rlDeterministicActorRepresentation` (Reinforcement Learning Toolbox).

```
actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo,'Observation',{'pendImage
```

View the actor network configuration.

```
figure
plot(actorNetwork)
```



To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',env.Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',128);
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

Then create the agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent` (Reinforcement Learning Toolbox).

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

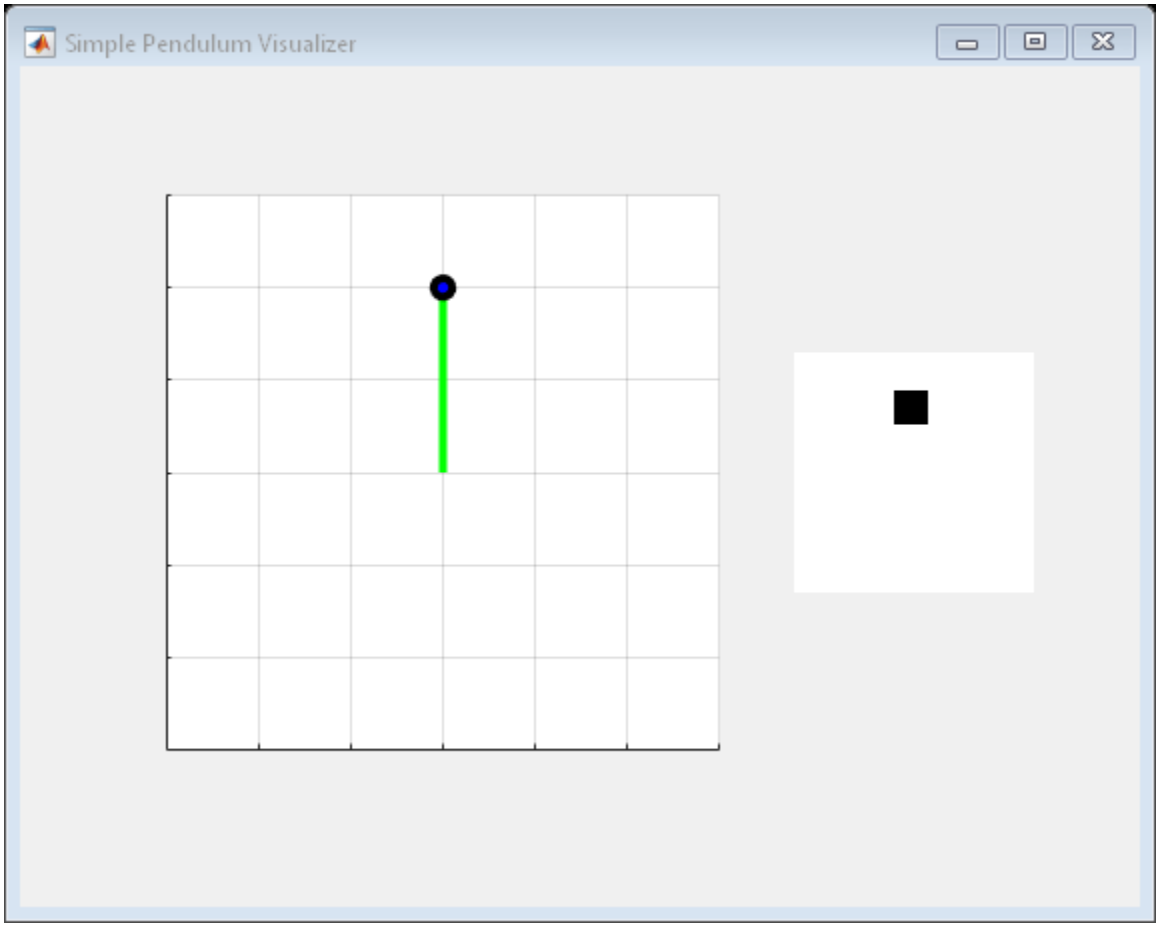
- Run each training for at most 5000 episodes, with each episode lasting at most 400 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option).
- Stop training when the agent receives a moving average cumulative reward greater than -740 over ten consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 5000;
maxsteps = 400;
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-740);
```

You can visualize the pendulum by using the `plot` function during training or simulation.

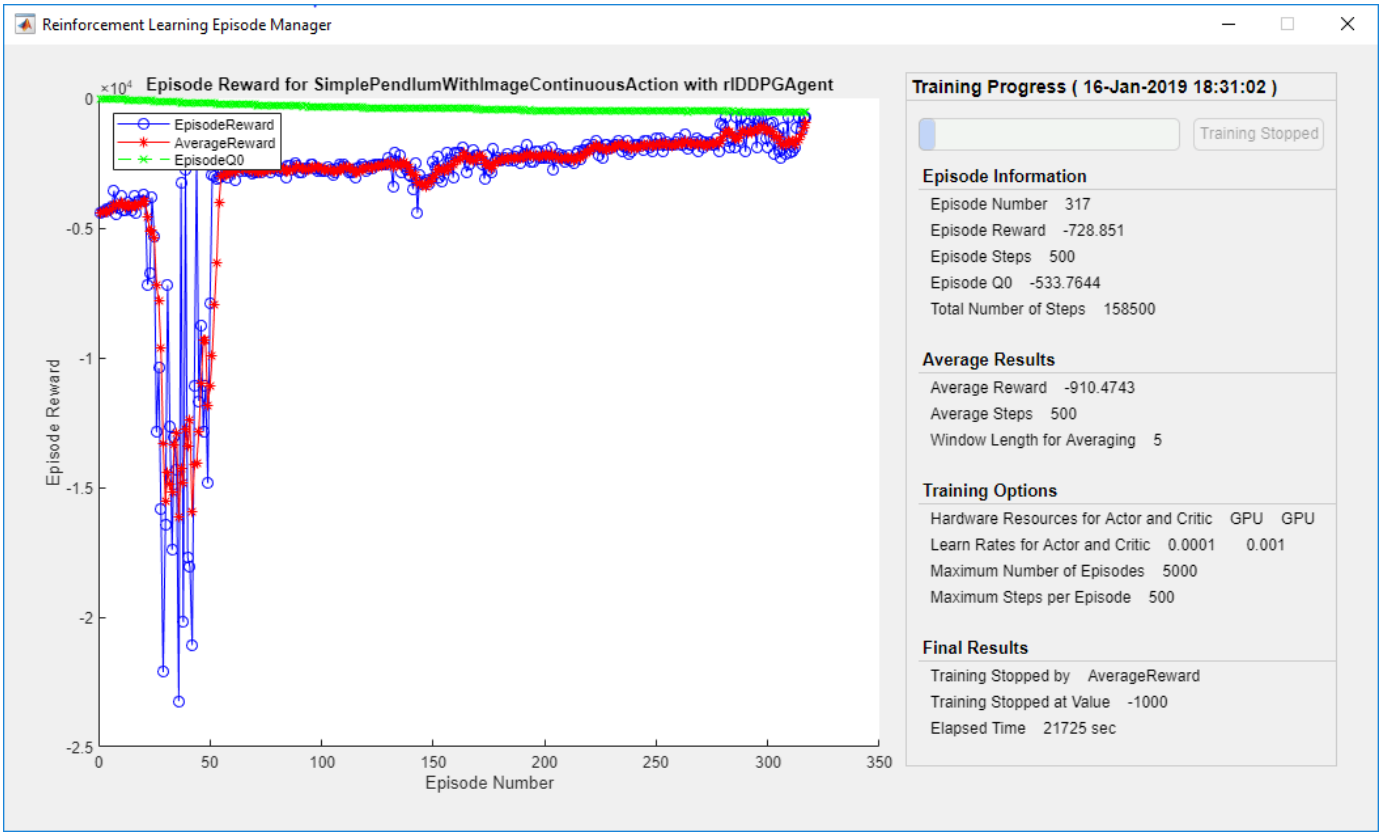
```
plot(env)
```



Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training this agent is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
```

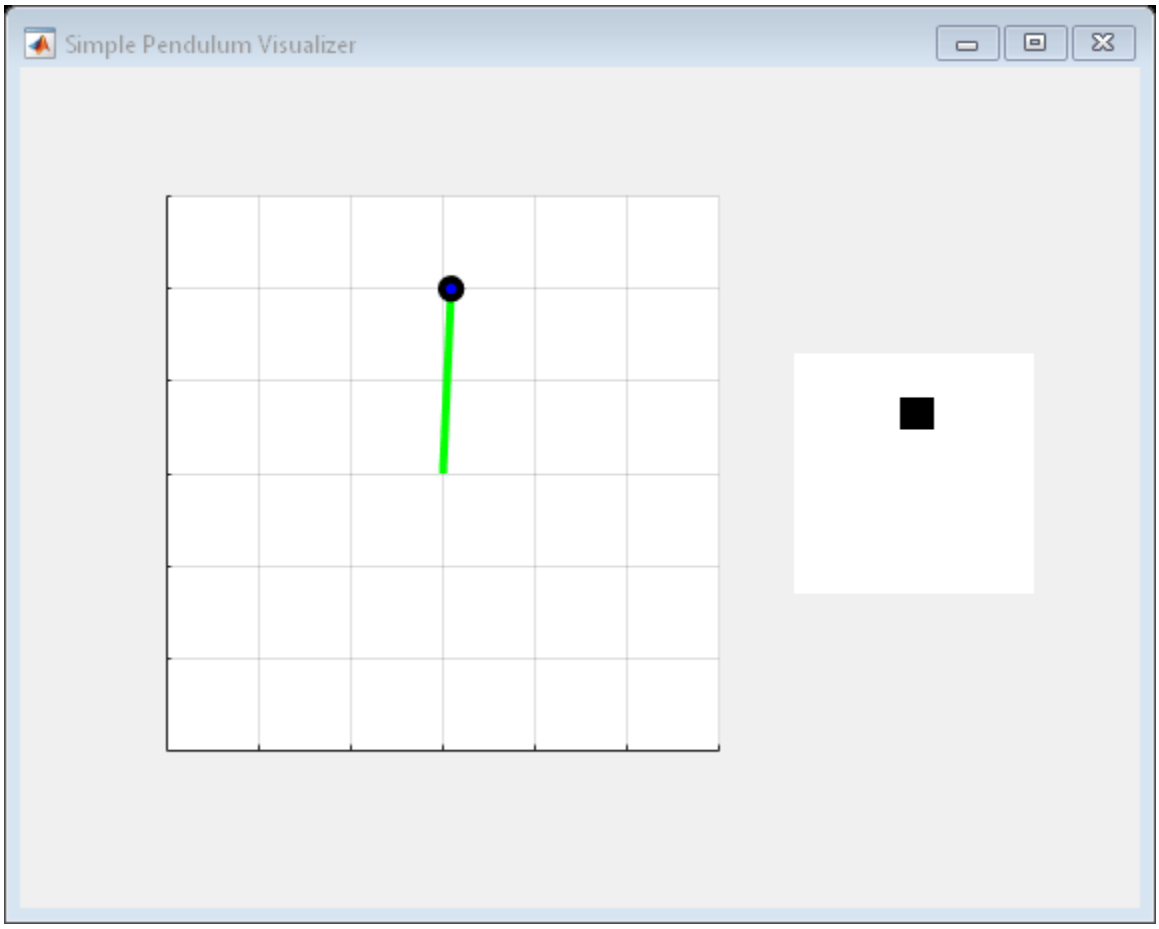
```
    % Load pretrained agent for the example.
    load('SimplePendulumWithImageDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```



See Also

`train`

More About

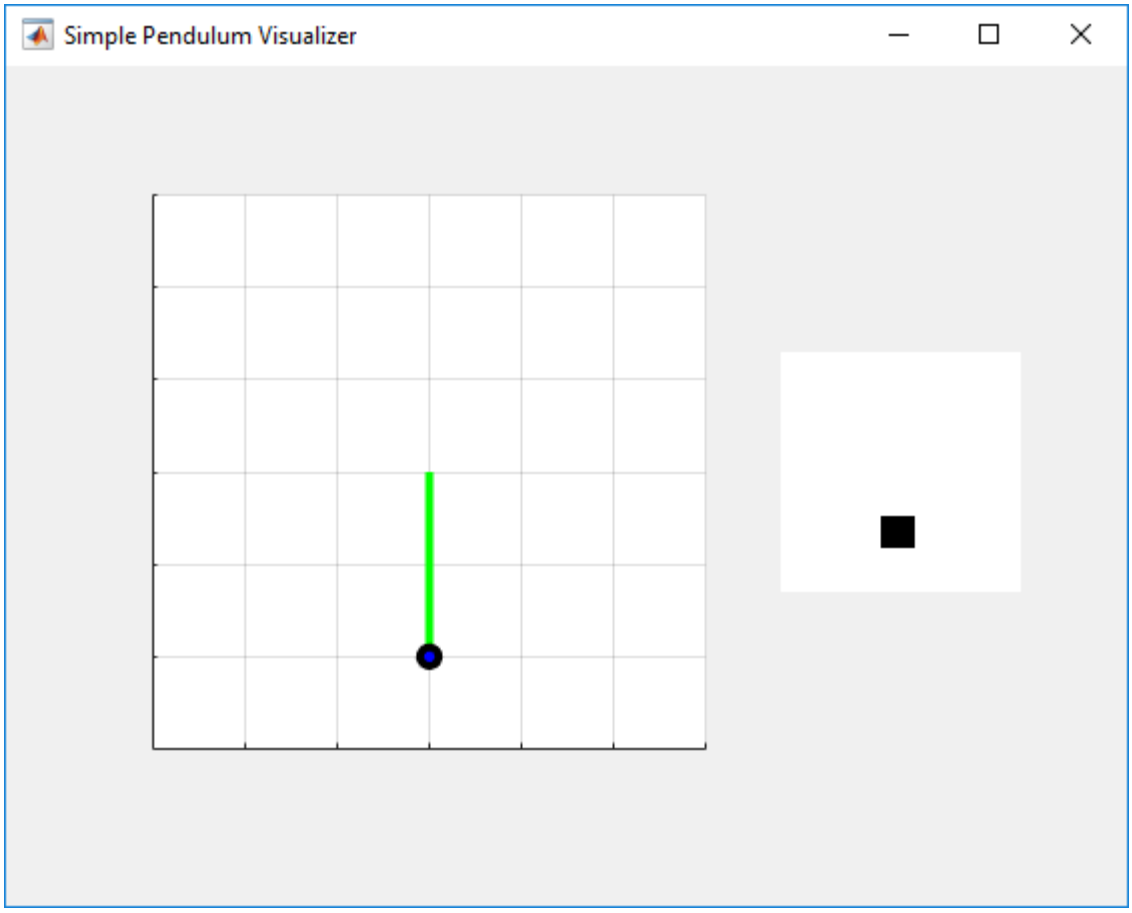
- “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox)
- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Create Agent Using Deep Network Designer and Train Using Image Observations

This example shows how to create a deep Q-learning network (DQN) agent that can swing up and balance a pendulum modeled in MATLAB®. In this example, you create the DQN agent using Deep Network Designer. For more information on DQN agents, see “Deep Q-Network Agents” (Reinforcement Learning Toolbox).

Pendulum Swing-Up with Image MATLAB Environment

The reinforcement learning environment for this example is a simple frictionless pendulum that initially hangs in a downward position. The training goal is to make the pendulum stand upright without falling over using minimal control effort.



For this environment:

- The upward balanced pendulum position is `0` radians, and the downward hanging position is `pi` radians.
- The torque action signal from the agent to the environment is from –2 to 2 N·m.
- The observations from the environment are the simplified grayscale image of the pendulum and the pendulum angle derivative.

- The reward r_t, provided at every time step, is

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

Here:

- θ_t is the angle of displacement from the upright position.
- $\dot{\theta}_t$ is the derivative of the displacement angle.
- u_{t-1} is the control effort from the previous time step.

For more information on this model, see “Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation” (Reinforcement Learning Toolbox).

Create Environment Interface

Create a predefined environment interface for the pendulum.

```
env = rlPredefinedEnv('SimplePendulumWithImage-Discrete');
```

The interface has two observations. The first observation, named `"pendImage"`, is a 50-by-50 grayscale image.

```
obsInfo = getObservationInfo(env);
obsInfo(1)
```

```
ans =
  rlNumericSpec with properties:

     LowerLimit: 0
     UpperLimit: 1
           Name: "pendImage"
    Description: [0x0 string]
      Dimension: [50 50]
       DataType: "double"
```

The second observation, named `"angularRate"`, is the angular velocity of the pendulum.

```
obsInfo(2)
```

```
ans =
  rlNumericSpec with properties:

     LowerLimit: -Inf
     UpperLimit: Inf
           Name: "angularRate"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

The interface has a discrete action space where the agent can apply one of five possible torque values to the pendulum: –2, –1, 0, 1, or 2 N·m.

```
actInfo = getActionInfo(env)
```

```
actInfo =
  rlFiniteSetSpec with properties:

       Elements: [-2 -1 0 1 2]
           Name: "torque"
    Description: [0x0 string]
      Dimension: [1 1]
       DataType: "double"
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

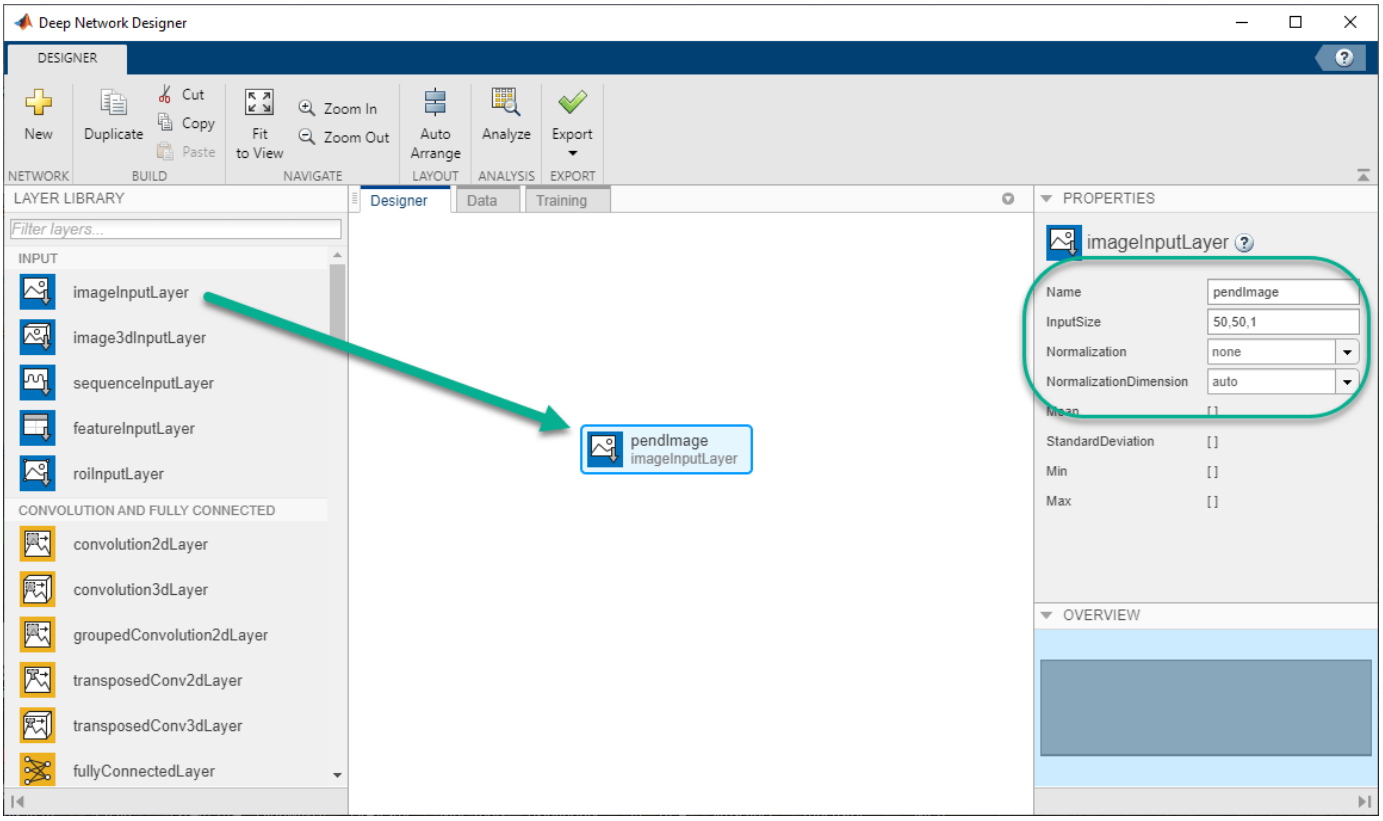
Construct Critic Network Using Deep Network Designer

A DQN agent approximates the long-term reward, given observations and actions, using a critic value function representation. For this environment, the critic is a deep neural network with three inputs (two observations and one action), and one output. For more information on creating a deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

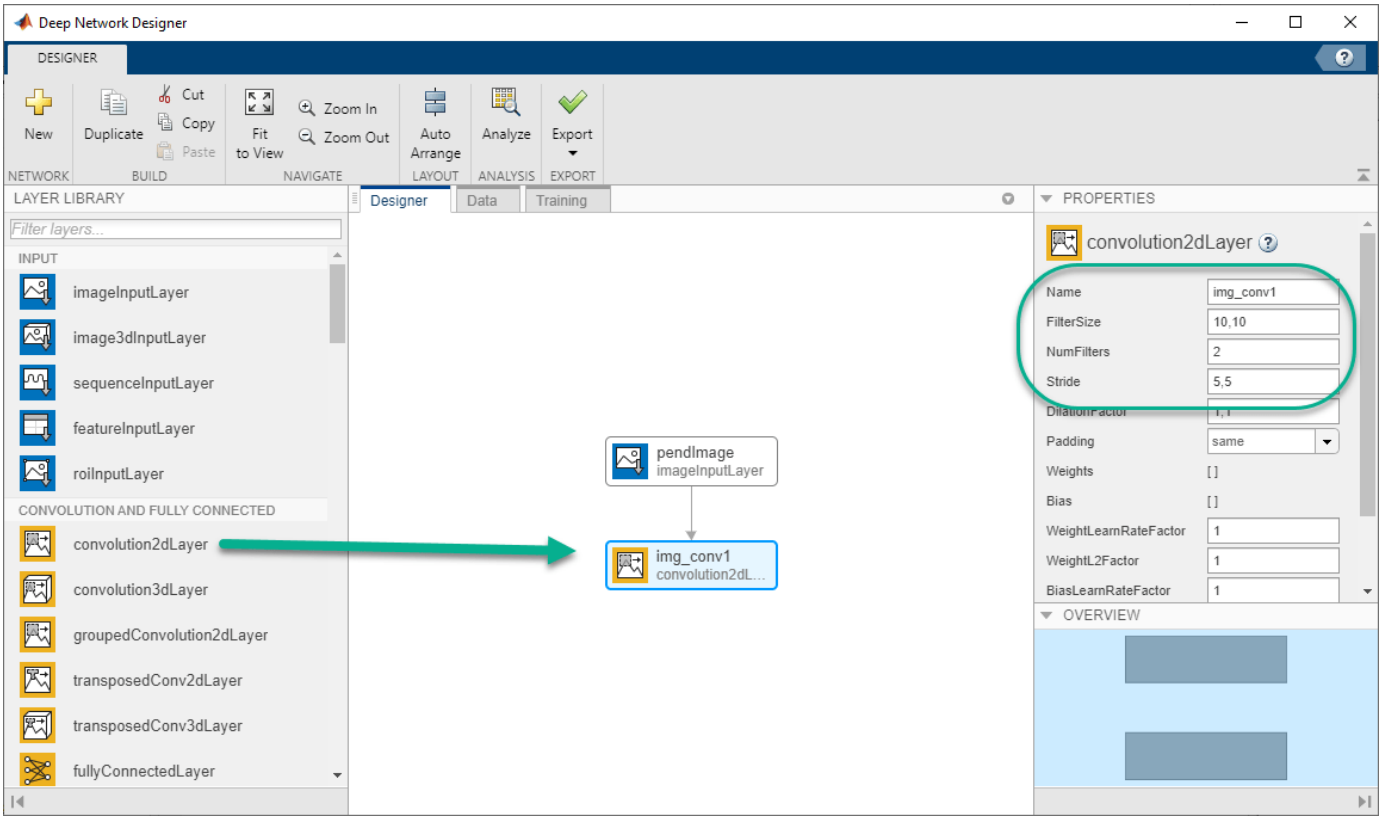
You can construct the critic network interactively by using the Deep Network Designer app. To do so, you first create separate input paths for each observation and action. These paths learn lower-level features from their respective inputs. You then create a common output path that combines the outputs from the input paths.

Create Image Observation Path

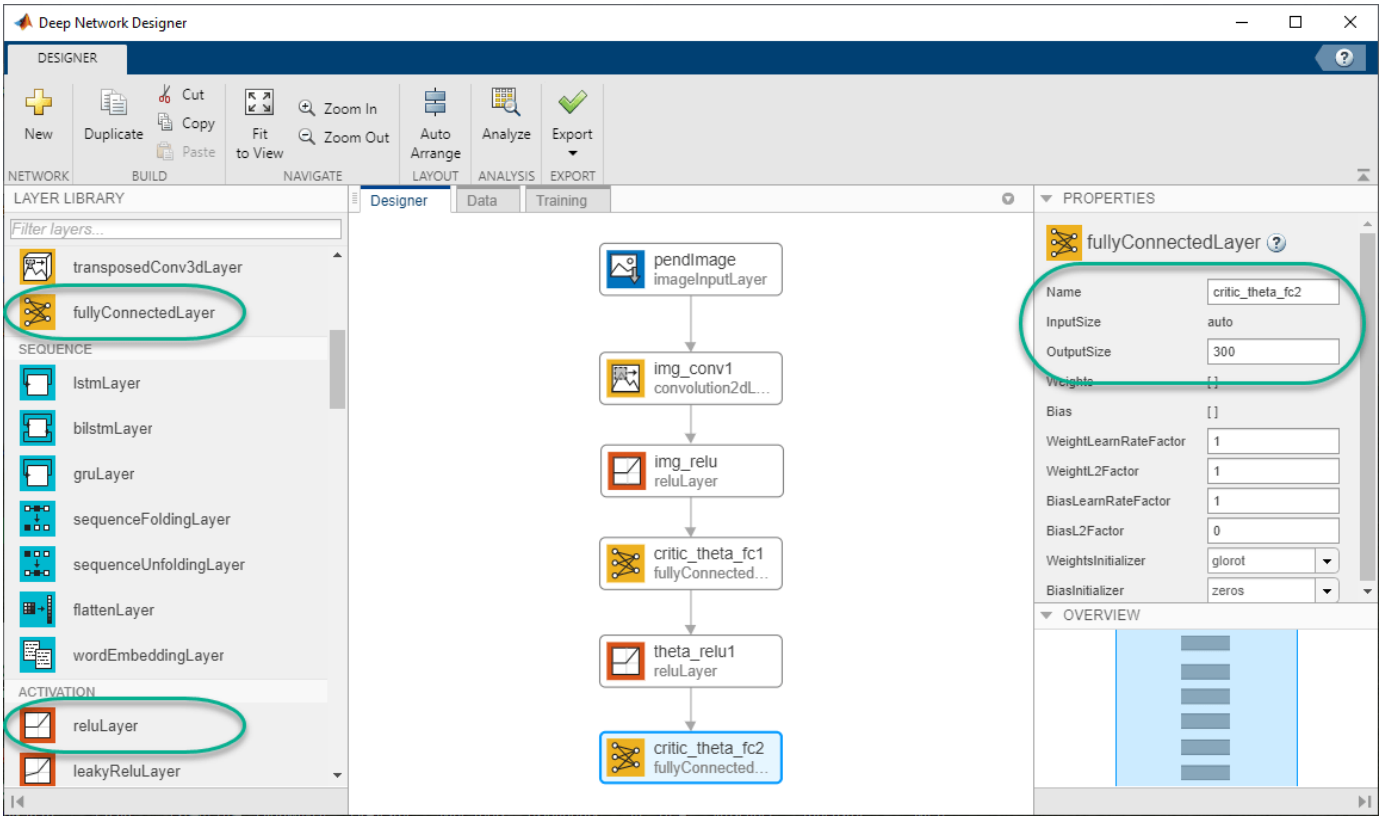
To create the image observation path, first drag an `imageInputLayer` from the **Layer Library** pane to the canvas. Set the layer **InputSize** to `50,50,1` for the image observation, and set **Normalization** to `none`.



Second, drag a `convolution2DLayer` to the canvas and connect the input of this layer to the output of the `imageInputLayer`. Create a convolution layer with `2` filters (**NumFilters** property) that have a height and width of `10` (**FilterSize** property), and use a stride of `5` in the horizontal and vertical directions (**Stride** property).



Finally, complete the image path network with two sets of `reLULayer` and `fullyConnectedLayer` layers. The output sizes of the first and second `fullyConnectedLayer` layers are 400 and 300, respectively.



Create All Input Paths and Output Path

Construct the other input paths and output path in a similar manner. For this example, use the following options.

Angular velocity path (scalar input):

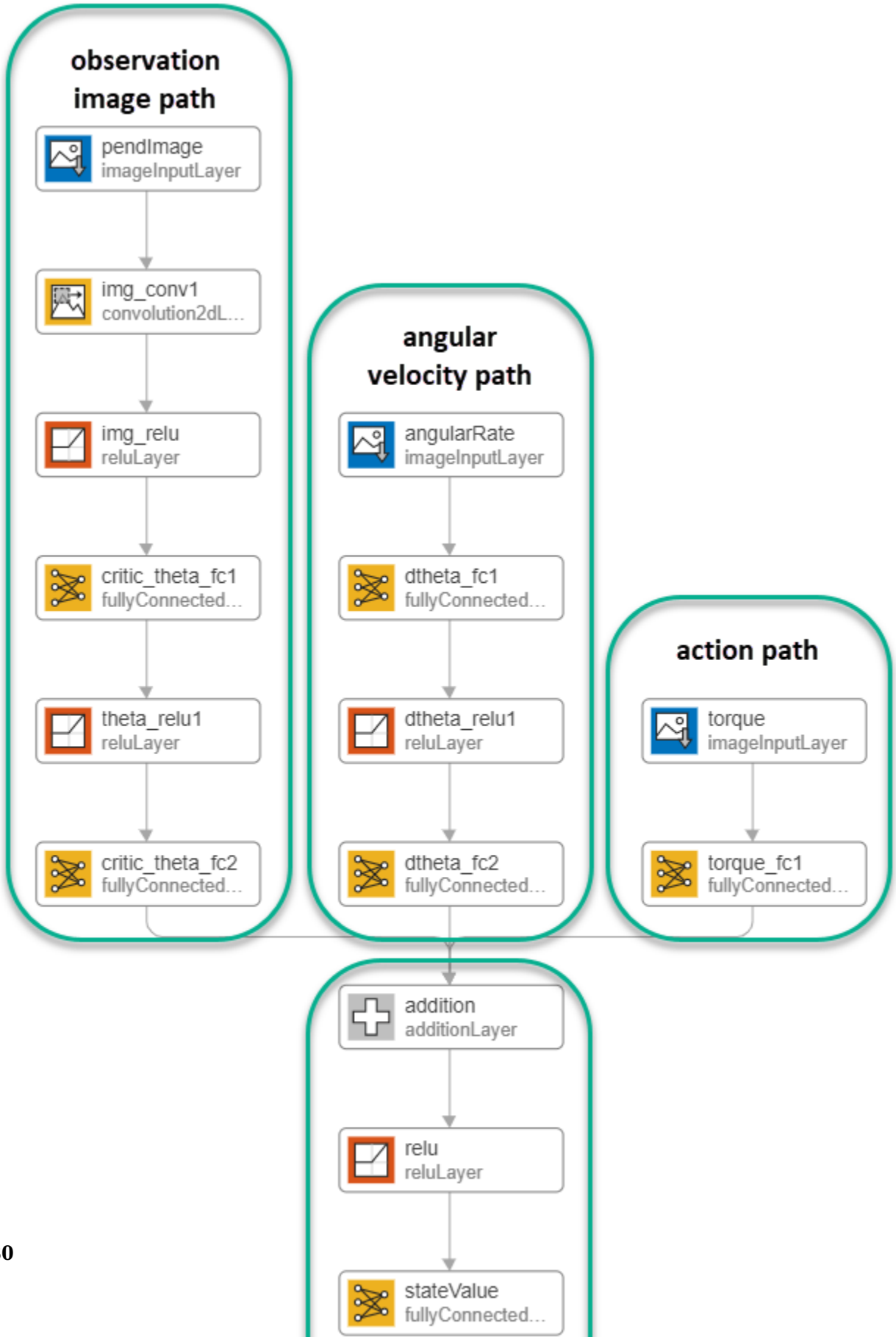
- `imageInputLayer` — Set **InputSize** to `1,1` and **Normalization** to `none`.
- `fullyConnectedLayer` — Set **OutputSize** to `400`.
- `reLULayer`
- `fullyConnectedLayer` — Set **OutputSize** to `300`.

Action path (scalar input):

- `imageInputLayer` — Set **InputSize** to `1,1` and **Normalization** to `none`.
- `fullyConnectedLayer` — Set **OutputSize** to `300`.

Output path:

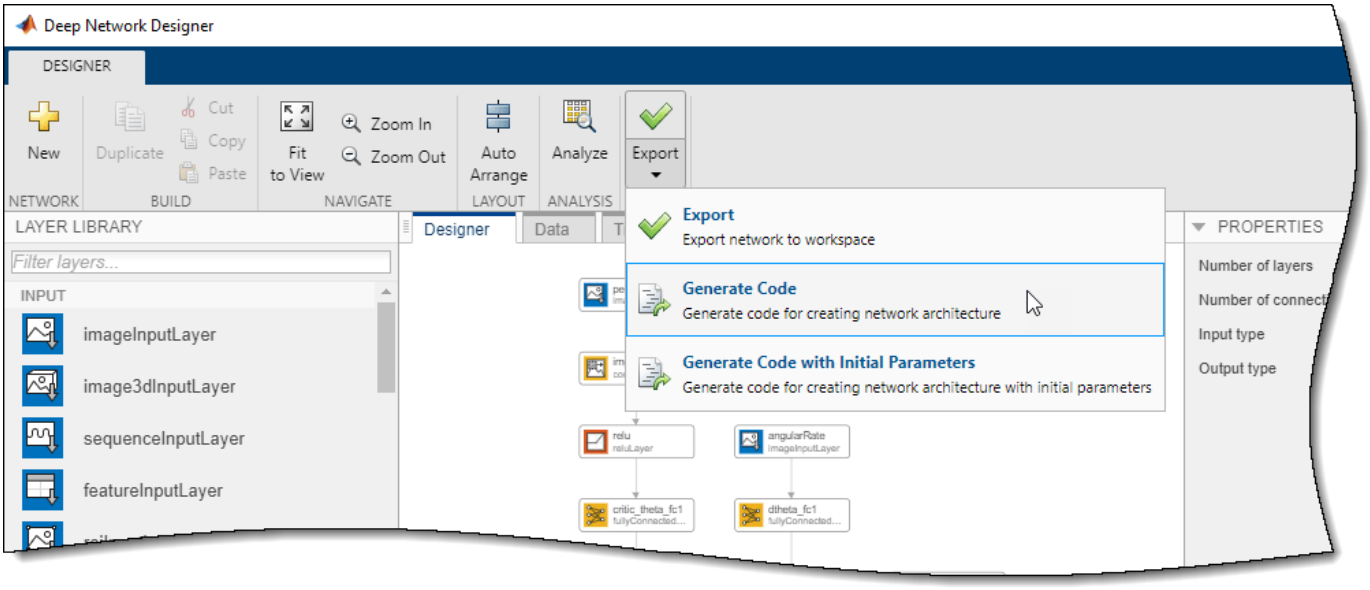
- `additionLayer` — Connect the output of all input paths to the input of this layer.
- `reLULayer`
- `fullyConnectedLayer` — Set **OutputSize** to `1` for the scalar value function.


observation
image path
pendImage
imageInputLayer
img_conv1
convolution2dL...
img_relu
reluLayer
critic_theta_fc1
fullyConnected...
theta_relu1
reluLayer
critic_theta_fc2
fullyConnected...
angular
velocity path
angularRate
imageInputLayer
dtheta_fc1
fullyConnected...
dtheta_relu1
reluLayer
dtheta_fc2
fullyConnected...
action path
torque
imageInputLayer
torque_fc1
fullyConnected...
addition
additionLayer
relu
reluLayer
stateValue
fullyConnected...

Export Network from Deep Network Designer

To export the network to the MATLAB workspace, in **Deep Network Designer**, click **Export**. **Deep Network Designer** exports the network as a new variable containing the network layers. You can create the critic representation using this layer network variable.

Alternatively, to generate equivalent MATLAB code for the network, click **Export > Generate Code**.



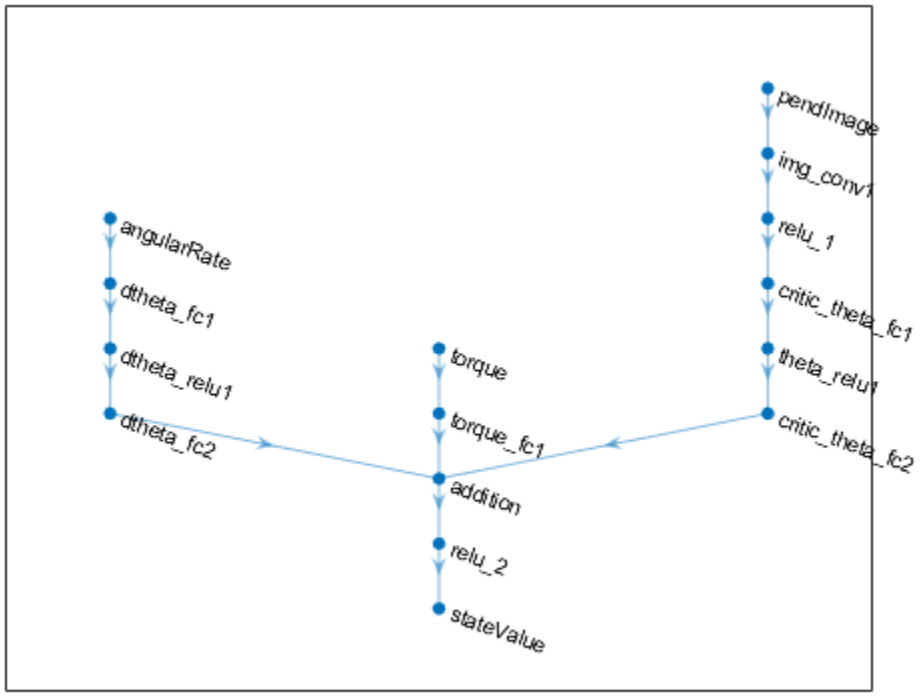
The generated code is as follows.

```
lgraph = layerGraph();

tempLayers = [
    imageInputLayer([1 1 1],"Name","angularRate","Normalization","none")
    fullyConnectedLayer(400,"Name","dtheta_fc1")
    reluLayer("Name","dtheta_relu1")
    fullyConnectedLayer(300,"Name","dtheta_fc2")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    imageInputLayer([1 1 1],"Name","torque","Normalization","none")
    fullyConnectedLayer(300,"Name","torque_fc1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    imageInputLayer([50 50 1],"Name","pendImage","Normalization","none")
    convolution2dLayer([10 10],2,"Name","img_conv1","Padding","same","Stride",[5 5])
    reluLayer("Name","relu_1")
    fullyConnectedLayer(400,"Name","critic_theta_fc1")
    reluLayer("Name","theta_relu1")
    fullyConnectedLayer(300,"Name","critic_theta_fc2")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(3,"Name","addition")
    reluLayer("Name","relu_2")
```

```
    fullyConnectedLayer(1,"Name","stateValue")];
lgraph = addLayers(lgraph,tempLayers);

lgraph = connectLayers(lgraph,"torque_fc1","addition/in3");
lgraph = connectLayers(lgraph,"critic_theta_fc2","addition/in1");
lgraph = connectLayers(lgraph,"dtheta_fc2","addition/in2");
```

View the critic network configuration.

```
figure
plot(lgraph)
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOpts = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified deep neural network `lgraph` and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(lgraph,obsInfo,actInfo,...
    'Observation',{'pendImage','angularRate'},'Action',{'torque'},criticOpts);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions` (Reinforcement Learning Toolbox).

```
agentOpts = rlDQNAgentOptions(...
    'UseDoubleDQN',false,...
    'TargetUpdateMethod',"smoothing",...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'SampleTime',env.Ts,...
    'MiniBatchSize',64);
agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-5;
```

Then, create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent` (Reinforcement Learning Toolbox).

```
agent = rlDQNAgent(critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options.

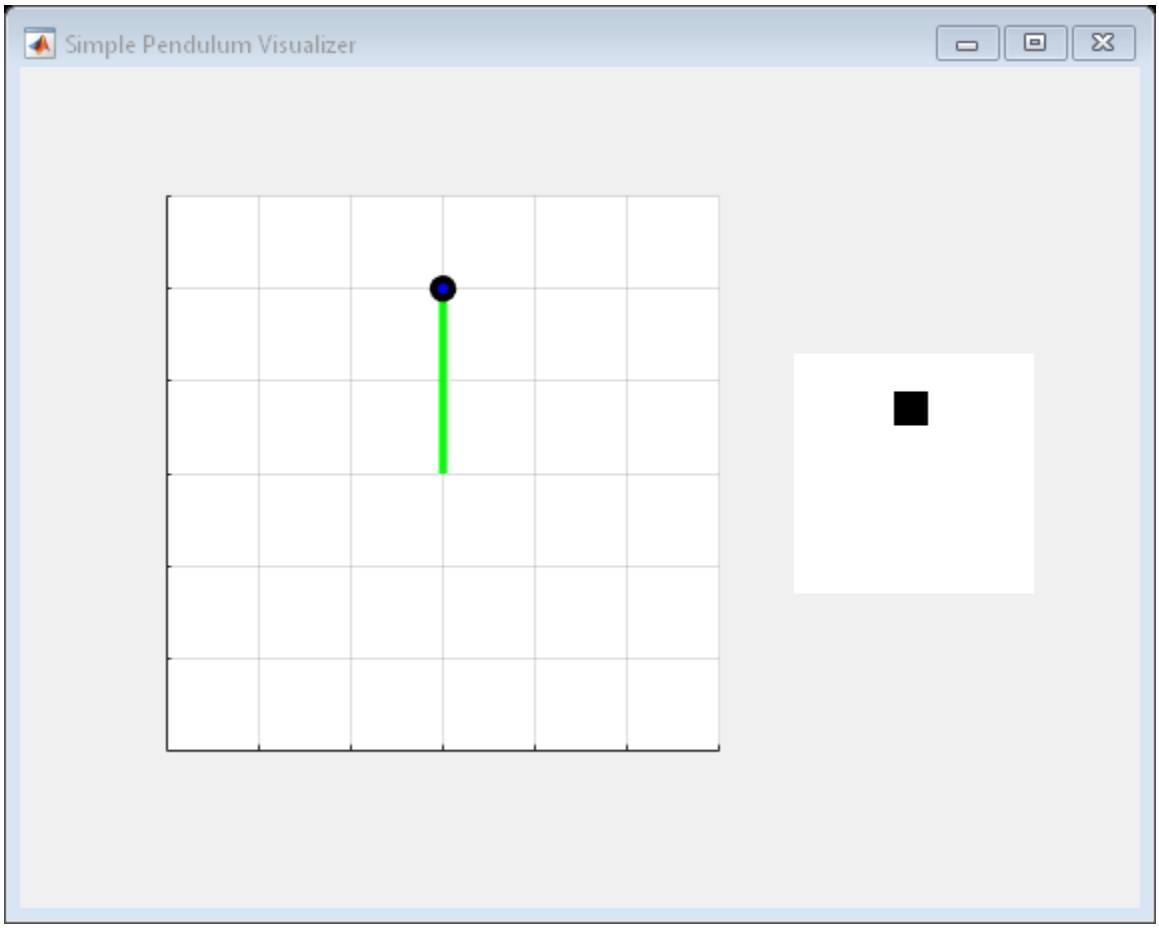
- Run each training for at most 5000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than –1000 over the default window length of five consecutive episodes. At this point, the agent can quickly balance the pendulum in the upright position using minimal control effort.

For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',5000,...
    'MaxStepsPerEpisode',500,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-1000);
```

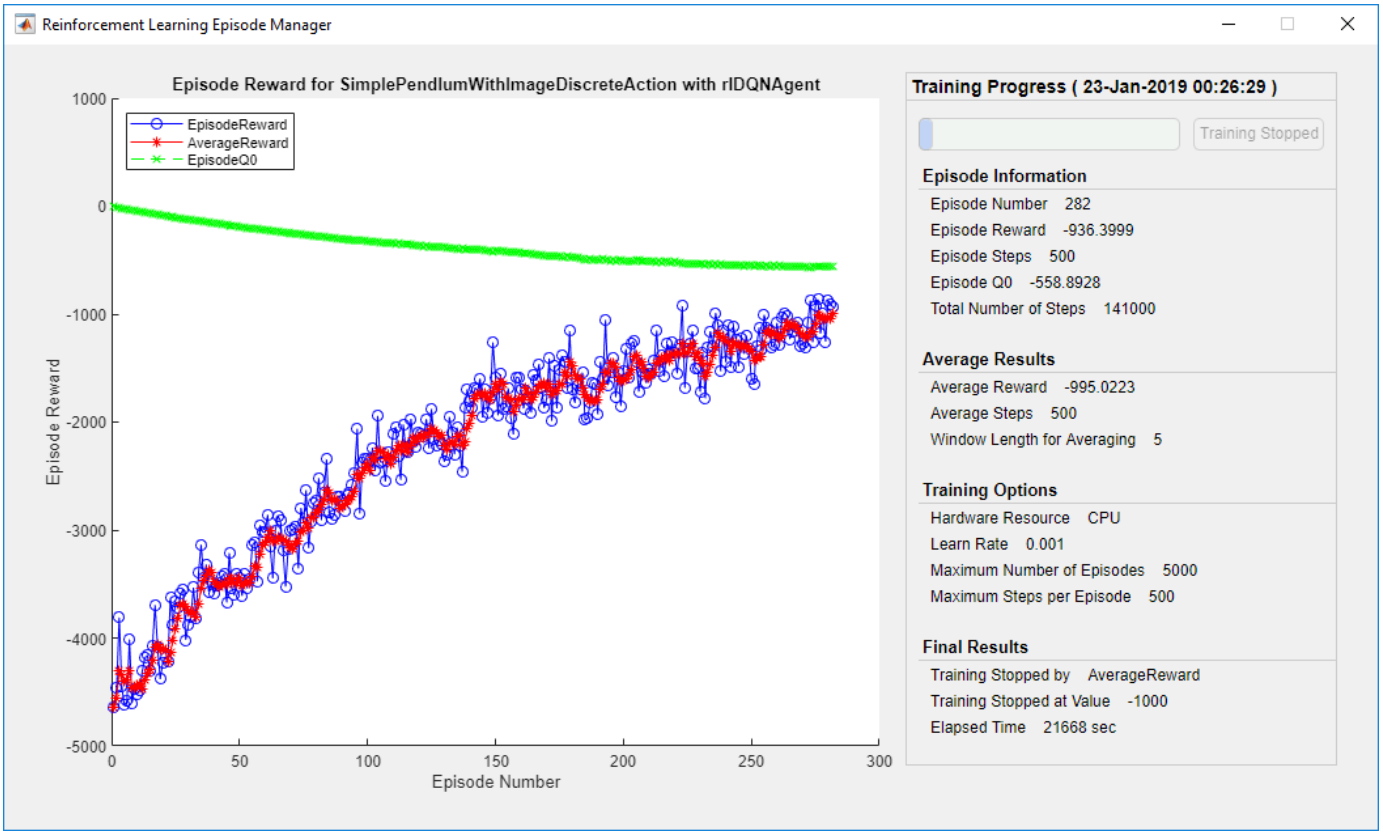
You can visualize the pendulum system during training or simulation by using the `plot` function.

```
plot(env)
```



Train the agent using the `train` (Reinforcement Learning Toolbox) function. This is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

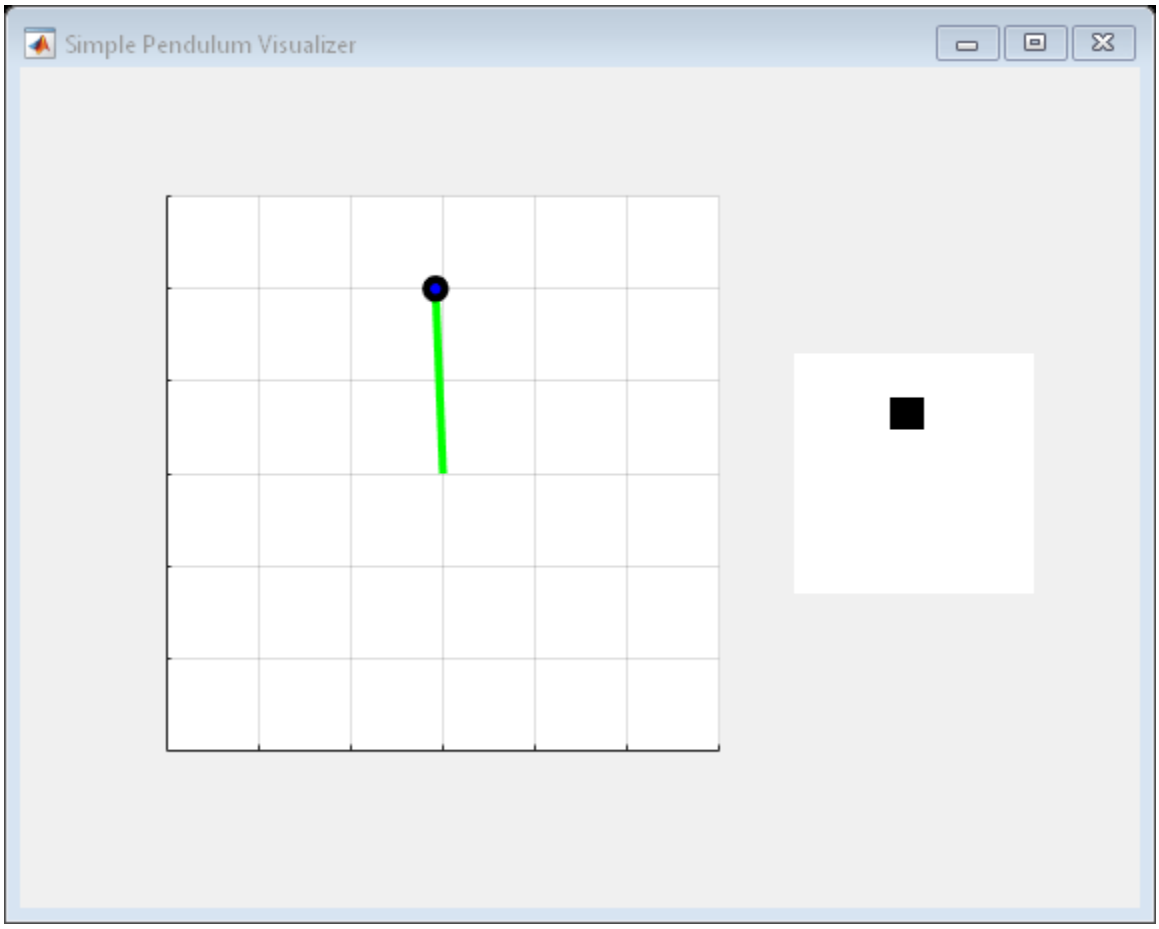
```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the example.
    load('MATLABPendImageDQN.mat','agent');
end
```



Simulate DQN Agent

To validate the performance of the trained agent, simulate it within the pendulum environment. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```




```
totalReward = sum(experience.Reward)
```

```
totalReward = -888.9802
```

See Also

Deep Network Designer | `rlDQNAgent`

More About

- “Train DQN Agent to Swing Up and Balance Pendulum” (Reinforcement Learning Toolbox)

Imitate MPC Controller for Lane Keeping Assist

This example shows how to train, validate, and test a deep neural network that imitates the behavior of a model predictive controller for an automotive lane keeping assist system. In the example, you also compare the behavior of the deep neural network with that of the original controller.

Model predictive control (MPC) solves a constrained quadratic-programming (QP) optimization problem in real time based on the current state of the plant. Because MPC solves its optimization problem in an open-loop fashion, you can potentially replace the controller with a deep neural network. Evaluating a deep neural network is more computationally efficient than solving a QP problem in real time.

If the training of the network sufficiently traverses the state-space for the application, you can create a reasonable approximation of the controller behavior. You can then deploy the network for your control application. You can also use the network as a warm starting point for training the actor network of a reinforcement learning agent. For an example, see “Train DDPG Agent with Pretrained Actor Network” (Reinforcement Learning Toolbox).

Design MPC Controller

Design an MPC controller for lane keeping assist. To do so, first create a dynamic model for the vehicle.

```
[sys,Vx] = createModelForMPCImLKA;
```

Create and design the MPC controller object `mpcobj`. Also, create an `mpcstate` object for setting the initial controller state. For details on the controller design, type `edit createMPCobjImLKA`.

```
[mpcobj,initialState] = createMPCobjImLKA(sys);
```

For more information on designing model predictive controllers for lane keeping assist applications, see “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox) and “Lane Keeping Assist with Lane Detection” (Model Predictive Control Toolbox).

Prepare Input Data

Load the input data from `InputDataFileImLKA.mat`. The columns of the data set follow:

1. Lateral velocity V_y
2. Yaw angle rate r
3. Lateral deviation e_1
4. Relative yaw angle e_2
5. Previous steering angle (control variable) u
6. Measured disturbance (road yaw rate: longitudinal velocity * curvature (ρ))
7. Cost function value
8. MPC iterations
9. Steering angle computed by MPC controller: u*

The data in `InputDataFileImLKA.mat` was created by computing the MPC control action for randomly generated states, previous control actions, and measured disturbances. To generate your own training data, use the `collectDataImLKA` function.

Load the input data.

```
dataStruct = load('InputDataFileImLKA.mat');
data = dataStruct.Data;
```

Divide the input data into training, validation, and testing data. First, determine the number of validation data rows based on a given percentage.

```
totalRows = size(data,1);
validationSplitPercent = 0.1;
numValidationDataRows = floor(validationSplitPercent*totalRows);
```

Determine the number of test data rows based on a given percentage.

```
testSplitPercent = 0.05;
numTestDataRows = floor(testSplitPercent*totalRows);
```

Randomly extract validation and testing data from the input data set. To do so, first randomly extract enough rows for both data sets.

```
randomIdx = randperm(totalRows,numValidationDataRows + numTestDataRows);
randomData = data(randomIdx,:);
```

Divide the random data into validation and testing data.

```
validationData = randomData(1:numValidationDataRows,:);
testData = randomData(numValidationDataRows + 1:end,:);
```

Extract the remaining rows as training data.

```
trainDataIdx = setdiff(1:totalRows,randomIdx);
trainData = data(trainDataIdx,:);
```

Randomize the training data.

```
numTrainDataRows = size(trainData,1);
shuffleIdx = randperm(numTrainDataRows);
shuffledTrainData = trainData(shuffleIdx,:);
```

Reshape the training and validation data into 4-D matrices for use with `trainNetwork`.

```
numObservations = 6;
numActions = 1;

trainInput = shuffledTrainData(:,1:6);
trainOutput = shuffledTrainData(:,9);

validationInput = validationData(:,1:6);
validationOutput = validationData(:,9);
validationCellArray = {validationInput,validationOutput};
```

Reshape the testing data for use with `predict`.

```
testDataInput = testData(:,1:6);
testDataOutput = testData(:,9);
```

Create Deep Neural Network

The deep neural network architecture uses the following layers.

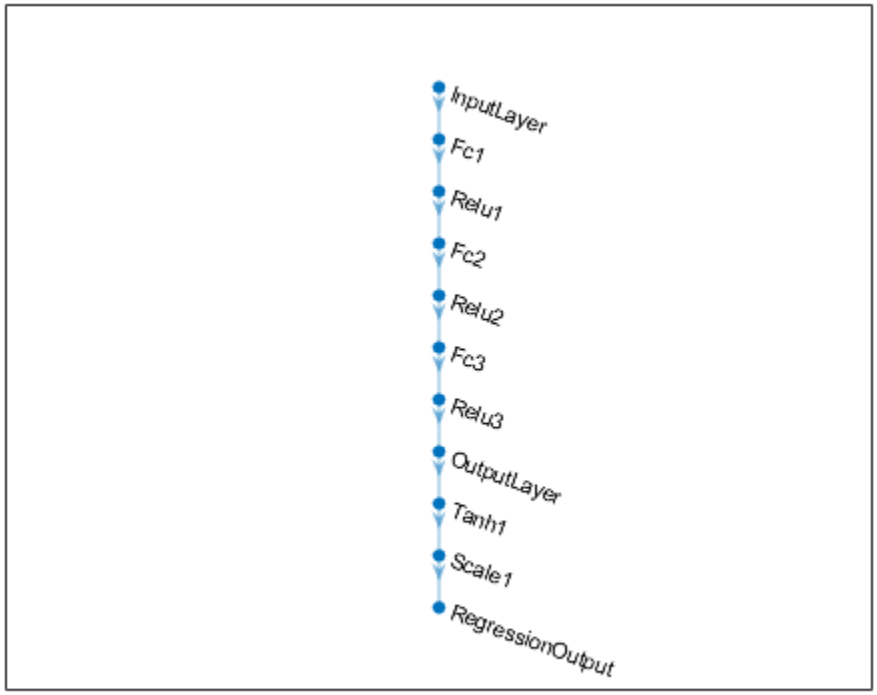
- `imageInputLayer` is the input layer of the neural network.
- `fullyConnectedLayer` multiplies the input by a weight matrix and then adds a bias vector.
- `reluLayer` is the activation function of the neural network.
- `tanhLayer` constrains the value to the range to [-1,1].
- `scalingLayer` scales the value to the range to [-1.04,1.04], implies that the steering angle is constrained to be [-60,60] degrees.
- `regressionLayer` defines the loss function of the neural network.

Create the deep neural network that will imitate the MPC controller after training.

```
imitateMPCNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','InputLayer')
    fullyConnectedLayer(45,'Name','Fc1')
    reluLayer('Name','Relu1')
    fullyConnectedLayer(45,'Name','Fc2')
    reluLayer('Name','Relu2')
    fullyConnectedLayer(45,'Name','Fc3')
    reluLayer('Name','Relu3')
    fullyConnectedLayer(numActions,'Name','OutputLayer')
    tanhLayer('Name','Tanh1')
    scalingLayer('Name','Scale1','Scale',1.04)
    regressionLayer('Name','RegressionOutput')
];
```

Plot the network.

```
plot(layerGraph(imitateMPCNetwork))
```



Train Deep Neural Network

Specify training options.

```
options = trainingOptions('adam', ...
    'Verbose',false, ...
    'Plots','training-progress', ...
    'Shuffle','every-epoch', ...
    'MaxEpochs', 30, ...
    'MiniBatchSize',512, ...
    'ValidationData',validationCellArray, ...
    'InitialLearnRate',1e-3, ...
    'GradientThresholdMethod','absolute-value', ...
    'ExecutionEnvironment','cpu', ...
    'GradientThreshold',10, ...
    'Epsilon',1e-8);
```

Train the deep neural network. To view detailed training information in the Command Window, set the `'Verbose'` training option to `true`.

```
imitateMPCNetObj = trainNetwork(trainInput,trainOutput,imitateMPCNetwork,options);
```

Training of the deep neural network stops after the final iteration.

The training and validation loss are nearly the same for each mini-batch, which indicates that the trained network does not overfit.

Test Trained Network

Check that the trained deep neural network returns steering angles similar to the MPC controller control actions given the test input data. Compute the network output using the `predict` function.

```
predictedTestDataOutput = predict(imitateMPCNetObj,testDataInput);
```

Calculate the root mean squared error (RMSE) between the network output and the testing data.

```
testRMSE = sqrt(mean((testDataOutput - predictedTestDataOutput).^2));
fprintf('Test Data RMSE = %d\n', testRMSE);
```

```
Test Data RMSE = 3.578066e-02
```

The small RMSE value indicates that the network outputs closely reproduce the MPC controller outputs.

Compare Trained Network with MPC Controller

To compare the performance of the MPC controller and the trained deep neural network, run closed-loop simulations using the vehicle plant model.

Generate random initial conditions for the vehicle that are not part of the original input data set, with values selected from the following ranges:

1. Lateral velocity V_y — Range (-2,2) *m/s*
2. Yaw angle rate r — Range (-60,60) *deg/s*
3. Lateral deviation e_1 — Range (-1,1) *m*
4. Relative yaw angle e_2 — Range (-45,45) *deg*
5. Last steering angle (control variable) u — Range (-60,60) *deg*
6. Measured disturbance (road yaw rate, defined as longitudinal velocity * curvature (ρ)) — Range (-0.01,0.01) with a minimum road radius of 100 *m*

```
rng(5e7)
[x0,u0,rho] = generateRandomDataImLKA(data);
```

Set the initial plant state and control action in the `mpcstate` object.

```
initialState.Plant = x0;
initialState.LastMove = u0;
```

Extract the sample time from the MPC controller. Also, set the number of simulation steps.

```
Ts = mpcobj.Ts;
Tsteps = 30;
```

Obtain the `A` and `B` state-space matrices for the vehicle model.

```
A = sys.A;
B = sys.B;
```

Initialize the state and input trajectories for the MPC controller simulation.

```
xHistoryMPC = repmat(x0',Tsteps+1,1);
uHistoryMPC = repmat(u0',Tsteps,1);
```

Run a closed-loop simulation of the MPC controller and the plant using the `mpcmove` function.

```
for k = 1:Tsteps
    % Obtain plant output measurements, which correspond to the plant outputs.
    xk = xHistoryMPC(k,:)';
    % Compute the next cotnrol action using the MPC controller.
    uk = mpcmove(mpcobj,initialState,xk,zeros(1,4),Vx*rho);
    % Store the control action.
    uHistoryMPC(k,:) = uk;
    % Update the state using the control action.
    xHistoryMPC(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Initialize the state and input trajectories for the deep neural network simulation.

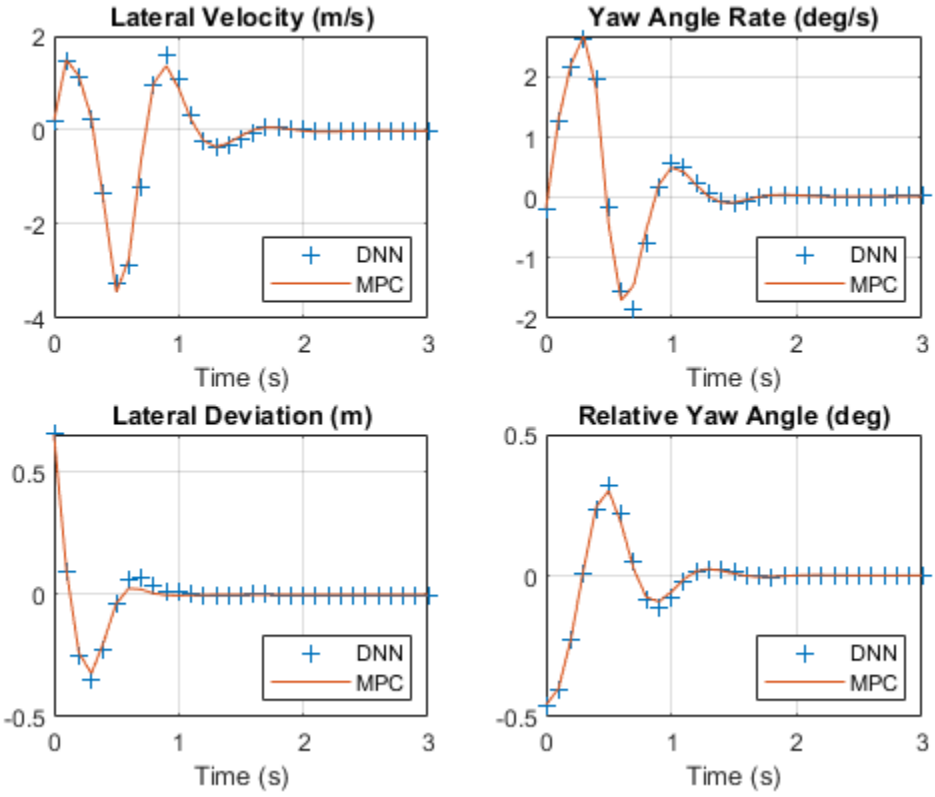
```
xHistoryDNN = repmat(x0',Tsteps+1,1);
uHistoryDNN = repmat(u0',Tsteps,1);
lastMV = u0;
```

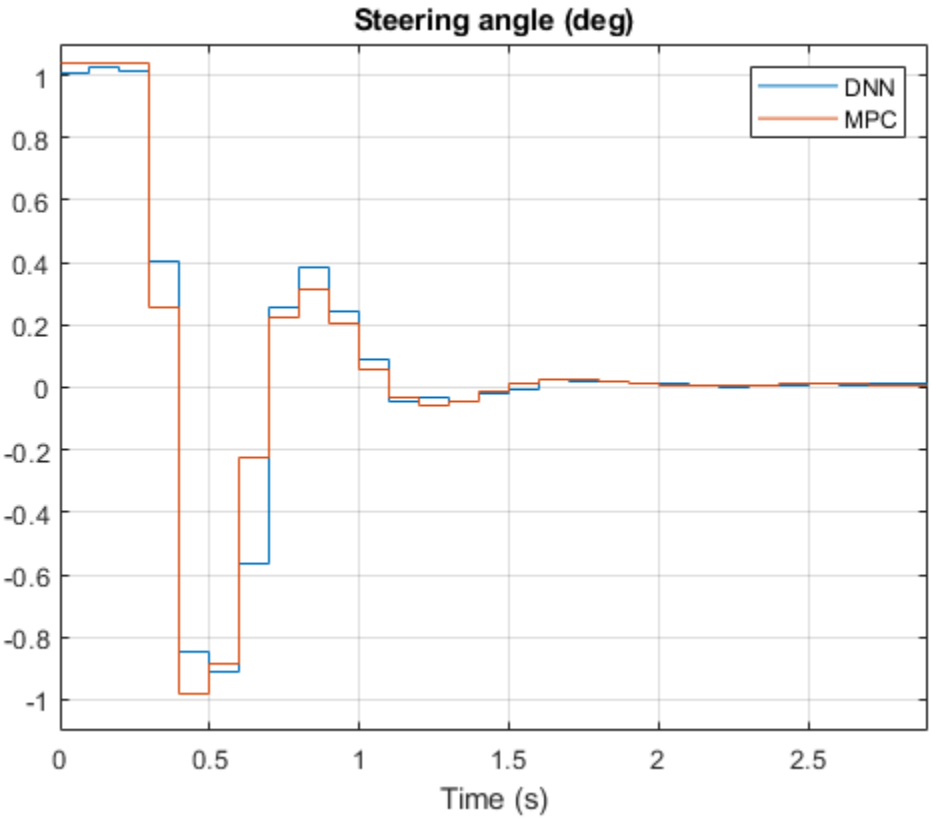
Run a closed-loop simulation of the trained network and the plant. The `neuralnetLKAmove` function computes the deep neural network output using the `predict` function.

```
for k = 1:Tsteps
    % Obtain plant output measurements, which correspond to the plant outputs.
    xk = xHistoryDNN(k,:)';
    % Predict the next move using the trained deep neural network.
    uk = neuralnetLKAmove(imitateMPCNetObj,xk,lastMV,rho);
    % Store the control action and update the last MV for the next step.
    uHistoryDNN(k,:) = uk;
    lastMV = uk;
    % Update the state using the control action.
    xHistoryDNN(k+1,:) = (A*xk + B*[uk;Vx*rho])';
end
```

Plot the results, and compare the MPC controller and trained deep neural network (DNN) trajectories.

```
plotValidationResultsImLKA(Ts,xHistoryDNN,uHistoryDNN,xHistoryMPC,uHistoryMPC);
```


Lateral Velocity (m/s)
Yaw Angle Rate (deg/s)
Lateral Deviation (m)
Relative Yaw Angle (deg)
DNN
MPC
Time (s)



The deep neural network successfully imitates the behavior of the MPC controller. The vehicle state and control action trajectories for the controller and the deep neural network closely align.

See Also

`trainNetwork` | `predict` | `mpcmove`

More About

- “Lane Keeping Assist System Using Model Predictive Control” (Model Predictive Control Toolbox)
- “Lane Keeping Assist with Lane Detection” (Model Predictive Control Toolbox)

Train DDPG Agent to Control Flying Robot

This example shows how to train a deep deterministic policy gradient (DDPG) agent to generate trajectories for a flying robot modeled in Simulink®. For more information on DDPG agents, see "Deep Deterministic Policy Gradient Agents" (Reinforcement Learning Toolbox).

Flying Robot Model

The reinforcement learning environment for this example is a flying robot with its initial condition randomized around a ring of radius 15 m. The orientation of the robot is also randomized. The robot has two thrusters mounted on the side of the body that are used to propel and steer the robot. The training goal is to drive the robot from its initial condition to the origin facing east.

Open the model.

```
mdl = 'rlFlyingRobotEnv';
open_system(mdl)
```

Set the initial model state variables.

```
theta0 = 0;
x0 = -15;
y0 = 0;
```

Define the sample time `Ts` and the simulation duration `Tf`.

```
Ts = 0.4;
Tf = 30;
```

For this model:

- The goal orientation is `0` rad (robot facing east).
- The thrust from each actuator is bounded from `-1` to `1` N
- The observations from the environment are the position, orientation (sine and cosine of orientation), velocity, and angular velocity of the robot.
- The reward r_t provided at every time step is

$$r_1 = 10\left(\left(x_t^2 + y_t^2 + \theta_t^2\right) < 0.5\right)$$

$$r_2 = -100(|x_t| \geq 20 \,||\, |y_t| \geq 20)$$

$$r_3 = -\left(0.2(R_{t-1} + L_{t-1})^2 + 0.3(R_{t-1} - L_{t-1})^2 + 0.03x_t^2 + 0.03y_t^2 + 0.02\theta_t^2\right)$$

$$r_t = r_1 + r_2 + r_3$$

where:

- x_t is the position of the robot along the x-axis.
- y_t is the position of the robot along the y-axis.
- θ_t is the orientation of the robot.
- L_{t-1} is the control effort from the left thruster.

- R_{t-1} is the control effort from the right thruster.
- r_1 is the reward when the robot is close to the goal.
- r_2 is the penalty when the robot drives beyond `20` m in either the x or y direction. The simulation is terminated when $r_2 < 0$.
- r_3 is a QR penalty that penalizes distance from the goal and control effort.

Create Integrated Model

To train an agent for the `FlyingRobotEnv` model, use the `createIntegratedEnv` function to automatically generate an integrated model with the RL Agent block that is ready for training.

```
integratedMdl = 'IntegratedFlyingRobot';
[~,agentBlk,observationInfo,actionInfo] = createIntegratedEnv(mdl,integratedMdl);
```

Actions and Observations

Before creating the environment object, specify names for the observation and action specifications, and bound the thrust actions between `-1` and `1`.

The observation signals for this environment are observation = $\left[x\ y\ \dot{x}\ \dot{y}\ \sin(\theta)\ \cos(\theta)\ \dot{\theta}\right]^T$.

```
numObs = prod(observationInfo.Dimension);
observationInfo.Name = 'observations';
```

The action signals for this environment are action = $\left[T_R\ T_L\right]^T$.

```
numAct = prod(actionInfo.Dimension);
actionInfo.LowerLimit = -ones(numAct,1);
actionInfo.UpperLimit =  ones(numAct,1);
actionInfo.Name = 'thrusts';
```

Create Environment Interface

Create an environment interface for the flying robot using the integrated model.

```
env = rlSimulinkEnv(integratedMdl,agentBlk,observationInfo,actionInfo);
```

Reset Function

Create a custom reset function that randomizes the initial position of the robot along a ring of radius `15` m and the initial orientation. For details on the reset function, see `flyingRobotResetFcn`.

```
env.ResetFcn = @(in) flyingRobotResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG agent

A DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. To create the critic, first create a deep neural network with two inputs (the observation and action) and one output. For more information on creating a neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
% Specify the number of outputs for the hidden layers.
hiddenLayerSize = 100;

observationPath = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];
actionPath = [
    featureInputLayer(numAct,'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc5')];

% Create the layer graph.
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);

% Connect actionPath to observationPath.
criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

Specify options for the critic using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation specification for the critic. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input (the observation) and one output (the action).

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation` (Reinforcement Learning Toolbox).

```
actorNetwork = [
    featureInputLayer(numObs,'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numAct,'Name','fc4')
    tanhLayer('Name','tanh1')];

actorOptions = rlRepresentationOptions('LearnRate',1e-04,'GradientThreshold',1);

actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'tanh1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6 ,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',256);
agentOptions.NoiseOptions.Variance = 1e-1;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
```

Then, create the agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent` (Reinforcement Learning Toolbox).

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training for at most `20000` episodes, with each episode lasting at most `ceil(Tf/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than `415` over 10 consecutive episodes. At this point, the agent can drive the flying robot to the goal position.
- Save a copy of the agent for each episode where the cumulative reward is greater than `415`.

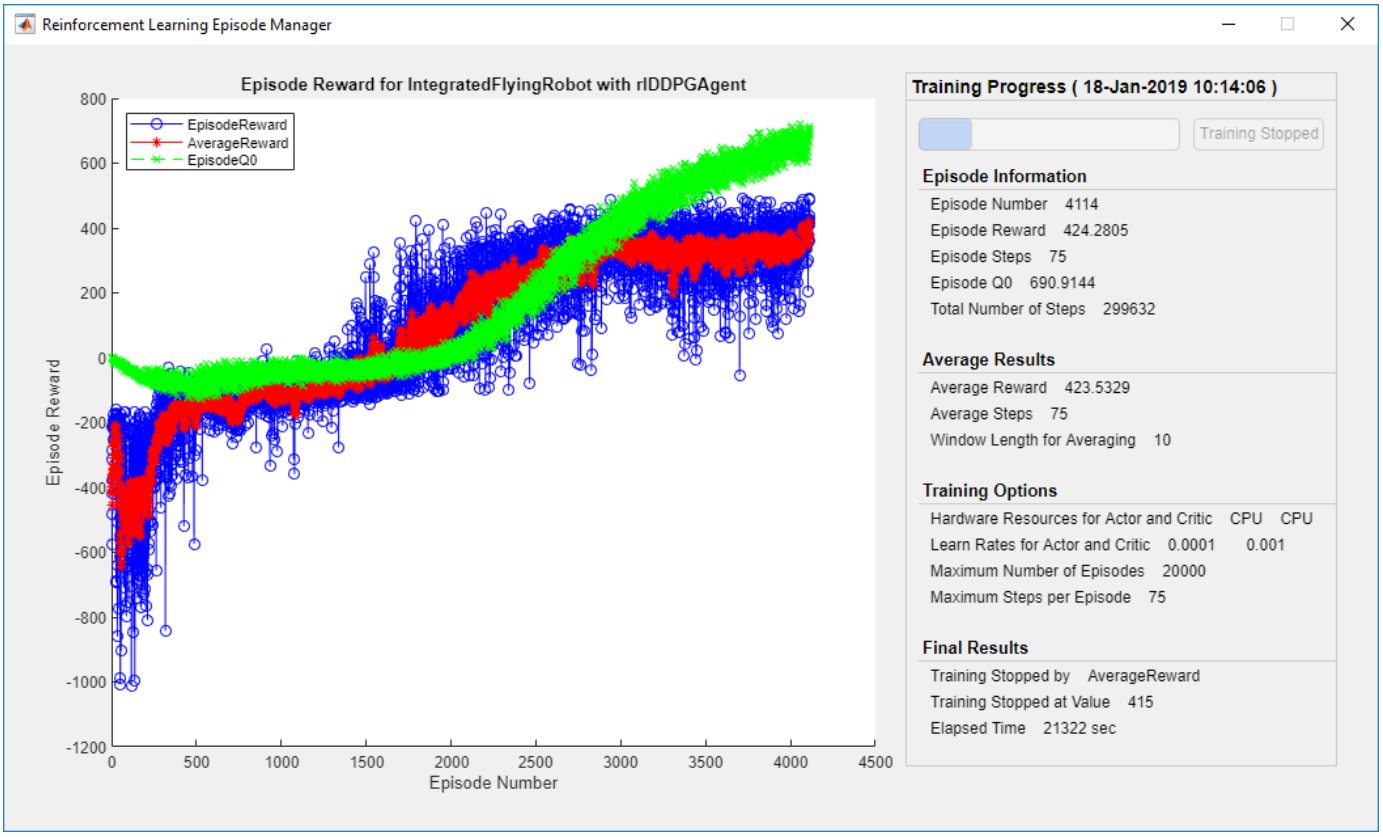
For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 20000;
maxsteps = ceil(Tf/Ts);
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'StopOnError',"on",...
    'Verbose',false,...
    'Plots',"training-progress",...
    'StopTrainingCriteria',"AverageReward",...
    'StopTrainingValue',415,...
    'ScoreAveragingWindowLength',10,...
    'SaveAgentCriteria',"EpisodeReward",...
    'SaveAgentValue',415);
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training is a computationally intensive process that takes several hours to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load the pretrained agent for the example.
```

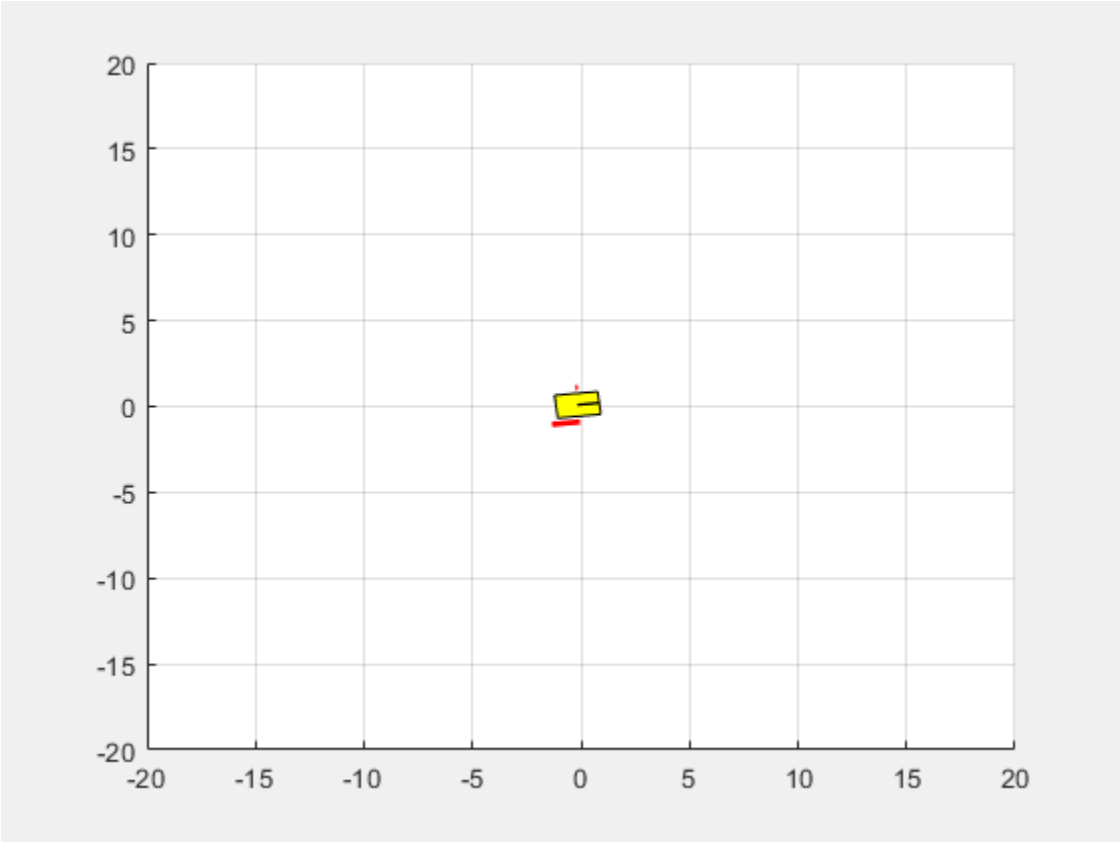
```
    load('FlyingRobotDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

```
simOptions = rlSimulationOptions('MaxSteps',maxsteps);
experience = sim(env,agent,simOptions);
```



See Also

`train` | `rlDDPGAgent`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)

Train Biped Robot to Walk Using Reinforcement Learning Agents

This example shows how to train a biped robot to walk using both a deep deterministic policy gradient (DDPG) agent and a twin-delayed deep deterministic policy gradient (TD3) agent. In the example, you also compare the performance of these trained agents. The robot in this example is modeled in Simscape™ Multibody™.

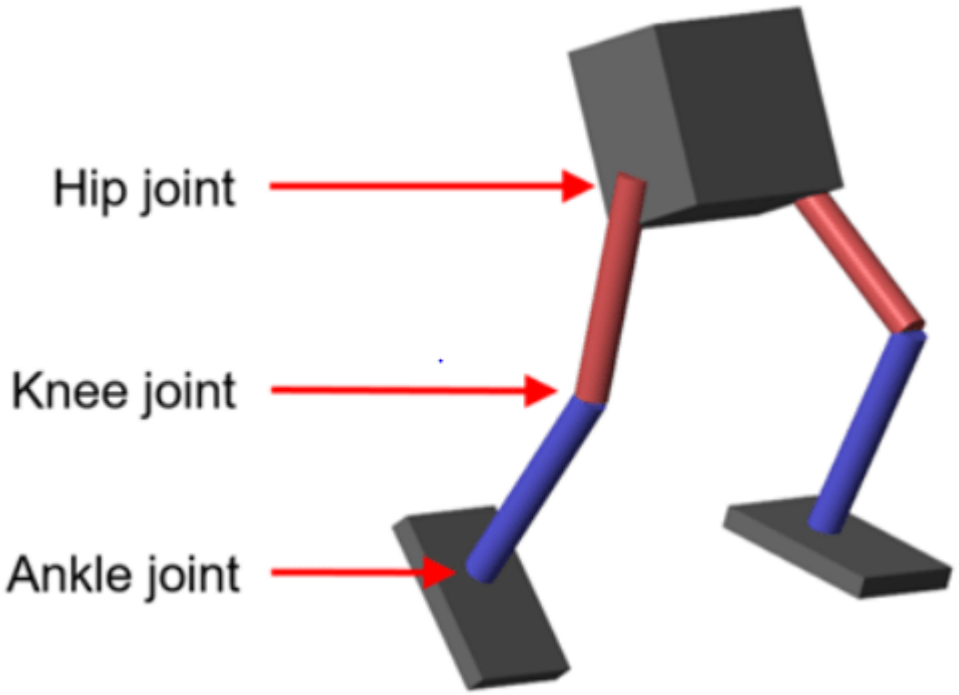
For more information on these agents, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox) and “Twin-Delayed Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

For the purpose of comparison in this example, this example trains both agents on the biped robot environment with the same model parameters. The example also configures the agents to have the following settings in common.

- Initial condition strategy of the biped robot
- Network structure of actor and critic, inspired by [1]
- Options for actor and critic representations
- Training options (sample time, discount factor, mini-batch size, experience buffer length, exploration noise)

Biped Robot Model

The reinforcement learning environment for this example is a biped robot. The training goal is to make the robot walk in a straight line using minimal control effort.

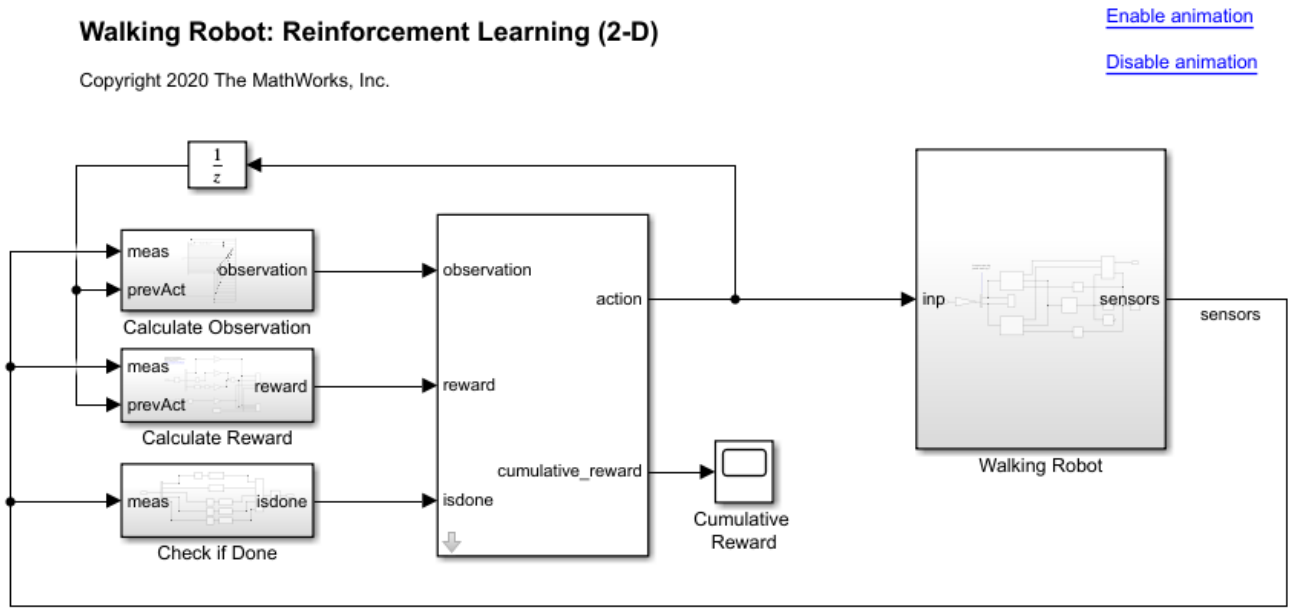


Load the parameters of the model into the MATLAB® workspace.

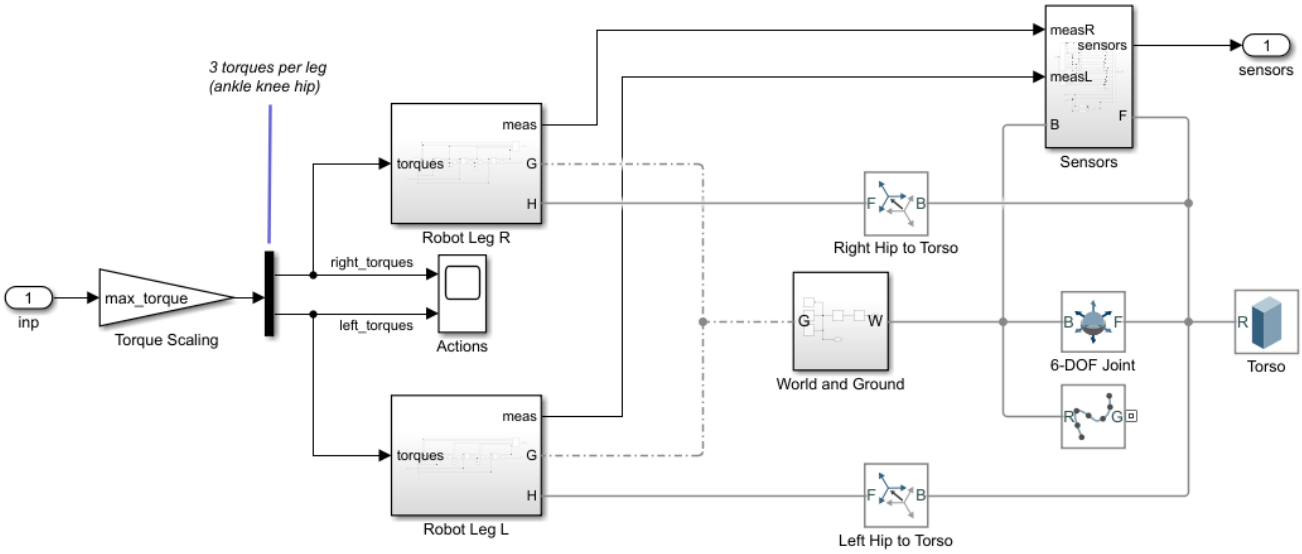
```
robotParametersRL
```

Open the Simulink model.

```
mdl = 'rlWalkingBipedRobot';
open_system(mdl)
```



The robot is modeled using Simscape Multibody.



For this model:

- In the neutral 0 rad position, both of the legs are straight and the ankles are flat.
- The foot contact is modeled using the Spatial Contact Force (Simscape Multibody) block.
- The agent can control 3 individual joints (ankle, knee, and hip) on both legs of the robot by applying torque signals from -3 to 3 N·m. The actual computed action signals are normalized between -1 and 1.

The environment provides the following 29 observations to the agent.

- Y (lateral) and Z (vertical) translations of the torso center of mass. The translation in the Z direction is normalized to a similar range as the other observations.
- X (forward), Y (lateral), and Z (vertical) translation velocities.
- Yaw, pitch, and roll angles of the torso.
- Yaw, pitch, and roll angular velocities of the torso.
- Angular positions and velocities of the three joints (ankle, knee, hip) on both legs.
- Action values from the previous time step.

The episode terminates if either of the following conditions occur.

- The robot torso center of mass is less than 0.1 m in the Z direction (the robot falls) or more than 1 m in the either Y direction (the robot moves too far to the side).
- The absolute value of either the roll, pitch, or yaw is greater than 0.7854 rad.

The following reward function r_t, which is provided at every time step is inspired by [2].

$$\boldsymbol{r_t} = v_x - 3y^2 - 50\hat{z}^2 + 25\frac{\mathrm{Ts}}{\mathrm{Tf}} - 0.02\sum_i {u_{t-1}^i}^2$$

Here:

- v_x is the translation velocity in the X direction (forward toward goal) of the robot.
- y is the lateral translation displacement of the robot from the target straight line trajectory.
- $\hat{z}$ is the normalized vertical translation displacement of the robot center of mass.
- u_{t-1}^i is the torque from joint i from the previous time step.
- Ts is the sample time of the environment.
- Tf is the final simulation time of the environment.

This reward function encourages the agent to move forward by providing a positive reward for positive forward velocity. It also encourages the agent to avoid episode termination by providing a constant reward ($25\frac{\mathrm{Ts}}{\mathrm{Tf}}$) at every time step. The other terms in the reward function are penalties for substantial changes in lateral and vertical translations, and for the use of excess control effort.

Create Environment Interface

Create the observation specification.

```
numObs = 29;
obsInfo = rlNumericSpec([numObs 1]);
obsInfo.Name = 'observations';
```

Create the action specification.

```
numAct = 6;
actInfo = rlNumericSpec([numAct 1],'LowerLimit',-1,'UpperLimit',1);
actInfo.Name = 'foot_torque';
```

Create the environment interface for the walking robot model.

```
blk = [mdl,'/RL Agent'];
env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo);
env.ResetFcn = @(in) walkerResetFcn(in,upper_leg_length/100,lower_leg_length/100,h/100);
```

Select and Create Agent for Training

This example provides the option to train the robot either using either a DDPG or TD3 agent. To simulate the robot with the agent of your choice, set the `AgentSelection` flag accordingly.

```
AgentSelection = 'TD3';
switch AgentSelection
    case 'DDPG'
        agent = createDDPGAgent(numObs,obsInfo,numAct,actInfo,Ts);
    case 'TD3'
        agent = createTD3Agent(numObs,obsInfo,numAct,actInfo,Ts);
    otherwise
        disp('Enter DDPG or TD3 for AgentSelection')
end
```

The `createDDPGAgent` and `createTD3Agent` helper functions perform the following actions.

- Create actor and critic networks.
- Specify options for actor and critic representations.
- Create actor and critic representations using created networks and specified options.
- Configure agent specific options.
- Create agent.

DDPG Agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. A DDPG agent decides which action to take given observations by using an actor representation. The actor and critic networks for this example are inspired by [1].

For details on the creating the DDPG agent, see the `createDDPGAgent` helper function. For information on configuring DDPG agent options, see `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

For more information on creating a deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox). For an example that creates neural networks for DDPG agents, see “Train DDPG Agent to Control Double Integrator System” (Reinforcement Learning Toolbox).

TD3 Agent

A TD3 agent approximates the long-term reward given observations and actions using two critic value function representations. A TD3 agent decides which action to take given observations using an actor representation. The structure of the actor and critic networks used for this agent are the same as the ones used for DDPG agent.

A DDPG agent can overestimate the Q value. Since the agent uses the Q value to update its policy (actor), the resultant policy can be suboptimal and accumulating training errors can lead to divergent behavior. The TD3 algorithm is an extension of DDPG with improvements that make it more robust by preventing overestimation of Q values [3].

- Two critic networks — TD3 agents learn two critic networks independently and use the minimum value function estimate to update the actor (policy). Doing so prevents accumulation of error in subsequent steps and overestimation of Q values.
- Addition of target policy noise — Adding clipped noise to value functions smooths out Q function values over similar actions. Doing so prevents learning an incorrect sharp peak of noisy value estimate.
- Delayed policy and target updates — For a TD3 agent, delaying the actor network update allows more time for the Q function to reduce error (get closer to the required target) before updating the policy. Doing so prevents variance in value estimates and results in a higher quality policy update.

For details on the creating the TD3 agent, see the `createTD3Agent` helper function. For information on configuring TD3 agent options, see `rlTD3AgentOptions` (Reinforcement Learning Toolbox).

Specify Training Options and Train Agent

For this example, the training options for the DDPG and TD3 agents are the same.

- Run each training session for 2000 episodes with each episode lasting at most `maxSteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option).
- Terminate the training only when it reaches the maximum number of episodes (`maxEpisodes`). Doing so allows the comparison of the learning curves for multiple agents over the entire training session.

For more information and additional options, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxEpisodes = 2000;
maxSteps = floor(Tf/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxEpisodes,...
    'MaxStepsPerEpisode',maxSteps,...
    'ScoreAveragingWindowLength',250,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeCount',...
    'StopTrainingValue',maxEpisodes,...
    'SaveAgentCriteria','EpisodeCount',...
    'SaveAgentValue',maxEpisodes);
```

To train the agent in parallel, specify the following training options. Training in parallel requires Parallel Computing Toolbox™. If you do not have Parallel Computing Toolbox software installed, set `UseParallel` to `false`.

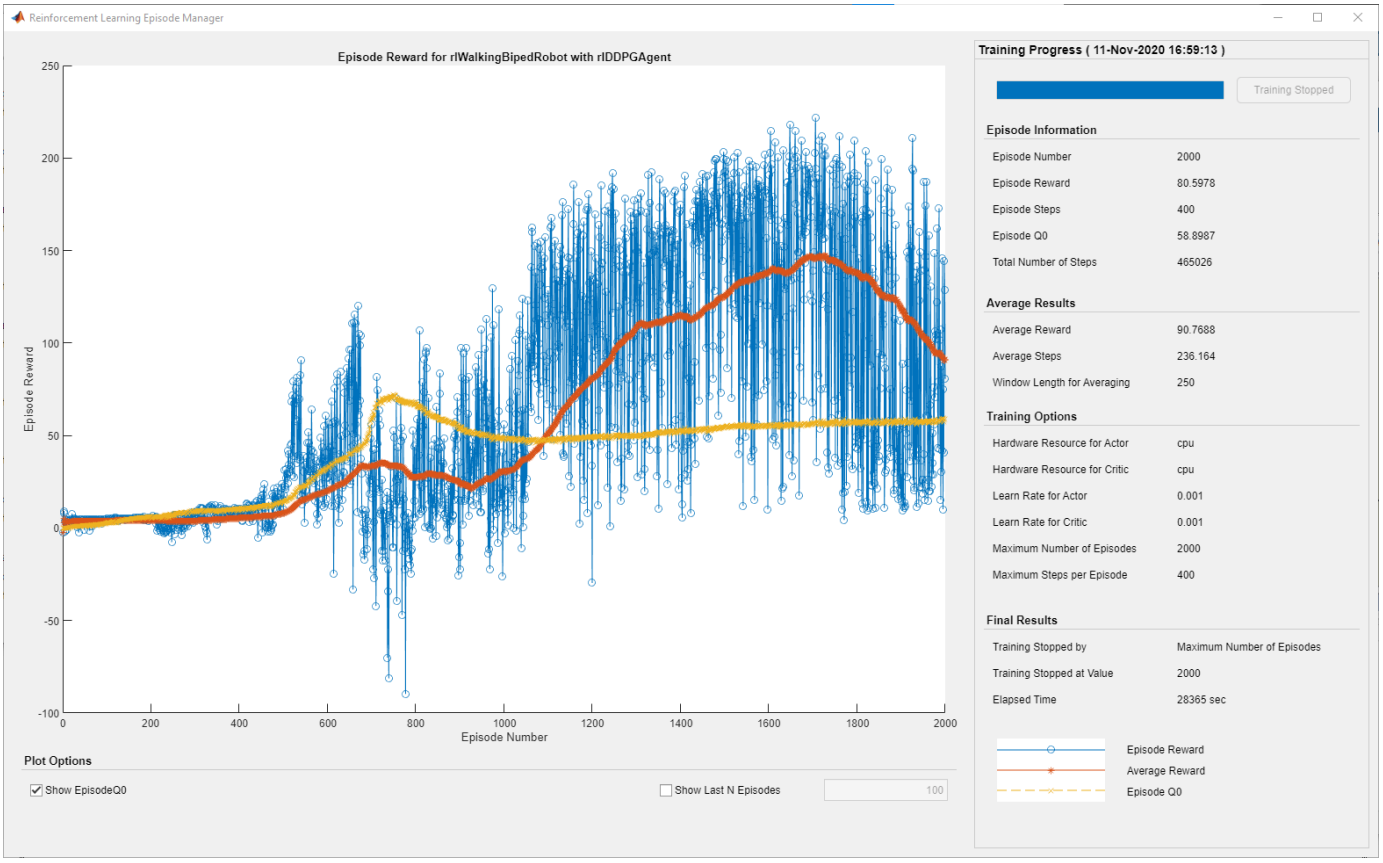
- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.

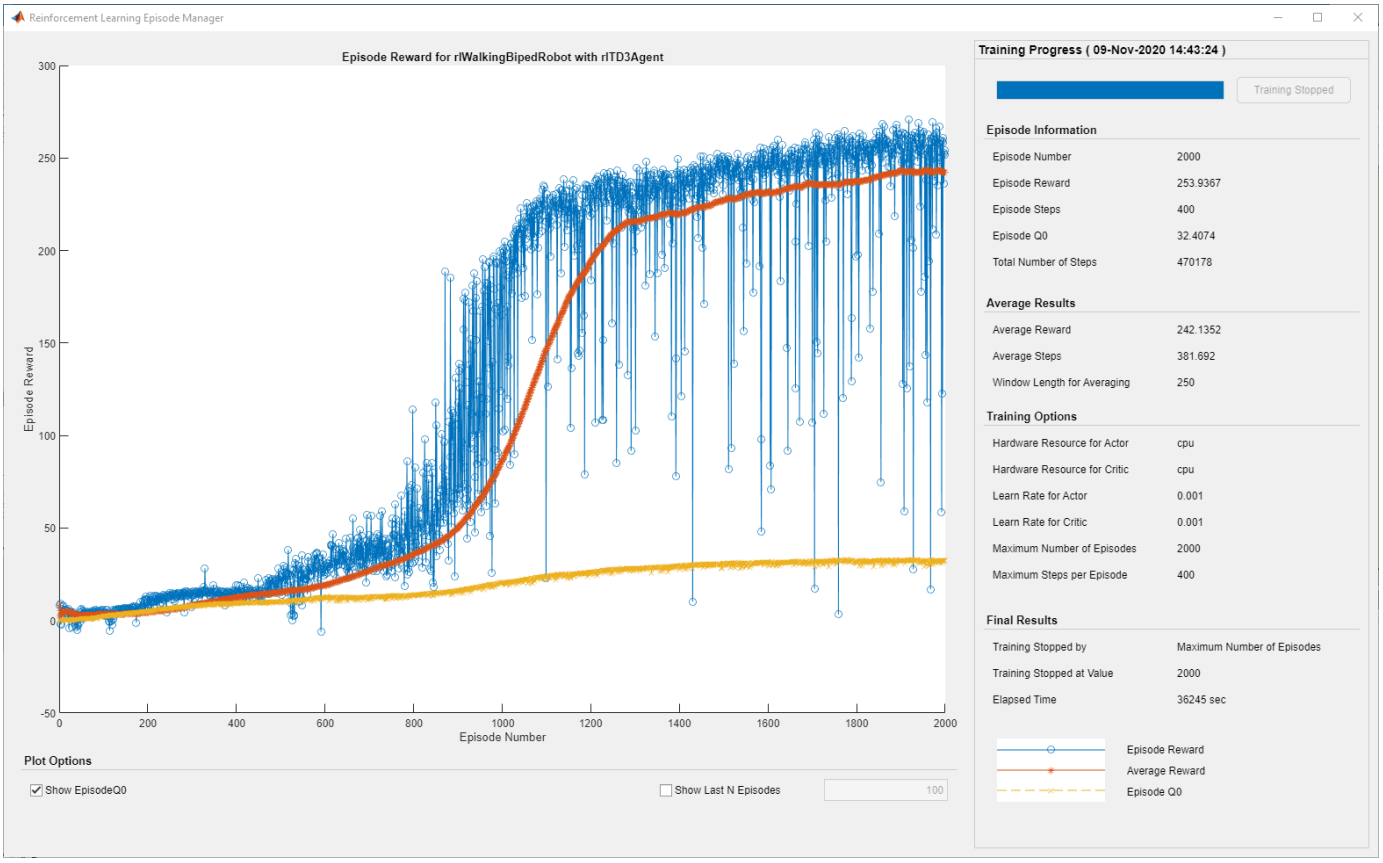
- After every 32 steps, have each worker send experiences to the parallel pool client (the MATLAB® process which starts the training). DDPG and TD3 agents require workers to send experiences to the client.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = 'async';
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = 'Experiences';
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. This process is computationally intensive and takes several hours to complete for each agent. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness in the parallel training, you can expect different training results from the plots that follow. The pretrained agents were trained in parallel using four workers.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load a pretrained agent for the selected agent type.
    if strcmp(AgentSelection,'DDPG')
       load('rlWalkingBipedRobotDDPG.mat','agent')
    else
       load('rlWalkingBipedRobotTD3.mat','agent')
    end
end
```


Reinforcement Learning Episode Manager
Episode Reward for rlWalkingBipedRobot with rlDDPGAgent
Episode Reward
Episode Number
Plot Options
Show EpisodeQ0
Show Last N Episodes
100
Training Progress (11-Nov-2020 16:59:13)
Training Stopped
Episode Information
Episode Number 2000
Episode Reward 80.5978
Episode Steps 400
Episode Q0 58.8987
Total Number of Steps 465026
Average Results
Average Reward 90.7688
Average Steps 236.164
Window Length for Averaging 250
Training Options
Hardware Resource for Actor cpu
Hardware Resource for Critic cpu
Learn Rate for Actor 0.001
Learn Rate for Critic 0.001
Maximum Number of Episodes 2000
Maximum Steps per Episode 400
Final Results
Training Stopped by Maximum Number of Episodes
Training Stopped at Value 2000
Elapsed Time 28365 sec
Episode Reward
Average Reward
Episode Q0



For the preceding example training curves, the average time per training step for the DDPG and TD3 agents are 0.11 and 0.12 seconds, respectively. The TD3 agent takes more training time per step because it updates two critic networks compared to the single critic used for DDPG.

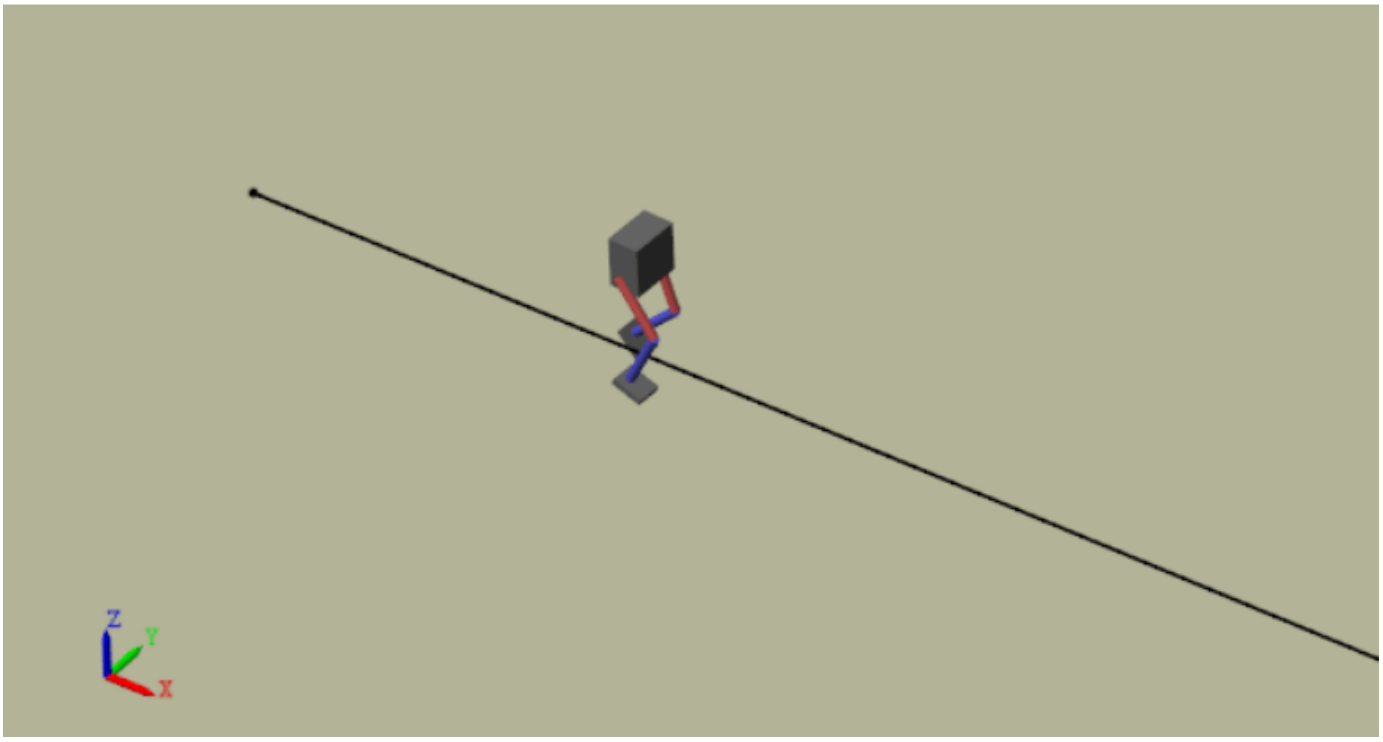
Simulate Trained Agents

Fix the random generator seed for reproducibility.

```
rng(0)
```

To validate the performance of the trained agent, simulate it within the biped robot environment. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

```
simOptions = rlSimulationOptions('MaxSteps',maxSteps);
experience = sim(env,agent,simOptions);
```

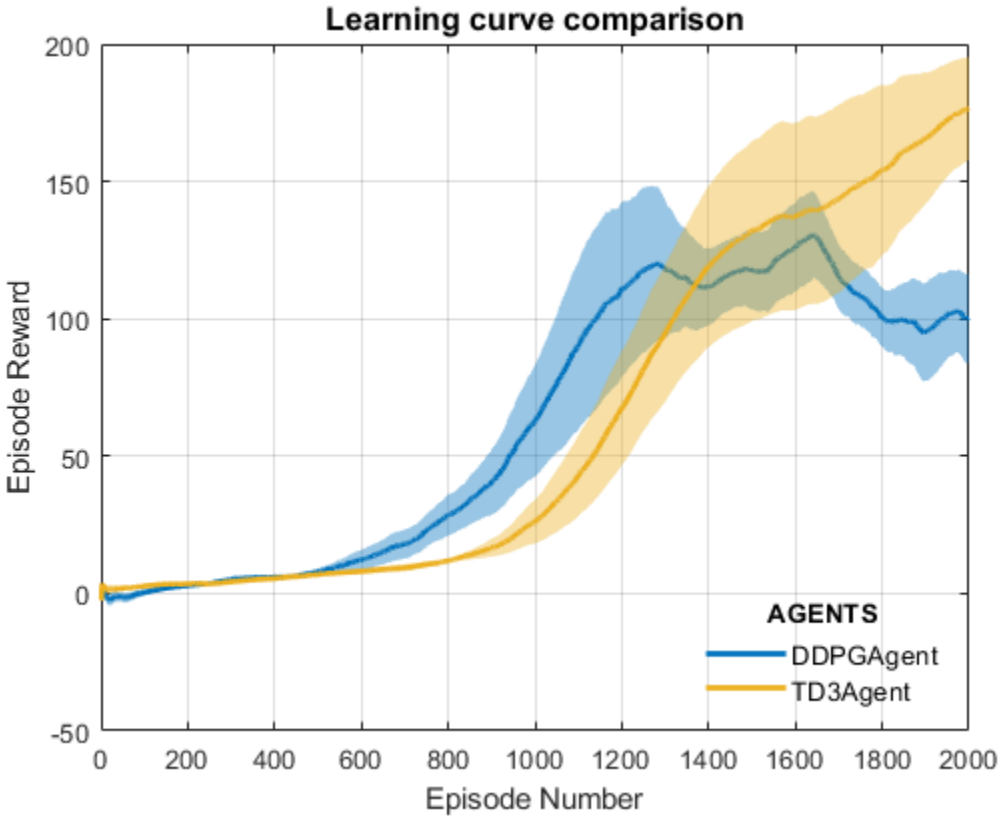


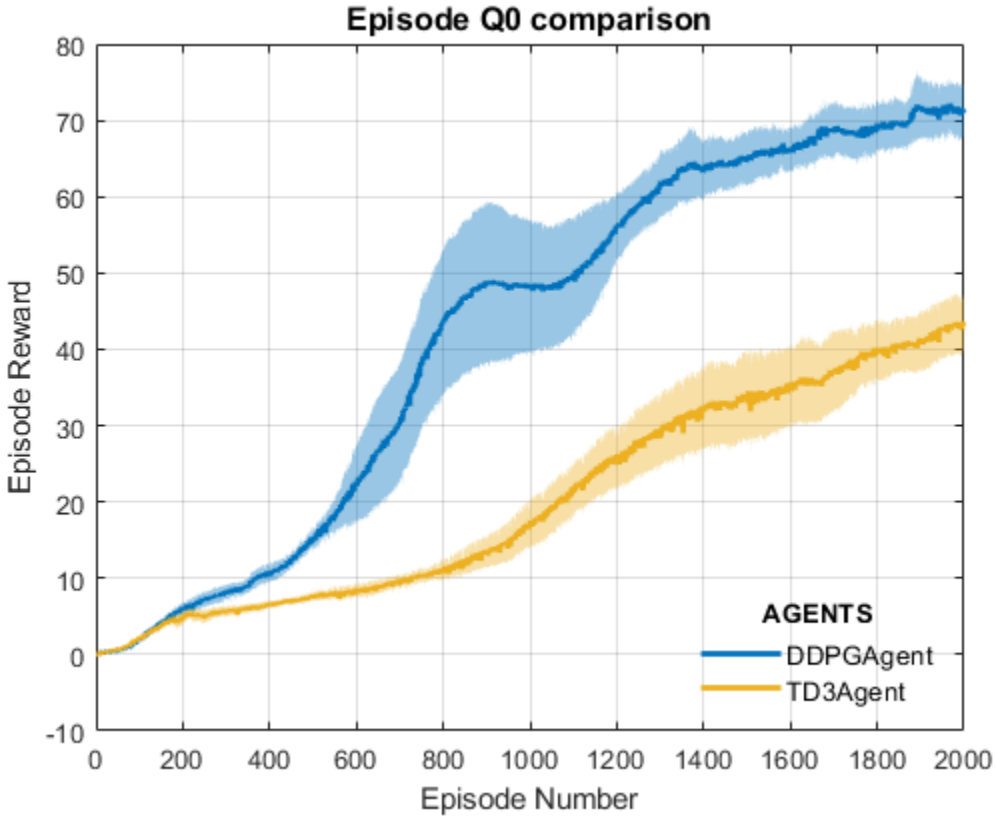
Compare Agent Performance

For the following agent comparison, each agent was trained five times using a different random seed each time. Due to the random exploration noise and the randomness in the parallel training, the learning curve for each run is different. Since the training of agents for multiple runs takes several days to complete, this comparison uses pretrained agents.

For the DDPG and TD3 agents, plot the average and standard deviation of the episode reward (top plot) and the episode Q0 value (bottom plot). The episode Q0 value is the critic estimate of the discounted long-term reward at the start of each episode given the initial observation of the environment. For a well-designed critic, the episode Q0 value approaches the true discounted long-term reward.

```
comparePerformance('DDPGAgent','TD3Agent')
```


Learning curve comparison
Episode Reward
Episode Number
200
150
100
50
0
-50
0
200
400
600
800
1000
1200
1400
1600
1800
2000
AGENTS
DDPGAgent
TD3Agent



Based on the Learning curve comparison plot:

- The DDPG agent appears to pick up learning faster (around episode number 600 on average) but hits a local minimum. TD3 starts slower but eventually achieves higher rewards than DDPG as it avoids overestimation of Q values.
- The TD3 agent shows a steady improvement in its learning curve, which suggests improved stability when compared to the DDPG agent.

Based on the Episode Q0 comparison plot:

- For the TD3 agent, the critic estimate of the discounted long-term reward (for 2000 episodes) is lower compared to the DDPG agent. This difference is because the TD3 algorithm takes a conservative approach in updating its targets by using a minimum of two Q functions. This behavior is further enhanced because of delayed updates to the targets.
- Although the TD3 estimate for these 2000 episodes is low, the TD3 agent shows a steady increase in the episode Q0 values, unlike the DDPG agent.

In this example, the training was stopped at 2000 episodes. For a larger training period, the TD3 agent with its steady increase in estimates shows the potential to converge to the true discounted long-term reward.

For another example on how to train a humanoid robot to walk using a DDPG agent, see “Train Humanoid Walker” (Simscape Multibody). For an example on how to train a quadruped robot to walk using a DDPG agent, see “Quadruped Robot Locomotion Using DDPG Agent” (Reinforcement Learning Toolbox).

References

[1] Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous Control with Deep Reinforcement Learning." Preprint, submitted July 5, 2019. https://arxiv.org/abs/1509.02971.

[2] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. "Emergence of Locomotion Behaviours in Rich Environments." Preprint, submitted July 10, 2017. https://arxiv.org/abs/1707.02286.

[3] Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods." Preprint, submitted October 22, 2018. https://arxiv.org/abs/1802.09477.

See Also

`train`

More About

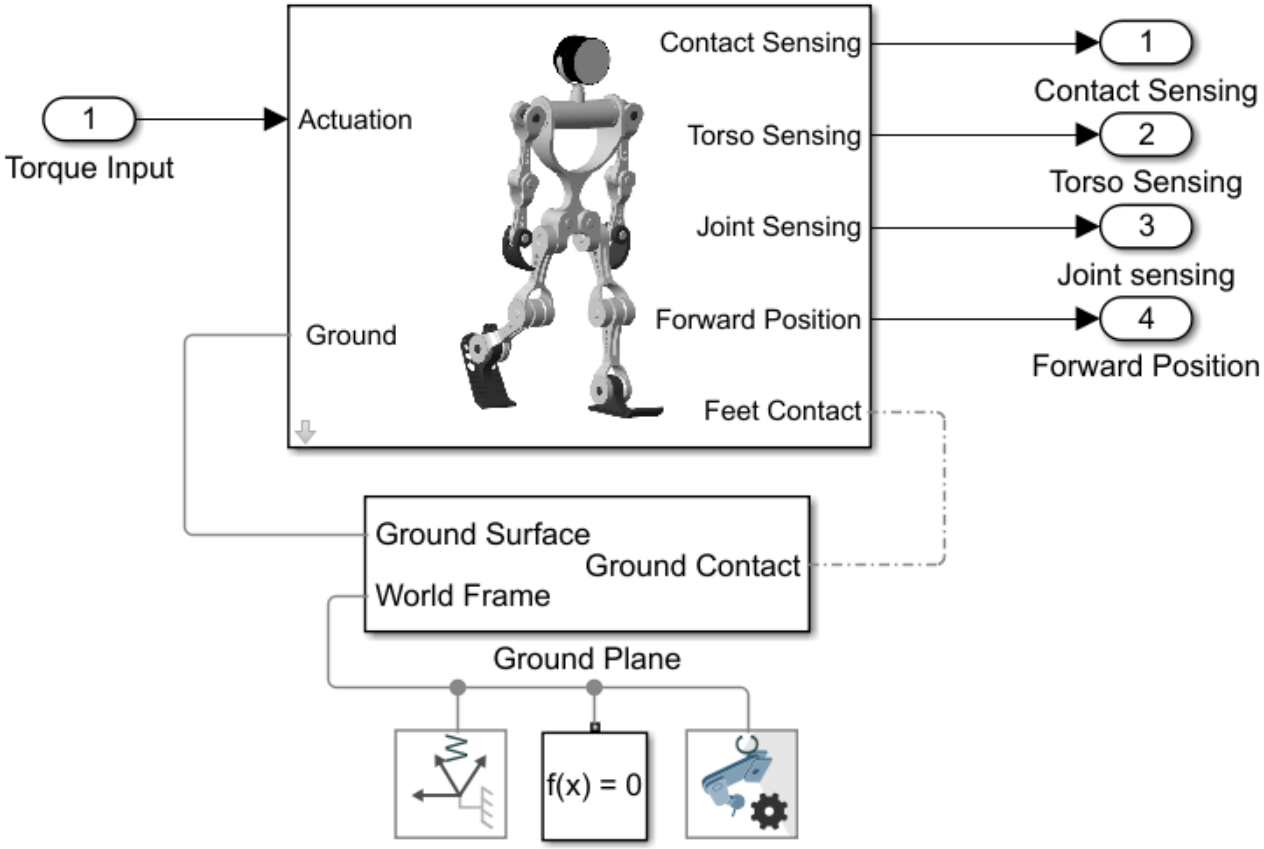
- “Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Define Reward Signals” (Reinforcement Learning Toolbox)

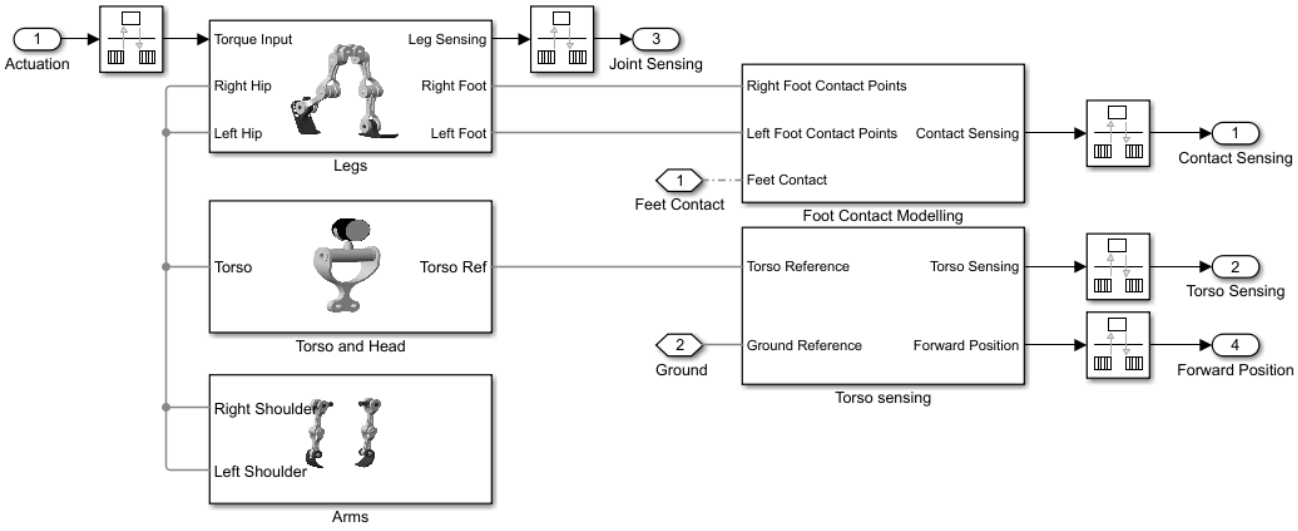
Train Humanoid Walker

This example shows how to model a humanoid robot using “Simscape Multibody”™ and train it using either a genetic algorithm (which requires a “Global Optimization Toolbox” license) or reinforcement learning (which requires “Deep Learning Toolbox”™ and “Reinforcement Learning Toolbox”™ licenses).

Humanoid Walker Model

This example is based on a humanoid robot model. You can open the model by entering `sm_import_humanoid_urdf` in the MATLAB® command prompt. Each leg of the robot has torque-actuated revolute joints in the frontal hip, knee, and ankle. Each arm has two passive revolute joints in the frontal and sagittal shoulder. During the simulation, the model senses the contact forces, position and orientation of the torso, joint states, and forward position. The figure shows the Simscape Multibody model on different levels.





Contact Modeling

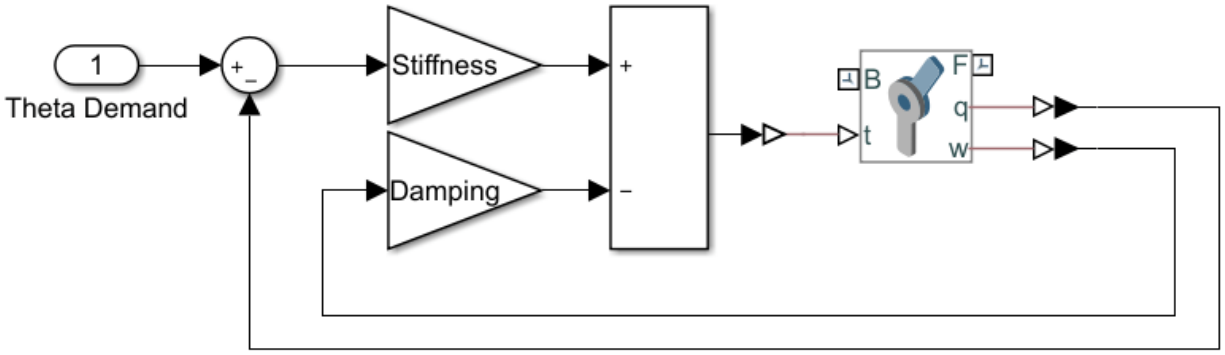
The model uses Spatial Contact Force (Simscape Multibody) blocks to simulate the contact between the feet and ground. To simplify the contact and speed up the simulation, red spheres are used to represent the bottoms of the robotic feet. For more details, see “Use Contact Proxies to Simulate Contact” (Simscape Multibody).

Joint Controller

The model uses a stiffness-based feedback controller to control each joint [1]. Model the joints as first-order systems with an associated stiffness (K) and damping (B), which you can set to make the joint behavior critically damped. The torque is applied when the setpoint θ_0 differs from the current joint position θ:

$T = B\dot{\theta} + K(\theta_0 - \theta)$.

You can vary the spring set-point θ_0 to elicit a feedback response to move the joint. The figure shows the Simulink model of the controller.



Humanoid Walker Training

The goal of this example is to train a humanoid robot to walk, and you can use various methods to train the robot. The example shows the genetic algorithm and reinforcement learning methods.

The Walking Objective Function

This example uses an objective function to evaluate different walking styles. The model gives a reward (r_t) at each timestep:

$$r_t = w_1 v_y + w_2 t_s - w_3 p - w_4 \Delta z - w_5 \Delta x$$

Here:

- v_y — Forward velocity (rewarded)
- p — Power consumption (penalized)
- Δz — Vertical displacement (penalized)
- Δx — Lateral displacement (penalized)
- $w_{1,\ldots,5}$: Weights, which represent the relative importance of each term in the reward function

Additionally, not falling over is rewarded.

Consequently, the total reward (R) for a walking trial is:

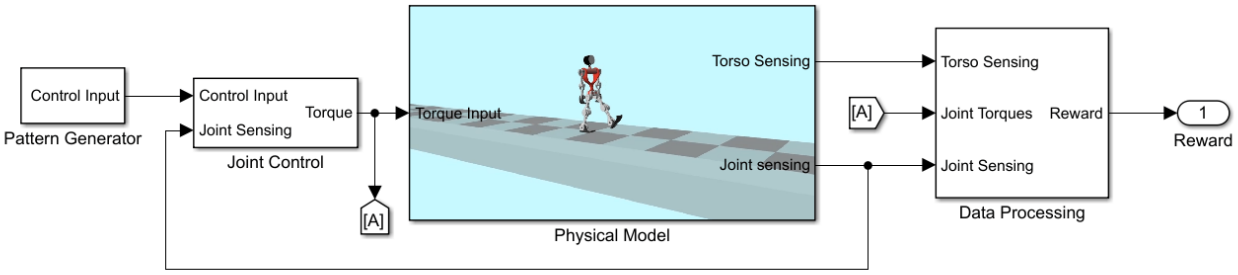
$$R = \sum_{t=0}^{T} r_t$$

Here T is the time at which the simulation terminates. You can change the reward weights in the `sm_humanoid_walker_rl_parameters` script. The simulation terminates when the simulation time is reached or the robot falls. Falling is defined as:

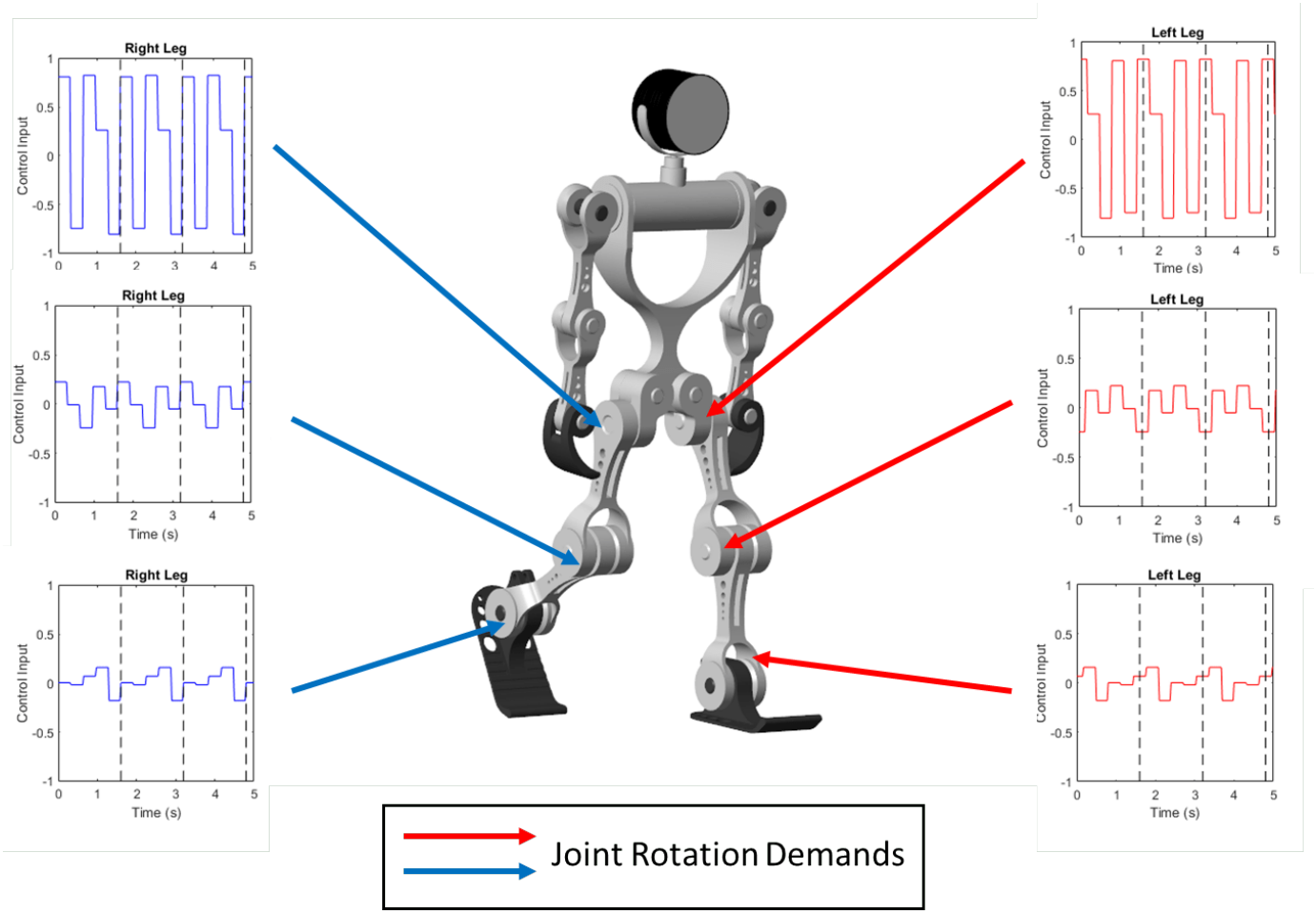
- The robot drops below 0.5 m.
- The robot moves laterally by more than 1 m.
- The robot torso rotates by more than 30 degrees.

Train with Genetic Algorithm

To optimize the walking of the robot, you can use a genetic algorithm. A genetic algorithm solves optimization problems based on a natural selection process that mimics biological evolution. Genetic algorithms are especially suited to problems when the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear. For more information, see `ga` (Global Optimization Toolbox).



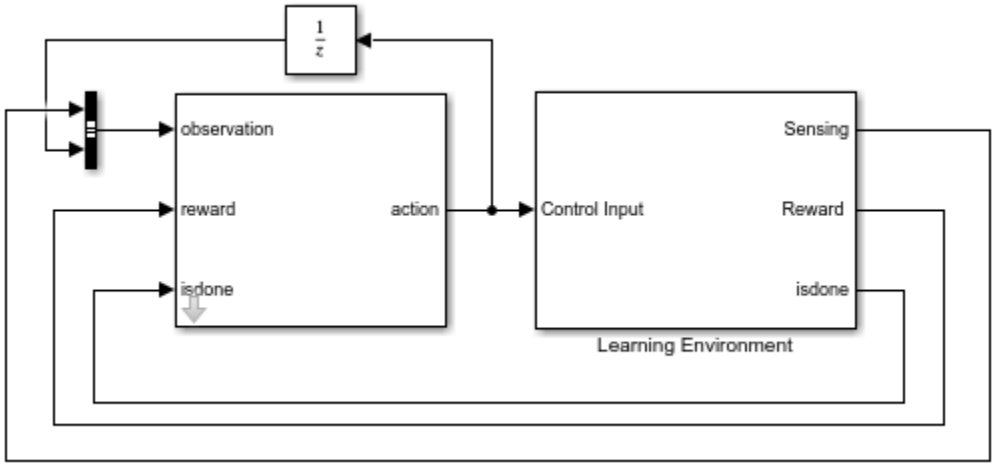
The model sets the angular demand for each joint to a repeating pattern that is analogous to the central pattern generators seen in nature [2]. The repeating pattern yields an open-loop controller. The periodicity of the signals is the gait period, which is the time taken to complete one full step. During each gait period, the signal switches between different angular demand values. Ideally, the humanoid robot walks symmetrically, and the control pattern for each joint in the right leg is transmitted to the corresponding joint in the left leg, with a delay of half a gait period. The pattern generator aims to determine the optimal control pattern for each joint and to maximize the walking objective function.



To train the robot with a genetic algorithm, open the `sm_humanoid_walker_ga_train` script. By default, this example uses a pretrained humanoid walker. To train the humanoid walker, set `trainWalker` to `true`.

Train with Reinforcement Learning

Alternatively, you can also train the robot using a deep deterministic policy gradient (DDPG) reinforcement learning agent. A DDPG agent is an actor-critic reinforcement learning agent that computes an optimal policy that maximizes the long-term reward. DDPG agents can be used in systems with continuous actions and states. For details about DDPG agents, see `rlDDPGAgent` (Reinforcement Learning Toolbox).



To train the robot with reinforcement learning, open the `sm_humanoid_walker_rl_train` script. By default, this example uses a pretrained humanoid walker. To train the humanoid walker, set `trainWalker` to `true`.

References

[1] Kalveram, Karl T., Thomas Schinauer, Steffen Beirle, Stefanie Richter, and Petra Jansen-Osmann. “Threading Neural Feedforward into a Mechanical Spring: How Biology Exploits Physics in Limb Control.” *Biological Cybernetics* 92, no. 4 (April 2005): 229–40. https://doi.org/10.1007/s00422-005-0542-6.

[2] Jiang Shan, Cheng Junshi, and Chen Jiapin. “Design of Central Pattern Generator for Humanoid Robot Walking Based on Multi-Objective GA.” In *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS 2000) (Cat. No.00CH37113), 3: 1930–35. Takamatsu, Japan: IEEE, 2000. https://doi.org/10.1109/IROS.2000.895253.

See Also

Point | Point Cloud | Spatial Contact Force

More About

- “Deep Learning Toolbox”
- “Global Optimization Toolbox”
- “Import a URDF Humanoid Model” (Simscape Multibody)
- “Modeling Contact Force Between Two Solids” (Simscape Multibody)
- “Reinforcement Learning Toolbox”
- “Use Contact Proxies to Simulate Contact” (Simscape Multibody)

Train DDPG Agent for Adaptive Cruise Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for adaptive cruise control (ACC) in Simulink®. For more information on DDPG agents, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

Simulink Model

The reinforcement learning environment for this example is the simple longitudinal dynamics for an ego car and lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration and braking. This example uses the same vehicle model as the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox) example.

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 25;   % initial velocity for lead car (m/s)
x0_ego = 10;    % initial position for ego car (m)
v0_ego = 20;    % initial velocity for ego car (m/s)
```

Specify standstill default spacing (m), time gap (s) and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 30;
```

To simulate the physical limitations of the vehicle dynamics, constraint the acceleration to the range `[-3,2]` m/s^2.

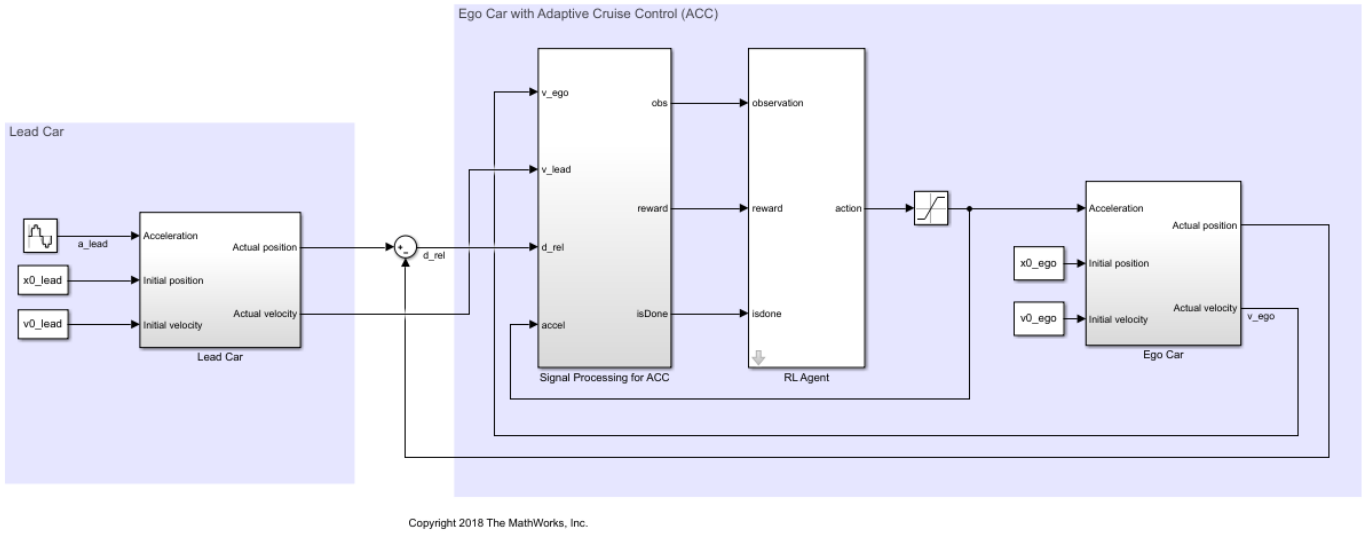
```
amin_ego = -3;
amax_ego = 2;
```

Define the sample time `Ts` and simulation duration `Tf` in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = 'rlACCMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```



For this model:

- The acceleration action signal from the agent to the environment is from –3 to 2 m/s^2.
- The reference velocity for the ego car V_{ref} is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity V; that is, $t_{gap}*V + D_{default}$. The safe distance determines the reference tracking velocity for the ego car.
- The observations from the environment are the velocity error $e = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity V.
- The simulation is terminated when longitudinal velocity of the ego car is less than 0, or the relative distance between the lead car and ego car becomes less than 0.
- The reward r_t, provided at every time step t, is

$$r_t = -(0.1e_t^2 + u_{t-1}^2) + M_t$$

where u_{t-1} is the control input from the previous time step. The logical value $M_t = 1$ if velocity error $e_t^2 <= 0.25$; otherwise, $M_t = 0$.

Create Environment Interface

Create a reinforcement learning environment interface for the model.

Create the observation specification.

```
observationInfo = rlNumericSpec([3 1],'LowerLimit',-inf*ones(3,1),'UpperLimit',inf*ones(3,1));
observationInfo.Name = 'observations';
observationInfo.Description = 'information on velocity error and ego velocity';
```

Create the action specification.

```
actionInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
actionInfo.Name = 'acceleration';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial condition for the position of the lead car, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng('default')
```

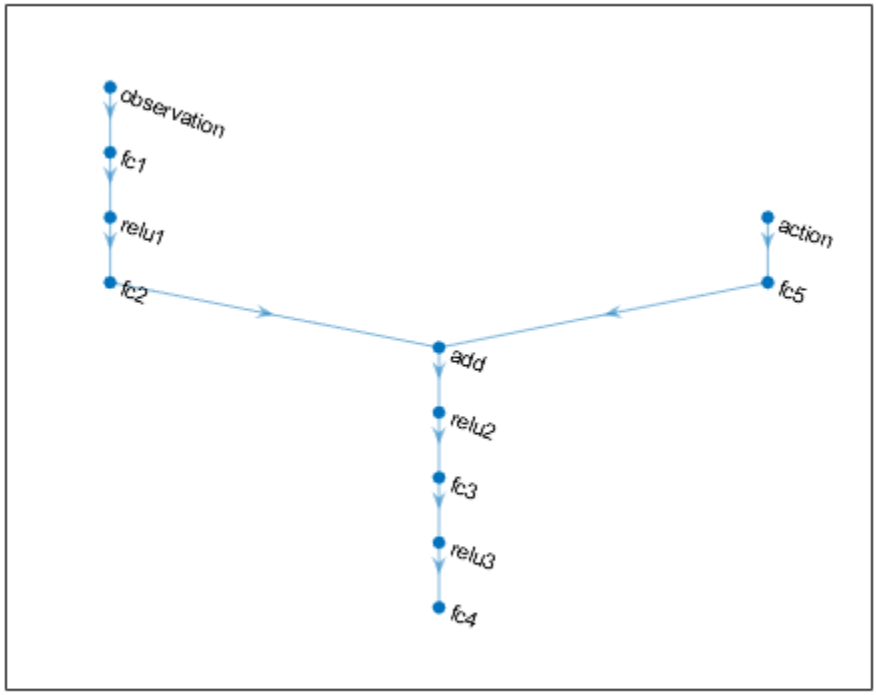
Create DDPG agent

A DDPG agent approximates the long-term reward given observations and actions using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the state and action, and one output. For more information on creating a neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
L = 48; % number of neurons
statePath = [
    featureInputLayer(3,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];

actionPath = [
    featureInputLayer(1,'Normalization','none','Name','action')
    fullyConnectedLayer(L, 'Name', 'fc5')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork, actionPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

View the critic network configuration.

```
plot(criticNetwork)
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1,'L2RegularizationF
```

Create the critic representation using the specified neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation` (Reinforcement Learning Toolbox).

```
actorNetwork = [
    featureInputLayer(3,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
```

```
    fullyConnectedLayer(1,'Name','fc4')
    tanhLayer('Name','tanh1')
    scalingLayer('Name','ActorScaling1','Scale',2.5,'Bias',-0.5)];

actorOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFa
actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'ActorScaling1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',64);
agentOptions.NoiseOptions.Variance = 0.6;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent` (Reinforcement Learning Toolbox).

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

- Run each training episode for at most `5000` episodes, with each episode lasting at most 600 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the agent receives an episode reward greater than 260.

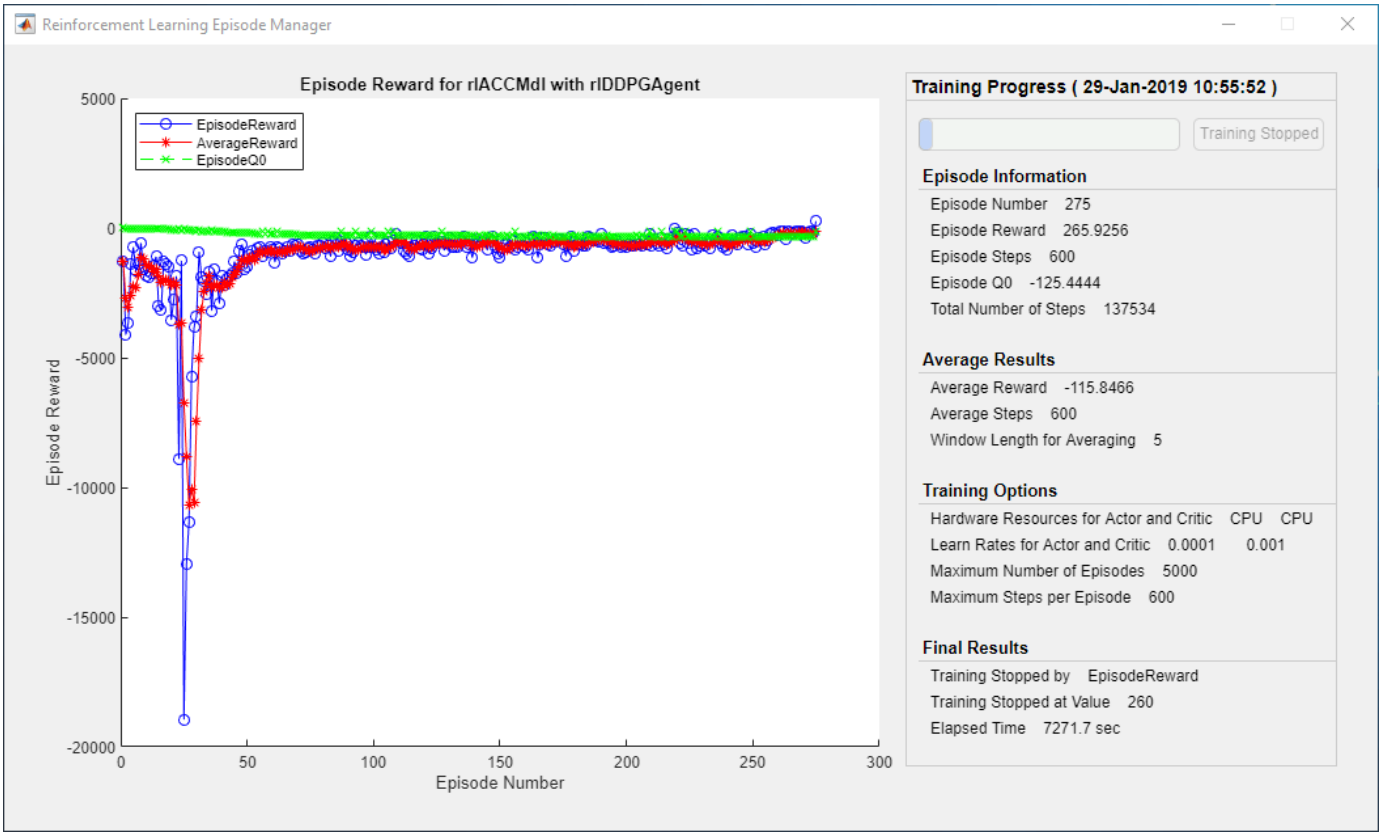
For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',260);
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
```

```
    % Load a pretrained agent for the example.
    load('SimulinkACCDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

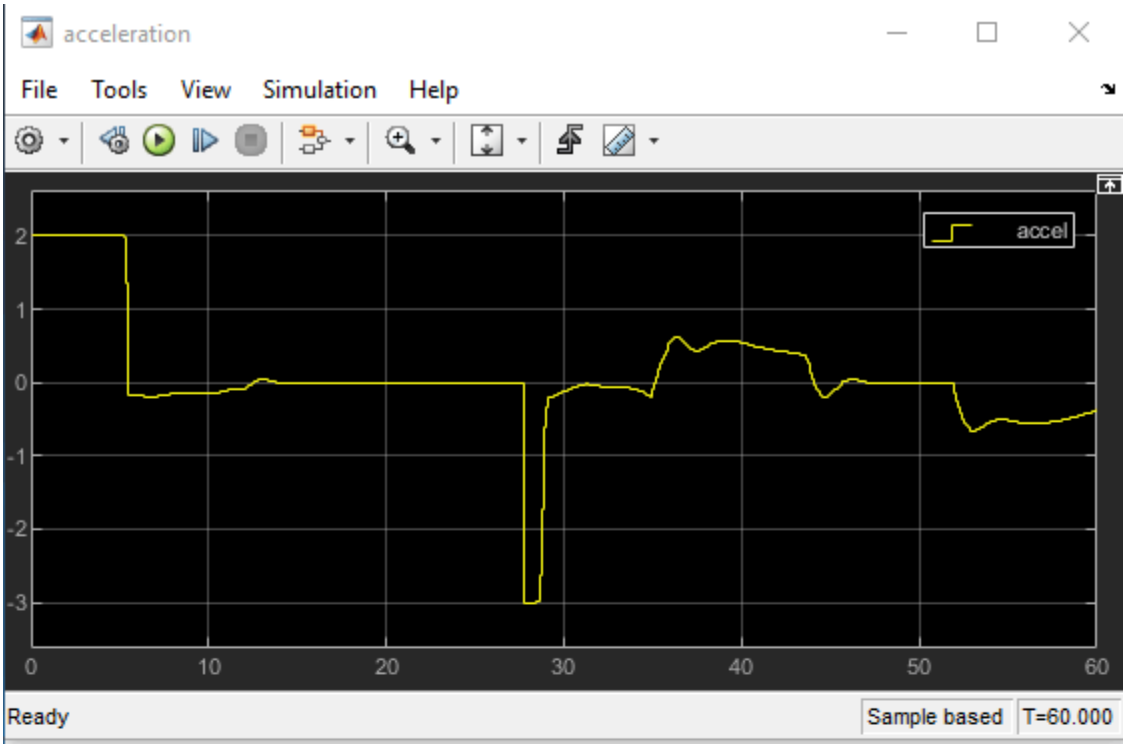
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

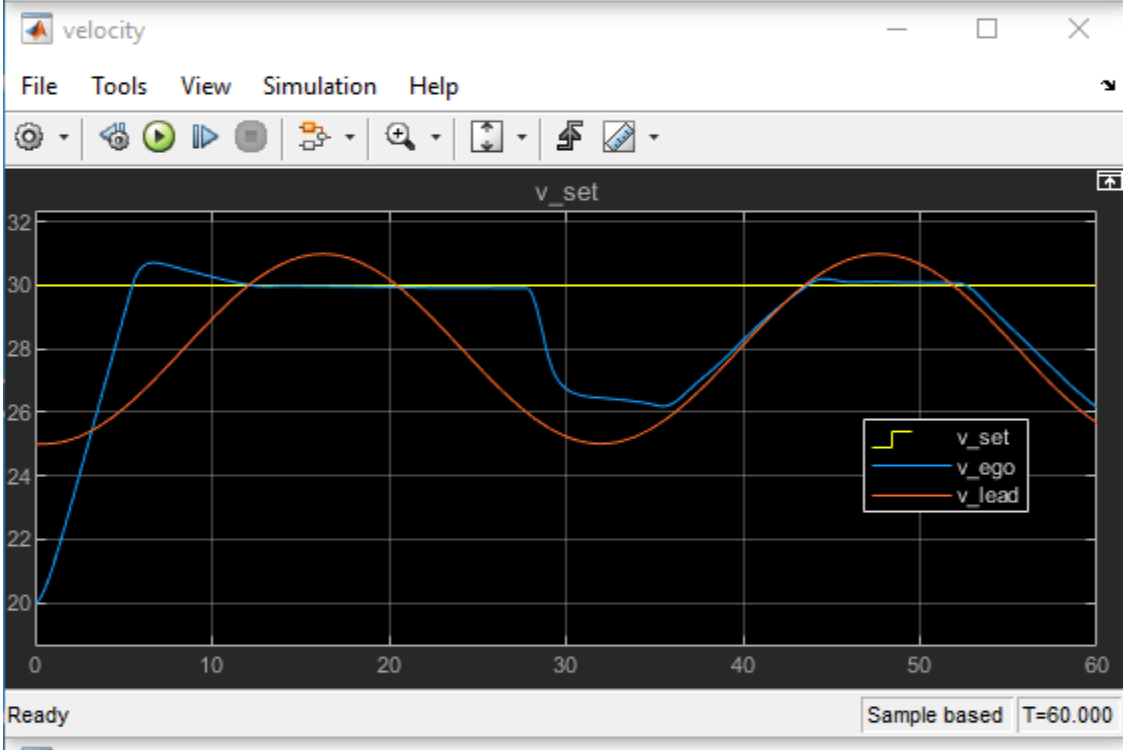
```
x0_lead = 80;
sim(mdl)
```

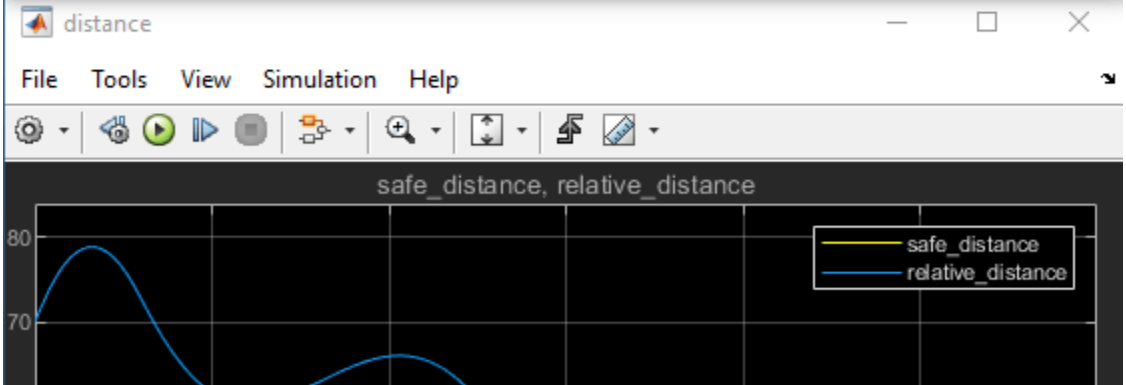
The following plots show the simulation results when lead car is 70 (m) ahead of the ego car.

- In the first 28 seconds, the relative distance is greater than the safe distance (bottom plot), so the ego car tracks set velocity (middle plot). To speed up and reach the set velocity, acceleration is positive (top plot).
- From 28 to 60 seconds, the relative distance is less than the safe distance (bottom plot), so the ego car tracks the minimum of the lead velocity and set velocity. From 28 to 36 seconds, the lead

velocity is less than the set velocity (middle plot). To slow down and track the lead car velocity, acceleration is negative (top plot). From 36 to 60 seconds, the ego car adjusts its acceleration to track the reference velocity closely (middle plot). Within this time interval, the ego car tracks the set velocity from 43 to 52 seconds and tracks lead velocity from 36 to 43 seconds and 52 to 60 seconds.


acceleration
File Tools View Simulation Help
accel
2
1
0
-1
-2
-3
0 10 20 30 40 50 60
Ready
Sample based
T=60.000


velocity
File Tools View Simulation Help
v_set
32
30
28
26
24
22
20
0 10 20 30 40 50 60
v_set
v_ego
v_lead
Ready
Sample based
T=60.000


distance
File Tools View Simulation Help
safe_distance, relative_distance
80
70
safe_distance
relative_distance

Close the Simulink model.

```
bdclose(mdl)
```

Reset Function

```
function in = localResetFcn(in)
% Reset the initial position of the lead car.
in = setVariable(in,'x0_lead',40+randi(60,1,1));
end
```

See Also

`train`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Train DQN Agent for Lane Keeping Assist Using Parallel Computing

This example shows how to train a deep Q-learning network (DQN) agent for lane keeping assist (LKA) in Simulink® using parallel training. For an example that shows how to train the agent without using parallel training, see “Train DQN Agent for Lane Keeping Assist” (Reinforcement Learning Toolbox).

For more information on DQN agents, see “Deep Q-Network Agents” (Reinforcement Learning Toolbox). For an example that trains a DQN agent in MATLAB®, see “Train DQN Agent to Balance Cart-Pole System” (Reinforcement Learning Toolbox).

DQN Parallel Training Overview

In a DQN agent, each worker generates new experiences from its copy of the agent and the environment. After every **N** steps, the worker sends experiences to the client agent (the agent associated with the MATLAB® process which starts the training). The client agent updates its parameters as follows.

- For asynchronous training, the client agent learns from received experiences without waiting for all workers to send experiences, and sends the updated parameters back to the worker that provided the experiences. Then, the worker continues to generate experiences from its environment using the updated parameters.
- For synchronous training, the client agent waits to receive experiences from all of the workers and learns from these experiences. The client then sends updated parameters to all the workers at the same time. Then, all workers continue to generate experiences using the updated parameters.

For more information about synchronous versus asynchronous parallelization, see “Train Agents Using Parallel Computing and GPUs” (Reinforcement Learning Toolbox).

Simulink Model for Ego Car

The reinforcement learning environment for this example is a simple bicycle model for ego vehicle dynamics. The training goal is to keep the ego vehicle traveling along the centerline of the lanes by adjusting the front steering angle. This example uses the same vehicle model as “Train DQN Agent for Lane Keeping Assist” (Reinforcement Learning Toolbox).

```
m = 1575;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.2;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
Vx = 15;    % longitudinal velocity (m/s)
```

Define the sample time `Ts` and simulation duration `T` in seconds.

```
Ts = 0.1;
T = 15;
```

The output of the LKA system is the front steering angle of the ego car. To simulate the physical steering limits of the ego car, constrain the steering angle to the range `[–0.5,0.5]` rad.

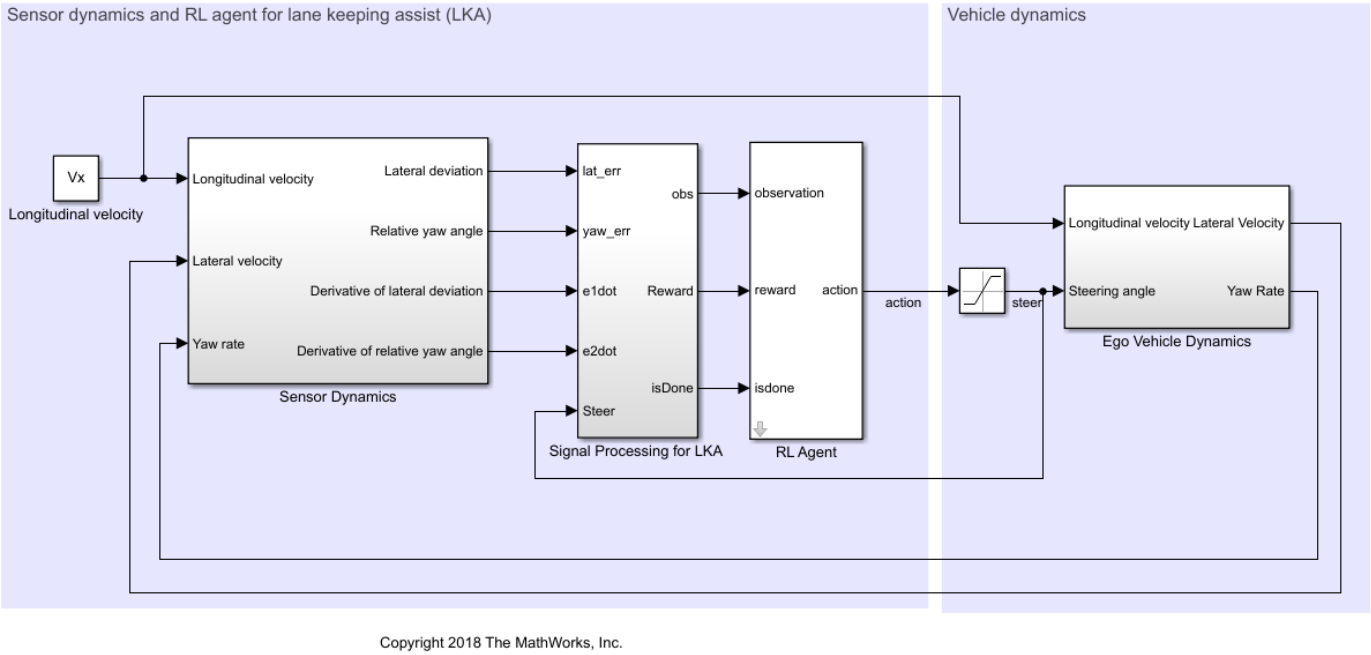
```
u_min = -0.5;
u_max = 0.5;
```

The curvature of the road is defined by a constant 0.001 (m^{-1}). The initial value for the lateral deviation is `0.2` m and the initial value for the relative yaw angle is `–0.1` rad.

```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Open the model.

```
mdl = 'rlLKAMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```



For this model:

- The steering-angle action signal from the agent to the environment is from –15 degrees to 15 degrees.
- The observations from the environment are the lateral deviation e_1, relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation is terminated when the lateral deviation $|e_1| > 1$.
- The reward r_t, provided at every time step t, is

$$r_t = -(10e_1^2 + 5e_2^2 + 2u^2 + 5\dot{e}_1^2 + 5\dot{e}_2^2)$$

where u is the control input from the previous time step $t - 1$.

Create Environment Interface

Create a reinforcement learning environment interface for the ego vehicle.

Define the observation information.

```
observationInfo = rlNumericSpec([6 1],'LowerLimit',-inf*ones(6,1),'UpperLimit',inf*ones(6,1));
observationInfo.Name = 'observations';
observationInfo.Description = 'information on lateral deviation and relative yaw angle';
```

Define the action information.

```
actionInfo = rlFiniteSetSpec((-15:15)*pi/180);
actionInfo.Name = 'steering';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

The interface has a discrete action space where the agent can apply one of 31 possible steering angles from –15 degrees to 15 degrees. The observation is the six-dimensional vector containing lateral deviation, relative yaw angle, as well as their derivatives and integrals with respect to time.

To define the initial condition for the lateral deviation and relative yaw angle, specify an environment reset function using an anonymous function handle. `localResetFcn`, which is defined at the end of this example, randomizes the initial lateral deviation and relative yaw angle.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DQN Agent

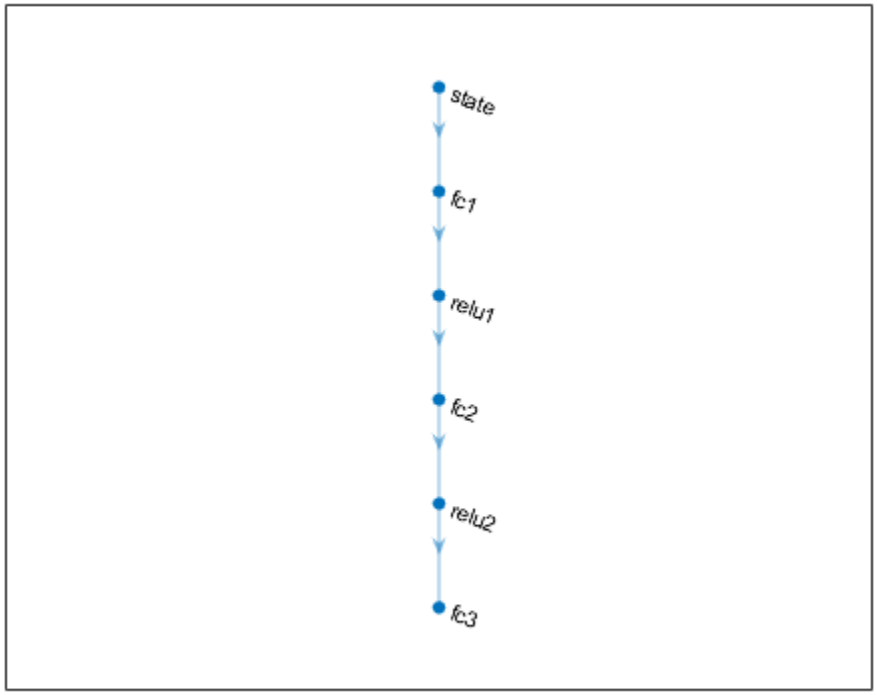
DQN agents can use multi-output Q-value critic approximators, which are generally more efficient. A multi-output approximator has observations as inputs and state-action values as outputs. Each output element represents the expected cumulative long-term reward for taking the corresponding discrete action from the state indicated by the observation inputs.

To create the critic, first create a deep neural network with one input (the six-dimensional observed state) and one output vector with 31 elements (evenly spaced steering angles from -15 to 15 degrees). For more information on creating a deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
nI = observationInfo.Dimension(1);  % number of inputs (6)
nL = 120;                           % number of neurons
nO = numel(actionInfo.Elements);    % number of outputs (31)

dnn = [
    featureInputLayer(nI,'Normalization','none','Name','state')
    fullyConnectedLayer(nL,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(nL,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(nO,'Name','fc3')];
```

View the network configuration.

```
figure
plot(layerGraph(dnn))
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationF
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(dnn,observationInfo,actionInfo,'Observation',{'state'},criticOpti
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions` (Reinforcement Learning Toolbox).

```
agentOpts = rlDQNAgentOptions(...
    'SampleTime',Ts,...
    'UseDoubleDQN',true,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',0.99,...
    'ExperienceBufferLength',1e6,...
    'MiniBatchSize',256);

agentOpts.EpsilonGreedyExploration.EpsilonDecay = 1e-4;
```

Then create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent` (Reinforcement Learning Toolbox).

```
agent = rlDQNAgent(critic,agentOpts);
```

Training Options

To train the agent, first specify the training options. For this example, use the following options.

- Run each training for at most `10000` episodes, with each episode lasting at most `ceil(T/Ts)` time steps.
- Display the training progress in the Episode Manager dialog box only (set the `Plots` and `Verbose` options accordingly).
- Stop training when the episode reward reaches `-1`.
- Save a copy of the agent for each episode where the cumulative reward is greater than 100.

For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 10000;
maxsteps = ceil(T/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue', -1,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',100);
```

Parallel Training Options

To train the agent in parallel, specify the following training options.

- Set the `UseParallel` option to `true`.
- Train agent in parallel asynchronously by setting the `ParallelizationOptions.Mode` option to `"async"`.
- After every 30 steps, each worker sends experiences to the client.
- DQN agent requires workers to send "`experiences`" to the client.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "experiences";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
```

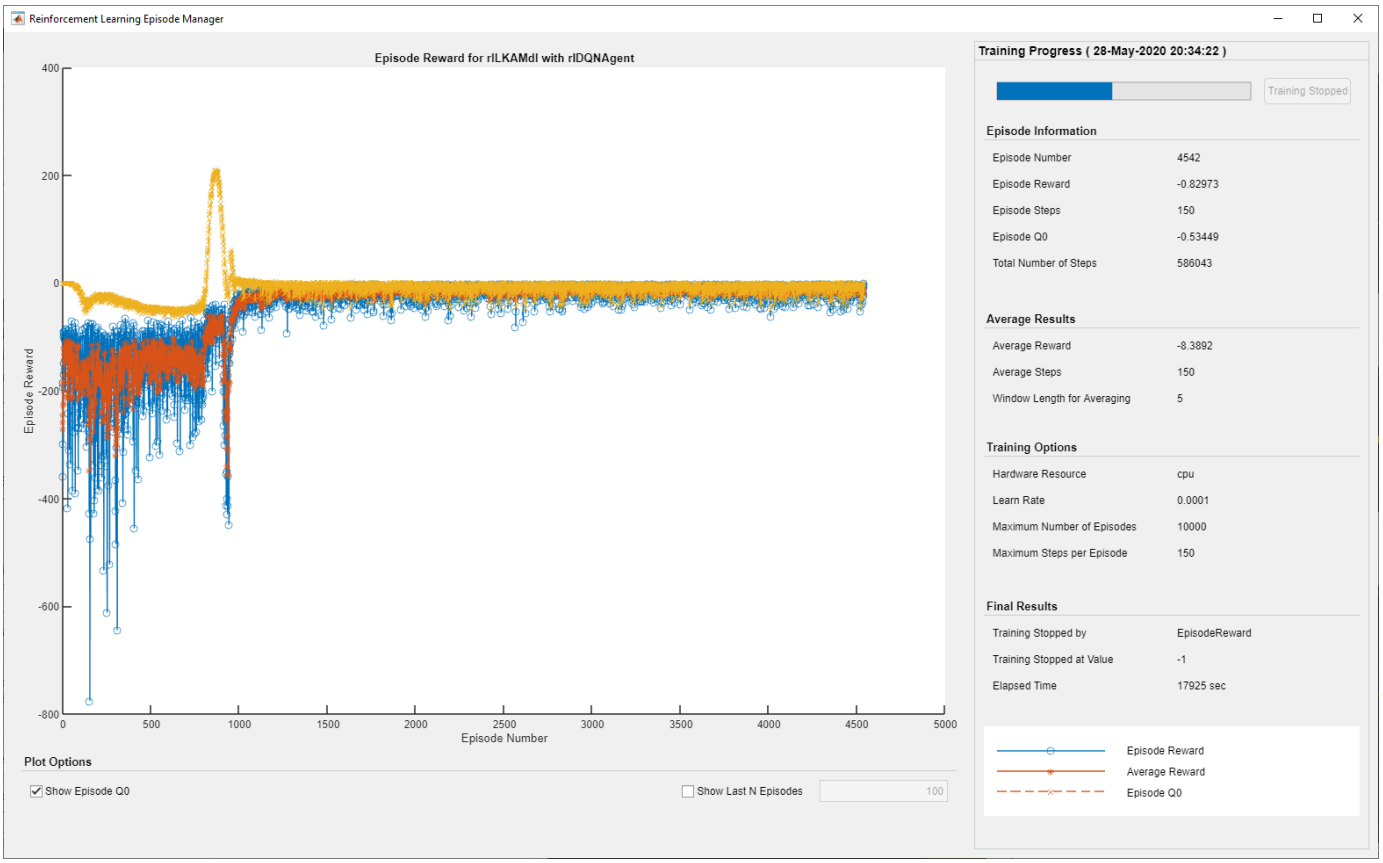
For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

Train Agent

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training the agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`. Due to randomness of the parallel training, you can expect different training results from the plot below. The plot shows the result of training with four workers.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
```

```
else
    % Load pretrained agent for the example.
    load('SimulinkLKADQNParallel.mat','agent')
end
```



Simulate DQN Agent

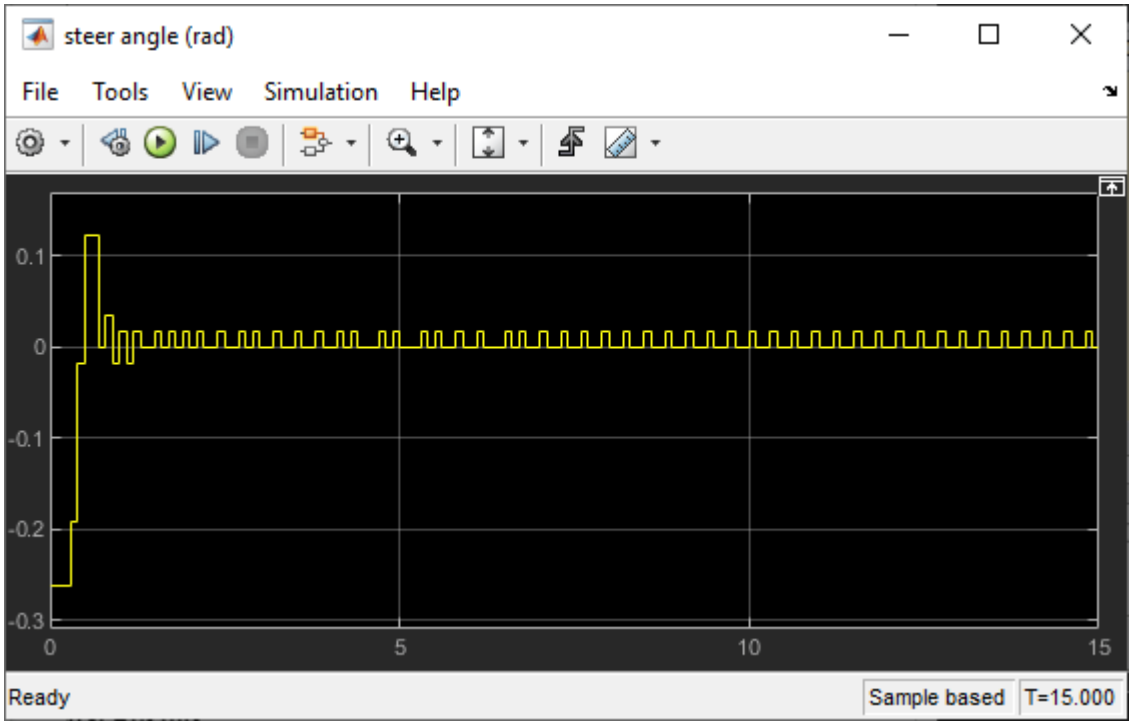
To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

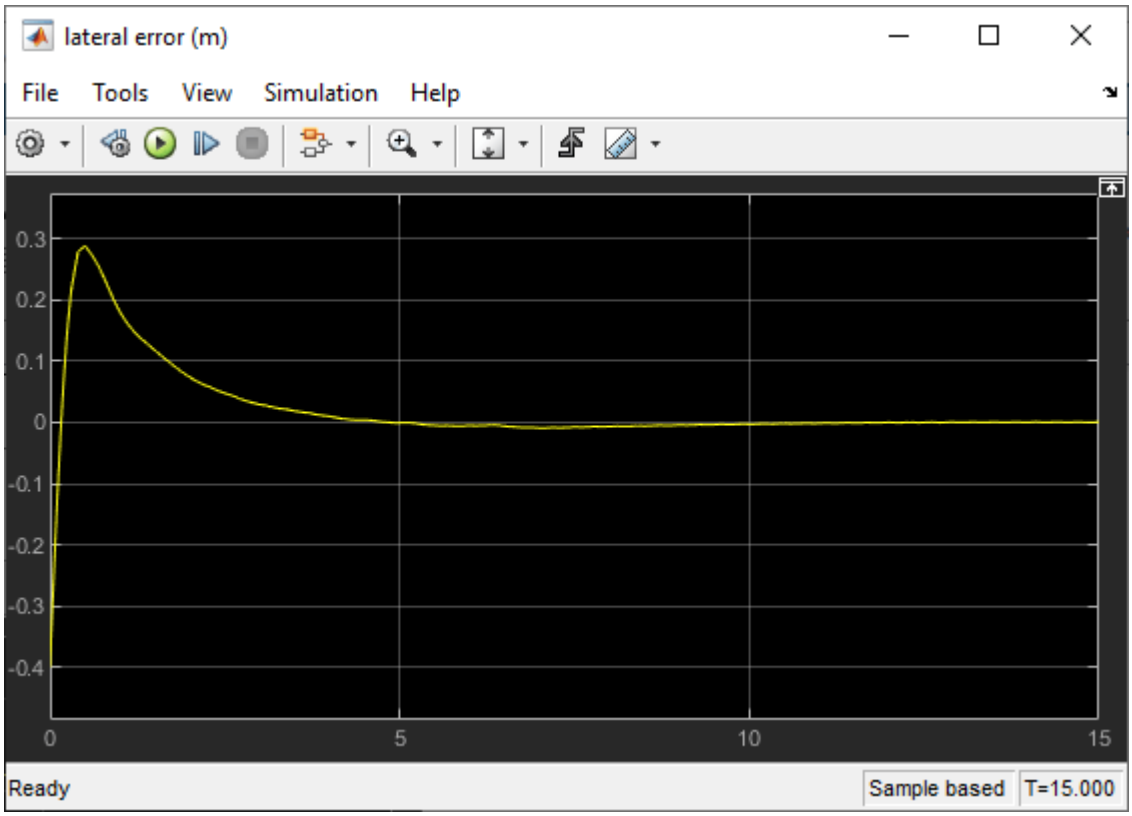
```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

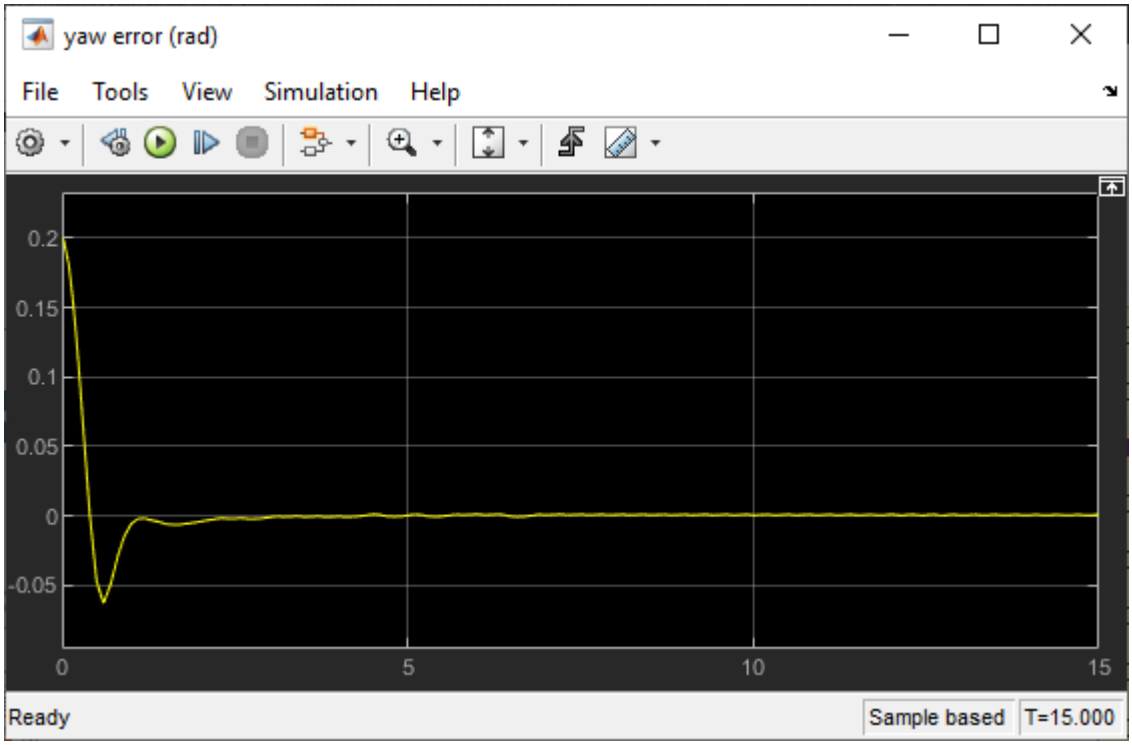
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl)
```

As shown below, the lateral error (middle plot) and relative yaw angle (bottom plot) are both driven to zero. The vehicle starts from off centerline (–0.4 m) and nonzero yaw angle error (0.2 rad). The LKA enables the ego car to travel along the centerline after 2.5 seconds. The steering angle (top plot) shows that the controller reaches steady state after 2 seconds.


steer angle (rad)
File
Tools
View
Simulation
Help
0.1
0
-0.1
-0.2
-0.3
0
5
10
15
Ready
Sample based
T=15.000


lateral error (m)
File
Tools
View
Simulation
Help
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
Ready
Sample based
T=15.000



Local Function

```
function in = localResetFcn(in)
% reset
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand)); % random value for lateral deviation
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand)); % random value for relative yaw angle
end
```

See Also

`train`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Train DDPG Agent for Path-Following Control

This example shows how to train a deep deterministic policy gradient (DDPG) agent for path-following control (PFC) in Simulink®. For more information on DDPG agents, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

Simulink Model

The reinforcement learning environment for this example is a simple bicycle model for the ego car and a simple longitudinal model for the lead car. The training goal is to make the ego car travel at a set velocity while maintaining a safe distance from lead car by controlling longitudinal acceleration and braking, and also while keeping the ego car travelling along the centerline of its lane by controlling the front steering angle. For more information on PFC, see Path Following Control System (Model Predictive Control Toolbox). The ego car dynamics are specified by the following parameters.

```
m = 1600;   % total vehicle mass (kg)
Iz = 2875;  % yaw moment of inertia (mNs^2)
lf = 1.4;   % longitudinal distance from center of gravity to front tires (m)
lr = 1.6;   % longitudinal distance from center of gravity to rear tires (m)
Cf = 19000; % cornering stiffness of front tires (N/rad)
Cr = 33000; % cornering stiffness of rear tires (N/rad)
tau = 0.5;  % longitudinal time constant
```

Specify the initial position and velocity for the two vehicles.

```
x0_lead = 50;   % initial position for lead car (m)
v0_lead = 24;   % initial velocity for lead car (m/s)
x0_ego = 10;    % initial position for ego car (m)
v0_ego = 18;    % initial velocity for ego car (m/s)
```

Specify the standstill default spacing (m), time gap (s), and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 28;
```

To simulate the physical limitations of the vehicle dynamics, constrain the acceleration to the range `[-3,2]` (m/s^2), and steering angle is constrained to be `[-0.5,0.5]` (rad).

```
amin_ego = -3;
amax_ego = 2;
umin_ego = -0.5;
umax_ego = 0.5;
```

The curvature of the road is defined by a constant 0.001 (m^{-1}). The initial value for lateral deviation is 0.2 m and the initial value for the relative yaw angle is –0.1 rad.

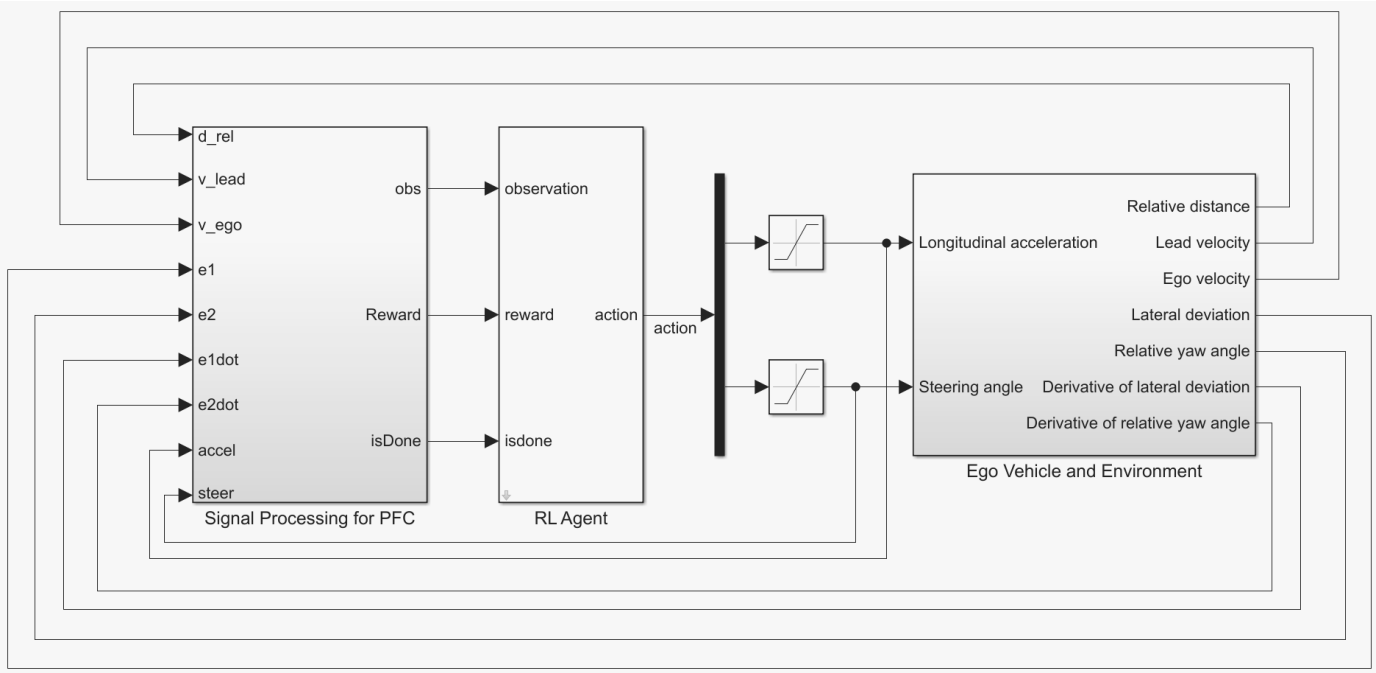
```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

Define the sample time `Ts` and simulation duration `Tf` in seconds.

```
Ts = 0.1;
Tf = 60;
```

Open the model.

```
mdl = 'rlPFCMdl';
open_system(mdl)
agentblk = [mdl '/RL Agent'];
```



For this model:

- The action signal consists of acceleration and steering angle actions. The acceleration action signal takes value between –3 and 2 (m/s^2). The steering action signal takes a value between –15 degrees (–0.2618 rad) and 15 degrees (0.2618 rad).
- The reference velocity for the ego car V_{ref} is defined as follows. If the relative distance is less than the safe distance, the ego car tracks the minimum of the lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from the lead car. If the relative distance is greater than the safe distance, the ego car tracks the driver-set velocity. In this example, the safe distance is defined as a linear function of the ego car longitudinal velocity V, that is, $t_{gap} * V + D_{default}$. The safe distance determines the tracking velocity for the ego car.
- The observations from the environment contain the longitudinal measurements: the velocity error $e_V = V_{ref} - V_{ego}$, its integral $\int e$, and the ego car longitudinal velocity V. In addition, the observations contain the lateral measurements: the lateral deviation e_1, relative yaw angle e_2, their derivatives $\dot{e}_1$ and $\dot{e}_2$, and their integrals $\int e_1$ and $\int e_2$.
- The simulation terminates when the lateral deviation $|e_1| > 1$, when the longitudinal velocity $V_{ego} < 0.5$, or when the relative distance between the lead car and ego car $D_{rel} < 0$.
- The reward r_t, provided at every time step t, is

$$r_t = -(100e_1^2 + 500u_{t-1}^2 + 10e_V^2 + 100a_{t-1}^2) \times 1e^{-3} - 10F_t + 2H_t + M_t$$

where u_{t-1} is the steering input from the previous time step $t-1$, a_{t-1} is the acceleration input from the previous time step. The three logical values are as follows.

- F_t = 1if simulation is terminated, otherwise $F_t = 0$
- $H_t = 1$ if lateral error $e_1^2 < 0.01$, otherwise $H_t = 0$
- $M_t = 1$ if velocity error $e_V^2 < 1$, otherwise $M_t = 0$

The three logical terms in the reward encourage the agent to make both lateral error and velocity error small, and in the meantime, penalize the agent if the simulation is terminated early.

Create Environment Interface

Create an environment interface for the Simulink model.

Create the observation specification.

```
observationInfo = rlNumericSpec([9 1],'LowerLimit',-inf*ones(9,1),'UpperLimit',inf*ones(9,1));
observationInfo.Name = 'observations';
```

Create the action specification.

```
actionInfo = rlNumericSpec([2 1],'LowerLimit',[-3;-0.2618],'UpperLimit',[2;0.2618]);
actionInfo.Name = 'accel;steer';
```

Create the environment interface.

```
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial conditions, specify an environment reset function using an anonymous function handle. The reset function `localResetFcn`, which is defined at the end of the example, randomizes the initial position of the lead car, the lateral deviation, and the relative yaw angle.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

Create DDPG Agent

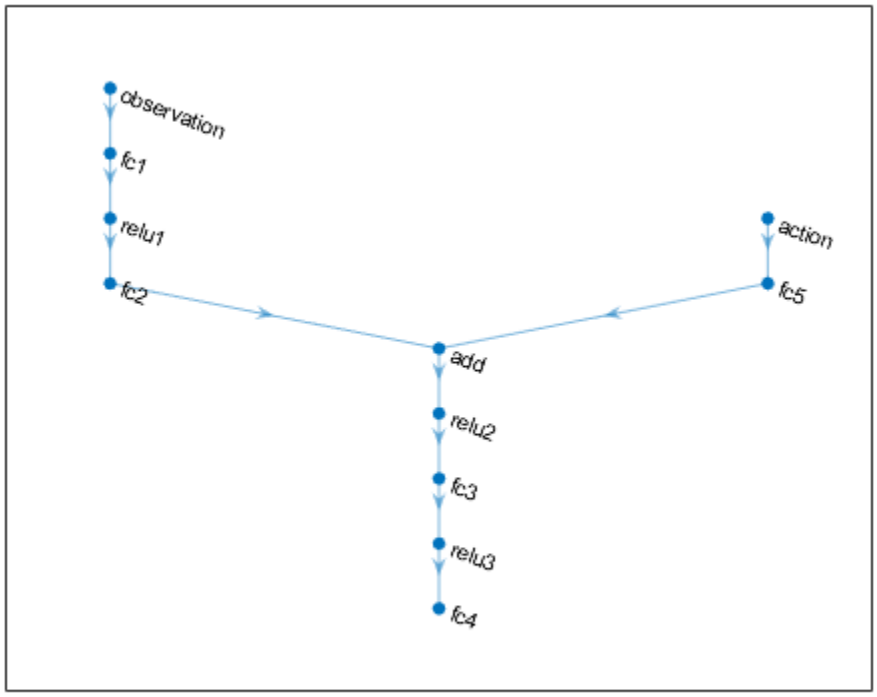
A DDPG agent approximates the long-term reward given observations and actions by using a critic value function representation. To create the critic, first create a deep neural network with two inputs, the state and action, and one output. For more information on creating a deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
L = 100; % number of neurons
statePath = [
    featureInputLayer(9,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];

actionPath = [
```

```
    featureInputLayer(2,'Normalization','none','Name','action')
    fullyConnectedLayer(L,'Name','fc5')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork,actionPath);

criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

View the critic network configuration.

```
figure
plot(criticNetwork)
```



Specify options for the critic representation using `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1,'L2RegularizationF
```

Create the critic representation using the specified deep neural network and options. You must also specify the action and observation info for the critic, which you obtain from the environment interface. For more information, see `rlQValueRepresentation` (Reinforcement Learning Toolbox).

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
```

A DDPG agent decides which action to take given observations by using an actor representation. To create the actor, first create a deep neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation` (Reinforcement Learning Toolbox).

```
actorNetwork = [
    featureInputLayer(9,'Normalization','none','Name','observation')
    fullyConnectedLayer(L,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(L,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(L,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(2,'Name','fc4')
    tanhLayer('Name','tanh1')
    scalingLayer('Name','ActorScaling1','Scale',[2.5;0.2618],'Bias',[-0.5;0])];
actorOptions = rlRepresentationOptions('LearnRate',1e-4,'GradientThreshold',1,'L2RegularizationFa
actor = rlDeterministicActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'Action',{'ActorScaling1'},actorOptions);
```

To create the DDPG agent, first specify the DDPG agent options using `rlDDPGAgentOptions` (Reinforcement Learning Toolbox).

```
agentOptions = rlDDPGAgentOptions(...
    'SampleTime',Ts,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',64);
agentOptions.NoiseOptions.Variance = [0.6;0.1];
agentOptions.NoiseOptions.VarianceDecayRate = 1e-5;
```

Then, create the DDPG agent using the specified actor representation, critic representation, and agent options. For more information, see `rlDDPGAgent` (Reinforcement Learning Toolbox).

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

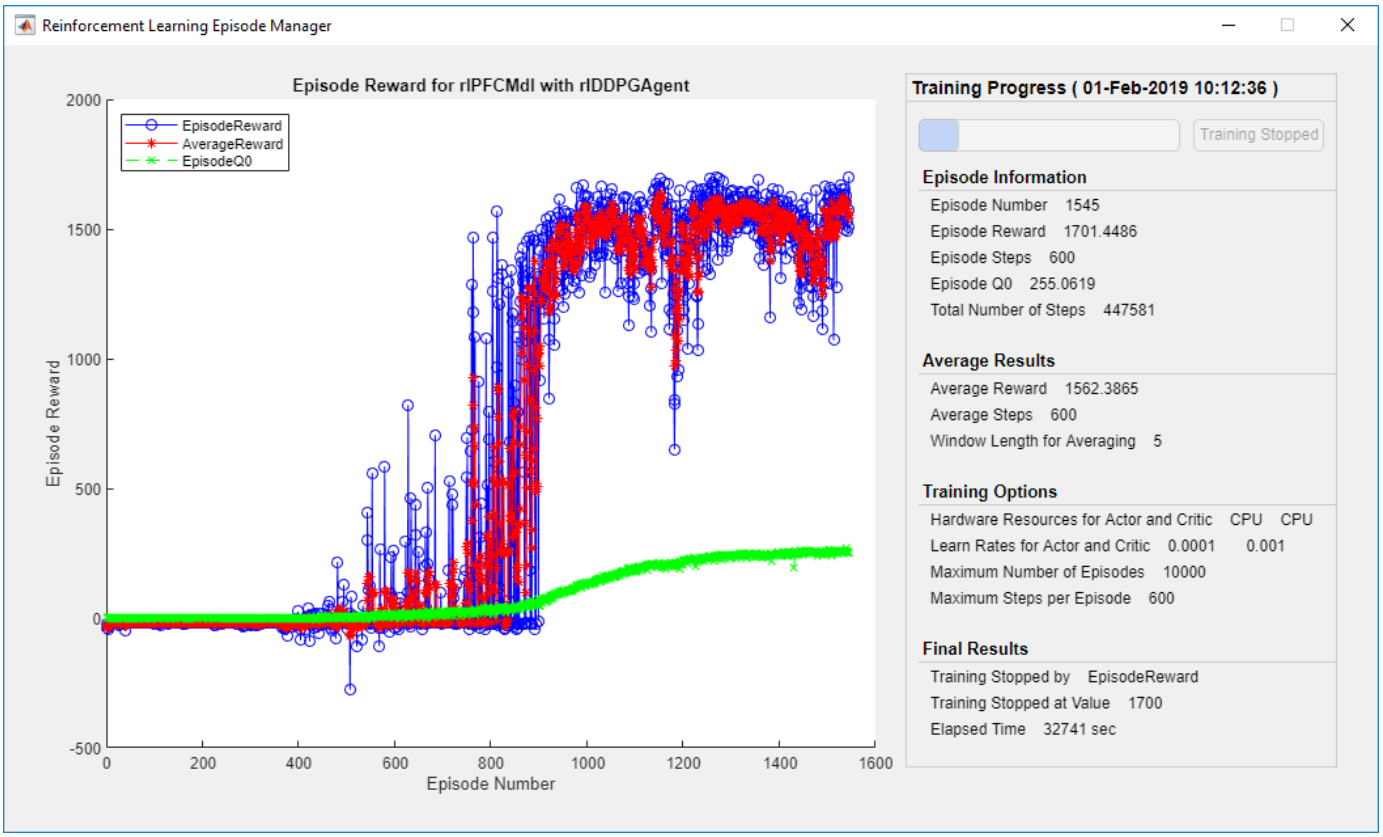
- Run each training episode for at most `10000` episodes, with each episode lasting at most `maxsteps` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Verbose` and `Plots` options).
- Stop training when the agent receives an cumulative episode reward greater than `1700`.

For more information, see `rlTrainingOptions` (Reinforcement Learning Toolbox).

```
maxepisodes = 1e4;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',1700);
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

```
doTraining = false;

if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOpts);
else
    % Load a pretrained agent for the example.
    load('SimulinkPFCDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, simulate the agent within the Simulink environment by uncommenting the following commands. For more information on agent simulation, see `rlSimulationOptions` (Reinforcement Learning Toolbox) and `sim` (Reinforcement Learning Toolbox).

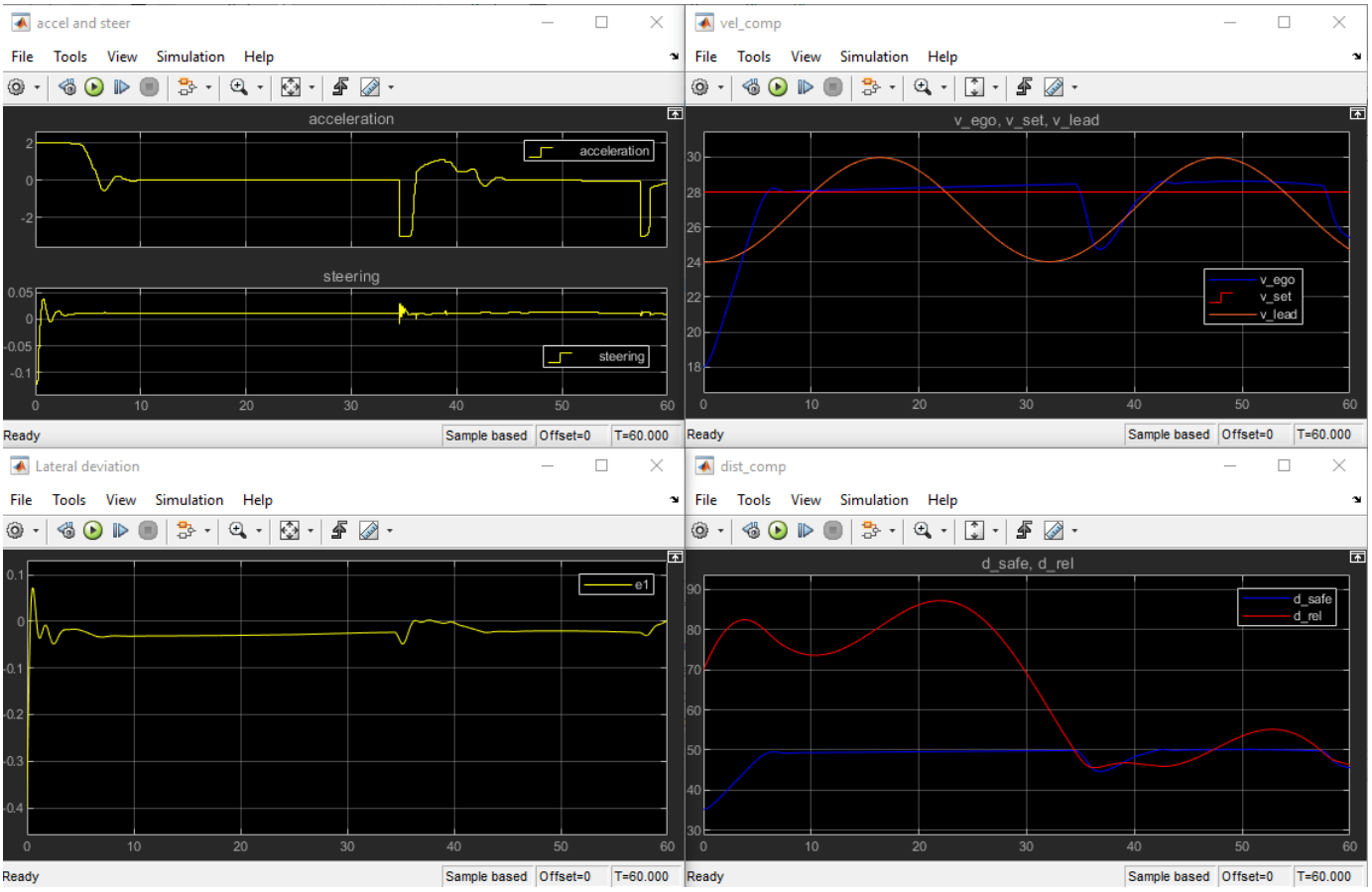
```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.1;
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when the lead car is 70 (m) ahead of the ego car.

- In the first 35 seconds, the relative distance is greater than the safe distance (bottom-right plot), so the ego car tracks the set velocity (top-right plot). To speed up and reach the set velocity, the acceleration is mostly nonnegative (top-left plot).
- From 35 to 42 seconds, the relative distance is mostly less than the safe distance (bottom-right plot), so the ego car tracks the minimum of the lead velocity and set velocity. Because the lead velocity is less than the set velocity (top-right plot), to track the lead velocity, the acceleration becomes nonzero (top-left plot).
- From 42 to 58 seconds, the ego car tracks the set velocity (top-right plot) and the acceleration remains zero (top-left plot).
- From 58 to 60 seconds, the relative distance becomes less than the safe distance (bottom-right plot), so the ego car slows down and tracks the lead velocity.
- The bottom-left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within 1 second. The lateral deviation remains less than 0.05 m.



Close the Simulink model.

```
bdclose(mdl)
```

Reset Function

```
function in = localResetFcn(in)
in = setVariable(in,'x0_lead',40+randi(60,1,1));    % random value for initial position of lead c
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand)); % random value for lateral deviation
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand)); % random value for relative yaw angle
end
```

See Also

`train`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Train PPO Agent for Automatic Parking Valet

This example demonstrates the design of a hybrid controller for an automatic search and parking task. The hybrid controller uses model predictive control (MPC) to follow a reference path in a parking lot and a trained reinforcement learning (RL) agent to perform the parking maneuver.

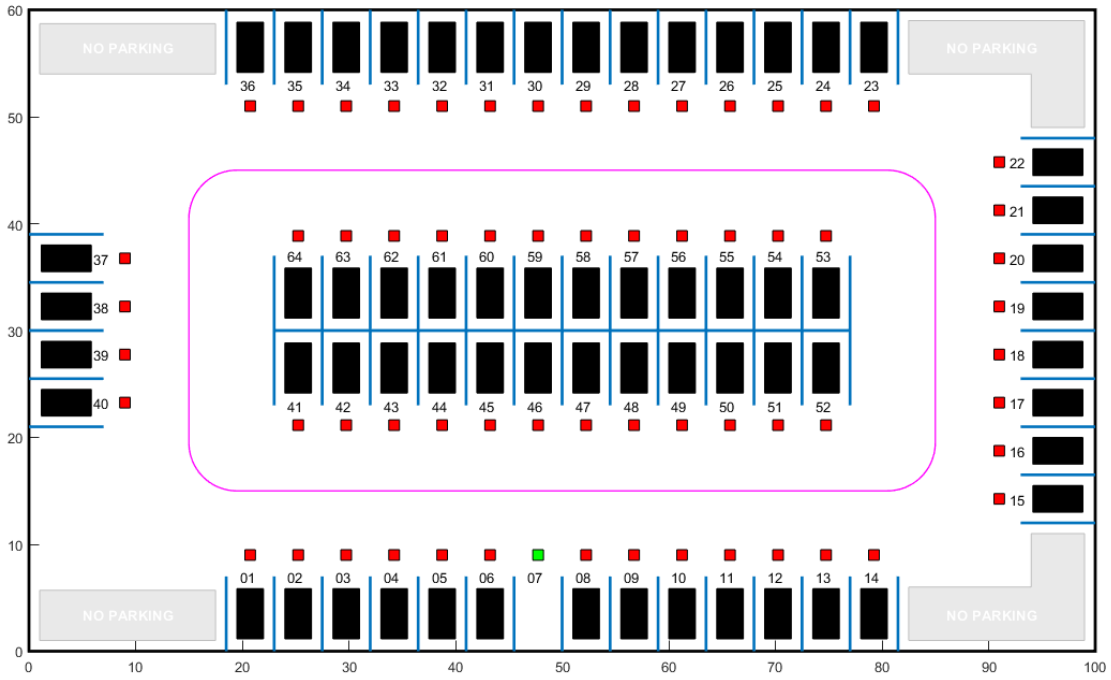
The automatic parking algorithm in this example executes a series of maneuvers while simultaneously sensing and avoiding obstacles in tight spaces. It switches between an adaptive MPC controller and an RL agent to complete the parking maneuver. The MPC controller moves the vehicle at a constant speed along a reference path while an algorithm searches for an empty parking spot. When a spot is found, the RL Agent takes over and executes a pretrained parking maneuver. Prior knowledge of the environment (the parking lot) including the locations of the empty spots and parked vehicles is available to the controllers.

Parking Lot

The parking lot is represented by the `ParkingLot` class, which stores information about the ego vehicle, empty parking spots, and static obstacles (parked cars). Each parking spot has a unique index number and an indicator light that is either green (free) or red (occupied). Parked vehicles are represented in black.

Create a `ParkingLot` object with a free spot at location 7.

```
freeSpotIdx = 7;
map = ParkingLot(freeSpotIdx);
```



Specify an initial pose (X_0, Y_0, θ_0) for the ego vehicle. The target pose is determined based on the first available free spot as the vehicle navigates the parking lot.

```
egoInitialPose = [20, 15, 0];
```

Compute the target pose for the vehicle using the `createTargetPose` function. The target pose corresponds to the location in `freeSpotIdx`.

```
egoTargetPose = createTargetPose(map,freeSpotIdx)
```

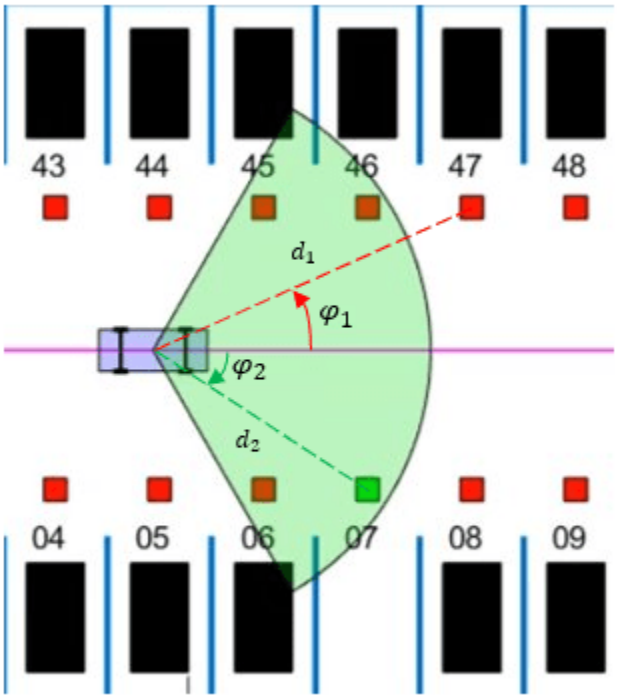
```
egoTargetPose = 1×3

   47.7500    4.9000   -1.5708
```

Sensor Modules

The parking algorithm uses camera and lidar sensors to gather information from the environment.

Camera

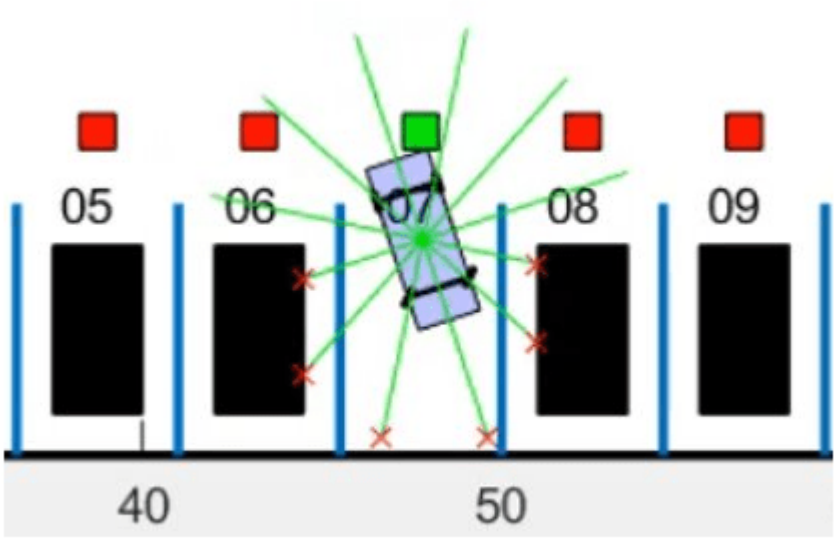
The field of view of a camera mounted on the ego vehicle is represented by the area shaded in green in the following figure. The camera has a field of view φ bounded by ±60 degrees and a maximum measurement depth $d_{\max}$ of 10 m.



As the ego vehicle moves forward, the camera module senses the parking spots that fall within the field of view and determines whether a spot is free or occupied. For simplicity, this action is implemented using geometrical relationships between the spot locations and the current vehicle pose. A parking spot is within the camera range if $d_i \le d_{\max}$ and $\varphi_{\min} \le \varphi_i \le \varphi_{\max}$, where d_i is the distance to the parking spot and φ_i is the angle to the parking spot.

Lidar

The reinforcement learning agent uses lidar sensor readings to determine the proximity of the ego vehicle to other vehicles in the environment. The lidar sensor in this example is also modeled using geometrical relationships. Lidar distances are measured along 12 line segments that radially emerge from the center of the ego vehicle. When a lidar line intersects an obstacle, it returns the distance of the obstacle from the vehicle. The maximum measurable lidar distance along any line segment is 6 m.



Auto Parking Valet Model

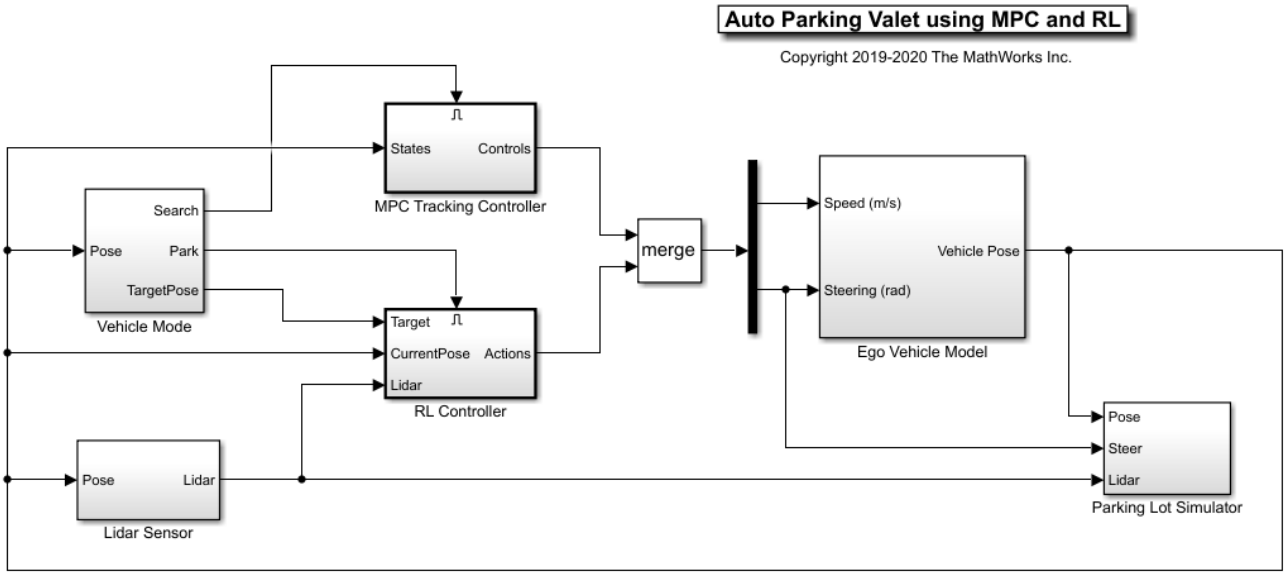
The parking valet model, including the controllers, ego vehicle, sensors, and parking lot, is implemented in Simulink®.

Load the auto parking valet parameters.

```
autoParkingValetParams
```

Open the Simulink model.

```
mdl = 'rlAutoParkingValet';
open_system(mdl)
```



The ego vehicle dynamics in this model are represented by a single-track bicycle model with two inputs: vehicle speed v (m/s) and steering angle δ (radians). The MPC and RL controllers are placed within Enabled Subsystem blocks that are activated by signals representing whether the vehicle has to search for an empty spot or execute a parking maneuver. The enable signals are determined by the Camera algorithm within the Vehicle Mode subsystem. Initially, the vehicle is in *search* mode and the MPC controller tracks the reference path. When a free spot is found, *park* mode is activated and the RL agent executes the parking maneuver.

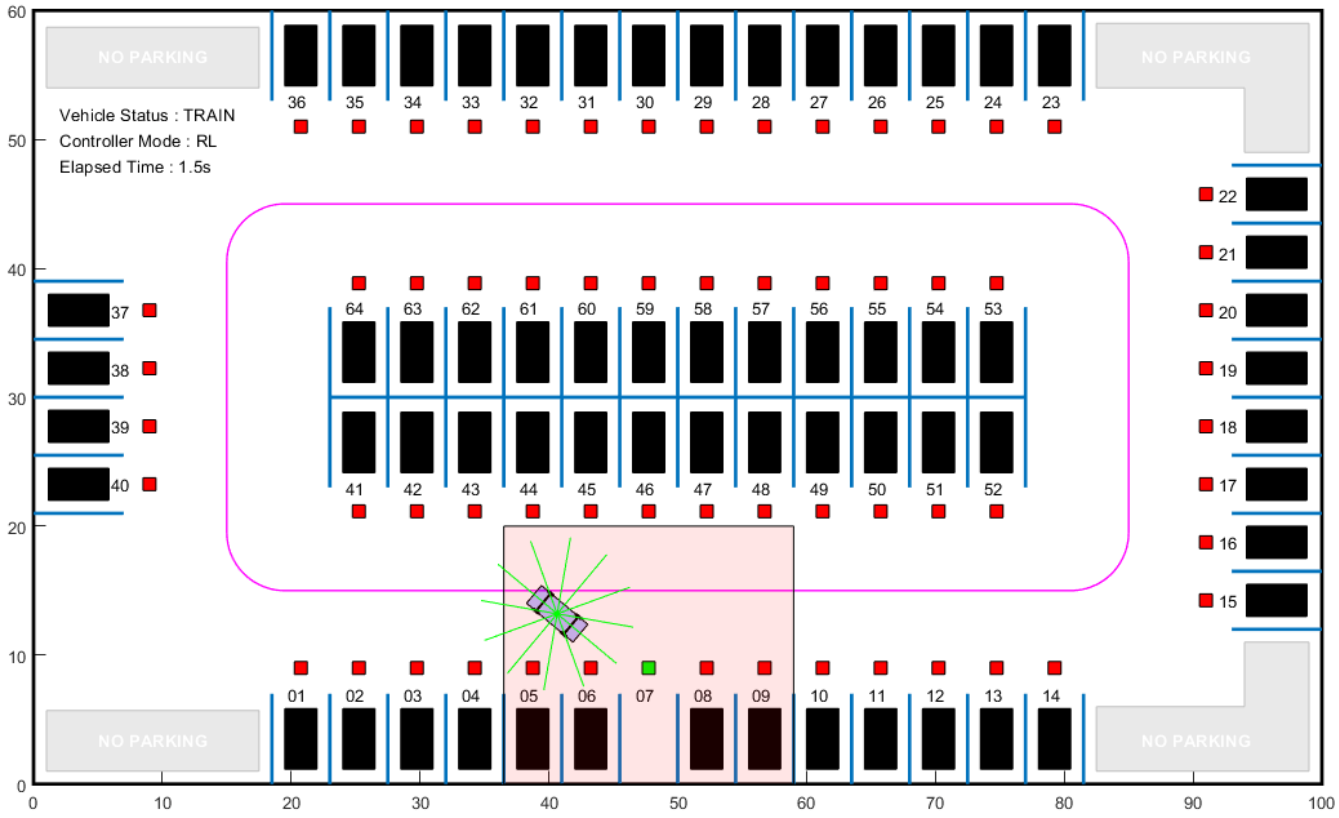
Adaptive Model Predictive Controller

Create the adaptive MPC controller object for reference trajectory tracking using the `createMPCForParking` script. For more information on adaptive MPC, see “Adaptive MPC” (Model Predictive Control Toolbox).

```
createMPCForParking
```

Reinforcement Learning Environment

The environment for training the RL agent is the region shaded in red in the following figure. Due to symmetry in the parking lot, training within this region is sufficient for the policy to adjust to other regions after applying appropriate coordinate transformations to the observations. Using this smaller training region significantly reduces training time compared to training over the entire parking lot.



For this environment:

- The training region is a 22.5 m x 20 m space with the target spot at its horizontal center.
- The observations are the position errors X_e and Y_e of the ego vehicle with respect to the target pose, the sine and cosine of the true heading angle θ, and the lidar sensor readings.
- The vehicle speed during parking is a constant 2 m/s.
- The action signals are discrete steering angles that range between +/- 45 degrees in steps of 15 degrees.
- The vehicle is considered parked if the errors with respect to target pose are within specified tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).
- The episode terminates if the ego vehicle goes out of the bounds of the training region, collides with an obstacle, or parks successfully.
- The reward r_t provided at time t, is:

$$r_t = 2e^{-\left(0.05{X_e}^2 + 0.04{Y_e}^2\right)} + 0.5e^{-40{\theta_e}^2} - 0.05\delta^2 + 100f_t - 50g_t$$

Here, X_e, Y_e, and θ_e are the position and heading angle errors of the ego vehicle from the target pose, and δ is the steering angle. f_t (0 or 1) indicates whether the vehicle has parked and g_t (0 or 1) indicates if the vehicle has collided with an obstacle at time t.

The coordinate transformations on vehicle pose (X, Y, θ) observations for different parking spot locations are as follows:

- 1-14: no transformation
- 15-22: $\bar{X} = Y, \bar{Y} = -X, \bar{\theta} = \theta - \pi/2$
- 23-36: $\bar{X} = 100 - X, \bar{Y} = 60 - Y, \bar{\theta} = \theta - \pi$
- 37-40: $\bar{X} = 60 - Y, \bar{Y} = X, \bar{\theta} = \theta - 3\pi/2$
- 41-52: $\bar{X} = 100 - X, \bar{Y} = 30 - Y, \bar{\theta} = \theta + \pi$
- 53-64: $\bar{X} = X, \bar{Y} = Y - 28, \bar{\theta} = \theta$

Create the observation and action specifications for the environment.

```
numObservations = 16;
observationInfo = rlNumericSpec([numObservations 1]);
observationInfo.Name = 'observations';

steerMax = pi/4;
discreteSteerAngles = -steerMax : deg2rad(15) : steerMax;
actionInfo = rlFiniteSetSpec(num2cell(discreteSteerAngles));
actionInfo.Name = 'actions';
numActions = numel(actionInfo.Elements);
```

Create the Simulink environment interface, specifying the path to the RL Agent block.

```
blk = [mdl '/RL Controller/RL Agent'];
env = rlSimulinkEnv(mdl,blk,observationInfo,actionInfo);
```

Specify a reset function for training. The `autoParkingValetResetFcn` function resets the initial pose of the ego vehicle to random values at the start of each episode.

```
env.ResetFcn = @autoParkingValetResetFcn;
```

For more information on creating Simulink environments, see `rlSimulinkEnv` (Reinforcement Learning Toolbox).

Create Agent

The RL agent in this example is a proximal policy optimization (PPO) agent with a discrete action space. PPO agents rely on actor and critic representations to learn the optimal policy. The agent maintains deep neural network-based function approximators for the actor and critic. To learn more about PPO agents, see “Proximal Policy Optimization Agents” (Reinforcement Learning Toolbox).

Set the random seed generator for reproducibility.

```
rng(0)
```

To create the critic representations, first create a deep neural network with 16 inputs and one output. The output of the critic network is the state value function for a particular observation.

```
criticNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observations')
    fullyConnectedLayer(128,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(128,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(128,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];
```

Create the critic for the PPO agent. For more information, see `rlValueRepresentation` (Reinforcement Learning Toolbox) and `rlRepresentationOptions` (Reinforcement Learning Toolbox).

```
criticOptions = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
critic = rlValueRepresentation(criticNetwork,observationInfo,...
    'Observation',{'observations'},criticOptions);
```

The outputs of the actor networks are the probabilities of taking each possible steering action when the vehicle is in a certain state. Create the actor deep neural network.

```
actorNetwork = [
    featureInputLayer(numObservations,'Normalization','none','Name','observations')
    fullyConnectedLayer(128,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(128,'Name','fc2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(numActions, 'Name', 'out')
    softmaxLayer('Name','actionProb')];
```

Create a stochastic actor representation for the PPO agent. For more information, see `rlStochasticActorRepresentation` (Reinforcement Learning Toolbox).

```
actorOptions = rlRepresentationOptions('LearnRate',2e-4,'GradientThreshold',1);
actor = rlStochasticActorRepresentation(actorNetwork,observationInfo,actionInfo,...
    'Observation',{'observations'},actorOptions);
```

Specify the agent options and create the PPO agent. For more information on PPO agent options, see `rlPPOAgentOptions` (Reinforcement Learning Toolbox).

```
agentOpts = rlPPOAgentOptions(...
    'SampleTime',Ts,...
    'ExperienceHorizon',200,...
    'ClipFactor',0.2,...
    'EntropyLossWeight',0.01,...
    'MiniBatchSize',64,...
    'NumEpoch',3,...
    'AdvantageEstimateMethod',"gae",...
    'GAEFactor',0.95,...
    'DiscountFactor',0.998);
agent = rlPPOAgent(actor,critic,agentOpts);
```

During training, the agent collects experiences until it reaches experience horizon of 200 steps or the episode terminates and then trains from mini-batches of 64 experiences for three epochs. An objective function clip factor of 0.2 improves training stability and a discount factor value of 0.998 encourages long term rewards. Variance in critic the output is reduced by the generalized advantage estimate method with a GAE factor of 0.95.

Train Agent

For this example, you train the agent for a maximum of 10000 episodes, with each episode lasting a maximum of 200 time steps. The training terminates when the maximum number of episodes is reached or the average reward over 100 episodes exceeds 100.

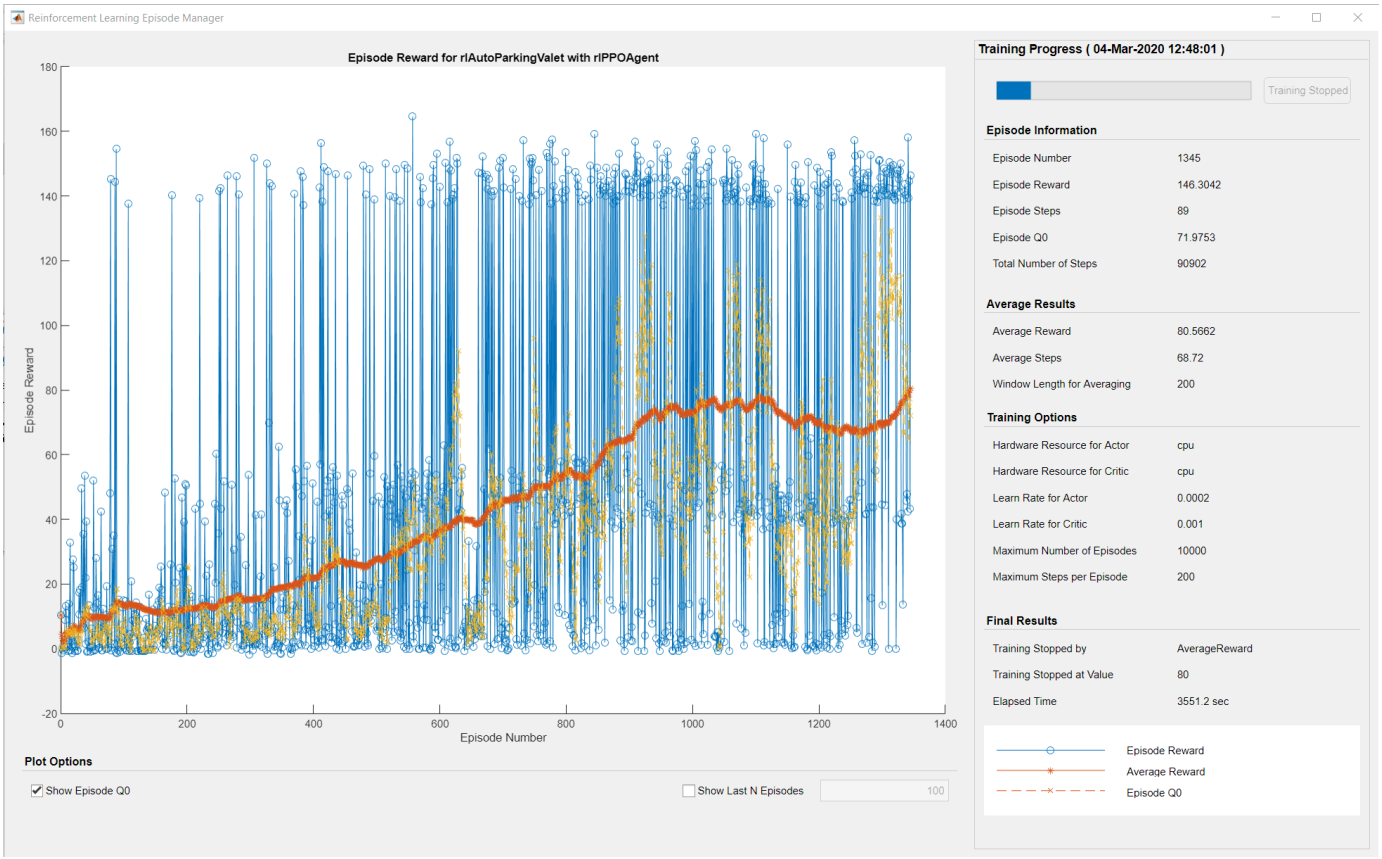
Specify the options for training using an `rlTrainingOptions` object.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',10000,...
```

```
    'MaxStepsPerEpisode',200,...
    'ScoreAveragingWindowLength',200,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',80);
```

Train the agent using the `train` (Reinforcement Learning Toolbox) function. Training this agent is a computationally intensive process that takes several minutes to complete. To save time while running this example, load a pretrained agent by setting `doTraining` to `false`. To train the agent yourself, set `doTraining` to `true`.

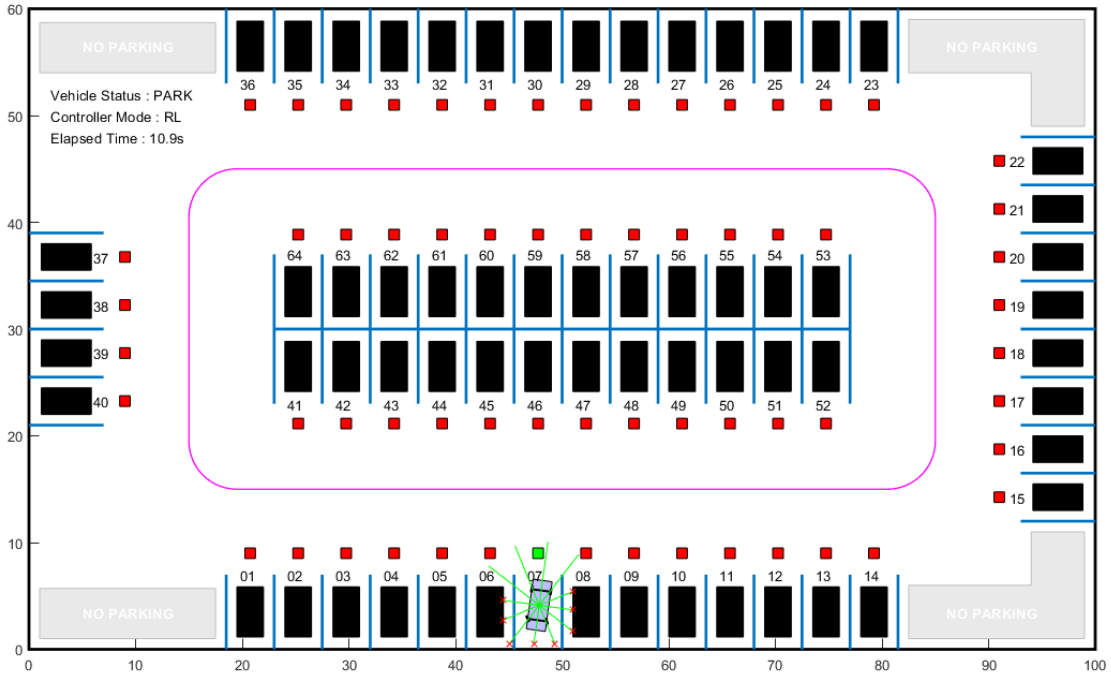
```
doTraining = false;
if doTraining
    trainingStats = train(agent,env,trainOpts);
else
    load('rlAutoParkingValetAgent.mat','agent');
end
```



Simulate Agent

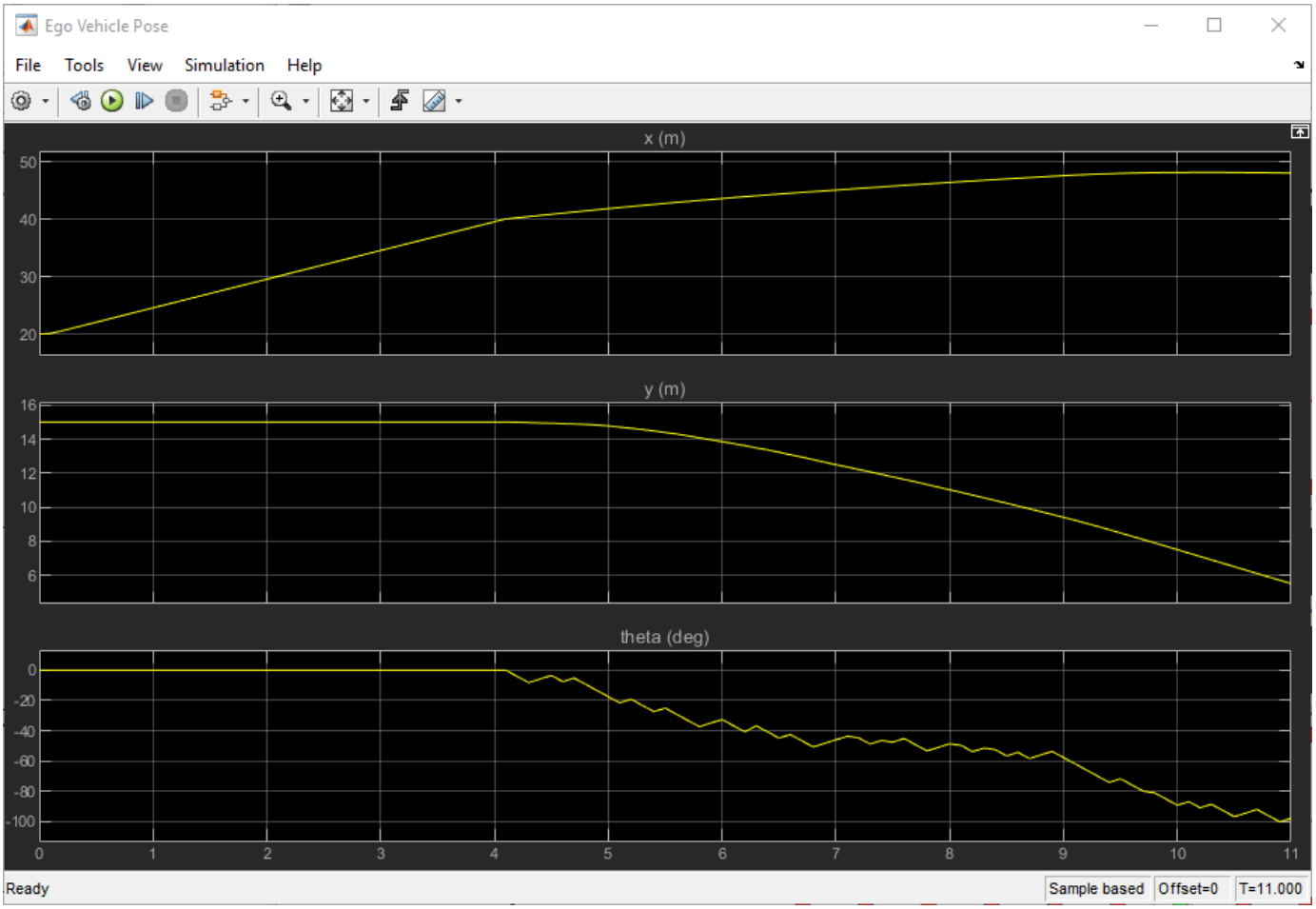
Simulate the model to park the vehicle in the free parking spot. To simulate the vehicle parking in different locations, change the free spot location in the following code.

```
freeSpotIdx = 7;  % free spot location
sim(mdl);
```



The vehicle reaches the target pose within the specified error tolerances of +/- 0.75 m (position) and +/-10 degrees (orientation).

To view the ego vehicle position and orientation, open the Ego Vehicle Pose scope.



See Also

`train`

More About

- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Predictive Maintenance Examples

Chemical Process Fault Detection Using Deep Learning

This example shows how to use simulation data to train a neural network that can detect faults in a chemical process. The network detects the faults in the simulated process with high accuracy. The typical workflow is as follows:

1. Preprocess the data
2. Design the layer architecture
3. Train the network
4. Perform validation
5. Test the network

Download Data Set

This example uses MATLAB-formatted files converted by MathWorks® from the Tennessee Eastman Process (TEP) simulation data [1] on page 16-0 . These files are available at the MathWorks support files site. See the disclaimer.

The data set consists of four components — fault-free training, fault-free testing, faulty training, and faulty testing. Download each file separately.

```
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultytesting.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultytraining.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultfreetesting.mat',url);
url = 'https://www.mathworks.com/supportfiles/predmaint/chemical-process-fault-detection-data/fau
websave('faultfreetraining.mat',url);
```

Load the downloaded files into the MATLAB® workspace.

```
load('faultfreetesting.mat');
load('faultfreetraining.mat');
load('faultytesting.mat');
load('faultytraining.mat');
```

Each component contains data from simulations that were run for every permutation of two parameters:

- Fault Number — For faulty data sets, an integer value from 1 to 20 that represents a different simulated fault. For fault-free data sets, a value of 0.
- Simulation run — For all data sets, integer values from 1 to 500, where each value represents a unique random generator state for the simulation.

The length of each simulation was dependent on the data set. All simulations were sampled every three minutes.

- Training data sets contain 500 time samples from 25 hours of simulation.
- Testing data sets contain 960 time samples from 48 hours of simulation.

Each data frame has the following variables in its columns:

- Column 1 (`faultNumber`) indicates the fault type, which varies from 0 through 20. A fault number 0 means fault-free while fault numbers 1 to 20 represent different fault types in the TEP.
- Column 2 (`simulationRun`) indicates the number of times the TEP simulation ran to obtain complete data. In the training and test data sets, the number of runs varies from 1 to 500 for all fault numbers. Every `simulationRun` value represents a different random generator state for the simulation.
- Column 3 (`sample`) indicates the number of times TEP variables were recorded per simulation. The number varies from 1 to 500 for the training data sets and from 1 to 960 for the testing data sets. The TEP variables (columns 4 to 55) were sampled every 3 minutes for a duration of 25 hours and 48 hours for the training and testing data sets respectively.
- Columns 4–44 (`xmeas_1` through `xmeas_41`) contain the measured variables of the TEP.
- Columns 45–55 (`xmv_1` through `xmv_11`) contain the manipulated variables of the TEP.

Examine subsections of two of the files.

```
head(faultfreetraining,4)
```

```
ans=4×55 table
    faultNumber    simulationRun    sample    xmeas_1    xmeas_2    xmeas_3    xmeas_4    xmeas_5
    ___________    _____________    ______    _______    _______    _______    _______    _______

        0              1              1       0.25038      3674       4529       9.232     26.889
        0              1              2       0.25109    3659.4     4556.6     9.4264     26.721
        0              1              3       0.25038    3660.3     4477.8     9.4426     26.875
        0              1              4       0.24977    3661.3     4512.1     9.4776     26.758
```

```
head(faultytraining,4)
```

```
ans=4×55 table
    faultNumber    simulationRun    sample    xmeas_1    xmeas_2    xmeas_3    xmeas_4    xmeas_5
    ___________    _____________    ______    _______    _______    _______    _______    _______

        1              1              1       0.25038      3674       4529       9.232     26.889
        1              1              2       0.25109    3659.4     4556.6     9.4264     26.721
        1              1              3       0.25038    3660.3     4477.8     9.4426     26.875
        1              1              4       0.24977    3661.3     4512.1     9.4776     26.758
```

Clean Data

Remove data entries with the fault numbers 3, 9, and 15 in both the training and testing data sets. These fault numbers are not recognizable, and the associated simulation results are erroneous.

```
faultytesting(faultytesting.faultNumber == 3,:) = [];
faultytesting(faultytesting.faultNumber == 9,:) = [];
faultytesting(faultytesting.faultNumber == 15,:) = [];


faultytraining(faultytraining.faultNumber == 3,:) = [];
faultytraining(faultytraining.faultNumber == 9,:) = [];
faultytraining(faultytraining.faultNumber == 15,:) = [];
```

Divide Data

Divide the training data into training and validation data by reserving 20 percent of the training data for validation. Using a validation data set enables you to evaluate the model fit on the training data

set while you tune the model hyperparameters. Data splitting is commonly used to prevent the network from overfitting and underfitting.

Get the total number of rows in both faulty and fault-free training data sets.

```
H1 = height(faultfreetraining);
H2 = height(faultytraining);
```

The simulation run is the number of times the TEP process was repeated with a particular fault type. Get the maximum simulation run from the training data set as well as from the testing data set.

```
msTrain = max(faultfreetraining.simulationRun);
msTest = max(faultytesting.simulationRun);
```

Calculate the maximum simulation run for the validation data.

```
rTrain = 0.80;
msVal = ceil(msTrain*(1 - rTrain));
msTrain = msTrain*rTrain;
```

Get the maximum number of samples or time steps (that is, the maximum number of times that data was recorded during a TEP simulation).

```
sampleTrain = max(faultfreetraining.sample);
sampleTest = max(faultfreetesting.sample);
```

Get the division point (row number) in the fault-free and faulty training data sets to create validation data sets from the training data sets.

```
rowLim1 = ceil(rTrain*H1);
rowLim2 = ceil(rTrain*H2);

trainingData = [faultfreetraining{1:rowLim1,:}; faultytraining{1:rowLim2,:}];
validationData = [faultfreetraining{rowLim1 + 1:end,:}; faultytraining{rowLim2 + 1:end,:}];
testingData = [faultfreetesting{:,:}; faultytesting{:,:}];
```

Network Design and Preprocessing

The final data set (consisting of training, validation, and testing data) contains 52 signals with 500 uniform time steps. Hence, the signal, or sequence, needs to be classified to its correct fault number which makes it a problem of sequence classification.

- Long short-term memory (LSTM) networks are suited to the classification of sequence data.
- LSTM networks are good for time-series data as they tend to remember the uniqueness of past signals in order to classify new signals
- An LSTM network enables you to input sequence data into a network and make predictions based on the individual time steps of the sequence data. For more information on LSTM networks, see “Long Short-Term Memory Networks” on page 1-75.
- To train the network to classify sequences using the `trainNetwork` function, you must first preprocess the data. The data must be in cell arrays, where each element of the cell array is a matrix representing a set of 52 signals in a single simulation. Each matrix in the cell array is the set of signals for a particular simulation of TEP and can either be faulty or fault-free. Each set of signals points to a specific fault class ranging from 0 through 20.

As was described previously in the Data Set section, the data contains 52 variables whose values are recorded over a certain amount of time in a simulation. The `sample` variable represents the number

of times these 52 variables are recorded in one simulation run. The maximum value of the `sample` variable is 500 in the training data set and 960 in the testing data set. Thus, for each simulation, there is a set of 52 signals of length 500 or 960. Each set of signals belongs to a particular simulation run of the TEP and points to a particular fault type in the range 0 – 20.

The training and test datasets both contain 500 simulations for each fault type. Twenty percent (from training) is kept for validation which leaves the training data set with 400 simulations per fault type and validation data with 100 simulations per fault type. Use the helper function `helperPreprocess` to create sets of signals, where each set is a double matrix in a single element of the cell array that represents a single TEP simulation. Hence, the sizes of the final training, validation, and testing data sets are as follows:

- Size of `Xtrain`: (Total number of simulations) X (Total number of fault types) = 400 X 18 = 7200
- Size of `XVal`: (Total number of simulations) X (Total number of fault types) = 100 X 18 = 1800
- Size of `Xtest`: (Total number of simulations) X (Total number of fault types) = 500 X 18 = 9000

In the data set, the first 500 simulations are of `0` fault type (fault-free) and the order of the subsequent faulty simulations is known. This knowledge enables the creation of true responses for the training, validation, and testing data sets.

```
Xtrain = helperPreprocess(trainingData,sampleTrain);
Ytrain = categorical([zeros(msTrain,1);repmat([1,2,4:8,10:14,16:20],1,msTrain)']);

XVal = helperPreprocess(validationData,sampleTrain);
YVal = categorical([zeros(msVal,1);repmat([1,2,4:8,10:14,16:20],1,msVal)']);

Xtest = helperPreprocess(testingData,sampleTest);
Ytest = categorical([zeros(msTest,1);repmat([1,2,4:8,10:14,16:20],1,msTest)']);
```

Normalize Data Sets

Normalization is a technique that scales the numeric values in a data set to a common scale without distorting differences in the range of values. This technique ensures that a variable with a larger value does not dominate other variables in the training. It also converts the numeric values in a higher range to a smaller range (usually –1 to 1) without losing any important information required for training.

Compute the mean and the standard deviation for 52 signals using data from all simulations in the training data set.

```
tMean = mean(trainingData(:,4:end))';
tSigma = std(trainingData(:,4:end))';
```

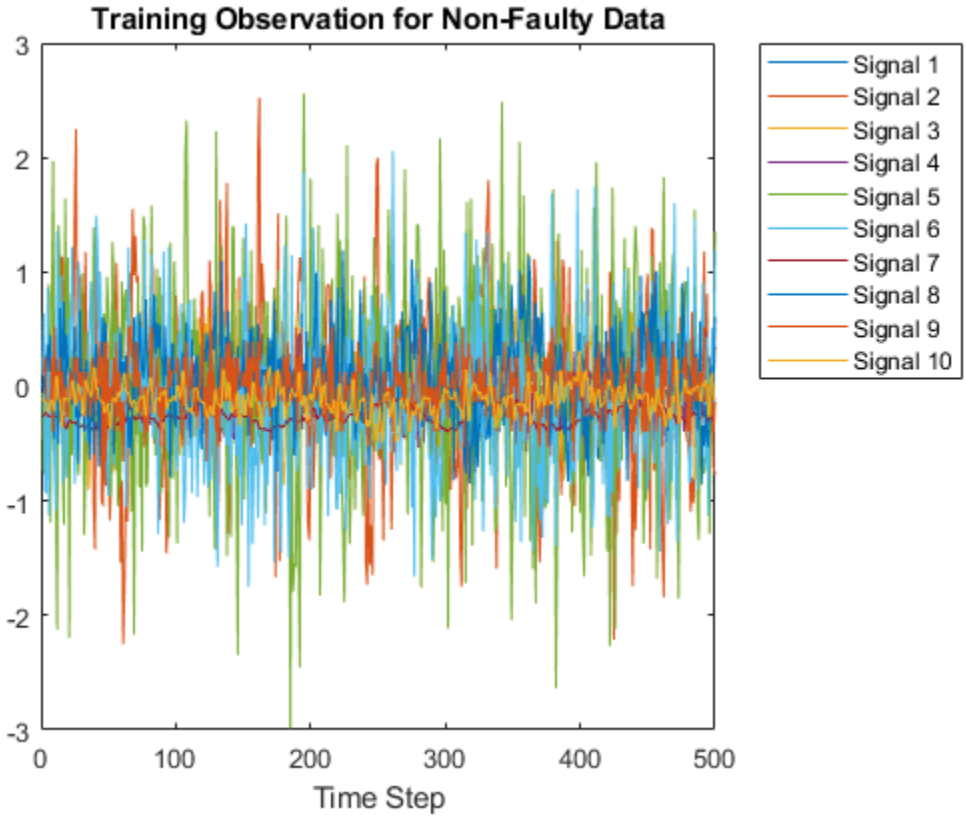
Use the helper function `helperNormalize` to apply normalization to each cell in the three data sets based on the mean and standard deviation of the training data.

```
Xtrain = helperNormalize(Xtrain, tMean, tSigma);
XVal = helperNormalize(XVal, tMean, tSigma);
Xtest = helperNormalize(Xtest, tMean, tSigma);
```

Visualize Data

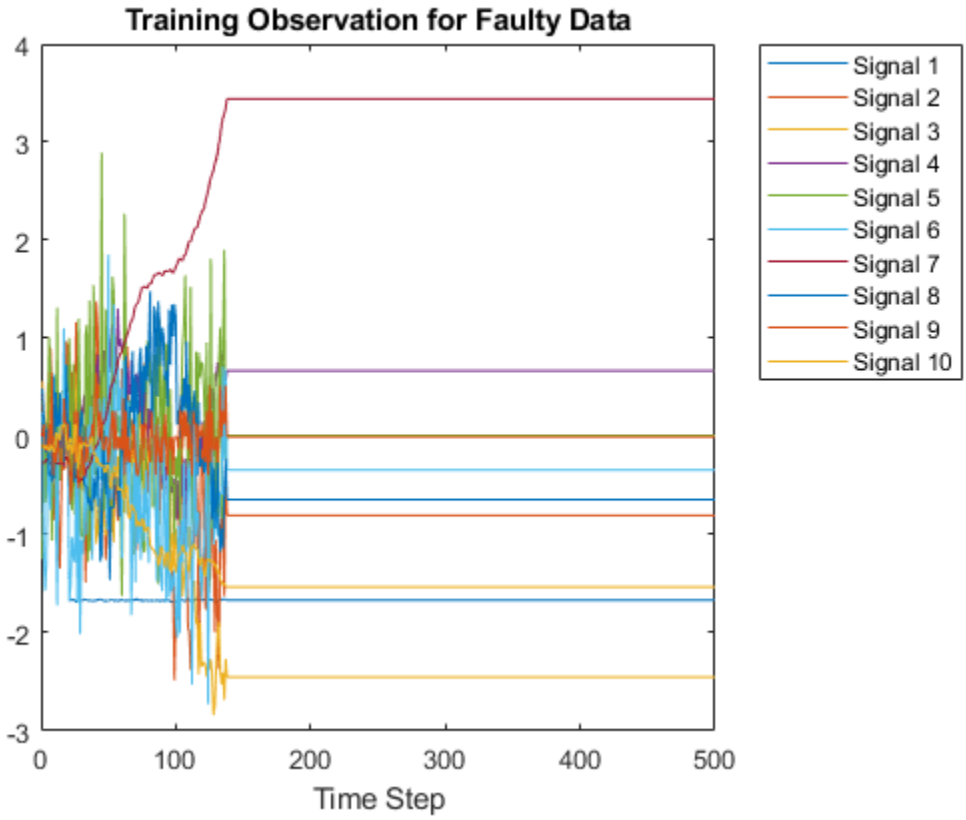
The `Xtrain` data set contains 400 fault-free simulations followed by 6800 faulty simulations.Visualize the fault-free and faulty data. First, create a plot of the fault-free data. For the purposes of this example, plot and label only 10 signals in the `Xtrain` data set to create an easy-to-read figure.

```
figure;
splot = 10;
plot(Xtrain{1}(1:10,:)');
xlabel("Time Step");
title("Training Observation for Non-Faulty Data");
legend("Signal " + string(1:splot),'Location','northeastoutside');
```



Now, compare the fault-free plot to a faulty plot by plotting any of the cell array elements after 400.

```
figure;
plot(Xtrain{1000}(1:10,:)');
xlabel("Time Step");
title("Training Observation for Faulty Data");
legend("Signal " + string(1:splot),'Location','northeastoutside');
```



Layer Architecture and Training Options

LSTM layers are a good choice for sequence classification as LSTM layers tend to remember only the important aspects of the input sequence.

- Specify the input layer `sequenceInputLayer` to be of the same size as the number of input signals (52).
- Specify 3 LSTM hidden layers with 52, 40, and 25 units. This specification is inspired by the experiment performed in [2] on page 16-0 . For more information on using LSTM networks for sequence classification, see “Sequence Classification Using Deep Learning” on page 4-2.
- Add 3 dropout layers in between the LSTM layers to prevent over-fitting. A dropout layer randomly sets input elements of the next layer to zero with a given probability so that the network does not become sensitive to a small set of neurons in the layer
- Finally, for classification, include a fully connected layer of the same size as the number of output classes (18). After the fully connected layer, include a softmax layer that assigns decimal probabilities (prediction possibility) to each class in a multi-class problem and a classification layer to output the final fault type based on output from the softmax layer.

```
numSignals = 52;
numHiddenUnits2 = 52;
numHiddenUnits3 = 40;
numHiddenUnits4 = 25;
numClasses = 18;

layers = [ ...
```

```
    sequenceInputLayer(numSignals)
    lstmLayer(numHiddenUnits2,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits3,'OutputMode','sequence')
    dropoutLayer(0.2)
    lstmLayer(numHiddenUnits4,'OutputMode','last')
    dropoutLayer(0.2)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Set the training options that `trainNetwork` uses.

Maintain the default value of name-value pair `'ExecutionEnvironment'` as `'auto'`. With this setting, the software chooses the execution environment automatically. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). Because this example uses a large amount of data, using GPU speeds up training time considerably.

Setting the name-value argument pair `'Shuffle'` to `'every-epoch'` avoids discarding the same data every epoch.

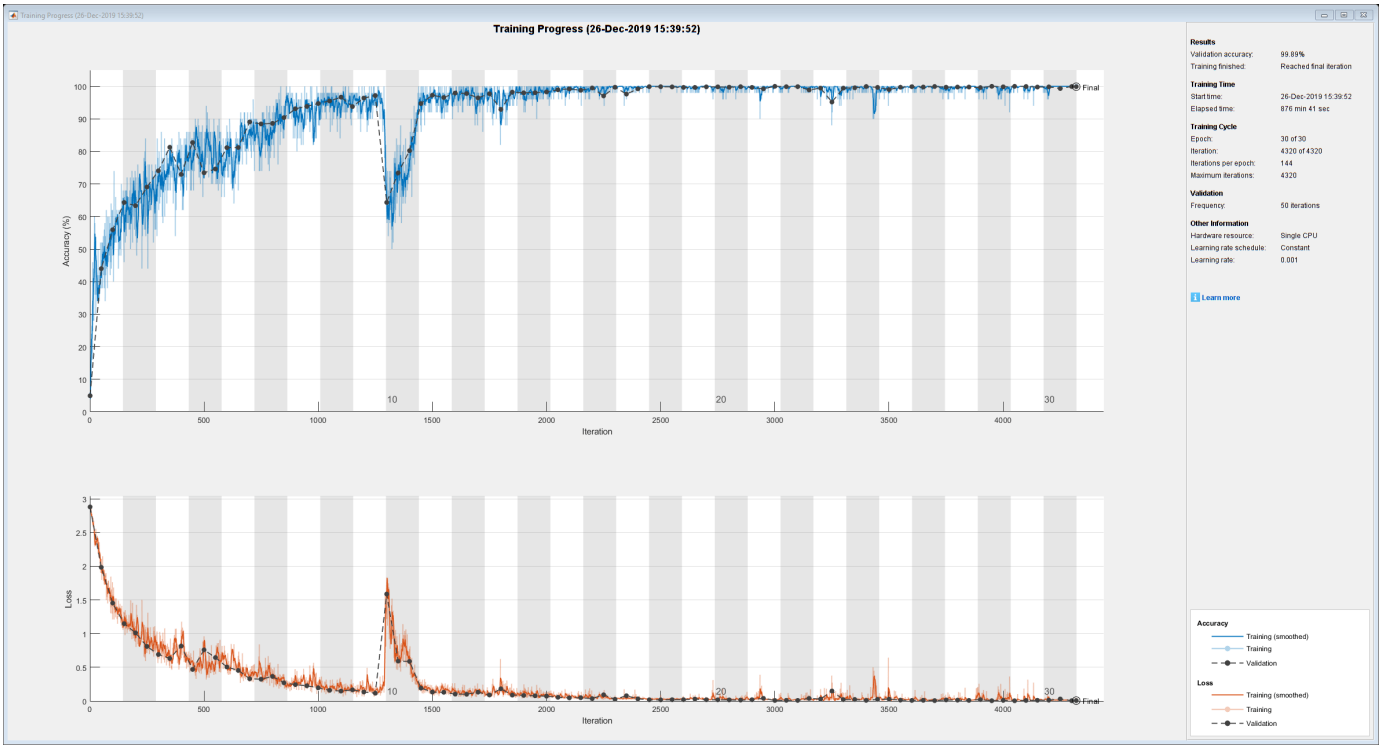
For more information on training options for deep learning, see `trainingOptions`.

```
maxEpochs = 30;
miniBatchSize = 50;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','auto', ...
    'GradientThreshold',1, ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize', miniBatchSize,...
    'Shuffle','every-epoch', ...
    'Verbose',0, ...
    'Plots','training-progress',...
    'ValidationData',{XVal,YVal});
```

Train Network

Train the LSTM network using `trainNetwork`.

```
net = trainNetwork(Xtrain,Ytrain,layers,options);
```



The training progress figure displays a plot of the network accuracy. To the right of the figure, view information on the training time and settings.

Testing Network

Run the trained network on the test set and predict the fault type in the signals.

```
Ypred = classify(net,Xtest,...
    'MiniBatchSize', miniBatchSize,...
    'ExecutionEnvironment','auto');
```

Calculate the accuracy. The accuracy is the number of true labels in the test data that match the classifications from `classify` divided by the number of images in the test data.

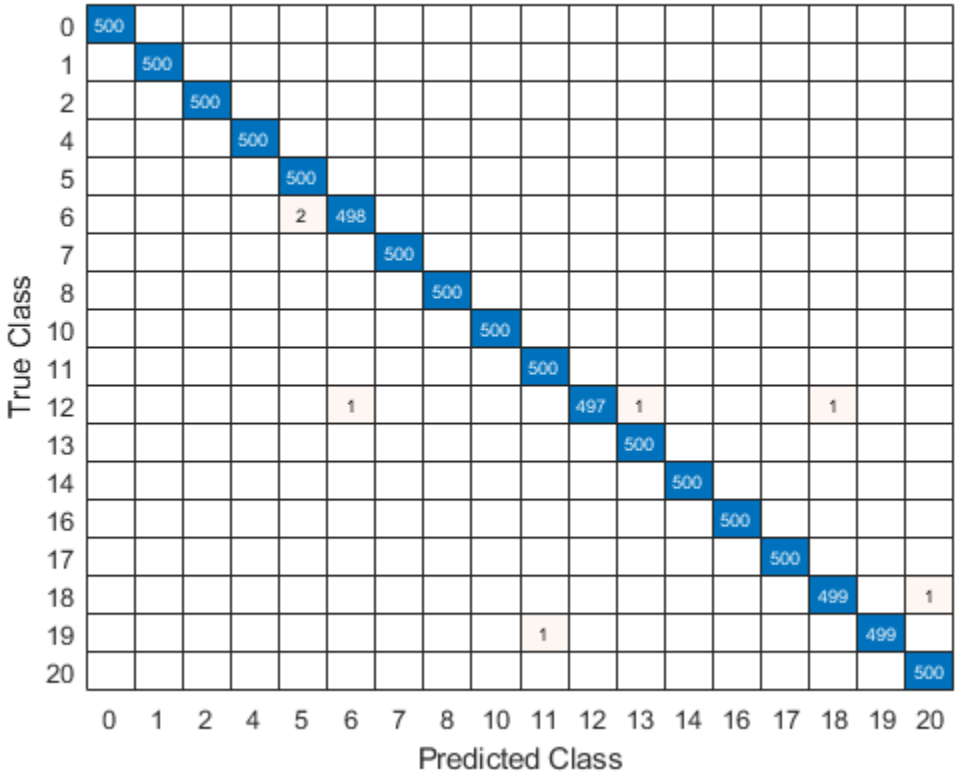
```
acc = sum(Ypred == Ytest)./numel(Ypred)
```

```
acc = 0.9992
```

High accuracy indicates that the neural network is successfully able to identify the fault type of unseen signals with minimal errors. Hence, the higher the accuracy, the better the network.

Plot a confusion matrix using true class labels of the test signals to determine how well the network identifies each fault.

```
confusionchart(Ytest,Ypred);
```



Using a confusion matrix, you can assess the effectiveness of a classification network. the confusion matrix has numerical values in the main diagonal and zeros elsewhere. The trained network in this example is effective and classifies more than 99% of signals correctly.

References

[1] Rieth, C. A., B. D. Amsel, R. Tran., and B. Maia. "Additional Tennessee Eastman Process Simulation Data for Anomaly Detection Evaluation." Harvard Dataverse, Version 1, 2017. https://doi.org/10.7910/DVN/6C3JR1.

[2] Heo, S., and J. H. Lee. "Fault Detection and Classification Using Artificial Neural Networks." Department of Chemical and Biomolecular Engineering, Korea Advanced Institute of Science and Technology.

Helper Functions

`helperPreprocess`

The helper function `helperPreprocess` uses the maximum sample number to preprocess the data. The sample number indicates the signal length, which is consistent across the data set. A for-loop goes over the data set with a signal length filter to form sets of 52 signals. Each set is an element of a cell array. Each cell array represents a single simulation.

```
function processed = helperPreprocess(mydata,limit)
    H = size(mydata);
    processed = {};
    for ind = 1:limit:H
```

```
        x = mydata(ind:(ind+(limit-1)),4:end);
        processed = [processed; x'];
    end
end
```

`helperNormalize`

The helper function `helperNormalize` uses the data, mean, and standard deviation to normalize the data.

```
function data = helperNormalize(data,m,s)
    for ind = 1:size(data)
        data{ind} = (data{ind} - m)./s;
    end
end
```

See Also

`lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

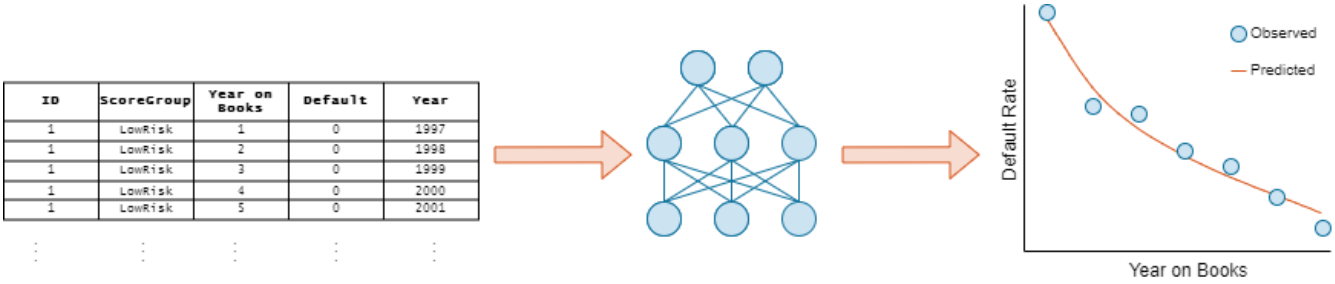
Computational Finance Examples

Compare Deep Learning Networks for Credit Default Prediction

This example shows how to create, train, and compare three deep learning networks for predicting credit default probability.

The panel data set of consumer loans enables you to identify and predict default rate patterns. You can train a neural network using the panel data to predict the default rate from year on books and risk level.

This example requires Deep Learning Toolbox™ and Risk Management Toolbox™.



In this example, you create and train three models for credit default prediction:

- Logistic regression network (also known as a single-layer perceptron)
- Multilayer perceptron (MLP)
- Residual network (ResNet)

You can express each of these models as a neural network of varying complexity and depth.

Load Credit Default Data

Load the retail credit panel data set. This data includes the following variables:

- `ID` — Loan identifier.
- `ScoreGroup` — Credit score at the beginning of the loan, discretized into three groups: `High Risk`, `Medium Risk`, and `Low Risk`.
- `YOB` — Years on books.
- `Default` — Default indicator. A value of `1` for `Default` means that the loan defaulted in the corresponding calendar year.
- `Year` — Calendar year.

```
filename = fullfile(toolboxdir('risk'),'riskdata','RetailCreditPanelData.mat');
tbl = load(filename).data;
```

Encode Categorical Variables

To train a deep learning network, you must first encode the categorical `ScoreGroup` variable to one-hot encoded vectors.

View the order of the `ScoreGroup` categories.

```
categories(tbl.ScoreGroup)'
```

```
ans = 1×3 cell
    {'High Risk'}    {'Medium Risk'}    {'Low Risk'}
```

Convert the categorical `ScoreGroup` variable to one-hot encoded vectors using the `onehotencode` function.

```
riskGroup = onehotencode(tbl.ScoreGroup,2);
```

Add the one-hot vectors to the table.

```
tbl.HighRisk = riskGroup(:,1);
tbl.MediumRisk = riskGroup(:,2);
tbl.LowRisk = riskGroup(:,3);
```

Remove the original `ScoreGroup` variable from the table using `removevars`.

```
tbl = removevars(tbl,{'ScoreGroup'});
```

Because you want to predict the `Default` variable response, move the `Default` variable to the end of the table.

```
tbl = movevars(tbl,'Default','After','LowRisk');
```

View the first few rows of the table. Notice that the `ScoreGroup` variable has been split into multiple columns, with the categorical values as the variable names.

```
head(tbl)
```

```
ans=8×7 table
    ID    YOB    Year    HighRisk    MediumRisk    LowRisk    Default
    __    ___    ____    ________    __________    _______    _______

    1      1     1997       0            0           1          0
    1      2     1998       0            0           1          0
    1      3     1999       0            0           1          0
    1      4     2000       0            0           1          0
    1      5     2001       0            0           1          0
    1      6     2002       0            0           1          0
    1      7     2003       0            0           1          0
    1      8     2004       0            0           1          0
```

Split Data

Partition the data set into training, validation, and test partitions using the unique loan ID numbers. Set aside 60% of the data for training, 20% for validation, and 20% for testing.

Find the unique loan IDs.

```
idx = unique(tbl.ID);
numObservations = length(idx);
```

Determine the number of observations for each partition.

```
numObservationsTrain = floor(0.6*numObservations);
numObservationsValidation = floor(0.2*numObservations);
numObservationsTest = numObservations - numObservationsTrain - numObservationsValidation;
```

Create an array of random indices corresponding to the observations and partition it using the partition sizes.

```
rng('default')
idxShuffle = idx(randperm(numObservations));

idxTrain = idxShuffle(1:numObservationsTrain);
idxValidation = idxShuffle(numObservationsTrain+1:numObservationsTrain+numObservationsValidation)
idxTest = idxShuffle(numObservationsTrain+numObservationsValidation+1:end);
```

Find the table entries corresponding to the data set partitions.

```
idxTrainTbl = ismember(tbl.ID,idxTrain);
idxValidationTbl = ismember(tbl.ID,idxValidation);
idxTestTbl = ismember(tbl.ID,idxTest);
```

Keep the variables of interest for the task (`YOB`, `HighRisk`, `MediumRisk`, `LowRisk`, and `Default`) and remove all other variables from the table.

```
tbl = removevars(tbl,{'ID','Year'});
head(tbl)
```

```
ans=8×5 table
    YOB    HighRisk    MediumRisk    LowRisk    Default
    ___    ________    __________    _______    _______

     1         0            0           1          0
     2         0            0           1          0
     3         0            0           1          0
     4         0            0           1          0
     5         0            0           1          0
     6         0            0           1          0
     7         0            0           1          0
     8         0            0           1          0
```

Partition the table of data into training, validation, and testing partitions using the indices.

```
tblTrain = tbl(idxTrainTbl,:);
tblValidation = tbl(idxValidationTbl,:);
tblTest = tbl(idxTestTbl,:);
```

Define Network Architectures

You can use different deep learning architectures for the task of predicting credit default probabilities. Smaller networks are quick to train, but deeper networks can learn more abstract features. Choosing a neural network architecture requires balancing computation time against accuracy. In this example, you define three network architectures, with varying levels of complexity.

Logistic Regression Network

The first network is a simple neural network containing four layers.

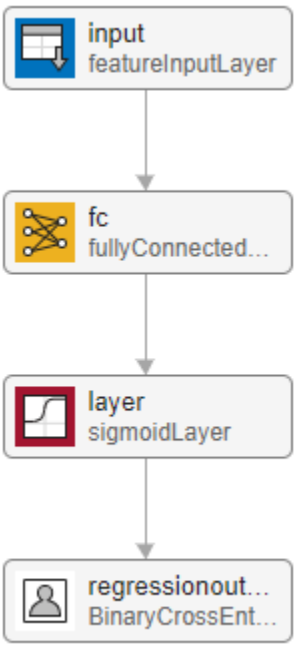
Start with a feature input layer, which passes tabular data (credit panel data) to the network. In this example, there are four input features: `YOB`, `HighRisk`, `MediumRisk`, and `LowRisk`. Configure the input layer to normalize the data using z-score normalization. Normalizing the data is important for tasks where the scale and range of the input variables is very different.

Next, use a fully connected layer with a single output followed by a sigmoid layer. For the final layer, use a custom binary cross-entropy loss layer. This layer is attached to this example as a supporting file.

```
logisticLayers = [
    featureInputLayer(4,'Normalization','zscore')
    fullyConnectedLayer(1)
    sigmoidLayer
    BinaryCrossEntropyLossLayer('output')];
```

This network is called a *single-layer perceptron*. You can visualize the network using Deep Network Designer or the `analyzeNetwork` function.

```
deepNetworkDesigner(logisticLayers)
```



You can easily show that the single-layer perceptron neural network is equivalent to logistic regression. Let $\mathbf{x}_i$ be a 1-by-4 vector containing the features for observation i. With input $\mathbf{x}_i$, the output of the fully connected layer is

$$\mathbf{W}\mathbf{x}_i + b = W_1 x_{i,1} + W_2 x_{i,2} + W_3 x_{i,3} + W_4 x_{i,4} + b.$$

The output of the fully connected layer provides the input for the sigmoid layer. The sigmoid layer then outputs

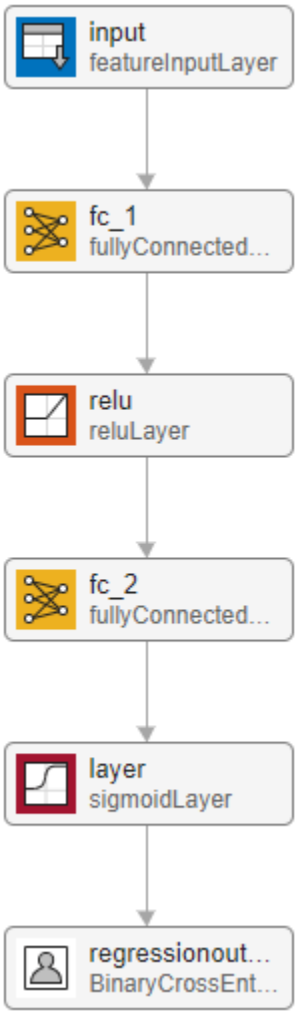
$$S(\mathbf{W}\mathbf{x}_i + b) = \frac{1}{1 + \exp\{-(W_1 x_{i,1} + W_2 x_{i,2} + W_3 x_{i,3} + W_4 x_{i,4} + b)\}}.$$

The output of the sigmoid layer is equivalent to the logistic regression model. The final layer of the neural network is a binary cross-entropy loss layer. Minimizing the binary cross-entropy loss is equivalent to maximizing the likelihood in a logistic regression model.

Multilayer Perceptron

The next network has a similar architecture to the logistic regression model, but has an additional fully connected layer with an output size of 100, followed by a ReLU nonlinear activation function. This type of network is called a *multilayer perceptron* due to the addition of another hidden layer and a nonlinear activation function. Whereas the single-layer perceptron can learn only linear functions, the multilayer perceptron can learn complex, nonlinear relationships between the input and output data.

```
mlpLayers = [
    featureInputLayer(4,'Normalization','zscore')
    fullyConnectedLayer(100)
    reluLayer
    fullyConnectedLayer(1)
    sigmoidLayer
    BinaryCrossEntropyLossLayer('output')];

deepNetworkDesigner(mlpLayers)
```



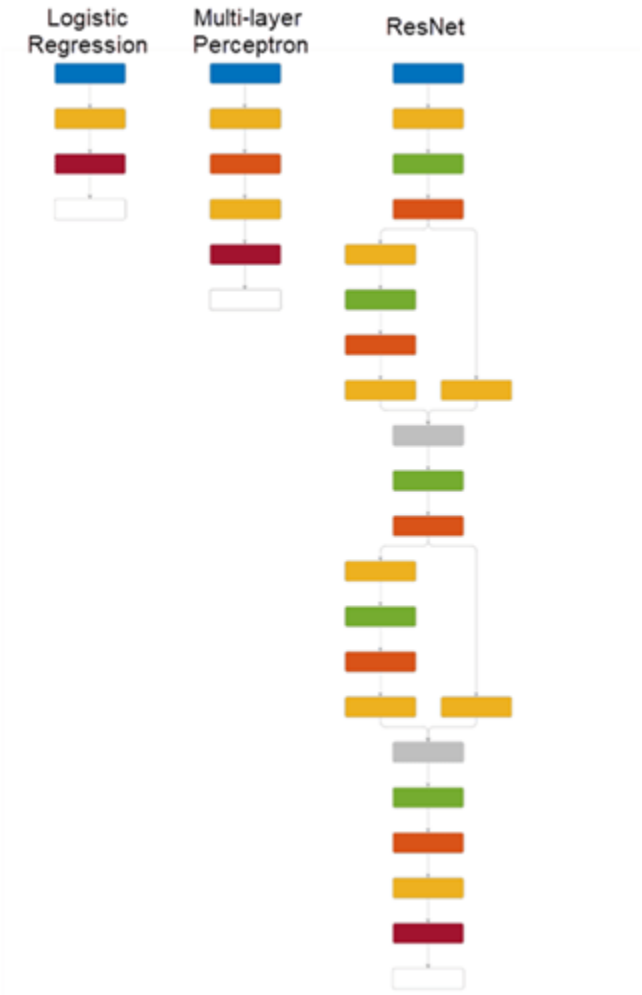
Residual Network

For the final network, create a residual network (ResNet) [1] on page 17-0 from multiple stacks of fully connected layers and ReLU activations. Originally developed for image classification, ResNets have proven successful across many domains. Because a ResNet has many more parameters than multilayer perceptrons or logistic networks, they take longer to train.

```
residualLayers = [
    featureInputLayer(4,'Normalization','zscore','Name','input')

    fullyConnectedLayer(16,'Name','fc1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')

    fullyConnectedLayer(32,'Name','resblock1-fc1')
    batchNormalizationLayer('Name','resblock1-bn1')
    reluLayer('Name','resblock1-relu1')

    fullyConnectedLayer(32,'Name','resblock1-fc2')
    additionLayer(2,'Name','resblock1-add')
    batchNormalizationLayer('Name','resblock1-bn2')
    reluLayer('Name','resblock1-relu2')

    fullyConnectedLayer(64,'Name','resblock2-fc1')
    batchNormalizationLayer('Name','resblock2-bn1')
    reluLayer('Name','resblock2-relu1')

    fullyConnectedLayer(64,'Name','resblock2-fc2')
    additionLayer(2,'Name','resblock2-add')
    batchNormalizationLayer('Name','resblock2-bn2')
    reluLayer('Name','resblock2-relu2')

    fullyConnectedLayer(1,'Name','fc2')
    sigmoidLayer('Name','sigmoid')
    BinaryCrossEntropyLossLayer('output')];

residualLayers = layerGraph(residualLayers);
residualLayers = addLayers(residualLayers,fullyConnectedLayer(32,'Name','resblock1-fc-shortcut'));
residualLayers = addLayers(residualLayers,fullyConnectedLayer(64,'Name','resblock2-fc-shortcut'));

residualLayers = connectLayers(residualLayers,'relu1','resblock1-fc-shortcut');
residualLayers = connectLayers(residualLayers,'resblock1-fc-shortcut','resblock1-add/in2');
residualLayers = connectLayers(residualLayers,'resblock1-relu2','resblock2-fc-shortcut');
residualLayers = connectLayers(residualLayers,'resblock2-fc-shortcut','resblock2-add/in2');

deepNetworkDesigner(residualLayers)
```

Network Depth

The *depth* of a network is an important concept in deep learning and is defined as the largest number of sequential convolutional or fully connected layers (represented by yellow blocks in the following diagram) on a path from the input layer to the output layer. The deeper a network is, the more complex features it can learn. In this example, the logistic network has a depth of 1, the multilayer perceptron has a depth of 2, and the residual network has a depth of 6.



Specify Training Options

Specify the training options.

- Train using the Adam optimizer.
- Set the initial learning rate to 0.001.
- Set the mini-batch size to 512.
- Turn on the training progress plot and turn off the command window output.
- Shuffle the data at the beginning of each epoch.
- Monitor the network accuracy during training by specifying validation data and using it to validate the network every 1000 iterations.

```
options = trainingOptions('adam', ...
    'InitialLearnRate',0.001, ...
    'MiniBatchSize',512, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'Shuffle','every-epoch', ...
    'ValidationData',tblValidation, ...
    'ValidationFrequency',1000);
```

The loss landscape of the logistic regression network is convex, therefore, it does not need to train for as many epochs. For the logistic regression and multilayer perceptron models, train for 15 epochs. For the more complex residual network, train for 50 epochs.

```
logisticOptions = options;
logisticOptions.MaxEpochs = 15;

mlpOptions = options;
mlpOptions.MaxEpochs = 15;

residualOptions = options;
residualOptions.MaxEpochs = 50;
```

The three networks have different architectures, so they require different sets of training options to achieve optimal performance. You can perform optimization programmatically or interactively using Experiment Manager. For an example showing how to perform a hyperparameter sweep of the training options, see “Create a Deep Learning Experiment for Classification” on page 6-2.

Train Network

Train the networks using the architectures defined, the training data, and the training options. By default, `trainNetwork` uses a GPU if one is available; otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `ExecutionEnvironment` name-value argument of `trainingOptions`.

To avoid waiting for training, load pretrained networks by setting the `doTrain` flag to `false`. To train the networks using `trainNetwork`, set the `doTrain` flag to `true`.

Training times using a NVIDIA® GeForce® RTX 2080 Ti are:

- Logistic network — Approximately 4 minutes
- Multilayer perceptron — Approximately 5 minutes
- Residual network — Approximately 35 minutes

```
doTrain = false;

if doTrain
    logisticNet = trainNetwork(tblTrain,'Default',logisticLayers,logisticOptions);
    mlpNet = trainNetwork(tblTrain,'Default',mlpLayers,mlpOptions);
    residualNet = trainNetwork(tblTrain,'Default',residualLayers,residualOptions);
else
    load logisticTrainedNetwork
    load mlpTrainedNetwork
    load residualTrainedNetwork
end
```

Test Network

Predict the default probability of the test data using the trained networks.

```
tblTest.logisticPred = predict(logisticNet,tblTest(:,1:end-1));
tblTest.mlpPred = predict(mlpNet,tblTest(:,1:end-1));
tblTest.residualPred = predict(residualNet,tblTest(:,1:end-1));
```

Default Rates by Year on Books

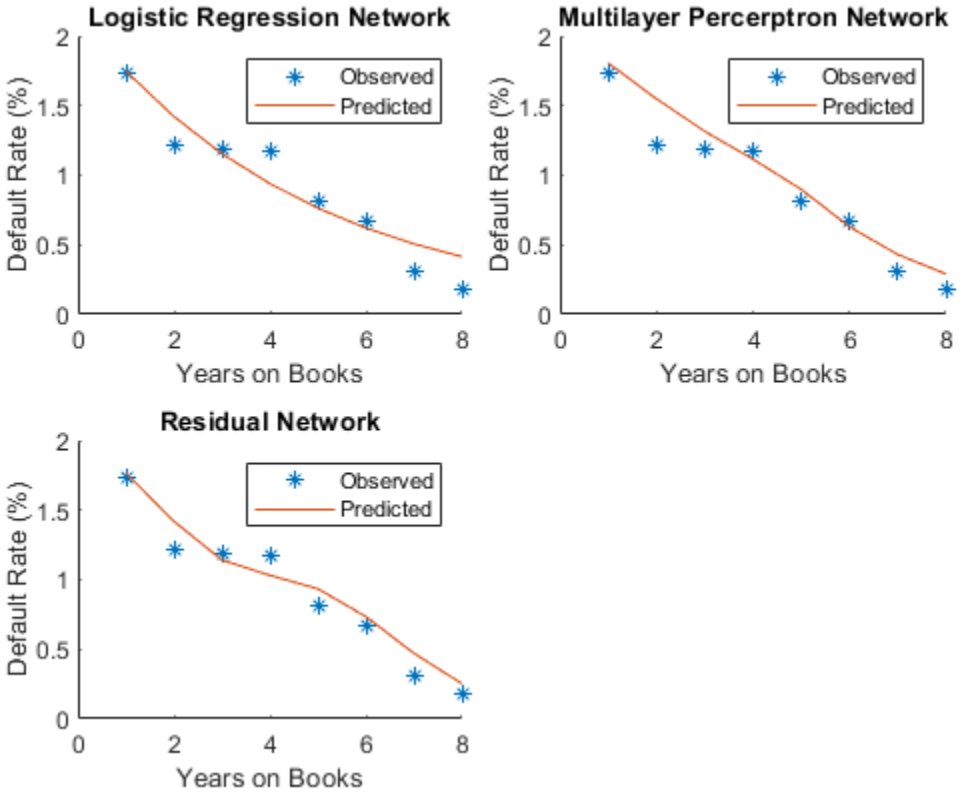
To assess the performance of the network, use the `groupsummary` function to group the true default rates and corresponding predictions by years on books (represented by the `YOB` variable) and calculate the mean value.

```
summaryYOB = groupsummary(tblTest,'YOB','mean',{'Default','logisticPred','mlpPred','residualPred
head(summaryYOB)
```

```
ans=8×6 table
    YOB    GroupCount    mean_Default    mean_logisticPred    mean_mlpPred    mean_residualPred
    ___    __________    ____________    _________________    ____________    _________________

     1       19364         0.017352           0.017471          0.018056           0.017663
     2       18917         0.012158           0.014209          0.015486           0.014192
     3       18526         0.011875           0.011538          0.013154           0.011409
     4       18232         0.011683          0.0093902          0.011151           0.010311
     5       17925        0.0082008           0.007626         0.0089826          0.0093438
     6       17727        0.0066565          0.0062047         0.0062967          0.0073401
     7       12294        0.0030909          0.0050643         0.0042998          0.0047071
     8        6361        0.0017293          0.0041463         0.0029052          0.0025272
```

Plot the true average default rate against the average predictions by years on books.

```
networks = ["Logistic Regression Network","Multilayer Percerptron Network","Residual Network"];

figure
tiledlayout('flow','TileSpacing','compact')

for i = 1:3
    nexttile
    scatter(summaryYOB.YOB,summaryYOB.mean_Default*100,'*');
    hold on
    plot(summaryYOB.YOB,summaryYOB{:,i+3}*100);
    hold off
    title(networks(i))
    xlabel('Years on Books')
    ylabel('Default Rate (%)')
    legend('Observed','Predicted')
end
```



All three networks show a clear downward trend, with default rates going down as the number of years on books increases. Years three and four are an exception to the downward trend. Overall, the three models predict the default rates well, and even the simpler logistic regression model predicts the general trend. The residual network captures a more complex, nonlinear relationship compared to the logistic model, which can fit only a linear relationship.

Default Rates by Score Groups

Use the credit score group as a grouping variable to compute the observed and predicted default rate for each score group.

Decode `ScoreGroup` back into the categorical score groups.

```
ScoreGroup = onehotdecode(tblTest{:,2:4},{'HighRisk','MediumRisk','LowRisk'},2);
tblTest.ScoreGroup = ScoreGroup;
tblTest = removevars(tblTest,{'HighRisk','MediumRisk','LowRisk'});

riskGroups = categories(tblTest.ScoreGroup);
```

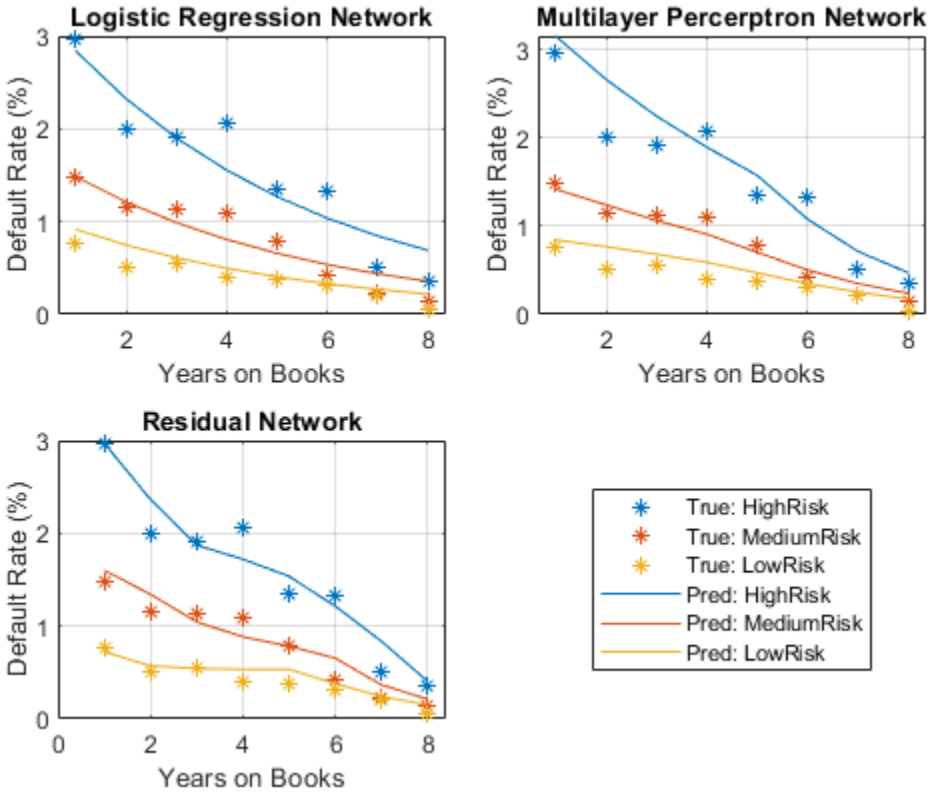
Use the `groupsummary` function to group the true default rate and the predictions by `YOB` and `ScoreGroup`, and return the mean for each group.

```
numYOB = height(summaryYOB);
numRiskGroups = height(riskGroups);

summaryYOBScore = groupsummary(tblTest,{'ScoreGroup','YOB'},'mean',{'Default','logisticPred','ml
head(summaryYOBScore)
```

```
ans=8×7 table
    ScoreGroup    YOB    GroupCount    mean_Default    mean_logisticPred    mean_mlpPred    mean
    __________    ___    __________    ____________    _________________    ____________    ____

     HighRisk      1        6424          0.029577           0.028404          0.031563
     HighRisk      2        6180          0.020065            0.02325          0.026649
     HighRisk      3        5949          0.019163           0.019013          0.022484
     HighRisk      4        5806          0.020668           0.015535          0.018957
     HighRisk      5        5634           0.01349           0.012686           0.01577
     HighRisk      6        5531          0.013379           0.010354          0.010799
     HighRisk      7        3862         0.0051787          0.0084466         0.0071398
     HighRisk      8        2027         0.0034534          0.0068881         0.0047145
```

Plot the true average default rate against the predicted rate by years on books and risk group.

```
figure
t = tiledlayout('flow','TileSpacing','compact');
color = lines(3);

YOB = summaryYOBScore.YOB;
default = summaryYOBScore.mean_Default*100;
group = summaryYOBScore.ScoreGroup;

for i = 1:3
    pred = summaryYOBScore{:,i+4}*100;
    meanScore = reshape(pred,numYOB,numRiskGroups);

    nexttile
    hs = gscatter(YOB,default,group,color,'*',6,false);
    hold on
    colororder(color)
    plot(meanScore)
    hold off
    title(networks(i))
    xlabel('Years on Books')
    ylabel('Default Rate (%)')
    grid on
end

labels = ["True: " + riskGroups; "Pred: " + riskGroups];
lgd = legend(labels);
lgd.Layout.Tile = 4;
```



The plot shows that all score groups behave similarly as time progresses, with a general downward trend. Across the high risk group, year four does not follow the downward trend. In the medium risk group, years three and four appear flat. Finally, in the low risk group, year three shows an increase. These irregular trends are difficult to discern with the simpler logistic regression model.

For an example showing how to use the locally-interpretable model-agnostic explanations (LIME) and Shapley values interpretability techniques to understand the predictions of a residual network for credit default prediction, see “Interpret and Stress-Test Deep Learning Networks for Probability of Default” (Risk Management Toolbox).

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

See Also

`trainNetwork` | `trainingOptions` | `fullyConnectedLayer` | **Deep Network Designer** | `featureInputLayer`

Related Examples

- “Interpret and Stress-Test Deep Learning Networks for Probability of Default” (Risk Management Toolbox)

- “Hedging an Option Using Reinforcement Learning Toolbox” (Financial Toolbox)
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Train Convolutional Neural Network for Regression” on page 3-53

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “List of Deep Learning Layers” on page 1-21

Interpret and Stress-Test Deep Learning Networks for Probability of Default

This example shows how to train a credit risk for probability of default (PD) prediction using a deep neural network. The example also shows how to use the locally interpretable model-agnostic explanations (LIME) and Shapley values interpretability techniques to understand the predictions of the model. In addition, the example analyzes model predictions for out-of-sample values and performs a stress-testing analysis.

The “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” (Risk Management Toolbox) example presents a similar workflow but uses a logistic model. The “Modeling Probabilities of Default with Cox Proportional Hazards” (Risk Management Toolbox) example uses a Cox regression, or Cox proportional hazards model. However, interpretability techniques are not discussed in either of these examples because the models are simpler and interpretable. The “Compare Deep Learning Networks for Credit Default Prediction” on page 17-2 example focuses on alternative network designs and fits simpler models without the macroeconomic variables.

While you can use these alternative, simpler models successfully to model credit risk, this example introduces explainability tools for exploring complex-modeling techniques in credit applications. To visualize and interpret the model predictions, you use Deep Learning Toolbox™ and the `lime` (Statistics and Machine Learning Toolbox) and `shapley` (Statistics and Machine Learning Toolbox) functions. To run this example, you:

1. Load and prepare credit data, reformat predictors, and split the data into training, validation, and testing sets.
2. Define a network architecture, select training options, and train the network. (A saved version of the trained network `residualTrainedNetworkMacro` is available for convenience.)
3. Apply the LIME and Shapley interpretability techniques on observations of interest (or "query points") to determine if the importance of predictors in the model is as expected.
4. Explore extreme predictor out-of-sample values to investigate the behavior of the model for new, extreme data.
5. Use the model to perform a stress-testing analysis of the predicted PD values.

Load Credit Default Data

Load the retail credit panel data set including its macroeconomic variables. The main data set (`data`) contains the following variables:

- `ID`: Loan identifier
- `ScoreGroup`: Credit score at the beginning of the loan, discretized into three groups, `High Risk`, `Medium Risk`, and `Low Risk`
- `YOB`: Years on books
- `Default`: Default indicator; the response variable
- `Year`: Calendar year

The small data set (`dataMacro`) contains macroeconomic data for the corresponding calendar years:

- `Year`: Calendar year
- `GDP`: Gross domestic product growth (year over year)

- `Market`: Market return (year over year)

The variables `YOB`, `Year`, `GDP`, and `Market` are observed at the end of the corresponding calendar year. The score group is a discretization of the original credit score when the loan started. A value of `1` for `Default` means that the loan defaulted in the corresponding calendar year.

The third data set (`dataMacroStress`) contains baseline, adverse, and severely adverse scenarios for the macroeconomic variables. This table is for the stress-testing analysis.

This example uses simulated data, but the same approach has been successfully applied to real data sets.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
head(data)
```

```
ans=8×7 table
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0        1997     2.72      7.61
    1      Low Risk      2        0        1998     3.57     26.24
    1      Low Risk      3        0        1999     2.86      18.1
    1      Low Risk      4        0        2000     2.43      3.19
    1      Low Risk      5        0        2001     1.26    -10.51
    1      Low Risk      6        0        2002    -0.59    -22.95
    1      Low Risk      7        0        2003     0.63      2.78
    1      Low Risk      8        0        2004     1.85      9.48
```

Encode Categorical Variables

To train a deep learning network, you must first encode the categorical `ScoreGroup` variable to one-hot encoded vectors.

View the order of the `ScoreGroup` categories.

```
categories(data.ScoreGroup)'
```

```
ans = 1×3 cell
    {'High Risk'}    {'Medium Risk'}    {'Low Risk'}
```

ans = 1×3 cell

{'High Risk'} {'Medium Risk'} {'Low Risk'}

One-hot encode the `ScoreGroup` variable.

```
riskGroup = onehotencode(data.ScoreGroup,2);
```

Add the one-hot vectors to the table.

```
data.HighRisk = riskGroup(:,1);
data.MediumRisk = riskGroup(:,2);
data.LowRisk = riskGroup(:,3);
```

Remove the original `ScoreGroup` variable from the table using `removevars`.

```
data = removevars(data,{'ScoreGroup'});
```

Move the `Default` variable to the end of the table, as this variable is the response you want to predict.

```
data = movevars(data,'Default','After','LowRisk');
```

View the first few rows of the table. The `ScoreGroup` variable is split into multiple columns with the categorical values as the variable names.

```
head(data)
```

```
ans=8×9 table
    ID    YOB    Year     GDP     Market    HighRisk    MediumRisk    LowRisk    Default
    __    ___    ____    _____    ______    ________    __________    _______    _______

    1      1     1997     2.72      7.61       0            0           1          0
    1      2     1998     3.57     26.24       0            0           1          0
    1      3     1999     2.86      18.1       0            0           1          0
    1      4     2000     2.43      3.19       0            0           1          0
    1      5     2001     1.26    -10.51       0            0           1          0
    1      6     2002    -0.59    -22.95       0            0           1          0
    1      7     2003     0.63      2.78       0            0           1          0
    1      8     2004     1.85      9.48       0            0           1          0
```

Split Data

Partition the data set into training, validation, and test partitions using the unique loan ID numbers. Set aside 60% of the data for training, 20% for validation, and 20% for testing.

Find the unique loan IDs.

```
idx = unique(data.ID);
numObservations = length(idx);
```

Determine the number of observations for each partition.

```
numObservationsTrain = floor(0.6*numObservations);
numObservationsValidation = floor(0.2*numObservations);
numObservationsTest = numObservations - numObservationsTrain - numObservationsValidation;
```

Create an array of random indices corresponding to the observations and partition it using the partition sizes.

```
rng('default'); % for reproducibility
idxShuffle = idx(randperm(numObservations));

idxTrain = idxShuffle(1:numObservationsTrain);
idxValidation = idxShuffle(numObservationsTrain+1:numObservationsTrain+numObservationsValidation)
idxTest = idxShuffle(numObservationsTrain+numObservationsValidation+1:end);
```

Find the table entries corresponding to the data set partitions.

```
idxTrainTbl = ismember(data.ID,idxTrain);
idxValidationTbl = ismember(data.ID,idxValidation);
idxTestTbl = ismember(data.ID,idxTest);
```

Keep the variables of interest for the task (`YOB`, `Default`, and `ScoreGroup`) and remove all other variables from the table.

```
data = removevars(data,{'ID','Year'});
head(data)
```

```
ans=8×7 table
    YOB     GDP     Market    HighRisk    MediumRisk    LowRisk    Default
    ___    _____    ______    ________    __________    _______    _______

     1      2.72      7.61        0            0            1          0
     2      3.57     26.24        0            0            1          0
     3      2.86      18.1        0            0            1          0
     4      2.43      3.19        0            0            1          0
     5      1.26    -10.51        0            0            1          0
     6     -0.59    -22.95        0            0            1          0
     7      0.63      2.78        0            0            1          0
     8      1.85      9.48        0            0            1          0
```

Partition the table of data into training, validation, and testing partitions using the indices.

```
tblTrain = data(idxTrainTbl,:);
tblValidation = data(idxValidationTbl,:);
tblTest = data(idxTestTbl,:);
```

Define Network Architecture

You can use different deep learning architectures for the task of predicting credit default probabilities. Smaller networks are quick to train, but deeper networks can learn more abstract features. Choosing a neural network architecture requires balancing computation time against accuracy. This example uses a residual architecture. For an example of other networks, see the “Compare Deep Learning Networks for Credit Default Prediction” on page 17-2 example.

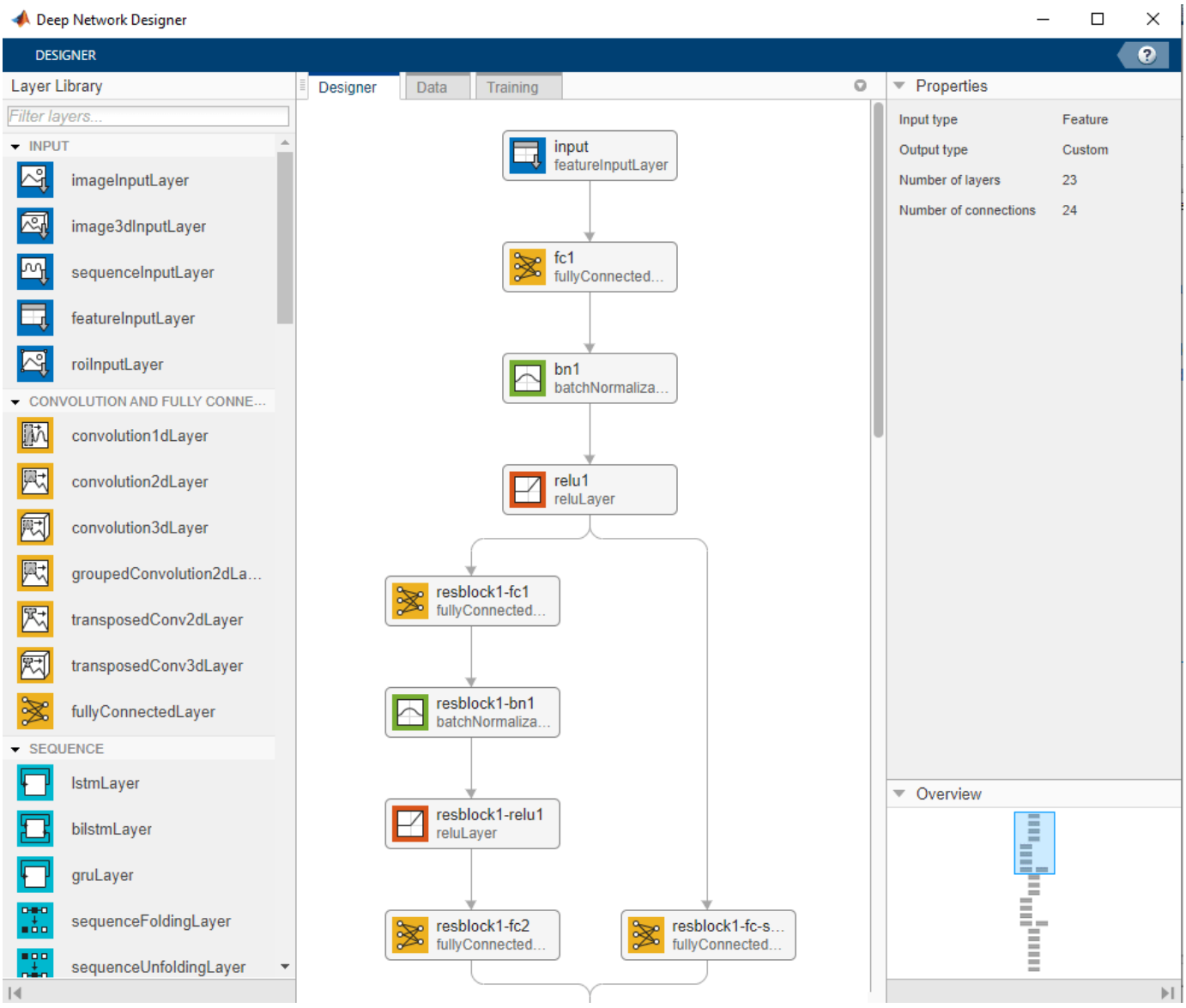
Create a residual architecture (ResNet) from multiple stacks of fully connected layers and ReLU activations. ResNet architectures are state of the art in deep learning applications and popular in deep learning literature. Originally developed for image classification, ResNets have proven successful across many domains [1 on page 17-0].

```
residualLayers = [
    featureInputLayer(6, 'Normalization', 'zscore', 'Name', 'input')
    fullyConnectedLayer(16, 'Name', 'fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'bn1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(32, 'Name', 'resblock1-fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'resblock1-bn1')
    reluLayer('Name', 'resblock1-relu1')
    fullyConnectedLayer(32, 'Name', 'resblock1-fc2','WeightsInitializer','he')
    additionLayer(2, 'Name', 'resblock1-add')
    batchNormalizationLayer('Name', 'resblock1-bn2')
    reluLayer('Name', 'resblock1-relu2')
    fullyConnectedLayer(64, 'Name', 'resblock2-fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'resblock2-bn1')
    reluLayer('Name', 'resblock2-relu1')
    fullyConnectedLayer(64, 'Name', 'resblock2-fc2','WeightsInitializer','he')
    additionLayer(2, 'Name', 'resblock2-add')
    batchNormalizationLayer('Name', 'resblock2-bn2')
    reluLayer('Name', 'resblock2-relu2')
```

```
    fullyConnectedLayer(1, 'Name', 'fc2','WeightsInitializer','he')
    sigmoidLayer('Name', 'sigmoid')
    BinaryCrossEntropyLossLayer('output')];

residualLayers = layerGraph(residualLayers);
residualLayers = addLayers(residualLayers,fullyConnectedLayer(32, 'Name', 'resblock1-fc-shortcut
residualLayers = addLayers(residualLayers,fullyConnectedLayer(64, 'Name', 'resblock2-fc-shortcut

residualLayers = connectLayers(residualLayers, 'relu1', 'resblock1-fc-shortcut');
residualLayers = connectLayers(residualLayers, 'resblock1-fc-shortcut', 'resblock1-add/in2');
residualLayers = connectLayers(residualLayers, 'resblock1-relu2', 'resblock2-fc-shortcut');
residualLayers = connectLayers(residualLayers, 'resblock2-fc-shortcut', 'resblock2-add/in2');
```

You can visualize the network using Deep Network Designer or the `analyzeNetwork` function.

```
deepNetworkDesigner(residualLayers)
```



Specify Training Options

In this example, train each network with these training options:

- Train using the Adam optimizer.
- Set the initial learning rate to `0.001`.
- Set the mini-batch size to `512`.
- Train for `75` epochs.
- Turn on the training progress plot and turn off the command window output.
- Shuffle the data at the beginning of each epoch.
- Monitor the network accuracy during training by specifying validation data and using it to validate the network every 1000 iterations.

```
options = trainingOptions('adam', ...
    'InitialLearnRate',0.001, ...
    'MiniBatchSize',512, ...
    'MaxEpochs',75, ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'Shuffle','every-epoch', ...
    'ValidationData',tblValidation, ...
    'ValidationFrequency',1000);
```

The “Compare Deep Learning Networks for Credit Default Prediction” on page 17-2 example fits the same type of network, but it excludes the macroeconomic predictors. In that example, if you increase the number of epochs from `50` to `75`, you can improve accuracy without overfitting concerns.

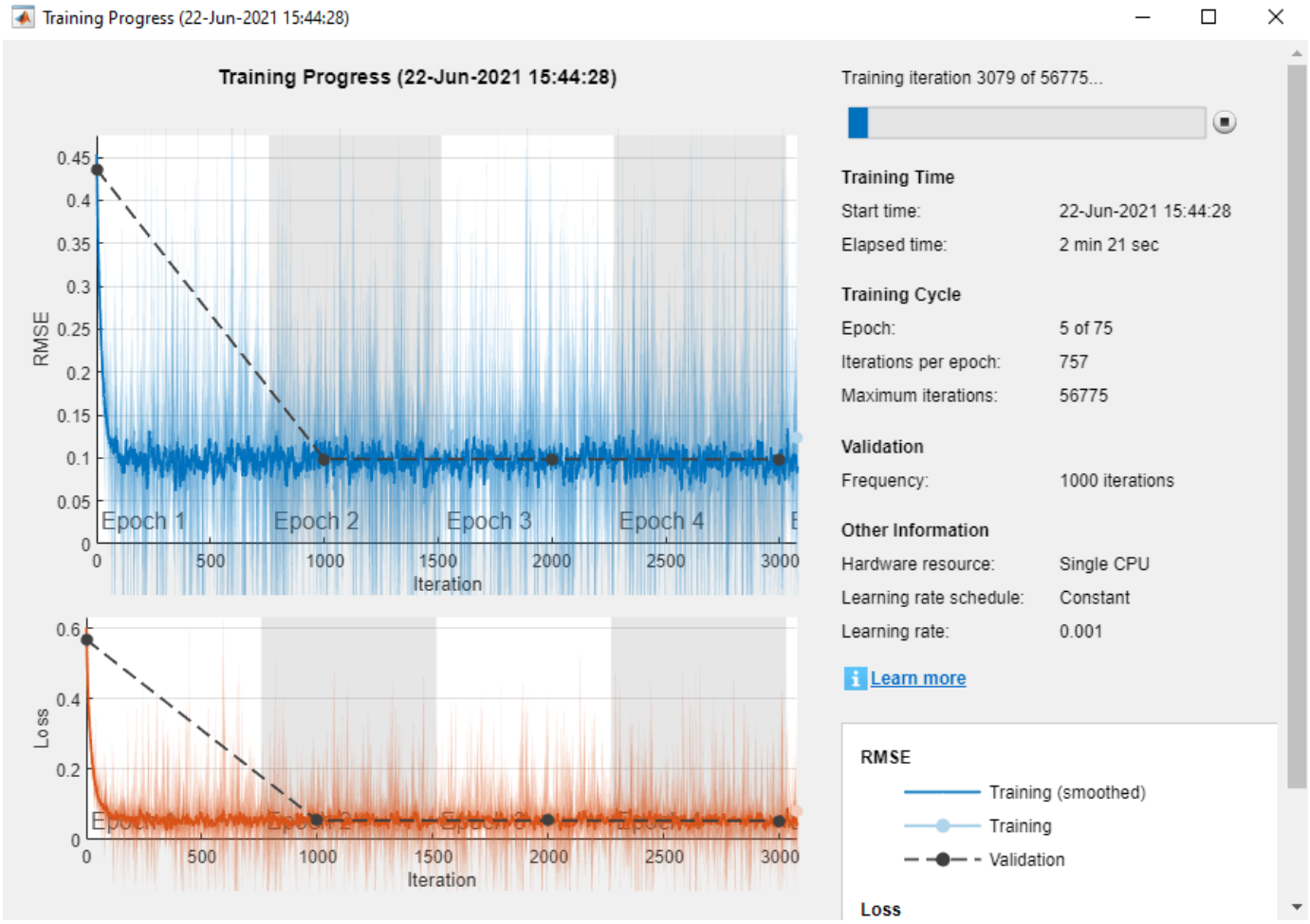
You can perform optimization programmatically or interactively using Experiment Manager. For an example showing how to perform a hyperparameter sweep of the training options, see “Create a Deep Learning Experiment for Classification” on page 6-2.

Train Network

Train the network using the architecture that yoy defined, the training data, and the training options. By default, `analyzeNetwork` uses a GPU if one is available; otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`.

To avoid waiting for the training, load pretrained networks by setting the `doTrain` flag to `false`. To train the networks using `analyzeNetwork`, set the `doTrain` flag to `true`. The Training Progress window displays progress. The training time using an NVIDIA® GeForce® RTX 2080 is about 35 minutes for 75 epochs.

```
doTrain = false;

if doTrain
    residualNetMacro = trainNetwork(tblTrain,'Default',residualLayers,options);
else
    load residualTrainedNetworkMacro.mat
end
```



Test Network

Use the `predict` function to predict the default probability of the test data using the trained networks.

```
tblTest.residualPred = predict(residualNetMacro,tblTest(:,1:end-1));
```

Plot Default Rates by Year on Books

To assess the performance of the network, use the `groupsummary` function to group the true default rates and corresponding predictions by years on the books (represented by the `YOB` variable) and calculate the mean value.

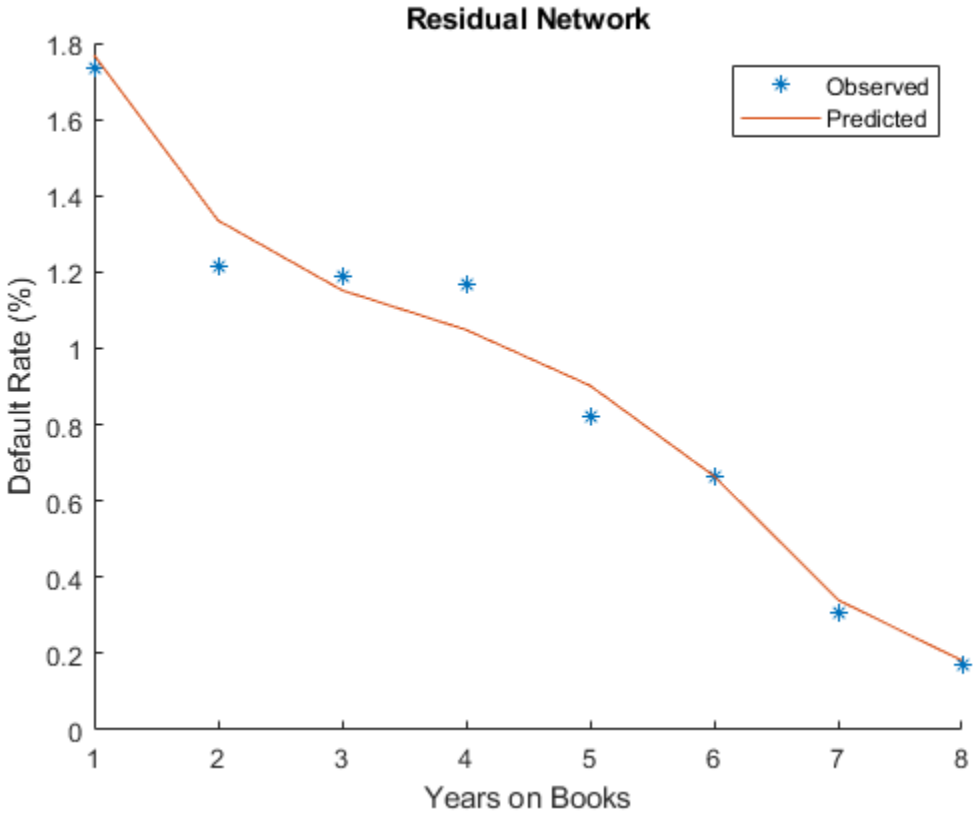
```
summaryYOB = groupsummary(tblTest,'YOB','mean',{'Default','residualPred'});
head(summaryYOB)
```

```
ans=8×4 table
    YOB    GroupCount    mean_Default    mean_residualPred
    ___    __________    ____________    _________________

     1       19364         0.017352          0.017688
     2       18917         0.012158          0.013354
     3       18526         0.011875          0.011522
```

```
    4       18232         0.011683          0.010485
    5       17925        0.0082008         0.0090247
    6       17727        0.0066565         0.0066525
    7       12294        0.0030909         0.0034051
    8        6361        0.0017293         0.0018151
```

Plot the true average default rate against the average predictions by `YOB`.

```
figure
scatter(summaryYOB.YOB,summaryYOB.mean_Default*100,'*');
hold on
plot(summaryYOB.YOB,summaryYOB.mean_residualPred*100);
hold off
title('Residual Network')
xlabel('Years on Books')
ylabel('Default Rate (%)')
legend('Observed','Predicted')
```



The plot shows a good fit on the test data. The model seems to capture the overall trend as the age of the loan (`YOB` value) increases, as well as changes in the steepness of the trend.

The rest of this example shows some ways to better understand the model. First, it reviews standard explainability techniques that you can apply to this model, specifically, the `lime` (Statistics and Machine Learning Toolbox) and `shapley` (Statistics and Machine Learning Toolbox) functions. Then, it explores the behavior of the model in new (out-of-sample) data values. Finally, the example uses the model to predict PD values under stressed macroeconomic conditions, also known as stress testing.

Explain Model with LIME and Shapley

The local interpretable model-agnostic explanations (LIME) method and the Shapley method both aim to explain the behavior of the model at a particular observation of interest or "query point." More specifically, these techniques help you to understand the importance of each variable in the prediction made for a particular observation. For more information, see `lime` (Statistics and Machine Learning Toolbox) and `shapley` (Statistics and Machine Learning Toolbox).

For illustration purposes, choose two observations from the data to better interpret the model predictions. The response values (last column) are not needed.

The first observation is a seasoned, low-risk loan. In other words, it has an initial score of `LowRisk` and eight years on the books.

```
obs1 = data(8,1:end-1);
disp(obs1)
```

YOB	GDP	Market	HighRisk	MediumRisk	LowRisk
8	1.85	9.48	0	0	1

The second observation is a new, high-risk loan. That is, the score is `HighRisk` and it is in its first year on the books.

```
obs2 = data(88,1:end-1);
disp(obs2)
```

YOB	GDP	Market	HighRisk	MediumRisk	LowRisk
1	2.72	7.61	1	0	0

Both `lime` (Statistics and Machine Learning Toolbox) and `shapley` (Statistics and Machine Learning Toolbox) require a reference data set with predictor values. This reference data can be the training data itself, or any other reference data where the model can be evaluated to explore the behavior of the model. More data points allow the explainability methods to understand the behavior of the model in more regions. However, a large data set can also slow down the computations, especially for `shapley` (Statistics and Machine Learning Toolbox). For illustration purposes, use the first 1000 rows from the training data set. The response values (last column) are not needed.

```
predictorData = data(1:1000,1:end-1);
```

`lime` (Statistics and Machine Learning Toolbox) and `shapley` (Statistics and Machine Learning Toolbox) also require a function handle to the `predict` function. Treat `predict` like a black-box model and call it multiple times to make predictions on data and gather information on the behavior of the model.

```
blackboxFcn = @(x)predict(residualNetMacro,x);
```

Create `lime` Object

Create a `lime` (Statistics and Machine Learning Toolbox) object by passing the black-box function handle and the selected predictor data.

Randomly generated synthetic data underlying `lime` (Statistics and Machine Learning Toolbox) can affect the importance. The report may change depending on the synthetic data generated. It can also

change due to optional arguments, such as the `'KernelWidth'` parameter that controls the area around the observation of interest ("query point") while you fit the local model.

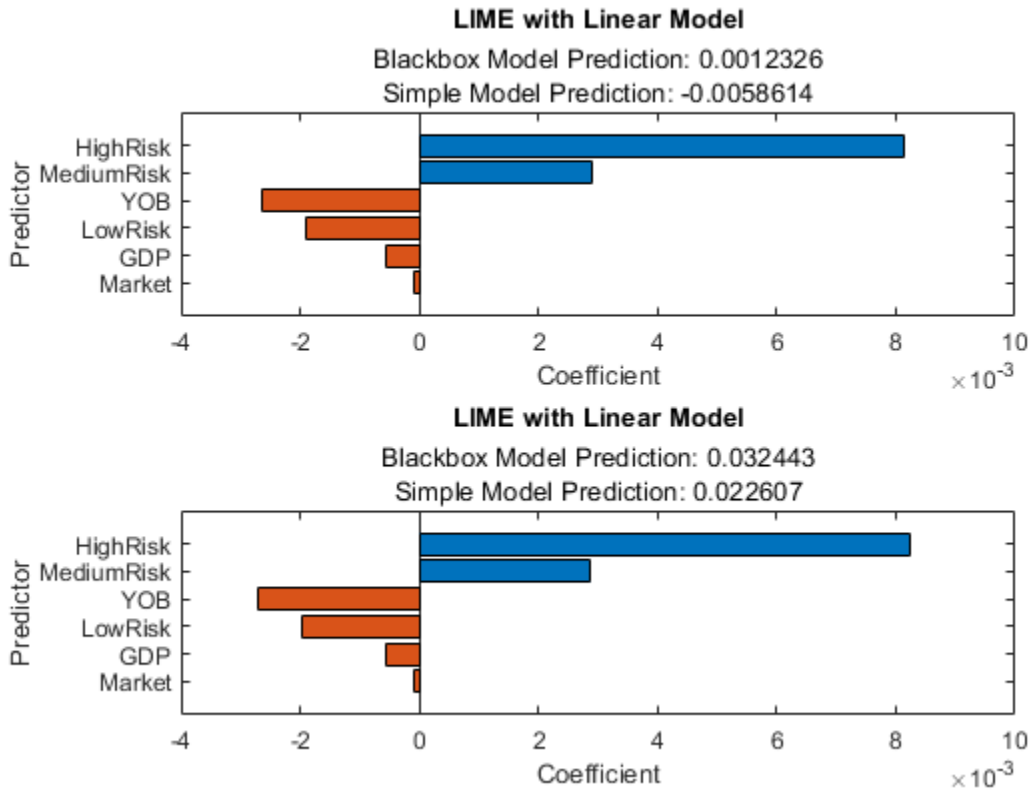
```
explainerLIME = lime(blackboxFcn,predictorData,'Type','regression');
```

Choose a number of important predictors of interest and fit a local model around the selected observations. For illustration purposes, the model contains all of the predictors.

```
numImportantPredictors = 6;
explainerObs1 = fit(explainerLIME,obs1,numImportantPredictors);
explainerObs2 = fit(explainerLIME,obs2,numImportantPredictors);
```

Plot the importance for each predictor.

```
figure
subplot(2,1,1)
plot(explainerObs1);
subplot(2,1,2)
plot(explainerObs2);
```



The `lime` (Statistics and Machine Learning Toolbox) results are quite similar for both observations. The information in the plots show that the most important variables are the `High Risk` and `Medium Risk` variables. `High Risk` and `Medium Risk` contribute positively to higher probabilities of default. On the other hand, `YOB`, `LowRisk`, `GDP`, and `Market` have a negative contribution to the default probability. The `Market` variable does not seem to contribute as much as the other variables. The values in the plots are coefficients of a simple model fitted around the point of interest, so the values can be interpreted as sensitivities of the PD to the different predictors, and these results seem to

align with expectations. For example, PD predictions decrease as the `YOB` value (age of the loan) increases, consistent with the downward trend observed in the model fit plot in the Test Network on page 17-0 section.

Create `shapley` Object

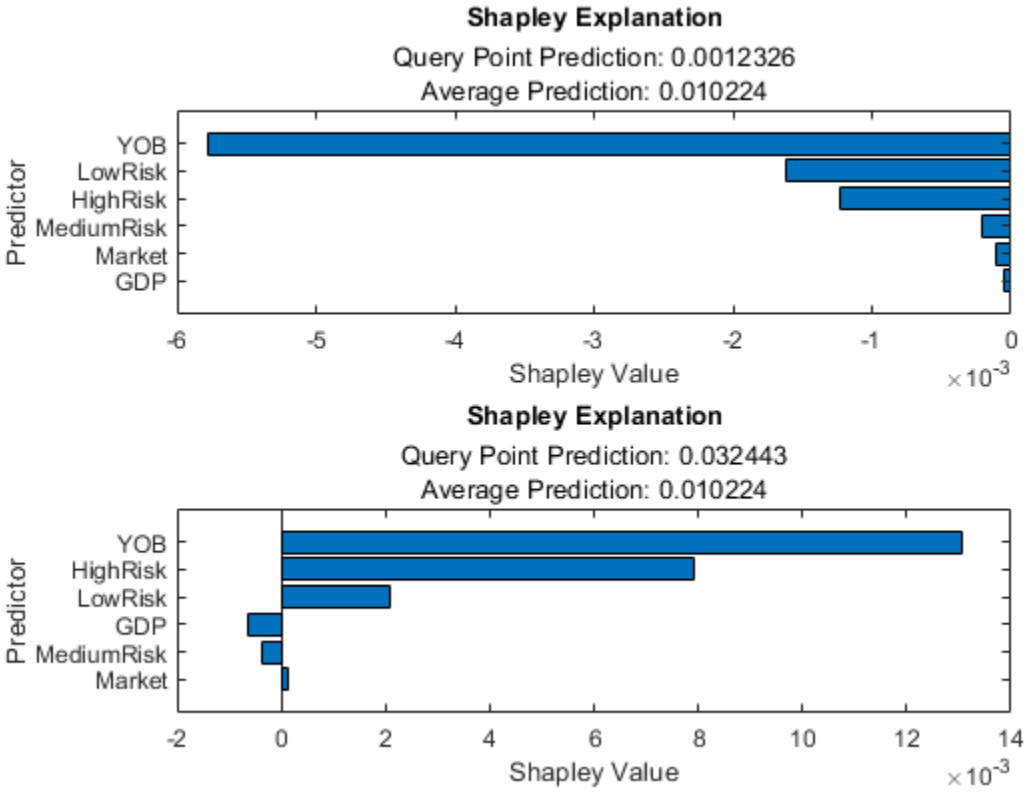
The steps for creating a `shapley` (Statistics and Machine Learning Toolbox) object are the same as for `lime` (Statistics and Machine Learning Toolbox). Create a `shapley` (Statistics and Machine Learning Toolbox) object by passing the black-box function handle and the predictor data selected previously.

The `shapley` (Statistics and Machine Learning Toolbox) analysis can also be affected by randomly generated data, and it requires different methods to control the simulations required for the analysis. For illustration purposes, create the `shapley` (Statistics and Machine Learning Toolbox) object with default settings.

```
explainerShapley = shapley(blackboxFcn,predictorData);
```

Find and plot the importance of predictors for each query point. `shapley` (Statistics and Machine Learning Toolbox) is more computationally intensive than `lime` (Statistics and Machine Learning Toolbox). As the number of rows in the predictor data increases, the computational time for the `shapley` (Statistics and Machine Learning Toolbox) results increases. For large data sets, using parallel computing is recommended (see the `'UseParallel'` option in `shapley` (Statistics and Machine Learning Toolbox)).

```
explainerShapleyObs1 = fit(explainerShapley, obs1);
explainerShapleyObs2 = fit(explainerShapley, obs2);
figure;
subplot(2,1,1)
plot(explainerShapleyObs1)
subplot(2,1,2)
plot(explainerShapleyObs2)
```



In this case, the results look different for the two observations. The `shapley` (Statistics and Machine Learning Toolbox) results explain the deviations from the average PD prediction. For the first observation, which is a very low risk observation, the predicted value is well below the average PD Therefore, all `shapley` (Statistics and Machine Learning Toolbox) values are negative, with `YOB` being the most important variable in this case, followed by `LowRisk`. For the second observation, which is a very high risk observation, most `shapley` (Statistics and Machine Learning Toolbox) values are positive, with `YOB` and `HighRisk` as the main contributors to a predicted PD well above average.

Explore Out-of-Sample Model Predictions

Splitting the original data set into training, validation, and testing helps prevent overfitting. However, the validation and test data sets share similar characteristics with the training data, for example, the range of values for `YOB`, or the observed values for the macroeconomic variables.

```
rangeYOB = [min(data.YOB) max(data.YOB)]
```

```
rangeYOB = 1×2

     1     8
```

```
rangeGDP = [min(data.GDP) max(data.GDP)]
```

```
rangeGDP = 1×2

   -0.5900    3.5700
```

```
rangeMarket = [min(data.Market) max(data.Market)]
```

```
rangeMarket = 1×2

  -22.9500   26.2400
```

You can explore the behavior of the out-of-sample (OOS) model in two different ways. First, you can predict for age values (`YOB` variable) larger than the maximum age value observed in the data. You can predict `YOB` values up to `15`. Second, you can predict for economic conditions not observed in the data either. This example uses two extremely severe macroeconomic situations, where both the `GDP` and `Market` values are very negative and outside the range of values in the data.

Start by setting up a baseline scenario where the last macroeconomic data in the sample is used as reference. The `YOB` values go out of sample for all scenarios.

```
dataBaseline = table;
dataBaseline.YOB = repmat((1:15)',3,1);
dataBaseline.GDP = zeros(size(dataBaseline.YOB));
dataBaseline.Market = zeros(size(dataBaseline.YOB));
dataBaseline.HighRisk = zeros(size(dataBaseline.YOB));
dataBaseline.MediumRisk = zeros(size(dataBaseline.YOB));
dataBaseline.LowRisk = zeros(size(dataBaseline.YOB));

dataBaseline.GDP(:) = data.GDP(8);
dataBaseline.Market(:) = data.Market(8);
dataBaseline.HighRisk(1:15) = 1;
dataBaseline.MediumRisk(16:30) = 1;
dataBaseline.LowRisk(31:45) = 1;

disp(head(dataBaseline))
```

YOB	GDP	Market	HighRisk	MediumRisk	LowRisk
1	1.85	9.48	1	0	0
2	1.85	9.48	1	0	0
3	1.85	9.48	1	0	0
4	1.85	9.48	1	0	0
5	1.85	9.48	1	0	0
6	1.85	9.48	1	0	0
7	1.85	9.48	1	0	0
8	1.85	9.48	1	0	0

Create two new extreme scenarios that include out-of-sample values not only for `YOB`, but also for the macroeconomic variables. This example uses pessimistic scenarios, but you could repeat the analysis for optimistic situations to explore the behavior of the model in either kind of extreme situation.

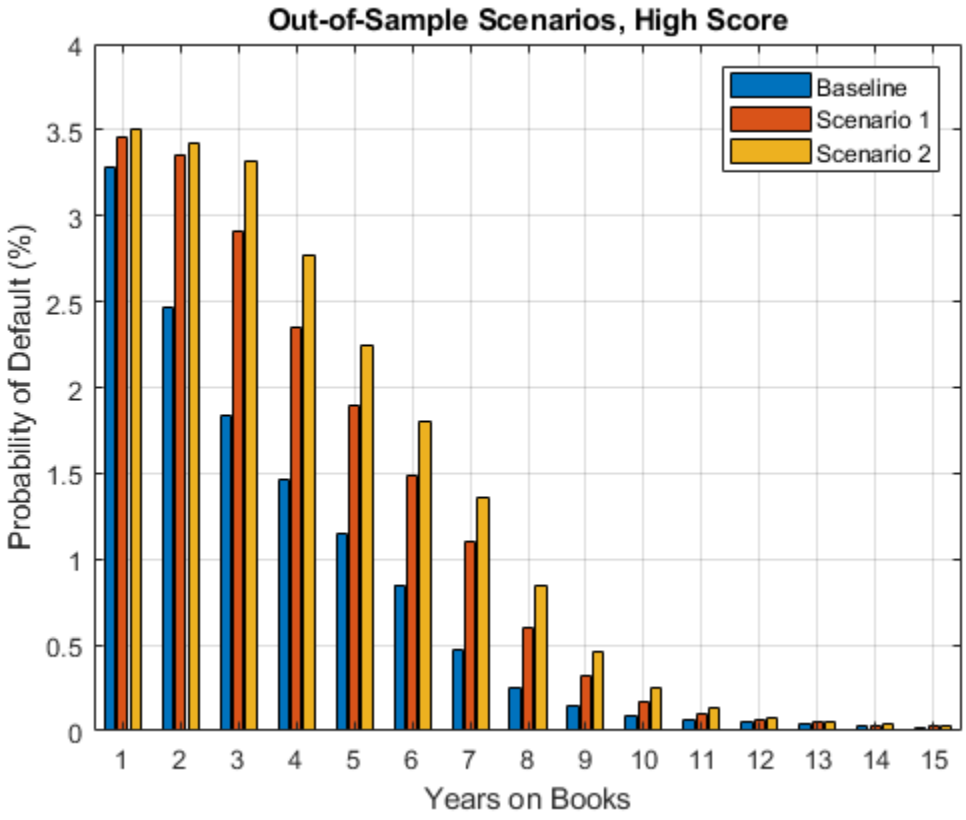
```
dataExtremeS1 = dataBaseline;
dataExtremeS1.GDP(:) = -1;
dataExtremeS1.Market(:) = -25;
dataExtremeS2 = dataBaseline;
dataExtremeS2.GDP(:) = -2;
dataExtremeS2.Market(:) = -40;
```

Predict PD values for all scenarios using `predict`.

```
dataBaseline.PD = predict(residualNetMacro,dataBaseline);
dataExtremeS1.PD = predict(residualNetMacro,dataExtremeS1);
dataExtremeS2.PD = predict(residualNetMacro,dataExtremeS2);
```

Visualize the results for a selected score. For convenience, the average of the PD values over the three scores is visualized as a summary.

```
ScoreSelected = High;
switch ScoreSelected
   case 'High'
      ScoreInd = dataBaseline.HighRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Medium'
      ScoreInd = dataBaseline.MediumRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Low'
      ScoreInd = dataBaseline.LowRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Average'
      PredPDYOBBase = groupsummary(dataBaseline,'YOB','mean','PD');
      PredPDYOBS1 = groupsummary(dataExtremeS1,'YOB','mean','PD');
      PredPDYOBS2 = groupsummary(dataExtremeS2,'YOB','mean','PD');
      PredPDYOB = [PredPDYOBBase.mean_PD PredPDYOBS1.mean_PD PredPDYOBS2.mean_PD];
end

figure;
bar(PredPDYOB*100);
xlabel('Years on Books')
ylabel('Probability of Default (%)')
legend('Baseline','Scenario 1','Scenario 2')
title(strcat("Out-of-Sample Scenarios, ",ScoreSelected," Score"))
grid on
```



The overall results are in line with expectations, since the PD values decrease as the `YOB` value increases, and worse economic conditions result in higher PD values. However, the relative increase of the predicted PD values shows an interesting result. For `Low` and `Medium` scores, there is a significant increase for the first year on books (`YOB = 1`). In contrast, for `High` scores, the relative increase from baseline, to the first extreme scenario, then to the second extreme case, is small. This result suggests an implicit upper limit in the predicted values in the structure of the model. The extreme scenarios in this exercise seem unlikely to occur, however, for extreme but plausible scenarios, this behavior would require investigation with stress testing.

Stress-Test Predicted Probabilities of Default (PD)

Because the model includes macroeconomic variables, it can be used to perform a stress-testing analysis (see for example [2 on page 17-0], [3 on page 17-0] on page 17-0 , [4 on page 17-0]). The steps are similar to the previous section except that the scenarios are plausible scenarios set periodically at an institution level, or set by regulators to be used by all institutions.

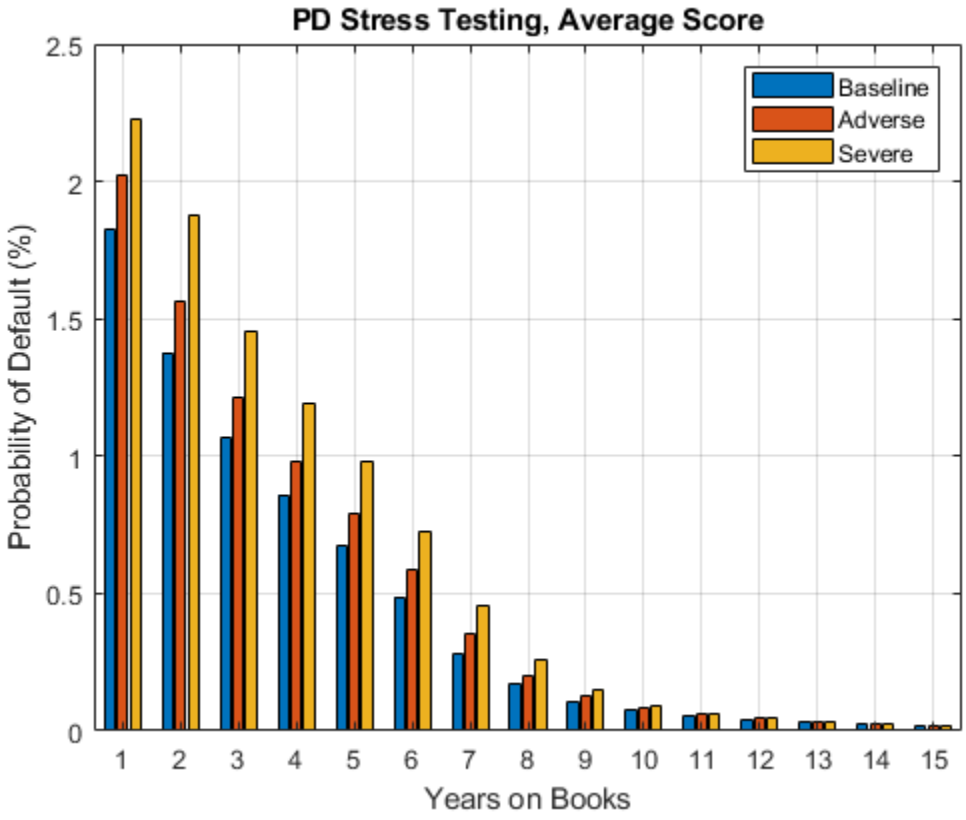
The `dataMacroStress` data set contains three scenarios for the stress testing of the model, namely, baseline, adverse, and severely adverse scenarios. The adverse and severe scenarios are relative to the baseline scenario, and the macroeconomic conditions are are plausible given the baseline. These scenarios fall within the range of values observed in the data used for training and validation. The stress testing of the PD values for given macroeconomic scenarios is conceptually different from the exercise in the previous section, where the focus is on exploring the behavior of the model on out-of-sample data, regardless of how plausible those extreme scenarios are from an economic point of view.

Following the prior steps, you generate PD predictions for each score level and each scenario.

```
dataBaselineStress = dataBaseline(:,1:end-1);
dataAdverse = dataBaselineStress;
dataSevere = dataBaselineStress;

dataBaselineStress.GDP(:) = dataMacroStress{'Baseline','GDP'};
dataBaselineStress.Market(:) = dataMacroStress{'Baseline','Market'};

dataAdverse.GDP(:) = dataMacroStress{'Adverse','GDP'};
dataAdverse.Market(:) = dataMacroStress{'Adverse','Market'};

dataSevere.GDP(:) = dataMacroStress{'Severe','GDP'};
dataSevere.Market(:) = dataMacroStress{'Severe','Market'};
```

Use the `predict` function to predict PD values for all scenarios. Visualize the results for a selected score.

```
dataBaselineStress.PD = predict(residualNetMacro,dataBaselineStress);
dataAdverse.PD = predict(residualNetMacro,dataAdverse);
dataSevere.PD = predict(residualNetMacro,dataSevere);

ScoreSelected = Average;
switch ScoreSelected
   case 'High'
      ScoreInd = dataBaselineStress.HighRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(S
   case 'Medium'
      ScoreInd = dataBaselineStress.MediumRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(S
   case 'Low'
      ScoreInd = dataBaselineStress.LowRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(S
   case 'Average'
      PredPDYOBBaseStress = groupsummary(dataBaselineStress,'YOB','mean','PD');
      PredPDYOBAdverse = groupsummary(dataAdverse,'YOB','mean','PD');
      PredPDYOBSevere = groupsummary(dataSevere,'YOB','mean','PD');
      PredPDYOBStress = [PredPDYOBBaseStress.mean_PD PredPDYOBAdverse.mean_PD PredPDYOBSevere.mea
end

figure;
bar(PredPDYOBStress*100);
xlabel('Years on Books')
ylabel('Probability of Default (%)')
legend('Baseline','Adverse','Severe')
title(strcat("PD Stress Testing, ",ScoreSelected," Score"))
grid on
```



The overall results are in line with expectations. As in the Explore Out-of-Sample Model Predictions on page 17-0 section, the predictions for the `High` score in the first year on books (`YOB = 1`) needs to be reviewed, since the relative increase in the predicted PD from one scenario to the next seems smaller than for other scores and loan ages. All other predictions show a reasonable pattern that are consistent with expectations.

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 770–778, 2016.

[2] Federal Reserve, Comprehensive Capital Analysis and Review (CCAR): https://www.federalreserve.gov/bankinforeg/ccar.htm

[3] Bank of England, Stress Testing: https://www.bankofengland.co.uk/financial-stability

[4] European Banking Authority, EU-Wide Stress Testing: https://www.eba.europa.eu/risk-analysis-and-data/eu-wide-stress-testing

See Also

`trainNetwork` | `trainingOptions` | `fullyConnectedLayer` | **Deep Network Designer** | `featureInputLayer`

Related Examples

- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Train Convolutional Neural Network for Regression” on page 3-53

More About

- “Deep Learning in MATLAB” on page 1-2
- “Specify Layers of Convolutional Neural Network” on page 1-31
- “List of Deep Learning Layers” on page 1-21

Hedging an Option Using Reinforcement Learning Toolbox

This example shows how to learn an optimal option hedging policy and outperform the traditional BSM approach using Reinforcement Learning Toolbox™ .

Option Modeling Using Black-Scholes-Merton Model

The Black-Scholes-Merton (BSM) model, which earned its creators a Nobel Prize in Economics in 1997, provides a modeling framework for pricing and analyzing financial derivatives or options. Options are financial instruments that derive their value from a particular underlying asset. The concept of *dynamic hedging* is fundamental to the BSM model. Dynamic hedging is the idea that, by continuously buying and selling shares in the relevant underlying asset, you can hedge the risk of the derivative instrument such that the risk is zero. This "risk-neutral" pricing framework is used to derive pricing formulae for many different financial instruments.

The simplest financial derivative is a European call option, which provides the buyer with the right, but not the obligation, to buy the underlying asset at a previously specified value (strike price) at a previously specified time (maturity).

You can use a BSM model to price a European call option. The BSM model makes the following simplifying assumptions:

- The behavior of the underlying asset is defined by geometric Brownian motion (GBM).
- There are no transaction costs.
- Volatility is constant.

The BSM dynamic hedging strategy is also called "delta-hedging," after the quantity *Delta*, which is the sensitivity of the option with respect to the underlying asset. In an environment that meets the previously stated BSM assumptions, using a delta-hedging strategy is an optimal approach to hedging an option. However, it is well-known that in an environment with transaction costs, the use of the BSM model leads to an inefficient hedging strategy. The goal of this example is to use Reinforcement Learning Toolbox™ to learn a strategy that outperforms the BSM hedging strategy, in the presence of transaction costs.

The goal of reinforcement learning (RL) is to train an agent to complete a task within an unknown environment. The agent receives observations and a reward from the environment and sends actions to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

The agent contains two components: a policy and a learning algorithm.

- The policy is a mapping that selects actions based on the observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the learning algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.

In other words, reinforcement learning involves an agent learning the optimal behavior through repeated trial-and-error interactions with the environment without human involvement. For more information on reinforcement learning, see “What Is Reinforcement Learning?” (Reinforcement Learning Toolbox).

Cao [2 on page 17-0] describes the setup for reinforcement learning as:

- S_i is the state at time i.
- A_i is the action taken at i.
- R_{i+1} is the resulting reward at time $i + 1$.

The aim of reinforcement learning is to maximize expected future rewards. In this financial application of reinforcement learning, maximizing expected rewards is learning a delta-hedging strategy as an optimal approach to hedging a European call option.

This example follows the framework outlined in Cao [2 on page 17-0]. Specifically, an accounting profit and loss (P&L) formulation from that paper is used to set up the reinforcement learning problem and a deep deterministic policy gradient (DDPG) agent is used. This example does not exactly reproduce the approach from [2 on page 17-0] because Cao *et. al.* recommend a Q-learning approach with two separate Q-functions (one for the hedging cost and one for the expected square of the hedging cost), but this example uses instead a simplified reward function.

Define Training Parameters

Next, specify an at-the-money option with three months to maturity is hedged. For simplicity, both the interest rate and dividend yield are set to `0`.

```
% Option parameters
Strike = 100;
Maturity = 21*3/250;

% Asset parameters
SpotPrice = 100;
ExpVol = .2;
ExpReturn = .05;

% Simulation parameters
rfRate = 0;
dT = 1/250;
nSteps = Maturity/dT;
nTrials = 5000;

% Transacation cost and cost function parameters
c = 1.5;
kappa = .01;
InitPosition = 0;

% Set the random generator seed for reproducibility.
rng(3)
```

Define Environment

In this section, the action and observation parameters, `actInfo` and `obsInfo`. The agent action is the current `hedge` value which can range between `0` and `1`. There are three variables in the agent observation:

- Moneyness (ratio of the spot price to the strike price)
- Time to maturity
- Position or amount of the underlying asset that is held

```
ObservationInfo             = rlNumericSpec([3 1],'LowerLimit',0,'UpperLimit',[10 Maturity 1]');
ObservationInfo.Name        = 'Hedging State';
ObservationInfo.Description = ['Moneyness', 'TimeToMaturity','Position'];

ActionInfo = rlNumericSpec([1 1],'LowerLimit',0,'UpperLimit',1);
ActionInfo.Name = 'Hedge';
```

Define Reward

From Cao [2 on page 17-0], the accounting P&L formulation and rewards (negative costs) are

$R_{i+1} = V_{i+1} - V_i + H_{i+1}(S_{i+1} - S_i) - \kappa|S_{i+1}(H_{i+1} - H_i)|$

where

R_i: Reward

V_i: Value of option

S_i: Spot price of underlying asset

H_i: Holding

κ: Transaction costs

A final reward at the last time step liquidates the hedge that is $\kappa|S_n(H_n)|$.

In this implementation, the reward (R_i) is penalized by the square of the reward multiplied by a constant to punish large swings in the value of the hedged position:

$R_{i+1} = R_{i+1} - c(R_{i+1})^2$

The reward is defined in `stepFcn` which is called at each step of the simulation.

```
env = rlFunctionEnv(ObservationInfo,ActionInfo, ...
    @(Hedge, LoggedSignals) stepFcn(Hedge,LoggedSignals,rfRate,ExpVol,dT,Strike,ExpReturn,c,kappa
    @() resetFcn(SpotPrice/Strike,Maturity,InitPosition));

obsInfo = getObservationInfo(env);
actInfo = getActionInfo(env);
```

Create Environment Interface for RL Agent

Create the DDPG agent using `rlDDPGAgent` (Reinforcement Learning Toolbox). While it is possible to create custom actor and critic networks, this example uses the default networks.

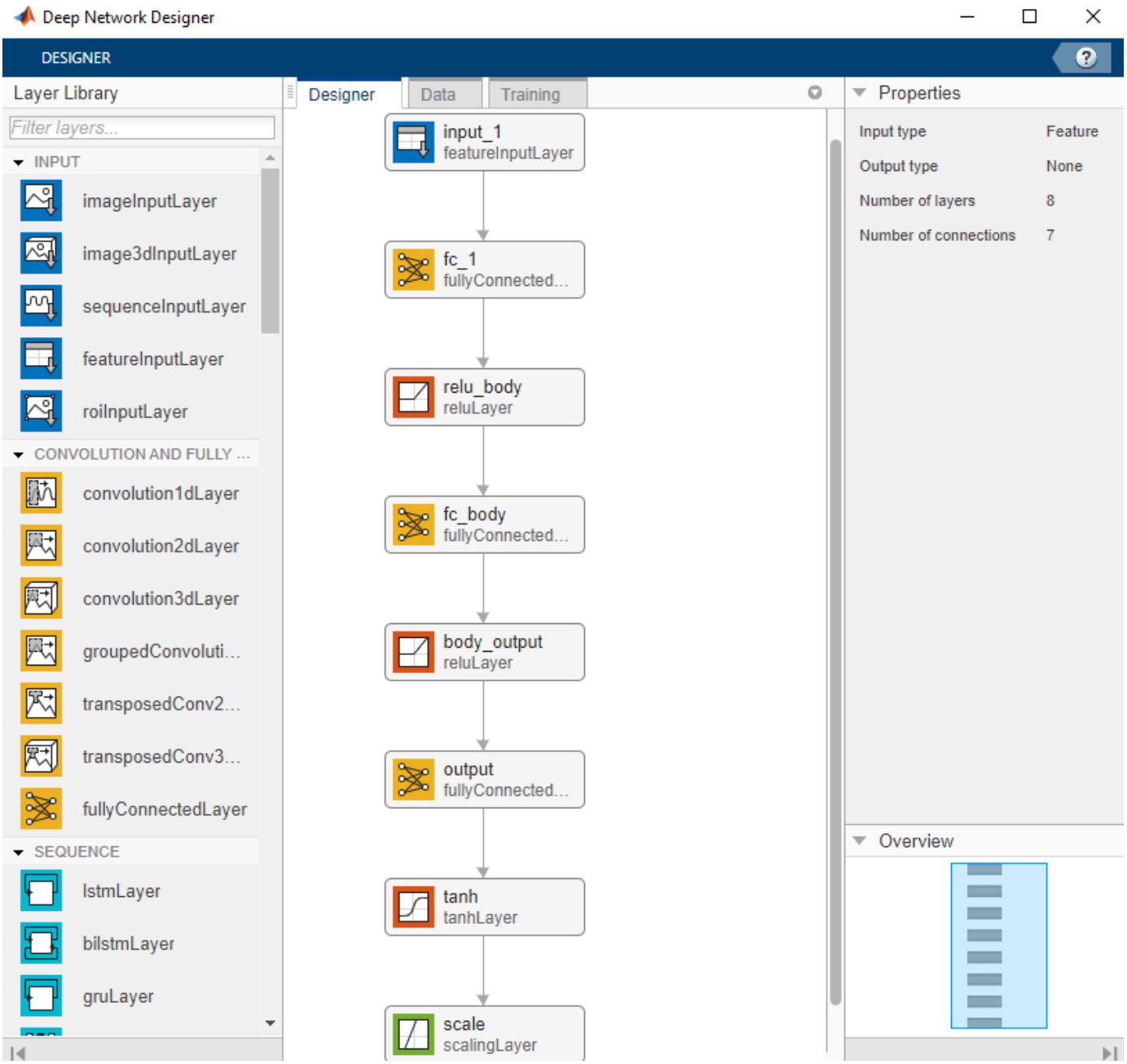
```
initOpts = rlAgentInitializationOptions('NumHiddenUnit',64);

agent = rlDDPGAgent(obsInfo,actInfo,initOpts);

critic = getCritic(agent);
critic.Options.LearnRate = 1e-4;
% critic.Options.UseDevice = "gpu";
agent  = setCritic(agent,critic);

actor = getActor(agent);
```

```
actor.Options.LearnRate = 1e-4;
% actor.Options.UseDevice = "gpu";
agent  = setActor(agent,actor);

agent.AgentOptions.DiscountFactor = .9995;
agent.AgentOptions.TargetSmoothFactor = 5e-4;
```

Visualize Actor and Critic Networks

Visualize the actor and critic networks using the Deep Network Designer.

```
deepNetworkDesigner(layerGraph(getModel(actor)))
```

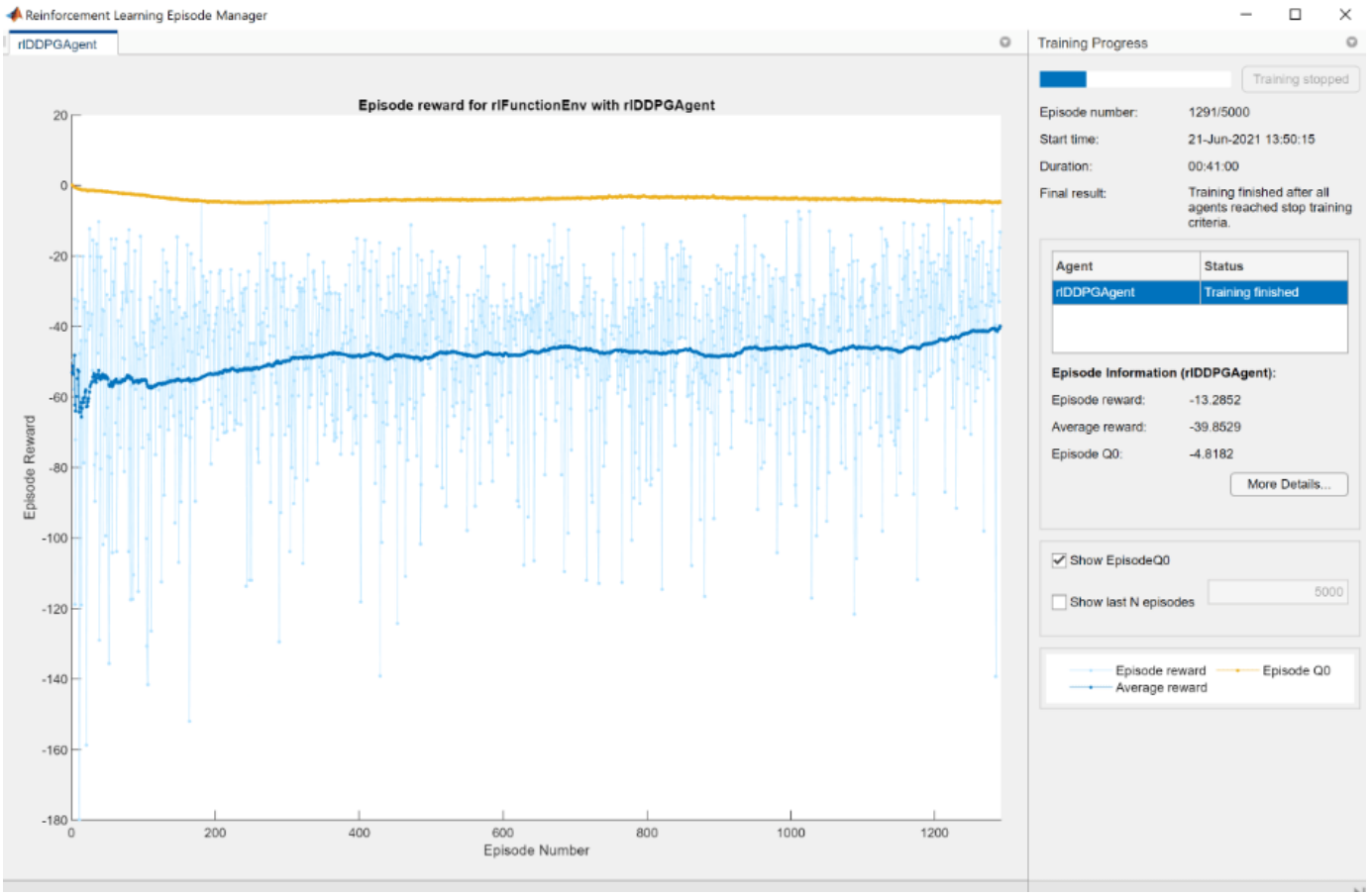


Train Agent

Train the agent using the `train` (Reinforcement Learning Toolbox) function.

```
trainOpts = rlTrainingOptions( ...
    'MaxEpisodes', nTrials, ...
    'MaxStepsPerEpisode',nSteps, ...
    'Verbose', false, ...
    'ScoreAveragingWindowLength',200,...
    'StopTrainingCriteria',"AverageReward", ...
    'StopTrainingValue',-40, ...
    'StopOnError',"on", ...
    "UseParallel",false);

doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load the pretrained agent for the example.
    load('DeepHedgingDDPG.mat','agent')
end
```

To avoid waiting for the training, load pretrained networks by setting the `doTraining` flag to `false`. If you set `doTraining` to `true`, the Reinforcement Learning Episode Manager displays the training progress.



Validate Agent

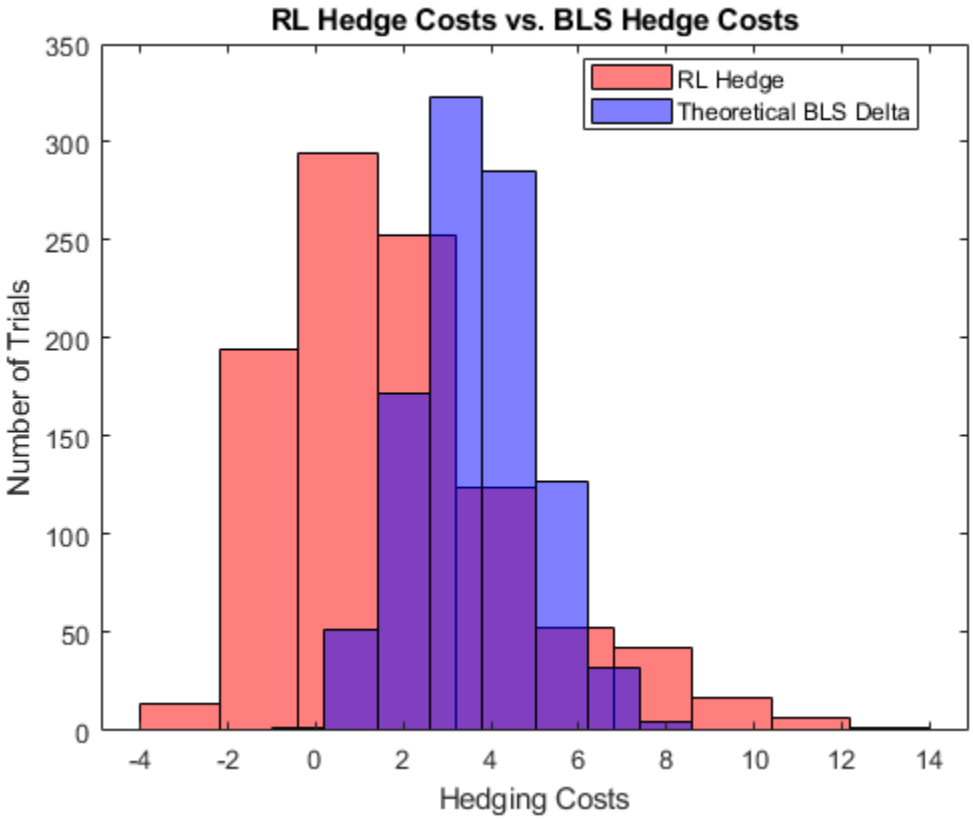
Use the Financial Toolbox™ functions `blsdelta` (Financial Toolbox) and `blsprice` (Financial Toolbox) for a coventional approach to caculate the price as a European call option. When comparing the conventional approach to the RL approach, the results are similar to the findings of Cao [2 on page 17-0] in Exhibit 4. This example demonstrates that the RL approach significantly reduces hedging costs.

```
% Simulation parameters
nTrials = 1000;

policy_BSM = @(mR,TTM,Pos) blsdelta(mR,1,rfRate,max(TTM,eps),ExpVol);
policy_RL = @(mR,TTM,Pos) arrayfun(@(mR,TTM,Pos) cell2mat(getAction(agent,[mR TTM Pos])),mR,TTM,

OptionPrice = blsprice(SpotPrice,Strike,rfRate,Maturity,ExpVol);

Costs_BSM = computeCosts(policy_BSM,nTrials,nSteps,SpotPrice,Strike,Maturity,rfRate,ExpVol,InitPo
Costs_RL = computeCosts(policy_RL,nTrials,nSteps,SpotPrice,Strike,Maturity,rfRate,ExpVol,InitPosi

HedgeComp = table(100*[-mean(Costs_BSM) std(Costs_BSM)]'/OptionPrice, ...
    100*[-mean(Costs_RL) std(Costs_RL)]'/OptionPrice, ...
    'RowNames',["Average Hedge Cost (% of Option Price)","STD Hedge Cost (% of Option Price)"],
    'VariableNames',["BSM","RL"]);
disp(HedgeComp)
```

```
                                          BSM        RL
                                        _______    _______

    Average Hedge Cost (% of Option Price)    91.259    47.022
    STD Hedge Cost (% of Option Price)        35.712    68.119
```

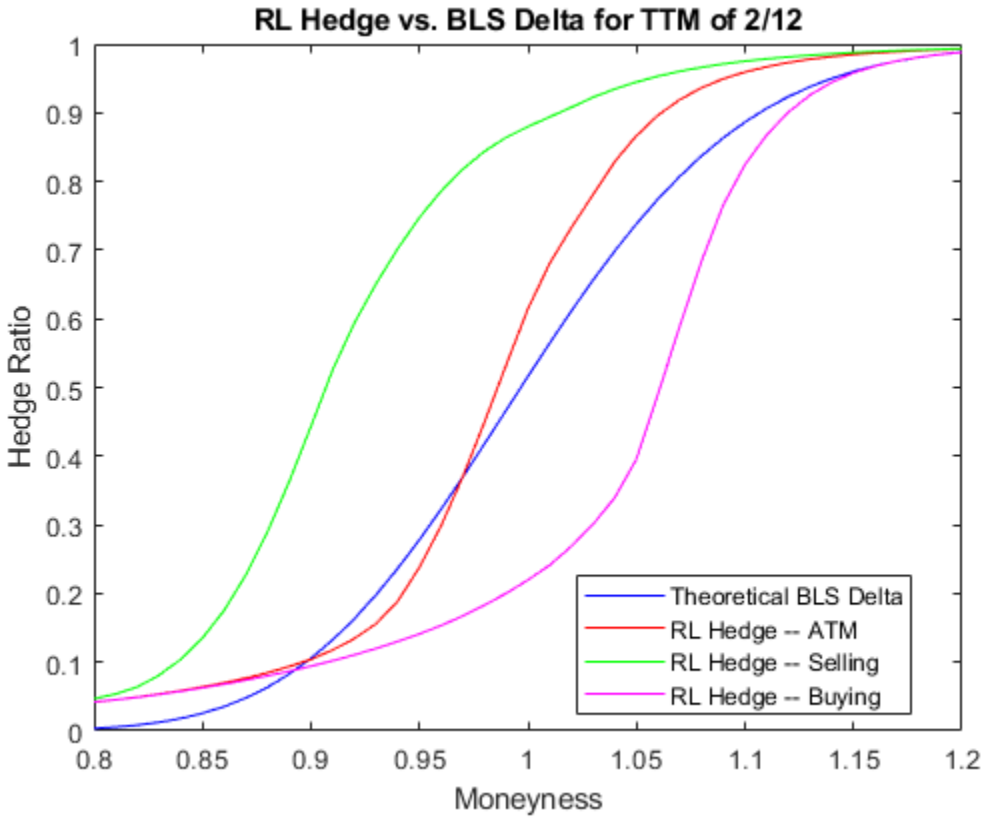
The following histogram shows the range of different hedging costs for both approaches. The RL approach performs better, but with a larger variance than the BSM approach. The RL appraoch in this example would likely benefit from the two Q-function approach that Cao [2 on page 17-0] discusses and implements.

```
figure
numBins = 10;
histogram(-Costs_RL,numBins,'FaceColor','r','FaceAlpha',.5)
hold on
histogram(-Costs_BSM,numBins,'FaceColor','b','FaceAlpha',.5)
xlabel('Hedging Costs')
ylabel('Number of Trials')
title('RL Hedge Costs vs. BLS Hedge Costs')
legend('RL Hedge','Theoretical BLS Delta','location','best')
```



A plot of the hedge Ratio with respect to moneyness shows the differences between the BSM and RL approaches. As discussed in Cao [2 on page 17-0], in the presence of transaction costs, the agent learns that "when delta hedging would require shares to be purchased, it tends to be optimal for a trader to be underhedged relative to delta. Similarly, when delta hedging would require shares to be sold, it tends to be optimal for a trader to be over-hedged relative to delta."

```
policy_RL_mR = @(mR,TTM,Pos) cell2mat(getAction(agent,[mR TTM Pos]));
mRange = (.8:.01:1.2)';

figure
t_plot = 2/12;
plot(mRange,blsdelta(mRange,1,rfRate,t_plot,ExpVol),'b')
hold on
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR,1,rfRate,t_plot,ExpVol)),mRange),'
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR+.1,1,rfRate,t_plot,ExpVol)),mRange
plot(mRange,arrayfun(@(mR) policy_RL_mR(mR,t_plot,blsdelta(mR-.1,1,rfRate,t_plot,ExpVol)),mRange
legend('Theoretical BLS Delta','RL Hedge -- ATM','RL Hedge -- Selling','RL Hedge -- Buying', ...
    'location','best')
xlabel('Moneyness')
ylabel('Hedge Ratio')
title('RL Hedge vs. BLS Delta for TTM of 2/12')
```



References

[1] Buehler H., L. Gonon, J. Teichmann, and B. Wood. "Deep hedging." *Quantitative Finance*. Vol. 19, No. 8, 2019, pp. 1271–91.

[2] Cao J., J. Chen, J. Hull, and Z. Poulos. "Deep Hedging of Derivatives Using Reinforcement Learning." *The Journal of Financial Data Science*. Vol. 3, No. 1, 2021, pp. 10–27.

[3] Halperin I. "QLBS: Q-learner in the Black-Scholes (-Merton) Worlds." *The Journal of Derivatives*. Vol. 28, No. 1, 2020, pp. 99–122.

[4] Kolm P.N. and G. Ritter. "Dynamic Replication and Hedging: A Reinforcement Learning Approach." *The Journal of Financial Data Science*. Vol. 1, No. 1, 2019, pp. 159–71.

Local Functions

```
function [InitialObservation,LoggedSignals] = resetFcn(Moneyness,TimeToMaturity,InitPosition)
% Reset function to reset at the beginning of each episode.

LoggedSignals.State = [Moneyness TimeToMaturity InitPosition]';
InitialObservation  = LoggedSignals.State;

end

function [NextObs,Reward,IsDone,LoggedSignals] = stepFcn(Position_next,LoggedSignals,r,vol,dT,X,r
% Step function to evaluate at each step of the episode.
```

```
Moneyness_prev = LoggedSignals.State(1);
TTM_prev = LoggedSignals.State(2);
Position_prev = LoggedSignals.State(3);

S_prev = Moneyness_prev*X;

% GBM Motion
S_next = S_prev*((1 + mu*dT) + (randn* vol).*sqrt(dT));
TTM_next = max(0,TTM_prev - dT);

IsDone = TTM_next < eps;

stepReward = (S_next - S_prev)*Position_prev - abs(Position_next - Position_prev)*S_next*kappa -
    blsprice(S_next,X,r,TTM_next,vol) + blsprice(S_prev,X,r,TTM_prev,vol);

if IsDone
    stepReward = stepReward - Position_next*S_next*kappa;
end

Reward = stepReward - c*stepReward.^2;

LoggedSignals.State = [S_next/X;TTM_next;Position_next];
NextObs = LoggedSignals.State;

end

function perCosts = computeCosts(policy,nTrials,nSteps,SpotPrice,Strike,T,r,ExpVol,InitPos,dT,mu
% Helper function to compute costs for any hedging approach.

rng(0)

simOBJ = gbm(mu,ExpVol,'StartState',SpotPrice);
[simPaths,simTimes] = simulate(simOBJ,nSteps,'nTrials',nTrials,'deltaTime',dT);
simPaths = squeeze(simPaths);

rew = zeros(nSteps,nTrials);

Position_prev = InitPos;
Position_next = policy(simPaths(1,:)/Strike,T*ones(1,nTrials),InitPos*ones(1,nTrials));
for timeidx=2:nSteps+1
    rew(timeidx-1,:) = (simPaths(timeidx,:) - simPaths(timeidx-1,:)).*Position_prev - ...
        abs(Position_next - Position_prev).*simPaths(timeidx,:)*kappa - ...
        blsprice(simPaths(timeidx,:),Strike,r,max(0,T - simTimes(timeidx)),ExpVol) + ...
        blsprice(simPaths(timeidx-1,:),Strike,r,T - simTimes(timeidx-1),ExpVol);

    if timeidx == nSteps+1
         rew(timeidx-1,:) =  rew(timeidx-1,:) - Position_next.*simPaths(timeidx,:,:)*kappa;
    else
        Position_prev = Position_next;
        Position_next = policy(simPaths(timeidx,:,:)/Strike,(T - simTimes(timeidx)).*ones(1,nTria
    end
end

perCosts = sum(rew);
```

```
end
```

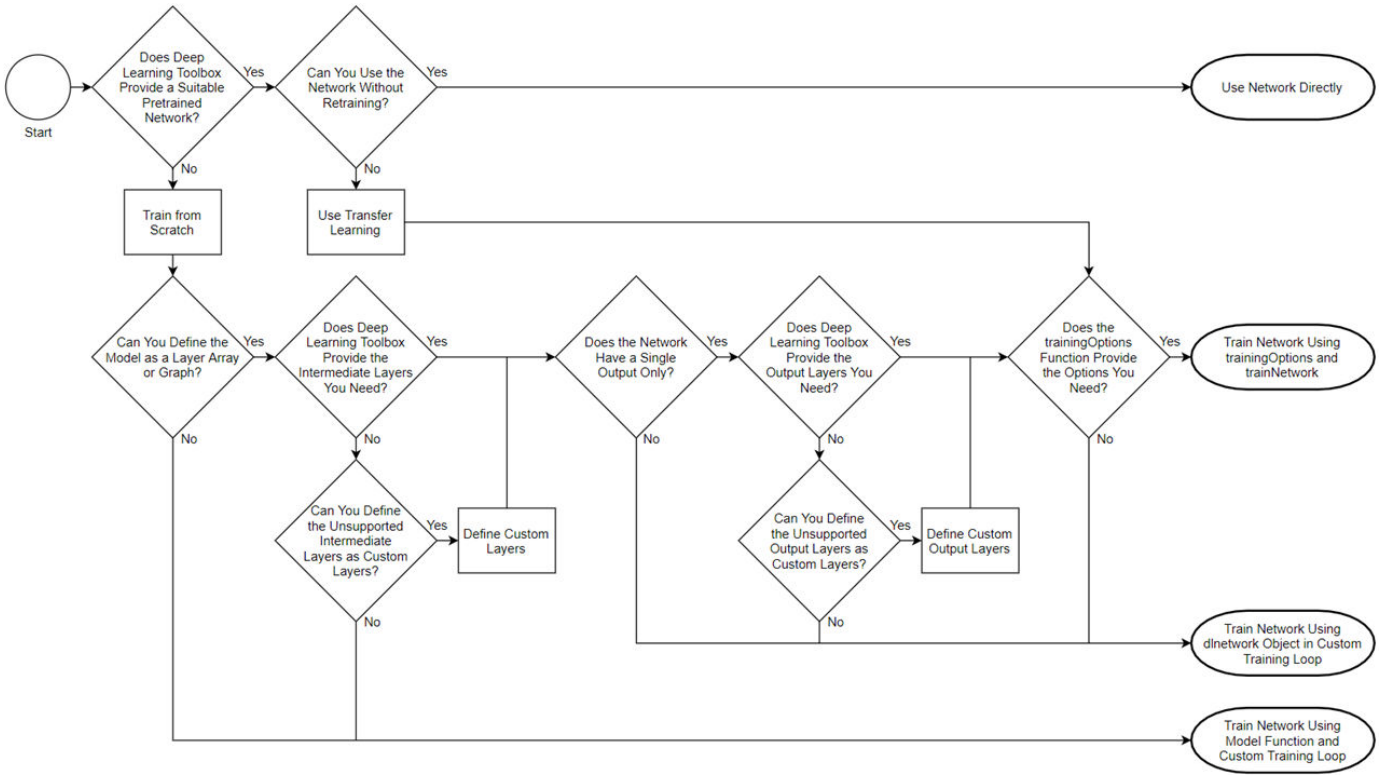
See Also

`blsprice` | `blsdelta` | `rlDDPGAgent` | **Deep Network Designer**

Import, Export, and Customization

Train Deep Learning Model in MATLAB

You can train and customize a deep learning model in various ways—for example, you can retrain a pretrained model with new data (transfer learning), train a network from scratch, or define a deep learning model as a function and use a custom training loop. Use this flow chart to choose the training method that is best suited for your task.



Tip For information on computer vision workflows, including for object detection, see “Computer Vision Using Deep Learning”. For information on importing networks and network architectures from TensorFlow-Keras, Caffe, and the ONNX (Open Neural Network Exchange) model format, see “Deep Learning Import and Export”.

Training Methods

This table provides information about the different training methods.

Method	More Information
Use network directly	If a pretrained network already performs the task you require, then you do not need to retrain the network. Instead, you can make predictions with the network directly by using the `classify` and `predict` functions. For an example, see “Classify Image Using GoogLeNet” on page 3-23.
Train network using `trainingOptions` and `trainNetwork`	If you have a network specified as a layer array or layer graph, and the `trainingOptions` function provides all the options you need, then you can train the network using the `trainNetwork` function. For an example showing how to retrain a network (transfer learning), see “Train Deep Learning Network to Classify New Images” on page 3-6. For an example showing how to train a network from scratch, see “Create Simple Deep Learning Network for Classification” on page 3-47.
Train network using `dlnetwork` object and custom training loop	For most tasks, you can control the training algorithm details using the `trainingOptions` and `trainNetwork` functions. If the `trainingOptions` function does not provide the options you need for your task (for example, a custom learning rate schedule), then you can define your own custom training loop using a `dlnetwork` object. A `dlnetwork` object allows you to train a network specified as a layer graph using automatic differentiation. For loss functions that cannot be specified using an output layer, you can specify the loss in a custom training loop. For an example showing how to train a network with a custom learning rate schedule, see “Train Network Using Custom Training Loop” on page 18-225. To learn, more see “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209.

Method	More Information
Train network using model function and custom training loop	For networks that cannot be created using layer graphs, you can define a custom network as a function. For an example showing how to train a deep learning model defined as a function, see “Train Network Using Model Function” on page 18-259. If you can create parts of the network using a layer graph, then you can define those parts as layer graphs and the unsupported parts using model functions.

Decisions

This table provides more information on each decision in the flow chart.

Decision	More Information
Does Deep Learning Toolbox provide a suitable pretrained network?	For most tasks, you can use or retrain a pretrained network such as `googlenet`. For a list of pretrained deep learning networks in MATLAB, see “Pretrained Deep Neural Networks” on page 1-8. You can use pretrained networks directly with new data, or you can retrain them with new data for different tasks using transfer learning.
Can you use the network without retraining?	If a pretrained network already performs the task you need, then you can use the network directly without retraining. For example, you can use the `googlenet` network to classify images in 1000 classes. To make predictions with the network directly, use the `classify` and `predict` functions. For an example, see “Classify Image Using GoogLeNet” on page 3-23. If you need to retrain the network—for example, to classify a different set of classes—then you can retrain the network using transfer learning.

Decision	More Information
Can you define the model as a layer array or graph?	You can specify most deep learning models as a layer array or layer graph. In other words, you can define the model as a collection of layers with layer outputs connected to other layer inputs. Some network architectures cannot be defined as a layer graph. For example, Siamese networks require weight sharing and cannot be defined as a layer graph. For these networks, you must define the model as a function. For an example, see “Train Network Using Model Function” on page 18-259.
Does the network have a single output only?	For networks with multiple outputs, you must train the network using a custom training loop. For an example, see “Train Network with Multiple Outputs” on page 3-61.
Does Deep Learning Toolbox provide the intermediate layers you need?	Deep Learning Toolbox provides many different layers for deep learning tasks. For a list of layers, see “List of Deep Learning Layers” on page 1-21. If Deep Learning Toolbox provides the intermediate layers (layers in the middle of the network) that you need, then you can define the network as a layer array or layer graph using these layers. Otherwise, try defining any unsupported layers as custom layers. For more information, see “Define Custom Deep Learning Layers” on page 18-9.
Can you define the unsupported intermediate layers as custom layers?	If Deep Learning Toolbox does not provide the layer you need, then you can try defining a custom deep learning layer. For more information, see “Define Custom Deep Learning Layers” on page 18-9. If you can define custom layers for any unsupported layers, then you can include these custom layers in a layer array or layer graph. Otherwise, specify the deep learning model using a function and train the model using a custom training loop. For an example, see “Train Network Using Model Function” on page 18-259.

Decision	More Information
Does Deep Learning Toolbox provide the output layers you need?	Output layers specify the loss function used for training. Deep Learning Toolbox provides different output layers for deep learning tasks. For example, `classificationLayer` and `regressionLayer`. For a list of output layers, see the "Output Layers" on page 1-28 section in the page "List of Deep Learning Layers" on page 1-21. If Deep Learning Toolbox provides the output layers that you need, then you can define a layer graph using these layers. Otherwise, try defining any unsupported output layers as a custom layer. For more information, see "Define Custom Deep Learning Layers" on page 18-9.
Can you define the unsupported output layers as custom layers?	If Deep Learning Toolbox does not provide the output layer you need, then you can try defining a custom output layer. For more information, see "Define Custom Deep Learning Layers" on page 18-9. If you can define a custom output layer for any unsupported output layers, then you can include these custom layers in a layer array or layer graph. Otherwise, train the model using a `dlnetwork` object and a custom training loop, and specify a custom loss function. For an example, see "Train Network Using Custom Training Loop" on page 18-225.
Does the `trainingOptions` function provide the options you need?	The `trainingOptions` function provides many options for customizing a the training process. If the `trainingOptions` function provides all the options you need for training, then you can train the deep learning network using the `trainNetwork` function. For an example, see "Create Simple Deep Learning Network for Classification" on page 3-47. If the `trainingOptions` function does not provide the training option you need, for example, a custom learning rate schedule, then you can define a custom training loop using a `dlnetwork` object. For an example, see "Train Network Using Custom Training Loop" on page 18-225.

See Also

`trainingOptions` | `trainNetwork`

More About

- “Pretrained Deep Neural Networks” on page 1-8
- “Classify Image Using GoogLeNet” on page 3-23
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “List of Deep Learning Layers” on page 1-21
- “Define Custom Deep Learning Layers” on page 18-9
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Model Function” on page 18-259

Define Custom Deep Learning Layers

Tip This topic explains how to define custom deep learning layers for your problems. For a list of built-in layers in Deep Learning Toolbox, see “List of Deep Learning Layers” on page 1-21.

You can define your own custom deep learning layers for your task. You can specify a custom loss function using a custom output layer and define custom layers with or without learnable and state parameters. After defining a custom layer, you can check that the layer is valid, GPU compatible, and outputs correctly defined gradients.

This topic explains the architecture of deep learning layers and how to define custom layers to use for your tasks.

Type	Description
Intermediate layer	Define a custom deep learning layer and specify optional learnable parameters and state parameters. For more information, see “Define Custom Deep Learning Intermediate Layers” on page 18-16. For an example showing how to define a custom layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35. For an example showing how to define a custom layer with multiple inputs, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.
Classification output layer	Define a custom classification output layer and specify a loss function. For more information, see “Define Custom Deep Learning Output Layers” on page 18-29. For an example showing how to define a custom classification output layer and specify a loss function, see “Define Custom Classification Output Layer” on page 18-91.
Regression output layer	Define a custom regression output layer and specify a loss function. For more information, see “Define Custom Deep Learning Output Layers” on page 18-29. For an example showing how to define a custom regression output layer and specify a loss function, see “Define Custom Regression Output Layer” on page 18-99.

Layer Templates

You can use the following templates to define new layers.

Intermediate Layer Template

This template outlines the structure of an intermediate layer. For more information, see “Define Custom Deep Learning Intermediate Layers” on page 18-16.

For an example showing how to define a layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Define layer constructor function here.
        end

        function [Z,state] = predict(layer,X)
            % Forward input data through the layer at prediction time and
            % output the result and updated state.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Input data
            % Outputs:
            %         Z     - Output of layer forward function
            %         state - (Optional) Updated layer state.
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer predict function here.
        end
```

```
        function [Z,state,memory] = forward(layer,X)
            % (Optional) Forward input data through the layer at training
            % time and output the result, updated state, and a memory
            % value.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Layer input data
            % Outputs:
            %         Z      - Output of layer forward function
            %         state  - (Optional) Updated layer state
            %         memory - (Optional) Memory value for custom backward
            %                  function
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer forward function here.
        end

        function layer = resetState(layer)
            % (Optional) Reset layer state.

            % Define reset state function here.
        end

        function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer   - Layer to backward propagate through
            %         X       - Layer input data
            %         Z       - Layer output data
            %         dLdZ    - Derivative of loss with respect to layer
            %                   output
            %         dLdSout - (Optional) Derivative of loss with respect
            %                   to state output
            %         memory  - Memory value from forward function
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Classification Output Layer Template

This template outlines the structure of a classification output layer with a loss function. For more information, see “Define Custom Deep Learning Output Layers” on page 18-29.

For an example showing how to define a classification output layer and specify a loss function, see “Define Custom Classification Output Layer” on page 18-91.

```
classdef myClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    methods
        function layer = myClassificationLayer()
            % (Optional) Create a myClassificationLayer.

            % Layer constructor function goes here.
        end

        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Regression Output Layer Template

This template outlines the structure of a regression output layer with a loss function. For an example showing how to define a regression output layer and specify a loss function, see “Define Custom Regression Output Layer” on page 18-99.

```
classdef myRegressionLayer < nnet.layer.RegressionLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end
```

```
    methods
        function layer = myRegressionLayer()
            % (Optional) Create a myRegressionLayer.

            % Layer constructor function goes here.
        end

        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Intermediate Layer Architecture

During training, the software iteratively performs forward and backward passes through the network.

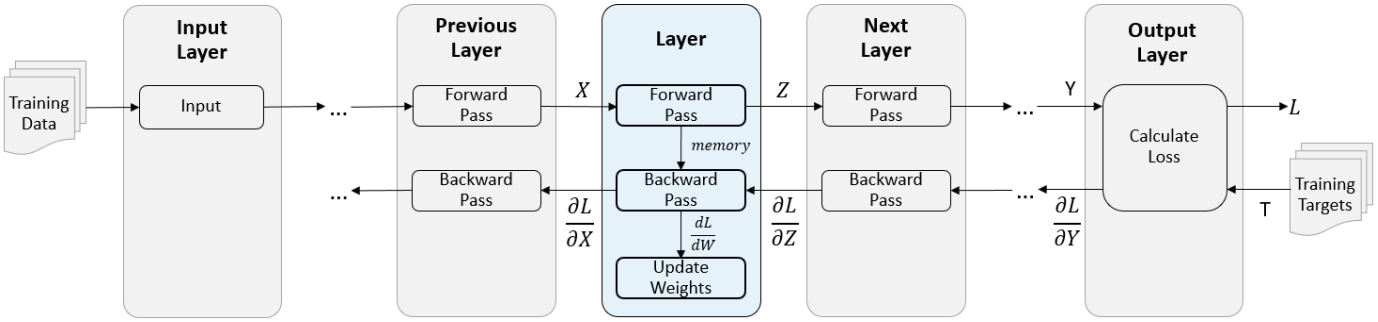
When making a forward pass through the network, each layer takes the outputs of the previous layers, applies a function, and then outputs (forward propagates) the results to the next layers. Stateful layers, such as LSTM layers, also update the layer state.

Layers can have multiple inputs or outputs. For example, a layer can take $X_1, \ldots, X_N$ from multiple previous layers and forward propagate the outputs $Z_1, \ldots, Z_M$ to subsequent layers.

At the end of a forward pass of the network, the output layer calculates the loss L between the predictions Y and the targets T.

During the backward pass of a network, each layer takes the derivatives of the loss with respect to the outputs of the layer, computes the derivatives of the loss L with respect to the inputs, and then backward propagates the results. If the layer has learnable parameters, then the layer also computes the derivatives of the layer weights (learnable parameters). The layer uses the derivatives of the weights to update the learnable parameters.

The following figure describes the flow of data through a deep neural network and highlights the data flow through a layer with a single input X, a single output Z, and a learnable parameter W.

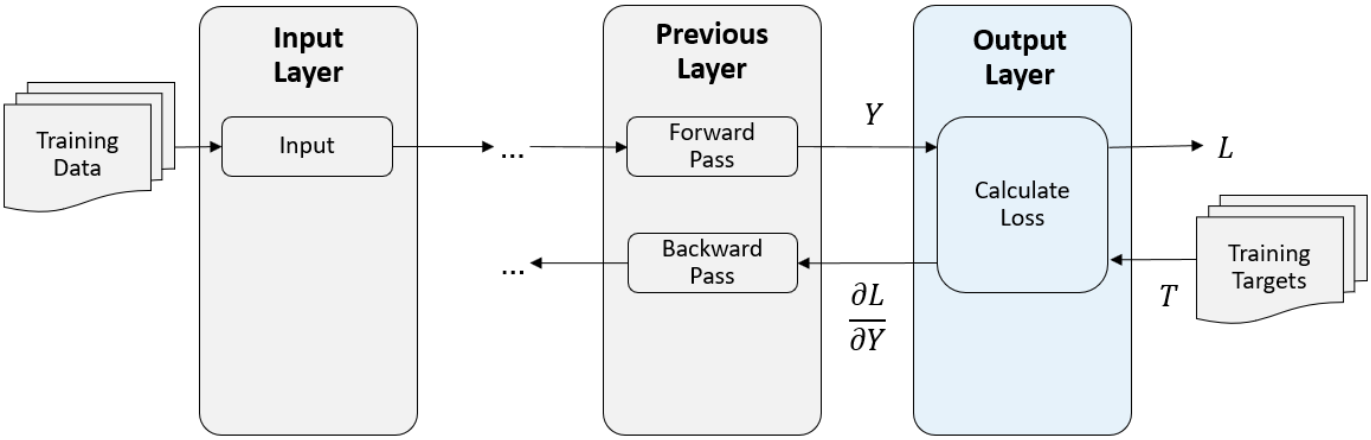


For more information about custom intermediate layers, see “Define Custom Deep Learning Intermediate Layers” on page 18-16.

Output Layer Architecture

At the end of a forward pass at training time, an output layer takes the outputs Y of the previous layer (the network predictions) and calculates the loss L between these predictions and the training targets. The output layer computes the derivatives of the loss L with respect to the predictions Y and outputs (backward propagates) results to the previous layer.

The following figure describes the flow of data through a neural network and an output layer.



For more information, see “Define Custom Deep Learning Output Layers” on page 18-29.

Check Validity of Custom Layer

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, correctly defined gradients, and code generation compatibility. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer. To check with multiple observations, use the `ObservationDimension` option. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to

1 (true). For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

For more information, see “Check Custom Layer Validity” on page 18-154.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Custom Classification Output Layer” on page 18-91
- “Define Custom Regression Output Layer” on page 18-99
- “Deep Learning Network Composition” on page 18-117
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154
- “List of Deep Learning Layers” on page 1-21

Define Custom Deep Learning Intermediate Layers

Tip This topic explains how to define custom deep learning layers for your problems. For a list of built-in layers in Deep Learning Toolbox, see “List of Deep Learning Layers” on page 1-21.

To learn how to define custom output layers, see “Define Custom Deep Learning Output Layers” on page 18-29.

If Deep Learning Toolbox does not provide the layer that you require for your task, then you can define your own custom layer using this topic as a guide. After defining the custom layer, you can automatically check that the layer is valid, GPU compatible, and outputs correctly defined gradients.

Intermediate Layer Architecture

During training, the software iteratively performs forward and backward passes through the network.

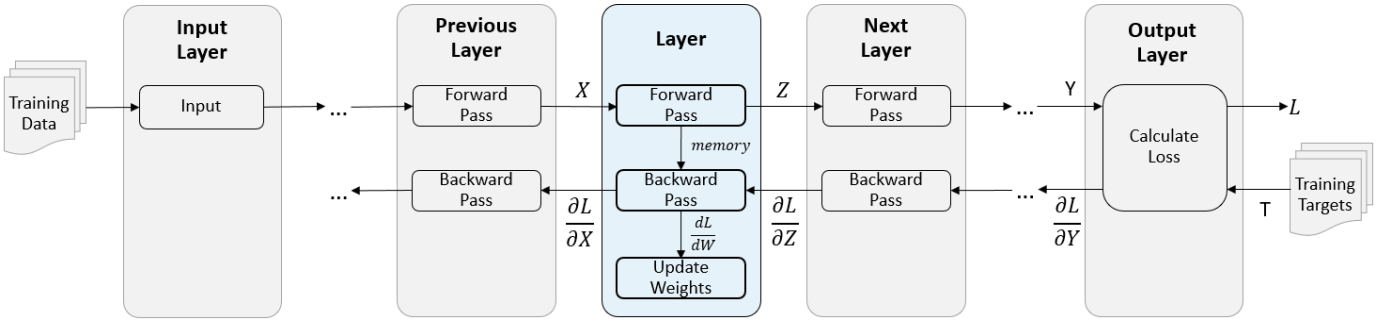
When making a forward pass through the network, each layer takes the outputs of the previous layers, applies a function, and then outputs (forward propagates) the results to the next layers. Stateful layers, such as LSTM layers, also update the layer state.

Layers can have multiple inputs or outputs. For example, a layer can take $X_1, \ldots, X_N$ from multiple previous layers and forward propagate the outputs $Z_1, \ldots, Z_M$ to subsequent layers.

At the end of a forward pass of the network, the output layer calculates the loss L between the predictions Y and the targets T.

During the backward pass of a network, each layer takes the derivatives of the loss with respect to the outputs of the layer, computes the derivatives of the loss L with respect to the inputs, and then backward propagates the results. If the layer has learnable parameters, then the layer also computes the derivatives of the layer weights (learnable parameters). The layer uses the derivatives of the weights to update the learnable parameters.

The following figure describes the flow of data through a deep neural network and highlights the data flow through a layer with a single input X, a single output Z, and a learnable parameter W.



Intermediate Layer Template

To define a custom intermediate layer, use this class definition template. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters. For more information, see “Intermediate Layer Properties” on page 18-19.
- The layer constructor function.
- The `predict` function and the optional `forward` function. For more information, see “Forward Functions” on page 18-21.
- The optional `resetState` function for layers with state properties. For more information, see “Reset State Function” on page 18-24.
- The optional `backward` function. For more information, see “Backward Function” on page 18-24.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Define layer constructor function here.
        end

        function [Z,state] = predict(layer,X)
            % Forward input data through the layer at prediction time and
            % output the result and updated state.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Input data
            % Outputs:
            %         Z     - Output of layer forward function
            %         state - (Optional) Updated layer state.
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer predict function here.
        end

        function [Z,state,memory] = forward(layer,X)
```

```
            % (Optional) Forward input data through the layer at training
            % time and output the result, updated state, and a memory
            % value.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Layer input data
            % Outputs:
            %         Z      - Output of layer forward function
            %         state  - (Optional) Updated layer state
            %         memory - (Optional) Memory value for custom backward
            %                  function
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer forward function here.
        end

        function layer = resetState(layer)
            % (Optional) Reset layer state.

            % Define reset state function here.
        end

        function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer   - Layer to backward propagate through
            %         X       - Layer input data
            %         Z       - Layer output data
            %         dLdZ    - Derivative of loss with respect to layer
            %                   output
            %         dLdSout - (Optional) Derivative of loss with respect
            %                   to state output
            %         memory  - Memory value from forward function
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Formatted Inputs and Outputs

Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Using formatted `dlarray` objects in custom layers also allows you to define layers where the inputs and outputs have different formats, such as layers that permute, add, or remove dimensions. For example, you can define a layer that takes as input a mini-batch of images with format `"SSCB"` (spatial, spatial, channel, batch) and output a mini-batch of sequences with format `"CBT"` (channel, batch, time). Using formatted `dlarray` objects also allows you to define layers that can operate on data with different input formats, for example, layers that support inputs with formats `"SSCB"` (spatial, spatial, channel, batch) and `"CBT"` (channel, batch, time).

If you do not specify a backward function, then the layer functions, by default, receive *unformatted* `dlarray` objects as input. To specify that the layer receives *formatted* `dlarray` objects as input and also outputs formatted `dlarray` objects, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

For an example showing how to define a custom layer with formatted inputs, see “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61.

Intermediate Layer Properties

Declare the layer properties in the `properties` section of the class definition.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Property	Description
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

Learnable Parameters

Declare the layer learnable parameters in the `properties (Learnable)` section of the class definition. You can specify numeric arrays or `dlnetwork` objects as learnable parameters. If the `dlnetwork` object has both learnable and state parameters (for example, a `dlnetwork` object that contains an LSTM layer), then you must specify it in the `properties (Learnable, State)` section. If the layer has no learnable parameters, then you can omit the `properties` sections with the `Learnable` attribute.

Optionally, you can specify the learning rate factor and the L2 factor of the learnable parameters. By default, each learnable parameter has its learning rate factor and L2 factor set to `1`. For both built-in and custom layers, you can set and get the learn rate factors and L2 regularization factors using the following functions.

Function	Description
`setLearnRateFactor`	Set the learn rate factor of a learnable parameter.

Function	Description
`setL2Factor`	Set the L2 regularization factor of a learnable parameter.
`getLearnRateFactor`	Get the learn rate factor of a learnable parameter.
`getL2Factor`	Get the L2 regularization factor of a learnable parameter.

To specify the learning rate factor and the L2 factor of a learnable parameter, use the syntaxes `layer = setLearnRateFactor(layer,parameterName,value)` and `layer = setL2Factor(layer,parameterName,value)`, respectively.

To get the value of the learning rate factor and the L2 factor of a learnable parameter, use the syntaxes `getLearnRateFactor(layer,parameterName)` and `getL2Factor(layer,parameterName)` respectively.

For example, this syntax sets the learn rate factor of the learnable parameter with the name `"Alpha"` to `0.1`.

```
layer = setLearnRateFactor(layer,"Alpha",0.1);
```

State Parameters

For stateful layers, such as recurrent layers, declare the layer state parameters in the `properties (State)` section of the class definition. For `dlnetwork` objects that have both learnable and state parameters (for example, a `dlnetwork` object that contains an LSTM layer), then you must specify it in the `properties (Learnable, State)` section. If the layer has no state parameters, then you can omit the `properties` sections with the `State` attribute.

If the layer has state parameters, then the forward functions must also return the updated layer state. For more information, see “Forward Functions” on page 18-21.

To specify a custom reset state function, include a function with syntax `layer = resetState(layer)` in the class definition. For more information, see “Reset State Function” on page 18-24.

Forward Functions

Some layers behave differently during training and during prediction. For example, a dropout layer performs dropout only during training and has no effect during prediction. A layer uses one of two functions to perform a forward pass: `predict` or `forward`. If the forward pass is at prediction time, then the layer uses the `predict` function. If the forward pass is at training time, then the layer uses the `forward` function. If you do not require two different functions for prediction time and training time, then you can omit the `forward` function. In this case, the layer uses `predict` at training time.

If the layer has state parameters, then the forward functions must also return the updated layer state parameters as numeric arrays.

If you define both a custom `forward` function and a custom `backward` function, then the forward function must return a `memory` output.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

The `forward` function syntax depends on the type of layer:

- `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = forward(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.
- `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `forward` function of the custom layer, use the `forward` function of the `dlnetwork` object. Using the `dlnetwork` object `forward` function ensures that the software uses the correct layer operations for training.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For layers that output sequences, the layers can output sequences of any length or output data with no time dimension. Note that when training a network that outputs sequences using the `trainNetwork` function, the lengths of the input and output sequences must match.

The outputs of the custom layer forward functions must not be complex. If the `predict` or `forward` functions of your custom layer involve complex numbers, convert all outputs to real values before returning them. Using complex numbers in the `predict` or `forward` functions of your custom layer can lead to complex learnable parameters. If you are using automatic differentiation (in other words, you are not writing a backward function for your custom layer) then convert all learnable parameters to real values at the beginning of the function computation. Doing so ensures that the outputs of automatically generated backward functions are not complex.

Reset State Function

When `DAGNetwork` or `SeriesNetwork` objects contain layers with state parameters, you can make predictions and update the layer states using the `predictAndUpdateState` and `classifyAndUpdateState` functions. You can reset the network state using the `resetState` function.

The `resetState` function for `DAGNetwork`, `SeriesNetwork`, and `dlnetwork` objects, by default, has no effect on custom layers with state parameters. To define the layer behavior for the `resetState` function for network objects, define the optional layer `resetState` function in the layer definition that resets the state parameters.

The `resetState` function must have the syntax `layer = resetState(layer)`, where the returned layer has the state properties reset.

Backward Function

The layer backward function computes the derivatives of the loss with respect to the input data and then outputs (backward propagates) results to the previous layer. If the layer has learnable parameters (for example, layer weights), then `backward` also computes the derivatives of the learnable parameters. When using the `trainNetwork` function, the layer automatically updates the learnable parameters using these derivatives during the backward pass.

Defining the backward function is optional. If you do not specify a backward function, and the layer forward functions support `dlarray` objects, then the software automatically determines the backward function using automatic differentiation. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. Define a custom backward function when you want to:

- Use a specific algorithm to compute the derivatives.
- Use operations in the forward functions that do not support `dlarray` objects.

Custom layers with learnable `dlnetwork` objects do not support custom backward functions.

To define a custom backward function, create a function named `backward`.

The `backward` function syntax depends on the type of layer.

- `dLdX = backward(layer,X,Z,dLdZ,memory)` returns the derivatives `dLdX` of the loss with respect to the layer input, where `layer` has a single input and a single output. `Z` corresponds to the forward function output and `dLdZ` corresponds to the derivative of the loss with respect to `Z`. The function input `memory` corresponds to the memory output of the forward function.
- `[dLdX,dLdW] = backward(layer,X,Z,dLdZ,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter, where `layer` has a single learnable parameter.

- `[dLdX,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdSin` of the loss with respect to the state input using any of the previous syntaxes, where `layer` has a single state parameter and `dLdSout` corresponds to the derivative of the loss with respect to the layer state output.
- `[dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter and returns the derivative `dLdSin` of the loss with respect to the layer state input using any of the previous syntaxes, where `layer` has a single state parameter and single learnable parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, multiple learnable parameters, or multiple state parameters:

- For layers with multiple inputs, replace `X` and `dLdX` with `X1,...,XN` and `dLdX1,...,dLdXN`, respectively, where `N` is the number of inputs.
- For layers with multiple outputs, replace `Z` and `dLdZ` with `Z1,...,ZM` and `dLdZ1,...,dLdZM`, respectively, where `M` is the number of outputs.
- For layers with multiple learnable parameters, replace `dLdW` with `dLdW1,...,dLdWP`, where `P` is the number of learnable parameters.
- For layers with multiple state parameters, replace `dLdSin` and `dLdSout` with `dLdSin1,...,dLdSinK` and `dLdSout1,...,dLdSoutK`, respectively, where `K` is the number of state parameters.

To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where the first `N` elements correspond to the `N` layer inputs, the next `M` elements correspond to the `M` layer outputs, the next `M` elements correspond to the derivatives of the loss with respect to the `M` layer outputs, the next `K` elements correspond to the `K` derivatives of the loss with respect to the `K` states outputs, and the last element corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where the first `N` elements correspond to the `N` the derivatives of the loss with respect to the `N` layer inputs, the next `P` elements correspond to the derivatives of the loss with respect to the `P` learnable parameters, and the next `K` elements correspond to the derivatives of the loss with respect to the `K` state inputs.

The values of `X` and `Z` are the same as in the forward functions. The dimensions of `dLdZ` are the same as the dimensions of `Z`.

The dimensions and data type of `dLdX` are the same as the dimensions and data type of `X`. The dimensions and data types of `dLdW` are the same as the dimensions and data types of `W`.

To calculate the derivatives of the loss, you can use the chain rule:

$$\frac{\partial L}{\partial X^{(i)}} = \sum_j \frac{\partial L}{\partial Z_j}\frac{\partial Z_j}{\partial X^{(i)}}$$

$$\frac{\partial L}{\partial W_i} = \sum_j \frac{\partial L}{\partial Z_j}\frac{\partial Z_j}{\partial W_i}$$

When using the `trainNetwork` function, the layer automatically updates the learnable parameters using the derivatives `dLdW` during the backward pass.

For an example showing how to define a custom backward function, see “Specify Custom Layer Backward Function” on page 18-107.

The outputs of the custom layer backward function must not be complex. If your backward function involves complex numbers, then convert all outputs of the backward function to real values before returning them.

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Code Generation Compatibility

To create a custom layer that supports code generation:

- The layer must specify the pragma `%#codegen` in the layer definition.
- The inputs of `predict` must be:
  - Consistent in dimension. Each input must have the same number of dimensions.
  - Consistent in batch size. Each input must have the same batch size.
- The outputs of `predict` must be consistent in dimension and batch size with the layer inputs.
- Nonscalar properties must have type single, double, or character array.
- Scalar properties must have type numeric, logical, or string.

Code generation supports intermediate layers with 2-D image or feature input only. Code generation does not support layers with state properties (properties with attribute `State`).

For an example showing how to create a custom layer that supports code generation, see “Define Custom Deep Learning Layer for Code Generation” on page 18-142.

Network Composition

To create a custom layer that itself defines a layer graph, you can declare a `dlnetwork` object as a learnable parameter in the `properties (Learnable)` section of the layer definition. This method is known as *network composition*. You can use network composition to:

- Create a single custom layer that represents a block of learnable layers, for example, a residual block.
- Create a network with control flow, for example, a network with a section that can dynamically change depending on the input data.

- Create a network with loops, for example, a network with sections that feed the output back into itself.

For nested networks that have both learnable and state parameters, for example, networks with batch normalization or LSTM layers, declare the network in the `properties (Learnable, State)` section of the layer definition.

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Check Validity of Layer

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, correctly defined gradients, and code generation compatibility. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer. To check with multiple observations, use the `ObservationDimension` option. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to `1` (true). For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

For more information, see “Check Custom Layer Validity” on page 18-154.

Check Validity of Custom Layer Using checkLayer

Check the layer validity of the custom layer `preluLayer`.

The custom layer `preluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set the `ObservationDimension` option to 4.

```
layer = preluLayer(20);
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,ObservationDimension=4)

Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 0.21297 seconds.
```

Here, the function does not detect any issues with the layer.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

Related Examples

- “Define Custom Deep Learning Layers” on page 18-9
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Specify Custom Layer Backward Function” on page 18-107
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Deep Learning Network Composition” on page 18-117
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Custom Deep Learning Output Layers

Tip This topic explains how to define custom deep learning output layers for your problems. For a list of built-in layers in Deep Learning Toolbox, see “List of Deep Learning Layers” on page 1-21.

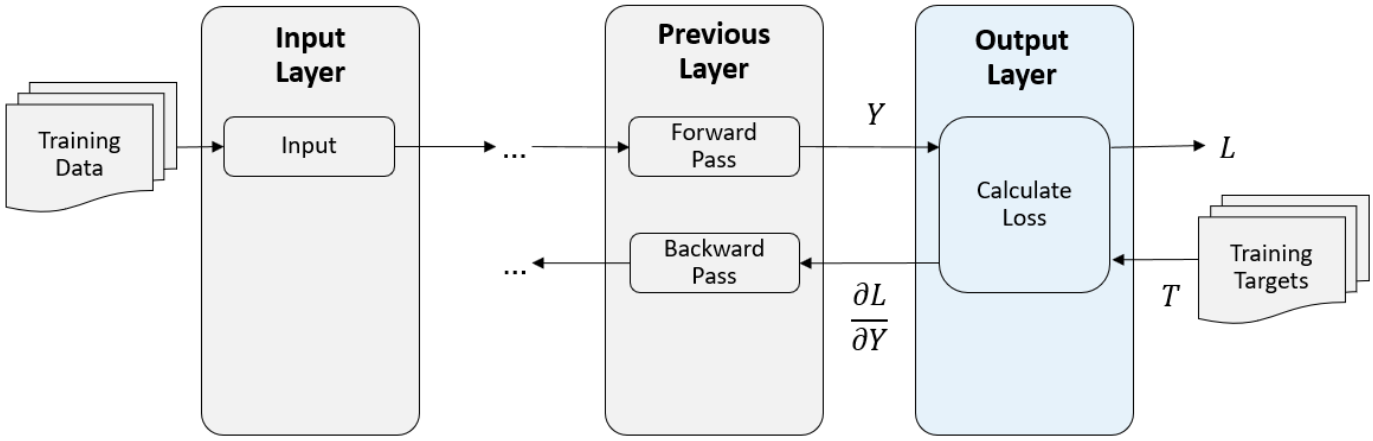
To learn how to define custom intermediate layers, see “Define Custom Deep Learning Intermediate Layers” on page 18-16.

If Deep Learning Toolbox does not provide the output layer that you require for your task, then you can define your own custom layer using this topic as a guide. After defining the custom layer, you can automatically check that the layer is valid, GPU compatible, and outputs correctly defined gradients.

Output Layer Architecture

At the end of a forward pass at training time, an output layer takes the predictions (network outputs) Y of the previous layer and calculates the loss L between these predictions and the training targets. The output layer computes the derivatives of the loss L with respect to the predictions Y and outputs (backward propagates) results to the previous layer.

The following figure describes the flow of data through a convolutional neural network and an output layer.



Output Layer Templates

To define a custom intermediate layer, use one of these class definition templates. The templates outline the structure of an output layer class definition. They outline:

- The optional `properties` blocks for the layer properties. For more information, see “Output Layer Properties” on page 18-31.
- The layer constructor function.
- The `forwardLoss` function. For more information, see “Forward Loss Function” on page 18-32.
- The optional `backwardLoss` function. For more information, see “Backward Loss Function” on page 18-32.

Classification Output Layer Template

This template outlines the structure of a classification output layer with a loss function. For an example showing how to define a classification output layer and specify a loss function, see “Define Custom Classification Output Layer” on page 18-91.

```
classdef myClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    methods
        function layer = myClassificationLayer()
            % (Optional) Create a myClassificationLayer.

            % Layer constructor function goes here.
        end

        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Regression Output Layer Template

This template outlines the structure of a regression output layer with a loss function. For an example showing how to define a regression output layer and specify a loss function, see “Define Custom Regression Output Layer” on page 18-99.

```
classdef myRegressionLayer < nnet.layer.RegressionLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    methods
        function layer = myRegressionLayer()
            % (Optional) Create a myRegressionLayer.
```

```
            % Layer constructor function goes here.
        end

        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Output Layer Properties

Declare the layer properties in the `properties` section of the class definition.

By default, custom output layers have the following properties:

- `Name` – Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
- `Description` – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays `"Classification Output"` or `"Regression Output"`.
- `Type` – Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- `Classes` – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. If `Classes` is `'auto'`, then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`.

Custom regression layers also have the following property:

- `ResponseNames` – Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is `{}`.

If the layer has no other properties, then you can omit the `properties` section.

Forward Loss Function

The output layer computes the loss `L` between predictions and targets using the forward loss function and computes the derivatives of the loss with respect to the predictions using the backward loss function.

The syntax for `forwardLoss` is `loss = forwardLoss(layer,Y,T)`. The input `Y` corresponds to the predictions made by the network. These predictions are the output of the previous layer. The input `T` corresponds to the training targets. The output `loss` is the loss between `Y` and `T` according to the specified loss function. The output `loss` must be scalar.

Backward Loss Function

If the layer forward loss function supports `dlarray` objects, then the software automatically determines the backward loss function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. Alternatively, to define a custom backward loss function, create a function named `backwardLoss`. For an example showing how to define a custom backward loss function, see “Specify Custom Output Layer Backward Loss Function” on page 18-113.

The syntax for `backwardLoss` is `dLdY = backwardLoss(layer,Y,T)`. The input `Y` contains the predictions made by the network and `T` contains the training targets. The output `dLdY` is the derivative of the loss with respect to the predictions `Y`. The output `dLdY` must be the same size as the layer input `Y`.

For classification problems, the dimensions of `T` depend on the type of problem.

Classification Task	Input Size	Observation Dimension
2-D image classification	1-by-1-by-*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	4
3-D image classification	1-by-1-by-1-by-*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	5
Sequence-to-label classification	*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	2
Sequence-to-sequence classification	*K*-by-*N*-by-*S*, where *K* is the number of classes, *N* is the number of observations, and *S* is the sequence length.	2

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, to ensure that `Y` is a 4-D array of prediction scores for *K* classes, you can include a fully connected layer of size *K* followed by a softmax layer before the output layer.

For regression problems, the dimensions of `T` also depend on the type of problem.

Regression Task	Input Size	Observation Dimension
2-D image regression	1-by-1-by-*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	4
2-D Image-to-image regression	*h*-by-*w*-by-*c*-by-`N`, where *h*, *w*, and *c* are the height, width, and number of channels of the output respectively, and *N* is the number of observations.	4
3-D image regression	1-by-1-by-1-by-*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	5
3-D Image-to-image regression	*h*-by-*w*-by-*d*-by-*c*-by-`N`, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the output respectively, and *N* is the number of observations.	5
Sequence-to-one regression	*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	2
Sequence-to-sequence regression	*R*-by-*N*-by-*S*, where *R* is the number of responses, *N* is the number of observations, and *S* is the sequence length.	2

For example, if the network defines an image regression network with one response and has mini-batches of size 50, then `T` is a 4-D array of size 1-by-1-by-1-by-50.

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, for image regression with *R* responses, to ensure that `Y` is a 4-D array of the correct size, you can include a fully connected layer of size *R* before the output layer.

The `forwardLoss` and `backwardLoss` functions have the following output arguments.

Function	Output Argument	Description
`forwardLoss`	`loss`	Calculated loss between the predictions `Y` and the true target `T`.
`backwardLoss`	`dLdY`	Derivative of the loss with respect to the predictions `Y`.

The `backwardLoss` must output `dLdY` with the size expected by the previous layer and `dLdY` to be the same size as `Y`.

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Check Validity of Layer

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, correctly defined gradients, and code generation compatibility. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer. To check with multiple observations, use the `ObservationDimension` option. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to `1` (true). For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

For more information, see “Check Custom Layer Validity” on page 18-154.

See Also

`checkLayer` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

Related Examples

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Classification Output Layer” on page 18-91
- “Define Custom Regression Output Layer” on page 18-99
- “Specify Custom Output Layer Backward Loss Function” on page 18-113
- “Check Custom Layer Validity” on page 18-154

Define Custom Deep Learning Layer with Learnable Parameters

If Deep Learning Toolbox does not provide the layer you require for your task, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

When defining the layer functions, you can use `dlarray` objects. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Using formatted `dlarray` objects in custom layers also allows you to define layers where the inputs and outputs have different formats, such as layers that permute, add, or remove dimensions. For example, you can define a layer that takes as input a mini-batch of images with format `"SSCB"` (spatial, spatial, channel, batch) and output a mini-batch of sequences with format `"CBT"` (channel, batch, time). Using formatted `dlarray` objects also allows you to define layers that can operate on data with different input formats, for example, layers that support inputs with formats `"SSCB"` (spatial, spatial, channel, batch) and `"CBT"` (channel, batch, time).

`dlarray` objects also enable support for automatic differentiation. This means that if your forward functions fully support `dlarray` objects, then defining the backward function is optional.

To enable support for using formatted `dlarray` objects in custom layer forward functions, also inherit from the `nnet.layer.Formattable` class when defining the custom layer. For an example, see “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61.

This example shows how to create a PReLU layer, which is a layer with a learnable parameter and use it in a convolutional neural network. A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar learned at training time.[1] For values less than zero, a PReLU layer applies scaling coefficients α_i to each channel of the input. These coefficients form a learnable parameter, which the layer learns during training.

This figure from [1] compares the ReLU and PReLU layer functions.

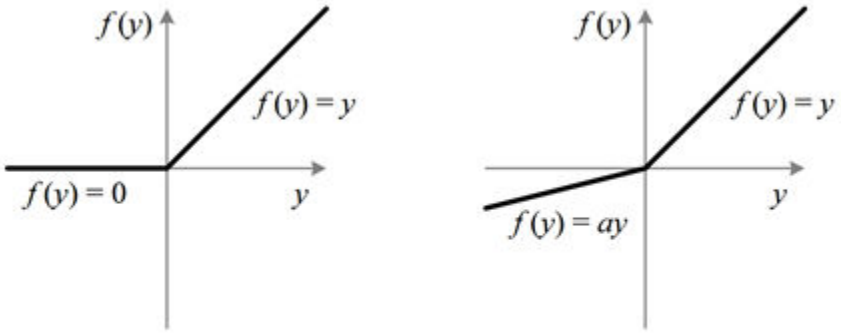


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
```

```
    % (Optional) Create a myLayer.
    % This function must have the same name as the class.

    % Define layer constructor function here.
end

function [Z,state] = predict(layer,X)
    % Forward input data through the layer at prediction time and
    % output the result and updated state.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Input data
    % Outputs:
    %         Z     - Output of layer forward function
    %         state - (Optional) Updated layer state.
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer predict function here.
end

function [Z,state,memory] = forward(layer,X)
    % (Optional) Forward input data through the layer at training
    % time and output the result, updated state, and a memory
    % value.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Layer input data
    % Outputs:
    %         Z      - Output of layer forward function
    %         state  - (Optional) Updated layer state
    %         memory - (Optional) Memory value for custom backward
    %                  function
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer forward function here.
end

function layer = resetState(layer)
    % (Optional) Reset layer state.

    % Define reset state function here.
end

function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
    % (Optional) Backward propagate the derivative of the loss
    % function through the layer.
    %
    % Inputs:
    %         layer   - Layer to backward propagate through
    %         X       - Layer input data
    %         Z       - Layer output data
    %         dLdZ    - Derivative of loss with respect to layer
    %                   output
    %         dLdSout - (Optional) Derivative of loss with respect
    %                   to state output
    %         memory  - Memory value from forward function
```

```
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Name Layer and Specify Superclasses

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `preluLayer`.

```
classdef preluLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)
    ...
end
```

If you do not specify a backward function, then the layer functions, by default, receive *unformatted* `dlarray` objects as input. To specify that the layer receives *formatted* `dlarray` objects as input and also outputs formatted `dlarray` objects, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

The layer does not require formattable inputs, so remove the optional `nnet.layer.Formattable` superclass.

```
classdef preluLayer < nnet.layer.Layer
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = preluLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `preluLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.

Property	Description
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

A PReLU layer does not require any additional properties, so you can remove the `properties` section.

A PReLU layer has only one learnable parameter, the scaling coefficient a. Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Alpha`.

```
properties (Learnable)
    % Layer learnable parameters

    % Scaling coefficient
    Alpha
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The PReLU layer constructor function requires one input argument (the number of channels) and one optional argument (the layer name). The number of channels specifies the size of the learnable parameter `Alpha`. Specify two input arguments named `numChannels` and `args` in the `preluLayer` function that correspond to the number of channels and optional input arguments, respectively. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = preluLayer(numChannels,args)
    % layer = preluLayer(numChannels) creates a PReLU layer
    % with numChannels channels.
    %
    % layer = preluLayer(numChannels,Name=name) also specifies the
    % layer name

    ...
end
```

Initialize Layer Properties

Initialize the layer properties, including learnable parameters in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Parse the input arguments using an `arguments` block and set the `Name` property.

```
arguments
    numChannels
    args.Name = "";
end

% Set layer name.
layer.Name = args.Name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "PReLU with " + numChannels + " channels";
```

For a PReLU layer, when the input values are negative, the layer multiplies each channel of the input by the corresponding channel of `Alpha`. Initialize the learnable parameter `Alpha` to be a random vector of size 1-by-1-by-`numChannels`. With the third dimension specified as size `numChannels`, the layer can use element-wise multiplication of the input in the forward function. `Alpha` is a property of the layer object, so you must assign the vector to `layer.Alpha`.

```
% Initialize scaling coefficient.
layer.Alpha = rand([1 1 numChannels]);
```

View the completed constructor function.

```
function layer = preluLayer(numChannels,args)
    % layer = preluLayer(numChannels) creates a PReLU layer
    % with numChannels channels.
    %
    % layer = preluLayer(numChannels,Name=name) also specifies the
    % layer name.

    arguments
        numChannels
        args.Name = "";
    end

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    layer.Description = "PReLU with " + numChannels + " channels";

    % Initialize scaling coefficient.
    layer.Alpha = rand([1 1 numChannels]);
end
```

With this constructor function, the command `preluLayer(3,Name="prelu")` creates a PReLU layer with three channels and the name `"prelu"`.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.

- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

Because a PReLU layer has only one input and one output, the syntax for `predict` for a PReLU layer is `Z = predict(layer,X)`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for a custom backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5

Layer Input	Input Size	Observation Dimension
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For layers that output sequences, the layers can output sequences of any length or output data with no time dimension. Note that when training a network that outputs sequences using the `trainNetwork` function, the lengths of the input and output sequences must match.

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The `forward` function syntax depends on the type of layer:

- `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = forward(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.
- `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `forward` function of the custom layer, use the `forward` function of the `dlnetwork` object. Using the `dlnetwork` object `forward` function ensures that the software uses the correct layer operations for training.

The PReLU operation is given by

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ \alpha_i x_i & \text{if } x_i \le 0 \end{cases}$$

where x_i is the input of the nonlinear activation f on channel i, and α_i is the coefficient controlling the slope of the negative part. The subscript i in α_i indicates that the nonlinear activation can vary on different channels.

Implement this operation in `predict`. In `predict`, the input `X` corresponds to x in the equation. The output `Z` corresponds to $f(x_i)$. The PReLU layer does not require memory or a different forward function for training, so you can remove the `forward` function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

```
function Z = predict(layer, X)
    % Z = predict(layer, X) forwards the input data X through the
    % layer and outputs the result Z.

    Z = max(X,0) + layer.Alpha .* min(0,X);
end
```

Because the `predict` function only uses functions that support `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed layer class file.

```
classdef preluLayer < nnet.layer.Layer
    % Example custom PReLU layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficient
        Alpha
    end
```

```
    methods
        function layer = preluLayer(numChannels,args)
            % layer = preluLayer(numChannels) creates a PReLU layer
            % with numChannels channels.
            %
            % layer = preluLayer(numChannels,Name=name) also specifies the
            % layer name.

            arguments
                numChannels
                args.Name = "";
            end

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = "PReLU with " + numChannels + " channels";

            % Initialize scaling coefficient.
            layer.Alpha = rand([1 1 numChannels]);
        end

        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            Z = max(X,0) + layer.Alpha .* min(0,X);
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

In this example, the MATLAB functions used in `predict` all support `dlarray` objects, so the layer is GPU compatible.

Check Validity of Custom Layer Using checkLayer

Check the layer validity of the custom layer `preluLayer`.

The custom layer `preluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set the `ObservationDimension` option to 4.

```
layer = preluLayer(20);
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,ObservationDimension=4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 0.21297 seconds.
```

Here, the function does not detect any issues with the layer.

Include Custom Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox. This section shows how to create and train a network for digit classification using the PReLU layer you created earlier.

Load the example training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
```

Create a layer array containing the custom layer `preluLayer`, attached to this is example as a supporting file. To access this layer, open this example as a live script.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    preluLayer(20)
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Set the training options and train the network.

```
options = trainingOptions("adam",MaxEpochs=10);
net = trainNetwork(XTrain,YTrain,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
```

```
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:00 |       10.94% |       3.0526 |          0.0010 |
|       2 |          50 |       00:00:07 |       71.88% |       0.8378 |          0.0010 |
|       3 |         100 |       00:00:15 |       85.94% |       0.4878 |          0.0010 |
|       4 |         150 |       00:00:22 |       88.28% |       0.4068 |          0.0010 |
|       6 |         200 |       00:00:31 |       96.09% |       0.1690 |          0.0010 |
|       7 |         250 |       00:00:38 |       97.66% |       0.1368 |          0.0010 |
|       8 |         300 |       00:00:45 |       99.22% |       0.0744 |          0.0010 |
|       9 |         350 |       00:00:52 |       99.22% |       0.0592 |          0.0010 |
|      10 |         390 |       00:00:59 |      100.00% |       0.0465 |          0.0010 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Evaluate the network performance by predicting on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
accuracy = mean(YTest==YPred)
```

```
accuracy = 0.9188
```

References

[1] "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *In 2015 IEEE International Conference on Computer Vision (ICCV)*, 1026–34. Santiago, Chile: IEEE, 2015. https://doi.org/10.1109/ICCV.2015.123.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Specify Custom Layer Backward Function” on page 18-107
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Custom Deep Learning Layer with Multiple Inputs

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

This example shows how to create a weighted addition layer, which is a layer with multiple inputs and learnable parameter, and use it in a convolutional neural network. A weighted addition layer scales and adds inputs from multiple neural network layers element-wise.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end
```

```
properties (State)
    % (Optional) Layer state parameters.

    % Declare state parameters here.
end

properties (Learnable, State)
    % (Optional) Nested dlnetwork objects with both learnable
    % parameters and state.

    % Declare nested networks with learnable and state parameters here.
end

methods
    function layer = myLayer()
        % (Optional) Create a myLayer.
        % This function must have the same name as the class.

        % Define layer constructor function here.
    end

    function [Z,state] = predict(layer,X)
        % Forward input data through the layer at prediction time and
        % output the result and updated state.
        %
        % Inputs:
        %         layer - Layer to forward propagate through
        %         X     - Input data
        % Outputs:
        %         Z     - Output of layer forward function
        %         state - (Optional) Updated layer state.
        %
        %  - For layers with multiple inputs, replace X with X1,...,XN,
        %    where N is the number of inputs.
        %  - For layers with multiple outputs, replace Z with
        %    Z1,...,ZM, where M is the number of outputs.
        %  - For layers with multiple state parameters, replace state
        %    with state1,...,stateK, where K is the number of state
        %    parameters.

        % Define layer predict function here.
    end

    function [Z,state,memory] = forward(layer,X)
        % (Optional) Forward input data through the layer at training
        % time and output the result, updated state, and a memory
        % value.
        %
        % Inputs:
        %         layer - Layer to forward propagate through
        %         X     - Layer input data
        % Outputs:
        %         Z      - Output of layer forward function
        %         state  - (Optional) Updated layer state
        %         memory - (Optional) Memory value for custom backward
        %                  function
        %
        %  - For layers with multiple inputs, replace X with X1,...,XN,
        %    where N is the number of inputs.
        %  - For layers with multiple outputs, replace Z with
        %    Z1,...,ZM, where M is the number of outputs.
        %  - For layers with multiple state parameters, replace state
        %    with state1,...,stateK, where K is the number of state
        %    parameters.

        % Define layer forward function here.
    end

    function layer = resetState(layer)
        % (Optional) Reset layer state.

        % Define reset state function here.
```

```
        end

        function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer   - Layer to backward propagate through
            %         X       - Layer input data
            %         Z       - Layer output data
            %         dLdZ    - Derivative of loss with respect to layer
            %                   output
            %         dLdSout - (Optional) Derivative of loss with respect
            %                   to state output
            %         memory  - Memory value from forward function
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Name Layer and Specify Superclasses

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `weightedAdditionLayer`.

```
classdef weightedAdditionLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)
    ...
end
```

If you do not specify a backward function, then the layer functions, by default, receive *unformatted* `dlarray` objects as input. To specify that the layer receives *formatted* `dlarray` objects as input and also outputs formatted `dlarray` objects, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

The layer does not require formattable inputs, so remove the optional `nnet.layer.Formattable` superclass.

```
classdef weightedAdditionLayer < nnet.layer.Layer
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = weightedAdditionLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `weightedAdditionLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.

Property	Description
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor.

A weighted addition layer does not require any additional properties, so you can remove the `properties` section.

A weighted addition layer has only one learnable parameter, the weights. Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Weights`.

```
properties (Learnable)
    % Layer learnable parameters

    % Scaling coefficients
    Weights
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The weighted addition layer constructor function requires two inputs: the number of inputs to the layer and the layer name. This number of inputs to the layer specifies the size of the learnable parameter `Weights`. Specify two input arguments named `numInputs` and `name` in the `weightedAdditionLayer` function. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = weightedAdditionLayer(numInputs,name)
    % layer = weightedAdditionLayer(numInputs,name) creates a
    % weighted addition layer and specifies the number of inputs
    % and the layer name.

    ...
end
```

Initialize Layer Properties

Initialize the layer properties, including learnable parameters, in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the `NumInputs` property to the input argument `numInputs`.

```
% Set number of inputs.
layer.NumInputs = numInputs;
```

Set the `Name` property to the input argument `name`.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "Weighted addition of " + numInputs + ...
    " inputs";
```

A weighted addition layer multiplies each layer input by the corresponding coefficient in `Weights` and adds the resulting values together. Initialize the learnable parameter `Weights` to be a random vector of size 1-by-`numInputs`. `Weights` is a property of the layer object, so you must assign the vector to `layer.Weights`.

```
% Initialize layer weights
layer.Weights = rand(1,numInputs);
```

View the completed constructor function.

```
function layer = weightedAdditionLayer(numInputs,name)
    % layer = weightedAdditionLayer(numInputs,name) creates a
    % weighted addition layer and specifies the number of inputs
    % and the layer name.

    % Set number of inputs.
    layer.NumInputs = numInputs;

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    layer.Description = "Weighted addition of " + numInputs +  ...
        " inputs";

    % Initialize layer weights.
```

```
            layer.Weights = rand(1,numInputs);
        end
```

With this constructor function, the command `weightedAdditionLayer(3,'add')` creates a weighted addition layer with three inputs and the name `'add'`.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

Because a weighted addition layer has only one output and a variable number of inputs, the syntax for `predict` for a weighted addition layer is `Z = predict(layer,varargin)`, where `varargin{i}` corresponds to `Xi` for positive integers `i` less than or equal to `NumInputs`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for the backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For layers that output sequences, the layers can output sequences of any length or output data with no time dimension. Note that when training a network that outputs sequences using the `trainNetwork` function, the lengths of the input and output sequences must match.

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The `forward` function syntax depends on the type of layer:

- `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = forward(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

- `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `forward` function of the custom layer, use the `forward` function of the `dlnetwork` object. Using the `dlnetwork` object `forward` function ensures that the software uses the correct layer operations for training.

The forward function of a weighted addition layer is

$$f(X^{(1)},...,X^{(n)}) = \sum_{i=1}^{n} W_i X^{(i)}$$

where $X^{(1)}, ..., X^{(n)}$ correspond to the layer inputs and $W_1,...,W_n$ are the layer weights.

Implement the forward function in `predict`. In `predict`, the output `Z` corresponds to $f(X^{(1)},...,X^{(n)})$. The weighted addition layer does not require memory or a different forward function for training, so you can remove the `forward` function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

```
function Z = predict(layer, varargin)
    % Z = predict(layer, X1, ..., Xn) forwards the input data X1,
    % ..., Xn through the layer and outputs the result Z.

    X = varargin;
    W = layer.Weights;
```

```
            % Initialize output
            X1 = X{1};
            sz = size(X1);
            Z = zeros(sz,'like',X1);

            % Weighted addition
            for i = 1:layer.NumInputs
                Z = Z + W(i)*X{i};
            end
        end
```

Because the `predict` function only uses functions that support `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed layer class file.

```
classdef weightedAdditionLayer < nnet.layer.Layer
    % Example custom weighted addition layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficients
        Weights
    end

    methods
        function layer = weightedAdditionLayer(numInputs,name)
            % layer = weightedAdditionLayer(numInputs,name) creates a
            % weighted addition layer and specifies the number of inputs
            % and the layer name.

            % Set number of inputs.
            layer.NumInputs = numInputs;

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = "Weighted addition of " + numInputs +  ...
                " inputs";

            % Initialize layer weights.
            layer.Weights = rand(1,numInputs);
        end

        function Z = predict(layer, varargin)
            % Z = predict(layer, X1, ..., Xn) forwards the input data X1,
            % ..., Xn through the layer and outputs the result Z.

            X = varargin;
            W = layer.Weights;

            % Initialize output
```

```
            X1 = X{1};
            sz = size(X1);
            Z = zeros(sz,'like',X1);

            % Weighted addition
            for i = 1:layer.NumInputs
                Z = Z + W(i)*X{i};
            end
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

In this example, the MATLAB functions used in `predict` all support `dlarray` objects, so the layer is GPU compatible.

Check Validity of Layer with Multiple Inputs

Check the layer validity of the custom layer `weightedAdditionLayer`.

Define a custom weighted addition layer. To create this layer, save the file `weightedAdditionLayer.m` in the current folder.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input sizes to be the typical sizes of a single observation for each input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set `'ObservationDimension'` to 4.

```
layer = weightedAdditionLayer(2,'add');
validInputSize = {[24 24 20],[24 24 20]};
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

```
Running nnet.checklayer.TestLayerWithoutBackward
.......... .......
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     17 Passed, 0 Failed, 0 Incomplete, 0 Skipped.
     Time elapsed: 0.55735 seconds.
```

Here, the function does not detect any issues with the layer.

Use Custom Weighted Addition Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox. This section shows how to create and train a network for digit classification using the weighted addition layer you created earlier.

Load the example training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
```

Define a custom weighted addition layer. To create this layer, save the file `weightedAdditionLayer.m` in the current folder.

Create a layer graph including the custom layer `weightedAdditionLayer`.

```
layers = [
    imageInputLayer([28 28 1],'Name','in')
    convolution2dLayer(5,20,'Name','conv1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    reluLayer('Name','relu3')
    weightedAdditionLayer(2,'add')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')
    classificationLayer('Name','classoutput')];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph, 'relu1', 'add/in2');
```

Set the training options and train the network.

```
options = trainingOptions('adam','MaxEpochs',10);
net = trainNetwork(XTrain,YTrain,lgraph,options);
```

```
Training on single CPU.
Initializing input data normalization.
|========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|========================================================================================|
|       1 |           1 |       00:00:03 |       12.50% |       2.2951 |          0.0010 |
|       2 |          50 |       00:00:20 |       72.66% |       0.7880 |          0.0010 |
|       3 |         100 |       00:00:35 |       89.84% |       0.3001 |          0.0010 |
|       4 |         150 |       00:00:48 |       94.53% |       0.1555 |          0.0010 |
|       6 |         200 |       00:01:00 |       99.22% |       0.0385 |          0.0010 |
|       7 |         250 |       00:01:12 |       99.22% |       0.0361 |          0.0010 |
|       8 |         300 |       00:01:25 |      100.00% |       0.0112 |          0.0010 |
|       9 |         350 |       00:01:37 |       99.22% |       0.0249 |          0.0010 |
|      10 |         390 |       00:01:46 |      100.00% |       0.0045 |          0.0010 |
|========================================================================================|
Training finished: Max epochs completed.
```

View the weights learned by the weighted addition layer.

```
net.Layers(8).Weights
```

```
ans = 1x2 single row vector

    1.0223    1.0005
```

Evaluate the network performance by predicting on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
accuracy = sum(YTest==YPred)/numel(YTest)
```

```
accuracy = 0.9878
```

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Specify Custom Layer Backward Function” on page 18-107
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Custom Deep Learning Layer with Formatted Inputs

If Deep Learning Toolbox does not provide the layer you require for your task, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

When defining the layer functions, you can use `dlarray` objects. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Using formatted `dlarray` objects in custom layers also allows you to define layers where the inputs and outputs have different formats, such as layers that permute, add, or remove dimensions. For example, you can define a layer that takes as input a mini-batch of images with format `"SSCB"` (spatial, spatial, channel, batch) and output a mini-batch of sequences with format `"CBT"` (channel, batch, time). Using formatted `dlarray` objects also allows you to define layers that can operate on data with different input formats, for example, layers that support inputs with formats `"SSCB"` (spatial, spatial, channel, batch) and `"CBT"` (channel, batch, time).

`dlarray` objects also enable support for automatic differentiation. This means that if your forward functions fully support `dlarray` objects, then defining the backward function is optional.

This example shows how to create a *project and reshape* layer, which is a layer commonly used in generative adversarial networks (GANs) that takes an array of noise with format `"CB"` (channel, batch) and projects and reshapes it to a mini-batch of images with format `"SSCB"` (spatial, spatial, channel, batch) using fully connected, reshape, and relabel operations.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Define layer constructor function here.
        end

        function [Z,state] = predict(layer,X)
            % Forward input data through the layer at prediction time and
            % output the result and updated state.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Input data
            % Outputs:
            %         Z     - Output of layer forward function
            %         state - (Optional) Updated layer state.
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer predict function here.
        end

        function [Z,state,memory] = forward(layer,X)
            % (Optional) Forward input data through the layer at training
            % time and output the result, updated state, and a memory
            % value.
```

```
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Layer input data
            % Outputs:
            %         Z      - Output of layer forward function
            %         state  - (Optional) Updated layer state
            %         memory - (Optional) Memory value for custom backward
            %                  function
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.

            % Define layer forward function here.
        end

        function layer = resetState(layer)
            % (Optional) Reset layer state.

            % Define reset state function here.
        end

        function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer   - Layer to backward propagate through
            %         X       - Layer input data
            %         Z       - Layer output data
            %         dLdZ    - Derivative of loss with respect to layer
            %                   output
            %         dLdSout - (Optional) Derivative of loss with respect
            %                   to state output
            %         memory  - Memory value from forward function
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Name Layer and Specify Superclasses

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `projectAndReshapeLayer`.

```
classdef projectAndRehapeLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)
    ...
end
```

If you do not specify a backward function, then the layer functions, by default, receive *unformatted* `dlarray` objects as input. To specify that the layer receives *formatted* `dlarray` objects as input and also outputs formatted `dlarray` objects, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

Because a project and reshape layer outputs data with different dimensions as the input data, that is, it outputs data with added spatial dimensions, the layer must also inherit from `nnet.layer.Formattable`. This enables the layer to receive and output formatted `dlarray` objects.

Next, specify to inherit from both the `nnet.layer.Layer` and `nnet.layer.Formattable` superclasses.

```
classdef projectAndRehapeLayer < nnet.layer.Layer & nnet.layer.Formattable
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = projectAndReshapeLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `projectAndReshapeLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs,

then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

A project and reshape layer requires an additional property that holds the layer output size. Specify a single property with name `OutputSize` in the `properties` section.

```
properties
    % Output size
    OutputSize
end
```

A project and reshape layer has two learnable parameters: the weights and the biases of the fully connect operation. Declare these learnable parameter in the `properties (Learnable)` section and call the parameters `Weights` and `Bias`, respectively.

```
properties (Learnable)
% Layer learnable parameters
    Weights
    Bias
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The project and reshape layer constructor function requires two input arguments:

- Layer output size
- Number of channels
- Layer name (optional, with default name `''`)

In the constructor function `projectAndReshapeLayer`, specify the two required input arguments named `outputSize` and `numChannels`, and the optional arguments as name-value pairs with the name `NameValueArgs`. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = projectAndReshapeLayer(outputSize,numChannels,NameValueArgs)
    % layer = projectAndReshapeLayer(outputSize,numChannels)
    % creates a projectAndReshapeLayer object that projects and
    % reshapes the input to the specified output size using and
    % specifies the number of input channels.
    %
    % layer = projectAndReshapeLayer(outputSize,numChannels,'Name',name)
    % also specifies the layer name.

    ...
end
```

Parse Input Arguments

Parse the input arguments using an `arguments` block. List the arguments in the same order as the function syntax and specify the default values. Then, extract the values from the `NameValueArgs` input.

```
    % Parse input arguments.
    arguments
        outputSize
        numChannels
```

```
    NameValueArgs.Name = ''
end

name = NameValueArgs.Name;
```

Initialize Layer Properties

Initialize the layer properties, including learnable parameters in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the `Name` property to the input argument `name`.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "Project and reshape layer with output size " + join(string(outpu
```

Specify the type of the layer by setting the `Type` property. The value of `Type` appears when the layer is displayed in a `Layer` array.

```
% Set layer type.
layer.Type = "Project and Reshape";
```

Set the layer property `OutputSize` to the specified input value.

```
% Set output size.
layer.OutputSize = outputSize;
```

A project and reshape layer applies a fully connect operation to project the input to batch of images. Initialize the weights using the Glorot initializer and initialize the bias with an array of zeros. The functions `initializeGlorot` and `initializeZeros` are attached to the example "Train Generative Adversarial Network (GAN)" on page 3-76 as supporting files. To access these functions, open this example as a live script. For more information about initializing learnable parameters for deep learning operations, see "Initialize Learnable Parameters for Model Function" on page 18-292.

```
% Initialize fully connect weights and bias.
sz = [prod(outputSize) numChannels];
numOut = prod(outputSize);
numIn = numChannels;
layer.Weights = initializeGlorot(sz,numOut,numIn);
layer.Bias = initializeZeros([prod(outputSize) 1]);
```

View the completed constructor function.

```
function layer = projectAndReshapeLayer(outputSize,numChannels,NameValueArgs)
    % layer = projectAndReshapeLayer(outputSize,numChannels)
    % creates a projectAndReshapeLayer object that projects and
    % reshapes the input to the specified output size using and
    % specifies the number of input channels.
    %
    % layer = projectAndReshapeLayer(outputSize,numChannels,'Name',name)
    % also specifies the layer name.

    % Parse input arguments.
```

```
        arguments
            outputSize
            numChannels
            NameValueArgs.Name = '';
        end

        name = NameValueArgs.Name;

        % Set layer name.
        layer.Name = name;

        % Set layer description.
        layer.Description = "Project and reshape layer with output size " + join(string(outpu

        % Set layer type.
        layer.Type = "Project and Reshape";

        % Set output size.
        layer.OutputSize = outputSize;

        % Initialize fully connect weights and bias.
        sz = [prod(outputSize) numChannels];
        numOut = prod(outputSize);
        numIn = numChannels;
        layer.Weights = initializeGlorot(sz,numOut,numIn);
        layer.Bias = initializeZeros([prod(outputSize) 1]);
    end
```

With this constructor function, the command `projectAndReshapeLayer([4 4 512],100,'Name','proj');` creates a project and reshape layer with name `'proj'` that projects the input arrays with size 100 to a batch of 512 4-by-4 images.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

Because a project and reshape layer has only one input and one output, the syntax for `predict` for a project and reshape layer is `Z = predict(layer,X)`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for a custom backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4

Layer Input	Input Size	Observation Dimension
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For layers that output sequences, the layers can output sequences of any length or output data with no time dimension. Note that when training a network that outputs sequences using the `trainNetwork` function, the lengths of the input and output sequences must match.

Because the custom layer inherits from the `nnet.layer.Formattable` class, the layer receives formatted `dlarray` objects with labels corresponding to the output of the previous layer.

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The `forward` function syntax depends on the type of layer:

- `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = forward(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.
- `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `forward` function of the custom layer, use the `forward` function of the `dlnetwork` object. Using the `dlnetwork` object `forward` function ensures that the software uses the correct layer operations for training.

The project and reshape operation consists of a three operations:

- Apply a fully connect operations with the learnable weights and biases.
- Reshape the output to the specified output size.
- Relabel the dimensions so that the output has format `'SSCB'` (spatial, spatial, channel, batch)

Implement this operation in the `predict` function. The project and reshape layer does not require memory or a different forward function for training, so you can remove the `forward` function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

```
function Z = predict(layer, X)
    % Forward input data through the layer at prediction time and
    % output the result.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Input data, specified as a formatted dlarray
    %                 with a 'C' and optionally a 'B' dimension.
    % Outputs:
    %         Z     - Output of layer forward function returned as
    %                 a formatted dlarray with format 'SSCB'.

    % Fully connect.
    weights = layer.Weights;
    bias = layer.Bias;
    X = fullyconnect(X,weights,bias);

    % Reshape.
    outputSize = layer.OutputSize;
    Z = reshape(X, outputSize(1), outputSize(2), outputSize(3), []);

    % Relabel.
    Z = dlarray(Z,'SSCB');
end
```

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

Because the `predict` function only uses functions that support `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed layer class file.

```
classdef projectAndReshapeLayer < nnet.layer.Layer & nnet.layer.Formattable
    % Example project and reshape layer.

    properties
        % Output size
        OutputSize
```

```
end

properties (Learnable)
% Layer learnable parameters
    Weights
    Bias
end

methods
    function layer = projectAndReshapeLayer(outputSize,numChannels,NameValueArgs)
        % layer = projectAndReshapeLayer(outputSize,numChannels)
        % creates a projectAndReshapeLayer object that projects and
        % reshapes the input to the specified output size using and
        % specifies the number of input channels.
        %
        % layer = projectAndReshapeLayer(outputSize,numChannels,'Name',name)
        % also specifies the layer name.

        % Parse input arguments.
        arguments
            outputSize
            numChannels
            NameValueArgs.Name = '';
        end

        name = NameValueArgs.Name;

        % Set layer name.
        layer.Name = name;

        % Set layer description.
        layer.Description = "Project and reshape layer with output size " + join(string(outpu

        % Set layer type.
        layer.Type = "Project and Reshape";

        % Set output size.
        layer.OutputSize = outputSize;

        % Initialize fully connect weights and bias.
        sz = [prod(outputSize) numChannels];
        numOut = prod(outputSize);
        numIn = numChannels;
        layer.Weights = initializeGlorot(sz,numOut,numIn);
        layer.Bias = initializeZeros([prod(outputSize) 1]);
    end

    function Z = predict(layer, X)
        % Forward input data through the layer at prediction time and
        % output the result.
        %
        % Inputs:
        %         layer - Layer to forward propagate through
        %         X     - Input data, specified as a formatted dlarray
        %                 with a 'C' and optionally a 'B' dimension.
        % Outputs:
        %         Z     - Output of layer forward function returned as
        %                 a formatted dlarray with format 'SSCB'.
```

```
            % Fully connect.
            weights = layer.Weights;
            bias = layer.Bias;
            X = fullyconnect(X,weights,bias);

            % Reshape.
            outputSize = layer.OutputSize;
            Z = reshape(X, outputSize(1), outputSize(2), outputSize(3), []);

            % Relabel.
            Z = dlarray(Z,'SSCB');
        end
    end
end
```

GPU Compatibility

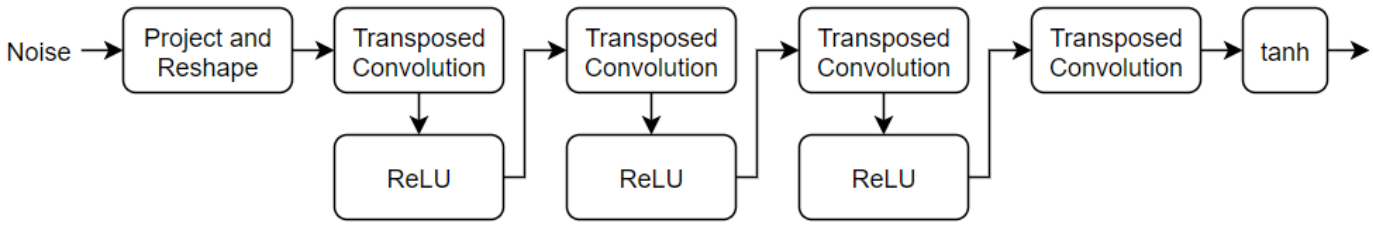
If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

In this example, the MATLAB functions used in `predict` all support `dlarray` objects, so the layer is GPU compatible.

Include Custom Layer in Network

Define the following generator network architecture for a GAN, which generates images from 1-by-1-by-100 arrays of random values:



This network:

- Converts the random vectors of size 100 to 7-by-7-by-128 arrays using a *project and reshape* layer.
- Upscales the resulting arrays to 64-by-64-by-3 arrays using a series of transposed convolution layers and ReLU layers.

Define this network architecture as a layer graph and specify the following network properties.

- For the transposed convolution layers, specify 5-by-5 filters with a decreasing number of filters for each layer, a stride of 2, and cropping of the output on each edge.
- For the final transposed convolution layer, specify three 5-by-5 filters corresponding to the three RGB channels of the generated images, and the output size of the previous layer.
- At the end of the network, include a tanh layer.

To project and reshape the noise input, use the custom layer `projectAndReshapeLayer.`

```
filterSize = 5;
numFilters = 64;
numLatentInputs = 100;

projectionSize = [4 4 512];

layersG = [
    featureInputLayer(numLatentInputs,'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'Name','proj');
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','same','Name','tconv2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(filterSize,numFilters,'Stride',2,'Cropping','same','Name','tconv3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(filterSize,3,'Stride',2,'Cropping','same','Name','tconv4')
    tanhLayer('Name','tanh')];

lgraphG = layerGraph(layersG);
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetG = dlnetwork(lgraphG);
```

See Also

`dlarray` | `functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Specify Custom Layer Backward Function” on page 18-107
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Custom Recurrent Deep Learning Layer

If Deep Learning Toolbox does not provide the layer you require for your task, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

When defining the layer functions, you can use `dlarray` objects. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `"S"`, `"T"`, `"C"`, and `"B"` labels, respectively. For unspecified and other dimensions, use the `"U"` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `DataFormat` option.

Using formatted `dlarray` objects in custom layers also allows you to define layers where the inputs and outputs have different formats, such as layers that permute, add, or remove dimensions. For example, you can define a layer that takes as input a mini-batch of images with format `"SSCB"` (spatial, spatial, channel, batch) and output a mini-batch of sequences with format `"CBT"` (channel, batch, time). Using formatted `dlarray` objects also allows you to define layers that can operate on data with different input formats, for example, layers that support inputs with formats `"SSCB"` (spatial, spatial, channel, batch) and `"CBT"` (channel, batch, time).

`dlarray` objects also enable support for automatic differentiation. This means that if your forward functions fully support `dlarray` objects, then defining the backward function is optional.

To enable support for using formatted `dlarray` objects in custom layer forward functions, also inherit from the `nnet.layer.Formattable` class when defining the custom layer. For an example, see “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61.

This example shows how to define a peephole LSTM layer [1], which is a recurrent layer with learnable parameters, and use it in a neural network. A peephole LSTM layer is a variant of an LSTM layer, where the gate calculations use the layer cell state.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Define layer constructor function here.
        end

        function [Z,state] = predict(layer,X)
            % Forward input data through the layer at prediction time and
            % output the result and updated state.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Input data
            % Outputs:
            %         Z     - Output of layer forward function
            %         state - (Optional) Updated layer state.
            %
            %  - For layers with multiple inputs, replace X with X1,...,XN,
            %    where N is the number of inputs.
            %  - For layers with multiple outputs, replace Z with
            %    Z1,...,ZM, where M is the number of outputs.
            %  - For layers with multiple state parameters, replace state
            %    with state1,...,stateK, where K is the number of state
            %    parameters.
```

```
    % Define layer predict function here.
end

function [Z,state,memory] = forward(layer,X)
    % (Optional) Forward input data through the layer at training
    % time and output the result, updated state, and a memory
    % value.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Layer input data
    % Outputs:
    %         Z      - Output of layer forward function
    %         state  - (Optional) Updated layer state
    %         memory - (Optional) Memory value for custom backward
    %                  function
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer forward function here.
end

function layer = resetState(layer)
    % (Optional) Reset layer state.

    % Define reset state function here.
end

function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
    % (Optional) Backward propagate the derivative of the loss
    % function through the layer.
    %
    % Inputs:
    %         layer   - Layer to backward propagate through
    %         X       - Layer input data
    %         Z       - Layer output data
    %         dLdZ    - Derivative of loss with respect to layer
    %                   output
    %         dLdSout - (Optional) Derivative of loss with respect
    %                   to state output
    %         memory  - Memory value from forward function
    % Outputs:
    %         dLdX   - Derivative of loss with respect to layer input
    %         dLdW   - (Optional) Derivative of loss with respect to
    %                  learnable parameter
    %         dLdSin - (Optional) Derivative of loss with respect to
    %                  state input
    %
    %  - For layers with state parameters, the backward syntax must
    %    include both dLdSout and dLdSin, or neither.
    %  - For layers with multiple inputs, replace X and dLdX with
    %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
    %    the number of inputs.
    %  - For layers with multiple outputs, replace Z and dlZ with
    %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
    %    number of outputs.
    %  - For layers with multiple learnable parameters, replace
    %    dLdW with dLdW1,...,dLdWP, where P is the number of
    %    learnable parameters.
    %  - For layers with multiple state parameters, replace dLdSin
    %    and dLdSout with dLdSin1,...,dLdSinK and
    %    dLdSout1,...dldSoutK, respectively, where K is the number
    %    of state parameters.

    % Define layer backward function here.
end
```

```
    end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `peepholeLSTMLayer`. To allow the layer to output different data formats, for example data with format `"CBT"` (channel, batch, time) for sequence output and format `"CB"` (channel, batch) for single time step or feature output, also include the `nnet.layer.Formattable` mixin.

```
classdef peepholeLSTMLayer < nnet.layer.Layer & nnet.layer.Formattable
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = peepholeLSTMLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `peepholeLSTMLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties, State, and Learnable Parameters

Declare the layer properties in the `properties` section, the layer states in the `properties (State)` section, and the learnable parameters in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.

Property	Description
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

Declare the following layer properties in the `properties` section:

- `NumHiddenUnits` — Number of hidden units in peephole LSTM operation.
- `OutputMode` — Flag indicating whether layer returns a sequence or a single time step.

```
properties
    % Layer properties.

    NumHiddenUnits
    OutputMode
end
```

A peephole LSTM layer has four learnable parameters: the input weights, the recurrent weights, the peephole weights, and the bias. Declare these learnable parameters in the `properties (Learnable)` section with names `InputWeights`, `RecurrentWeights`, `PeepholeWeights`, and `Bias`, respectively.

```
properties (Learnable)
    % Layer learnable parameters.

    InputWeights
    RecurrentWeights
    PeepholeWeights
    Bias
end
```

A peephole LSTM layer has two state parameters: the hidden state and the cell state. Declare these state parameters in the `properties (State)` section with names `HiddenState` and `CellState`, respectively.

```
properties (State)
    % Layer state parameters.

    HiddenState
    CellState
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The peephole LSTM layer constructor function requires two input arguments (the number of hidden units and the number of input channels) and two optional arguments (the layer name and output mode). Specify two input arguments named `numHiddenUnits` and `inputSize` in the `peepholeLSTMLayer` function that correspond to the number of hidden units and the number of input channels, respectively. Specify the optional input arguments as a single argument with name `args`. Add a comment to the top of the function that explains the syntaxes of the function.

```
function layer = peepholeLSTMLayer(numHiddenUnits,inputSize,args)
    %PEEPHOLELSTMLAYER Peephole LSTM Layer
    %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize)
    %   creates a peephole LSTM layer with the specified number of
    %   hidden units and input channels.
    %
    %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize,Name=Value)
    %   creates a peephole LSTM layer and specifies additional
    %   options using one or more name-value arguments:
    %
    %      Name       - Name for the layer, specified as a string.
    %                   The default is "".
    %
    %      OutputMode - Output mode, specified as one of the
    %                   following:
    %                      "sequence" - Output the entire sequence
    %                                   of data.
    %
    %                      "last"     - Output the last time step
    %                                   of the data.
    %                   The default is "sequence".

    ...
end
```

Initialize Layer Properties

Initialize the layer properties, including the learnable and state parameters in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Parse the input arguments using an `arguments` block and set the `Name` and output properties.

```
arguments
    numHiddenUnits
    inputSize
    args.Name = "";
    args.OutputMode = "sequence"
end

layer.NumHiddenUnits = numHiddenUnits;
layer.Name = args.Name;
layer.OutputMode = args.OutputMode;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "Peephole LSTM with " + numHiddenUnits + " hidden units";
```

Initialize the learnable parameters. Initialize the input weights using Glorot initialization. Initialize the recurrent weights using orthogonal initialization. Initialize the bias using unit-forget-gate normalization. This code uses the helper functions `initializeGlorot`, `initializeOrthogonal`, and `initializeUnitForgetGate`. To access these functions, open the example as a live script. For more information about initializing weights, see “Initialize Learnable Parameters for Model Function” on page 18-292.

Note that the recurrent weights of a peephole LSTM layer and standard LSTM layers have different sizes. A peephole LSTM layer does not require recurrent weights for the cell candidate calculation, so the recurrent weights is a `3*NumHiddenUnits`-by-`NumHiddenUnits` array.

```
% Initialize weights and bias.
sz = [4*numHiddenUnits inputSize];
numOut = 4*numHiddenUnits;
numIn = inputSize;
layer.InputWeights = initializeGlorot(sz,numOut,numIn);

sz = [4*numHiddenUnits numHiddenUnits];
layer.RecurrentWeights = initializeOrthogonal(sz);

sz = [3*numHiddenUnits 1];
numOut = 3*numHiddenUnits;
numIn = 1;
layer.PeepholeWeights = initializeGlorot(sz,numOut,numIn);

layer.Bias = initializeUnitForgetGate(numHiddenUnits);
```

Initialize the layer state parameters. For convenience, use the `resetState` function defined in the section.

```
% Initialize layer states.
layer = resetState(layer);
```

View the completed constructor function.

```
function layer = peepholeLSTMLayer(numHiddenUnits,inputSize,args)
    %PEEPHOLELSTMLAYER Peephole LSTM Layer
    %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize)
```

```
            %   creates a peephole LSTM layer with the specified number of
            %   hidden units and input channels.
            %
            %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize,Name=Value)
            %   creates a peephole LSTM layer and specifies additional
            %   options using one or more name-value arguments:
            %
            %      Name       - Name for the layer, specified as a string.
            %                   The default is "".
            %
            %      OutputMode - Output mode, specified as one of the
            %                   following:
            %                      "sequence" - Output the entire sequence
            %                                   of data.
            %
            %                      "last"     - Output the last time step
            %                                   of the data.
            %                   The default is "sequence".

            % Parse input arguments.
            arguments
                numHiddenUnits
                inputSize
                args.Name = "";
                args.OutputMode = "sequence";
            end

            layer.NumHiddenUnits = numHiddenUnits;
            layer.Name = args.Name;
            layer.OutputMode = args.OutputMode;

            % Set layer description.
            layer.Description = "Peephole LSTM with " + numHiddenUnits + " hidden units";

            % Initialize weights and bias.
            sz = [4*numHiddenUnits inputSize];
            numOut = 4*numHiddenUnits;
            numIn = inputSize;
            layer.InputWeights = initializeGlorot(sz,numOut,numIn);

            sz = [4*numHiddenUnits numHiddenUnits];
            layer.RecurrentWeights = initializeOrthogonal(sz);

            sz = [3*numHiddenUnits 1];
            numOut = 3*numHiddenUnits;
            numIn = 1;
            layer.PeepholeWeights = initializeGlorot(sz,numOut,numIn);

            layer.Bias = initializeUnitForgetGate(numHiddenUnits);

            % Initialize layer states.
            layer = resetState(layer);
        end
```

With this constructor function, the command `peepholeLSTMLayer(200,12,OutputMode="last",Name="peephole")` creates a peephole

LSTM layer with 200 hidden units, an input size of 12, and name `"peephole"`, and outputs the last time step of the peephole LSTM operation.

Create Predict Function

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

Because a peephole LSTM layer has only one input, one output, and two state parameters, the syntax for `predict` for a peephole LSTM layer is `[Z,hiddenState,cellState] = predict(layer,X)`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for a custom backward function, you must also create a function named `forward`.

Because the layer inherits from `nnet.layer.Formattable`, the layer inputs are formatted `dlarray` objects and the `predict` function must also output data as formatted `dlarray` objects.

The hidden state at time step t is given by

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \odot \mathbf{o}_t,$$

where ⊙ denotes the Hadamard product (element-wise multiplication of vectors).

The cell state at time step t is given by

$$\mathbf{c}_t = \mathbf{g}_t \odot \mathbf{i}_t + \mathbf{c}_{t-1} \odot \mathbf{f}_t .$$

The following formulas describe the components at time step t.

Component	Formula
Input gate	$i_t = \sigma_g(W_i\mathbf{x}_t + R_i\mathbf{h}_{t-1} + \mathbf{p}_i \odot \mathbf{c}_{t-1} + \mathbf{b}_i)$
Forget gate	$f_t = \sigma_g\left(W_f\mathbf{x}_t + R_f\mathbf{h}_{t-1} + \mathbf{p}_f \odot \mathbf{c}_{t-1} + \mathbf{b}_f\right)$
Cell candidate	$g_t = \sigma_c\left(W_g\mathbf{x}_t + R_h\mathbf{h}_{t-1} + b_g\right)$
Output gate	$o_t = \sigma_g(W_o\mathbf{x}_t + R_o\mathbf{h}_{t-1} + \mathbf{p}_o \odot \mathbf{c}_t + \mathbf{b}_o)$

Note that the output gate calculation requires the updated cell state $\mathbf{c}_t$.

In these calculations, σ_g and σ_c denote the gate and state activation functions. For peephole LSTM layers, use the sigmoid and hyperbolic tangent functions as the gate and state activation functions, respectively.

Implement this operation in the `predict` function. Because the layer does not require a different forward function for training and memory value for a custom backward function, you can remove the `forward` function from the class file. Add a comment to the top of the function that explains the syntaxes of the function.

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

```
function [Z,cellState,hiddenState] = predict(layer,X)
    %PREDICT Peephole LSTM predict function
    %   [Z,hiddenState,cellState] = predict(layer,X) forward
    %   propagates the data X through the layer and returns the
    %   layer output Z and the updated hidden and cell states. X
    %   is a dlarray with format "CBT" and Z is a dlarray with
    %   format "CB" or "CBT", depending on the layer OutputMode
    %   property.

    % Initialize sequence output.
    numHiddenUnits = layer.NumHiddenUnits;
    miniBatchSize = size(X,finddim(X,"B"));
    numTimeSteps = size(X,finddim(X,"T"));

    if layer.OutputMode == "sequence"
        Z = zeros(numHiddenUnits,miniBatchSize,numTimeSteps,"like",X);
        Z = dlarray(Z,"CBT");
    end

    % Calculate WX + b.
    X = stripdims(X);
    WX = pagemtimes(layer.InputWeights,X) + layer.Bias;

    % Indices of concatenated weight arrays.
    idx1 = 1:numHiddenUnits;
    idx2 = 1+numHiddenUnits:2*numHiddenUnits;
    idx3 = 1+2*numHiddenUnits:3*numHiddenUnits;
    idx4 = 1+3*numHiddenUnits:4*numHiddenUnits;

    % Initial states.
```

```
        hiddenState = layer.HiddenState;
        cellState = layer.CellState;

        % Loop over time steps.
        for t = 1:numTimeSteps
            % Calculate R*h_{t-1}.
            Rht = layer.RecurrentWeights * hiddenState;

            % Calculate p*c_{t-1}.
            pict = layer.PeepholeWeights(idx1) .* cellState;
            pfct = layer.PeepholeWeights(idx2) .* cellState;

            % Gate calculations.
            it = sigmoid(WX(idx1,:,t) + Rht(idx1,:) + pict);
            ft = sigmoid(WX(idx2,:,t) + Rht(idx2,:) + pfct);
            gt = tanh(WX(idx3,:,t) + Rht(idx3,:));

            % Calculate ot using updated cell state.
            cellState = gt .* it + cellState .* ft;
            poct = layer.PeepholeWeights(idx3) .* cellState;
            ot = sigmoid(WX(idx4,:,t) + Rht(idx4,:) + poct);

            % Update hidden state.
            hiddenState = tanh(cellState) .* ot;

            % Update sequence output.
            if layer.OutputMode == "sequence"
                Z(:,:,t) = hiddenState;
            end
        end

        % Last time step output.
        if layer.OutputMode == "last"
            Z = dlarray(hiddenState,"CB");
        end
    end
```

Because the `predict` function only uses functions that support `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Create Reset State Function

When `DAGNetwork` or `SeriesNetwork` objects contain layers with state parameters, you can make predictions and update the layer states using the `predictAndUpdateState` and `classifyAndUpdateState` functions. You can reset the network state using the `resetState` function.

The `resetState` function for `DAGNetwork`, `SeriesNetwork`, and `dlnetwork` objects, by default, has no effect on custom layers with state parameters. To define the layer behavior for the `resetState` function for network objects, define the optional layer `resetState` function in the layer definition that resets the state parameters.

The `resetState` function must have the syntax `layer = resetState(layer)`, where the returned layer has the state properties reset.

Create a function named `resetState` that resets the layer state parameters to vectors of zeros.

```
    function layer = resetState(layer)
        %RESETSTATE Reset layer state
        % layer = resetState(layer) resets the state properties of the
        % layer.

        numHiddenUnits = layer.NumHiddenUnits;
        layer.HiddenState = zeros(numHiddenUnits,1);
        layer.CellState = zeros(numHiddenUnits,1);
    end
```

Completed Layer

View the completed layer class file.

```
classdef peepholeLSTMLayer < nnet.layer.Layer & nnet.layer.Formattable
    %PEEPHOLELSTMLAYER Peephole LSTM Layer

    properties
        % Layer properties.

        NumHiddenUnits
        OutputMode
    end

    properties (Learnable)
        % Layer learnable parameters.

        InputWeights
        RecurrentWeights
        PeepholeWeights
        Bias
    end

    properties (State)
        % Layer state parameters.

        HiddenState
        CellState
    end

    methods
        function layer = peepholeLSTMLayer(numHiddenUnits,inputSize,args)
            %PEEPHOLELSTMLAYER Peephole LSTM Layer
            %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize)
            %   creates a peephole LSTM layer with the specified number of
            %   hidden units and input channels.
            %
            %   layer = peepholeLSTMLayer(numHiddenUnits,inputSize,Name=Value)
            %   creates a peephole LSTM layer and specifies additional
            %   options using one or more name-value arguments:
            %
            %      Name       - Name for the layer, specified as a string.
            %                   The default is "".
            %
            %      OutputMode - Output mode, specified as one of the
            %                   following:
            %                      "sequence" - Output the entire sequence
            %                                   of data.
            %
            %                      "last"     - Output the last time step
            %                                   of the data.
            %                   The default is "sequence".

            % Parse input arguments.
            arguments
                numHiddenUnits
                inputSize
                args.Name = "";
```

```
            args.OutputMode = "sequence";
        end

        layer.NumHiddenUnits = numHiddenUnits;
        layer.Name = args.Name;
        layer.OutputMode = args.OutputMode;

        % Set layer description.
        layer.Description = "Peephole LSTM with " + numHiddenUnits + " hidden units";

        % Initialize weights and bias.
        sz = [4*numHiddenUnits inputSize];
        numOut = 4*numHiddenUnits;
        numIn = inputSize;
        layer.InputWeights = initializeGlorot(sz,numOut,numIn);

        sz = [4*numHiddenUnits numHiddenUnits];
        layer.RecurrentWeights = initializeOrthogonal(sz);

        sz = [3*numHiddenUnits 1];
        numOut = 3*numHiddenUnits;
        numIn = 1;
        layer.PeepholeWeights = initializeGlorot(sz,numOut,numIn);

        layer.Bias = initializeUnitForgetGate(numHiddenUnits);

        % Initialize layer states.
        layer = resetState(layer);
    end

    function [Z,cellState,hiddenState] = predict(layer,X)
        %PREDICT Peephole LSTM predict function
        %   [Z,hiddenState,cellState] = predict(layer,X) forward
        %   propagates the data X through the layer and returns the
        %   layer output Z and the updated hidden and cell states. X
        %   is a dlarray with format "CBT" and Z is a dlarray with
        %   format "CB" or "CBT", depending on the layer OutputMode
        %   property.

        % Initialize sequence output.
        numHiddenUnits = layer.NumHiddenUnits;
        miniBatchSize = size(X,finddim(X,"B"));
        numTimeSteps = size(X,finddim(X,"T"));

        if layer.OutputMode == "sequence"
            Z = zeros(numHiddenUnits,miniBatchSize,numTimeSteps,"like",X);
            Z = dlarray(Z,"CBT");
        end

        % Calculate WX + b.
        X = stripdims(X);
        WX = pagemtimes(layer.InputWeights,X) + layer.Bias;

        % Indices of concatenated weight arrays.
        idx1 = 1:numHiddenUnits;
        idx2 = 1+numHiddenUnits:2*numHiddenUnits;
        idx3 = 1+2*numHiddenUnits:3*numHiddenUnits;
        idx4 = 1+3*numHiddenUnits:4*numHiddenUnits;
```

```
            % Initial states.
            hiddenState = layer.HiddenState;
            cellState = layer.CellState;

            % Loop over time steps.
            for t = 1:numTimeSteps
                % Calculate R*h_{t-1}.
                Rht = layer.RecurrentWeights * hiddenState;

                % Calculate p*c_{t-1}.
                pict = layer.PeepholeWeights(idx1) .* cellState;
                pfct = layer.PeepholeWeights(idx2) .* cellState;

                % Gate calculations.
                it = sigmoid(WX(idx1,:,t) + Rht(idx1,:) + pict);
                ft = sigmoid(WX(idx2,:,t) + Rht(idx2,:) + pfct);
                gt = tanh(WX(idx3,:,t) + Rht(idx3,:));

                % Calculate ot using updated cell state.
                cellState = gt .* it + cellState .* ft;
                poct = layer.PeepholeWeights(idx3) .* cellState;
                ot = sigmoid(WX(idx4,:,t) + Rht(idx4,:) + poct);

                % Update hidden state.
                hiddenState = tanh(cellState) .* ot;

                % Update sequence output.
                if layer.OutputMode == "sequence"
                    Z(:,:,t) = hiddenState;
                end
            end

            % Last time step output.
            if layer.OutputMode == "last"
                Z = dlarray(hiddenState,"CB");
            end
        end

        function layer = resetState(layer)
            %RESETSTATE Reset layer state
            % layer = resetState(layer) resets the state properties of the
            % layer.

            numHiddenUnits = layer.NumHiddenUnits;
            layer.HiddenState = zeros(numHiddenUnits,1);
            layer.CellState = zeros(numHiddenUnits,1);
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

In this example, the MATLAB functions used in `predict` all support `dlarray` objects, so the layer is GPU compatible.

Include Custom Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox. This section shows how to create and train a network for sequence classification using the peephole LSTM layer you created earlier.

Load the example training data.

```
[XTrain,TTrain] = japaneseVowelsTrainData;
```

Define the network architecture. Create a layer array containing a peephole LSTM layer.

```
inputSize = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [
    sequenceInputLayer(inputSize)
    peepholeLSTMLayer(numHiddenUnits,inputSize,OutputMode="last")
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options and train the network. Train with a mini-batch size of 27 and left-pad the data.

```
options = trainingOptions("adam",MiniBatchSize=27,SequencePaddingDirection="left");
net = trainNetwork(XTrain,TTrain,layers,options);
```

```
Training on single CPU.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:01 |        3.70% |       2.2060 |          0.0010 |
|       5 |          50 |       00:00:19 |       92.59% |       0.5917 |          0.0010 |
|      10 |         100 |       00:00:37 |       92.59% |       0.2182 |          0.0010 |
|      15 |         150 |       00:00:54 |      100.00% |       0.0588 |          0.0010 |
|      20 |         200 |       00:01:16 |       96.30% |       0.0843 |          0.0010 |
|      25 |         250 |       00:01:37 |      100.00% |       0.0331 |          0.0010 |
|      30 |         300 |       00:01:58 |      100.00% |       0.0130 |          0.0010 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Evaluate the network performance by predicting on new data and calculating the accuracy.

```
[XTest,TTest] = japaneseVowelsTestData;
YTest = classify(net,XTest,MiniBatchSize=27);
accuracy = mean(YTest==TTest)
```

```
accuracy = 0.8757
```

References

[1] Greff, Klaus, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. "LSTM: A search space odyssey." *IEEE transactions on neural networks and learning systems* 28, no. 10 (2016): 2222–2232.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

Related Examples

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Specify Custom Layer Backward Function” on page 18-107
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Custom Classification Output Layer

Tip To construct a classification output layer with cross entropy loss for k mutually exclusive classes, use `classificationLayer`. If you want to use a different loss function for your classification problems, then you can define a custom classification output layer using this example as a guide.

This example shows how to define a custom classification output layer with the sum of squares error (SSE) loss and use it in a convolutional neural network.

To define a custom classification output layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer – Give the layer a name so it can be used in MATLAB.
2. Declare the layer properties – Specify the properties of the layer.
3. Create a constructor function (optional) – Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with `''` at creation.
4. Create a forward loss function – Specify the loss between the predictions and the training targets.
5. Create a backward loss function (optional) – Specify the derivative of the loss with respect to the predictions. If you do not specify a backward loss function, then the forward loss function must support `dlarray` objects.

A classification SSE layer computes the sum of squares error loss for classification problems. SSE is an error measure between two continuous random variables. For predictions Y and training targets T, the SSE loss between Y and T is given by

$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}(Y_{ni} - T_{ni})^2,$$

where N is the number of observations and K is the number of classes.

Classification Output Layer Template

Copy the classification output layer template into a new file in MATLAB. This template outlines the structure of a classification output layer and includes the functions that define the layer behavior.

```
classdef myClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    methods
        function layer = myClassificationLayer()
            % (Optional) Create a myClassificationLayer.

            % Layer constructor function goes here.
        end
```

```
        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myClassificationLayer` with `sseClassificationLayer`.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    ...
end
```

Next, rename the `myClassificationLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = sseClassificationLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `sseClassificationLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Layer Properties

Declare the layer properties in the `properties` section.

By default, custom output layers have the following properties:

- `Name` – Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
- `Description` – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays `"Classification Output"` or `"Regression Output"`.
- `Type` – Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- `Classes` – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. If `Classes` is `'auto'`, then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`.

Custom regression layers also have the following property:

- `ResponseNames` – Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is `{}`.

If the layer has no other properties, then you can omit the `properties` section.

In this example, the layer does not require any additional properties, so you can remove the `properties` section.

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

Specify the input argument `name` to assign to the `Name` property at creation. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = sseClassificationLayer(name)
    % layer = sseClassificationLayer(name) creates a sum of squares
    % error classification layer and specifies the layer name.

    ...
end
```

Initialize Layer Properties

Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Give the layer a one-line description by setting the `Description` property of the layer. Set the `Name` property to the input argument `name`.

```
function layer = sseClassificationLayer(name)
    % layer = sseClassificationLayer(name) creates a sum of squares
```

```
        % error classification layer and specifies the layer name.

        % Set layer name.
        layer.Name = name;

        % Set layer description.
        layer.Description = 'Sum of squares error';
    end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the SSE loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` represents the training targets.

For classification problems, the dimensions of `T` depend on the type of problem.

Classification Task	Input Size	Observation Dimension
2-D image classification	1-by-1-by-*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	4
3-D image classification	1-by-1-by-1-by-*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	5
Sequence-to-label classification	*K*-by-*N*, where *K* is the number of classes and *N* is the number of observations.	2
Sequence-to-sequence classification	*K*-by-*N*-by-*S*, where *K* is the number of classes, *N* is the number of observations, and *S* is the sequence length.	2

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, to ensure that `Y` is a 4-D array of prediction scores for *K* classes, you can include a fully connected layer of size *K* followed by a softmax layer before the output layer.

A classification SSE layer computes the sum of squares error loss for classification problems. SSE is an error measure between two continuous random variables. For predictions *Y* and training targets *T*, the SSE loss between *Y* and *T* is given by

$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}(Y_{ni} - T_{ni})^2,$$

where *N* is the number of observations and *K* is the number of classes.

The inputs `Y` and `T` correspond to *Y* and *T* in the equation, respectively. The output `loss` corresponds to *L*. Add a comment to the top of the function that explains the syntaxes of the function.

```
        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the SSE loss between
            % the predictions Y and the training targets T.

            % Calculate sum of squares.
            sumSquares = sum((Y-T).^2);

            % Take mean over mini-batch.
            N = size(Y,4);
            loss = sum(sumSquares)/N;
        end
```

Because the `forwardLoss` function only uses functions that support `dlarray` objects, defining the `backwardLoss` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed classification output layer class file.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    % Example custom classification layer with sum of squares error loss.

    methods
        function layer = sseClassificationLayer(name)
            % layer = sseClassificationLayer(name) creates a sum of squares
            % error classification layer and specifies the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = 'Sum of squares error';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the SSE loss between
            % the predictions Y and the training targets T.

            % Calculate sum of squares.
            sumSquares = sum((Y-T).^2);

            % Take mean over mini-batch.
            N = size(Y,4);
            loss = sum(sumSquares)/N;
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss` all support `dlarray` objects, so the layer is GPU compatible.

Check Output Layer Validity

Check the layer validity of the custom classification output layer `sseClassificationLayer`.

Define a custom sum-of-squares error classification layer. To create this layer, save the file `sseClassificationLayer.m` in the current folder. Create an instance of the layer.

```
layer = sseClassificationLayer('sse');
```

Check the layer is valid using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a 1-by-1-by-*K*-by-*N* array inputs, where *K* is the number of classes, and *N* is the number of observations in the mini-batch.

```
validInputSize = [1 1 10];
checkLayer(layer,validInputSize,'ObservationDimension',4);
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 0.6486 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Include Custom Classification Output Layer in Network

You can use a custom output layer in the same way as any other output layer in Deep Learning Toolbox. This section shows how to create and train a network for classification using the custom classification output layer that you created earlier.

Load the example training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
```

Define a custom sum-of-squares error classification layer. To create this layer, save the file `sseClassificationLayer.m` in the current folder. Create an instance of the layer. Create a layer array including the custom classification output layer `sseClassificationLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(10)
    softmaxLayer
    sseClassificationLayer('sse')]

layers =
  7x1 Layer array with layers:

     1   ''      Image Input             28x28x1 images with 'zerocenter' normalization
     2   ''      Convolution             20 5x5 convolutions with stride [1  1] and padding [0  0
     3   ''      Batch Normalization     Batch normalization
     4   ''      ReLU                    ReLU
     5   ''      Fully Connected         10 fully connected layer
     6   ''      Softmax                 softmax
     7   'sse'   Classification Output   Sum of squares error
```

Set the training options and train the network.

```
options = trainingOptions('sgdm');
net = trainNetwork(XTrain,YTrain,layers,options);

Training on single CPU.
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:00 |        9.38% |       0.9944 |          0.0100 |
|       2 |          50 |       00:00:08 |       74.22% |       0.3544 |          0.0100 |
|       3 |         100 |       00:00:13 |       92.97% |       0.1306 |          0.0100 |
|       4 |         150 |       00:00:19 |       96.09% |       0.0964 |          0.0100 |
|       6 |         200 |       00:00:26 |       95.31% |       0.0772 |          0.0100 |
|       7 |         250 |       00:00:31 |       97.66% |       0.0446 |          0.0100 |
|       8 |         300 |       00:00:38 |       99.22% |       0.0201 |          0.0100 |
|       9 |         350 |       00:00:43 |       99.22% |       0.0262 |          0.0100 |
|      11 |         400 |       00:00:47 |      100.00% |       0.0080 |          0.0100 |
|      12 |         450 |       00:00:52 |      100.00% |       0.0059 |          0.0100 |
|      13 |         500 |       00:00:57 |      100.00% |       0.0092 |          0.0100 |
|      15 |         550 |       00:01:01 |      100.00% |       0.0064 |          0.0100 |
|      16 |         600 |       00:01:05 |      100.00% |       0.0020 |          0.0100 |
|      17 |         650 |       00:01:10 |      100.00% |       0.0039 |          0.0100 |
|      18 |         700 |       00:01:15 |      100.00% |       0.0023 |          0.0100 |
|      20 |         750 |       00:01:20 |      100.00% |       0.0024 |          0.0100 |
|      21 |         800 |       00:01:24 |      100.00% |       0.0019 |          0.0100 |
|      22 |         850 |       00:01:29 |      100.00% |       0.0017 |          0.0100 |
|      24 |         900 |       00:01:34 |      100.00% |       0.0020 |          0.0100 |
|      25 |         950 |       00:01:38 |      100.00% |       0.0012 |          0.0100 |
|      26 |        1000 |       00:01:43 |      100.00% |       0.0011 |          0.0100 |
|      27 |        1050 |       00:01:48 |       99.22% |       0.0104 |          0.0100 |
|      29 |        1100 |       00:01:53 |      100.00% |       0.0012 |          0.0100 |
|      30 |        1150 |       00:01:58 |      100.00% |       0.0011 |          0.0100 |
|      30 |        1170 |       00:02:00 |       99.22% |       0.0079 |          0.0100 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Evaluate the network performance by making predictions on new data and calculating the accuracy.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net, XTest);
accuracy = mean(YTest == YPred)
```

```
accuracy = 0.9846
```

See Also

`classificationLayer` | `checkLayer` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Regression Output Layer” on page 18-99
- “Specify Custom Output Layer Backward Loss Function” on page 18-113
- “Check Custom Layer Validity” on page 18-154

Define Custom Regression Output Layer

Tip To create a regression output layer with mean squared error loss, use `regressionLayer`. If you want to use a different loss function for your regression problems, then you can define a custom regression output layer using this example as a guide.

This example shows how to create a custom regression output layer with the mean absolute error (MAE) loss.

To define a custom regression output layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer – Give the layer a name so it can be used in MATLAB.
2. Declare the layer properties – Specify the properties of the layer.
3. Create a constructor function (optional) – Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with `''` at creation.
4. Create a forward loss function – Specify the loss between the predictions and the training targets.
5. Create a backward loss function (optional) – Specify the derivative of the loss with respect to the predictions. If you do not specify a backward loss function, then the forward loss function must support `dlarray` objects.

A regression MAE layer computes the mean absolute error loss for regression problems. MAE loss is an error measure between two continuous random variables. For predictions Y and training targets T, the MAE loss between Y and T is given by

$$L = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{1}{R}\sum_{i=1}^{R}|Y_{ni} - T_{ni}|\right),$$

where N is the number of observations and R is the number of responses.

Regression Output Layer Template

Copy the regression output layer template into a new file in MATLAB. This template outlines the structure of a regression output layer and includes the functions that define the layer behavior.

```
classdef myRegressionLayer < nnet.layer.RegressionLayer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    methods
        function layer = myRegressionLayer()
            % (Optional) Create a myRegressionLayer.

            % Layer constructor function goes here.
        end

        function loss = forwardLoss(layer, Y, T)
            % Return the loss between the predictions Y and the training
```

```
            % targets T.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         loss  - Loss between Y and T

            % Layer forward loss function goes here.
        end

        function dLdY = backwardLoss(layer, Y, T)
            % (Optional) Backward propagate the derivative of the loss
            % function.
            %
            % Inputs:
            %         layer - Output layer
            %         Y     – Predictions made by network
            %         T     – Training targets
            %
            % Output:
            %         dLdY  - Derivative of the loss with respect to the
            %                 predictions Y

            % Layer backward loss function goes here.
        end
    end
end
```

Name the Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myRegressionLayer` with `maeRegressionLayer`.

```
classdef maeRegressionLayer < nnet.layer.RegressionLayer
    ...
end
```

Next, rename the `myRegressionLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = maeRegressionLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `maeRegressionLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Layer Properties

Declare the layer properties in the `properties` section.

By default, custom output layers have the following properties:

- `Name` – Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
- `Description` – One-line description of the layer, specified as a character vector or a string scalar. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays `"Classification Output"` or `"Regression Output"`.
- `Type` – Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Custom classification layers also have the following property:

- `Classes` – Classes of the output layer, specified as a categorical vector, string array, cell array of character vectors, or `'auto'`. If `Classes` is `'auto'`, then the software automatically sets the classes at training time. If you specify the string array or cell array of character vectors `str`, then the software sets the classes of the output layer to `categorical(str,str)`.

Custom regression layers also have the following property:

- `ResponseNames` – Names of the responses, specified a cell array of character vectors or a string array. At training time, the software automatically sets the response names according to the training data. The default is `{}`.

If the layer has no other properties, then you can omit the `properties` section.

The layer does not require any additional properties, so you can remove the `properties` section.

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

To initialize the `Name` property at creation, specify the input argument `name`. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = maeRegressionLayer(name)
    % layer = maeRegressionLayer(name) creates a
    % mean-absolute-error regression layer and specifies the layer
    % name.

    ...
end
```

Initialize Layer Properties

Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Give the layer a one-line description by setting the `Description` property of the layer. Set the `Name` property to the input argument `name`. Set the description to describe the type of layer and its size.

```
function layer = maeRegressionLayer(name)
    % layer = maeRegressionLayer(name) creates a
    % mean-absolute-error regression layer and specifies the layer
    % name.

    % Set layer name.
```

```
        layer.Name = name;

        % Set layer description.
        layer.Description = 'Mean absolute error';
    end
```

Create Forward Loss Function

Create a function named `forwardLoss` that returns the MAE loss between the predictions made by the network and the training targets. The syntax for `forwardLoss` is `loss = forwardLoss(layer, Y, T)`, where `Y` is the output of the previous layer and `T` contains the training targets.

For regression problems, the dimensions of `T` also depend on the type of problem.

Regression Task	Input Size	Observation Dimension
2-D image regression	1-by-1-by-*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	4
2-D Image-to-image regression	*h*-by-*w*-by-*c*-by-`N`, where *h*, *w*, and *c* are the height, width, and number of channels of the output respectively, and *N* is the number of observations.	4
3-D image regression	1-by-1-by-1-by-*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	5
3-D Image-to-image regression	*h*-by-*w*-by-*d*-by-*c*-by-`N`, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the output respectively, and *N* is the number of observations.	5
Sequence-to-one regression	*R*-by-*N*, where *R* is the number of responses and *N* is the number of observations.	2
Sequence-to-sequence regression	*R*-by-*N*-by-*S*, where *R* is the number of responses, *N* is the number of observations, and *S* is the sequence length.	2

For example, if the network defines an image regression network with one response and has mini-batches of size 50, then `T` is a 4-D array of size 1-by-1-by-1-by-50.

The size of `Y` depends on the output of the previous layer. To ensure that `Y` is the same size as `T`, you must include a layer that outputs the correct size before the output layer. For example, for image regression with *R* responses, to ensure that `Y` is a 4-D array of the correct size, you can include a fully connected layer of size *R* before the output layer.

A regression MAE layer computes the mean absolute error loss for regression problems. MAE loss is an error measure between two continuous random variables. For predictions Y and training targets T, the MAE loss between Y and T is given by

$$L = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{1}{R}\sum_{i=1}^{R}|Y_{ni} - T_{ni}|\right),$$

where N is the number of observations and R is the number of responses.

The inputs `Y` and `T` correspond to Y and T in the equation, respectively. The output `loss` corresponds to L. To ensure that `loss` is scalar, output the mean loss over the mini-batch. Add a comment to the top of the function that explains the syntaxes of the function.

```
function loss = forwardLoss(layer, Y, T)
    % loss = forwardLoss(layer, Y, T) returns the MAE loss between
    % the predictions Y and the training targets T.

    % Calculate MAE.
    R = size(Y,3);
    meanAbsoluteError = sum(abs(Y-T),3)/R;

    % Take mean over mini-batch.
    N = size(Y,4);
    loss = sum(meanAbsoluteError)/N;
end
```

Because the `forwardLoss` function only uses functions that support `dlarray` objects, defining the `backwardLoss` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed regression output layer class file.

```
classdef maeRegressionLayer < nnet.layer.RegressionLayer
    % Example custom regression layer with mean-absolute-error loss.

    methods
        function layer = maeRegressionLayer(name)
            % layer = maeRegressionLayer(name) creates a
            % mean-absolute-error regression layer and specifies the layer
            % name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = 'Mean absolute error';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the MAE loss between
            % the predictions Y and the training targets T.
```

```
            % Calculate MAE.
            R = size(Y,3);
            meanAbsoluteError = sum(abs(Y-T),3)/R;

            % Take mean over mini-batch.
            N = size(Y,4);
            loss = sum(meanAbsoluteError)/N;
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss` in `maeRegressionLayer` all support `dlarray` objects, so the layer is GPU compatible.

Check Output Layer Validity

Check the layer validity of the custom classification output layer `maeRegressionLayer`.

Define a custom mean absolute error regression layer. To create this layer, save the file `maeRegressionLayer.m` in the current folder. Create an instance of the layer.

```
layer = maeRegressionLayer('mae');
```

Check the layer is valid using `checkLayer`. Specify the valid input size to be the size of a single observation of typical input to the layer. The layer expects a 1-by-1-by-R-by-*N* array inputs, where R is the number of responses, and *N* is the number of observations in the mini-batch.

```
validInputSize = [1 1 10];
checkLayer(layer,validInputSize,'ObservationDimension',4);
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 0.14109 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

Include Custom Regression Output Layer in Network

You can use a custom output layer in the same way as any other output layer in Deep Learning Toolbox. This section shows how to create and train a network for regression using the custom output layer you created earlier.

The example constructs a convolutional neural network architecture, trains a network, and uses the trained network to predict angles of rotated, handwritten digits. These predictions are useful for optical character recognition.

Load the example training data.

```
[XTrain,~,YTrain] = digitTrain4DArrayData;
```

Create a layer array including the regression output layer `maeRegressionLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(1)
    maeRegressionLayer('mae')]
```

```
layers =
  6x1 Layer array with layers:

     1   ''      Image Input           28x28x1 images with 'zerocenter' normalization
     2   ''      Convolution           20 5x5 convolutions with stride [1  1] and padding [0  0
     3   ''      Batch Normalization   Batch normalization
     4   ''      ReLU                  ReLU
     5   ''      Fully Connected       1 fully connected layer
     6   'mae'   Regression Output     Mean absolute error
```

Set the training options and train the network.

```
options = trainingOptions('sgdm');
net = trainNetwork(XTrain,YTrain,layers,options);
```

```
Training on single CPU.
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |     RMSE     |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:00 |        28.28 |         25.1 |          0.0100 |
|       2 |          50 |       00:00:04 |        14.27 |         11.3 |          0.0100 |
|       3 |         100 |       00:00:09 |        13.45 |         10.5 |          0.0100 |
|       4 |         150 |       00:00:14 |        10.35 |          8.2 |          0.0100 |
|       6 |         200 |       00:00:19 |        10.28 |          8.0 |          0.0100 |
|       7 |         250 |       00:00:24 |        10.49 |          7.9 |          0.0100 |
|       8 |         300 |       00:00:28 |         9.21 |          7.3 |          0.0100 |
|       9 |         350 |       00:00:34 |         9.18 |          7.0 |          0.0100 |
|      11 |         400 |       00:00:38 |        10.49 |          8.2 |          0.0100 |
```

```
|      12 |         450 |       00:00:42 |         8.12 |          6.1 |          0.0100 |
|      13 |         500 |       00:00:47 |         9.13 |          6.0 |          0.0100 |
|      15 |         550 |       00:00:52 |         9.85 |          7.4 |          0.0100 |
|      16 |         600 |       00:00:57 |         8.70 |          6.4 |          0.0100 |
|      17 |         650 |       00:01:01 |         8.21 |          6.1 |          0.0100 |
|      18 |         700 |       00:01:06 |         9.05 |          6.3 |          0.0100 |
|      20 |         750 |       00:01:11 |         8.05 |          6.0 |          0.0100 |
|      21 |         800 |       00:01:15 |         7.80 |          5.7 |          0.0100 |
|      22 |         850 |       00:01:19 |         6.86 |          5.4 |          0.0100 |
|      24 |         900 |       00:01:23 |         7.37 |          5.5 |          0.0100 |
|      25 |         950 |       00:01:28 |         7.00 |          5.0 |          0.0100 |
|      26 |        1000 |       00:01:32 |         6.99 |          5.0 |          0.0100 |
|      27 |        1050 |       00:01:36 |         8.29 |          6.6 |          0.0100 |
|      29 |        1100 |       00:01:40 |         8.34 |          6.8 |          0.0100 |
|      30 |        1150 |       00:01:44 |         6.26 |          4.5 |          0.0100 |
|      30 |        1170 |       00:01:46 |         6.90 |          5.1 |          0.0100 |
|=========================================================================================|
Training finished: Max epochs completed.
```

Evaluate the network performance by calculating the prediction error between the predicted and actual angles of rotation.

```
[XTest,~,YTest] = digitTest4DArrayData;
YPred = predict(net,XTest);
predictionError = YTest - YPred;
```

Calculate the number of predictions within an acceptable error margin from the true angles. Set the threshold to be 10 degrees and calculate the percentage of predictions within this threshold.

```
thr = 10;
numCorrect = sum(abs(predictionError) < thr);
numTestImages = size(XTest,4);
accuracy = numCorrect/numTestImages
```

```
accuracy = 0.7586
```

See Also

`regressionLayer` | `checkLayer` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Classification Output Layer” on page 18-91
- “Specify Custom Output Layer Backward Loss Function” on page 18-113
- “Check Custom Layer Validity” on page 18-154

Specify Custom Layer Backward Function

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

The example “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35 shows how to create a custom PreLU layer and goes through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

If the forward function only uses functions that support `dlarray` objects, then creating a backward function is optional. In this case, the software determines the derivatives automatically using automatic differentiation. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. If you want to use functions that do not support `dlarray` objects, or want to use a specific algorithm for the backward function, then you can define a custom backward function using this example as a guide.

Create Custom Layer

The example “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35 shows how to create a PReLU layer. A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar learned at training time.[1] For values less than zero, a PReLU layer applies scaling coefficients α_i to each channel of the input. These coefficients form a learnable parameter, which the layer learns during training.

The PReLU operation is given by

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ \alpha_i x_i & \text{if } x_i \le 0 \end{cases}$$

where x_i is the input of the nonlinear activation f on channel i, and α_i is the coefficient controlling the slope of the negative part. The subscript i in α_i indicates that the nonlinear activation can vary on different channels.

View the layer created in the example “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35. This layer does not have a `backward` function.

```
classdef preluLayer < nnet.layer.Layer
    % Example custom PReLU layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficient
        Alpha
    end

    methods
        function layer = preluLayer(numChannels,args)
            % layer = preluLayer(numChannels) creates a PReLU layer
            % with numChannels channels.
            %
            % layer = preluLayer(numChannels,Name=name) also specifies the
            % layer name.

            arguments
                numChannels
                args.Name = "";
            end

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = "PReLU with " + numChannels + " channels";

            % Initialize scaling coefficient.
            layer.Alpha = rand([1 1 numChannels]);
        end

        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            Z = max(X,0) + layer.Alpha .* min(0,X);
        end
    end
end
```

Note If the layer has a custom backward function, then you can still inherit from `nnet.layer.Formattable`.

Create Backward Function

Implement the `backward` function that returns the derivatives of the loss with respect to the input data and the learnable parameters.

The `backward` function syntax depends on the type of layer.

- `dLdX = backward(layer,X,Z,dLdZ,memory)` returns the derivatives `dLdX` of the loss with respect to the layer input, where `layer` has a single input and a single output. `Z` corresponds to

the forward function output and `dLdZ` corresponds to the derivative of the loss with respect to `Z`. The function input `memory` corresponds to the memory output of the forward function.

- `[dLdX,dLdW] = backward(layer,X,Z,dLdZ,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter, where `layer` has a single learnable parameter.
- `[dLdX,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdSin` of the loss with respect to the state input using any of the previous syntaxes, where `layer` has a single state parameter and `dLdSout` corresponds to the derivative of the loss with respect to the layer state output.
- `[dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter and returns the derivative `dLdSin` of the loss with respect to the layer state input using any of the previous syntaxes, where `layer` has a single state parameter and single learnable parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, multiple learnable parameters, or multiple state parameters:

- For layers with multiple inputs, replace `X` and `dLdX` with `X1,...,XN` and `dLdX1,...,dLdXN`, respectively, where `N` is the number of inputs.
- For layers with multiple outputs, replace `Z` and `dLdZ` with `Z1,...,ZM` and `dLdZ1,...,dLdZM`, respectively, where `M` is the number of outputs.
- For layers with multiple learnable parameters, replace `dLdW` with `dLdW1,...,dLdWP`, where `P` is the number of learnable parameters.
- For layers with multiple state parameters, replace `dLdSin` and `dLdSout` with `dLdSin1,...,dLdSinK` and `dLdSout1,...,dLdSoutK`, respectively, where `K` is the number of state parameters.

To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where the first `N` elements correspond to the `N` layer inputs, the next `M` elements correspond to the `M` layer outputs, the next `M` elements correspond to the derivatives of the loss with respect to the `M` layer outputs, the next `K` elements correspond to the `K` derivatives of the loss with respect to the `K` states outputs, and the last element corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where the first `N` elements correspond to the `N` the derivatives of the loss with respect to the `N` layer inputs, the next `P` elements correspond to the derivatives of the loss with respect to the `P` learnable parameters, and the next `K` elements correspond to the derivatives of the loss with respect to the `K` state inputs.

Note `dlnetwork` objects do not support custom layers that require a memory value in a custom backward function. To use a custom layer with a custom backward function in a `dlnetwork` object, the `memory` input of the `backward` function definition must be `~`.

Because a PReLU layer has only one input, one output, one learnable parameter, and does not require the outputs of the layer forward function or a memory value, the syntax for `backward` for a PReLU layer is `[dLdX,dLdAlpha] = backward(layer,X,~,dLdZ,~)`. The dimensions of `X` are the same

as in the forward function. The dimensions of `dLdZ` are the same as the dimensions of the output `Z` of the forward function. The dimensions and data type of `dLdX` are the same as the dimensions and data type of `X`. The dimension and data type of `dLdAlpha` is the same as the dimension and data type of the learnable parameter `Alpha`.

During the backward pass, the layer automatically updates the learnable parameters using the corresponding derivatives.

To include a custom layer in a network, the layer forward functions must accept the outputs of the previous layer and forward propagate arrays with the size expected by the next layer. Similarly, when `backward` is specified, the `backward` function must accept inputs with the same size as the corresponding output of the forward function and backward propagate derivatives with the same size.

The derivative of the loss with respect to the input data is

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial f(x_i)} \frac{\partial f(x_i)}{\partial x_i}$$

where $\partial L / \partial f(x_i)$ is the gradient propagated from the next layer, and the derivative of the activation is

$$\frac{\partial f(x_i)}{\partial x_i} = \begin{cases} 1 & \text{if } x_i \geq 0 \\ \alpha_i & \text{if } x_i < 0 \end{cases}.$$

The derivative of the loss with respect to the learnable parameters is

$$\frac{\partial L}{\partial \alpha_i} = \sum_j \frac{\partial L}{\partial f(x_{ij})} \frac{\partial f(x_{ij})}{\partial \alpha_i}$$

where i indexes the channels, j indexes the elements over height, width, and observations, and the gradient of the activation is

$$\frac{\partial f(x_i)}{\partial \alpha_i} = \begin{cases} 0 & \text{if } x_i \geq 0 \\ x_i & \text{if } x_i < 0 \end{cases}.$$

Create the backward function that returns these derivatives.

```
function [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
    % [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
    % backward propagates the derivative of the loss function
    % through the layer.
    % Inputs:
    %         layer    - Layer to backward propagate through
    %         X        - Input data
    %         dLdZ     - Gradient propagated from the deeper layer
    % Outputs:
    %         dLdX     - Derivative of the loss with respect to the
    %                    input data
    %         dLdAlpha - Derivative of the loss with respect to the
    %                    learnable parameter Alpha

    dLdX = layer.Alpha .* dLdZ;
    dLdX(X>0) = dLdZ(X>0);
    dLdAlpha = min(0,X) .* dLdZ;
    dLdAlpha = sum(dLdAlpha,[1 2]);

    % Sum over all observations in mini-batch.
    dLdAlpha = sum(dLdAlpha,4);
end
```

Complete Layer

View the completed layer class file.

```
classdef preluLayer < nnet.layer.Layer
    % Example custom PReLU layer.

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficient
        Alpha
    end

    methods
        function layer = preluLayer(numChannels,args)
            % layer = preluLayer(numChannels) creates a PReLU layer
            % with numChannels channels.
            %
            % layer = preluLayer(numChannels,Name=name) also specifies the
            % layer name.

            arguments
                numChannels
                args.Name = "";
            end

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = "PReLU with " + numChannels + " channels";

            % Initialize scaling coefficient.
            layer.Alpha = rand([1 1 numChannels]);
        end

        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            Z = max(X,0) + layer.Alpha .* min(0,X);
        end

        function [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
            % [dLdX, dLdAlpha] = backward(layer, X, ~, dLdZ, ~)
            % backward propagates the derivative of the loss function
            % through the layer.
            % Inputs:
            %         layer    - Layer to backward propagate through
            %         X        - Input data
            %         dLdZ     - Gradient propagated from the deeper layer
            % Outputs:
            %         dLdX     - Derivative of the loss with respect to the
            %                    input data
            %         dLdAlpha - Derivative of the loss with respect to the
            %                    learnable parameter Alpha
```

```
            dLdX = layer.Alpha .* dLdZ;
            dLdX(X>0) = dLdZ(X>0);
            dLdAlpha = min(0,X) .* dLdZ;
            dLdAlpha = sum(dLdAlpha,[1 2]);

            % Sum over all observations in mini-batch.
            dLdAlpha = sum(dLdAlpha,4);
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

References

[1] "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *In 2015 IEEE International Conference on Computer Vision (ICCV)*, 1026–34. Santiago, Chile: IEEE, 2015. https://doi.org/10.1109/ICCV.2015.123.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Specify Custom Output Layer Backward Loss Function

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

The example “Define Custom Classification Output Layer” on page 18-91 shows how to define and create a custom classification output layer with sum of squares error (SSE) loss and goes through the following steps:

1. Name the layer – Give the layer a name so it can be used in MATLAB.
2. Declare the layer properties – Specify the properties of the layer.
3. Create a constructor function (optional) – Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then the software initializes the properties with `''` at creation.
4. Create a forward loss function – Specify the loss between the predictions and the training targets.
5. Create a backward loss function (optional) – Specify the derivative of the loss with respect to the predictions. If you do not specify a backward loss function, then the forward loss function must support `dlarray` objects.

Creating a backward loss function is optional. If the forward loss function only uses functions that support `dlarray` objects, then software determines the derivatives automatically using automatic differentiation. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. If you want to use functions that do not support `dlarray` objects, or want to use a specific algorithm for the backward loss function, then you can define a custom backward function using this example as a guide.

Create Custom Layer

The example “Define Custom Classification Output Layer” on page 18-91 shows how to create a SSE classification layer.

A classification SSE layer computes the sum of squares error loss for classification problems. SSE is an error measure between two continuous random variables. For predictions Y and training targets T, the SSE loss between Y and T is given by

$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K}(Y_{ni} - T_{ni})^2,$$

where N is the number of observations and K is the number of classes.

View the layer created in the example “Define Custom Classification Output Layer” on page 18-91. This layer does not have a `backwardLoss` function.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    % Example custom classification layer with sum of squares error loss.

    methods
        function layer = sseClassificationLayer(name)
```

```
            % layer = sseClassificationLayer(name) creates a sum of squares
            % error classification layer and specifies the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = 'Sum of squares error';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the SSE loss between
            % the predictions Y and the training targets T.

            % Calculate sum of squares.
            sumSquares = sum((Y-T).^2);

            % Take mean over mini-batch.
            N = size(Y,4);
            loss = sum(sumSquares)/N;
        end
    end
end
```

Create Backward Loss Function

Implement the `backwardLoss` function that returns the derivatives of the loss with respect to the input data and the learnable parameters.

The syntax for `backwardLoss` is `dLdY = backwardLoss(layer,Y,T)`. The input `Y` contains the predictions made by the network and `T` contains the training targets. The output `dLdY` is the derivative of the loss with respect to the predictions `Y`. The output `dLdY` must be the same size as the layer input `Y`.

The dimensions of `Y` and `T` are the same as the inputs in `forwardLoss`.

The derivative of the SSE loss with respect to the predictions Y is given by

$$\frac{\delta L}{\delta Y_i} = \frac{2}{N}(Y_i - T_i),$$

where N is the number of observations in the input.

Create the backward loss function that returns these derivatives.

```
        function dLdY = backwardLoss(layer, Y, T)
            % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
            % the SSE loss with respect to the predictions Y.

            N = size(Y,4);
            dLdY = 2*(Y-T)/N;
        end
```

Complete Layer

View the completed layer class file.

```
classdef sseClassificationLayer < nnet.layer.ClassificationLayer
    % Example custom classification layer with sum of squares error loss.
```

```
    methods
        function layer = sseClassificationLayer(name)
            % layer = sseClassificationLayer(name) creates a sum of squares
            % error classification layer and specifies the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
            layer.Description = 'Sum of squares error';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the SSE loss between
            % the predictions Y and the training targets T.

            % Calculate sum of squares.
            sumSquares = sum((Y-T).^2);

            % Take mean over mini-batch.
            N = size(Y,4);
            loss = sum(sumSquares)/N;
        end

        function dLdY = backwardLoss(layer, Y, T)
            % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
            % the SSE loss with respect to the predictions Y.

            N = size(Y,4);
            dLdY = 2*(Y-T)/N;
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

See Also

`checkLayer` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Classification Output Layer” on page 18-91
- “Define Custom Regression Output Layer” on page 18-99
- “Check Custom Layer Validity” on page 18-154

Deep Learning Network Composition

To create a custom layer that itself defines a layer graph, you can declare a `dlnetwork` object as a learnable parameter in the `properties (Learnable)` section of the layer definition. This method is known as *network composition*. You can use network composition to:

- Create a single custom layer that represents a block of learnable layers, for example, a residual block.
- Create a network with control flow, for example, a network with a section that can dynamically change depending on the input data.
- Create a network with loops, for example, a network with sections that feed the output back into itself.

For nested networks that have both learnable and state parameters, for example, networks with batch normalization or LSTM layers, declare the network in the `properties (Learnable, State)` section of the layer definition.

For an example showing how to define a custom layer containing a learnable `dlnetwork` object, see “Define Nested Deep Learning Layer” on page 18-120.

For an example showing how to train a network with nested layers, see “Train Deep Learning Network with Nested Layers” on page 18-135.

Automatically Initialize Learnable dlnetwork Objects for Training

You can create a custom layer and allow the software to automatically initialize the learnable parameters of any nested `dlnetwork` objects after the parent network is fully constructed. Automatic initialization of the nested network means that you do not need to keep track of the size and shape of the inputs passed to each custom layer containing a nested `dlnetwork`

To take advantage of automatic initialization, you must specify that the constructor function creates an uninitialized `dlnetwork` object. To create an uninitialized `dlnetwork` object, set the `Initialize` name-value option to false. You do not need to specify an input layer, so you do not need to specify an input size for the layer.

```
function layer = myLayer

    % Initialize layer properties.
    ...

    % Define network.
    layers = [
        % Network layers go here.
        ];

    layer.Network = dlnetwork(lgraph,'Initialize',false);
end
```

When the parent network is initialized, the learnable parameters of any nested `dlnetwork` objects are initialized at the same time. The size of the learnable parameters depends on the size of the input data of the custom layer. The software propagates the data through the nested network and automatically initializes the parameters according to the propagated sizes and the initialization properties of the layers of the nested network.

If the parent network is trained using the `trainNetwork` function, then any nested `dlnetwork` objects are initialized when you call `trainNetwork`. If the parent network is a `dlnetwork`, then any nested `dlnetwork` objects are initialized when the parent network is constructed (if the parent `dlnetwork` is initialized at construction) or when you use the `initialize` function with the parent network (if the parent `dlnetwork` is not initialized at construction).

If you do not want to make use of automatic initialization, you can construct the custom layer with the nested network already initialized. In this case, the nested network is initialized before the parent network. To initialize the nested network at construction, you must manually specify the size of any inputs to the nested network. This requires manually specifying the size of any inputs to the nested network. You can do so either by using input layers or by providing example inputs to the `dlnetwork` constructor function. Because you must specify the sizes of any inputs to the `dlnetwork` object, you might need to specify input sizes when you create the layer. For help determining the size of the inputs to the layer, you can use the `analyzeNetwork` function and check the size of the activations of the previous layers.

Predict and Forward Functions

Some layers behave differently during training and during prediction. For example, a dropout layer performs dropout only during training and has no effect during prediction. A layer uses one of two functions to perform a forward pass: `predict` or `forward`. If the forward pass is at prediction time, then the layer uses the `predict` function. If the forward pass is at training time, then the layer uses the `forward` function. If you do not require two different functions for prediction time and training time, then you can omit the `forward` function. In this case, the layer uses `predict` at training time.

When implementing the `predict` and the `forward` functions of the custom layer, to ensure that the layers in the `dlnetwork` object behave in the correct way, use the `predict` and `forward` functions for `dlnetwork` objects, respectively.

Custom layers with learnable `dlnetwork` objects do not support custom backward functions.

You must still assign a value to the memory output argument of the `forward` function.

This example code shows how to use the `predict` and `forward` functions with `dlnetwork` input.

```
function Z = predict(layer,X)

    % Convert input data to formatted dlarray.
    X = dlarray(X,"SSCB");

    % Predict using network.
    dlnet = layer.Network;
    Z = predict(dlnet,X);

    % Strip dimension labels.
    Z = stripdims(Z);
end

function Z = forward(layer,X)

    % Convert input data to formatted dlarray.
    X = dlarray(X,"SSCB");

    % Forward pass using network.
    dlnet = layer.Network;
```

```
    Z = forward(dlnet,X);

    % Strip dimension labels.
    Z = stripdims(Z);
end
```

If the `dlnetwork` object does not behave differently during training and prediction, then you can omit the forward function. In this case, the software uses the `predict` function during training.

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

See Also

`checkLayer` | `trainNetwork` | `trainingOptions` | `analyzeNetwork` | `dlnetwork`

More About

- “Train Deep Learning Network with Nested Layers” on page 18-135
- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Define Nested Deep Learning Layer

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

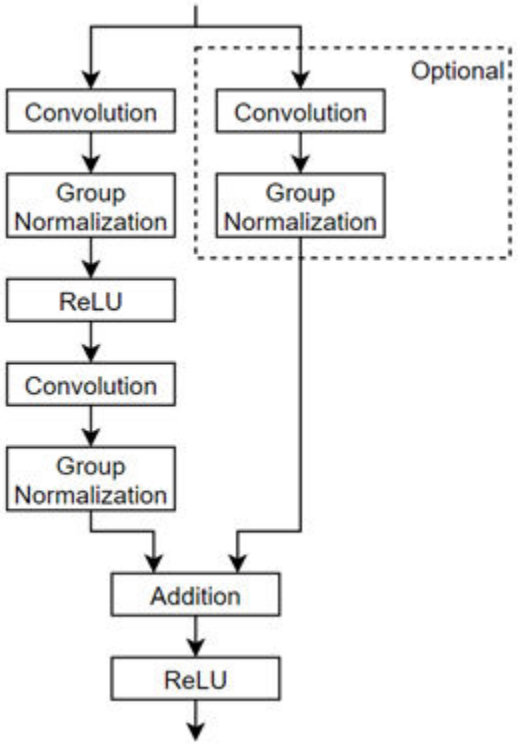
To create a custom layer that itself defines a layer graph, you can declare a `dlnetwork` object as a learnable parameter in the `properties (Learnable)` section of the layer definition. This method is known as *network composition*. You can use network composition to:

- Create a single custom layer that represents a block of learnable layers, for example, a residual block.
- Create a network with control flow, for example, a network with a section that can dynamically change depending on the input data.
- Create a network with loops, for example, a network with sections that feed the output back into itself.

For nested networks that have both learnable and state parameters, for example, networks with batch normalization or LSTM layers, declare the network in the `properties (Learnable, State)` section of the layer definition.

For more information, see “Deep Learning Network Composition” on page 18-117.

This example shows how to create a custom layer representing a residual block. The custom layer `residualBlockLayer` contains a learnable block of layers consisting of convolution, group normalization, ReLU, and addition layers, and also includes a skip connection and an optional convolution layer and group normalization layer in the skip connection. The layer has a single input that is used twice, as the input to each branch. This diagram highlights the residual block structure.



To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
```

```
function layer = myLayer()
    % (Optional) Create a myLayer.
    % This function must have the same name as the class.

    % Define layer constructor function here.
end

function [Z,state] = predict(layer,X)
    % Forward input data through the layer at prediction time and
    % output the result and updated state.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Input data
    % Outputs:
    %         Z     - Output of layer forward function
    %         state - (Optional) Updated layer state.
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer predict function here.
end

function [Z,state,memory] = forward(layer,X)
    % (Optional) Forward input data through the layer at training
    % time and output the result, updated state, and a memory
    % value.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Layer input data
    % Outputs:
    %         Z      - Output of layer forward function
    %         state  - (Optional) Updated layer state
    %         memory - (Optional) Memory value for custom backward
    %                  function
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer forward function here.
end

function layer = resetState(layer)
    % (Optional) Reset layer state.

    % Define reset state function here.
end

function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
    % (Optional) Backward propagate the derivative of the loss
    % function through the layer.
    %
    % Inputs:
    %         layer   - Layer to backward propagate through
    %         X       - Layer input data
    %         Z       - Layer output data
    %         dLdZ    - Derivative of loss with respect to layer
    %                   output
    %         dLdSout - (Optional) Derivative of loss with respect
    %                   to state output
```

```
            %         memory  - Memory value from forward function
            % Outputs:
            %         dLdX   - Derivative of loss with respect to layer input
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Name Layer

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `residualBlockLayer`.

```
classdef residualBlockLayer < nnet.layer.Layer
    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = residualBlockLayer()
            ...
        end

        ...
     end
```

Save Layer

Save the layer class file in a new file named `residualBlockLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs,

then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

The residual block layer does not require any additional properties, so you can remove the `properties` section.

This custom layer has only one learnable parameter, the residual block itself specified as a `dlnetwork` object. Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Network`.

```
properties (Learnable)
    % Layer learnable parameters

    % Residual block.
    Network
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The residual block layer constructor function requires four input arguments:

- Number of convolutional filters
- Stride (optional, with default stride 1)
- Flag to include convolution in skip connection (optional, with default flag `false`)
- Layer name (optional, with default name `''`)

In the constructor function `residualBlockLayer`, specify the required input argument `numFilters` and the optional arguments as name-value pairs with the name `NameValueArgs`. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = residualBlockLayer(numFilters,NameValueArgs)
    % layer = residualBlockLayer(numFilters) creates a residual
    % block layer with the specified number of filters.
    %
    % layer = residualBlockLayer(numFilters,Name,Value) specifies
    % additional options using one or more name-value pair
    % arguments:
    %
    %     'Stride'                - Stride of convolution operation
    %                               (default 1)
    %
    %     'IncludeSkipConvolution' - Flag to include convolution in
    %                               skip connection
    %                               (default false)
    %
    %     'Name'                  - Layer name
    %                               (default '')

    ...
end
```

Parse Input Arguments

Parse the input arguments using an `arguments` block. List the arguments in the same order as the function syntax and specify the default values. Then, extract the values from the `NameValueArgs` input.

```
% Parse input arguments.
arguments
    numFilters
    NameValueArgs.Stride = 1
    NameValueArgs.IncludeSkipConvolution = false
    NameValueArgs.Name = ''
end

stride = NameValueArgs.Stride;
includeSkipConvolution = NameValueArgs.IncludeSkipConvolution;
name = NameValueArgs.Name;
```

Initialize Layer Properties

In the constructor function, initialize the layer properties, including the `dlnetwork` object. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the `Name` property to the input argument `name`.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the layer and any optional properties.

```
% Set layer description.
description = "Residual block with " + numFilters + " filters, stride " + stride;
if includeSkipConvolution
    description = description + ", and skip convolution";
end
layer.Description = description;
```

Specify the type of the layer by setting the `Type` property. The value of `Type` appears when the layer is displayed in a `Layer` array.

```
% Set layer type.
layer.Type = "Residual Block";
```

Define the residual block. You can create the residual block layers as an uninitialized nested `dlnetwork` object without an input layer and allow the software to automatically initialize the learnable and state parameters at training time. For more information, see “Automatically Initialize Learnable dlnetwork Objects for Training” on page 18-117.

First, create a layer array containing the main layers of the block and convert it to a layer graph.

```
% Define nested layer graph.
layers = [
    convolution2dLayer(3,numFilters,'Padding','same','Stride',stride,'Name','conv1')
    groupNormalizationLayer('all-channels','Name','gn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,numFilters,'Padding','same','Name','conv2')
    groupNormalizationLayer('channel-wise','Name','gn2')

    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')];

lgraph = layerGraph(layers);
```

Next, add the skip connection. If the `includeSkipConvolution` flag is `true`, then also include a convolution layer and group normalization layer in the skip connection.

```
% Add skip connection.
if includeSkipConvolution
    layers = [
        convolution2dLayer(1,numFilters,'Stride',stride,'Name','convSkip')
        groupNormalizationLayer('all-channels','Name','gnSkip')];

    lgraph = addLayers(lgraph,layers);
```

```
        lgraph = connectLayers(lgraph,'gnSkip','add/in2');
    end
```

Since there is no input layer, this network has two unconnected inputs. If the network does not have the skip connection, the input to the `'conv2'` layer and one of the inputs to the `'add'` layer are unconnected. If the network does have the skip connection, then the unconnected inputs are the inputs to the `'conv1'` and `'convSkip'` layers.

Finally, convert the layer graph to a `dlnetwork` object and set the layer `Network` property. Create an uninitialized `dlnetwork` object. The weights and learnable parameters in the `dlnetwork` object are automatically initialized when the complete network is assembled for training.

```
% Convert to dlnetwork.
dlnet = dlnetwork(lgraph,'Initialize',false);

% Set Network property.
layer.Network = dlnet;
```

View the completed constructor function.

```
function layer = residualBlockLayer(numFilters,NameValueArgs)
    % layer = residualBlockLayer(numFilters) creates a residual
    % block layer with the specified number of filters.
    %
    % layer = residualBlockLayer(numFilters,Name,Value) specifies
    % additional options using one or more name-value pair
    % arguments:
    %
    %     'Stride'                - Stride of convolution operation
    %                               (default 1)
    %
    %     'IncludeSkipConvolution' - Flag to include convolution in
    %                               skip connection
    %                               (default false)
    %
    %     'Name'                  - Layer name
    %                               (default '')

    % Parse input arguments.
    arguments
        numFilters
        NameValueArgs.Stride = 1
        NameValueArgs.IncludeSkipConvolution = false
        NameValueArgs.Name = ''
    end

    stride = NameValueArgs.Stride;
    includeSkipConvolution = NameValueArgs.IncludeSkipConvolution;
    name = NameValueArgs.Name;

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    description = "Residual block with " + numFilters + " filters, stride " + stride;
    if includeSkipConvolution
        description = description + ", and skip convolution";
    end
    layer.Description = description;

    % Set layer type.
    layer.Type = "Residual Block";

    % Define nested layer graph.
    layers = [
        convolution2dLayer(3,numFilters,'Padding','same','Stride',stride,'Name','conv1')
        groupNormalizationLayer('all-channels','Name','gn1')
        reluLayer('Name','relu1')
        convolution2dLayer(3,numFilters,'Padding','same','Name','conv2')
        groupNormalizationLayer('channel-wise','Name','gn2')

        additionLayer(2,'Name','add')
        reluLayer('Name','relu2')];
```

```
        lgraph = layerGraph(layers);

        % Add skip connection.
        if includeSkipConvolution
            layers = [
                convolution2dLayer(1,numFilters,'Stride',stride,'Name','convSkip')
                groupNormalizationLayer('all-channels','Name','gnSkip')];

            lgraph = addLayers(lgraph,layers);
            lgraph = connectLayers(lgraph,'gnSkip','add/in2');
        end

        % Convert to dlnetwork.
        dlnet = dlnetwork(lgraph,'Initialize',false);

        % Set Network property.
        layer.Network = dlnet;
    end
```

With this constructor function, the command `residualBlockLayer(64,'Stride',2,'IncludeSkipConvolution',true,'Name','res5')` creates a residual block layer with 64 filters, a stride of 2, a convolution in the skip connection, and with the name `'res5'`. The required sizes of weights and parameters are determined when the completed network is assembled for training.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Tip If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.

Because the residual block has only one input and one output, the syntax for predict for the custom layer is `Z = predict(layer,X)`.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for a custom backward function, you must also create a function named `forward`.

The dimensions of the inputs depend on the type of data and the output of the connected layers.

Layer Input	Input Size	Observation Dimension
Feature vectors	*c*-by-*N*, where *c* corresponds to the number of channels and *N* is the number of observations.	2
2-D images	*h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations.	4
3-D images	*h*-by-*w*-by-*d*-by-*c*-by-*N*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, and *N* is the number of observations.	5
Vector sequences	*c*-by-*N*-by-*S*, where *c* is the number of features of the sequences, *N* is the number of observations, and *S* is the sequence length.	2
2-D image sequences	*h*-by-*w*-by-*c*-by-*N*-by-*S*, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, *N* is the number of observations, and *S* is the sequence length.	4
3-D image sequences	*h*-by-*w*-by-*d*-by-*c*-by-*N*-by-*S*, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the 3-D images, respectively, *N* is the number of observations, and *S* is the sequence length.	5

For layers that output sequences, the layers can output sequences of any length or output data with no time dimension. Note that when training a network that outputs sequences using the `trainNetwork` function, the lengths of the input and output sequences must match.

For the residual block layer, a forward pass of the layer is simply a forward pass of the `dlnetwork` object. To pass the input data to the `dlnetwork` object, you must first convert it to a formatted `dlarray` object.

Implement this operation in the custom layer function `predict`. To perform a forward pass of the `dlnetwork` for prediction, use the `predict` function for `dlnetwork` objects. In this case, the input to the residual block layer is used as the input to both of the unconnected inputs to the `dlnetwork` object, so the syntax for `predict` for the `dlnetwork` object is `Z = predict(dlnet,X,X)`.

Because the layers in the `dlnetwork` object do not behave differently during training and that the residual block layer does not require memory or a different forward function for training, you can remove the `forward` function from the class file.

Create the `predict` function and add a comment to the top of the function that explains the syntaxes of the function.

```
function Z = predict(layer, X)
    % Forward input data through the layer at prediction time and
    % output the result.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Input data
    % Outputs:
    %         Z - Output of layer forward function

    % Convert input data to formatted dlarray.
    X = dlarray(X,'SSCB');

    % Predict using network.
    dlnet = layer.Network;
    Z = predict(dlnet,X,X);

    % Strip dimension labels.
    Z = stripdims(Z);
end
```

Because the `predict` function only uses functions that support `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed layer class file.

```
classdef residualBlockLayer < nnet.layer.Layer
    % Example custom residual block layer.


    properties (Learnable)
        % Layer learnable parameters

        % Residual block.
        Network
    end

    methods
        function layer = residualBlockLayer(numFilters,NameValueArgs)
            % layer = residualBlockLayer(numFilters) creates a residual
            % block layer with the specified number of filters.
```

```
%
% layer = residualBlockLayer(numFilters,Name,Value) specifies
% additional options using one or more name-value pair
% arguments:
%
%     'Stride'                - Stride of convolution operation
%                               (default 1)
%
%     'IncludeSkipConvolution' - Flag to include convolution in
%                               skip connection
%                               (default false)
%
%     'Name'                  - Layer name
%                               (default '')

% Parse input arguments.
arguments
    numFilters
    NameValueArgs.Stride = 1
    NameValueArgs.IncludeSkipConvolution = false
    NameValueArgs.Name = ''
end

stride = NameValueArgs.Stride;
includeSkipConvolution = NameValueArgs.IncludeSkipConvolution;
name = NameValueArgs.Name;

% Set layer name.
layer.Name = name;

% Set layer description.
description = "Residual block with " + numFilters + " filters, stride " + stride;
if includeSkipConvolution
    description = description + ", and skip convolution";
end
layer.Description = description;

% Set layer type.
layer.Type = "Residual Block";

% Define nested layer graph.
layers = [
    convolution2dLayer(3,numFilters,'Padding','same','Stride',stride,'Name','conv1')
    groupNormalizationLayer('all-channels','Name','gn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,numFilters,'Padding','same','Name','conv2')
    groupNormalizationLayer('channel-wise','Name','gn2')

    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')];

lgraph = layerGraph(layers);

% Add skip connection.
if includeSkipConvolution
    layers = [
        convolution2dLayer(1,numFilters,'Stride',stride,'Name','convSkip')
        groupNormalizationLayer('all-channels','Name','gnSkip')];
```

```
                lgraph = addLayers(lgraph,layers);
                lgraph = connectLayers(lgraph,'gnSkip','add/in2');
            end

            % Convert to dlnetwork.
            dlnet = dlnetwork(lgraph,'Initialize',false);

            % Set Network property.
            layer.Network = dlnet;
        end

        function Z = predict(layer, X)
            % Forward input data through the layer at prediction time and
            % output the result.
            %
            % Inputs:
            %         layer - Layer to forward propagate through
            %         X     - Input data
            % Outputs:
            %         Z - Output of layer forward function

            % Convert input data to formatted dlarray.
            X = dlarray(X,'SSCB');

            % Predict using network.
            dlnet = layer.Network;
            Z = predict(dlnet,X,X);

            % Strip dimension labels.
            Z = stripdims(Z);
        end
    end
end
```

GPU Compatibility

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

In this example, the MATLAB functions used in `predict` all support `dlarray` objects, so the layer is GPU compatible.

Check Validity of Layer Using checkLayer

Check the layer validity of the custom layer `residualBlockLayer` using the `checkLayer` function.

Create an instance of a residual block layer. To access this layer, open this example as a live script.

```
numFilters = 64;

layer = residualBlockLayer(numFilters)
```

```
layer =
  residualBlockLayer with properties:

       Name: ''

   Learnable Parameters
    Network: [1x1 dlnetwork]

   State Parameters
    No properties.

  Show all properties
```

Check the layer validity using the `checkLayer` function. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations. Specify a typical input size and set the `'ObservationDimension'` option to 4.

```
validInputSize = [56 56 64];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 7.1651 seconds.
```

The function does not detect any issues with the layer.

See Also

`setLearnRateFactor` | `checkLayer` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `assembleNetwork`

More About

- “Deep Learning Network Composition” on page 18-117
- “Define Custom Deep Learning Layers” on page 18-9
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48

- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Custom Classification Output Layer” on page 18-91
- “Define Custom Regression Output Layer” on page 18-99
- “Check Custom Layer Validity” on page 18-154

Train Deep Learning Network with Nested Layers

This example shows how to train a network with nested layers.

To create a custom layer that itself defines a layer graph, you can specify a `dlnetwork` object as a learnable parameter. This method is known as *network composition*. You can use network composition to:

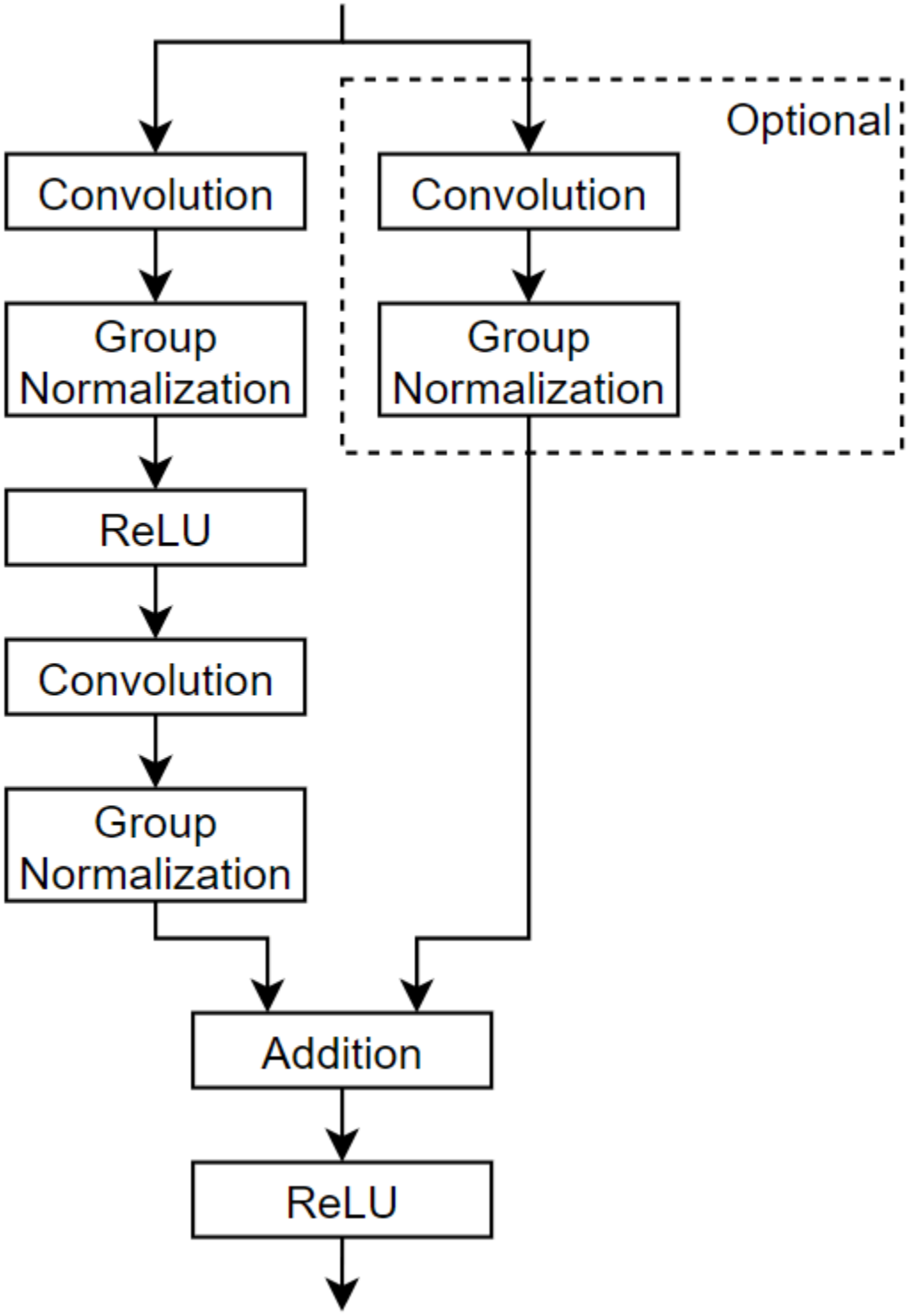
- Create a single custom layer that represents a block of learnable layers, for example, a residual block.
- Create a network with control flow. For example, a network with a section that can dynamically change depending on the input data.
- Create a network with loops. For example, a network with sections that feed the output back into itself.

For more information, see “Deep Learning Network Composition” on page 18-117.

This example shows how to train a network using custom layers representing residual blocks, each containing multiple convolution, group normalization, and ReLU layers with a skip connection. For an example showing how to create a residual network *without* using custom layers, see “Train Residual Network for Image Classification” on page 3-13.

Residual connections are a popular element in convolutional neural network architectures. A residual network is a type of network that has residual (or shortcut) connections that bypass the main network layers. Using residual connections improves gradient flow through the network and enables the training of deeper networks. This increased network depth can yield higher accuracies on more difficult tasks.

This example uses the custom layer `residualBlockLayer`, which contains a learnable block of layers consisting of convolution, group normalization, ReLU, and addition layers, and also includes a skip connection and an optional convolution layer and group normalization layer in the skip connection. This diagram highlights the residual block structure.



For an example showing how to create the custom layer `residualBlockLayer`, see “Define Nested Deep Learning Layer” on page 18-120.

Prepare Data

Download and extract the Flowers data set [1].

```
url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';
downloadFolder = tempdir;
filename = fullfile(downloadFolder,'flower_dataset.tgz');

imageFolder = fullfile(downloadFolder,'flower_photos');
if ~exist(imageFolder,'dir')
    disp('Downloading Flowers data set (218 MB)...')
    websave(filename,url);
    untar(filename,downloadFolder)
end
```

Create an image datastore containing the photos.

```
datasetFolder = fullfile(imageFolder);
imds = imageDatastore(datasetFolder, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Partition the data into training and validation data sets. Use 70% of the images for training and 30% for validation.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');
```

View the number of classes of the data set.

```
classes = categories(imds.Labels);
numClasses = numel(classes)
```

```
numClasses = 5
```

Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images. Resize and augment the images for training using an `imageDataAugmenter` object:

- Randomly reflect the images in the vertical axis.
- Randomly translate the images up to 30 pixels vertically and horizontally.
- Randomly rotate the images up to 45 degrees clockwise and counterclockwise.
- Randomly scale the images up to 10% vertically and horizontally.

```
pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange, ...
    'RandRotation',[-45 45], ...
    'RandXScale',scaleRange, ...
    'RandYScale',scaleRange);
```

Create an augmented image datastore containing the training data using the image data augmenter. To automatically resize the images to the network input size, specify the height and width of the input size of the network. This example uses a network with input size `[224 224 3]`.

```
inputSize = [224 224 3];
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain,'DataAugmentation',imageAugmente
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore([224 224],imdsValidation);
```

Define Network Architecture

Define a residual network with six residual blocks using the custom layer `residualBlockLayer`. To access this layer, open the example as a live script. For an example showing how to create this custom layer, see “Define Nested Deep Learning Layer” on page 18-120.

Because you must specify the input size of the input layer of the `dlnetwork` object, you must specify the input size when creating the layer. To help determine the input size to the layer, you can use the `analyzeNetwork` function and check the size of the activations of the previous layer.

```
numFilters = 32;

layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(7,numFilters,'Stride',2,'Padding','same')
    groupNormalizationLayer('all-channels')
    reluLayer
    maxPooling2dLayer(3,'Stride',2)
    residualBlockLayer(numFilters)
    residualBlockLayer(numFilters)
    residualBlockLayer(2*numFilters,'Stride',2,'IncludeSkipConvolution',true)
    residualBlockLayer(2*numFilters)
    residualBlockLayer(4*numFilters,'Stride',2,'IncludeSkipConvolution',true)
    residualBlockLayer(4*numFilters)
    globalAveragePooling2dLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  15×1 Layer array with layers:

     1   ''   Image Input              224×224×3 images with 'zerocenter' normalization
     2   ''   Convolution              32 7×7 convolutions with stride [2  2] and padding 'same'
     3   ''   Group Normalization      Group normalization
     4   ''   ReLU                     ReLU
     5   ''   Max Pooling              3×3 max pooling with stride [2  2] and padding [0  0  0  0
     6   ''   Residual Block           Residual block with 32 filters, stride 1
     7   ''   Residual Block           Residual block with 32 filters, stride 1
     8   ''   Residual Block           Residual block with 64 filters, stride 2, and skip convolu
     9   ''   Residual Block           Residual block with 64 filters, stride 1
    10   ''   Residual Block           Residual block with 128 filters, stride 2, and skip convo
    11   ''   Residual Block           Residual block with 128 filters, stride 1
    12   ''   Global Average Pooling   Global average pooling
    13   ''   Fully Connected          5 fully connected layer
    14   ''   Softmax                  softmax
    15   ''   Classification Output    crossentropyex
```

Train Network

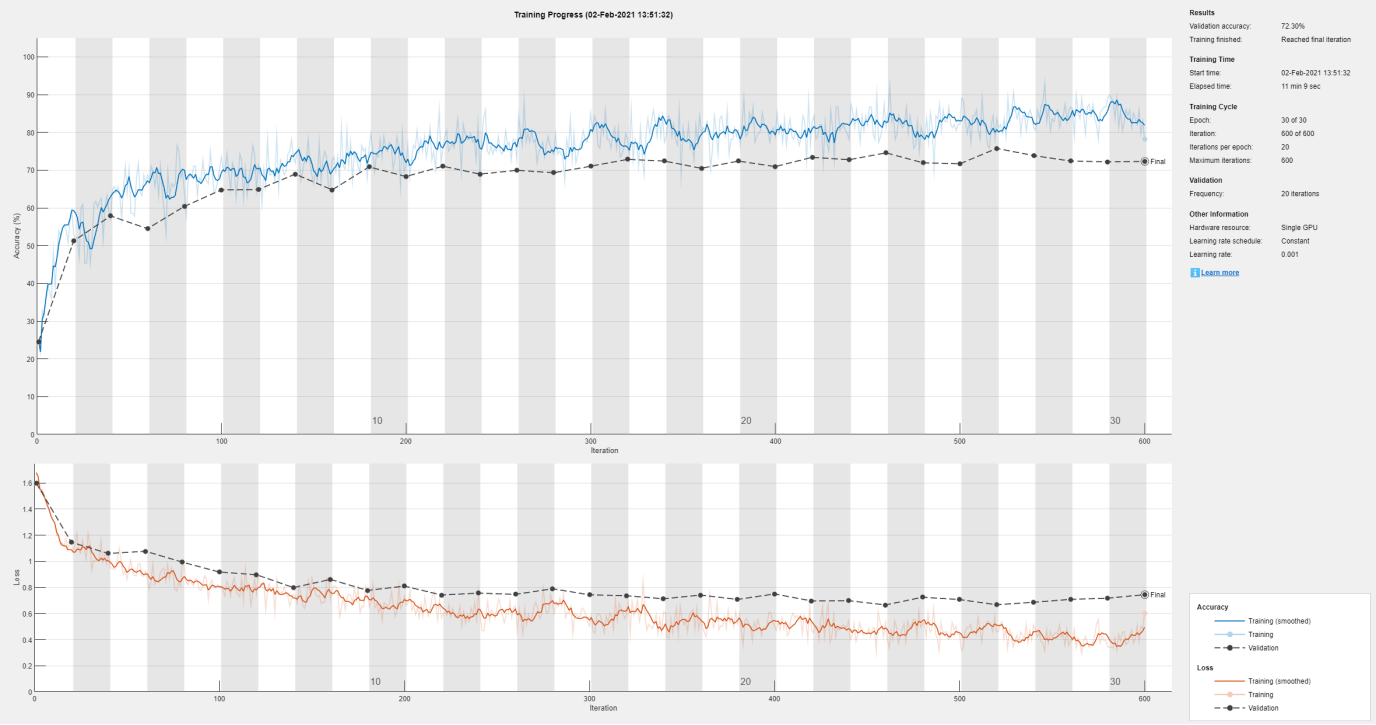
Specify training options:

- Train the network with a mini-batch size of 128.
- Shuffle the data every epoch.
- Validate the network once per epoch using the validation data.
- Display the training progress in a plot and disable the verbose output.

```
miniBatchSize = 128;
numIterationsPerEpoch = floor(augimdsTrain.NumObservations/miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the network using the `trainNetwork` function. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
net = trainNetwork(augimdsTrain,layers,options);
```



Evaluate Trained Network

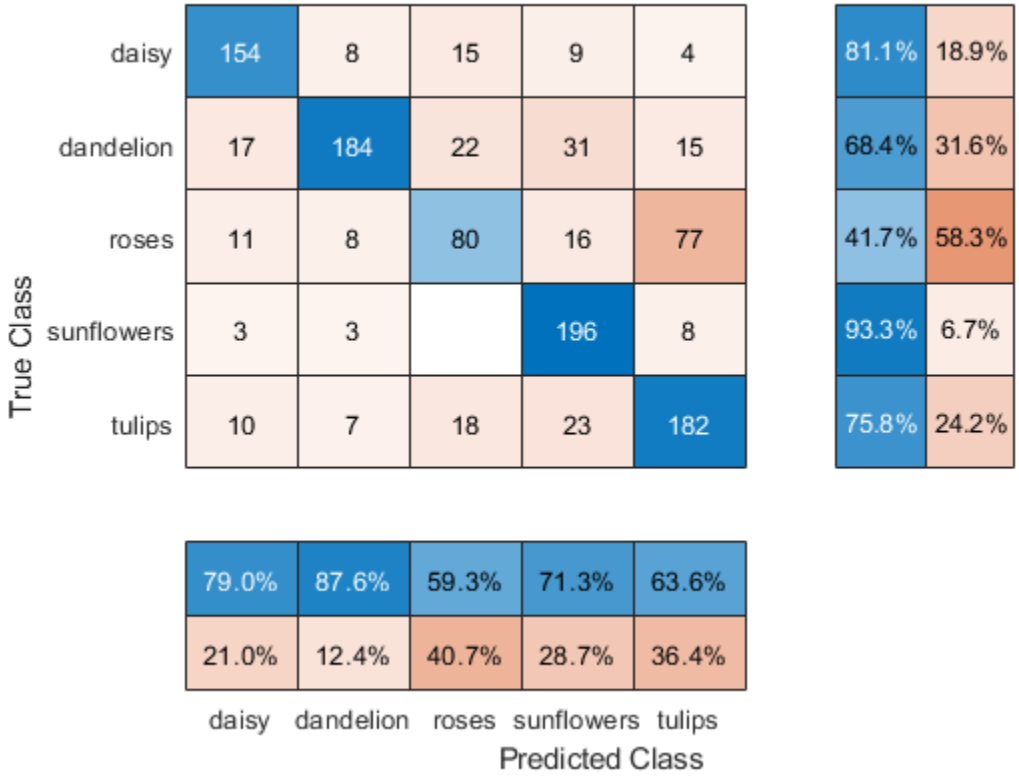
Calculate the final accuracy of the network on the training set (without data augmentation) and validation set. The accuracy is the proportion of images that the network classifies correctly.

```
YPred = classify(net,augimdsValidation);
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 0.7230
```

Visualize the classification accuracy in a confusion matrix. Display the precision and recall for each class by using column and row summaries.

```
figure
confusionchart(YValidation,YPred, ...
    'RowSummary','row-normalized', ...
    'ColumnSummary','column-normalized');
```



You can display four sample validation images with predicted labels and the predicted probabilities of the images having those labels using the following code.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title("Predicted class: " + string(label));
end
```

References

1 The TensorFlow Team. *Flowers* http://download.tensorflow.org/example_images/flower_photos.tgz

See Also

`checkLayer` | `trainNetwork` | `trainingOptions` | `analyzeNetwork` | `dlnetwork`

More About

- “Define Nested Deep Learning Layer” on page 18-120
- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154
- “List of Deep Learning Layers” on page 1-21

Define Custom Deep Learning Layer for Code Generation

If Deep Learning Toolbox does not provide the layer you require for your classification or regression problem, then you can define your own custom layer using this example as a guide. For a list of built-in layers, see “List of Deep Learning Layers” on page 1-21.

To define a custom deep learning layer, you can use the template provided in this example, which takes you through the following steps:

1. Name the layer — Give the layer a name so that you can use it in MATLAB.
2. Declare the layer properties — Specify the properties of the layer including learnable parameters and state parameters.
3. Create a constructor function (optional) — Specify how to construct the layer and initialize its properties. If you do not specify a constructor function, then at creation, the software initializes the `Name`, `Description`, and `Type` properties with `[]` and sets the number of layer inputs and outputs to 1.
4. Create forward functions — Specify how data passes forward through the layer (forward propagation) at prediction time and at training time.
5. Create reset state function (optional) — Specify how to reset state parameters.
6. Create a backward function (optional) — Specify the derivatives of the loss with respect to the input data and the learnable parameters (backward propagation). If you do not specify a backward function, then the forward functions must support `dlarray` objects.

To create a custom layer that supports code generation:

- The layer must specify the pragma `%#codegen` in the layer definition.
- The inputs of `predict` must be:
  - Consistent in dimension. Each input must have the same number of dimensions.
  - Consistent in batch size. Each input must have the same batch size.
- The outputs of `predict` must be consistent in dimension and batch size with the layer inputs.
- Nonscalar properties must have type single, double, or character array.
- Scalar properties must have type numeric, logical, or string.

Code generation supports intermediate layers with 2-D image or feature input only. Code generation does not support layers with state properties (properties with attribute `State`).

This example shows how to create a PReLU layer [1], which is a layer with a learnable parameter, and use it in a convolutional neural network. A PReLU layer performs a threshold operation, where for each channel, any input value less than zero is multiplied by a scalar learned at training time. For values less than zero, a PReLU layer applies scaling coefficients α_i to each channel of the input. These coefficients form a learnable parameter, which the layer learns during training.

This figure from [1] compares the ReLU and PReLU layer functions.

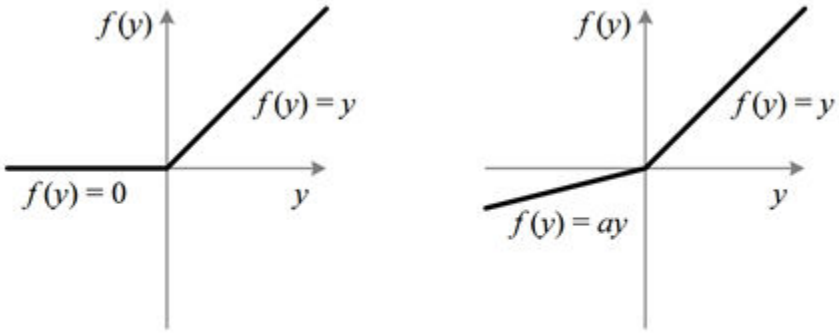


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Intermediate Layer Template

Copy the intermediate layer template into a new file in MATLAB. This template outlines the structure of an intermediate layer class definition. It outlines:

- The optional `properties` blocks for the layer properties, learnable parameters, and state parameters.
- The layer constructor function.
- The `predict` function and the optional `forward` function.
- The optional `resetState` function for layers with state properties.
- The optional `backward` function.

```
classdef myLayer < nnet.layer.Layer % & nnet.layer.Formattable (Optional)

    properties
        % (Optional) Layer properties.

        % Declare layer properties here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Declare learnable parameters here.
    end

    properties (State)
        % (Optional) Layer state parameters.

        % Declare state parameters here.
    end

    properties (Learnable, State)
        % (Optional) Nested dlnetwork objects with both learnable
        % parameters and state.

        % Declare nested networks with learnable and state parameters here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.
```

```
    % Define layer constructor function here.
end

function [Z,state] = predict(layer,X)
    % Forward input data through the layer at prediction time and
    % output the result and updated state.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Input data
    % Outputs:
    %         Z     - Output of layer forward function
    %         state - (Optional) Updated layer state.
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer predict function here.
end

function [Z,state,memory] = forward(layer,X)
    % (Optional) Forward input data through the layer at training
    % time and output the result, updated state, and a memory
    % value.
    %
    % Inputs:
    %         layer - Layer to forward propagate through
    %         X     - Layer input data
    % Outputs:
    %         Z      - Output of layer forward function
    %         state  - (Optional) Updated layer state
    %         memory - (Optional) Memory value for custom backward
    %                  function
    %
    %  - For layers with multiple inputs, replace X with X1,...,XN,
    %    where N is the number of inputs.
    %  - For layers with multiple outputs, replace Z with
    %    Z1,...,ZM, where M is the number of outputs.
    %  - For layers with multiple state parameters, replace state
    %    with state1,...,stateK, where K is the number of state
    %    parameters.

    % Define layer forward function here.
end

function layer = resetState(layer)
    % (Optional) Reset layer state.

    % Define reset state function here.
end

function [dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)
    % (Optional) Backward propagate the derivative of the loss
    % function through the layer.
    %
    % Inputs:
    %         layer   - Layer to backward propagate through
    %         X       - Layer input data
    %         Z       - Layer output data
    %         dLdZ    - Derivative of loss with respect to layer
    %                   output
    %         dLdSout - (Optional) Derivative of loss with respect
    %                   to state output
    %         memory  - Memory value from forward function
    % Outputs:
    %         dLdX   - Derivative of loss with respect to layer input
```

```
            %         dLdW   - (Optional) Derivative of loss with respect to
            %                  learnable parameter
            %         dLdSin - (Optional) Derivative of loss with respect to
            %                  state input
            %
            %  - For layers with state parameters, the backward syntax must
            %    include both dLdSout and dLdSin, or neither.
            %  - For layers with multiple inputs, replace X and dLdX with
            %    X1,...,XN and dLdX1,...,dLdXN, respectively, where N is
            %    the number of inputs.
            %  - For layers with multiple outputs, replace Z and dlZ with
            %    Z1,...,ZM and dLdZ,...,dLdZM, respectively, where M is the
            %    number of outputs.
            %  - For layers with multiple learnable parameters, replace
            %    dLdW with dLdW1,...,dLdWP, where P is the number of
            %    learnable parameters.
            %  - For layers with multiple state parameters, replace dLdSin
            %    and dLdSout with dLdSin1,...,dLdSinK and
            %    dLdSout1,...dldSoutK, respectively, where K is the number
            %    of state parameters.

            % Define layer backward function here.
        end
    end
end
```

Name Layer and Specify Superclasses

First, give the layer a name. In the first line of the class file, replace the existing name `myLayer` with `codegenPreluLayer` and add a comment describing the layer.

```
classdef codegenPreluLayer < nnet.layer.Layer & nnet.layer.Formattable
    % Example custom PReLU layer with codegen support.

    ...
end
```

If you do not specify a backward function, then the layer functions, by default, receive *unformatted* `dlarray` objects as input. To specify that the layer receives *formatted* `dlarray` objects as input and also outputs formatted `dlarray` objects, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

The layer does not require formattable inputs, so remove the optional `nnet.layer.Formattable` superclass.

```
classdef codegenPreluLayer < nnet.layer.Layer
    % Example custom PReLU layer with codegen support.

    ...
end
```

Next, rename the `myLayer` constructor function (the first function in the `methods` section) so that it has the same name as the layer.

```
    methods
        function layer = codegenPreluLayer()
            ...
        end

        ...
     end
```

Save Layer

Save the layer class file in a new file named `codegenPreluLayer.m`. The file name must match the layer name. To use the layer, you must save the file in the current folder or in a folder on the MATLAB path.

Specify Code Generation Pragma

Add the `%#codegen` directive (or pragma) to your layer definition to indicate that you intend to generate code for this layer. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that result in errors during code generation.

```
classdef codegenPreluLayer < nnet.layer.Layer
    % Example custom PReLU layer with codegen support.

    %#codegen

    ...
end
```

Declare Properties and Learnable Parameters

Declare the layer properties in the `properties` section and declare learnable parameters by listing them in the `properties (Learnable)` section.

By default, custom intermediate layers have these properties. Do not declare these properties in the `properties` section.

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. For `Layer` array input, the `trainNetwork`, `assembleNetwork`, `layerGraph`, and `dlnetwork` functions automatically assign names to layers with `Name` set to `''`.
`Description`	One-line description of the layer, specified as a string scalar or a character vector. This description appears when the layer is displayed in a `Layer` array. If you do not specify a layer description, then the software displays the layer class name.
`Type`	Type of the layer, specified as a character vector or a string scalar. The value of `Type` appears when the layer is displayed in a `Layer` array. If you do not specify a layer type, then the software displays the layer class name.

Property	Description
`NumInputs`	Number of inputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumInputs` to the number of names in `InputNames`. The default value is 1.
`InputNames`	Input names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumInputs` is greater than 1, then the software automatically sets `InputNames` to `{'in1',...,'inN'}`, where `N` is equal to `NumInputs`. The default value is `{'in'}`.
`NumOutputs`	Number of outputs of the layer, specified as a positive integer. If you do not specify this value, then the software automatically sets `NumOutputs` to the number of names in `OutputNames`. The default value is 1.
`OutputNames`	Output names of the layer, specified as a cell array of character vectors. If you do not specify this value and `NumOutputs` is greater than 1, then the software automatically sets `OutputNames` to `{'out1',...,'outM'}`, where `M` is equal to `NumOutputs`. The default value is `{'out'}`.

If the layer has no other properties, then you can omit the `properties` section.

Tip If you are creating a layer with multiple inputs, then you must set either the `NumInputs` or `InputNames` properties in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` properties in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48.

To support code generation:

- Nonscalar properties must have type single, double, or character array.
- Scalar properties must be numeric or have type logical or string.

A PReLU layer does not require any additional properties, so you can remove the `properties` section.

A PReLU layer has only one learnable parameter, the scaling coefficient a. Declare this learnable parameter in the `properties (Learnable)` section and call the parameter `Alpha`.

```
properties (Learnable)
    % Layer learnable parameters

    % Scaling coefficient
    Alpha
end
```

Create Constructor Function

Create the function that constructs the layer and initializes the layer properties. Specify any variables required to create the layer as inputs to the constructor function.

The PReLU layer constructor function requires two input arguments: the number of channels of the expected input data and the layer name. The number of channels specifies the size of the learnable parameter `Alpha`. Specify two input arguments named `numChannels` and `name` in the `codegenPreluLayer` function. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = codegenPreluLayer(numChannels, name)
    % layer = codegenPreluLayer(numChannels) creates a PReLU layer with
    % numChannels channels and specifies the layer name.

    ...
end
```

Code generation does not support `arguments` blocks.

Initialize Layer Properties

Initialize the layer properties, including learnable parameters, in the constructor function. Replace the comment `% Layer constructor function goes here` with code that initializes the layer properties.

Set the `Name` property to the input argument `name`.

```
% Set layer name.
layer.Name = name;
```

Give the layer a one-line description by setting the `Description` property of the layer. Set the description to describe the type of layer and its size.

```
% Set layer description.
layer.Description = "PReLU with " + numChannels + " channels";
```

For a PReLU layer, when the input values are negative, the layer multiplies each channel of the input by the corresponding channel of `Alpha`. Initialize the learnable parameter `Alpha` as a random vector of size 1-by-1-by-`numChannels`. With the third dimension specified as size `numChannels`, the layer can use element-wise multiplication of the input in the forward function. `Alpha` is a property of the layer object, so you must assign the vector to `layer.Alpha`.

```
% Initialize scaling coefficient.
layer.Alpha = rand([1 1 numChannels]);
```

View the completed constructor function.

```
function layer = codegenPreluLayer(numChannels, name)
    % layer = codegenPreluLayer(numChannels, name) creates a PReLU
    % layer for 2-D image input with numChannels channels and specifies
    % the layer name.

    % Set layer name.
    layer.Name = name;

    % Set layer description.
    layer.Description = "PReLU with " + numChannels + " channels";
```

```
        % Initialize scaling coefficient.
        layer.Alpha = rand([1 1 numChannels]);
    end
```

With this constructor function, the command `codegenPreluLayer(3,'prelu')` creates a PReLU layer with three channels and the name `'prelu'`.

Create Forward Functions

Create the layer forward functions to use at prediction time and training time.

Create a function named `predict` that propagates the data forward through the layer at *prediction time* and outputs the result.

The `predict` function syntax depends on the type of layer.

- `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

Because a PReLU layer has only one input and one output, the syntax for `predict` for a PReLU layer is `Z = predict(layer,X)`.

Code generation supports custom intermediate layers with 2-D image input only. The inputs are *h*-by-*w*-by-*c*-by-*N* arrays, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the images, respectively, and *N* is the number of observations. The observation dimension is 4.

For code generation support, all the layer inputs must have the same number of dimensions and batch size.

By default, the layer uses `predict` as the forward function at training time. To use a different forward function at training time, or retain a value required for a custom backward function, you must also create a function named `forward`. The software does not generate code for the `forward` function but it must be code generation compatible.

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The `forward` function syntax depends on the type of layer:

- `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output.
- `[Z,state] = forward(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter.
- `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters:

- For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`.
- For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`.
- For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters.

Tip If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`.

If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

The PReLU operation is given by

$$f(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ \alpha_i x_i & \text{if } x_i \le 0 \end{cases}$$

where x_i is the input of the nonlinear activation f on channel i, and α_i is the coefficient controlling the slope of the negative part. The subscript i in α_i indicates that the nonlinear activation can vary on different channels.

Implement this operation in `predict`. In `predict`, the input `X` corresponds to x in the equation. The output `Z` corresponds to $f(x_i)$.

Add a comment to the top of the function that explains the syntaxes of the function.

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

Implementing the `backward` function is optional when the forward functions fully support `dlarray` input. For code generation support, the `predict` function must also support numeric input.

One way to calculate the output of the PReLU operation is to use the following code.

```
Z = max(X,0) + layer.Alpha .* min(0,X);
```

Because code generation does not support implicit expansion via the `.*` operation, you can use the `bsxfun` function instead.

```
Z = max(X,0) + bsxfun(@times, layer.Alpha, min(0,X));
```

However, the `bsxfun` does not support `dlarray` input. To implement the `predict` function, which supports both code generation and `dlarray` input, use an `if` statement with the `isdlarray` function to select the appropriate code for the type of input.

```
        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            if isdlarray(X)
                Z = max(X,0) + layer.Alpha .* min(0,X);
            else
                Z = max(X,0) + bsxfun(@times, layer.Alpha, min(0,X));
            end
        end
```

Because the `predict` function fully supports `dlarray` objects, defining the `backward` function is optional. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Completed Layer

View the completed layer class file.

```
classdef codegenPreluLayer < nnet.layer.Layer
    % Example custom PReLU layer with codegen support.

    %#codegen

    properties (Learnable)
        % Layer learnable parameters

        % Scaling coefficient
        Alpha
    end

    methods
        function layer = codegenPreluLayer(numChannels, name)
            % layer = codegenPreluLayer(numChannels, name) creates a PReLU
            % layer for 2-D image input with numChannels channels and specifies
            % the layer name.

            % Set layer name.
            layer.Name = name;

            % Set layer description.
```

```
            layer.Description = "PReLU with " + numChannels + " channels";

            % Initialize scaling coefficient.
            layer.Alpha = rand([1 1 numChannels]);
        end

        function Z = predict(layer, X)
            % Z = predict(layer, X) forwards the input data X through the
            % layer and outputs the result Z.

            if isdlarray(X)
                Z = max(X,0) + layer.Alpha .* min(0,X);
            else
                Z = max(X,0) + bsxfun(@times, layer.Alpha, min(0,X));
            end
        end
    end
end
```

Check Custom Layer for Code Generation Compatibility

Check the code generation compatibility of the custom layer `codegenPreluLayer`.

The custom layer `codegenPreluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check its validity using `checkLayer`. Specify the valid input size as the size of a single observation of typical input to the layer. The layer expects 4-D array inputs, where the first three dimensions correspond to the height, width, and number of channels of the previous layer output, and the fourth dimension corresponds to the observations.

Specify the typical size of the input of an observation and set the `'ObservationDimension'` option to 4. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to `true`. The `checkLayer` function does not check for functions that are not compatible with code generation. To check that the custom layer definition is supported for code generation, first use the **Code Generation Readiness** app. For more information, see “Check Code by Using the Code Generation Readiness Tool” (MATLAB Coder).

```
layer = codegenPreluLayer(20,"prelu");
validInputSize = [24 24 20];
checkLayer(layer,validInputSize,ObservationDimension=4,CheckCodegenCompatibility=true)
```

```
Skipping GPU tests. No compatible GPU device found.

Running nnet.checklayer.TestLayerWithoutBackward
.......... .......... ...
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     23 Passed, 0 Failed, 0 Incomplete, 5 Skipped.
     Time elapsed: 1.0234 seconds.
```

The function does not detect any issues with the layer.

References

[1] "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *In 2015 IEEE International Conference on Computer Vision (ICCV)*, 1026–34. Santiago, Chile: IEEE, 2015. https://doi.org/10.1109/ICCV.2015.123.

See Also

`functionLayer` | `checkLayer` | `setLearnRateFactor` | `setL2Factor` | `getLearnRateFactor` | `getL2Factor` | `findPlaceholderLayers` | `replaceLayer` | `assembleNetwork` | `PlaceholderLayer`

More About

- “Code Generation for Deep Learning Networks” on page 20-2
- “Code Generation For Object Detection Using YOLO v3 Deep Learning” on page 20-35
- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Nested Deep Learning Layer” on page 18-120
- “Check Custom Layer Validity” on page 18-154

Check Custom Layer Validity

If you create a custom deep learning layer, then you can use the `checkLayer` function to check that the layer is valid. The function checks layers for validity, GPU compatibility, correctly defined gradients, and code generation compatibility. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer. To check with multiple observations, use the `ObservationDimension` option. To check for code generation compatibility, set the `CheckCodegenCompatibility` option to `1` (true). For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Check Custom Layer Validity

Check the validity of the example custom layer `preluLayer`.

The custom layer `preluLayer`, attached to this is example as a supporting file, applies the PReLU operation to the input data. To access this layer, open this example as a live script.

Create an instance of the layer and check that it is valid using `checkLayer`. Set the valid input size to the typical size of a single observation input to the layer. For a single input, the layer expects observations of size *h*-by-*w*-by-*c*, where *h*, *w*, and *c* are the height, width, and number of channels of the previous layer output, respectively.

Specify `validInputSize` as the typical size of an input array.

```
layer = preluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize)
```

```
Skipping multi-observation tests. To enable tests with multiple observations, specify the 'Observ
For 2-D image data, set 'ObservationDimension' to 4.
For 3-D image data, set 'ObservationDimension' to 5.
For sequence data, set 'ObservationDimension' to 2.

Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ..
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     12 Passed, 0 Failed, 0 Incomplete, 16 Skipped.
     Time elapsed: 0.17942 seconds.
```

The results show the number of passed, failed, and skipped tests. If you do not specify the `ObservationsDimension` option, or do not have a GPU, then the function skips the corresponding tests.

Check Multiple Observations

For multi-observation input, the layer expects an array of observations of size *h*-by-*w*-by-*c*-by-*N*, where *h*, *w*, and *c* are the height, width, and number of channels, respectively, and *N* is the number of observations.

To check the layer validity for multiple observations, specify the typical size of an observation and set the `ObservationDimension` option to 4.

```
layer = preluLayer(20);
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,ObservationDimension=4)
```

```
Skipping GPU tests. No compatible GPU device found.

Skipping code generation compatibility tests. To check validity of the layer for code generation

Running nnet.checklayer.TestLayerWithoutBackward
.......... ........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     18 Passed, 0 Failed, 0 Incomplete, 10 Skipped.
     Time elapsed: 0.077629 seconds.
```

In this case, the function does not detect any issues with the layer.

List of Tests

The `checkLayer` function checks the validity of a custom layer by performing a series of tests.

Intermediate Layers

The `checkLayer` function uses these tests to check the validity of custom intermediate layers (layers of type `nnet.layer.Layer`).

Test	Description
`functionSyntaxesAreCorrect`	The syntaxes of the layer functions are correctly defined.
`predictDoesNotError`	`predict` function does not error.
`forwardDoesNotError`	When specified, the `forward` function does not error.
`forwardPredictAreConsistentInSize`	When `forward` is specified, `forward` and `predict` output values of the same size.
`backwardDoesNotError`	When specified, `backward` does not error.

Test	Description
`backwardIsConsistentInSize`	When `backward` is specified, the outputs of `backward` are consistent in size: • The derivatives with respect to each input are the same size as the corresponding input. • The derivatives with respect to each learnable parameter are the same size as the corresponding learnable parameter.
`predictIsConsistentInType`	The outputs of `predict` are consistent in type with the inputs.
`forwardIsConsistentInType`	When `forward` is specified, the outputs of `forward` are consistent in type with the inputs.
`backwardIsConsistentInType`	When `backward` is specified, the outputs of `backward` are consistent in type with the inputs.
`gradientsAreNumericallyCorrect`	When `backward` is specified, the gradients computed in `backward` are consistent with the numerical gradients.
`backwardPropagationDoesNotError`	When `backward` is not specified, the derivatives can be computed using automatic differentiation.
`predictReturnsValidStates`	For layers with state properties, the `predict` function returns valid states.
`forwardReturnsValidStates`	For layers with state properties, the `forward` function, if specified, returns valid states.
`resetStateDoesNotError`	For layers with state properties, the `resetState` function, if specified, does not error and resets the states to valid states.
`codegenPragmaDefinedInClassDef`	The pragma `"%#codegen"` for code generation is specified in class file.
`checkForSupportedLayerPropertiesForCodegen`	The layer properties support code generation.
`predictIsValidForCodeGeneration`	`predict` is valid for code generation.
`doesNotHaveStateProperties`	For code generation, the layer does not have state properties.
`supportedFunctionLayer`	For code generation, the layer is not a `FunctionLayer` object.

Some tests run multiple times. These tests also check different data types and for GPU compatibility:

- `predictIsConsistentInType`
- `forwardIsConsistentInType`
- `backwardIsConsistentInType`

To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

Output Layers

The `checkLayer` function uses these tests to check the validity of custom output layers (layers of type `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`).

Test	Description
`forwardLossDoesNotError`	`forwardLoss` does not error.
`backwardLossDoesNotError`	`backwardLoss` does not error.
`forwardLossIsScalar`	The output of `forwardLoss` is scalar.
`backwardLossIsConsistentInSize`	When `backwardLoss` is specified, the output of `backwardLoss` is consistent in size: `dLdY` is the same size as the predictions `Y`.
`forwardLossIsConsistentInType`	The output of `forwardLoss` is consistent in type: `loss` is the same type as the predictions `Y`.
`backwardLossIsConsistentInType`	When `backwardLoss` is specified, the output of `backwardLoss` is consistent in type: `dLdY` must be the same type as the predictions `Y`.
`gradientsAreNumericallyCorrect`	When `backwardLoss` is specified, the gradients computed in `backwardLoss` are numerically correct.
`backwardPropagationDoesNotError`	When `backwardLoss` is not specified, the derivatives can be computed using automatic differentiation.

The `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` tests also check for GPU compatibility. To execute the layer functions on a GPU, the functions must support inputs and outputs of type `gpuArray` with the underlying data type `single`.

Generated Data

To check the layer validity, the `checkLayer` function generates data depending on the type of layer:

Layer Type	Description of Generated Data
Intermediate	Values in the range [-1,1]
Regression output	Predictions and targets with values in the range [-1,1]
Classification output	Predictions with values in the range [0,1]. If you specify the `ObservationDimension` option, then the targets are one-hot encoded vectors (vectors containing a single 1, and 0 elsewhere). If you do not specify the `ObservationDimension` option, then the targets are values in the range [0,1].

To check for multiple observations, specify the observation dimension using the `ObservationDimension` option. If you specify the observation dimension, then the `checkLayer`

function checks that the layer functions are valid using generated data with mini-batches of size 1 and 2. If you do not specify this name-value pair, then the function skips the tests that check that the layer functions are valid for multiple observations.

Diagnostics

If a test fails when you use `checkLayer`, then the function provides a test diagnostic and a framework diagnostic. The test diagnostic highlights any issues found with the layer. The framework diagnostic provides more detailed information.

Function Syntaxes

The test `functionSyntaxesAreCorrect` checks that the layer functions have correctly defined syntaxes.

Test Diagnostic	Description	Possible Solution
`Incorrect number of input arguments for 'predict' in Layer.`	The syntax for the `predict` function is not consistent with the number of layer inputs.	Specify the correct number of input and output arguments in `predict`. The `predict` function syntax depends on the type of layer. • `Z = predict(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output. • `[Z,state] = predict(layer,X)` also outputs the updated state parameter `state`, where `layer` has a single state parameter. You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters: • For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`. • For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The

Test Diagnostic	Description	Possible Solution
`Incorrect number of output arguments for 'predict' in Layer`	The syntax for the `predict` function is not consistent with the number of layer outputs.	`NumOutputs` property must match `M`. • For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters. **Tip** If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`. If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`. **Tip** If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `predict` function of the custom layer, use the `predict` function for the `dlnetwork`. Using the `dlnetwork` object `predict` function ensures that the software uses the correct layer operations for prediction.
`Incorrect number of input arguments for 'forward' in Layer`	The syntax for the optional `forward` function is not consistent with the number of layer inputs.	Specify the correct number of input and output arguments in `forward`. The `forward` function syntax depends on the type of layer: • `Z = forward(layer,X)` forwards the input data `X` through the layer and outputs the result `Z`, where `layer` has a single input, a single output. • `[Z,state] = forward(layer,X)` also

Test Diagnostic	Description	Possible Solution
`Incorrect number of output arguments for 'forward' in Layer`	The syntax for the optional `forward` function is not consistent with the number of layer outputs.	outputs the updated state parameter `state`, where `layer` has a single state parameter. • `[__,memory] = forward(layer,X)` also returns a memory value for a custom `backward` function using any of the previous syntaxes. If the layer has both a custom `forward` function and a custom `backward` function, then the forward function must return a memory value. You can adjust the syntaxes for layers with multiple inputs, multiple outputs, or multiple state parameters: • For layers with multiple inputs, replace `X` with `X1,...,XN`, where `N` is the number of inputs. The `NumInputs` property must match `N`. • For layers with multiple outputs, replace `Z` with `Z1,...,ZM`, where `M` is the number of outputs. The `NumOutputs` property must match `M`. • For layers with multiple state parameters, replace `state` with `state1,...,stateK`, where `K` is the number of state parameters. **Tip** If the number of inputs to the layer can vary, then use `varargin` instead of `X1,…,XN`. In this case, `varargin` is a cell array of the inputs, where `varargin{i}` corresponds to `Xi`. If the number of outputs can vary, then use `varargout` instead of `Z1,…,ZN`. In this

Test Diagnostic	Description	Possible Solution
		case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`. **Tip** If the custom layer has a `dlnetwork` object for a learnable parameter, then in the `forward` function of the custom layer, use the `forward` function of the `dlnetwork` object. Using the `dlnetwork` object `forward` function ensures that the software uses the correct layer operations for training.
`Incorrect number of input arguments for 'backward' in Layer`	The syntax for the optional `backward` function is not consistent with the number of layer inputs and outputs.	Specify the correct number of input and output arguments in `backward`. The `backward` function syntax depends on the type of layer. • `dLdX = backward(layer,X,Z,dLdZ,memory)` returns the derivatives `dLdX` of the loss with respect to the layer input, where `layer` has a single input and a single output. `Z` corresponds to the forward function output and `dLdZ` corresponds to the derivative of the loss with respect to `Z`. The function input `memory` corresponds to the memory output of the forward function. • `[dLdX,dLdW] = backward(layer,X,Z,dLdZ,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter, where `layer` has a single learnable parameter. • `[dLdX,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdSin` of the loss with

Test Diagnostic	Description	Possible Solution
`Incorrect number of output arguments for 'backward' in Layer`	The syntax for the optional `backward` function is not consistent with the number of layer outputs.	respect to the state input using any of the previous syntaxes, where `layer` has a single state parameter and `dLdSout` corresponds to the derivative of the loss with respect to the layer state output. • `[dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter and returns the derivative `dLdSin` of the loss with respect to the layer state input using any of the previous syntaxes, where `layer` has a single state parameter and single learnable parameter. You can adjust the syntaxes for layers with multiple inputs, multiple outputs, multiple learnable parameters, or multiple state parameters: • For layers with multiple inputs, replace `X` and `dLdX` with `X1,...,XN` and `dLdX1,...,dLdXN`, respectively, where `N` is the number of inputs. • For layers with multiple outputs, replace `Z` and `dLdZ` with `Z1,...,ZM` and `dLdZ1,...,dLdZM`, respectively, where `M` is the number of outputs. • For layers with multiple learnable parameters, replace `dLdW` with `dLdW1,...,dLdWP`, where `P` is the number of learnable parameters. • For layers with multiple state parameters, replace `dLdSin` and `dLdSout` with

Test Diagnostic	Description	Possible Solution
		`dLdSin1,...,dLdSinK` and `dLdSout1,...,dLdSoutK`, respectively, where `K` is the number of state parameters. To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with ~. **Tip** If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where the first `N` elements correspond to the `N` layer inputs, the next `M` elements correspond to the `M` layer outputs, the next `M` elements correspond to the derivatives of the loss with respect to the `M` layer outputs, the next `K` elements correspond to the `K` derivatives of the loss with respect to the `K` states outputs, and the last element corresponds to `memory`. If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where the first `N` elements correspond to the `N` the derivatives of the loss with respect to the `N` layer inputs, the next `P` elements correspond to the derivatives of the loss with respect to the `P` learnable parameters, and the next `K` elements correspond to the derivatives of the loss with respect to the `K` state inputs. **Tip** If the layer forward functions support `dlarray`

Test Diagnostic	Description	Possible Solution
		objects, then the software automatically determines the backward function and you do not need to specify the `backward` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

For layers with multiple inputs or outputs, you must set the values of the layer properties `NumInputs` (or alternatively, `InputNames`) and `NumOutputs` (or alternatively, `OutputNames`) in the layer constructor function, respectively.

Multiple Observations

The `checkLayer` function checks that the layer functions are valid for single and multiple observations. To check for multiple observations, specify the observation dimension using the `ObservationDimension` option. If you specify the observation dimension, then the `checkLayer` function checks that the layer functions are valid using generated data with mini-batches of size 1 and 2. If you do not specify this name-value pair, then the function skips the tests that check that the layer functions are valid for multiple observations.

Test Diagnostic	Description	Possible Solution
`Skipping multi-observation tests. To enable checks with multiple observations, specify the 'ObservationDimension' parameter in checkLayer.`	If you do not specify the `'ObservationDimension'` parameter in `checkLayer`, then the function skips the tests that check data with multiple observations.	Use the command `checkLayer(layer,validInputSize,'ObservationDimension',dim)`, where `layer` is an instance of the custom layer, `validInputSize` is a vector specifying the valid input size to the layer, and `dim` specifies the dimension of the observations in the layer input. For more information, see “Layer Input Sizes”.

Functions Do Not Error

These tests check that the layers do not error when passed input data of valid size.

Intermediate Layers

The tests `predictDoesNotError`, `forwardDoesNotError`, and `backwardDoesNotError` check that the layer functions do not error when passed inputs of valid size. If you specify an observation dimension, then the function checks the layer for both a single observation and multiple observations.

Test Diagnostic	Description	Possible Solution
`The function 'predict' threw an error:`	The `predict` function errors when passed data of size `validInputSize`.	Address the error described in the `Framework Diagnostic` section. **Tip** If the layer forward functions support `dlarray` objects, then the software automatically determines the backward function and you do not need to specify the `backward` function. For a list of functions that support `dlarray` objects, see "List of Functions with dlarray Support" on page 18-423.
`The function 'forward' threw an error:`	The optional `forward` function errors when passed data of size `validInputSize`.	
`The function 'backward' threw an error:`	The optional `backward` function errors when passed the output of `predict`.	

Output Layers

The tests `forwardLossDoesNotError` and `backwardLossDoesNotError` check that the layer functions do not error when passed inputs of valid size. If you specify an observation dimension, then the function checks the layer for both a single observation and multiple observations.

Test Diagnostic	Description	Possible Solution
`The function 'forwardLoss' threw an error:`	The `forwardLoss` function errors when passed data of size `validInputSize`.	Address the error described in the `Framework Diagnostic` section. **Tip** If the `forwardLoss` function supports `dlarray` objects, then the software automatically determines the backward loss function and you do not need to specify the `backwardLoss` function. For a list of functions that support `dlarray` objects, see "List of Functions with dlarray Support" on page 18-423.
`The function 'backwardLoss' threw an error:`	The optional `backwardLoss` function errors when passed data of size `validInputSize`.	

Outputs Are Consistent in Size

These tests check that the layer function outputs are consistent in size.

Intermediate Layers

The test `backwardIsConsistentInSize` checks that the `backward` function outputs derivatives of the correct size.

The `backward` function syntax depends on the type of layer.

- `dLdX = backward(layer,X,Z,dLdZ,memory)` returns the derivatives `dLdX` of the loss with respect to the layer input, where `layer` has a single input and a single output. `Z` corresponds to the forward function output and `dLdZ` corresponds to the derivative of the loss with respect to `Z`. The function input `memory` corresponds to the memory output of the forward function.
- `[dLdX,dLdW] = backward(layer,X,Z,dLdZ,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter, where `layer` has a single learnable parameter.
- `[dLdX,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdSin` of the loss with respect to the state input using any of the previous syntaxes, where `layer` has a single state parameter and `dLdSout` corresponds to the derivative of the loss with respect to the layer state output.
- `[dLdX,dLdW,dLdSin] = backward(layer,X,Z,dLdZ,dLdSout,memory)` also returns the derivative `dLdW` of the loss with respect to the learnable parameter and returns the derivative `dLdSin` of the loss with respect to the layer state input using any of the previous syntaxes, where `layer` has a single state parameter and single learnable parameter.

You can adjust the syntaxes for layers with multiple inputs, multiple outputs, multiple learnable parameters, or multiple state parameters:

- For layers with multiple inputs, replace `X` and `dLdX` with `X1,...,XN` and `dLdX1,...,dLdXN`, respectively, where `N` is the number of inputs.
- For layers with multiple outputs, replace `Z` and `dLdZ` with `Z1,...,ZM` and `dLdZ1,...,dLdZM`, respectively, where `M` is the number of outputs.
- For layers with multiple learnable parameters, replace `dLdW` with `dLdW1,...,dLdWP`, where `P` is the number of learnable parameters.
- For layers with multiple state parameters, replace `dLdSin` and `dLdSout` with `dLdSin1,...,dLdSinK` and `dLdSout1,...,dLdSoutK`, respectively, where `K` is the number of state parameters.

To reduce memory usage by preventing unused variables being saved between the forward and backward pass, replace the corresponding input arguments with `~`.

Tip If the number of inputs to `backward` can vary, then use `varargin` instead of the input arguments after `layer`. In this case, `varargin` is a cell array of the inputs, where the first `N` elements correspond to the `N` layer inputs, the next `M` elements correspond to the `M` layer outputs, the next `M` elements correspond to the derivatives of the loss with respect to the `M` layer outputs, the next `K` elements correspond to the `K` derivatives of the loss with respect to the `K` states outputs, and the last element corresponds to `memory`.

If the number of outputs can vary, then use `varargout` instead of the output arguments. In this case, `varargout` is a cell array of the outputs, where the first `N` elements correspond to the `N` the derivatives of the loss with respect to the `N` layer inputs, the next `P` elements correspond to the derivatives of the loss with respect to the `P` learnable parameters, and the next `K` elements correspond to the derivatives of the loss with respect to the `K` state inputs.

The derivatives `dLdX1`, ..., `dLdXn` must be the same size as the corresponding layer inputs, and `dLdW1,…,dLdWk` must be the same size as the corresponding learnable parameters. The sizes must be consistent for input data with single and multiple observations.

Test Diagnostic	Description	Possible Solution
`Incorrect size of 'dLdX' for 'backward'.`	The derivatives of the loss with respect to the layer inputs must be the same size as the corresponding layer input.	Return the derivatives `dLdX1,…,dLdXn` with the same size as the corresponding layer inputs `X1,…,Xn`.
`Incorrect size of the derivative of the loss with respect to the input 'in1' for 'backward'`		
`The size of 'Z' returned from 'forward' must be the same as for 'predict'.`	The outputs of `predict` must be the same size as the corresponding outputs of `forward`.	Return the outputs `Z1,…,Zm` of `predict` with the same size as the corresponding outputs `Z1,…,Zm` of `forward`.
`Incorrect size of the derivative of the loss with respect to 'W' for 'backward'.`	The derivatives of the loss with respect to the learnable parameters must be the same size as the corresponding learnable parameters.	Return the derivatives `dLdW1,…,dLdWk` with the same size as the corresponding learnable parameters `W1,…,Wk`.

Tip If the layer forward functions support `dlarray` objects, then the software automatically determines the backward function and you do not need to specify the `backward` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Output Layers

The test `forwardLossIsScalar` checks that the output of the `forwardLoss` function is scalar. When the `backwardLoss` function is specified, the test `backwardLossIsConsistentInSize` checks that the outputs of `forwardLoss` and `backwardLoss` are of the correct size.

The syntax for `forwardLoss` is `loss = forwardLoss(layer,Y,T)`. The input `Y` corresponds to the predictions made by the network. These predictions are the output of the previous layer. The input `T` corresponds to the training targets. The output `loss` is the loss between `Y` and `T` according to the specified loss function. The output `loss` must be scalar.

If the `forwardLoss` function supports `dlarray` objects, then the software automatically determines the backward loss function and you do not need to specify the `backwardLoss` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

The syntax for `backwardLoss` is `dLdY = backwardLoss(layer,Y,T)`. The input `Y` contains the predictions made by the network and `T` contains the training targets. The output `dLdY` is the derivative of the loss with respect to the predictions `Y`. The output `dLdY` must be the same size as the layer input `Y`.

Test Diagnostic	Description	Possible Solution
`Incorrect size of 'loss' for 'forwardLoss'.`	The output `loss` of `forwardLoss` must be a scalar.	Return the output `loss` as a scalar. For example, if you have multiple values of the loss, then you can use `mean` or `sum`.

Test Diagnostic	Description	Possible Solution
`Incorrect size of the derivative of loss 'dLdY' for 'backwardLoss'.`	When `backwardLoss` is specified, the derivatives of the loss with respect to the layer input must be the same size as the layer input.	Return derivative `dLdY` with the same size as the layer input `Y`. If the `forwardLoss` function supports `dlarray` objects, then the software automatically determines the backward loss function and you do not need to specify the `backwardLoss` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Consistent Data Types and GPU Compatibility

These tests check that the layer function outputs are consistent in type and that the layer functions are GPU compatible.

If the layer forward functions fully support `dlarray` objects, then the layer is GPU compatible. Otherwise, to be GPU compatible, the layer functions must support inputs and return outputs of type `gpuArray`.

Many MATLAB built-in functions support `gpuArray` and `dlarray` input arguments. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. For a list of functions that execute on a GPU, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox). To use a GPU for deep learning, you must also have a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

Intermediate Layers

The tests `predictIsConsistentInType`, `forwardIsConsistentInType`, and `backwardIsConsistentInType` check that the layer functions output variables of the correct data type. The tests check that the layer functions return consistent data types when given inputs of the data types `single`, `double`, and `gpuArray` with the underlying types `single` or `double`.

Tip If you preallocate arrays using functions such as `zeros`, then you must ensure that the data types of these arrays are consistent with the layer function inputs. To create an array of zeros of the same data type as another array, use the `"like"` option of `zeros`. For example, to initialize an array of zeros of size `sz` with the same data type as the array `X`, use `Z = zeros(sz,"like",X)`.

Test Diagnostic	Description	Possible Solution
`Incorrect type of 'Z' for 'predict'.`	The types of the outputs `Z1,…,Zm` of the `predict` function must be consistent with the inputs `X1,…,Xn`.	Return the outputs `Z1,…,Zm` with the same type as the inputs `X1,…,Xn`.
`Incorrect type of output 'out1' for 'predict'.`		

Test Diagnostic	Description	Possible Solution
`Incorrect type of 'Z' for 'forward'.`	The types of the outputs `Z1,…,Zm` of the optional `forward` function must be consistent with the inputs `X1,…,Xn`.	
`Incorrect type of output 'out1' for 'forward'.`		
`Incorrect type of 'dLdX' for 'backward'.`	The types of the derivatives `dLdX1,…,dLdXn` of the optional `backward` function must be consistent with the inputs `X1,…,Xn`.	Return the derivatives `dLdX1,…,dLdXn` with the same type as the inputs `X1,…,Xn`.
`Incorrect type of the derivative of the loss with respect to the input 'in1' for 'backward'.`		
`Incorrect type of the derivative of loss with respect to 'W' for 'backward'.`	The type of the derivative of the loss of the learnable parameters must be consistent with the corresponding learnable parameters.	For each learnable parameter, return the derivative with the same type as the corresponding learnable parameter.

Tip If the layer forward functions support `dlarray` objects, then the software automatically determines the backward function and you do not need to specify the `backward` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Output Layers

The tests `forwardLossIsConsistentInType` and `backwardLossIsConsistentInType` check that the layer functions output variables of the correct data type. The tests check that the layers return consistent data types when given inputs of the data types `single`, `double`, and `gpuArray` with the underlying types `single` or `double`.

Test Diagnostic	Description	Possible Solution
`Incorrect type of 'loss' for 'forwardLoss'.`	The type of the output `loss` of the `forwardLoss` function must be consistent with the input `Y`.	Return `loss` with the same type as the input `Y`.
`Incorrect type of the derivative of loss 'dLdY' for 'backwardLoss'.`	The type of the output `dLdY` of the optional `backwardLoss` function must be consistent with the input `Y`.	Return `dLdY` with the same type as the input `Y`.

Tip If the `forwardLoss` function supports `dlarray` objects, then the software automatically determines the backward loss function and you do not need to specify the `backwardLoss` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Correct Gradients

The test `gradientsAreNumericallyCorrect` checks that the gradients computed by the layer functions are numerically correct. The test `backwardPropagationDoesNotError` checks that the derivatives can be computed using automatic differentiation.

Intermediate Layers

When the optional `backward` function is not specified, the test `backwardPropagationDoesNotError` checks that the derivatives can be computed using automatic differentiation. When the optional `backward` function is specified, the test `gradientsAreNumericallyCorrect` tests that the gradients computed in `backward` are numerically correct.

Test Diagnostic	Description	Possible Solution
`Expected a dlarray with no dimension labels, but instead found labels.`	When the optional `backward` function is not specified, the layer forward functions must output `dlarray` objects without dimension labels.	Ensure that any `dlarray` objects created in the layer forward functions do not contain dimension labels.
`Unable to backward propagate through the layer. Check that the 'forward' function fully supports automatic differentiation. Alternatively, implement the 'backward' function manually.`	One or more of the following: • When the optional `backward` function is not specified, the layer forward functions do not support `dlarray` objects. • When the optional `backward` function is not specified, the tracing of the input `dlarray` objects in the forward functions have been broken. For example, by using the `extractdata` function.	Check that the forward functions support `dlarray` objects. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. Check that the derivatives of the input `dlarray` objects can be traced. To learn more about the derivative trace of `dlarray` objects, see “Derivative Trace” on page 18-206. Alternatively, define a custom backward function by creating a function named `backward`. To learn more, see .
`Unable to backward propagate through the layer. Check that the 'predict' function fully supports automatic differentiation. Alternatively, implement the 'backward' function manually.`		
`The derivative 'dLdX' for 'backward' is inconsistent with the numerical gradient.`	One or more of the following: • When the optional `backward` function is specified, the derivative is incorrectly computed • The forward functions are non-differentiable at some input points • Error tolerance is too small	If the layer forward functions support `dlarray` objects, then the software automatically determines the backward function and you can omit the backward function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.
`The derivative of the loss with respect to the input 'in1' for 'backward' is inconsistent with the numerical gradient.`		

Test Diagnostic	Description	Possible Solution
`The derivative of loss with respect to 'W' for 'backward' is inconsistent with the numerical gradient.`		Check that the derivatives in `backward` are correctly computed. If the derivatives are correctly computed, then in the `Framework Diagnostic` section, manually check the absolute and relative error between the actual and expected values of the derivative. If the absolute and relative errors are within an acceptable margin of the tolerance, then you can ignore this test diagnostic.

Tip If the layer forward functions support `dlarray` objects, then the software automatically determines the backward function and you do not need to specify the `backward` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Output Layers

When the optional `backwardLoss` function is not specified, the test `backwardPropagationDoesNotError` checks that the derivatives can be computed using automatic differentiation. When the optional `backwardLoss` function is specified, the test `gradientsAreNumericallyCorrect` tests that the gradients computed in `backwardLoss` are numerically correct.

Test Diagnostic	Description	Possible Solution
`Expected a dlarray with no dimension labels, but instead found labels`	When the optional `backwardLoss` function is not specified, the `forwardLoss` function must output `dlarray` objects without dimension labels.	Ensure that any `dlarray` objects created in the `forwardLoss` function does not contain dimension labels.

Test Diagnostic	Description	Possible Solution
`Unable to backward propagate through the layer. Check that the 'forwardLoss' function fully supports automatic differentiation. Alternatively, implement the 'backwardLoss' function manually`	One or more of the following: • When the optional `backwardLoss` function is not specified, the layer `forwardLoss` function does not support `dlarray` objects. • When the optional `backwardLoss` function is not specified, the tracing of the input `dlarray` objects in the `forwardLoss` function has been broken. For example, by using the `extractdata` function.	Check that the `forwardLoss` function supports `dlarray` objects. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423. Check that the derivatives of the input `dlarray` objects can be traced. To learn more about the derivative trace of `dlarray` objects, see “Derivative Trace” on page 18-206. Alternatively, define a custom backward loss function by creating a function named `backwardLoss`. To learn more, see .
`The derivative 'dLdY' for 'backwardLoss' is inconsistent with the numerical gradient.`	One or more of the following: • The derivative with respect to the predictions `Y` is incorrectly computed • Function is non-differentiable at some input points • Error tolerance is too small	Check that the derivatives in `backwardLoss` are correctly computed. If the derivatives are correctly computed, then in the `Framework Diagnostic` section, manually check the absolute and relative error between the actual and expected values of the derivative. If the absolute and relative errors are within an acceptable margin of the tolerance, then you can ignore this test diagnostic.

Tip If the `forwardLoss` function supports `dlarray` objects, then the software automatically determines the backward loss function and you do not need to specify the `backwardLoss` function. For a list of functions that support `dlarray` objects, see “List of Functions with dlarray Support” on page 18-423.

Valid States

For layers with state properties, the test `predictReturnsValidStates` checks that the predict function returns valid states. When `forward` is specified, the test `forwardReturnsValidStates`

checks that the forward function returns valid states. The test `resetStateDoesNotError` checks that the `resetState` function returns a layer with valid state properties.

Test Diagnostic	Description	Possible Solution
`Error using 'predict' in Layer. 'State' must be real-values numeric array or unformatted dlarray object.`	State outputs must be real-valued numeric arrays or unformatted `dlarray` objects.	Ensure that the states identified in the `Framework Diagnostic` are real-valued numeric arrays or unformatted `dlarray` objects.
`Error using 'resetState' in Layer. 'State' must be real-values numeric array or unformatted dlarray object`	State properties of returned layer must be real-valued numeric arrays or unformatted `dlarray` objects.	

Code Generation Compatibility

If you set the `CheckCodegenCompatibility` option to `1` (true), then the `checkLayer` function checks the layer for code generation compatibility.

The test `codegenPragmaDefinedInClassDef` checks that the layer definition contains the code generation pragma `%#codegen`. The test `checkForSupportedLayerPropertiesForCodegen` checks that the layer properties support code generation. The test `predictIsValidForCodegeneration` checks that the outputs of `predict` are consistent in dimension and batch size.

Code generation supports intermediate layers with 2-D image or feature input only. Code generation does not support layers with state properties (properties with attribute `State`).

The `checkLayer` function does not check that functions used by the layer are compatible with code generation. To check that functions used by the custom layer also support code generation, first use the **Code Generation Readiness** app. For more information, see “Check Code by Using the Code Generation Readiness Tool” (MATLAB Coder).

Test Diagnostic	Description	Possible Solution
`Specify '%#codegen' in the class definition of custom layer`	The layer definition does not include the pragma `"%#codegen"` for code generation.	Add the `%#codegen` directive (or pragma) to your layer definition to indicate that you intend to generate code for this layer. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that result in errors during code generation.

Test Diagnostic	Description	Possible Solution
`Nonscalar layer properties must be type single or double or character array for custom layer`	The layer contains non-scalar properties of type other than single, double, or character array.	Convert non-scalar properties to use a representation of type single, double, or character array. For example, convert a categorical array to an array of integers of type `double` representing the categories.
`Scalar layer properties must be numeric, logical, or string for custom layer`	The layer contains scalar properties of type other than numeric, logical, or string.	Convert scalar properties to use a numeric representation, or a representation of type logical or string. For example, convert a categorical scalar to an integer of type `double` representing the category.
`For code generation, 'Z' must have the same number of dimensions as the layer input.`	The number of dimensions of the output `Z` of `predict` does not match the number of dimensions of the layer inputs.	In the `predict` function, return the outputs with the same number of dimensions as the layer inputs.
`For code generation, 'Z' must have the same batch size as the layer input.`	The size of the batch size of the output `Z` of `predict` does not match the size of the batch size of the layer inputs.	In the `predict` function, return the outputs with the batch size as the layer inputs.

See Also

`checkLayer` | `analyzeNetwork`

More About

- “Define Custom Deep Learning Intermediate Layers” on page 18-16
- “Define Custom Deep Learning Output Layers” on page 18-29
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 18-35
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 18-48
- “Define Custom Deep Learning Layer with Formatted Inputs” on page 18-61
- “Define Custom Recurrent Deep Learning Layer” on page 18-75
- “Define Custom Deep Learning Layer for Code Generation” on page 18-142
- “Define Custom Classification Output Layer” on page 18-91
- “Define Custom Regression Output Layer” on page 18-99
- “Define Nested Deep Learning Layer” on page 18-120

Specify Custom Weight Initialization Function

This example shows how to create a custom He weight initialization function for convolution layers followed by leaky ReLU layers.

The He initializer for convolution layers followed by leaky ReLU layers samples from a normal distribution with zero mean and variance $\sigma^2 = \frac{2}{(1+a^2)n}$, where a is the scale of the leaky ReLU layer that follows the convolution layer and `n = FilterSize(1) * FilterSize(2) * NumChannels`.

For learnable layers, when setting the options `'WeightsInititializer'`, `'InputWeightsInitializer'`, or `'RecurrentWeightsInitializer'` to `'he'`, the software uses $a=0$. To set a to different value, create a custom function to use as a weights initializer.

Load Data

Load the digit sample data as an image datastore. The `imageDatastore` function automatically labels the images based on folder names.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label. `splitEachLabel` splits the datastore into two new datastores for training and validation.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');
```

Define Network Architecture

Define the convolutional neural network architecture:

- Image input layer size of `[28 28 1]`, the size of the input images
- Three 2-D convolution layers with filter size 3 and with 8, 16, and 32 filters respectively
- A leaky ReLU layer following each convolutional layer
- Fully connected layer of size 10, the number of classes
- Softmax layer
- Classification layer

For each of the convolutional layers, set the weights initializer to the `leakyHe` function. The `leakyHe` function, listed at the end of the example, takes the input `sz` (the size of the layer weights) and returns an array of weights given by the He Initializer for convolution layers followed by a leaky ReLU layer.

```
inputSize = [28 28 1];
numClasses = 10;

layers = [
    imageInputLayer(inputSize)
    convolution2dLayer(3,8,'WeightsInitializer',@leakyHe)
```

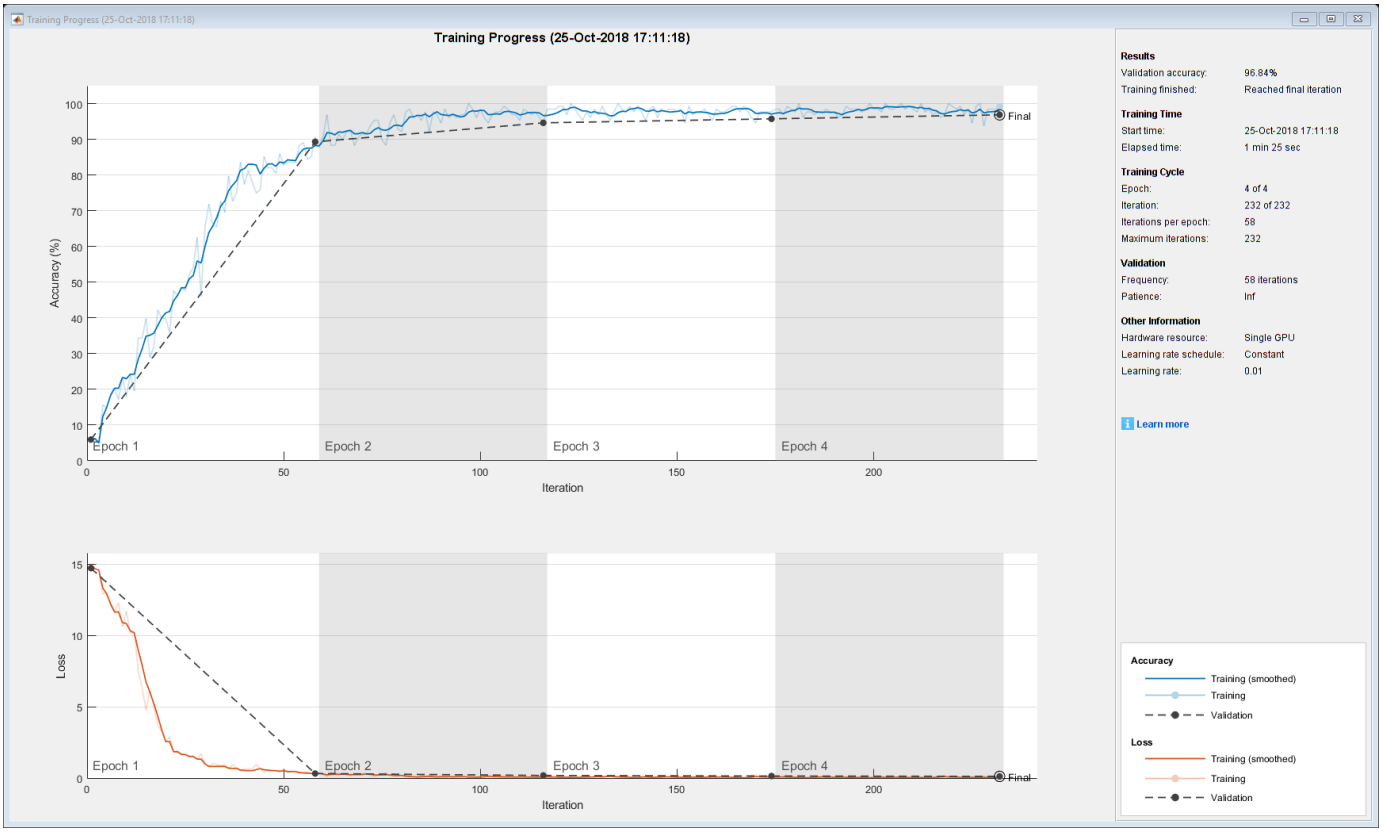
```
    leakyReluLayer
    convolution2dLayer(3,16,'WeightsInitializer',@leakyHe)
    leakyReluLayer
    convolution2dLayer(3,32,'WeightsInitializer',@leakyHe)
    leakyReluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Train Network

Specify the training options and train the network. Train for four epochs. To prevent the gradients from exploding, set the gradient threshold to 2. Validate the network once per epoch. View the training progress plot.

By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

```
maxEpochs = 4;
miniBatchSize = 128;
numObservations = numel(imdsTrain.Files);
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('sgdm', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Verbose',false, ...
    'Plots','training-progress');

[netDefault,infoDefault] = trainNetwork(imdsTrain,layers,options);
```



Test Network

Classify the validation data and calculate the classification accuracy.

```
YPred = classify(netDefault,imdsValidation);
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 0.9684
```

Specify Additional Options

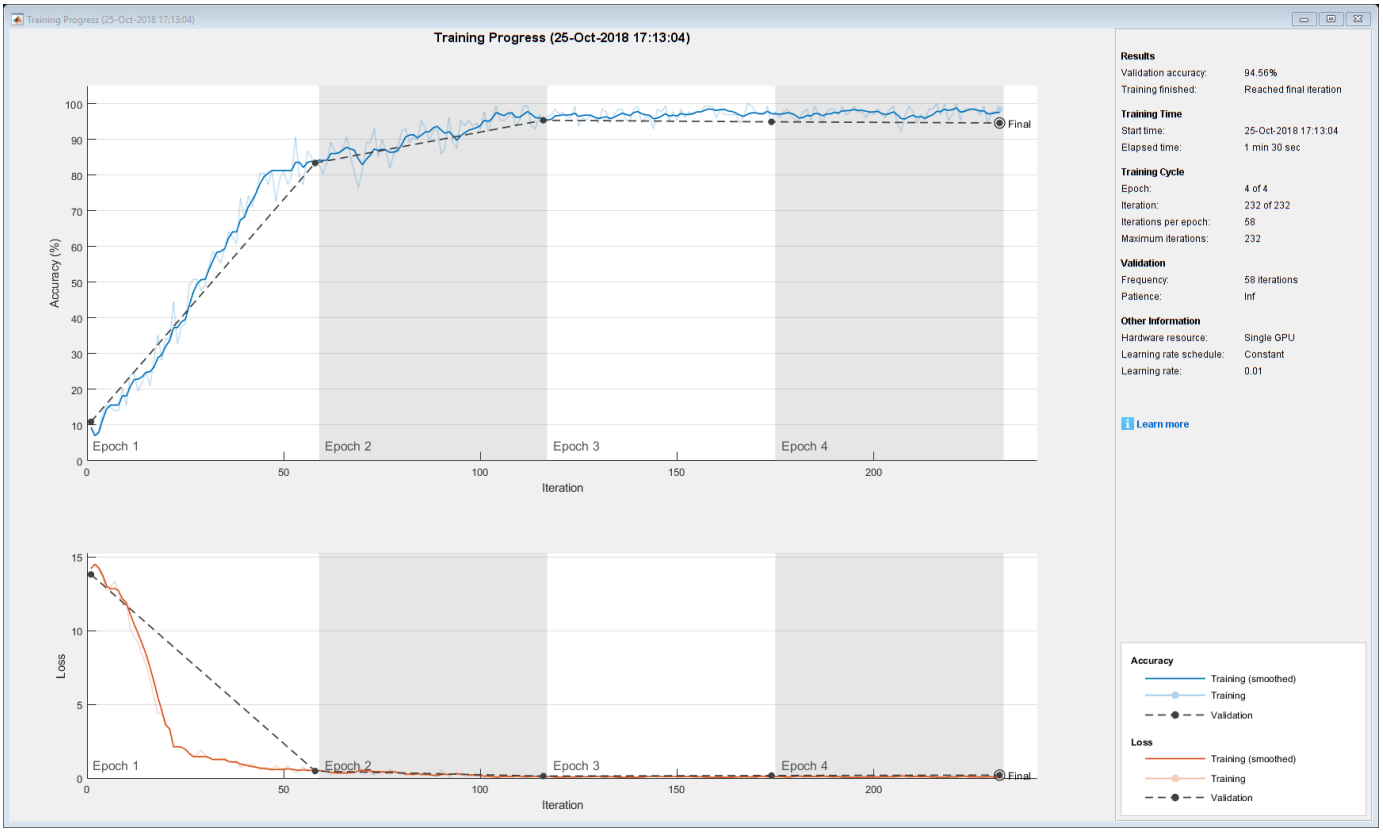
The `leakyHe` function accepts the optional input argument `scale`. To input extra variables into the custom weight initialization function, specify the function as an anonymous function that accepts a single input `sz`. To do this, replace instances of `@leakyHe` with `@(sz) leakyHe(sz,scale)`. Here, the anonymous function accepts the single input argument `sz` only and calls the `leakyHe` function with the specified `scale` input argument.

Create and train the same network as before with the following changes:

- For the leaky ReLU layers, specify a scale multiplier of 0.01.
- Initialize the weights of the convolutional layers with the `leakyHe` function and also specify the scale multiplier.

```
scale = 0.01;

layers = [
    imageInputLayer(inputSize)
```

```
    convolution2dLayer(3,8,'WeightsInitializer',@(sz) leakyHe(sz,scale))
    leakyReluLayer(scale)
    convolution2dLayer(3,16,'WeightsInitializer',@(sz) leakyHe(sz,scale))
    leakyReluLayer(scale)
    convolution2dLayer(3,32,'WeightsInitializer',@(sz) leakyHe(sz,scale))
    leakyReluLayer(scale)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

[netCustom,infoCustom] = trainNetwork(imdsTrain,layers,options);
```



Classify the validation data and calculate the classification accuracy.

```
YPred = classify(netCustom,imdsValidation);
YValidation = imdsValidation.Labels;
accuracy = mean(YPred == YValidation)
```

```
accuracy = 0.9456
```

Compare Results

Extract the validation accuracy from the information structs output from the `trainNetwork` function.

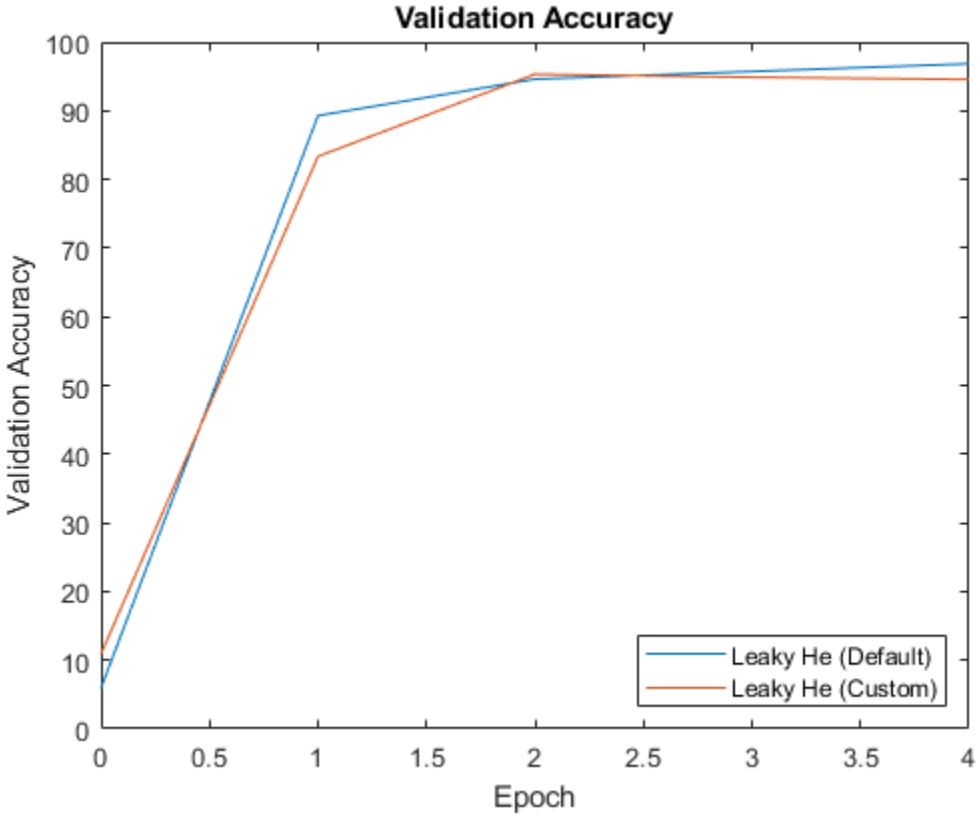
```
validationAccuracy = [
    infoDefault.ValidationAccuracy;
    infoCustom.ValidationAccuracy];
```

The vectors of validation accuracy contain `NaN` for iterations that the validation accuracy was not computed. Remove the `NaN` values.

```
idx = all(isnan(validationAccuracy));
validationAccuracy(:,idx) = [];
```

For each of the networks, plot the epoch numbers against the validation accuracy.

```
figure
epochs = 0:maxEpochs;
plot(epochs,validationAccuracy)
title("Validation Accuracy")
xlabel("Epoch")
ylabel("Validation Accuracy")
legend(["Leaky He (Default)" "Leaky He (Custom)"],'Location','southeast')
```



Custom Weight Initialization Function

The `leakyHe` function takes the input `sz` (the size of the layer weights) and returns an array of weights given by the He Initializer for convolution layers followed by a leaky ReLU layer. The function also accepts the optional input argument `scale` which specifies the scale multiplier for the leaky ReLU layer.

```
function weights = leakyHe(sz,scale)

% If not specified, then use default scale = 0.1
if nargin < 2
    scale = 0.1;
```

```
end

filterSize = [sz(1) sz(2)];
numChannels = sz(3);
numIn = filterSize(1) * filterSize(2) * numChannels;

varWeights = 2 / ((1 + scale^2) * numIn);
weights = randn(sz) * sqrt(varWeights);

end
```

Bibliography

1 He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

See Also

`trainNetwork` | `trainingOptions`

Related Examples

- “Compare Layer Weight Initializers” on page 18-181
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Deep Learning in MATLAB” on page 1-2

Compare Layer Weight Initializers

This example shows how to train deep learning networks with different weight initializers.

When training a deep learning network, the initialization of layer weights and biases can have a big impact on how well the network trains. The choice of initializer has a bigger impact on networks without batch normalization layers.

Depending on the type of layer, you can change the weights and bias initialization using the `'WeightsInitializer'`, `'InputWeightsInitializer'`, `'RecurrentWeightsInitializer'`, and `'BiasInitializer'` options.

This example shows the effect of using these three different weight initializers when training an LSTM network:

1. **Glorot Initializer** – Initialize the input weights with the Glorot initializer. [1]
2. **He Initializer** – Initialize the input weights with the He initializer. [2]
3. **Narrow-Normal Initializer** – Initialize the input weights by independently sampling from a normal distribution with zero mean and standard deviation 0.01.

Load Data

Load the Japanese Vowels data set. `XTrain` is a cell array containing 270 sequences of varying length with a feature dimension of 12. `Y` is a categorical vector of labels 1,2,...,9. The entries in `XTrain` are matrices with 12 rows (one row for each feature) and a varying number of columns (one column for each time step).

```
[XTrain,YTrain] = japaneseVowelsTrainData;
[XValidation,YValidation] = japaneseVowelsTestData;
```

Specify Network Architecture

Specify the network architecture. For each initializer, use the same network architecture.

Specify the input size as 12 (the number of features of the input data). Specify an LSTM layer with 100 hidden units and to output the last element of the sequence. Finally, specify nine classes by including a fully connected layer of size 9, followed by a softmax layer and a classification layer.

```
numFeatures = 12;
numHiddenUnits = 100;
numClasses = 9;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer]
```

```
layers =
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 12 dimensions
     2   ''   LSTM                    LSTM with 100 hidden units
     3   ''   Fully Connected         9 fully connected layer
```

```
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

Training Options

Specify the training options. For each initializer, use the same training options to train the network.

```
maxEpochs = 30;
miniBatchSize = 27;
numObservations = numel(XTrain);
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'ValidationData',{XValidation,YValidation}, ...
    'ValidationFrequency',numIterationsPerEpoch, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

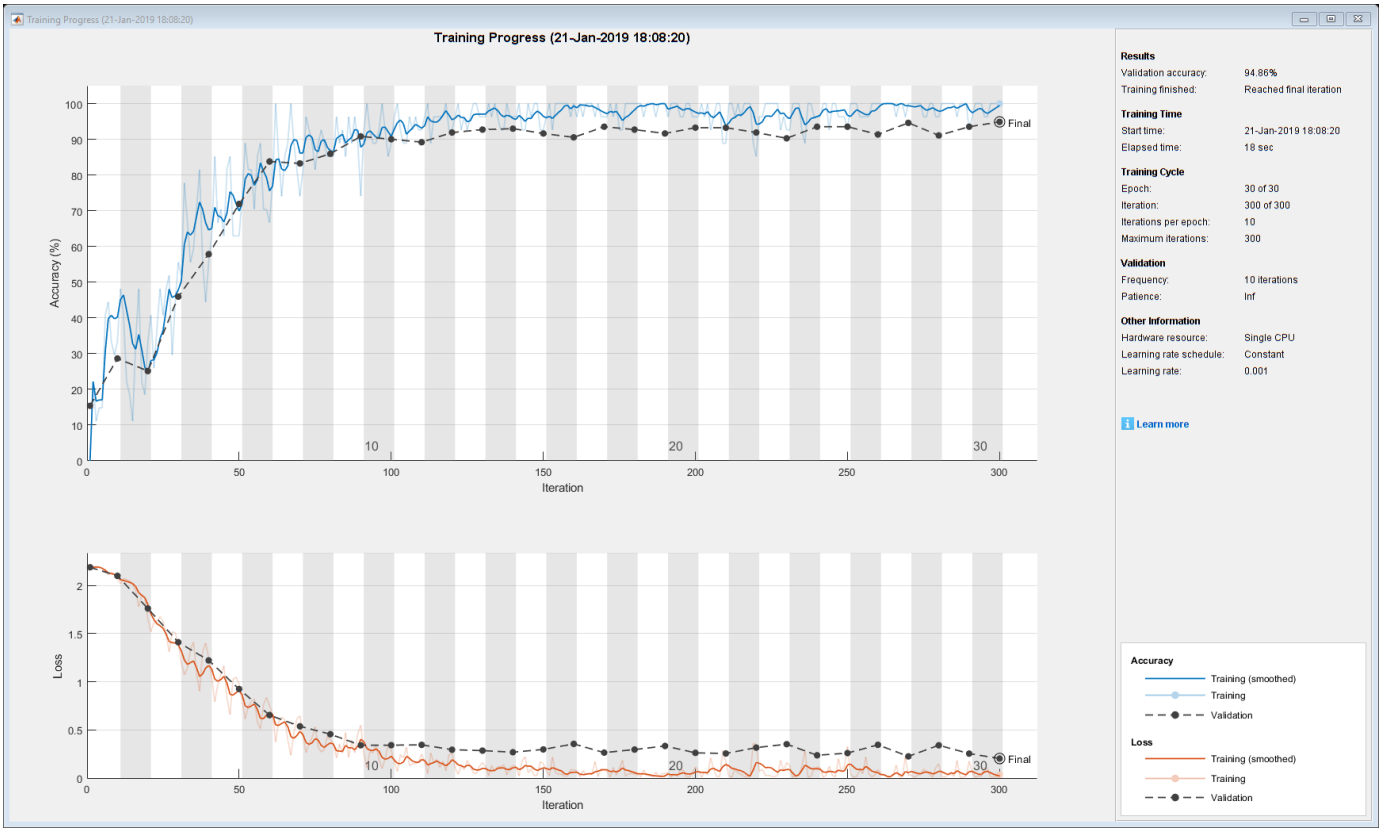
Glorot Initializer

Specify the network architecture listed earlier in the example and set the input weights initializer of the LSTM layer and the weights initializer of the fully connected layer to `'glorot'`.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last','InputWeightsInitializer','glorot')
    fullyConnectedLayer(numClasses,'WeightsInitializer','glorot')
    softmaxLayer
    classificationLayer];
```

Train the network using the layers with the Glorot weights initializers.

```
[netGlorot,infoGlorot] = trainNetwork(XTrain,YTrain,layers,options);
```



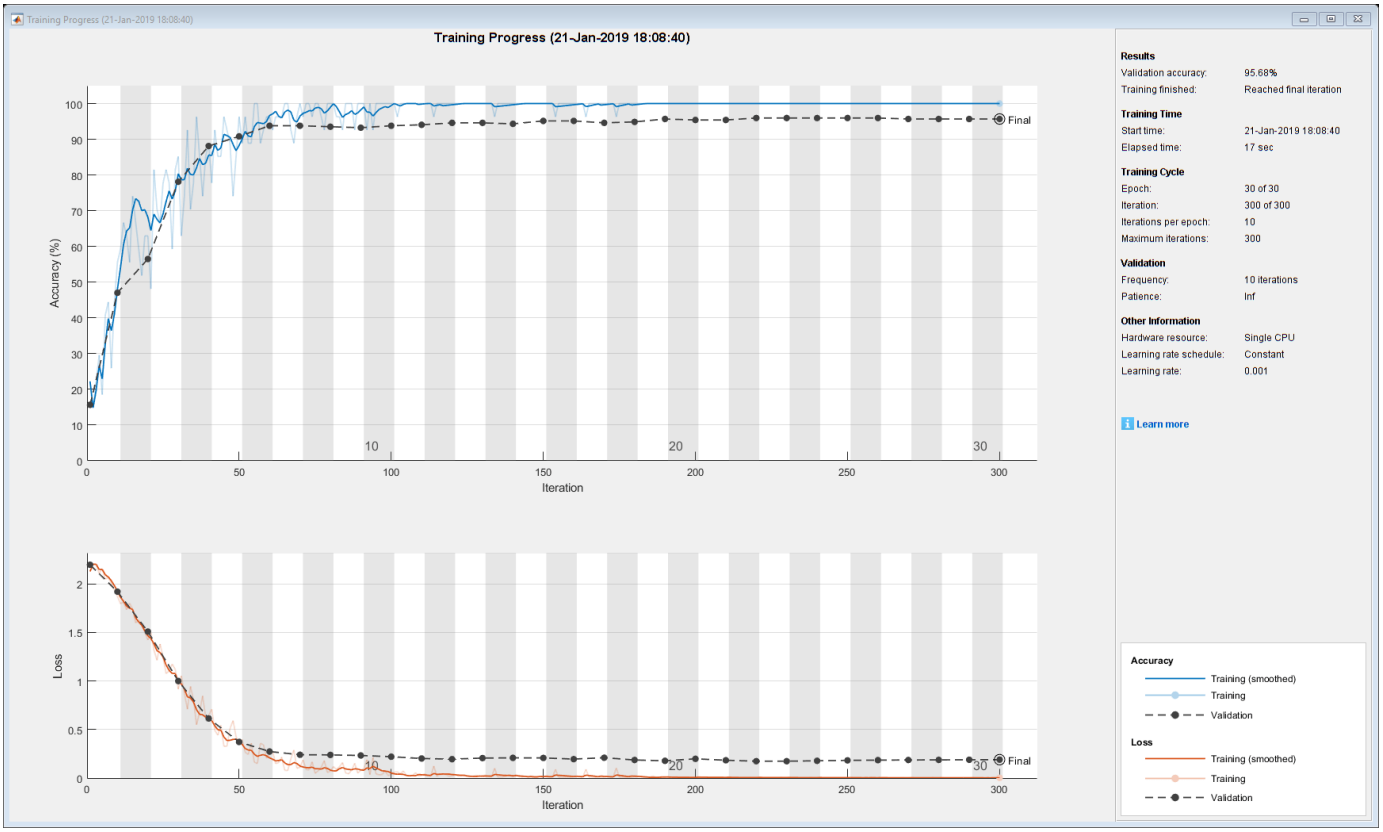
He Initializer

Specify the network architecture listed earlier in the example and set the input weights initializer of the LSTM layer and the weights initializer of the fully connected layer to `'he'`.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last','InputWeightsInitializer','he')
    fullyConnectedLayer(numClasses,'WeightsInitializer','he')
    softmaxLayer
    classificationLayer];
```

Train the network using the layers with the He weights initializers.

```
[netHe,infoHe] = trainNetwork(XTrain,YTrain,layers,options);
```



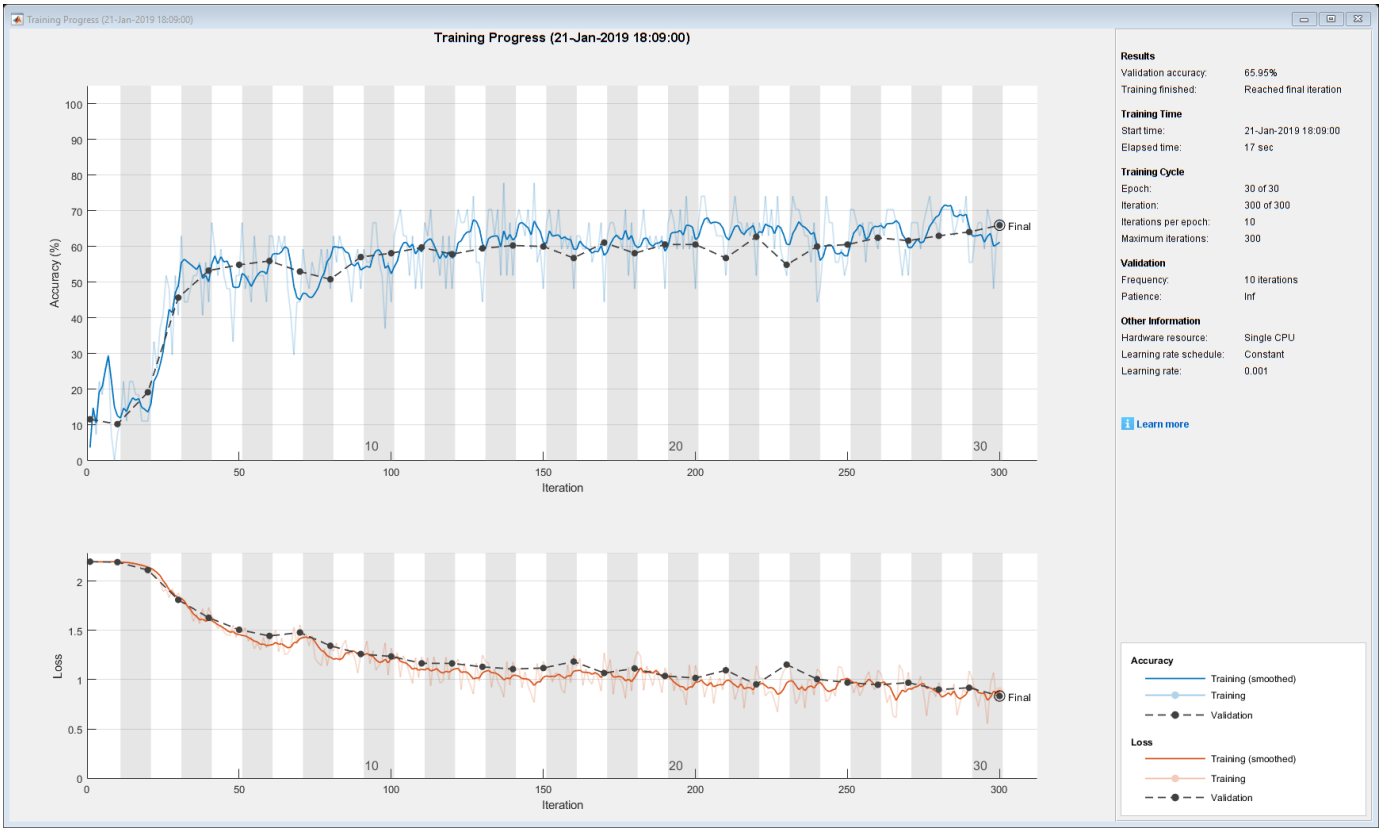
Narrow-Normal Initializer

Specify the network architecture listed earlier in the example and set the input weights initializer of the LSTM layer and the weights initializer of the fully connected layer to `'narrow-normal'`.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last','InputWeightsInitializer','narrow-normal')
    fullyConnectedLayer(numClasses,'WeightsInitializer','narrow-normal')
    softmaxLayer
    classificationLayer];
```

Train the network using the layers with the narrow-normal weights initializers.

```
[netNarrowNormal,infoNarrowNormal] = trainNetwork(XTrain,YTrain,layers,options);
```



Plot Results

Extract the validation accuracy from the information structs output from the `trainNetwork` function.

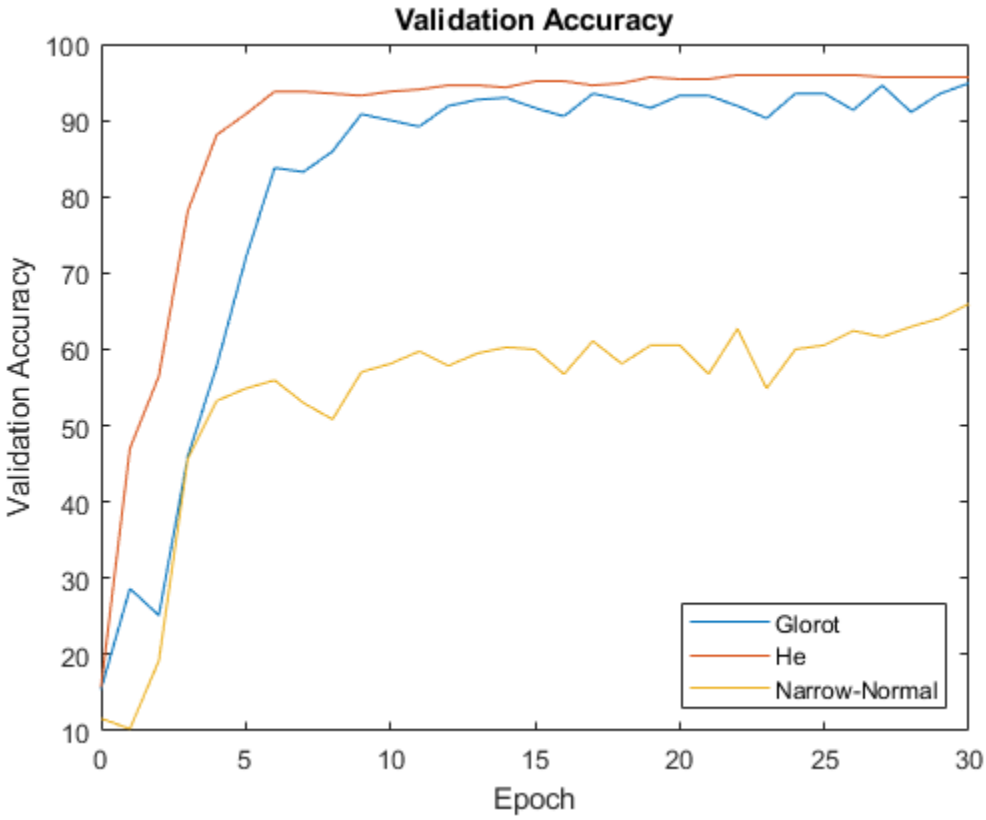
```
validationAccuracy = [
    infoGlorot.ValidationAccuracy;
    infoHe.ValidationAccuracy;
    infoNarrowNormal.ValidationAccuracy];
```

The vectors of validation accuracy contain `NaN` for iterations that the validation accuracy was not computed. Remove the `NaN` values.

```
idx = all(isnan(validationAccuracy));
validationAccuracy(:,idx) = [];
```

For each of the initializers, plot the epoch numbers against the validation accuracy.

```
figure
epochs = 0:maxEpochs;
plot(epochs,validationAccuracy)
title("Validation Accuracy")
xlabel("Epoch")
ylabel("Validation Accuracy")
legend(["Glorot" "He" "Narrow-Normal"],'Location','southeast')
```



This plot shows the overall effect of the different initializers and how quickly the training converges for each one.

Bibliography

1 Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249-256. 2010.

2 He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

See Also

`trainNetwork` | `trainingOptions`

Related Examples

- “Specify Custom Weight Initialization Function” on page 18-175
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Deep Learning in MATLAB” on page 1-2

Assemble Network from Pretrained Keras Layers

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with custom layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `'digitsDAGnetwithnoise.h5'` classifies images of digits.

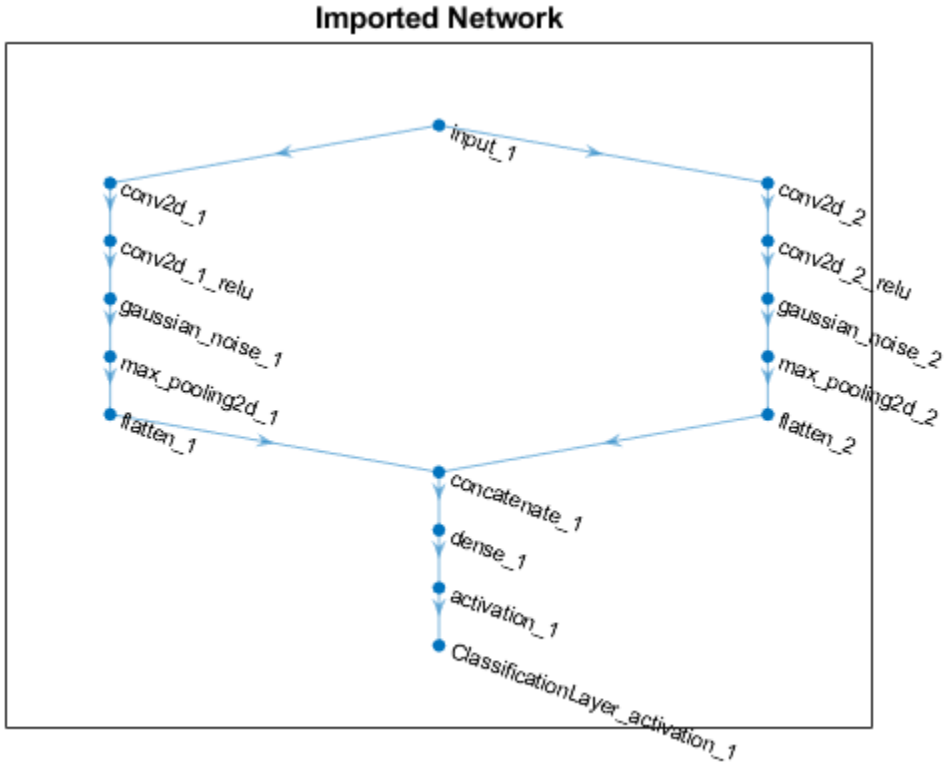
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Plot the layer graph using `plot`.

```
figure
plot(lgraph)
title("Imported Network")
```



Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using `findPlaceholderLayers`.

```
placeholderLayers = findPlaceholderLayers(lgraph)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'gaussian_noise_1'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
     2   'gaussian_noise_2'   PLACEHOLDER LAYER   Placeholder for 'GaussianNoise' Keras layer
```

Display the Keras configurations of these layers.

```
placeholderLayers.KerasConfiguration
```

```
ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_1'
       stddev: 1.5000


ans = struct with fields:
    trainable: 1
         name: 'gaussian_noise_2'
       stddev: 0.7000
```

Define a custom Gaussian noise layer. To create this layer, save the file `gaussianNoiseLayer.m` in the current folder. Then, create two Gaussian noise layers with the same configurations as the imported Keras layers.

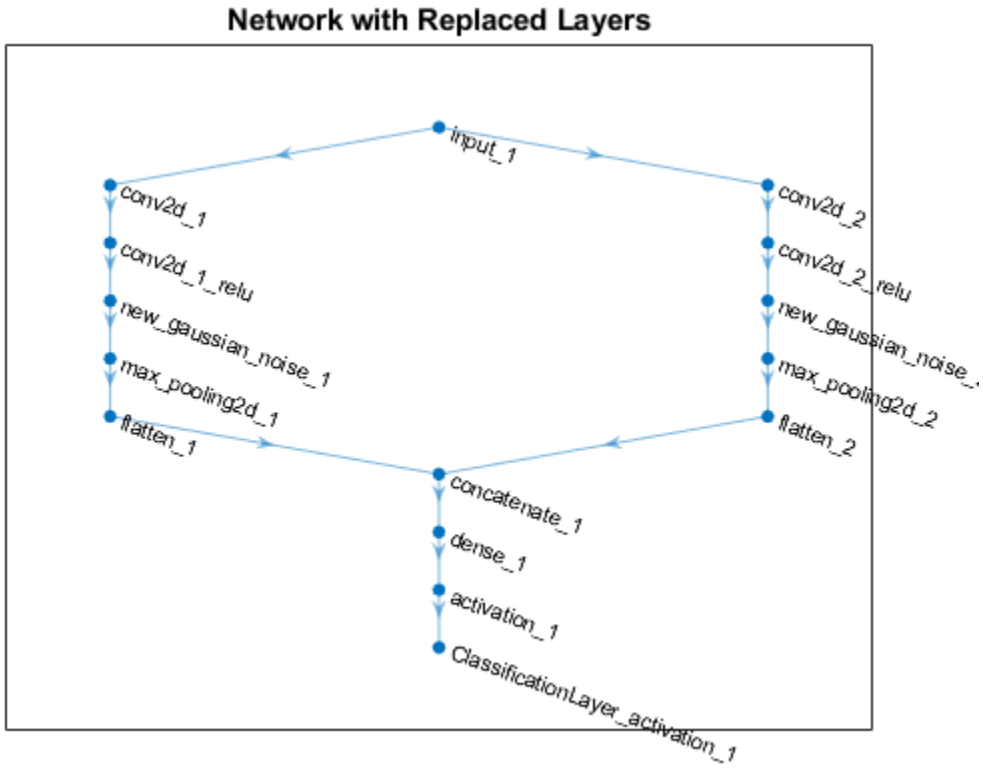
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

Replace the placeholder layers with the custom layers using `replaceLayer`.

```
lgraph = replaceLayer(lgraph,'gaussian_noise_1',gnLayer1);
lgraph = replaceLayer(lgraph,'gaussian_noise_2',gnLayer2);
```

Plot the updated layer graph using `plot`.

```
figure
plot(lgraph)
title("Network with Replaced Layers")
```



Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  15x1 Layer array with layers:

     1   'input_1'                          Image Input             28x28x1 images
     2   'conv2d_1'                         Convolution             20 7x7x1 convolutions with
     3   'conv2d_1_relu'                    ReLU                    ReLU
     4   'conv2d_2'                         Convolution             20 3x3x1 convolutions with
     5   'conv2d_2_relu'                    ReLU                    ReLU
     6   'new_gaussian_noise_1'             Gaussian Noise          Gaussian noise with standar
     7   'new_gaussian_noise_2'             Gaussian Noise          Gaussian noise with standar
     8   'max_pooling2d_1'                  Max Pooling             2x2 max pooling with stride
     9   'max_pooling2d_2'                  Max Pooling             2x2 max pooling with stride
    10   'flatten_1'                        Keras Flatten           Flatten activations into 1
    11   'flatten_2'                        Keras Flatten           Flatten activations into 1
    12   'concatenate_1'                    Depth concatenation     Depth concatenation of 2 in
    13   'dense_1'                          Fully Connected         10 fully connected layer
    14   'activation_1'                     Softmax                 softmax
    15   'ClassificationLayer_activation_1' Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_activation_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `'auto'`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_activation_1'
         Classes: [0    1    2    3    4    5    6    7    8    9]
    ClassWeights: 'none'
      OutputSize: 10

   Hyperparameters
    LossFunction: 'crossentropyex'
```

```
lgraph = replaceLayer(lgraph,'ClassificationLayer_activation_1',cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [15x1 nnet.cnn.layer.Layer]
    Connections: [15x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

See Also

`importKerasNetwork` | `assembleNetwork` | `replaceLayer` | `importKerasLayers` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `findPlaceholderLayers`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Define Custom Deep Learning Layers” on page 18-9

Replace Unsupported Keras Layer with Function Layer

This example shows how to import the layers from a pretrained Keras network, replace the unsupported layers with function layers, and assemble the layers into a network ready for prediction.

Import Keras Network

Import the layers from a Keras network model. The network in `"digitsNet.h5"` classifies images of digits.

```
filename = "digitsNet.h5";
layers = importKerasLayers(filename,ImportWeights=true)
```

```
Warning: Unable to import layer. Keras layer 'Activation' with the specified settings is not supp

Warning: Unable to import layer. Keras layer 'Activation' with the specified settings is not supp

Warning: Unable to import some Keras layers, because they are not supported by the Deep Learning

layers =
  13x1 Layer array with layers:

     1   'ImageInputLayer'               Image Input             28x28x1 images
     2   'conv2d'                        Convolution             8 3x3x1 convolutions with stride
     3   'conv2d_softsign'               PLACEHOLDER LAYER       Placeholder for 'Activation' Ker
     4   'max_pooling2d'                 Max Pooling             2x2 max pooling with stride [2
     5   'conv2d_1'                      Convolution             16 3x3x8 convolutions with strid
     6   'conv2d_1_softsign'             PLACEHOLDER LAYER       Placeholder for 'Activation' Ker
     7   'max_pooling2d_1'               Max Pooling             2x2 max pooling with stride [2
     8   'flatten'                       Keras Flatten           Flatten activations into 1-D ass
     9   'dense'                         Fully Connected         100 fully connected layer
    10   'dense_relu'                    ReLU                    ReLU
    11   'dense_1'                       Fully Connected         10 fully connected layer
    12   'dense_1_softmax'               Softmax                 softmax
    13   'ClassificationLayer_dense_1'   Classification Output   crossentropyex
```

The Keras network contains some layers that are not supported by Deep Learning Toolbox. The `importKerasLayers` function displays a warning and replaces the unsupported layers with placeholder layers.

Replace Placeholder Layers

To replace the placeholder layers, first identify the names of the layers to replace. Find the placeholder layers using the `findPlaceholderLayers` function.

```
placeholderLayers = findPlaceholderLayers(layers)
```

```
placeholderLayers =
  2x1 PlaceholderLayer array with layers:

     1   'conv2d_softsign'     PLACEHOLDER LAYER   Placeholder for 'Activation' Keras layer
     2   'conv2d_1_softsign'   PLACEHOLDER LAYER   Placeholder for 'Activation' Keras layer
```

Replace the placeholder layers with function layers with function specified by the `softsign` function, listed at the end of the example.

Create a function layer with function specified by the `softsign` function, attached to this example as a supporting file. To access this function, open this example as a live script. Set the layer description to `"softsign"`.

```
layer = functionLayer(@softsign,Description="softsign");
```

Replace the layers using the `replaceLayer` function. To use the `replaceLayer` function, first convert the layer array to a layer graph.

```
lgraph = layerGraph(layers);
lgraph = replaceLayer(lgraph,"conv2d_softsign",layer);
lgraph = replaceLayer(lgraph,"conv2d_1_softsign",layer);
```

Specify Class Names

If the imported classification layer does not contain the classes, then you must specify these before prediction. If you do not specify the classes, then the software automatically sets the classes to `1`, `2`, ..., `N`, where `N` is the number of classes.

Find the index of the classification layer by viewing the `Layers` property of the layer graph.

```
lgraph.Layers
```

```
ans =
  13x1 Layer array with layers:

     1   'ImageInputLayer'              Image Input             28x28x1 images
     2   'conv2d'                       Convolution             8 3x3x1 convolutions with stride
     3   'layer'                        Function                softsign
     4   'max_pooling2d'                Max Pooling             2x2 max pooling with stride [2
     5   'conv2d_1'                     Convolution             16 3x3x8 convolutions with strid
     6   'layer_1'                      Function                softsign
     7   'max_pooling2d_1'              Max Pooling             2x2 max pooling with stride [2
     8   'flatten'                      Keras Flatten           Flatten activations into 1-D ass
     9   'dense'                        Fully Connected         100 fully connected layer
    10   'dense_relu'                   ReLU                    ReLU
    11   'dense_1'                      Fully Connected         10 fully connected layer
    12   'dense_1_softmax'              Softmax                 softmax
    13   'ClassificationLayer_dense_1'  Classification Output   crossentropyex
```

The classification layer has the name `'ClassificationLayer_dense_1'`. View the classification layer and check the `Classes` property.

```
cLayer = lgraph.Layers(end)
```

```
cLayer =
  ClassificationOutputLayer with properties:

            Name: 'ClassificationLayer_dense_1'
         Classes: 'auto'
    ClassWeights: 'none'
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Because the `Classes` property of the layer is `"auto"`, you must specify the classes manually. Set the classes to `0`, `1`, ..., `9`, and then replace the imported classification layer with the new one.

```
cLayer.Classes = string(0:9);
lgraph = replaceLayer(lgraph,"ClassificationLayer_dense_1",cLayer);
```

Assemble Network

Assemble the layer graph using `assembleNetwork`. The function returns a `DAGNetwork` object that is ready to use for prediction.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [13x1 nnet.cnn.layer.Layer]
    Connections: [12x2 table]
     InputNames: {'ImageInputLayer'}
    OutputNames: {'ClassificationLayer_dense_1'}
```

Test Network

Make predictions with the network using a test data set.

```
[XTest,YTest] = digitTest4DArrayData;
YPred = classify(net,XTest);
```

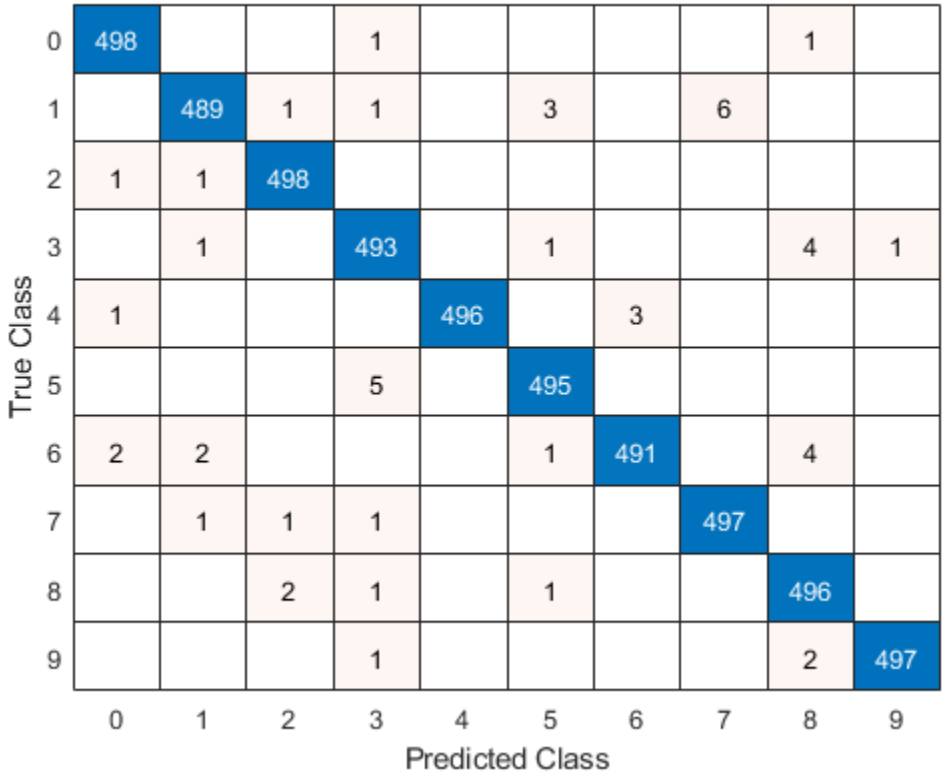
View the accuracy.

```
mean(YPred == YTest)
```

```
ans = 0.9900
```

Visualize the predictions in a confusion matrix.

```
confusionchart(YTest,YPred)
```



See Also

`importKerasNetwork` | `assembleNetwork` | `replaceLayer` | `importKerasLayers` | `trainNetwork` | `layerGraph` | `DAGNetwork` | `findPlaceholderLayers`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-8
- “Define Custom Deep Learning Layers” on page 18-9

Assemble Multiple-Output Network for Prediction

This example shows how to assemble a multiple output network for prediction.

Instead of using the `dlnetwork` object for prediction, you can assemble the network into a `DAGNetwork` ready for prediction using the `assembleNetwork` function. This lets you use the `predict` function with other data types such as datastores.

Load Model Function and Parameters

Load the model parameters from the MAT file `dlnetDigits.mat`. The MAT file contains a `dlnetwork` object that predicts both the scores for categorical labels and numeric angles of rotation of images of digits, and the corresponding class names.

```
s = load("dlnetDigits.mat");
dlnet = s.dlnet;
classNames = s.classNames;
```

Assemble Network for Prediction

Extract the layer graph from the `dlnetwork` object using the `layerGraph` function.

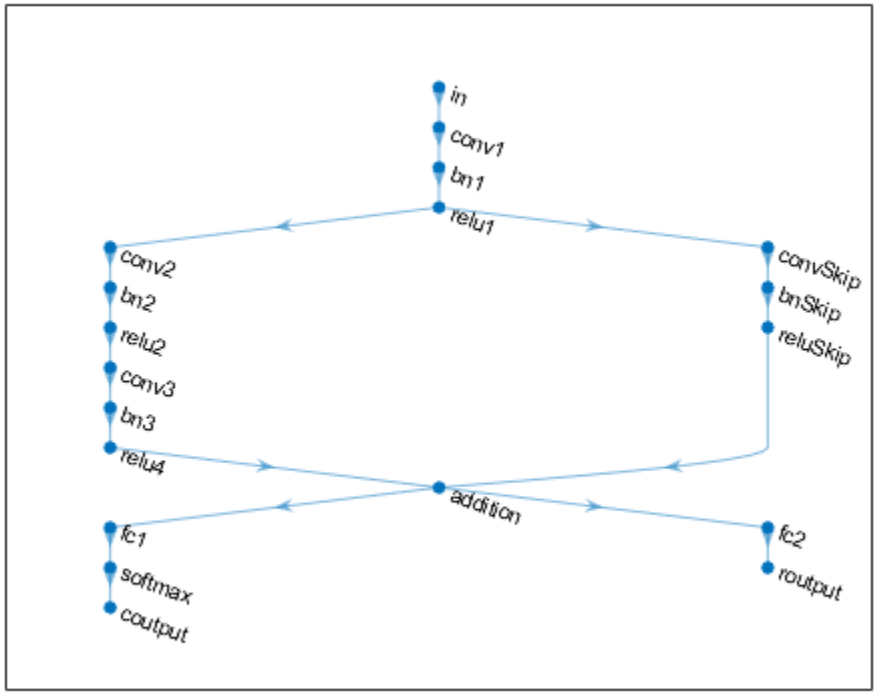
```
lgraph = layerGraph(dlnet);
```

The layer graph does not include output layers. Add a classification layer and a regression layer to the layer graph using the `addLayers` and `connectLayers` functions.

```
layers = classificationLayer('Classes',classNames,'Name','coutput');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'softmax','coutput');

layers = regressionLayer('Name','routput');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'fc2','routput');
```

View a plot of the network.

```
figure
plot(lgraph)
```



Assemble the network using the `assembleNetwork` function.

```
net = assembleNetwork(lgraph)
```

```
net =
  DAGNetwork with properties:

         Layers: [19x1 nnet.cnn.layer.Layer]
    Connections: [19x2 table]
     InputNames: {'in'}
    OutputNames: {'coutput'  'routput'}
```

Make Predictions on New Data

Load the test data.

```
[XTest,Y1Test,Y2Test] = digitTest4DArrayData;
```

To make predictions using the assembled network, use the `predict` function. To return categorical labels for the classification output, set the `'ReturnCategorical'` option to `true`.

```
[Y1Pred,Y2Pred] = predict(net,XTest,'ReturnCategorical',true);
```

Evaluate the classification accuracy.

```
accuracy = mean(Y1Pred==Y1Test)
```

```
accuracy = 0.9870
```

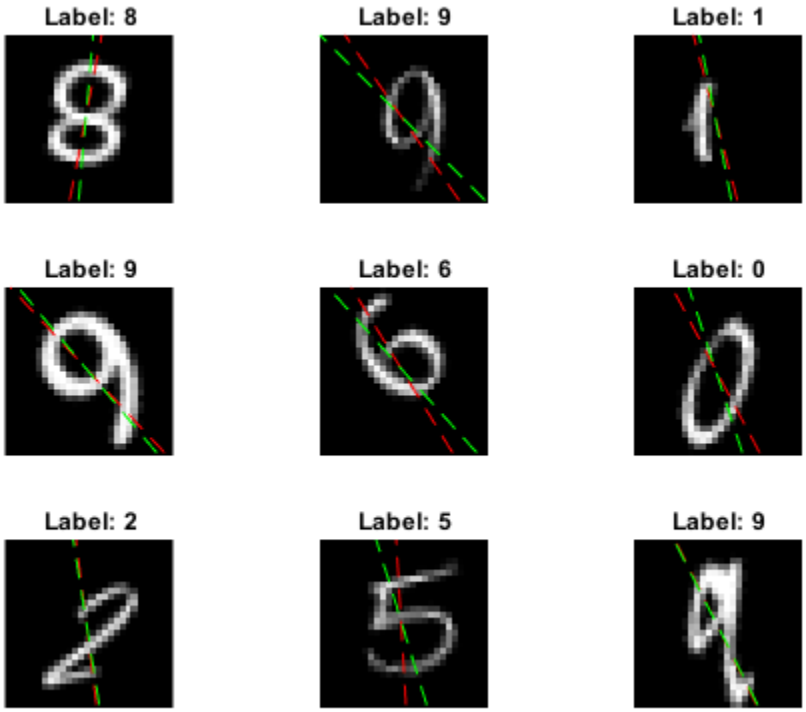
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean((Y2Pred - Y2Test).^2))
```

```
angleRMSE = single
    6.0091
```

View some of the images with their predictions. Display the predicted angles in red and the correct labels in green.

```
idx = randperm(size(XTest,4),9);
figure
for i = 1:9
    subplot(3,3,i)
    I = XTest(:,:,:,idx(i));
    imshow(I)
    hold on

    sz = size(I,1);
    offset = sz/2;

    thetaPred = Y2Pred(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    thetaValidation = Y2Test(idx(i));
    plot(offset*[1-tand(thetaValidation) 1+tand(thetaValidation)],[sz 0],'g--')

    hold off
    label = string(Y1Pred(idx(i)));
    title("Label: " + label)
end
```



See Also

`convolution2dLayer` | `batchNormalizationLayer` | `reluLayer` | `fullyConnectedLayer` | `softmaxLayer` | `assembleNetwork` | `predict`

More About

- “Multiple-Input and Multiple-Output Networks” on page 1-19
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Train Network with Multiple Outputs” on page 3-61
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Deep Learning Layers” on page 1-21

Automatic Differentiation Background

What Is Automatic Differentiation?

Automatic differentiation (also known as autodiff, AD, or algorithmic differentiation) is a widely used tool for deep learning. See Books on Automatic Differentiation. It is particularly useful for creating and training complex deep learning models without needing to compute derivatives manually for optimization. For examples showing how to create and customize deep learning models, training loops, and loss functions, see “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209.

Automatic differentiation is a set of techniques for evaluating derivatives (gradients) numerically. The method uses symbolic rules for differentiation, which are more accurate than finite difference approximations. Unlike a purely symbolic approach, automatic differentiation evaluates expressions numerically early in the computations, rather than carrying out large symbolic computations. In other words, automatic differentiation evaluates derivatives at particular numeric values; it does not construct symbolic expressions for derivatives.

- Forward mode evaluates a numerical derivative by performing elementary derivative operations concurrently with the operations of evaluating the function itself. As detailed in the next section, the software performs these computations on a computational graph.
- Reverse mode automatic differentiation uses an extension of the forward mode computational graph to enable the computation of a gradient by a reverse traversal of the graph. As the software runs the code to compute the function and its derivative, it records operations in a data structure called a trace.

As many researchers have noted (for example, Baydin, Pearlmutter, Radul, and Siskind [1]), for a scalar function of many variables, reverse mode calculates the gradient more efficiently than forward mode. Because a deep learning loss function is a scalar function of all the weights, Deep Learning Toolbox automatic differentiation uses reverse mode.

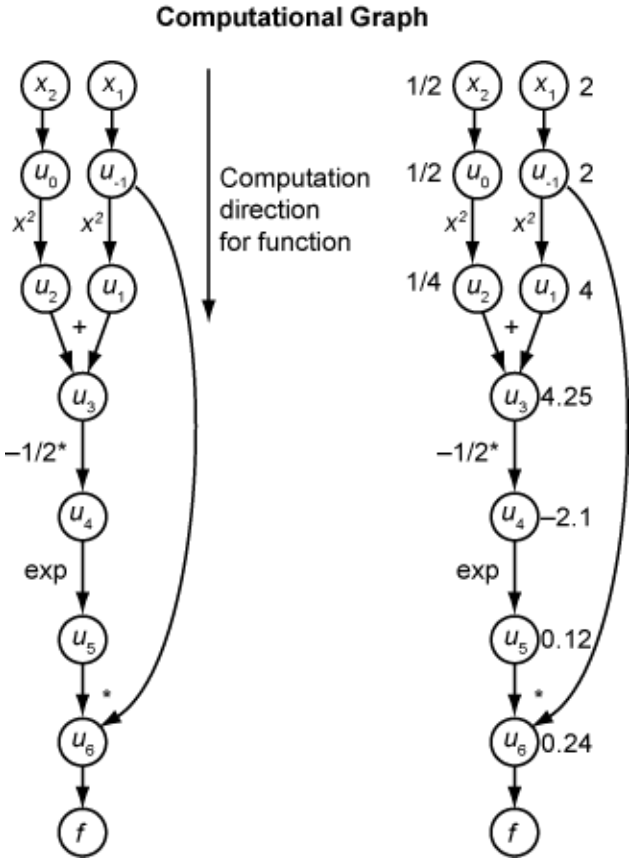
Forward Mode

Consider the problem of evaluating this function and its gradient:

$$f(x) = x_1 \exp\left(-\frac{1}{2}\left(x_1^2 + x_2^2\right)\right).$$

Automatic differentiation works at particular points. In this case, take $x_1 = 2$, $x_2 = 1/2$.

The following computational graph encodes the calculation of the function $f(x)$.



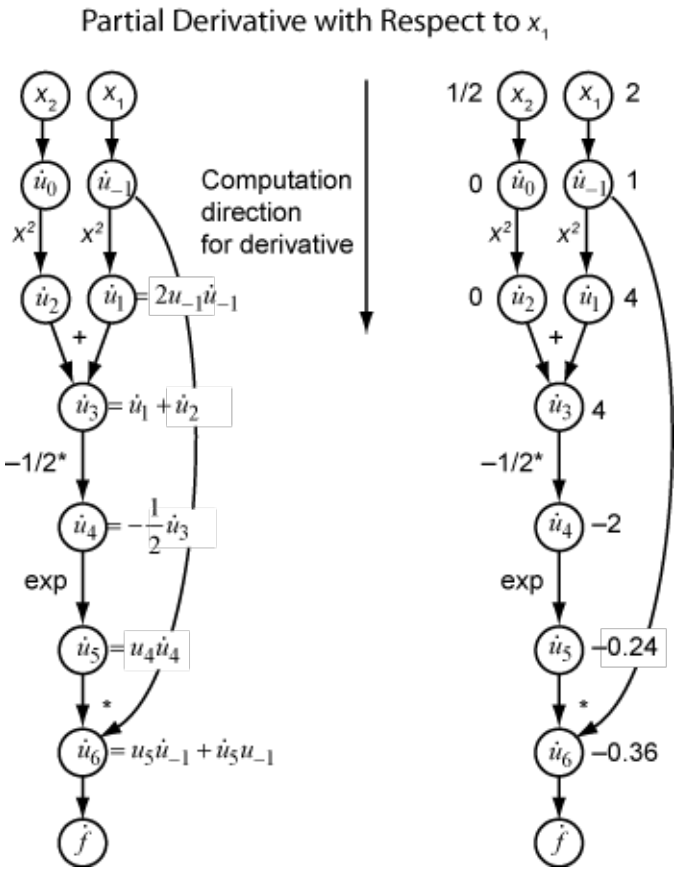
To compute the gradient of $f(x)$ using forward mode, you compute the same graph in the same direction, but modify the computation based on the elementary rules of differentiation. To further simplify the calculation, you fill in the value of the derivative of each subexpression u_i as you go. To compute the entire gradient, you must traverse the graph twice, once for the partial derivative with respect to each independent variable. Each subexpression in the chain rule has a numeric value, so the entire expression has the same sort of evaluation graph as the function itself.

The computation is a repeated application of the chain rule. In this example, the derivative of f with respect to x_1 expands to this expression:

$$\begin{aligned}\frac{df}{dx_1} &= \frac{du_6}{dx_1}\\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial x_1}\\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial x_1}\\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial x_1}\\ &= \frac{\partial u_6}{\partial u_{-1}} + \frac{\partial u_6}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial u_1}\frac{\partial u_1}{\partial x_1}.\end{aligned}$$

Let $\dot{u}_i$ represent the derivative of the expression u_i with respect to x_1. Using the evaluated values of the u_i from the function evaluation, you compute the partial derivative of f with respect to x_1 as shown

in the following figure. Notice that all the values of the $\dot{u}_i$ become available as you traverse the graph from top to bottom.



To compute the partial derivative with respect to x_2, you traverse a similar computational graph. Therefore, when you compute the gradient of the function, the number of graph traversals is the same as the number of variables. This process is too slow for typical deep learning applications, which have thousands or millions of variables.

Reverse Mode

Reverse mode uses one forward traversal of a computational graph to set up the trace. Then it computes the entire gradient of the function in one traversal of the graph in the opposite direction. For deep learning applications, this mode is far more efficient.

The theory behind reverse mode is also based on the chain rule, along with associated adjoint variables denoted with an overbar. The adjoint variable for u_i is

$$\bar{u}_i = \frac{\partial f}{\partial u_i}.$$

In terms of the computational graph, each outgoing arrow from a variable contributes to the corresponding adjoint variable by its term in the chain rule. For example, the variable u_{-1} has outgoing arrows to two variables, u_1 and u_6. The graph has the associated equation

$$\frac{\partial f}{\partial u_{-1}} = \frac{\partial f}{\partial u_1}\frac{\partial u_1}{\partial u_{-1}} + \frac{\partial f}{\partial u_6}\frac{\partial u_6}{\partial u_{-1}}$$
$$= \bar{u}_1\frac{\partial u_1}{\partial u_{-1}} + \bar{u}_6\frac{\partial u_6}{\partial u_{-1}}.$$

In this calculation, recalling that $u_1 = u_{-1}^2$ and $u_6 = u_5u_{-1}$, you obtain

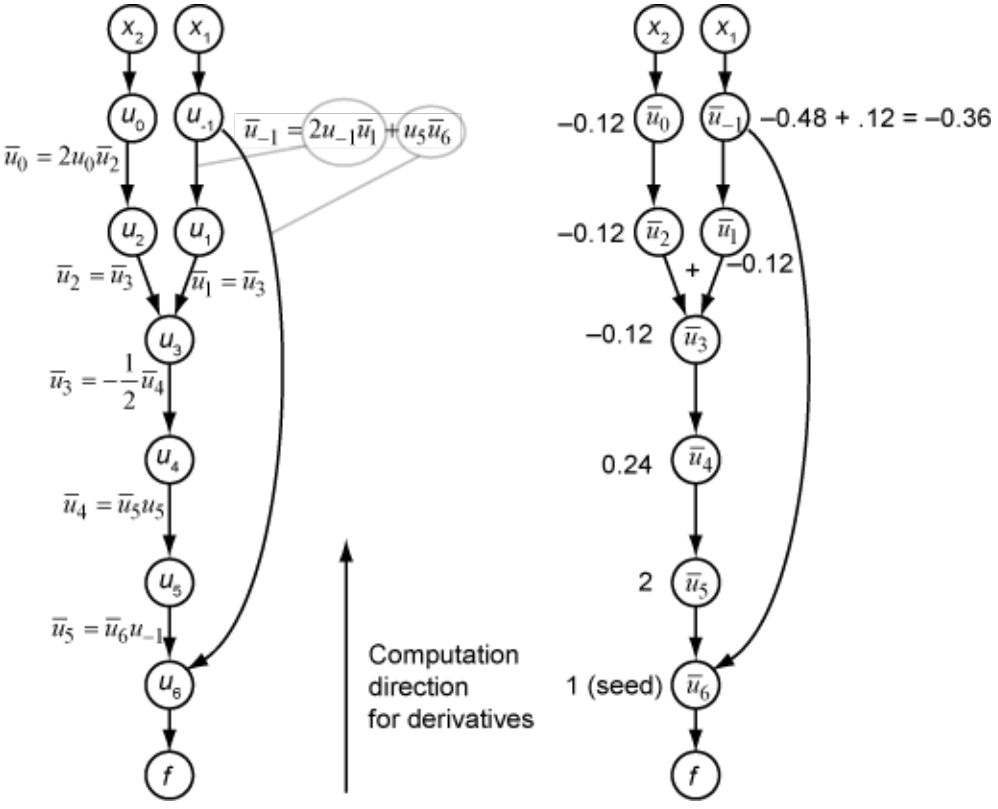
$$\bar{u}_{-1} = \bar{u}_1 2u_{-1} + \bar{u}_6u_5.$$

During the forward traversal of the graph, the software calculates the intermediate variables u_i. During the reverse traversal, starting from the seed value $\bar{u}_6 = \frac{\partial f}{\partial f} = 1$, the reverse mode computation obtains the adjoint values for all variables. Therefore, the reverse mode computes the gradient in just one computation, saving a great deal of time compared to forward mode.

The following figure shows the computation of the gradient in reverse mode for the function

$$f(x) = x_1\exp\left(-\frac{1}{2}\left(x_1^2 + x_2^2\right)\right).$$

Again, the computation takes $x_1 = 2$, $x_2 = 1/2$. The reverse mode computation relies on the u_i values that are obtained during the computation of the function in the original computational graph. In the right portion of the figure, the computed values of the adjoint variables appear next to the adjoint variable names, using the formulas from the left portion of the figure.



The final gradient values appear as $\bar{u}_0 = \frac{\partial f}{\partial u_0} = \frac{\partial f}{\partial x_2}$ and $\bar{u}_{-1} = \frac{\partial f}{\partial u_{-1}} = \frac{\partial f}{\partial x_1}$.

For more details, see Baydin, Pearlmutter, Radul, and Siskind [1] or the Wikipedia article on automatic differentiation [2].

References

[1] Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. "Automatic Differentiation in Machine Learning: a Survey." *The Journal of Machine Learning Research,* 18(153), 2018, pp. 1–43. Available at https://arxiv.org/abs/1502.05767.

[2] *Automatic differentiation.* Wikipedia. Available at https://en.wikipedia.org/wiki/Automatic_differentiation.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Train Network Using Model Function” on page 18-259
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “List of Functions with dlarray Support” on page 18-423

Use Automatic Differentiation In Deep Learning Toolbox

In this section...

Custom Training and Calculations Using Automatic Differentiation

Automatic differentiation makes it easier to create custom training loops, custom layers, and other deep learning customizations.

Generally, the simplest way to customize deep learning training is to create a `dlnetwork`. Include the layers you want in the network. Then perform training in a custom loop by using some sort of gradient descent, where the gradient is the gradient of the objective function. The objective function can be classification error, cross-entropy, or any other relevant scalar function of the network weights. See “List of Functions with dlarray Support” on page 18-423.

This example is a high-level version of a custom training loop. Here, `f` is the objective function, such as loss, and `g` is the gradient of the objective function with respect to the weights in the network `net`. The `update` function represents some type of gradient descent.

```
% High-level training loop
n = 1;
while (n < nmax)
    [f,g] = dlfeval(@model,net,dlX,t);
    net = update(net,g);
    n = n + 1;
end
```

You call `dlfeval` to compute the numeric value of the objective and gradient. To enable the automatic computation of the gradient, the data `dlX` must be a `dlarray`.

```
dlX = dlarray(X);
```

The objective function has a `dlgradient` call to calculate the gradient. The `dlgradient` call must be inside of the function that `dlfeval` evaluates.

```
function [f,g] = model(net,dlX,T)
% Calculate objective using supported functions for dlarray
    y = forward(net,dlX);
    f = fcnvalue(y,T); % crossentropy or similar
    g = dlgradient(f,net.Learnables); % Automatic gradient
end
```

For an example using a `dlnetwork` with a `dlfeval`-`dlgradient`-`dlarray` syntax and a custom training loop, see “Train Network Using Custom Training Loop” on page 18-225. For further details on custom training using automatic differentiation, see “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209.

Use dlgradient and dlfeval Together for Automatic Differentiation

To use automatic differentiation, you must call `dlgradient` inside a function and evaluate the function using `dlfeval`. Represent the point where you take a derivative as a `dlarray` object, which manages the data structures and enables tracing of evaluation. For example, the Rosenbrock function is a common test function for optimization.

```
function [f,grad] = rosenbrock(x)

f = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
grad = dlgradient(f,x);

end
```

Calculate the value and gradient of the Rosenbrock function at the point `x0` = [–1,2]. To enable automatic differentiation in the Rosenbrock function, pass `x0` as a `dlarray`.

```
x0 = dlarray([-1,2]);
[fval,gradval] = dlfeval(@rosenbrock,x0)
```

```
fval =

  1x1 dlarray

   104


gradval =

  1x2 dlarray

   396   200
```

For an example using automatic differentiation, see “Train Network Using Custom Training Loop” on page 18-225.

Derivative Trace

To evaluate a gradient numerically, a `dlarray` constructs a data structure for reverse mode differentiation, as described in “Automatic Differentiation Background” on page 18-200. This data structure is the trace of the derivative computation. Keep in mind these guidelines when using automatic differentiation and the derivative trace:

- Do not introduce a new `dlarray` inside of an objective function calculation and attempt to differentiate with respect to that object. For example:

  ```
  function [dy,dy1] = fun(x1)
  x2 = dlarray(0);
  y = x1 + x2;
  dy = dlgradient(y,x2); % Error: x2 is untraced
  dy1 = dlgradient(y,x1); % No error even though y has an untraced portion
  end
  ```

- Do not use `extractdata` with a traced argument. Doing so breaks the tracing. For example:

  ```
  fun = @(x)dlgradient(x + atan(extractdata(x)),x);
  % Gradient for any point is 1 due to the leading 'x' term in fun.
  dlfeval(fun,dlarray(2.5))
  ```

```
ans =

  1x1 dlarray

     1
```

However, you can use `extractdata` to introduce a new independent variable from a dependent one.

- When working in parallel, moving traced dlarray objects between the client and workers breaks the tracing. The traced dlarray object is saved on the worker and loaded in the client as an untraced dlarray object. To avoid breaking tracing when working in parallel, compute all required gradients on the worker and then combine the gradients on the client. For an example, see “Train Network in Parallel with Custom Training Loop” on page 7-55.
- Use only supported functions. For a list of supported functions, see “List of Functions with dlarray Support” on page 18-423. To use an unsupported function *f*, try to implement *f* using supported functions.

Characteristics of Automatic Derivatives

- You can evaluate gradients using automatic differentiation only for scalar-valued functions. Intermediate calculations can have any number of variables, but the final function value must be scalar. If you need to take derivatives of a vector-valued function, take derivatives of one component at a time. In this case, consider setting the `dlgradient` `'RetainData'` name-value pair argument to `true`.
- A call to `dlgradient` evaluates derivatives at a particular point. The software generally makes an arbitrary choice for the value of a derivative when there is no theoretical value. For example, the `relu` function, `relu(x) = max(x,0)`, is not differentiable at `x = 0`. However, `dlgradient` returns a value for the derivative.

```
x = dlarray(0);
y = dlfeval(@(t)dlgradient(relu(t),t),x)

y =

  1x1 dlarray

     0
```

The value at the nearby point `eps` is different.

```
x = dlarray(eps);
y = dlfeval(@(t)dlgradient(relu(t),t),x)

y =

  1x1 dlarray

     1
```

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Train Network Using Model Function” on page 18-259
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “List of Functions with dlarray Support” on page 18-423

Define Custom Training Loops, Loss Functions, and Networks

In this section...

For most deep learning tasks, you can use a pretrained network and adapt it to your own data. For an example showing how to use transfer learning to retrain a convolutional neural network to classify a new set of images, see “Train Deep Learning Network to Classify New Images” on page 3-6. Alternatively, you can create and train networks from scratch using `layerGraph` objects with the `trainNetwork` and `trainingOptions` functions.

If the `trainingOptions` function does not provide the training options that you need for your task, then you can create a custom training loop using automatic differentiation. To learn more, see “Define Deep Learning Network for Custom Training Loops” on page 18-209.

If Deep Learning Toolbox does not provide the layers you need for your task (including output layers that specify loss functions), then you can create a custom layer. To learn more, see “Define Custom Deep Learning Layers” on page 18-9. For loss functions that cannot be specified using an output layer, you can specify the loss in a custom training loop. To learn more, see “Specify Loss Functions” on page 18-212. For networks that cannot be created using layer graphs, you can define custom networks as a function. To learn more, see “Define Network as Model Function” on page 18-209.

For more information about which training method to use for which task, see “Train Deep Learning Model in MATLAB” on page 18-3.

Define Deep Learning Network for Custom Training Loops

Define Network as dlnetwork Object

For most tasks, you can control the training algorithm details using the `trainingOptions` and `trainNetwork` functions. If the `trainingOptions` function does not provide the options you need for your task (for example, a custom learning rate schedule), then you can define your own custom training loop using a `dlnetwork` object. A `dlnetwork` object allows you to train a network specified as a layer graph using automatic differentiation.

For networks specified as a layer graph, you can create a `dlnetwork` object from the layer graph by using the `dlnetwork` function directly.

```
dlnet = dlnetwork(lgraph);
```

For a list of layers supported by `dlnetwork` objects, see the “Supported Layers” section of the `dlnetwork` page. For an example showing how to train a network with a custom learning rate schedule, see “Train Network Using Custom Training Loop” on page 18-225.

Define Network as Model Function

For architectures that cannot be created using layer graphs (for example, a Siamese network that requires shared weights), you can define the model as a function of the form `[dlY1,...,dlYM] = model(parameters,dlX1,...,dlXN)`, where `parameters` contains the network parameters, `dlX1,...,dlXN` corresponds to the input data for the `N` model inputs, and `dlY1,...,dlYM`

corresponds to the `M` model outputs. To train a deep learning model defined as a function, use a custom training loop. For an example, see “Train Network Using Model Function” on page 18-259.

When you define a deep learning model as a function, you must manually initialize the layer weights. For more information, see “Initialize Learnable Parameters for Model Function” on page 18-292.

If you define a custom network as a function, then the model function must support automatic differentiation. You can use the following deep learning operations. The functions listed here are only a subset. For a complete list of functions that support `dlarray` input, see “List of Functions with dlarray Support” on page 18-423.

Function	Description
`avgpool`	The average pooling operation performs downsampling by dividing the input into pooling regions and computing the average value of each region.
`batchnorm`	The batch normalization operation normalizes the input data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization between convolution and nonlinear operations such as `relu`.
`crossentropy`	The cross-entropy operation computes the cross-entropy loss between network predictions and target values for single-label and multi-label classification tasks.
`crosschannelnorm`	The cross-channel normalization operation uses local responses in different channels to normalize each activation. Cross-channel normalization typically follows a `relu` operation. Cross-channel normalization is also known as local response normalization.
`ctc`	The CTC operation computes the connectionist temporal classification (CTC) loss between unaligned sequences.
`dlconv`	The convolution operation applies sliding filters to the input data. Use the `dlconv` function for deep learning convolution, grouped convolution, and channel-wise separable convolution.
`dlode45`	The neural ordinary differential equation (ODE) operation returns the solution of a specified ODE.
`dltranspconv`	The transposed convolution operation upsamples feature maps.
`embed`	The embed operation converts numeric indices to numeric vectors, where the indices correspond to discrete data. Use embeddings to map discrete data such as categorical values or words to numeric vectors.

Function	Description
`fullyconnect`	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
`groupnorm`	The group normalization operation normalizes the input data across grouped subsets of channels for each observation independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use group normalization between convolution and nonlinear operations such as `relu`.
`gru`	The gated recurrent unit (GRU) operation allows a network to learn dependencies between time steps in time series and sequence data.
`huber`	The Huber operation computes the Huber loss between network predictions and target values for regression tasks. When the `'TransitionPoint'` option is 1, this is also known as *smooth L_1 loss*.
`instancenorm`	The instance normalization operation normalizes the input data across each channel for each observation independently. To improve the convergence of training the convolutional neural network and reduce the sensitivity to network hyperparameters, use instance normalization between convolution and nonlinear operations such as `relu`.
`l1loss`	The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean absolute error (MAE).
`l2loss`	The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean squared error (MSE).
`layernorm`	The layer normalization operation normalizes the input data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization after the learnable operations, such as LSTM and fully connect operations.

Function	Description
`leakyrelu`	The leaky rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is multiplied by a fixed scale factor.
`lstm`	The long short-term memory (LSTM) operation allows a network to learn long-term dependencies between time steps in time series and sequence data.
`maxpool`	The maximum pooling operation performs downsampling by dividing the input into pooling regions and computing the maximum value of each region.
`maxunpool`	The maximum unpooling operation unpools the output of a maximum pooling operation by upsampling and padding with zeros.
`mse`	The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression tasks.
`onehotdecode`	The one-hot decode operation decodes probability vectors, such as the output of a classification network, into classification labels. The input `A` can be a `dlarray`. If `A` is formatted, the function ignores the data format.
`relu`	The rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is set to zero.
`sigmoid`	The sigmoid activation operation applies the sigmoid function to the input data.
`softmax`	The softmax activation operation applies the softmax function to the channel dimension of the input data.

Specify Loss Functions

When you use a custom training loop, you must calculate the loss in the model gradients function. Use the loss value when computing gradients for updating the network weights. To compute the loss, you can use the following functions.

Function	Description
`softmax`	The softmax activation operation applies the softmax function to the channel dimension of the input data.

Function	Description
`sigmoid`	The sigmoid activation operation applies the sigmoid function to the input data.
`crossentropy`	The cross-entropy operation computes the cross-entropy loss between network predictions and target values for single-label and multi-label classification tasks.
`l1loss`	The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean absolute error (MAE).
`l2loss`	The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean squared error (MSE).
`huber`	The Huber operation computes the Huber loss between network predictions and target values for regression tasks. When the `'TransitionPoint'` option is 1, this is also known as *smooth L_1 loss*.
`mse`	The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression tasks.
`ctc`	The CTC operation computes the connectionist temporal classification (CTC) loss between unaligned sequences.

Alternatively, you can use a custom loss function by creating a function of the form `loss = myLoss(Y,T)`, where `Y` and `T` correspond to the network predictions and targets, respectively, and `loss` is the returned loss.

For an example showing how to train a generative adversarial network (GAN) that generates images using a custom loss function, see “Train Generative Adversarial Network (GAN)” on page 3-76.

Update Learnable Parameters Using Automatic Differentiation

When you train a deep learning model with a custom training loop, the software minimizes the loss with respect to the learnable parameters. To minimize the loss, the software uses the gradients of the loss with respect to the learnable parameters. To calculate these gradients using automatic differentiation, you must define a model gradients function.

Define Model Gradients Function

For a model specified as a `dlnetwork` object, create a function of the form `gradients = modelGradients(dlnet,dlX,T)`, where `dlnet` is the network, `dlX` is the network input, `T` contains the targets, and `gradients` contains the returned gradients. Optionally, you can pass extra arguments to the gradients function (for example, if the loss function requires extra information), or return extra arguments (for example, metrics for plotting the training progress).

For a model specified as a function, create a function of the form `gradients = modelGradients(parameters,dlX,T)`, where `parameters` contains the learnable parameters, `dlX` is the model input, `T` contains the targets, and `gradients` contains the returned gradients. Optionally, you can pass extra arguments to the gradients function (for example, if the loss function requires extra information), or return extra arguments (for example, metrics for plotting the training progress).

To learn more about defining model gradients functions for custom training loops, see “Define Model Gradients Function for Custom Training Loop” on page 18-231.

Update Learnable Parameters

To evaluate the model gradients using automatic differentiation, use the `dlfeval` function, which evaluates a function with automatic differentiation enabled. For the first input of `dlfeval`, pass the model gradients function specified as a function handle. For the following inputs, pass the required variables for the model gradients function. For the outputs of the `dlfeval` function, specify the same outputs as the model gradients function.

To update the learnable parameters using the gradients, you can use the following functions.

Function	Description
`adamupdate`	Update parameters using adaptive moment estimation (Adam)
`rmspropupdate`	Update parameters using root mean squared propagation (RMSProp)
`sgdmupdate`	Update parameters using stochastic gradient descent with momentum (SGDM)
`dlupdate`	Update parameters using custom function

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Make Predictions Using dlnetwork Object” on page 18-255

- “Make Predictions Using Model Function” on page 18-286
- “Train Network Using Model Function” on page 18-259
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “Train Deep Learning Model in MATLAB” on page 18-3
- “Define Custom Deep Learning Layers” on page 18-9
- “List of Functions with dlarray Support” on page 18-423
- “Automatic Differentiation Background” on page 18-200
- “Use Automatic Differentiation In Deep Learning Toolbox” on page 18-205

Specify Training Options in Custom Training Loop

For most tasks, you can control the training algorithm details using the `trainingOptions` and `trainNetwork` functions. If the `trainingOptions` function does not provide the options you need for your task (for example, a custom learning rate schedule), then you can define your own custom training loop using a `dlnetwork` object. A `dlnetwork` object allows you to train a network specified as a layer graph using automatic differentiation.

To specify the same options as the `trainingOptions`, use these examples as a guide:

Training Option	`trainingOptions` Argument	Example
Adam solver	• `solverName` • `'GradientDecayFactor'` • `'SquaredGradientDecayFactor'`	"Adaptive Moment Estimation (ADAM)" on page 18-217
RMSProp solver	• `solverName` • `'Epsilon'`	"Root Mean Square Propagation (RMSProp)" on page 18-217
SGDM solver	• `solverName` • `'Momentum'`	"Stochastic Gradient Descent with Momentum (SGDM)" on page 18-217
Learn rate	`'InitialLearnRate'`	"Learn Rate" on page 18-217
Learn rate schedule	• `'LearnRateSchedule'` • `'LearnRateDropPeriod'` • `'LearnRateDropFactor'`	"Piecewise Learn Rate Schedule" on page 18-217
Training progress	`'Plots'`	"Plots" on page 18-218
Verbose output	• `'Verbose'` • `'VerboseFrequency'`	"Verbose Output" on page 18-219
Mini-batch size	`'MiniBatchSize'`	"Mini-Batch Size" on page 18-220
Number of epochs	`'MaxEpochs'`	"Number of Epochs" on page 18-220
Validation	• `'ValidationData` • `'ValidationPatience'`	"Validation" on page 18-220
L_2 regularization	`'L2Regularization'`	"L2 Regularization" on page 18-222
Gradient clipping	• `'GradientThreshold'` • `'GradientThresholdMethod'`	"Gradient Clipping" on page 18-222
Single CPU or GPU training	`'ExecutionEnvironment'`	"Single CPU or GPU Training" on page 18-223
Checkpoints	`'CheckpointPath'`	"Checkpoints" on page 18-223

Solver Options

To specify the solver, use the `adamupdate`, `rmspropupdate`, and `sgdmupdate` functions for the update step in your training loop. To implement your own custom solver, update the learnable parameters using the `dlupdate` function.

Adaptive Moment Estimation (ADAM)

To update your network parameters using Adam, use the `adamupdate` function. Specify the gradient decay and the squared gradient decay factors using the corresponding input arguments.

Root Mean Square Propagation (RMSProp)

To update your network parameters using RMSProp, use the `rmspropupdate` function. Specify the denominator offset (epsilon) value using the corresponding input argument.

Stochastic Gradient Descent with Momentum (SGDM)

To update your network parameters using SGDM, use the `sgdmupdate` function. Specify the momentum using the corresponding input argument.

Learn Rate

To specify the learn rate, use the learn rate input arguments of the `adamupdate`, `rmspropupdate`, and `sgdmupdate` functions.

To easily adjust the learn rate or use it for custom learn rate schedules, set the initial learn rate before the custom training loop.

```
learnRate = 0.01;
```

Piecewise Learn Rate Schedule

To automatically drop the learn rate during training using a piecewise learn rate schedule, multiply the learn rate by a given drop factor after a specified interval.

To easily specify a piecewise learn rate schedule, create the variables `learnRate`, `learnRateSchedule`, `learnRateDropFactor`, and `learnRateDropPeriod`, where `learnRate` is the initial learn rate, `learnRateScedule` contains either `"piecewise"` or `"none"`, `learnRateDropFactor` is a scalar in the range [0, 1] that specifies the factor for dropping the learning rate, and `learnRateDropPeriod` is a positive integer that specifies how many epochs between dropping the learn rate.

```
learnRate = 0.01;
learnRateSchedule = "piecewise"
learnRateDropPeriod = 10;
learnRateDropFactor = 0.1;
```

Inside the training loop, at the end of each epoch, drop the learn rate when the `learnRateSchedule` option is `"piecewise"` and the current epoch number is a multiple of `learnRateDropPeriod`. Set the new learn rate to the product of the learn rate and the learn rate drop factor.

```
if learnRateSchedule == "piecewise" && mod(epoch,learnRateDropPeriod) == 0
    learnRate = learnRate * learnRateDropFactor;
end
```

Plots

To plot the training loss and accuracy during training, calculate the mini-batch loss and either the accuracy or the root-mean-squared-error (RMSE) in the model gradients function and plot them using an animated line.

To easily specify that the plot should be on or off, create the variable `plots` that contains either `"training-progress"` or `"none"`. To also plot validation metrics, use the same options `validationData` and `validationFrequency` described in “Validation” on page 18-220.

```
plots = "training-progress";

validationData = {XValidation, YValidation};
validationFrequency = 50;
```

Before training, initialize the animated lines using the `animatedline` function. For classification tasks create a plot for the training accuracy and the training loss. Also initialize animated lines for validation metrics when validation data is specified.

```
if plots == "training-progress"
    figure
    subplot(2,1,1)
    lineAccuracyTrain = animatedline;
    ylabel("Accuracy")

    subplot(2,1,2)
    lineLossTrain = animatedline;
    xlabel("Iteration")
    ylabel("Loss")

    if ~isempty(validationData)
        subplot(2,1,1)
        lineAccuracyValidation = animatedline;

        subplot(2,1,2)
        lineLossValidation = animatedline;
    end
end
```

For regression tasks, adjust the code by changing the variable names and labels so that it initializes plots for the training and validation RMSE instead of the training and validation accuracy.

Inside the training loop, at the end of an iteration, update the plot so that it includes the appropriate metrics for the network. For classification tasks, add points corresponding to the mini-batch accuracy and the mini-batch loss. If the validation data is nonempty, and the current iteration is either 1 or a multiple of the validation frequency option, then also add points for the validation data.

```
if plots == "training-progress"
    addpoints(lineAccuracyTrain,iteration,accuracyTrain)
    addpoints(lineLossTrain,iteration,lossTrain)

    if ~isempty(validationData) && (iteration == 1 || mod(iteration,validationFrequency) == 0)
        addpoints(lineAccuracyValidation,iteration,accuracyValidation)
        addpoints(lineLossValidation,iteration,lossValidation)
    end
end
```

where `accuracyTrain` and `lossTrain` correspond to the mini-batch accuracy and loss calculated in the model gradients function. For regression tasks, use the mini-batch RMSE losses instead of the mini-batch accuracies.

Tip The `addpoints` function requires the data points to have type `double`. To extract numeric data from `dlarray` objects, use the `extractdata` function. To collect data from a GPU, use the `gather` function.

To learn how to compute validation metrics, see “Validation” on page 18-220.

Verbose Output

To display the training loss and accuracy during training in a verbose table, calculate the mini-batch loss and either the accuracy (for classification tasks) or the RMSE (for regression tasks) in the model gradients function and display them using the `disp` function.

To easily specify that the verbose table should be on or off, create the variables `verbose` and `verboseFrequency`, where `verbose` is `true` or `false` and `verbosefrequency` specifies how many iterations between printing verbose output. To display validation metrics, use the same options `validationData` and `validationFrequency` described in “Validation” on page 18-220.

```
verbose = true
verboseFrequency = 50;

validationData = {XValidation, YValidation};
validationFrequency = 50;
```

Before training, display the verbose output table headings and initialize a timer using the `tic` function.

```
disp("|========================================================================================================================|")
disp("|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Validation  |  Mini-batch  |  Validation  |  Base Learning  |")
disp("|         |             |   (hh:mm:ss)   |   Accuracy   |   Accuracy   |     Loss     |     Loss     |      Rate       |")
disp("|========================================================================================================================|")

start = tic;
```

For regression tasks, adjust the code so that it displays the training and validation RMSE instead of the training and validation accuracy.

Inside the training loop, at the end of an iteration, print the verbose output when the `verbose` option is `true` and it is either the first iteration or the iteration number is a multiple of `verboseFrequency`.

```
if verbose && (iteration == 1 || mod(iteration,verboseFrequency) == 0
    D = duration(0,0,toc(start),'Format','hh:mm:ss');

    if isempty(validationData) || mod(iteration,validationFrequency) ~= 0
        accuracyValidation = "";
        lossValidation = "";
    end

    disp("| " + ...
        pad(epoch,7,'left') + " | " + ...
        pad(iteration,11,'left') + " | " + ...
        pad(D,14,'left') + " | " + ...
        pad(accuracyTrain,12,'left') + " | " + ...
        pad(accuracyValidation,12,'left') + " | " + ...
        pad(lossTrain,12,'left') + " | " + ...
        pad(lossValidation,12,'left') + " | " + ...
        pad(learnRate,15,'left') + " |")
end
```

For regression tasks, adjust the code so that it displays the training and validation RMSE instead of the training and validation accuracy.

When training is finished, print the last border of the verbose table.

```
disp("|=======================================================================================================================|")
```

To learn how to compute validation metrics, see “Validation” on page 18-220.

Mini-Batch Size

Setting the mini-batch size depends on the format of data or type of datastore used.

To easily specify the mini-batch size, create a variable `miniBatchSize`.

```
miniBatchSize = 128;
```

For data in an image datastore, before training, set the `ReadSize` property of the datastore to the mini-batch size.

```
imds.ReadSize = miniBatchSize;
```

For data in an augmented image datastore, before training, set the `MiniBatchSize` property of the datastore to the mini-batch size.

```
augimds.MiniBatchSize = miniBatchSize;
```

For in-memory data, during training at the start of each iteration, read the observations directly from the array.

```
idx = ((iteration - 1)*miniBatchSize + 1):(iteration*miniBatchSize);
X = XTrain(:,:,:,idx);
```

Number of Epochs

Specify the maximum number of epochs for training in the outer `for` loop of the training loop.

To easily specify the maximum number of epochs, create the variable `maxEpochs` that contains the maximum number of epochs.

```
maxEpochs = 30;
```

In the outer `for` loop of the training loop, specify to loop over the range 1, 2, ..., `maxEpochs`.

```
for epoch = 1:maxEpochs
    ...
end
```

Validation

To validate your network during training, set aside a held-out validation set and evaluate how well the network performs on that data.

To easily specify validation options, create the variables `validationData` and `validationFrequency`, where `validationData` contains the validation data or is empty and `validationFrequency` specifies how many iterations between validating the network.

```
validationData = {XValidation,YValidation};
validationFrequency = 50;
```

During the training loop, after updating the network parameters, test how well the network performs on the held-out validation set using the `predict` function. Validate the network only when validation

data is specified and it is either the first iteration or the current iteration is a multiple of the `validationFrequency` option.

```
if iteration == 1 || mod(iteration,validationFrequency) == 0
    dlYPredValidation = predict(dlnet,dlXValidation);
    lossValidation = crossentropy(softmax(dlYPredValidation), YValidation);

    [~,idx] = max(dlYPredValidation);
    labelsPredValidation = classNames(idx);

    accuracyValidation = mean(labelsPredValidation == labelsValidation);
end
```

Here, `YValidation` is a dummy variable corresponding to the labels in `classNames`. To calculate the accuracy, convert `YValidation` to an array of labels.

For regression tasks, adjust the code so that it calculates the validation RMSE instead of the validation accuracy.

Early Stopping

To stop training early when the loss on the held-out validation stops decreasing, use a flag to break out of the training loops.

To easily specify the validation patience (the number of times that the validation loss can be larger than or equal to the previously smallest loss before network training stops), create the variable `validationPatience`.

```
validationPatience = 5;
```

Before training, initialize a variables `earlyStop` and `validationLosses`, where `earlyStop` is a flag to stop training early and `validationLosses` contains the losses to compare. Initialize the early stopping flag with `false` and array of validation losses with `inf`.

```
earlyStop = false;
if isfinite(validationPatience)
    validationLosses = inf(1,validationPatience);
end
```

Inside the training loop, in the loop over mini-batches, add the `earlyStop` flag to the loop condition.

```
while hasdata(ds) && ~earlyStop
    ...
end
```

During the validation step, append the new validation loss to the array `validationLosses`. If the first element of the array is the smallest, then set the `earlyStop` flag to `true`. Otherwise, remove the first element.

```
if isfinite(validationPatience)
    validationLosses = [validationLosses validationLoss];
    if min(validationLosses) == validationLosses(1)
        earlyStop = true;
    else
        validationLosses(1) = [];
    end
end
```

L2 Regularization

To apply L_2 regularization to the weights, use the `dlupdate` function.

To easily specify the L_2 regularization factor, create the variable `l2Regularization` that contains the L_2 regularization factor.

```
l2Regularization = 0.0001;
```

During training, after computing the model gradients, for each of the weight parameters, add the product of the L_2 regularization factor and the weights to the computed gradients using the `dlupdate` function. To update only the weight parameters, extract the parameters with name `"Weights"`.

```
idx = dlnet.Learnables.Parameter == "Weights";
gradients(idx,:) = dlupdate(@(g,w) g + l2Regularization*w, gradients(idx,:), dlnet.Learnables(idx,:));
```

After adding the L_2 regularization parameter to the gradients, update the network parameters.

Gradient Clipping

To clip the gradients, use the `dlupdate` function.

To easily specify gradient clipping options, create the variables `gradientThresholdMethod` and `gradientThreshold`, where `gradientThresholdMethod` contains `"global-l2norm"`, `"l2norm"`, or `"absolute-value"`, and `gradientThreshold` is a positive scalar containing the threshold or `inf`.

```
gradientThresholdMethod = "global-l2norm";
gradientThreshold = 2;
```

Create functions named `thresholdGlobalL2Norm`, `thresholdL2Norm`, and `thresholdAbsoluteValue` that apply the `"global-l2norm"`, `"l2norm"`, and `"absolute-value"` threshold methods, respectively.

For the `"global-l2norm"` option, the function operates on all gradients of the model.

```
function gradients = thresholdGlobalL2Norm(gradients,gradientThreshold)

globalL2Norm = 0;
for i = 1:numel(gradients)
    globalL2Norm = globalL2Norm + sum(gradients{i}(:).^2);
end
globalL2Norm = sqrt(globalL2Norm);

if globalL2Norm > gradientThreshold
    normScale = gradientThreshold / globalL2Norm;
    for i = 1:numel(gradients)
        gradients{i} = gradients{i} * normScale;
    end
end

end
```

For the `"l2norm"` and `"absolute-value"` options, the functions operate on each gradient independently.

```
function gradients = thresholdL2Norm(gradients,gradientThreshold)

gradientNorm = sqrt(sum(gradients(:).^2));
if gradientNorm > gradientThreshold
    gradients = gradients * (gradientThreshold / gradientNorm);
end

end
```

```
function gradients = thresholdAbsoluteValue(gradients,gradientThreshold)

gradients(gradients > gradientThreshold) = gradientThreshold;
gradients(gradients < -gradientThreshold) = -gradientThreshold;

end
```

During training, after computing the model gradients, apply the appropriate gradient clipping method to the gradients using the `dlupdate` function. Because the `"global-l2norm"` option requires all the model gradients, apply the `thresholdGlobalL2Norm` function directly to the gradients. For the `"l2norm"` and `"absolute-value"` options, update the gradients independently using the `dlupdate` function.

```
switch gradientThresholdMethod
    case "global-l2norm"
        gradients = thresholdGlobalL2Norm(gradients, gradientThreshold);
    case "l2norm"
        gradients = dlupdate(@(g) thresholdL2Norm(g, gradientThreshold),gradients);
    case "absolute-value"
        gradients = dlupdate(@(g) thresholdAbsoluteValue(g, gradientThreshold),gradients);
end
```

After applying the gradient threshold operation, update the network parameters.

Single CPU or GPU Training

The software, by default, performs calculations using only the CPU. To train on a single GPU, convert the data to `gpuArray` objects. Using a GPU requires Parallel Computing Toolbox and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

To easily specify the execution environment, create the variable `executionEnvironment` that contains either `"cpu"`, `"gpu"`, or `"auto"`.

```
executionEnvironment = "auto"
```

During training, after reading a mini-batch, check the execution environment option and convert the data to a `gpuArray` if necessary. The `canUseGPU` function checks for useable GPUs.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end
```

Checkpoints

To save checkpoint networks during training save the network using the `save` function.

To easily specify whether checkpoints should be switched on, create the variable `checkpointPath` contains the folder for the checkpoint networks or is empty.

```
checkpointPath = fullfile(tempdir,"checkpoints");
```

If the checkpoint folder does not exist, then before training, create the checkpoint folder.

```
if ~exist(checkpointPath,"dir")
    mkdir(checkpointPath)
end
```

During training, at the end of an epoch, save the network in a MAT file. Specify a file name containing the current iteration number, date, and time.

```
if ~isempty(checkpointPath)
    D = datestr(now,'yyyy_mm_dd__HH_MM_SS');
    filename = "dlnet_checkpoint__" + iteration + "__" + D + ".mat";
    save(filename,"dlnet")
end
```

where `dlnet` is the `dlnetwork` object to be saved.

See Also

`adamupdate` | `rmspropupdate` | `sgdmupdate` | `dlupdate` | `dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Train Network Using Custom Training Loop” on page 18-225
- “Train Network Using Model Function” on page 18-259
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Make Predictions Using Model Function” on page 18-286
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Train Generative Adversarial Network (GAN)” on page 3-76
- “List of Functions with dlarray Support” on page 18-423

Train Network Using Custom Training Loop

This example shows how to train a network that classifies handwritten digits with a custom learning rate schedule.

If `trainingOptions` does not provide the options you need (for example, a custom learning rate schedule), then you can define your own custom training loop using automatic differentiation.

This example trains a network to classify handwritten digits with the *time-based decay* learning rate schedule: for each iteration, the solver uses the learning rate given by $\rho_t = \frac{\rho_0}{1 + k\,t}$, where t is the iteration number, ρ_0 is the initial learning rate, and k is the decay.

Load Training Data

Load the digits data as an image datastore using the `imageDatastore` function and specify the folder containing the image data.

```
dataFolder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ....
    'LabelSource','foldernames');
```

Partition the data into training and validation sets. Set aside 10% of the data for validation using the `splitEachLabel` function.

```
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.9,'randomize');
```

The network used in this example requires input images of size 28-by-28-by-1. To automatically resize the training images, use an augmented image datastore. Specify additional augmentation operations to perform on the training images: randomly translate the images up to 5 pixels in the horizontal and vertical axes. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
inputSize = [28 28 1];
pixelRange = [-5 5];
imageAugmenter = imageDataAugmenter( ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain,'DataAugmentation',imageAugmente
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Determine the number of classes in the training data.

```
classes = categories(imdsTrain.Labels);
numClasses = numel(classes);
```

Define Network

Define the network for image classification.

```
layers = [
    imageInputLayer(inputSize,'Normalization','none','Name','input')
```

```
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding','same','Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    convolution2dLayer(3,20,'Padding','same','Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numClasses,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [11×2 table]
     Learnables: [14×3 table]
          State: [6×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes a `dlnetwork` object, a mini-batch of input data with corresponding labels and returns the gradients of the loss with respect to the learnable parameters in the network and the corresponding loss.

Specify Training Options

Train for ten epochs with a mini-batch size of 128.

```
numEpochs = 10;
miniBatchSize = 128;
```

Specify the options for SGDM optimization. Specify an initial learn rate of 0.01 with a decay of 0.01, and momentum 0.9.

```
initialLearnRate = 0.01;
decay = 0.01;
momentum = 0.9;
```

Train Model

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to convert the labels to one-hot encoded variables.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.

- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(augimdsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn',@preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Initialize the training progress plot.

```
figure
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the velocity parameter for the SGDM solver.

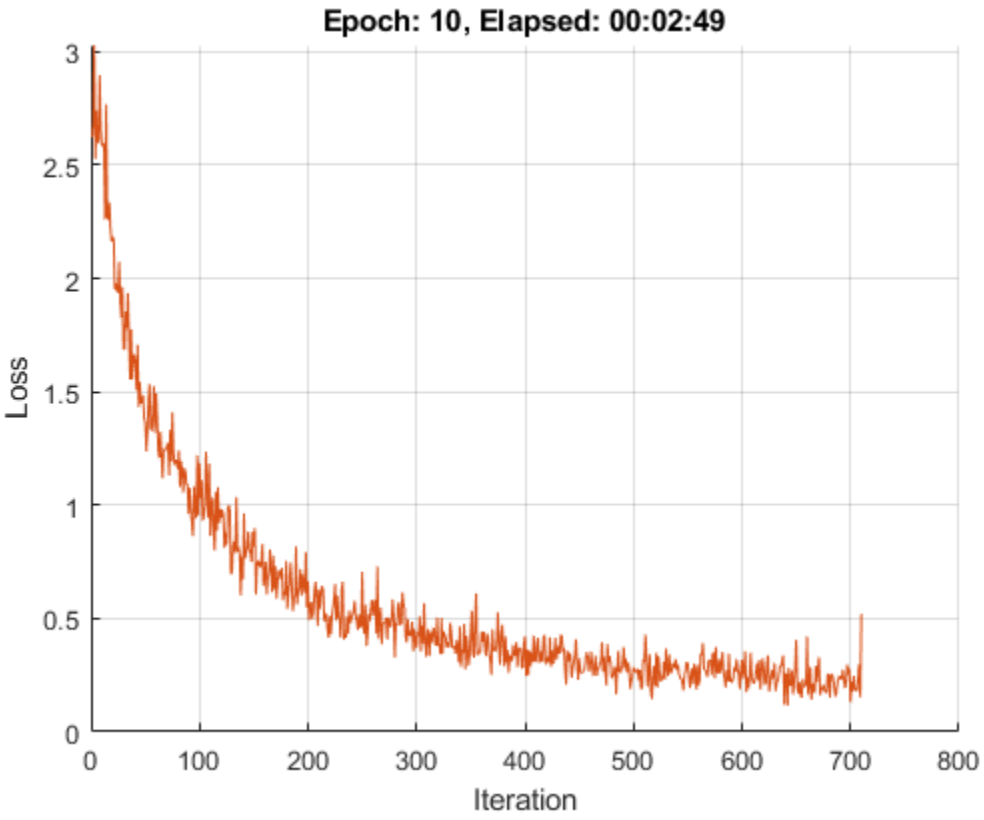
```
velocity = [];
```

Train the network using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using the `dlfeval` and `modelGradients` functions and update the network state.
- Determine the learning rate for the time-based decay learning rate schedule.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq);

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX, dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function and update the network state.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);
        dlnet.State = state;

        % Determine learning rate for time-based decay learning rate schedule.
        learnRate = initialLearnRate/(1 + decay*iteration);

        % Update the network parameters using the SGDM optimizer.
        [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate,momentum);
```

```
        % Display the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + ", Elapsed: " + string(D))
        drawnow
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on the validation set with the true labels.

After training, making predictions on new data does not require the labels. Create `minibatchqueue` object containing only the predictors of the test data:

- To ignore the labels for testing, set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessMiniBatchPredictors` function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCB'` (spatial, spatial, channel, batch).

```
numOutputs = 1;
mbqTest = minibatchqueue(augimdsValidation,numOutputs, ...
```

```
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatchPredictors, ...
    'MiniBatchFormat','SSCB');
```

Loop over the mini-batches and classify the images using `modelPredictions` function, listed at the end of the example.

```
predictions = modelPredictions(dlnet,mbqTest,classes);
```

Evaluate the classification accuracy.

```
YTest = imdsValidation.Labels;
accuracy = mean(predictions == YTest)
```

```
accuracy = 0.9530
```

Model Gradients Function

The `modelGradients` function takes a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX` with corresponding labels `Y` and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,Y)

[dlYPred,state] = forward(dlnet,dlX);

loss = crossentropy(dlYPred,Y);
gradients = dlgradient(loss,dlnet.Learnables);

loss = double(gather(extractdata(loss)));

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)

predictions = [];

while hasdata(mbq)

    dlXTest = next(mbq);
    dlYPred = predict(dlnet,dlXTest);

    YPred = onehotdecode(dlYPred,classes,1)';

    predictions = [predictions; YPred];
end

end
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors and labels using the following steps:

1. Preprocess the images using the `preprocessMiniBatchPredictors` function.
2. Extract the label data from the incoming cell array and concatenate into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate.
Y = cat(2,YCell{1:end});

% One-hot encode labels.
Y = onehotencode(Y,1);

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

end
```

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `forward` | `adamupdate` | `predict` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “List of Functions with dlarray Support” on page 18-423

Define Model Gradients Function for Custom Training Loop

When you train a deep learning model with a custom training loop, the software minimizes the loss with respect to the learnable parameters. To minimize the loss, the software uses the gradients of the loss with respect to the learnable parameters. To calculate these gradients using automatic differentiation, you must define a model gradients function.

For an example showing how to train deep learning model with a `dlnetwork` object, see “Train Network Using Custom Training Loop” on page 18-225. For an example showing how to training a deep learning model defined as a function, see “Train Network Using Model Function” on page 18-259.

Create Model Gradients Function for Model Defined as dlnetwork Object

If you have a deep learning model defined as a `dlnetwork` object, then create a model gradients function that takes the `dlnetwork` object as input.

For a model specified as a `dlnetwork` object, create a function of the form `gradients = modelGradients(dlnet,dlX,T)`, where `dlnet` is the network, `dlX` is the network input, `T` contains the targets, and `gradients` contains the returned gradients. Optionally, you can pass extra arguments to the gradients function (for example, if the loss function requires extra information), or return extra arguments (for example, metrics for plotting the training progress).

For example, this function returns the gradients and the cross entropy loss for the specified `dlnetwork` object `dlnet`, input data `dlX`, and targets `T`.

```
function [gradients, loss] = modelGradients(dlnet, dlX, T)

    % Forward data through the dlnetwork object.
    dlY = forward(dlnet,dlX);

    % Compute loss.
    loss = crossentropy(dlY,T);

    % Compute gradients.
    gradients = dlgradient(loss,dlnet);

end
```

Create Model Gradients Function for Model Defined as Function

If you have a deep learning model defined as a function, then create a model gradients function that takes the model learnable parameters as input.

For a model specified as a function, create a function of the form `gradients = modelGradients(parameters,dlX,T)`, where `parameters` contains the learnable parameters, `dlX` is the model input, `T` contains the targets, and `gradients` contains the returned gradients. Optionally, you can pass extra arguments to the gradients function (for example, if the loss function requires extra information), or return extra arguments (for example, metrics for plotting the training progress).

For example, this function returns the gradients and the cross-entropy loss for the deep learning model function `model` with the specified learnable parameters `parameters`, input data `dlX`, and targets `T`.

```
function [gradients, loss] = modelGradients(parameters, dlX, T)

    % Forward data through the model function.
    dlY = model(parameters,dlX);

    % Compute loss.
    loss = crossentropy(dlY,T);

    % Compute gradients.
    gradients = dlgradient(loss,parameters);

end
```

Evaluate Model Gradients Function

To evaluate the model gradients using automatic differentiation, use the `dlfeval` function, which evaluates a function with automatic differentiation enabled. For the first input of `dlfeval`, pass the model gradients function specified as a function handle. For the following inputs, pass the required variables for the model gradients function. For the outputs of the `dlfeval` function, specify the same outputs as the model gradients function.

For example, evaluate the model gradients function `modelGradients` with a `dlnetwork` object `dlnet`, input data `dlX`, and targets `T`, and return the model gradients and loss.

```
[gradients, loss] = dlfeval(@modelGradients,dlnet,dlX,T);
```

Similarly, evaluate the model gradients function `modelGradients` using a model function with learnable parameters specified by the structure `parameters`, input data `dlX`, and targets `T`, and return the model gradients and loss.

```
[gradients, loss] = dlfeval(@modelGradients,parameters,dlX,T);
```

Update Learnable Parameters Using Gradients

To update the learnable parameters using the gradients, you can use the following functions.

Function	Description
`adamupdate`	Update parameters using adaptive moment estimation (Adam)
`rmspropupdate`	Update parameters using root mean squared propagation (RMSProp)
`sgdmupdate`	Update parameters using stochastic gradient descent with momentum (SGDM)
`dlupdate`	Update parameters using custom function

For example, update the learnable parameters of a `dlnetwork` object `dlnet` using the `adamupdate` function.

```
[dlnet,trailingAvg,trailingAvgSq] = adamupdate(dlnet,gradients, ...
    trailingAvg,trailingAverageSq,iteration);
```

`gradients` is the output of the model gradients function, and `trailingAvg`, `trailingAvgSq`, and `iteration` are the hyperparameters required by the `adamupdate` function.

Similarly, update the learnable parameters for a model function `parameters` using the `adamupdate` function.

```
[parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
    trailingAvg,trailingAverageSq,iteration);
```

`gradients` is the output of the model gradients function, and `trailingAvg`, `trailingAvgSq`, and `iteration` are the hyperparameters required by the `adamupdate` function.

Use Model Gradients Function in Custom Training Loop

When training a deep learning model using a custom training loop, evaluate the model gradients and update the learnable parameters for each mini-batch.

This code snippet shows an example of using the `dlfeval` and `adamupdate` functions in a custom training loop.

```
iteration = 0;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Prepare mini-batch.
        % ...

        % Evaluate model gradients.
        [gradients, loss] = dlfeval(@modelGradients,dlnet,dlX,T);

        % Update learnable parameters.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAverageSq,iteration);

    end
end
```

For an example showing how to train a deep learning model with a `dlnetwork` object, see “Train Network Using Custom Training Loop” on page 18-225. For an example showing how to training a deep learning model defined as a function, see “Train Network Using Model Function” on page 18-259.

Debug Model Gradients Functions

If the implementation of the model gradients function has an issue, then the call to `dlfeval` can throw an error. Sometimes, when you use the `dlfeval` function, it is not clear which line of code is throwing the error. To help locate the error, you can try the following.

Call Model Gradients Function Directly

Try calling the model gradients function directly (that is, without using the `dlfeval` function) with generated inputs of the expected sizes. If any of the lines of code throw an error, then the error

message provides extra detail. Note that when you do not use the `dlfeval` function, any calls to the `dlgradient` function throw an error.

```
% Generate image input data.
X = rand([28 28 1 100],'single');
dlX = dlarray(dlX);

% Generate one-hot encoded target data.
T = repmat(eye(10,'single'),[1 10]);

[gradients, loss] = modelGradients(dlnet,dlX,T);
```

Run Model Gradients Code Manually

Run the code inside the model gradients function manually with generated inputs of the expected sizes and inspect the output and any thrown error messages.

For example, consider the following model gradients function.

```
function [gradients, loss] = modelGradients(dlnet, dlX, T)

    % Forward data through the dlnetwork object.
    dlY = forward(dlnet,dlX);

    % Compute loss.
    loss = crossentropy(dlY,T);

    % Compute gradients.
    gradients = dlgradient(loss,dlnet);

end
```

Check the model gradients function by running the following code.

```
% Generate image input data.
X = rand([28 28 1 100],'single');
dlX = dlarray(dlX);

% Generate one-hot encoded target data.
T = repmat(eye(10,'single'),[1 10]);

% Check forward pass.
dlY = forward(dlnet,dlX);

% Check loss calculation.
loss = crossentropy(dlX,T)
```

See Also

More About

- “Train Network Using Custom Training Loop” on page 18-225
- “Train Network Using Model Function” on page 18-259
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Specify Training Options in Custom Training Loop” on page 18-216

- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Make Predictions Using dlnetwork Object” on page 18-255
- “List of Functions with dlarray Support” on page 18-423

Update Batch Normalization Statistics in Custom Training Loop

This example shows how to update the network state in a custom training loop.

A batch normalization layer normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers.

During training, batch normalization layers first normalize the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the layer shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

When network training finishes, batch normalization layers calculate the mean and variance over the full training set and stores the values in the `TrainedMean` and `TrainedVariance` properties. When you use a trained network to make predictions on new images, the batch normalization layers use the trained mean and variance instead of the mini-batch mean and variance to normalize the activations.

To compute the data set statistics, batch normalization layers keep track of the mini-batch statistics by using a continually updating state. If you are implementing a custom training loop, then you must update the network state between mini-batches.

Load Training Data

The `digitTrain4DArrayData` function loads images of handwritten digits and their digit labels. Create an `arrayDatastore` object for the images and the angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names.

```
[XTrain,YTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsYTrain = arrayDatastore(YTrain);

dsTrain = combine(dsXTrain,dsYTrain);

classNames = categories(YTrain);
numClasses = numel(classNames);
```

Define Network

Define the network and specify the average image using the `'Mean'` option in the image input layer.

```
layers = [
    imageInputLayer([28 28 1], 'Name', 'input', 'Mean', mean(XTrain,4))
    convolution2dLayer(5, 20, 'Name', 'conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name', 'relu1')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name', 'relu2')
    convolution2dLayer(3, 20, 'Padding', 1, 'Name', 'conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(numClasses, 'Name', 'fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph)

dlnet =
  dlnetwork with properties:

         Layers: [12×1 nnet.cnn.layer.Layer]
    Connections: [11×2 table]
     Learnables: [14×3 table]
          State: [6×3 table]
     InputNames: {'input'}
    OutputNames: {'softmax'}
```

View the network state. Each batch normalization layer has a `TrainedMean` parameter and a `TrainedVariance` parameter containing the data set mean and variance, respectively.

```
dlnet.State
```

```
ans=6×3 table
    Layer        Parameter             Value
    _____    ________________    _______________

    "bn1"    "TrainedMean"       {1×1×20 single}
    "bn1"    "TrainedVariance"   {1×1×20 single}
    "bn2"    "TrainedMean"       {1×1×20 single}
    "bn2"    "TrainedVariance"   {1×1×20 single}
    "bn3"    "TrainedMean"       {1×1×20 single}
    "bn3"    "TrainedVariance"   {1×1×20 single}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, which takes as input a `dlnetwork` object `dlnet`, and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the corresponding loss.

Specify Training Options

Train for five epochs using a mini-batch size of 128. For the SGDM optimization, specify a learning rate of 0.01 and a momentum of 0.9.

```
numEpochs = 5;
miniBatchSize = 128;

learnRate = 0.01;
momentum = 0.9;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

Train Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.

- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Train the model using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. At the end of each iteration, display the training progress. For each mini-batch:

- Evaluate the model gradients, state, and loss using `dlfeval` and the `modelGradients` function and update the network state.
- Update the network parameters using the `sgdmupdate` function.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

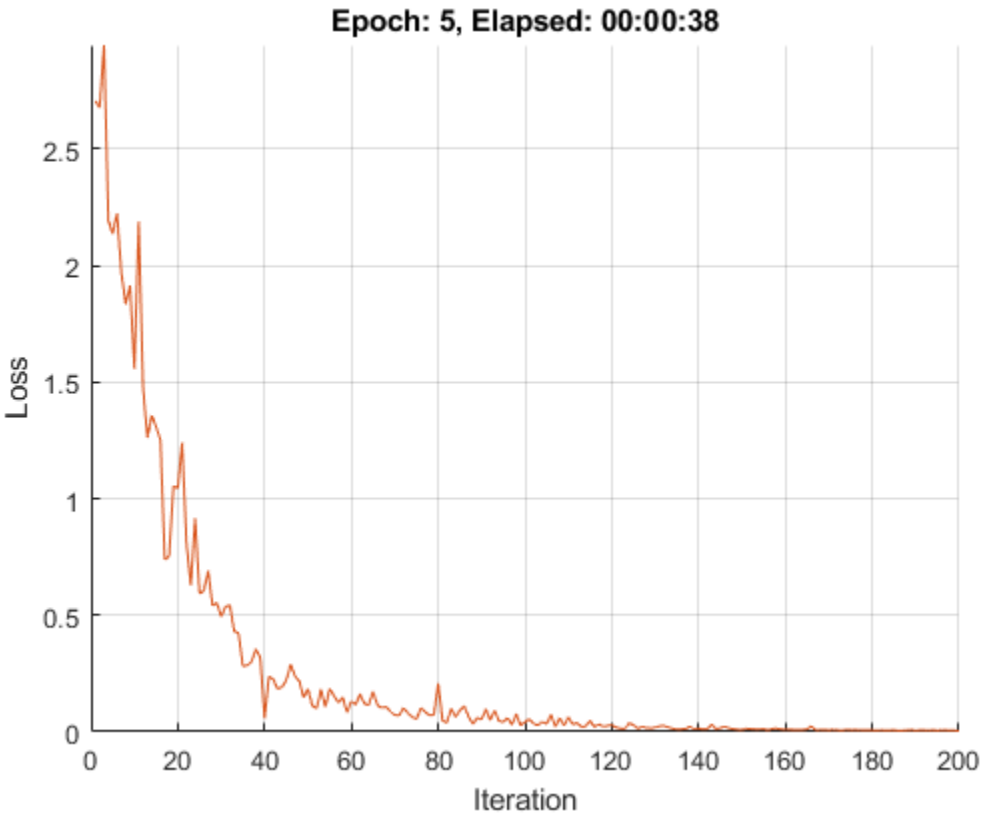
Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Train the network.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)

        iteration = iteration + 1;

        % Read mini-batch of data and convert the labels to dummy
        % variables.
        [dlX,dlY] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function and update the network state.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,dlY);
```

```
        dlnet.State = state;

        % Update the network parameters using the SGDM optimizer.
        [dlnet, velocity] = sgdmupdate(dlnet, gradients, velocity, learnRate, momentum);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels and angles. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
[XTest,YTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsYTest = arrayDatastore(YTest);

dsTest = combine(dsXTest,dsYTest);
```

```
mbqTest = minibatchqueue(dsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Classify the images using the `modelPredictions` function, listed at the end of the example. The function returns the predicted classes and the comparison with the true values.

```
[classesPredictions,classCorr] = modelPredictions(dlnet,mbqTest,classNames);
```

Evaluate the classification accuracy.

```
accuracy = mean(classCorr)
```

```
accuracy = 0.9946
```

Model Gradients Function

The `modelGradients` function takes as input a `dlnetwork` object `dlnet` and a mini-batch of input data `dlX` with corresponding labels `Y`, and returns the gradients of the loss with respect to the learnable parameters in `dlnet`, the network state, and the loss. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,state,loss] = modelGradients(dlnet,dlX,Y)

    [dlYPred,state] = forward(dlnet,dlX);

    loss = crossentropy(dlYPred,Y);
    gradients = dlgradient(loss,dlnet.Learnables);

end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and computes the model predictions by iterating all data in the `minibatchqueue`. The function uses the `onehotdecode` function to find the predicted class with the highest score and then compares the prediction with the true class. The function returns the predictions and a vector of ones and zeros that represents correct and incorrect predictions.

```
function [classesPredictions,classCorr] = modelPredictions(dlnet,mbq,classes)

    classesPredictions = [];
    classCorr = [];

    while hasdata(mbq)
        [dlX,dlY] = next(mbq);

        % Make predictions using the model function.
        dlYPred = predict(dlnet,dlX);

        % Determine predicted classes.
        YPredBatch = onehotdecode(dlYPred,classes,1);
        classesPredictions = [classesPredictions YPredBatch];

        % Compare predicted and true classes
        Y = onehotdecode(dlY,classes,1);
        classCorr = [classCorr YPredBatch == Y];
```

```
    end

end
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label data from the incoming cell arrays and concatenate into a categorical array along the second dimension..
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)

    % Extract image data from cell and concatenate
    X = cat(4,XCell{:});
    % Extract label data from cell and concatenate
    Y = cat(2,YCell{:});

    % One-hot encode labels
    Y = onehotencode(Y,1);

end
```

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `forward` | `adamupdate` | `predict` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

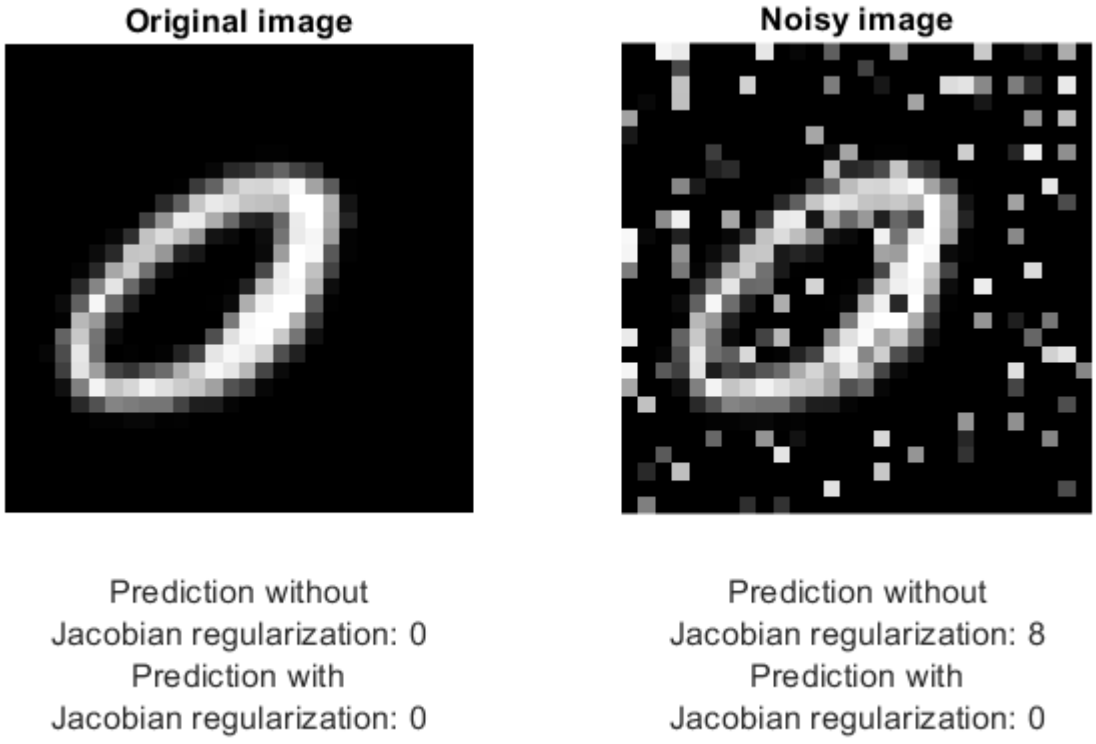
- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Automatic Differentiation Background” on page 18-200

Train Robust Deep Learning Network with Jacobian Regularization

This example shows how to train a neural network that is robust to adversarial examples using a Jacobian regularization scheme [1].

Neural networks can be susceptible to a phenomenon known as *adversarial examples* [2], where very small changes to an input can cause it to be misclassified. These changes are often imperceptible to humans. Second-order methods such as Jacobian regularization have been shown to help increase the robustness of networks against adversaries [3].

For example, consider the figure below. On the left is an image of a zero, and on the right is the same image with white noise added to create an adversarial example. A network trained without Jacobian regularization classifies the original image of a zero correctly but misclassifies the adversarial example. In contrast, a network trained with Jacobian regularization classifies both the original and the noisy image correctly.



This example shows how to:

1. Train a robust image classification network using a Jacobian regularization scheme.
2. Compare its predictions to a network trained without Jacobian regularization.

Load Training Data

The `digitTrain4DArrayData` function loads the images and their digit labels. Create an `arrayDatastore` object for the images and the labels, and then use the `combine` function to make a single datastore that contains all of the training data.

```
[XTrain,YTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsYTrain = arrayDatastore(YTrain);

dsTrain = combine(dsXTrain,dsYTrain);
```

Determine the number of classes in the training data.

```
classes = categories(YTrain);
numClasses = numel(classes);
```

Next, apply noise to the training images to create adversarial examples. Compare images from the training data with no noise and with noise affecting 10% of the pixels.

Select 16 images at random.

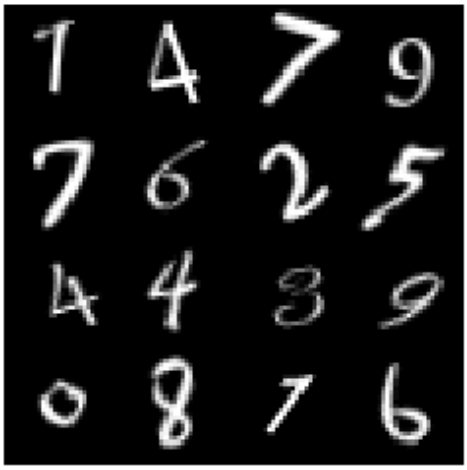
```
numImages = size(XTrain,4);
randompick = randperm(numImages,16);
XOriginal = XTrain(:,:,:,randompick);
```

Create the noisy images by setting a proportion of the pixels to a random grayscale value.

```
noiseProp = 0.1;
noise = rand(size(XOriginal));
idx = rand(size(XOriginal)) < noiseProp;

XNoisy = XOriginal;
XNoisy(idx) = noise(idx);
```

Plot the original images next to the noisy images.

```
I1 = imtile(XOriginal);
I2 = imtile(XNoisy);

figure;
subplot(1,2,1)
imshow(I1)
subplot(1,2,2)
imshow(I2)
```



Define Network

Define the architecture of the network.

Specify an image input layer of the same size as the training images.

```
imageInputSize = size(XTrain, 1:3)
```

```
imageInputSize = 1×3

    28    28     1
```

```
layers = [
    imageInputLayer(imageInputSize,'Name','input','Mean',mean(XTrain,4))
    convolution2dLayer(5,20,'Name','conv1')
    batchNormalizationLayer('Name','bn1')
    reluLayer('Name','relu1')
    convolution2dLayer(3,20,'Padding',1,'Name','conv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
```

```
    convolution2dLayer(3,20,'Padding',1,'Name','conv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(10,'Name','fc')
    softmaxLayer('Name','softmax')];
lgraph = layerGraph(layers);
```

Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph);
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, which takes a `dlnetwork` object, and a mini-batch of input data with corresponding labels, and returns the gradients of the loss with respect to the learnable parameters in the network, the state of the network, and the loss.

Specify Training Options

Train for 15 epochs with a mini-batch size of 32.

```
numEpochs = 15;
miniBatchSize = 32;
```

Specify the options for SGDM optimization. Specify a learning rate of 0.01 and a momentum of 0.9.

```
learningRate = 0.01;
momentum = 0.9;
```

The Jacobian regularization $\lambda_{JR}$is a hyperparameter that controls the effect of the Jacobian regularization term on the training of the network. If the coefficient is too large, then the cross-entropy term is not effectively minimized and the accuracy of the network classification is poor. If the coefficient is too small, the trained network does not have the expected robustness to white noise. For example, choose $\lambda_{JR} = 1$.

```
jacobianRegularizationCoefficient = 1;
```

Train Model

Create a `minibatchqueue` object that processes and manages mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to convert the labels to one-hot encoded variables.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.
- Train on a GPU if one is available. If a GPU is available, the `minibatchqueue` object converts each output to a `gpuArray` by default. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'PartialMiniBatch',"discard",...
    'MiniBatchFormat',{'SSCB',''});
```

Initialize the training progress plot.

```
figure;
lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Initialize the velocity parameter for the SGDM solver.

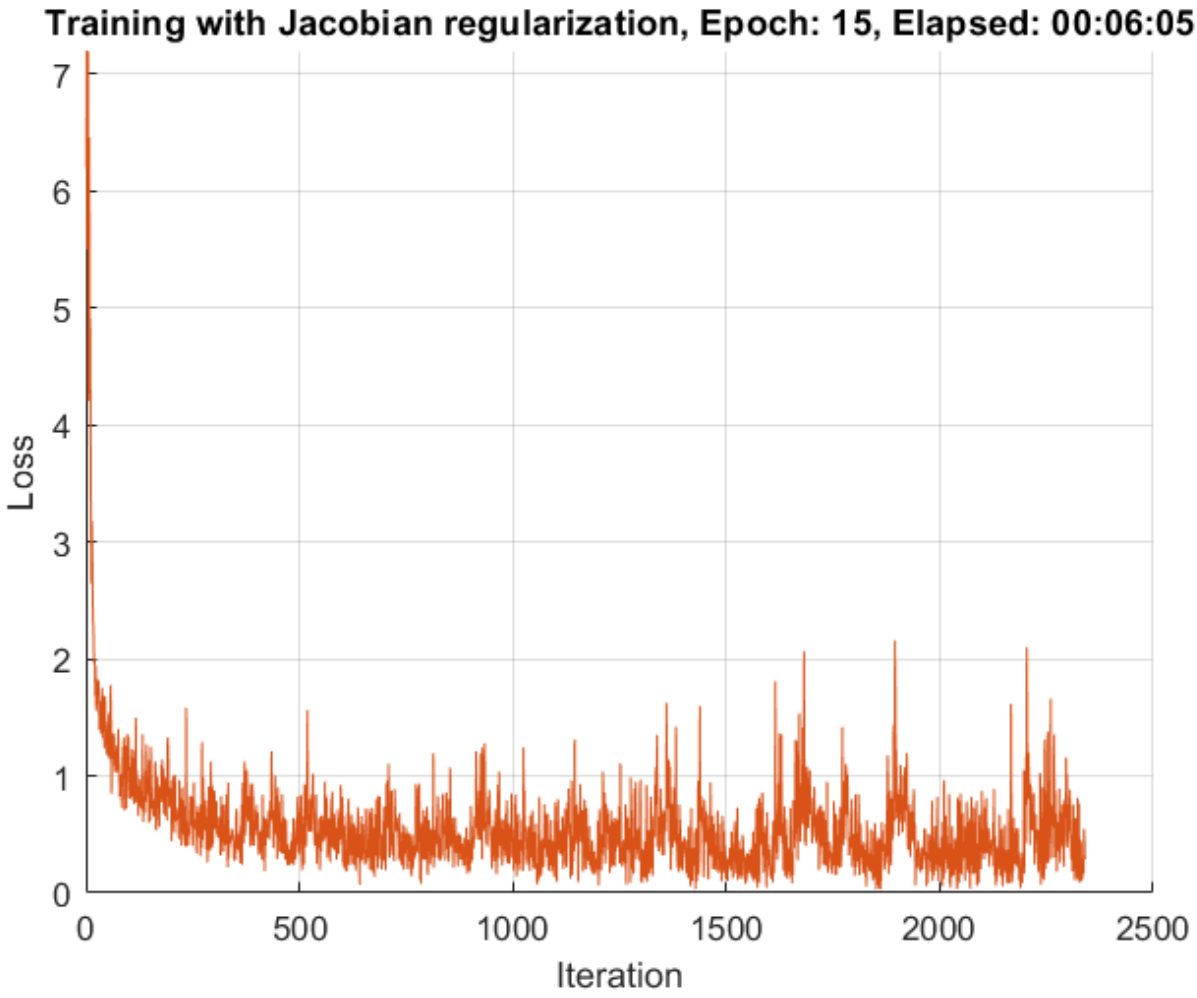
```
velocity = [];
```

Train the network using a custom training loop.

For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients, state, and loss using the `dlfeval` and `modelGradients` functions and update the network state.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Reset and shuffle mini-batch queue.
    shuffle(mbq);

    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX, dlY] = next(mbq);

        % Evaluate the model gradients and the network state using
        % dlfeval and the modelGradients function listed at the end of the example.
        [gradTotalLoss, state, totalLoss] = dlfeval(@modelGradients, dlnet, dlX, dlY, ...
            miniBatchSize, jacobianRegularizationCoefficient);
        dlnet.State = state;

        % Update the network parameters.
        [dlnet, velocity] = sgdmupdate(dlnet,gradTotalLoss,velocity,learningRate,momentum);

        % Plot the training progress.
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        addpoints(lineLossTrain,iteration,double(gather(extractdata(totalLoss))))
        title("Training with Jacobian regularization" + ", Epoch: " + epoch + ", Elapsed: " + st
        drawnow

    end
end
```



Load Reference Network

Load a reference network, with the same layers, but trained without Jacobian regularization.

```
dlnetReference = load("dlnetWithoutJacobian.mat").dlnetReference;
```

Test Model

Load the test data for a comparison test between the network trained with Jacobian regularization and the reference network.

```
[XTest, YTest] = digitTest4DArrayData;
classes = categories(YTest);
```

Pass through test images that the networks have not seen before. With each pass, add noise affecting 0% to 50% of the pixels in increments of 5%.

```
% Initialize test parameters and arrays.
noiseProps = 0:0.05:0.5;

% Prepare arguments for mini-batch queue.
```

```
dsYTest = arrayDatastore(YTest);
miniBatchSize = 5000;

for i = 1:numel(noiseProps)
    % Load the noise proportion.
    noiseProp = noiseProps(i);
    fprintf("Testing robustness to noise proportion %1.2g\n", noiseProp)

    % Set a proportion of the pixels to a random grayscale value.
    noise = rand(size(XTest));
    idx = rand(size(XTest)) < noiseProp;
    XNoisy = XTest;
    XNoisy(idx) = noise(idx);

    % Prepare mini-batch queue with noisy test data.
    dsXTest = arrayDatastore(XNoisy,'IterationDimension',4);
    dsTest = combine(dsXTest,dsYTest);
    mbq = minibatchqueue(dsTest,...
        'MiniBatchSize',miniBatchSize,...
        'MiniBatchFcn', @preprocessMiniBatch,...
        'MiniBatchFormat',{'SSCB',''});

    % Loop over batches to find predicted classifications.
    while hasdata(mbq)
        [dlXNoisy, dlY] = next(mbq);

        % Classify noisy data with the robust network.
        dlYPredNoisy = predict(dlnet, dlXNoisy);

        % Convert the predictions into labels.
        YPred = onehotdecode(dlYPredNoisy, classes, 1)';
        YTestBatch = onehotdecode(dlY, classes, 1)';

        % Evaluate accuracy of predictions.
        accuracyRobust(i) =  mean(YPred == YTestBatch);

        % Classify noisy data with reference network.
        dlYPredNoisy = predict(dlnetReference, dlXNoisy);

        % Convert the predictions into labels.
        YPred = onehotdecode(dlYPredNoisy, classes, 1)';

        % Evaluate accuracy of predictions.
        accuracyReference(i) =  mean(YPred == YTestBatch);
    end

end
```

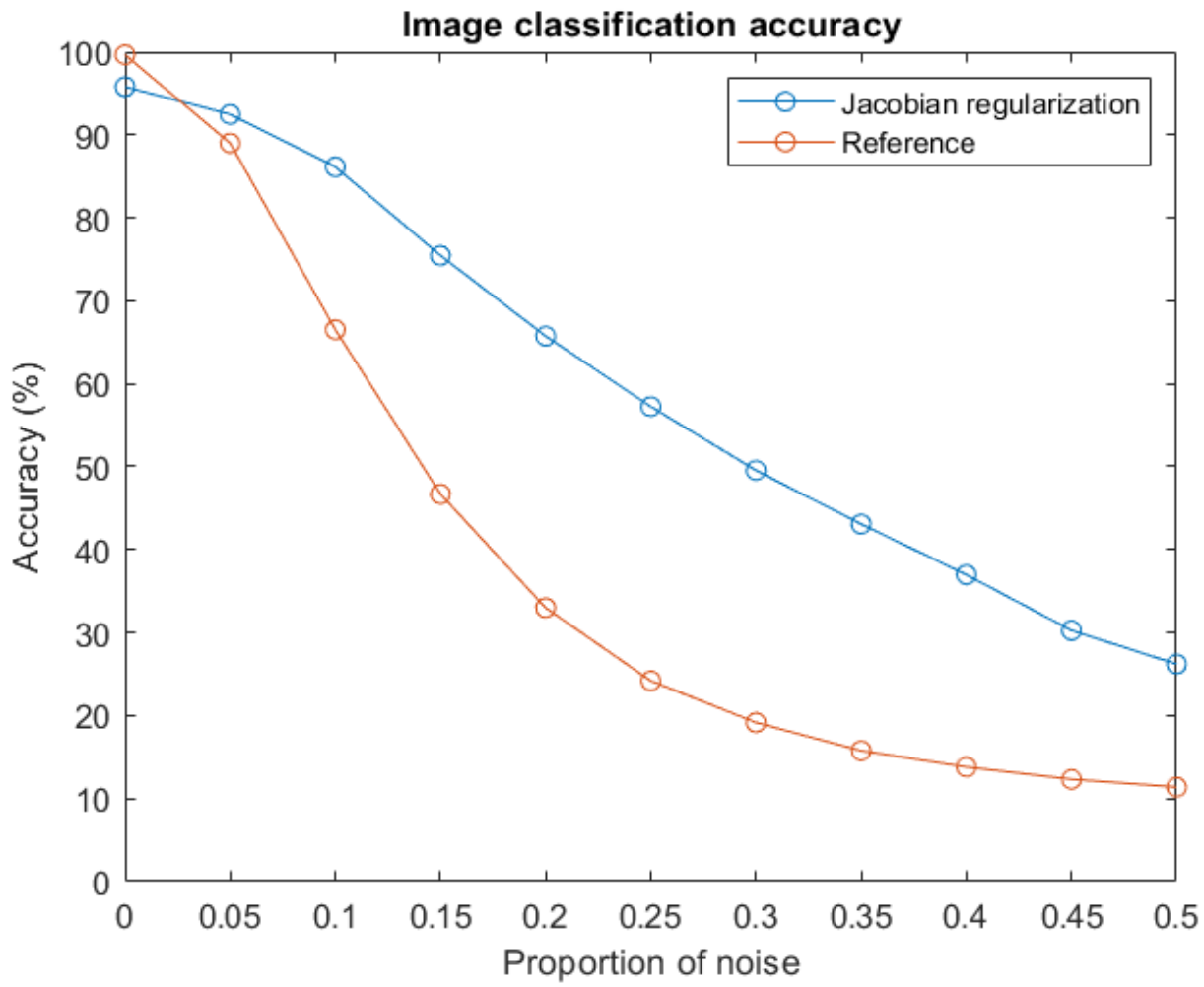
```
Testing robustness to noise proportion 0
Testing robustness to noise proportion 0.05
Testing robustness to noise proportion 0.1
Testing robustness to noise proportion 0.15
Testing robustness to noise proportion 0.2
Testing robustness to noise proportion 0.25
Testing robustness to noise proportion 0.3
Testing robustness to noise proportion 0.35
Testing robustness to noise proportion 0.4
```

```
Testing robustness to noise proportion 0.45
Testing robustness to noise proportion 0.5
```

Plot the results of the percentage accuracy of both networks against the proportion of the white noise.

Note that the network trained with Jacobian regularization has a slightly lower accuracy when the noise proportion is equal to 0 but achieves higher accuracy than the reference network when noise is added to the images.

```
x = noiseProps';

figure;

plot(x,accuracyRobust*100, '-o', x,accuracyReference*100, '-o')
xlabel('Proportion of noise')
ylabel('Accuracy (%)')
xlim([0,0.5]);
ylim([0,100]);
title('Image classification accuracy')
legend('Jacobian regularization', 'Reference');
```



Test Specific Example

Add noise affecting 15% of the pixels to the first test image, which contains the number 0. Plot both the original image and the image perturbed by the white noise. Use the network trained with Jacobian regularization and the reference network to classify the image.

```
% Choose test image
testchoice = 1;

% Add noise, with a proportion of 0.15, to the first image.
noise = rand(size(XTest(:,:,:,testchoice)));
idx = rand(size(XTest(:,:,:,testchoice))) < 0.15;
XNoisy = XTest(:,:,:,testchoice);
XNoisy(idx) = noise(idx);

% Convert to dlarray.
dlXTest  = dlarray(XTest(:,:,:,testchoice),'SSCB');
dlXNoisy = dlarray(XNoisy,'SSCB');

% Print true number classification
disp("True digit label: " + char(YTest(testchoice)));
```

```
True digit label: 0
```

Classify the original image by using the network trained with Jacobian regularization.

```
dlYPredTestJR = predict(dlnet, dlXTest);
YPredTestJR = onehotdecode(dlYPredTestJR, classes, 1)';
disp("Robust network classification of original image: " + char(YPredTestJR));
```

```
Robust network classification of original image: 0
```

Classify the noisy image by using the network trained with Jacobian regularization.

```
dlYPredNoisyJR = predict(dlnet, dlXNoisy);
YPredNoisyJR = onehotdecode(dlYPredNoisyJR, classes, 1)';
disp("Robust network classification of noisy image: " + char(YPredNoisyJR));
```

```
Robust network classification of noisy image: 0
```

Classify the original image by using the network trained without Jacobian regularization.

```
dlYPredTest = predict(dlnetReference, dlXTest);
YPredTestR = onehotdecode(dlYPredTest, classes, 1)';
disp("Reference network classification of original image: " + char(YPredTestR));
```

```
Reference network classification of original image: 0
```

Classify the noisy image by using the network trained without Jacobian regularization.

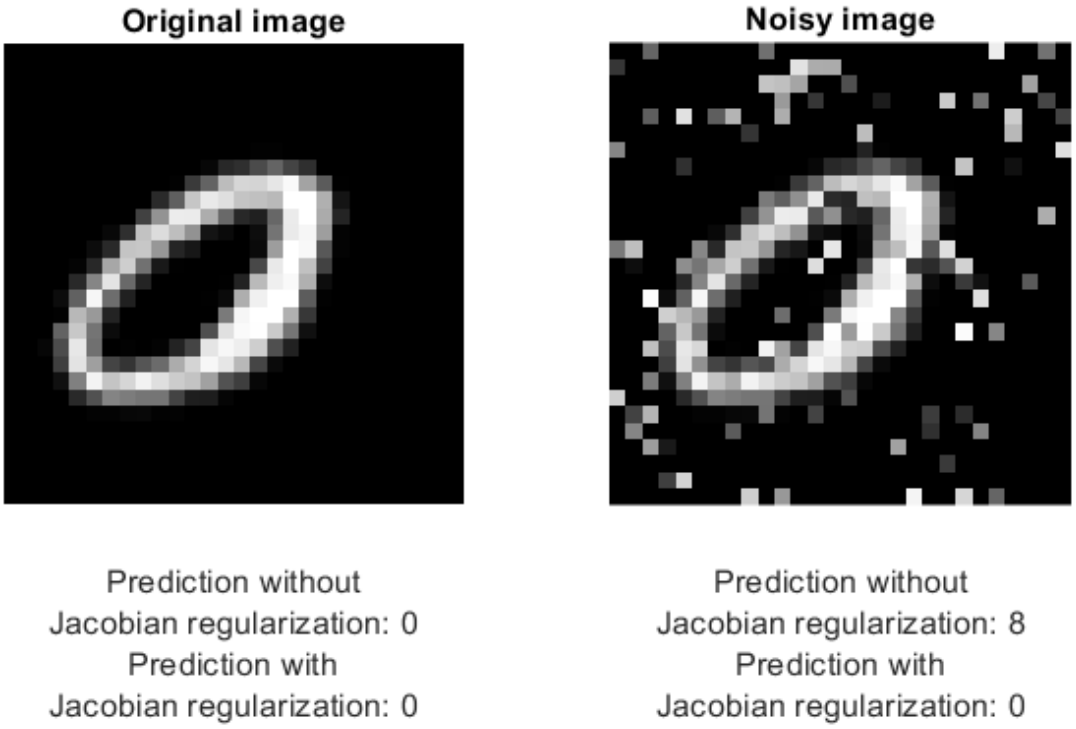
```
dlYPredNoisy = predict(dlnetReference, dlXNoisy);
YPredNoisyR = onehotdecode(dlYPredNoisy, classes, 1)';
disp("Reference network classification of noisy image: " + char(YPredNoisyR));
```

```
Reference network classification of noisy image: 8
```

Plot the original and noisy images and display the predictions given by each network.

```
figure;
I1 = XTest(:,:,:,testchoice);
```

```
subplot(1,2,1)
imshow(I1)
title('Original image')
xlabel({"Prediction without"; "Jacobian regularization: " + char(YPredTestR);...
    "Prediction with"; "Jacobian regularization: " + char(YPredTestJR)})
I2 = XNoisy;
subplot(1,2,2)
imshow(I2)
title('Noisy image')
xlabel({"Prediction without"; "Jacobian regularization: " + char(YPredNoisyR);...
    "Prediction with"; "Jacobian regularization: " + char(YPredNoisyJR)})
```



Model Gradients Function

The goal of Jacobian regularization is to penalize large changes of the prediction y with respect to small changes in the input x. Doing so makes the network more robust to input data polluted by noise. The Jacobian J encodes the change of prediction with respect to the input by containing the partial derivatives of y with respect to x.

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Jacobian regularization is achieved by adding the Frobenius norm of the Jacobian to the loss function that is subsequently minimized when you train the network. However, the Jacobian can be expensive to compute, requiring m backward passes through the network, where m is the size of the output y. Therefore, instead of computing the full Jacobian, an approximation to the Frobenius norm of the Jacobian (JR) is computed as follows [4]:

$$\| J \|_F^2 = \mathrm{tr}\left(J J^\top\right) = E_{v \sim N(0, I_m)} v^\top J J^\top v \approx \frac{1}{m_{\mathrm{proj}}} \sum_{k=1}^{m_{\mathrm{proj}}} \left\| v_k^\top J \right\|_2^2 = \frac{1}{m_{\mathrm{proj}}} \sum_{k=1}^{m_{\mathrm{proj}}} \left\| \nabla_x \left[v_k^\top y \right] \right\|_2^2.$$

where $v_k \sim N(0, I_m)$ is a draw from the standard Normal distribution and I_m is is the m-by-m identity matrix. This can be implemented as follows:

JR = 0

Choose a mini-batch size B

For $i = 1, \ldots, m_{\mathrm{proj}}$

1. Sample a random vector $v \sim N(0, I_m)$
2. Normalize the random vector $v = \frac{v}{\|v\|_2}$
3. Compute the derivative $\nabla_x\left(v^\top y\right)$
4. $\mathrm{JR} = \mathrm{JR} + \frac{m \left\| \nabla_x\left(v^\top y\right) \right\|_2^2}{B\, m_{\mathrm{proj}}}$

The gradient of the vector inner product requires one backward pass to compute. So, this approximation requires only m_{proj} backward passes to compute, and in practice, $m_{\mathrm{proj}} = 1$.

In this example, the cross-entropy between the predicted classification y and the true classification z is used, resulting in the loss function

$$\mathrm{loss} = \mathrm{crossentropy} + \frac{\lambda_{\mathrm{JR}}}{2} \mathrm{JR}$$

where λ_{JR} is the Jacobian regularization coefficient. The approximation of the Frobenius norm of the Jacobian requires taking gradients with respect to x, and the training phase requires taking gradients of the loss with respect to the parameters. These calculations require support for second-order automatic differentiation.

The `modelGradients` function is used during the training of the network. It takes as input the network, the input data `dlX`, their respective classifications `dlY`, the mini-batch size `miniBatchSize`, and the Jacobian regularization coefficient `jacobianRegularizationCoefficient`. The function returns the gradient of the loss with respect to the network parameters `gradTotalLoss`, the state of the network `state`, and the total loss `totalLoss.` To compute the approximation to the norm of the Jacobian, a derivative of a vector-

vector dot product is taken. Since a second order derivative is necessary to compute the gradient of the loss function with respect to the network parameters, you must set the option `'EnableHigherDerivatives'` to `'true'` when calling the function `dlgradient`.

```
function [gradTotalLoss, state, totalLoss] = modelGradients(net, dlX, dlY, miniBatchSize, jacobia

% Find prediction and loss.
[dlZ,state] = forward(net, dlX);
loss = crossentropy(dlZ, dlY);

numClasses = size(dlZ,1);
numProjections = 1;
regularizationTerm = 0;

% Compute Jacobian term and its gradient.
for i = 1:numProjections

    % Sample a matrix whose elements are drawn from the standard Normal distribution.
    rndarray = randn(numClasses, miniBatchSize);

    % Normalize the columns of the random matrix.
    rndarray = normc(rndarray);

    % Compute the vector-vector product.
    vectorproduct = rndarray(:)' * dlZ(:);

    % Compute the gradient of the vector-vector product. Since another
    % derivative will be taken, set EnableHigherDerivatives to true.
    vectorJacobianTerm = dlgradient(vectorproduct, dlX, 'EnableHigherDerivatives', true);

    % Multiply by necessary constants to obtain approximation of the
    % Frobenius norm of the Jacobian.
    regularizationTerm = regularizationTerm + numClasses*sum(vectorJacobianTerm.^2,'all') /(numP
end
totalLoss = loss + jacobianRegularizationCoefficient/2 * regularizationTerm;

% Calculate the gradient of the loss.
gradTotalLoss = dlgradient(totalLoss, net.Learnables);
end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label data from the incoming cell array and concatenate the data into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding the labels into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X, Y] = preprocessMiniBatch(XCell,YCell)
    % Extract image data from cell and concatenate
    X = cat(4,XCell{1:end});

    % Extract label data from cell and concatenate
```

```
    Y = cat(2,YCell{1:end});

    % One-hot encode labels
    Y = onehotencode(Y,1);
end
```

References

1. Hoffman, Judy, Daniel A. Roberts, and Sho Yaida. “Robust Learning with Jacobian Regularization.” Preprint, submitted August 7, 2019. https://arxiv.org/abs/1908.02729.
2. Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. “Intriguing Properties of Neural Networks.” Preprint, submitted February 19, 2014. http://arxiv.org/abs/1312.6199.
3. Ma, Avery, Fartash Faghri, and Amir-Massoud Farahmand. “Adversarial Robustness through Regularization: A Second-Order Approach.” Preprint, submitted, April 3, 2020. http://arxiv.org/abs/2004.01832.
4. Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples.” Preprint, submitted, March 20, 2015. http://arxiv.org/abs/1412.6572.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork` | `predict` | `minibatchqueue` | `onehotencode` | `onehotdecode` | `sgdmupdate`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Automatic Differentiation Background” on page 18-200
- “Generate Untargeted and Targeted Adversarial Examples for Image Classification” on page 5-76
- “Train Image Classification Network Robust to Adversarial Examples” on page 5-83

Make Predictions Using dlnetwork Object

This example shows how to make predictions using a `dlnetwork` object by splitting data into mini-batches.

For large data sets, or when predicting on hardware with limited memory, make predictions by splitting the data into mini-batches. When making predictions with `SeriesNetwork` or `DAGNetwork` objects, the `predict` function automatically splits the input data into mini-batches. For `dlnetwork` objects, you must split the data into mini-batches manually.

Load `dlnetwork` Object

Load a trained `dlnetwork` object and the corresponding classes.

```
s = load("digitsCustom.mat");
dlnet = s.dlnet;
classes = s.classes;
```

Load Data for Prediction

Load the digits data for prediction.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true);
```

Make Predictions

Loop over the mini-batches of the test data and make predictions using a custom prediction loop.

Use `minibatchqueue` to process and manage the mini-batches of images. Specify a mini-batch size of 128. Set the read size property of the image datastore to the mini-batch size.

For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to concatenate the data into a batch and normalize the images.
- Format the images with the dimensions `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Make predictions on a GPU if one is available. By default, the `minibatchqueue` object converts the output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

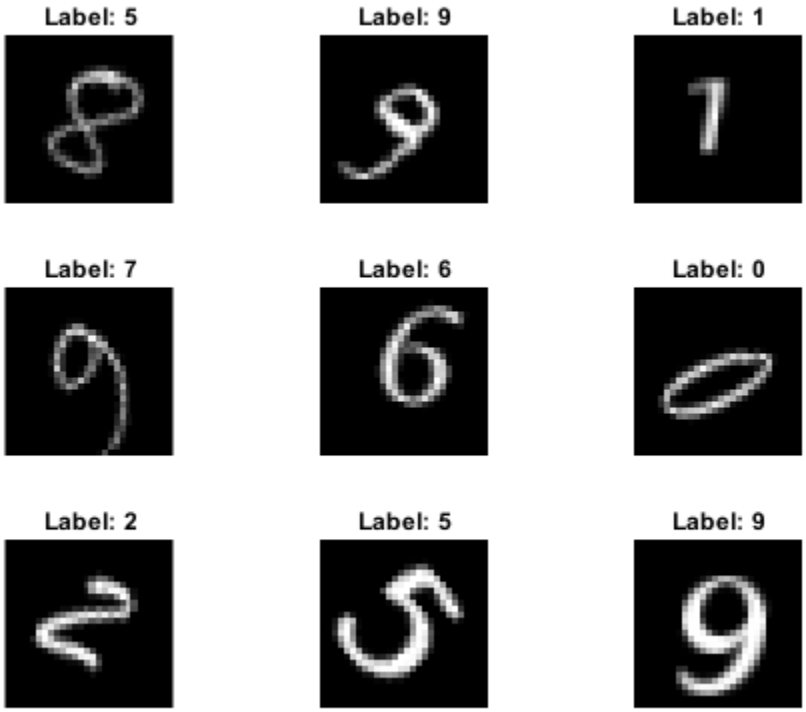
```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;

mbq = minibatchqueue(imds,...
    "MiniBatchSize",miniBatchSize,...
    "MiniBatchFcn", @preprocessMiniBatch,...
    "MiniBatchFormat","SSCB");
```

Loop over the minibatches of data and make predictions using the `predict` function. Use the `onehotdecode` function to determing the class labels. Store the predicted class labels.

```
numObservations = numel(imds.Files);
YPred = strings(1,numObservations);

predictions = [];

% Loop over mini-batches.
while hasdata(mbq)

    % Read mini-batch of data.
    dlX = next(mbq);

    % Make predictions using the predict function.
    dlYPred = predict(dlnet,dlX);

    % Determine corresponding classes.
    predBatch = onehotdecode(dlYPred,classes,1);
    predictions = [predictions predBatch];

end
```

Visualize some of the predictions.

```
idx = randperm(numObservations,9);

figure
for i = 1:9
    subplot(3,3,i)
    I = imread(imds.Files{idx(i)});
    label = predictions(idx(i));
    imshow(I)
    title("Label: " + string(label))

end
```



Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the data from the incoming cell array and concatenate into a numeric array. Concatenating over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Normalize the pixel values between `0` and `1`.

```
function X = preprocessMiniBatch(data)
    % Extract image data from cell and concatenate
    X = cat(4,data{:});

    % Normalize the images.
    X = X/255;
end
```

See Also

`dlarray` | `dlnetwork` | `predict` | `minibatchqueue` | `onehotdecode`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Train Network Using Custom Training Loop” on page 18-225

- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67
- “Automatic Differentiation Background” on page 18-200

Train Network Using Model Function

This example shows how to create and train a deep learning network by using functions rather than a layer graph or a `dlnetwork`. The advantage of using functions is the flexibility to describe a wide variety of networks. The disadvantage is that you must complete more steps and prepare your data carefully. This example uses images of handwritten digits, with the dual objectives of classifying the digits and determining the angle of each digit from the vertical.

Load Training Data

The `digitTrain4DArrayData` function loads the images, their digit labels, and their angles of rotation from the vertical. Create `arrayDatastore` objects for the images, labels, and angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and number of nondiscrete responses.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsYTrain = arrayDatastore(YTrain);
dsAnglesTrain = arrayDatastore(anglesTrain);

dsTrain = combine(dsXTrain,dsYTrain,dsAnglesTrain);

classNames = categories(YTrain);
numClasses = numel(classNames);
numResponses = size(anglesTrain,2);
numObservations = numel(YTrain);
```

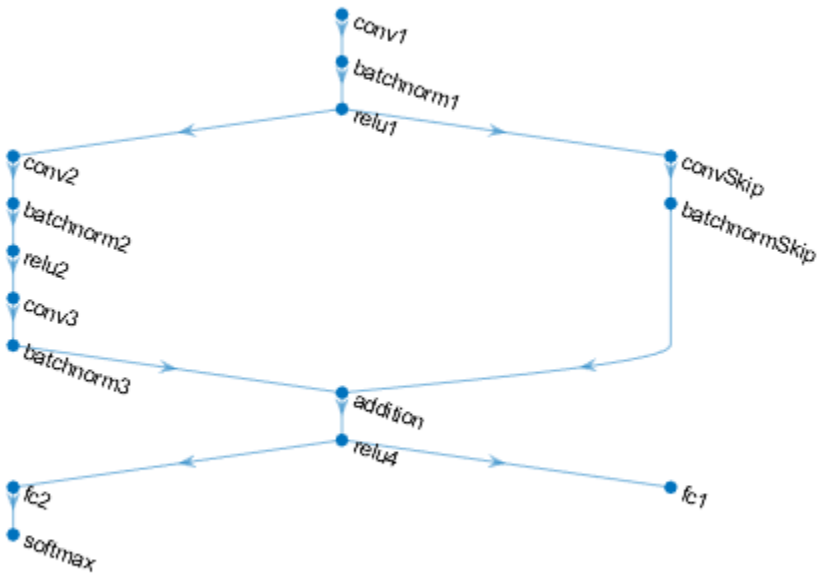
View some images from the training data.

```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

Define Deep Learning Model

Define the following network that predicts both labels and angles of rotation.

- A convolution-batchnorm-ReLU block with 16 5-by-5 filters.
- A branch of two convolution-batchnorm blocks each with 32 3-by-3 filters with a ReLU operation between
- A skip connection with a convolution-batchnorm block with 32 1-by-1 convolutions.
- Combine both branches using addition followed by a ReLU operation
- For the regression output, a branch with a fully connected operation of size 1 (the number of responses).
- For classification output, a branch with a fully connected operation of size 10 (the number of classes) and a softmax operation.



Define and Initialize Model Parameters and State

Define the parameters for each of the operations and include them in a struct. Use the format `parameters.OperationName.ParameterName` where `parameters` is the struct, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

Create a struct `parameters` containing the model parameters. Initialize the learnable layer weights and biases using the `initializeGlorot` and `initializeZeros` example functions, respectively. Initialize the batch normalization offset and scale parameters with the `initializeZeros` and `initializeOnes` example functions, respectively.

To perform training and inference using batch normalization layers, you must also manage the network state. Before prediction, you must specify the dataset mean and variance derived from the training data. Create a struct `state` containing the state parameters. The batch normalization statistics must not be `dlarray` objects. Initialize the batch normalization trained mean and trained variance states using the `zeros` and `ones` functions, respectively.

The initialization example functions are attached to this example as supporting files.

Initialize the parameters for the first convolutional layer.

```
filterSize = [5 5];
numChannels = 1;
numFilters = 16;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv1.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the first batch normalization layer.

```
parameters.batchnorm1.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm1.Scale = initializeOnes([numFilters 1]);
state.batchnorm1.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm1.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the second convolutional layer.

```
filterSize = [3 3];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv2.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the second batch normalization layer.

```
parameters.batchnorm2.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm2.Scale = initializeOnes([numFilters 1]);
state.batchnorm2.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm2.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the third convolutional layer.

```
filterSize = [3 3];
numChannels = 32;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;
```

```
parameters.conv3.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv3.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the third batch normalization layer.

```
parameters.batchnorm3.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm3.Scale = initializeOnes([numFilters 1]);
state.batchnorm3.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm3.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the convolutional layer in the skip connection.

```
filterSize = [1 1];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.convSkip.Weights = initializeGlorot(sz,numOut,numIn);
parameters.convSkip.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the batch normalization layer in the skip connection.

```
parameters.batchnormSkip.Offset = initializeZeros([numFilters 1]);
parameters.batchnormSkip.Scale = initializeOnes([numFilters 1]);
state.batchnormSkip.TrainedMean = zeros([numFilters 1],'single');
state.batchnormSkip.TrainedVariance = ones([numFilters 1],'single');
```

Initialize the parameters for the fully connected layer corresponding to the classification output.

```
sz = [numClasses 6272];
numOut = numClasses;
numIn = 6272;
parameters.fc1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc1.Bias = initializeZeros([numClasses 1]);
```

Initialize the parameters for the fully connected layer corresponding to the regression output.

```
sz = [numResponses 6272];
numOut = numResponses;
numIn = 6272;
parameters.fc2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc2.Bias = initializeZeros([numResponses 1]);
```

View the struct of the parameters.

```
parameters
```

```
parameters = struct with fields:
            conv1: [1×1 struct]
       batchnorm1: [1×1 struct]
            conv2: [1×1 struct]
       batchnorm2: [1×1 struct]
            conv3: [1×1 struct]
       batchnorm3: [1×1 struct]
          convSkip: [1×1 struct]
    batchnormSkip: [1×1 struct]
```

```
              fc1: [1×1 struct]
              fc2: [1×1 struct]
```

View the parameters for the "conv1" operation.

```
parameters.conv1
```

```
ans = struct with fields:
    Weights: [5×5×1×16 dlarray]
       Bias: [16×1 dlarray]
```

View the struct of the state.

```
state
```

```
state = struct with fields:
       batchnorm1: [1×1 struct]
       batchnorm2: [1×1 struct]
       batchnorm3: [1×1 struct]
    batchnormSkip: [1×1 struct]
```

View the state parameters for the "batchnorm1" operation.

```
state.batchnorm1
```

```
ans = struct with fields:
        TrainedMean: [16×1 single]
    TrainedVariance: [16×1 single]
```

Define Model Function

Create the function `model`, listed at the end of the example, that computes the outputs of the deep learning model described earlier.

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

Specify Training Options

Specify the training options. Train for 20 epochs with a mini-batch size of 128.

```
numEpochs = 20;
miniBatchSize = 128;
```

To monitor the training progress, you can plot the training loss after each iteration. Create the variable plots that contains "training-progress". If you do not want to plot the training progress, then set this value to "none".

```
plots = "training-progress";
```

Train Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or angles.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''});
```

For each epoch, shuffle the data and loop over mini-batches of data. At the end of each iteration, display the training progress. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

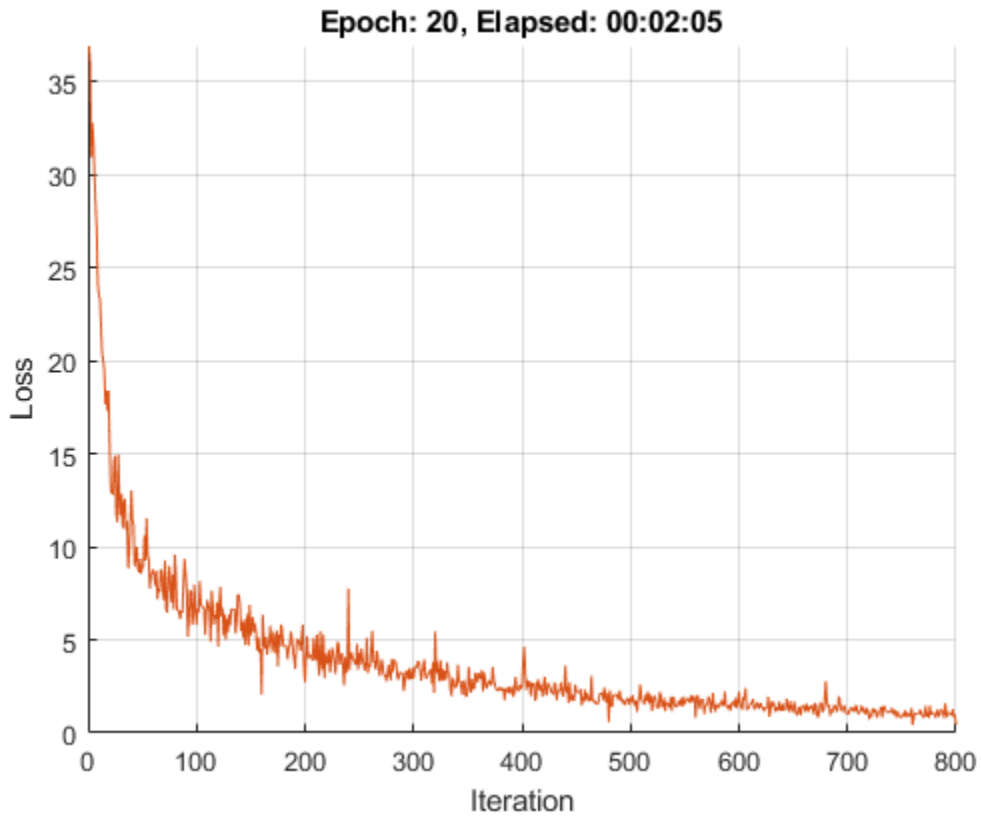
Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
```

```
    shuffle(mbq)

    % Loop over mini-batches
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,dlY1,dlY2] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,state,loss] = dlfeval(@modelGradients, parameters, dlX, dlY1, dlY2, state);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels and angles. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
[XTest,YTest,anglesTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsYTest = arrayDatastore(YTest);
dsAnglesTest = arrayDatastore(anglesTest);

dsTest = combine(dsXTest,dsYTest,dsAnglesTest);

mbqTest = minibatchqueue(dsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''});
```

To predict the labels and angles of the validation data, loop over the mini-batches and use the model function with the `doTraining` option set to `false`. Store the predicted classes and angles. Compare the predicted and true classes and angles and store the results.

```
doTraining = false;

classesPredictions = [];
anglesPredictions = [];
classCorr = [];
angleDiff = [];

% Loop over mini-batches.
while hasdata(mbqTest)

    % Read mini-batch of data.
    [dlXTest,dlY1Test,dlY2Test] = next(mbqTest);

    % Make predictions using the predict function.
    [dlY1Pred,dlY2Pred] = model(parameters,dlXTest,doTraining,state);

    % Determine predicted classes.
    Y1PredBatch = onehotdecode(dlY1Pred,classNames,1);
    classesPredictions = [classesPredictions Y1PredBatch];

    % Dermine predicted angles
    Y2PredBatch = extractdata(dlY2Pred);
    anglesPredictions = [anglesPredictions Y2PredBatch];

    % Compare predicted and true classes
    Y1Test = onehotdecode(dlY1Test,classNames,1);
    classCorr = [classCorr Y1PredBatch == Y1Test];

    % Compare predicted and true angles
    angleDiffBatch = Y2PredBatch - dlY2Test;
    angleDiff = [angleDiff extractdata(gather(angleDiffBatch))];

end
```

Evaluate the classification accuracy.

```
accuracy = mean(classCorr)
```

```
accuracy = 0.9730
```

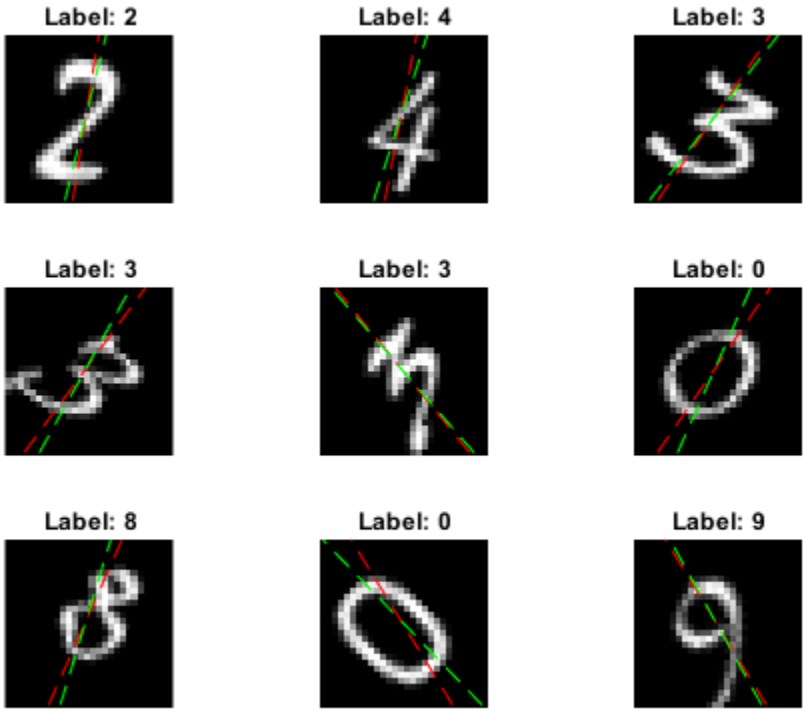
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean(angleDiff.^2))
```

```
angleRMSE = single
    6.6909
```

View some of the images with their predictions. Display the predicted angles in red and the correct labels in green.

```
idx = randperm(size(XTest,4),9);
figure
for i = 1:9
    subplot(3,3,i)
    I = XTest(:,:,:,idx(i));
    imshow(I)
    hold on

    sz = size(I,1);
    offset = sz/2;

    thetaPred = anglesPredictions(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    thetaValidation = anglesTest(idx(i));
    plot(offset*[1-tand(thetaValidation) 1+tand(thetaValidation)],[sz 0],'g--')

    hold off
    label = string(classesPredictions(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

```
function [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state)

% Convolution
weights = parameters.conv1.Weights;
bias = parameters.conv1.Bias;
dlY = dlconv(dlX,weights,bias,'Padding','same');

% Batch normalization, ReLU
offset = parameters.batchnorm1.Offset;
scale = parameters.batchnorm1.Scale;
trainedMean = state.batchnorm1.TrainedMean;
trainedVariance = state.batchnorm1.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm1.TrainedMean = trainedMean;
    state.batchnorm1.TrainedVariance = trainedVariance;
else
```

```
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution, batch normalization (Skip connection)
weights = parameters.convSkip.Weights;
bias = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,weights,bias,'Stride',2);

offset = parameters.batchnormSkip.Offset;
scale = parameters.batchnormSkip.Scale;
trainedMean = state.batchnormSkip.TrainedMean;
trainedVariance = state.batchnormSkip.TrainedVariance;

if doTraining
    [dlYSkip,trainedMean,trainedVariance] = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVa

    % Update state
    state.batchnormSkip.TrainedMean = trainedMean;
    state.batchnormSkip.TrainedVariance = trainedVariance;
else
    dlYSkip = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVariance);
end

% Convolution
weights = parameters.conv2.Weights;
bias = parameters.conv2.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same','Stride',2);

% Batch normalization, ReLU
offset = parameters.batchnorm2.Offset;
scale = parameters.batchnorm2.Scale;
trainedMean = state.batchnorm2.TrainedMean;
trainedVariance = state.batchnorm2.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution
weights = parameters.conv3.Weights;
bias = parameters.conv3.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same');

% Batch normalization
offset = parameters.batchnorm3.Offset;
scale = parameters.batchnorm3.Scale;
trainedMean = state.batchnorm3.TrainedMean;
trainedVariance = state.batchnorm3.TrainedVariance;
```

```
if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm3.TrainedMean = trainedMean;
    state.batchnorm3.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

% Addition, ReLU
dlY = dlYSkip + dlY;
dlY = relu(dlY);

% Fully connect, softmax (labels)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

% Fully connect (angles)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

end
```

Model Gradients Function

The `modelGradients` function, takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

```
function [gradients,state,loss] = modelGradients(parameters,dlX,T1,T2,state)

doTraining = true;
[dlY1,dlY2,state] = model(parameters,dlX,doTraining,state);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,parameters);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label and angle data from the incoming cell arrays and concatenate along the second dimension into a categorical array and a numeric array, respectively.

3 One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,angle] = preprocessMiniBatch(XCell,YCell,angleCell)

    % Extract image data from cell and concatenate
    X = cat(4,XCell{:});
    % Extract label data from cell and concatenate
    Y = cat(2,YCell{:});
    % Extract angle data from cell and concatenate
    angle = cat(2,angleCell{:});

    % One-hot encode labels
    Y = onehotencode(Y,1);

end
```

See Also

`dlarray` | `sgdmupdate` | `dlfeval` | `dlgradient` | `fullyconnect` | `dlconv` | `softmax` | `relu` | `batchnorm` | `crossentropy` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Functions with dlarray Support” on page 18-423

Update Batch Normalization Statistics Using Model Function

This example shows how to update the network state in a network defined as a function.

A batch normalization operation normalizes each input channel across a mini-batch. To speed up training of convolutional neural networks and reduce the sensitivity to network initialization, use batch normalization operations between convolutions and nonlinearities, such as ReLU layers.

During training, batch normalization operations first normalize the activations of each channel by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. Then, the operation shifts the input by a learnable offset β and scales it by a learnable scale factor γ.

When you use a trained network to make predictions on new data, the batch normalization operations use the trained data set mean and variance instead of the mini-batch mean and variance to normalize the activations.

To compute the data set statistics, you must keep track of the mini-batch statistics by using a continually updating state.

If you use batch normalization operations in a model function, then you must define the behavior for both training and prediction. For example, you can specify a Boolean option `doTraining` to control whether the model uses mini-batch statistics for training or data set statistics for prediction.

This example piece of code from a model function shows how to apply a batch normalization operation and update only the data set statistics during training.

```
if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm1.TrainedMean = trainedMean;
    state.batchnorm1.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end
```

Load Training Data

The `digitTrain4DArrayData` function loads the images, their digit labels, and their angles of rotation from the vertical. Create an `arrayDatastore` object for the images, labels, and the angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and number of nondiscrete responses.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;

dsXTrain = arrayDatastore(XTrain,'IterationDimension',4);
dsYTrain = arrayDatastore(YTrain);
dsAnglesTrain = arrayDatastore(anglesTrain);

dsTrain = combine(dsXTrain,dsYTrain,dsAnglesTrain);

classNames = categories(YTrain);
numClasses = numel(classNames);
numResponses = size(anglesTrain,2);
numObservations = numel(YTrain);
```

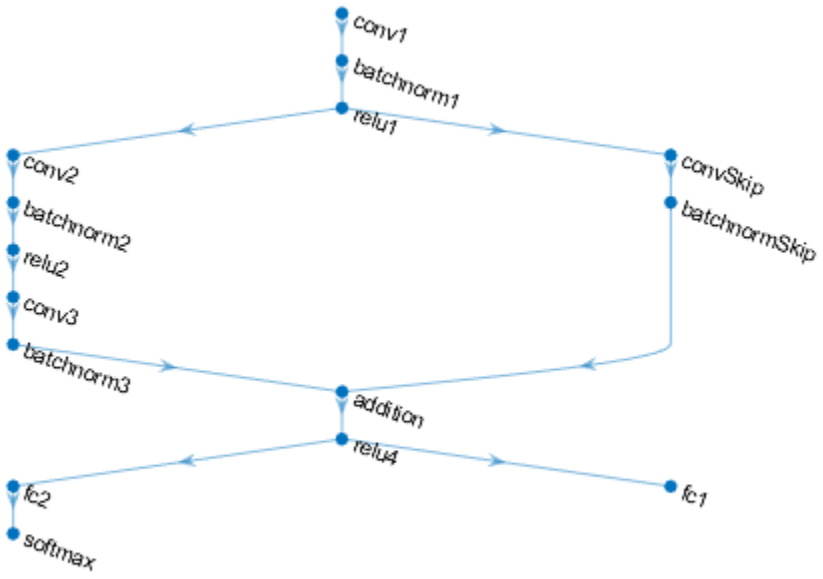
View some images from the training data.

```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

Define Deep Learning Model

Define the following network that predicts both labels and angles of rotation.

- A convolution-batchnorm-ReLU block with 16 5-by-5 filters.
- A branch of two convolution-batchnorm blocks each with 32 3-by-3 filters with a ReLU operation between
- A skip connection with a convolution-batchnorm block with 32 1-by-1 convolutions.
- Combine both branches using addition followed by a ReLU operation
- For the regression output, a branch with a fully connected operation of size 1 (the number of responses).
- For classification output, a branch with a fully connected operation of size 10 (the number of classes) and a softmax operation.



Define and Initialize Model Parameters and State

Define the parameters for each of the operations and include them in a struct. Use the format `parameters.OperationName.ParameterName` where `parameters` is the struct, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

Create a struct `parameters` containing the model parameters. Initialize the learnable layer weights and biases using the `initializeGlorot` and `initializeZeros` example functions, respectively. Initialize the batch normalization offset and scale parameters with the `initializeZeros` and `initializeOnes` example functions, respectively.

To perform training and inference using batch normalization layers, you must also manage the network state. Before prediction, you must specify the dataset mean and variance derived from the training data. Create a struct `state` containing the state parameters. The batch normalization statistics must not be `dlarray` objects. Initialize the batch normalization trained mean and trained variance states using the `zeros` and `ones` functions, respectively.

The initialization example functions are attached to this example as supporting files.

Initialize the parameters for the first convolutional layer.

```
filterSize = [5 5];
numChannels = 1;
numFilters = 16;
```

```
sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv1.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the first batch normalization layer.

```
parameters.batchnorm1.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm1.Scale = initializeOnes([numFilters 1]);
state.batchnorm1.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm1.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the second convolutional layer.

```
filterSize = [3 3];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv2.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the second batch normalization layer.

```
parameters.batchnorm2.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm2.Scale = initializeOnes([numFilters 1]);
state.batchnorm2.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm2.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the third convolutional layer.

```
filterSize = [3 3];
numChannels = 32;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv3.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv3.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the third batch normalization layer.

```
parameters.batchnorm3.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm3.Scale = initializeOnes([numFilters 1]);
state.batchnorm3.TrainedMean = zeros(numFilters,1,'single');
state.batchnorm3.TrainedVariance = ones(numFilters,1,'single');
```

Initialize the parameters for the convolutional layer in the skip connection.

```
filterSize = [1 1];
numChannels = 16;
numFilters = 32;
```

```
sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.convSkip.Weights = initializeGlorot(sz,numOut,numIn);
parameters.convSkip.Bias = initializeZeros([numFilters 1]);
```

Initialize the parameters and state for the batch normalization layer in the skip connection.

```
parameters.batchnormSkip.Offset = initializeZeros([numFilters 1]);
parameters.batchnormSkip.Scale = initializeOnes([numFilters 1]);
state.batchnormSkip.TrainedMean = zeros([numFilters 1],'single');
state.batchnormSkip.TrainedVariance = ones([numFilters 1],'single');
```

Initialize the parameters for the fully connected layer corresponding to the classification output.

```
sz = [numClasses 6272];
numOut = numClasses;
numIn = 6272;
parameters.fc1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc1.Bias = initializeZeros([numClasses 1]);
```

Initialize the parameters for the fully connected layer corresponding to the regression output.

```
sz = [numResponses 6272];
numOut = numResponses;
numIn = 6272;
parameters.fc2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc2.Bias = initializeZeros([numResponses 1]);
```

View the struct of the state.

```
state
```

```
state = struct with fields:
       batchnorm1: [1×1 struct]
       batchnorm2: [1×1 struct]
       batchnorm3: [1×1 struct]
    batchnormSkip: [1×1 struct]
```

View the state parameters for the `batchnorm1` operation.

```
state.batchnorm1
```

```
ans = struct with fields:
        TrainedMean: [16×1 single]
    TrainedVariance: [16×1 single]
```

Define Model Function

Create the function `model`, listed at the end of the example, which computes the outputs of the deep learning model described earlier.

The function `model` takes as input the model parameters `parameters`, input data `dlX`, the flag `doTraining`, which specifies whether the model returns outputs for training or prediction, and the

network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, which takes as input a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

Specify Training Options

Specify the training options.

```
numEpochs = 20;
miniBatchSize = 128;

plots = "training-progress";
```

Train Model

Train the model using a custom training loop. Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or the angles.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''});
```

For each epoch, shuffle the data and loop over mini-batches of data. At the end of each epoch, display the training progress. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.

Initialize the parameters for the Adam solver.

```
trailingAvg = [];
trailingAvgSq = [];
```

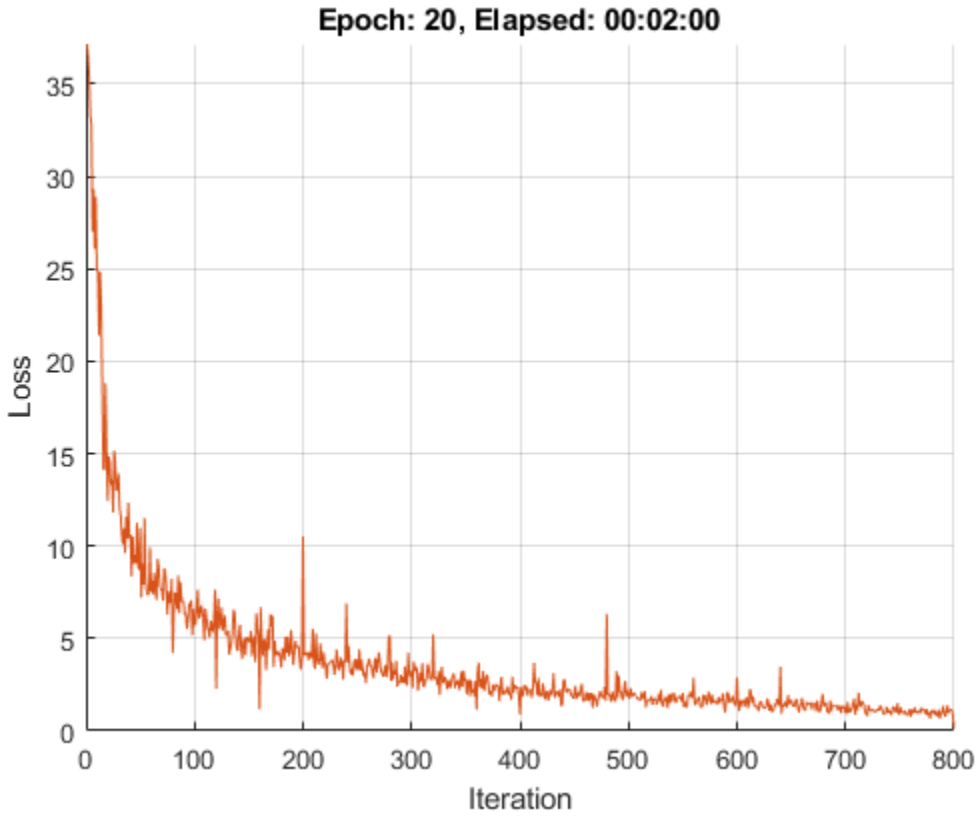
Initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
```

```
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches
    while hasdata(mbq)
        iteration = iteration + 1;

        [dlX,dlY1,dlY2] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,state,loss] = dlfeval(@modelGradients, parameters, dlX, dlY1, dlY2, state);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            addpoints(lineLossTrain,iteration,double(gather(extractdata(loss))))
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
```



Test Model

Test the classification accuracy of the model by comparing the predictions on a test set with the true labels and angles. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
[XTest,Y1Test,anglesTest] = digitTest4DArrayData;

dsXTest = arrayDatastore(XTest,'IterationDimension',4);
dsYTest = arrayDatastore(Y1Test);
dsAnglesTest = arrayDatastore(anglesTest);

dsTest = combine(dsXTest,dsYTest,dsAnglesTest);

mbqTest = minibatchqueue(dsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''});
```

To predict the labels and angles of the validation data, use the `modelPredictions` function, listed at the end of the example. The function returns the predicted classes and angles, as well as comparison with the true values.

```
[classesPredictions,anglesPredictions,classCorr,angleDiff] = modelPredictions(parameters,state,mb
```

Evaluate the classification accuracy.

```
accuracy = mean(classCorr)
```

```
accuracy = 0.9840
```

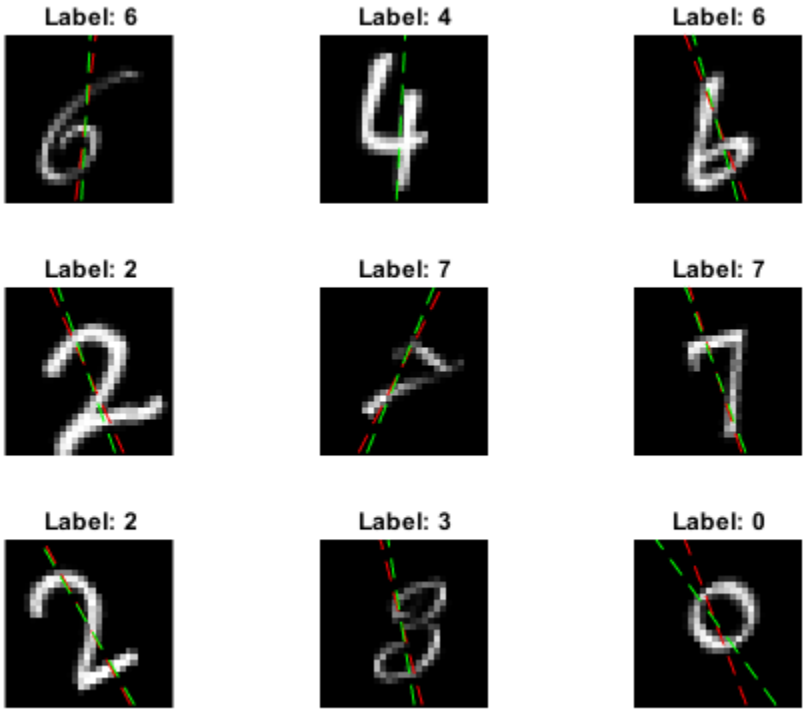
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean(angleDiff.^2))
```

```
angleRMSE = single
    6.3669
```

View some of the images with their predictions. Display the predicted angles in red and the correct angles in green.

```
idx = randperm(size(XTest,4),9);
figure
for i = 1:9
    subplot(3,3,i)
    I = XTest(:,:,:,idx(i));
    imshow(I)
    hold on

    sz = size(I,1);
    offset = sz/2;

    thetaPred = anglesPredictions(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    thetaValidation = anglesTest(idx(i));
    plot(offset*[1-tand(thetaValidation) 1+tand(thetaValidation)],[sz 0],'g--')

    hold off
    label = string(classesPredictions(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes as input the model parameters `parameters`, the input data `dlX`, the flag `doTraining`, which specifies whether the model returns the outputs for training or prediction, and the network state `state`. The function returns the predictions for the labels, the predictions for the angles, and the updated network state.

```
function [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state)

    % Convolution
    weights = parameters.conv1.Weights;
    bias = parameters.conv1.Bias;
    dlY = dlconv(dlX,weights,bias,'Padding',2);

    % Batch normalization, ReLU
    offset = parameters.batchnorm1.Offset;
    scale = parameters.batchnorm1.Scale;
    trainedMean = state.batchnorm1.TrainedMean;
    trainedVariance = state.batchnorm1.TrainedVariance;

    if doTraining
        [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVarianc

        % Update state
        state.batchnorm1.TrainedMean = trainedMean;
        state.batchnorm1.TrainedVariance = trainedVariance;
    else
```

```
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution, batch normalization (skip connection)
weights = parameters.convSkip.Weights;
bias = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,weights,bias,'Stride',2);

offset = parameters.batchnormSkip.Offset;
scale = parameters.batchnormSkip.Scale;
trainedMean = state.batchnormSkip.TrainedMean;
trainedVariance = state.batchnormSkip.TrainedVariance;

if doTraining
    [dlYSkip,trainedMean,trainedVariance] = batchnorm(dlYSkip,offset,scale,trainedMean,traine

    % Update state
    state.batchnormSkip.TrainedMean = trainedMean;
    state.batchnormSkip.TrainedVariance = trainedVariance;
else
    dlYSkip = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVariance);
end

% Convolution
weights = parameters.conv2.Weights;
bias = parameters.conv2.Bias;
dlY = dlconv(dlY,weights,bias,'Padding',1,'Stride',2);

% Batch normalization, ReLU
offset = parameters.batchnorm2.Offset;
scale = parameters.batchnorm2.Scale;
trainedMean = state.batchnorm2.TrainedMean;
trainedVariance = state.batchnorm2.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVarianc

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution
weights = parameters.conv3.Weights;
bias = parameters.conv3.Bias;
dlY = dlconv(dlY,weights,bias,'Padding',1);

% Batch normalization
offset = parameters.batchnorm3.Offset;
scale = parameters.batchnorm3.Scale;
trainedMean = state.batchnorm3.TrainedMean;
trainedVariance = state.batchnorm3.TrainedVariance;
```

```
    if doTraining
        [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVarianc

        % Update state
        state.batchnorm3.TrainedMean = trainedMean;
        state.batchnorm3.TrainedVariance = trainedVariance;
    else
        dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
    end

    % Addition, ReLU
    dlY = dlYSkip + dlY;
    dlY = relu(dlY);

    % Fully connect, softmax (labels)
    weights = parameters.fc1.Weights;
    bias = parameters.fc1.Bias;
    dlY1 = fullyconnect(dlY,weights,bias);
    dlY1 = softmax(dlY1);

    % Fully connect (angles)
    weights = parameters.fc2.Weights;
    bias = parameters.fc2.Bias;
    dlY2 = fullyconnect(dlY,weights,bias);

end
```

Model Gradients Function

The `modelGradients` function takes as input the model parameters, a mini-batch of the input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

```
function [gradients,state,loss] = modelGradients(parameters,dlX,T1,T2,state)

    doTraining = true;
    [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state);

    lossLabels = crossentropy(dlY1,T1);
    lossAngles = mse(dlY2,T2);

    loss = lossLabels + 0.1*lossAngles;
    gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes the model parameters, the network state, a `minibatchqueue` of input data `mbq`, and the network classes, and returns the model predictions by iterating over all data in the `minibatchqueue` using the `model` function with the `doTraining` option set to `false`. The function returns the predicted classes and angles, as well as comparison with the true values. For the classes, the comparison is a vector of ones and zeros that represents correct and incorrect predictions. For the angles, the comparison is the difference between the predicted angle and the true value.

```
function [classesPredictions,anglesPredictions,classCorr,angleDiff] = modelPredictions(parameters

    doTraining = false;

    classesPredictions = [];
    anglesPredictions = [];
    classCorr = [];
    angleDiff = [];

    while hasdata(mbq)
        [dlX,dlY1,dlY2] = next(mbq);

        % Make predictions using the model function.
        [dlY1Pred,dlY2Pred] = model(parameters,dlX,doTraining,state);

        % Determine predicted classes.
        Y1PredBatch = onehotdecode(dlY1Pred,classes,1);
        classesPredictions = [classesPredictions Y1PredBatch];

        % Dermine predicted angles
        Y2PredBatch = extractdata(dlY2Pred);
        anglesPredictions = [anglesPredictions Y2PredBatch];

        % Compare predicted and true classes
        Y1 = onehotdecode(dlY1,classes,1);
        classCorr = [classCorr Y1PredBatch == Y1];

        % Compare predicted and true angles
        angleDiffBatch = Y2PredBatch - dlY2;
        angleDiff = [angleDiff extractdata(gather(angleDiffBatch))];

    end

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label and angle data from the incoming cell arrays and concatenate into a categorical array and a numeric array, respectively.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,angle] = preprocessMiniBatch(XCell,YCell,angleCell)

    % Extract image data from cell and concatenate
    X = cat(4,XCell{:});
    % Extract label data from cell and concatenate
    Y = cat(2,YCell{:});
    % Extract angle data from cell and concatenate
    angle = cat(2,angleCell{:});

    % One-hot encode labels
```

```
    Y = onehotencode(Y,1);

end
```

See Also

`dlarray` | `sgdmupdate` | `dlfeval` | `dlgradient` | `fullyconnect` | `dlconv` | `softmax` | `relu` | `batchnorm` | `crossentropy` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Train Network Using Model Function” on page 18-259
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Functions with dlarray Support” on page 18-423

Make Predictions Using Model Function

This example shows how to make predictions using a model function by splitting data into mini-batches.

For large data sets, or when predicting on hardware with limited memory, make predictions by splitting the data into mini-batches. When making predictions with `SeriesNetwork` or `DAGNetwork` objects, the `predict` function automatically splits the input data into mini-batches. For model functions, you must split the data into mini-batches manually.

Create Model Function and Load Parameters

Load the model parameters from the MAT file `digitsMIMO.mat`. The MAT file contains the model parameters in a struct named `parameters`, the model state in a struct named `state`, and the class names in `classNames`.

```
s = load("digitsMIMO.mat");
parameters = s.parameters;
state = s.state;
classNames = s.classNames;
```

The model function `model`, listed at the end of the example, defines the model given the model parameters and state.

Load Data for Prediction

Load the digits data for prediction.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

numObservations = numel(imds.Files);
```

Make Predictions

Loop over the mini-batches of the test data and make predictions using a custom prediction loop.

Use `minibatchqueue` to process and manage the mini-batches of images. Specify a mini-batch size of 128. Set the read size property of the image datastore to the mini-batch size.

For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to concatenate the data into a batch and normalize the images.
- Format the images with the dimensions `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Make predictions on a GPU if one is available. By default, the `minibatchqueue` object converts the output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;
```

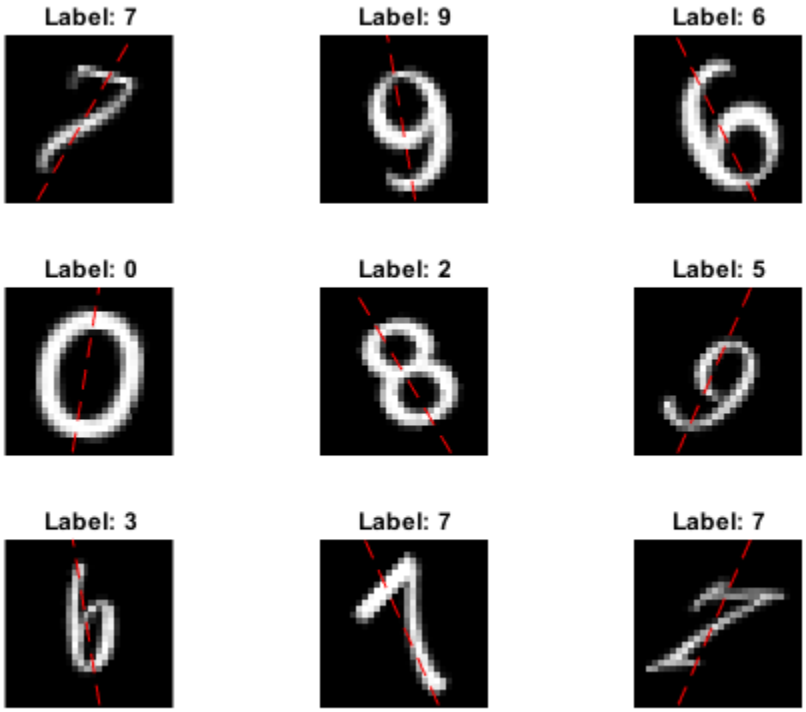
```
mbq = minibatchqueue(imds,...
    "MiniBatchSize",miniBatchSize,...
    "MiniBatchFcn", @preprocessMiniBatch,...
    "MiniBatchFormat","SSCB");
```

Loop over the minibatches of data and make predictions using the `predict` function. Use the `onehotdecode` function to determine the class labels. Store the predicted class labels.

```
doTraining = false;

Y1Predictions = [];
Y2Predictions = [];

% Loop over mini-batches.
while hasdata(mbq)

    % Read mini-batch of data.
    dlX = next(mbq);

    % Make predictions using the predict function.
    [dlY1Pred,dlY2Pred] = model(parameters,dlX,doTraining,state);

    % Determine corresponding classes.
    Y1PredBatch = onehotdecode(dlY1Pred,classNames,1);
    Y1Predictions = [Y1Predictions Y1PredBatch];

    Y2PredBatch = extractdata(dlY2Pred);
    Y2Predictions = [Y2Predictions Y2PredBatch];

end
```

View some of the images with their predictions.

```
idx = randperm(numObservations,9);
figure
for i = 1:9
    subplot(3,3,i)
    I = imread(imds.Files{idx(i)});
    imshow(I)
    hold on

    sz = size(I,1);
    offset = sz/2;

    thetaPred = Y2Predictions(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    hold off
    label = string(Y1Predictions(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

```
function [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state)

% Convolution
weights = parameters.conv1.Weights;
bias = parameters.conv1.Bias;
dlY = dlconv(dlX,weights,bias,'Padding','same');

% Batch normalization, ReLU
offset = parameters.batchnorm1.Offset;
scale = parameters.batchnorm1.Scale;
trainedMean = state.batchnorm1.TrainedMean;
trainedVariance = state.batchnorm1.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm1.TrainedMean = trainedMean;
    state.batchnorm1.TrainedVariance = trainedVariance;
else
```

```
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution, batch normalization (Skip connection)
weights = parameters.convSkip.Weights;
bias = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,weights,bias,'Stride',2);

offset = parameters.batchnormSkip.Offset;
scale = parameters.batchnormSkip.Scale;
trainedMean = state.batchnormSkip.TrainedMean;
trainedVariance = state.batchnormSkip.TrainedVariance;

if doTraining
    [dlYSkip,trainedMean,trainedVariance] = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVa

    % Update state
    state.batchnormSkip.TrainedMean = trainedMean;
    state.batchnormSkip.TrainedVariance = trainedVariance;
else
    dlYSkip = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVariance);
end

% Convolution
weights = parameters.conv2.Weights;
bias = parameters.conv2.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same','Stride',2);

% Batch normalization, ReLU
offset = parameters.batchnorm2.Offset;
scale = parameters.batchnorm2.Scale;
trainedMean = state.batchnorm2.TrainedMean;
trainedVariance = state.batchnorm2.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution
weights = parameters.conv3.Weights;
bias = parameters.conv3.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same');

% Batch normalization
offset = parameters.batchnorm3.Offset;
scale = parameters.batchnorm3.Scale;
trainedMean = state.batchnorm3.TrainedMean;
trainedVariance = state.batchnorm3.TrainedVariance;
```

```
if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm3.TrainedMean = trainedMean;
    state.batchnorm3.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

% Addition, ReLU
dlY = dlYSkip + dlY;
dlY = relu(dlY);

% Fully connect, softmax (labels)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

% Fully connect (angles)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the data from the incoming cell array and concatenate into a numeric array. Concatenating over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Normalize the pixel values between `0` and `1`.

```
function X = preprocessMiniBatch(data)
    % Extract image data from cell and concatenate
    X = cat(4,data{:});

    % Normalize the images.
    X = X/255;
end
```

See Also

`dlarray` | `dlgradient` | `dlfeval` | `sgdmupdate` | `dlconv` | `batchnorm` | `relu` | `fullyconnect` | `softmax` | `minibatchqueue` | `onehotdecode`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Train Network Using Model Function” on page 18-259

- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Initialize Learnable Parameters for Model Function” on page 18-292
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Functions with dlarray Support” on page 18-423

Initialize Learnable Parameters for Model Function

When you train a network using layers, layer graphs, or `dlnetwork` objects, the software automatically initializes the learnable parameters according to the layer initialization properties. When you define a deep learning model as a function, you must initialize the learnable parameters manually.

How you initialize learnable parameters (for example, weights and biases) can have a big impact on how quickly a deep learning model converges.

Tip This topic explains how to initialize learnable parameters for a deep learning model defined a function in a custom training loop. To learn how to specify the learnable parameter initialization for a deep learning layer, use the corresponding layer property. For example, to set the weights initializer of a `convolution2dLayer` object, use the `WeightsInitializer` property.

Default Layer Initializations

This table shows the default initializations for the learnable parameters for each layer, and provides links that show how to initialize learnable parameters for model functions by using the same initialization.

Layer	Learnable Parameter	Default Initialization
`convolution2dLayer`	Weights	“Glorot Initialization” on page 18-296
	Bias	“Zeros Initialization” on page 18-302
`convolution3dLayer`	Weights	“Glorot Initialization” on page 18-296
	Bias	“Zeros Initialization” on page 18-302
`groupedConvolution2dLayer`	Weights	“Glorot Initialization” on page 18-296
	Bias	“Zeros Initialization” on page 18-302
`transposedConv2dLayer`	Weights	“Glorot Initialization” on page 18-296
	Bias	“Zeros Initialization” on page 18-302
`transposedConv3dLayer`	Weights	“Glorot Initialization” on page 18-296
	Bias	“Zeros Initialization” on page 18-302
`fullyConnectedLayer`	Weights	“Glorot Initialization” on page 18-296

Layer	Learnable Parameter	Default Initialization
	Bias	“Zeros Initialization” on page 18-302
`batchNormalizationLayer`	Offset	“Zeros Initialization” on page 18-302
	Scale	“Ones Initialization” on page 18-301
`lstmLayer`	Input weights	“Glorot Initialization” on page 18-296
	Recurrent weights	“Orthogonal Initialization” on page 18-300
	Bias	“Unit Forget Gate Initialization” on page 18-301
`gruLayer`	Input weights	“Glorot Initialization” on page 18-296
	Recurrent weights	“Orthogonal Initialization” on page 18-300
	Bias	“Zeros Initialization” on page 18-302
`wordEmbeddingLayer`	Weights	“Gaussian Initialization” on page 18-299, with mean 0 and standard deviation 0.01

Learnable Parameter Sizes

When initializing learnable parameters for model functions, you must specify parameters of the correct size. The size of the learnable parameters depends on the type of deep learning operation.

Operation	Learnable Parameter	Size
`batchnorm`	Offset	`[numChannels 1]`, where `numChannels` is the number of input channels
	Scale	`[numChannels 1]`, where `numChannels` is the number of input channels
`dlconv`	Weights	`[filterSize numChannels numFilters]`, where `filterSize` is a 1-by-`K` vector specifying the filter size, `numChannels` is the number of input channels, `numFilters` is the number of filters, and `K` is the number of spatial dimensions

Operation	Learnable Parameter	Size
	Bias	One of the following: • `[numFilters 1]`, where `numFilters` is the number of filters • `[1 1]`
`dlconv` (grouped)	Weights	`[filterSize numChannelsPerGroup numFiltersPerGroup numGroups]`, where `filterSize` is a 1-by-`K` vector specifying the filter size, `numChannelsPerGroup` is the number of input channels for each group, `numFiltersPerGroup` is the number of filters for each group, `numGroups` is the number of groups, and `K` is the number of spatial dimensions
	Bias	One of the following: • `[numFiltersPerGroup 1]`, where `numFiltersPerGroup` is the number of filters for each group. • `[1 1]`
`dltranspconv`	Weights	`[filterSize numFilters numChannels]`, where `filterSize` is a 1-by-`K` vector specifying the filter size, `numChannels` is the number of input channels, `numFilters` is the number of filters, and `K` is the number of spatial dimensions
	Bias	One of the following: • `[numFilters 1]`, where `numFilters` is the number of filters for each group. • `[1 1]`

Operation	Learnable Parameter	Size
`dltranspconv` (grouped)	Weights	`[filterSize numFiltersPerGroup numChannelsPerGroup numGroups]`, where `filterSize` is a 1-by-`K` vector specifying the filter size, `numChannelsPerGroup` is the number of input channels for each group, `numFiltersPerGroup` is the number of filters for each group, `numGroups` is the number of groups, and `K` is the number of spatial dimensions
	Bias	One of the following: • `[numFiltersPerGroup 1]`, where `numFiltersPerGroup` is the number of filters for each group. • `[1 1]`
`fullyconnect`	Weights	`[outputSize inputSize]`, where `outputSize` and `inputSize` is the number of output and input channels, respectively
	Bias	`[outputSize 1]`, where `outputSize` is the number of output channels
`gru`	Input weights	`[3*numHiddenUnits inputSize]`, where `numHiddenUnits` is the number of hidden units of the operation and `inputSize` is the number of input channels
	Recurrent weights	`[3*numHiddenUnits numHiddenUnits]`, where `numHiddenUnits` is the number of hidden units of the operation
	Bias	`[3*numHiddenUnits 1]`, where `numHiddenUnits` is the number of hidden units of the operation

Operation	Learnable Parameter	Size
`lstm`	Input weights	`[4*numHiddenUnits inputSize]`, where `numHiddenUnits` is the number of hidden units of the operation and `inputSize` is the number of input channels
	Recurrent weights	`[4*numHiddenUnits numHiddenUnits]`, where `numHiddenUnits` is the number of hidden units of the operation
	Bias	`[4*numHiddenUnits 1]`, where `numHiddenUnits` is the number of hidden units of the operation

Glorot Initialization

The Glorot (also known as Xavier) initializer [1] samples weights from the uniform distribution with bounds $\left[-\sqrt{\frac{6}{N_o + N_i}}, \sqrt{\frac{6}{N_o + N_i}}\right]$, where the values of N_o and N_i depend on the type of deep learning operation.

Operation	Learnable Parameter	N_o	N_i
`dlconv`	Weights	`prod(filterSize)*numFilters`, where `filterSize` is a 1-by-`K` vector containing the filter size, `numFilters` is the number of filters, and `K` is the number of spatial dimensions	`prod(filterSize)*numChannels`, where `filterSize` is a 1-by-`K` vector containing the filter size, `numChannels` is the number of input channels, and `K` is the number of spatial dimensions
`dlconv` (grouped)	Weights	`prod(filterSize)*numFiltersPerGroup`, where `filterSize` is a 1-by-`K` vector containing the filter size, `numFiltersPerGroup` is the number of filters for each group, and `K` is the number of spatial dimensions	`prod(filterSize)*numChannelsPerGroup`, where `filterSize` is a 1-by-`K` vector containing the filter size, `numChannelsPerGroup` is the number of input channels for each group, and `K` is the number of spatial dimensions

Operation	Learnable Parameter	N_o	N_i
`dltranspconv`	Weights	`prod(filterSize)*numFilters`, where `filterSize` is a 1-by-K vector containing the filter size, `numFilters` is the number of filters, and K is the number of spatial dimensions	`prod(filterSize)*numChannels`, where `filterSize` is a 1-by-K vector containing the filter size, `numChannels` is the number of input channels, and K is the number of spatial dimensions
`dltranspconv` (grouped)	Weights	`prod(filterSize)*numFiltersPerGroup`, where `filterSize` is a 1-by-K vector containing the filter size, `numFiltersPerGroup` is the number of filters for each group, and K is the number of spatial dimensions	`prod(filterSize)*numChannelsPerGroup`, where `filterSize` is a 1-by-K vector containing the filter size, `numChannelsPerGroup` is the number of input channels for each group, and K is the number of spatial dimensions
`fullyconnect`	Weights	Number of output channels of the operation	Number of input channels of the operation
`gru`	Input weights	`3*numHiddenUnits`, where `numHiddenUnits` is the number of hidden units of the operation	Number of input channels of the operation
	Recurrent weights	`3*numHiddenUnits`, where `numHiddenUnits` is the number of hidden units of the operation	Number of hidden units of the operation
`lstm`	Input weights	`4*numHiddenUnits`, where `numHiddenUnits` is the number of hidden units of the operation	Number of input channels of the operation
	Recurrent weights	`4*numHiddenUnits`, where `numHiddenUnits` is the number of hidden units of the operation	Number of hidden units of the operation

To initialize learnable parameters using the Glorot initializer easily, you can define a custom function. The function `initializeGlorot` takes as input the size of the learnable parameters `sz` and the

values N_o and N_i (`numOut` and `numIn`, respectively), and returns the sampled weights as a `dlarray` object with underlying type `'single'`.

```
function weights = initializeGlorot(sz,numOut,numIn)

Z = 2*rand(sz,'single') - 1;
bound = sqrt(6 / (numIn + numOut));

weights = bound * Z;
weights = dlarray(weights);

end
```

Example

Initialize the weights for a convolutional operation with 128 filters of size 5-by-5 and 3 input channels.

```
filterSize = [5 5];
numChannels = 3;
numFilters = 128;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv.Weights = initializeGlorot(sz,numOut,numIn);
```

He Initialization

The He initializer [2] samples weights from the normal distribution with zero mean and variance $\frac{2}{N_i}$, where the value N_i depends on the type of deep learning operation.

Operation	Learnable Parameter	N_i
`dlconv`	Weights	`prod(filterSize)*numChannelsPerGroup`, where `filterSize` is a 1-by-K vector containing the filter size, `numChannelsPerGroup` is the number of input channels for each group, and K is the number of spatial dimensions
`dltranspconv`	Weights	`prod(filterSize)*numChannelsPerGroup`, where `filterSize` is a 1-by-K vector containing the filter size, `numChannelsPerGroup` is the number of input channels for each group, and K is the number of spatial dimensions

Operation	Learnable Parameter	N_i
`fullyconnect`	Weights	Number of input channels of the operation
`gru`	Input weights	Number of input channels of the operation
	Recurrent weights	Number of hidden units of the operation.
`lstm`	Input weights	Number of input channels of the operation
	Recurrent weights	Number of hidden units of the operation.

To initialize learnable parameters using the He initializer easily, you can define a custom function. The function `initializeHe` takes as input the size of the learnable parameters `sz`, and the value N_i, and returns the sampled weights as a `dlarray` object with underlying type `'single'`.

```
function weights = initializeHe(sz,numIn)

weights = randn(sz,'single') * sqrt(2/numIn);
weights = dlarray(weights);

end
```

Example

Initialize the weights for a convolutional operation with 128 filters of size 5-by-5 and 3 input channels.

```
filterSize = [5 5];
numChannels = 3;
numFilters = 128;

sz = [filterSize numChannels numFilters];
numIn = prod(filterSize) * numFilters;

parameters.conv.Weights = initializeHe(sz,numIn);
```

Gaussian Initialization

The Gaussian initializer samples weights from a normal distribution.

To initialize learnable parameters using the Gaussian initializer easily, you can define a custom function. The function `initializeGaussian` takes as input the size of the learnable parameters `sz`, the distribution mean `mu`, and the distribution standard deviation `sigma`, and returns the sampled weights as a `dlarray` object with underlying type `'single'`.

```
function weights = initializeGaussian(sz,mu,sigma)

weights = randn(sz,'single')*sigma + mu;
weights = dlarray(weights);

end
```

Example

Initialize the weights for an embedding operation with a dimension of 300 and vocabulary size of 5000 using the Gaussian initializer with mean 0 and standard deviation 0.01.

```
embeddingDimension = 300;
vocabularySize = 5000;
mu = 0;
sigma = 0.01;

sz = [embeddingDimension vocabularySize];

parameters.emb.Weights = initializeGaussian(sz,mu,sigma);
```

Uniform Initialization

The uniform initializer samples weights from a uniform distribution.

To initialize learnable parameters using the uniform initializer easily, you can define a custom function. The function `initializeUniform` takes as input the size of the learnable parameters `sz`, and the distribution bound `bound`, and returns the sampled weights as a `dlarray` object with underlying type `'single'`.

```
function parameter = initializeUniform(sz,bound)

Z = 2*rand(sz,'single') - 1;
parameter = bound * Z;
parameter = dlarray(parameter);

end
```

Example

Initialize the weights for an attention mechanism with size 100-by-100 and bound 0.1 using the uniform initializer.

```
sz = [100 100];
bound = 0.1;

parameters.attentionn.Weights = initializeUniform(sz,bound);
```

Orthogonal Initialization

The orthogonal initializer returns the orthogonal matrix Q given by the QR decomposition of $Z = QR$, where Z is sampled from a unit normal distribution and the size of Z matches the size of the learnable parameter.

To initialize learnable parameters using the orthogonal initializer easily, you can define a custom function. The function `initializeOrthogonal` takes as input the size of the learnable parameters `sz`, and returns the orthogonal matrix as a `dlarray` object with underlying type `'single'`.

```
function parameter = initializeOrthogonal(sz)

Z = randn(sz,'single');
[Q,R] = qr(Z,0);
```

```
D = diag(R);
Q = Q * diag(D ./ abs(D));

parameter = dlarray(Q);

end
```

Example

Initialize the recurrent weights for an LSTM operation with 100 hidden units using the orthogonal initializer.

```
numHiddenUnits = 100;

sz = [4*numHiddenUnits numHiddenUnits];

parameters.lstm.RecurrentWeights = initializeOrthogonal(sz);
```

Unit Forget Gate Initialization

The unit forget gate initializer initializes the bias for an LSTM operation such that the forget gate component of the biases are ones and the remaining entries are zeros.

To initialize learnable parameters using the orthogonal initializer easily, you can define a custom function. The function `initializeUnitForgetGate` takes as input the number of hidden units in the LSTM operation, and returns the bias as a `dlarray` object with underlying type `'single'`.

```
function bias = initializeUnitForgetGate(numHiddenUnits)

bias = zeros(4*numHiddenUnits,1,'single');

idx = numHiddenUnits+1:2*numHiddenUnits;
bias(idx) = 1;

bias = dlarray(bias);

end
```

Example

Initialize the bias of an LSTM operation with 100 hidden units using the unit forget gate initializer.

```
numHiddenUnits = 100;

parameters.lstm.Bias = initializeUnitForgetGate(numHiddenUnits,'single');
```

Ones Initialization

To initialize learnable parameters with ones easily, you can define a custom function. The function `initializeOnes` takes as input the size of the learnable parameters `sz`, and returns the parameters as a `dlarray` object with underlying type `'single'`.

```
function parameter = initializeOnes(sz)

parameter = ones(sz,'single');
parameter = dlarray(parameter);
```

```
end
```

Example

Initialize the scale for a batch normalization operation with 128 input channels with ones.

```
numChannels = 128;

sz = [numChannels 1];

parameters.bn.Scale = initializeOnes(sz);
```

Zeros Initialization

To initialize learnable parameters with zeros easily, you can define a custom function. The function `initializeZeros` takes as input the size of the learnable parameters `sz`, and returns the parameters as a `dlarray` object with underlying type `'single'`.

```
function parameter = initializeZeros(sz)

parameter = zeros(sz,'single');
parameter = dlarray(parameter);

end
```

Example

Initialize the offset for a batch normalization operation with 128 input channels with zeros.

```
numChannels = 128;

sz = [numChannels 1];

parameters.bn.Offset = initializeZeros(sz);
```

Storing Learnable Parameters

It is recommended to store the learnable parameters for a given model function in a single object, such as a struct, table, or cell array. For an example showing how to initialize learnable parameters as a struct, see “Train Network Using Model Function” on page 18-259.

Storing Parameters on GPU

If you train your model using training data that is stored on the GPU, the learnable parameters of the model function are converted to gpuArray objects and stored on the GPU.

Before saving your learnable parameters, it is recommended practice to gather all parameters, in case they are loaded onto a machine without a GPU. Use `dlupdate` to gather learnable parameters stored as a struct, table, or cell array. For example, if you have network learnable parameters stored on the GPU in the struct, table, or cell array `parameters`, you can gather all parameters on the to CPU by using the following code:

```
parameters = dlupdate(@gather,parameters);
```

If you load learnable parameters that are not on the GPU, you can move the parameters onto the GPU using `dlupdate`. Doing so ensures that your network executes on the GPU for training and inference, regardless of where the input data is stored. To move the learnable parameters onto the GPU, use the `dlupdate` function:

```
parameters = dlupdate(@gpuArray,parameters);
```

References

[1] Glorot, Xavier, and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–356. Sardinia, Italy: AISTATS, 2010.

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, 1026–1034. Washington, DC: IEEE Computer Vision Society, 2015.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Train Network Using Model Function” on page 18-259
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Make Predictions Using Model Function” on page 18-286
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “List of Functions with dlarray Support” on page 18-423

Deep Learning Function Acceleration for Custom Training Loops

When using the `dlfeval` function in a custom training loop, the software traces each input `dlarray` object of the model gradients function to determine the computation graph used for automatic differentiation. This tracing process can take some time and can spend time recomputing the same trace. By optimizing, caching, and reusing the traces, you can speed up gradient computation in deep learning functions. You can also optimize, cache, and reuse traces to accelerate other deep learning functions that do not require automatic differentiation, for example you can also accelerate model functions and functions used for prediction.

To speed up calls to deep learning functions, you can use the `dlaccelerate` function to create an `AcceleratedFunction` object that automatically optimizes, caches, and reuses the traces. You can use the `dlaccelerate` function to accelerate model functions and model gradients functions directly.

The returned `AcceleratedFunction` object caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` objects, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

Invoke the accelerated function as you would invoke the underlying function. Note that the accelerated function is not a function handle.

Note When using the `dlfeval` function, the software automatically accelerates the `forward` and `predict` functions for `dlnetwork` input. If you accelerate a deep learning function where the majority of the computation takes place in calls to the `forward` or `predict` functions for `dlnetwork` input, then you might not see an improvement in training time.

Because of the nature of caching traces, not all functions support acceleration.

The caching process can cache values that you might expect to change or that depend on external factors. You must take care when accelerating functions that:

- have inputs with random or frequently changing values
- have outputs with frequently changing values
- generate random numbers
- use `if` statements and `while` loops with conditions that depend on the values of `dlarray` objects
- have inputs that are handles or that depend on handles
- Read data from external sources (for example, by using a datastore or a `minibatchqueue` object)

Accelerated functions can do the following when calculating a new trace only.

- modify the global state such as, the random number stream or global variables

- use file input or output
- display data using graphics or the command line display

When using accelerated functions in parallel, such as when using a `parfor` loop, then each worker maintains its own cache. The cache is not transferred to the host.

Functions and custom layers used in accelerated functions must also support acceleration.

You can nest and recursively call accelerated functions. However, it is usually more efficient to have a single accelerated function.

Accelerate Deep Learning Function Directly

In most cases, you can accelerate deep learning functions directly. For example, you can accelerate the model gradients function directly by replacing calls to the model gradients function with calls to the corresponding accelerated function:

Consider the following use of the `dlfeval` function in a custom training loop.

```
[gradients,state,loss] = dlfeval(@modelGradients,parameters,dlX,dlT,state)
```

To accelerate the model gradients function and evaluate the accelerated function, use the `dlaccelerate` function and evaluate the returned `AcceleratedFunction` object:

```
accfun = dlaccelerate(@modelGradients);
[gradients,state,loss] = dlfeval(accfun,parameters,dlX,dlT,state)
```

Because the cached traces are not directly attached to the `AcceleratedFunction` object and that they are shared between `AcceleratedFunction` objects that use the same underlying function, you can create the `AcceleratedFunction` either in or before the custom training loop body.

Accelerate Parts of Deep Learning Function

If a deep learning function does not fully support acceleration, for example, functions that require an `if` statement with a condition that depends on the value of a `dlarray` object, then you can accelerate parts of a deep learning function by creating a separate function contains any supported function calls you want to accelerate.

For example, consider the following code snippet that calls different functions depending on whether the sum of the `dlarray` object `dlX` is negative or nonnegative.

```
if sum(dlX,'all') < 0
   dlX = negFun1(parameters,dlX);
   dlX = negFun2(parameters,dlX);
else
   dlX = posFun1(parameters,dlX);
   dlX = posFun2(parameters,dlX);
end
```

Because the `if` statement depends on the value of a `dlarray` object, a function that contains this code snippet does not support acceleration. However, if the blocks of code used inside the body of the `if` statement support acceleration, then you can accelerate these parts separately by creating a new function containing those blocks and accelerating the new functions instead.

For example, create the functions `negFunAll` and `posFunAll` that contain the blocks of code used in the body of the `if` statement.

```
function dlX = negFunAll(parameters,dlX)

dlX = negFun1(parameters,dlX);
dlX = negFun2(parameters,dlX);

end

function dlX = posFunAll(parameters,dlX)

dlX = posFun1(parameters,dlX);
dlX = posFun2(parameters,dlX);

end
```

Then, accelerate these functions and use them in the body of the `if` statement instead.

```
accfunNeg = dlaccelerate(@negFunAll)
accfunPos = dlaccelerate(@posFunAll)

if sum(dlX,'all') < 0
   dlX = accfunNeg(parameters,dlX);
else
   dlX = accfunPos(parameters,dlX);
end
```

Reusing Caches

Reusing a cached trace depends on the function inputs and outputs:

- For any `dlarray` object or structure of `dlarray` object inputs, the trace depends on the size, format, and underlying datatype of the `dlarray`. That is, the accelerated function triggers a new trace for `dlarray` inputs with size, format, or underlying datatype not contained in the cache. Any `dlarray` inputs differing only by value to a previously cached trace do not trigger a new trace.
- For any `dlnetwork` inputs, the trace depends on the size, format, and underlying datatype of the `dlnetwork` state and learnable parameters. That is, the accelerated function triggers a new trace for `dlnetwork` inputs with learnable parameters or state with size, format, and underlying datatype not contained in the cache. Any `dlnetwork` inputs differing only by the value of the state and learnable parameters to a previously cached trace do not trigger a new trace.
- For other types of input, the trace depends on the values of the input. That is, the accelerated function triggers a new trace for other types of input with value not contained in the cache. Any other inputs that have the same value as a previously cached trace do not trigger a new trace.
- The trace depends on the number of function outputs. That is, the accelerated function triggers a new trace for function calls with previously unseen numbers of output arguments. Any function calls with the same number of output arguments as a previously cached trace do not trigger a new trace.

When necessary, the software caches any new traces by evaluating the underlying function and caching the resulting trace in the `AcceleratedFunction` object.

Caution An `AcceleratedFunction` object is not aware of updates to the underlying function. If you modify the function associated with the accelerated function, then clear the cache using the `clearCache` object function or alternatively use the command `clear functions`.

Storing and Clearing Caches

`AcceleratedFunction` objects store the cache in a queue:

- The software adds new traces to the back of the queue.
- When the cache is full, the software discards the cached item at the head of the queue.
- When a cache is reused, the software moves the cached item towards the back of the queue. This helps prevents the software from discarding commonly reused cached items.

The `AcceleratedFunction` objects do not directly hold the cache. This means that:

- Multiple `AcceleratedFunction` objects that have the same underlying function share the same cache.
- Clearing or overwriting a variable containing an `AcceleratedFunction` object does not clear the cache.
- Overwriting a variable containing an `AcceleratedFunction` with another `AcceleratedFunction` with the same underlying function does not clear the cache.

Accelerated functions that have the same underlying function share the same cache.

To clear the cache of an accelerated function, use the `clearCache` object function. Alternatively, you can clear all functions in the current MATLAB session using the commands `clear functions` or `clear all`.

Note Clearing the `AcceleratedFunction` variable does not clear the cache associated with the input function. To clear the cache for an `AcceleratedFunction` object that no longer exists in the workspace, create a new `AcceleratedFunction` object to the same function, and use the `clearCache` function on the new object. Alternatively, you can clear all functions in the current MATLAB session using the commands `clear functions` or `clear all`.

Acceleration Considerations

Because of the nature of caching traces, not all functions support acceleration.

The caching process can cache values that you might expect to change or that depend on external factors. You must take care when accelerating functions that:

- have inputs with random or frequently changing values
- have outputs with frequently changing values
- generate random numbers
- use `if` statements and `while` loops with conditions that depend on the values of `dlarray` objects
- have inputs that are handles or that depend on handles
- Read data from external sources (for example, by using a datastore or a `minibatchqueue` object)

Accelerated functions can do the following when calculating a new trace only.

- modify the global state such as, the random number stream or global variables
- use file input or output

- display data using graphics or the command line display

When using accelerated functions in parallel, such as when using a `parfor` loop, then each worker maintains its own cache. The cache is not transferred to the host.

Functions and custom layers used in accelerated functions must also support acceleration.

Function Inputs with Random or Frequently Changing Values

You must take care when accelerating functions that take random or frequently changing values as input, such as a model gradients function that takes random noise as input and adds it to the input data. If any random or frequently changing inputs to an accelerated function are not `dlarray` objects, then the function trigger a new trace for each previously unseen value.

You can check for scenarios like this by inspecting the `Occupancy` and `HitRate` properties of the `AcceleratedFunction` object. If the `Occupancy` property is high and the `HitRate` is low, then this can indicate that the `AcceleratedFunction` object creates many new traces that it does not reuse.

For `dlarray` object input, changes in value to not trigger new traces. To prevent frequently changing input from triggering new traces for each evaluation, refactor your code such that the random inputs are `dlarray` inputs.

For example, consider the model gradients function that accepts a random array of noise values:

```
function [gradients,state,loss] = modelGradients(parameters,dlX,dlT,state,noise)

dlX = dlX + noise;
[dlYPred,state] = model(parameters,dlX,state);
loss = crossentropy(dlYPred,dlT);
gradients = dlgradient(loss,parameters);

end
```

To accelerate this model gradients function, convert the input `noise` to `dlarray` before evaluating the accelerated function. Because the `modelGradients` function also supports `dlarray` input for noise, you do not need to make changes to the function.

```
noise = dlarray(noise,'SSCB');
accfun = dlaccelerate(@modelGradients);
[gradients,state,loss] = dlfeval(accfun,parameters,dlX,dlT,state,noise);
```

Alternatively, you can accelerate the parts of the model gradients function that do not require the random input.

Functions with Random Number Generation

You must take care when accelerating functions that use random number generation, such as a model gradients function that generates random noise to add to the network input. When caching the trace of a function that generates random numbers that are not `dlarray` objects, the accelerated function caches resulting random numbers in the trace. When reusing the trace, the accelerated function uses the cached random values. The accelerated function does not generate new random values.

Random number generation using the `'like'` option of the `rand` function with a `dlarray` object supports acceleration. To use random number generation in an accelerated function, ensure that the function uses the `rand` function with the `'like'` option set to a traced `dlarray` object (a `dlarray` object that depends on an input `dlarray` object).

For example, consider the following model gradients function.

```
[gradients,state,loss] = modelGradients(parameters,dlX,dlT,state)

sz = size(dlX);
noise = rand(sz);
dlX = dlX + noise;

[dlYPred,state] = model(parameters,dlX,state);
loss = crossentropy(dlYPred,dlT);
gradients = dlgradient(loss,parameters);

end
```

To ensure that the `rand` function generates a new value for each evaluation, use the `'like'` option with the traced `dlarray` object `dlX`.

```
[gradients,state,loss] = modelGradients(parameters,dlX,dlT,state)

sz = size(dlX);
noise = rand(sz,'like',dlX);
dlX = dlX + noise;

[dlYPred,state] = model(parameters,dlX);
loss = crossentropy(dlYPred,dlT);
gradients = dlgradient(loss,parameters);

end
```

Alternatively, you can accelerate the parts of the model gradients function that do not require random number generation.

Using if Statements and while Loops

You must take care when accelerating functions that use `if` statements and `while` loops. In particular, `if` statements and `while` loops support acceleration only when the conditions do not depend on the values of `dlarray` objects.

Attempting to accelerate functions that have `if` statements and `while` loops with conditions that depend on `dlarray` values can lead to unexpected behavior. When the accelerated function caches a new trace, if the trace contains an `if` statement or `while` loop condition that depends on the value of a `dlarray` object, then the function caches the trace of the resulting code path given by the `if` statement or `while` loop for that value. Because changes in value of `dlarray` input do not trigger a new trace, when reusing the trace, the function uses the same cached trace (which represents the same code path) even when difference in value can result in a different code path.

To use `if` statements and `while` loops that depend on `dlarray` object values, accelerate the body of the `if` statement or `while` loop only.

Function Inputs that Depend on Handles

You must take care when accelerating functions that take objects that depend on handles as input, such as a `minibatchqueue` object that has a preprocessing function specified as a function handle. The `AcceleratedFunction` object throws an error when evaluating the function with inputs depending on handles.

Instead, you can accelerate the parts of the model gradients function that do not require inputs that depend on handles.

Debugging

You must take care when debugging accelerated functions. Cached traces do not support break points. When using accelerated functions, the software reaches break points in the underlying function during the tracing process only.

To debug the code in the underlying function using breakpoints, disable the acceleration by setting the `Enabled` property to `false`.

To debug the cached traces, you can compare the outputs of the accelerated functions with the outputs of the underlying function, by setting the `CheckMode` property to `'tolerance'`.

dlode45 Does Not Support Acceleration When GradientMode is "direct"

The `dlaccelerate` function does not support accelerating the `dlode45` function when the `GradientMode` option is `"direct"`. The resulting accelerated function might return unexpected results. To accelerate the code that calls the `dlode45` function, set the `GradientMode` option to `"adjoint"` or accelerate parts of your code that do not call the `dlode45` with the `GradientMode` option set to `"direct"`.

See Also

`dlaccelerate` | `AcceleratedFunction` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Related Examples

- “Accelerate Custom Training Loop Functions” on page 18-311
- “Check Accelerated Deep Learning Function Outputs” on page 18-338
- “Evaluate Performance of Accelerated Deep Learning Function” on page 18-323

Accelerate Custom Training Loop Functions

This example shows how to accelerate deep learning custom training loop and prediction functions.

When using the `dlfeval` function in a custom training loop, the software traces each input `dlarray` object of the model gradients function to determine the computation graph used for automatic differentiation. This tracing process can take some time and can spend time recomputing the same trace. By optimizing, caching, and reusing the traces, you can speed up gradient computation in deep learning functions. You can also optimize, cache, and reuse traces to accelerate other deep learning functions that do not require automatic differentiation, for example you can also accelerate model functions and functions used for prediction.

To speed up calls to deep learning functions, use the `dlaccelerate` function to create an `AcceleratedFunction` object that automatically optimizes, caches, and reuses the traces. You can use the `dlaccelerate` function to accelerate model functions and model gradients functions directly, or to accelerate subfunctions used by these functions.

The returned `AcceleratedFunction` object caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` object, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

Load Training and Test Data

The `digitTrain4DArrayData` function loads the images, their digit labels, and their angles of rotation from the vertical. Create `arrayDatastore` objects for the images, labels, and angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and number of nondiscrete responses.

```
[imagesTrain,labelsTrain,anglesTrain] = digitTrain4DArrayData;

dsImagesTrain = arrayDatastore(imagesTrain,'IterationDimension',4);
dsLabelsTrain = arrayDatastore(labelsTrain);
dsAnglesTrain = arrayDatastore(anglesTrain);

dsTrain = combine(dsImagesTrain,dsLabelsTrain,dsAnglesTrain);

classNames = categories(labelsTrain);
numClasses = numel(classNames);
numResponses = size(anglesTrain,2);
numObservations = numel(labelsTrain);
```

View some images from the training data.

```
idx = randperm(numObservations,64);
I = imtile(imagesTrain(:,:,:,idx));
figure
imshow(I)
```

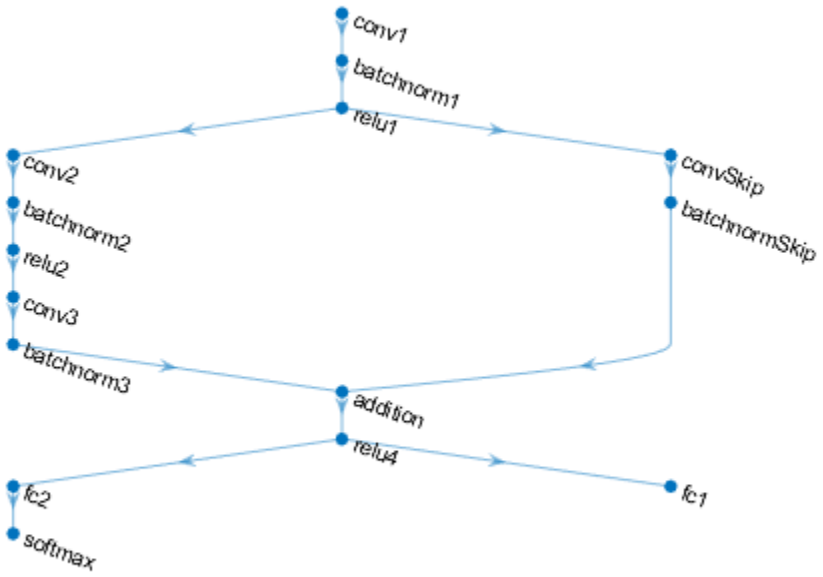
Create a datastore containing the test data given by the `digitTest4DArrayData` function using the same steps.

```
[imagesTest,labelsTest,anglesTest] = digitTest4DArrayData;

dsImagesTest = arrayDatastore(imagesTest,'IterationDimension',4);
dsLabelsTest = arrayDatastore(labelsTest);
dsAnglesTest = arrayDatastore(anglesTest);

dsTest = combine(dsImagesTest,dsLabelsTest,dsAnglesTest);
```

Define Deep Learning Model

Define the following network that predicts both labels and angles of rotation.

- A convolution-batchnorm-ReLU block with 16 5-by-5 filters.
- A branch of two convolution-batchnorm blocks each with 32 3-by-3 filters with a ReLU operation between
- A skip connection with a convolution-batchnorm block with 32 1-by-1 convolutions.
- Combine both branches using addition followed by a ReLU operation
- For the regression output, a branch with a fully connected operation of size 1 (the number of responses).
- For classification output, a branch with a fully connected operation of size 10 (the number of classes) and a softmax operation.



Define and Initialize Model Parameters and State

Create structures `parameters` and `state` that contain the initialized model parameters and state, respectively, using the `modelParameters` function, listed in the Model Parameters Function on page 18-0 section of the example.

The output uses the format `parameters.OperationName.ParameterName` where `parameters` is the structure, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

```
[parameters,state] = modelParameters(numClasses,numResponses);
```

Define Model Function

Create the function `model`, listed at the end of the example, that computes the outputs of the deep learning model described earlier.

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the

labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

Specify Training Options

Specify the training options. Train for 20 epochs with a mini-batch size of 32. Displaying the plot can make training take longer to complete. Disable the plot by setting the `plots` variable to `"none"`. To enable the plot, set this variable to `"training-progress"`.

```
numEpochs = 20;
miniBatchSize = 32;
plots = "none";
```

Train Accelerated Model

Accelerate the model gradients function using the `dlaccelerate` function.

```
accfun = dlaccelerate(@modelGradients);
```

Clear any previoulsy cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or angles.
- Discard any partial mini-batches returned at the end of an epoch.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''}, ...
    'PartialMiniBatch','discard');
```

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

If required, initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model using the accelerated model gradients function. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the accelerated model gradients function.
- Update the network parameters using the `adamupdate` function.
- If required, update the training progress plot.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,dlT1,dlT2] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % accelerated function.
        [gradients,state,loss] = dlfeval(accfun, parameters, dlX, dlT1, dlT2, state);

        % Update the network parameters using the Adam optimizer.
        [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
            trailingAvg,trailingAvgSq,iteration);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            loss = double(gather(extractdata(loss)));
            addpoints(lineLossTrain,iteration,loss)
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
```

Check the efficiency of the accelerated function by inspecting the `HitRate` property. The `HitRate` property contains the percentage of function calls that reuse a cached trace.

```
accfun.HitRate
```

```
ans = 99.9679
```

Accelerate Predictions

Measure the time required to make predictions using the test data set.

Because the model predictions function requires a mini-batch queue as input, the function does not support acceleration. To speed up prediction, accelerate the model function.

Accelerate the model function using the `dlaccelerate` function.

```
accfun2 = dlaccelerate(@model);
```

Clear any previoulsy cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun2)
```

After training, making predictions on new data does not require the labels. Create `minibatchqueue` object containing only the predictors of the test data:

- To ignore the labels for testing, set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessMiniBatchPredictors` function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCB'` (spatial, spatial, channel, batch).

```
numOutputs = 1;
mbqTest = minibatchqueue(dsTest,numOutputs, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatchPredictors, ...
    'MiniBatchFormat','SSCB');
```

Loop over the mini-batches and classify the images using the `modelPredictions` function, listed at the end of the example.

```
[labelsPred,anglesPred] = modelPredictions(accfun2,parameters,state,mbqTest,classNames);
```

Check the efficiency of the accelerated function by inspecting the `HitRate` property. The `HitRate` property contains the percentage of function calls that reuse a cached trace.

```
accfun2.HitRate
```

```
ans = 98.7261
```

Model Parameters Function

The `modelParameters` function creates structures `parameters` and `state` that contain the initialized model parameters and state, respectively for the model described in the Define Deep Learning Model on page 18-0 section. The function takes as input the number of classes and the number of responses and initializes the learnable parameters. The function:

- initializes the layer weights using the `initializeGlorot` function
- initializes the layer biases using the `initializeZeros` function
- initializes the batch normalization offset and scale parameters with the `initializeZeros` function
- initializes the batch normalization scale parameters with the `initializeOnes` function
- initializes the batch normalization state trained mean with the `initializeZeros` function
- initializes the batch normalization state trained variance with the `initializeOnes` example function

The initialization example functions are attached to this example as supporting files. To access these files, open the example as a live script. To learn more about initializing learnable parameters for deep learning models, see “Initialize Learnable Parameters for Model Function” on page 18-292.

The output uses the format `parameters.OperationName.ParameterName` where `parameters` is the structure, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

```
function [parameters,state] = modelParameters(numClasses,numResponses)

% First convolutional layer.
filterSize = [5 5];
numChannels = 1;
numFilters = 16;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv1.Bias = initializeZeros([numFilters 1]);

% First batch normalization layer.
parameters.batchnorm1.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm1.Scale = initializeOnes([numFilters 1]);
state.batchnorm1.TrainedMean = initializeZeros([numFilters 1]);
state.batchnorm1.TrainedVariance = initializeOnes([numFilters 1]);

% Second convolutional layer.
filterSize = [3 3];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv2.Bias = initializeZeros([numFilters 1]);

% Second batch normalization layer.
parameters.batchnorm2.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm2.Scale = initializeOnes([numFilters 1]);
state.batchnorm2.TrainedMean = initializeZeros([numFilters 1]);
state.batchnorm2.TrainedVariance = initializeOnes([numFilters 1]);

% Third convolutional layer.
filterSize = [3 3];
numChannels = 32;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv3.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv3.Bias = initializeZeros([numFilters 1]);

% Third batch normalization layer.
parameters.batchnorm3.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm3.Scale = initializeOnes([numFilters 1]);
state.batchnorm3.TrainedMean = initializeZeros([numFilters 1]);
```

```
state.batchnorm3.TrainedVariance = initializeOnes([numFilters 1]);

% Convolutional layer in the skip connection.
filterSize = [1 1];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.convSkip.Weights = initializeGlorot(sz,numOut,numIn);
parameters.convSkip.Bias = initializeZeros([numFilters 1]);

% Batch normalization layer in the skip connection.
parameters.batchnormSkip.Offset = initializeZeros([numFilters 1]);
parameters.batchnormSkip.Scale = initializeOnes([numFilters 1]);

state.batchnormSkip.TrainedMean = initializeZeros([numFilters 1]);
state.batchnormSkip.TrainedVariance = initializeOnes([numFilters 1]);

% Fully connected layer corresponding to the classification output.
sz = [numClasses 6272];
numOut = numClasses;
numIn = 6272;
parameters.fc1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc1.Bias = initializeZeros([numClasses 1]);

% Fully connected layer corresponding to the regression output.
sz = [numResponses 6272];
numOut = numResponses;
numIn = 6272;
parameters.fc2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc2.Bias = initializeZeros([numResponses 1]);

end
```

Model Function

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

```
function [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state)

% Convolution
weights = parameters.conv1.Weights;
bias = parameters.conv1.Bias;
dlY = dlconv(dlX,weights,bias,'Padding','same');

% Batch normalization, ReLU
offset = parameters.batchnorm1.Offset;
scale = parameters.batchnorm1.Scale;
trainedMean = state.batchnorm1.TrainedMean;
trainedVariance = state.batchnorm1.TrainedVariance;

if doTraining
```

```
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm1.TrainedMean = trainedMean;
    state.batchnorm1.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution, batch normalization (Skip connection)
weights = parameters.convSkip.Weights;
bias = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,weights,bias,'Stride',2);

offset = parameters.batchnormSkip.Offset;
scale = parameters.batchnormSkip.Scale;
trainedMean = state.batchnormSkip.TrainedMean;
trainedVariance = state.batchnormSkip.TrainedVariance;

if doTraining
    [dlYSkip,trainedMean,trainedVariance] = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVa

    % Update state
    state.batchnormSkip.TrainedMean = trainedMean;
    state.batchnormSkip.TrainedVariance = trainedVariance;
else
    dlYSkip = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVariance);
end

% Convolution
weights = parameters.conv2.Weights;
bias = parameters.conv2.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same','Stride',2);

% Batch normalization, ReLU
offset = parameters.batchnorm2.Offset;
scale = parameters.batchnorm2.Scale;
trainedMean = state.batchnorm2.TrainedMean;
trainedVariance = state.batchnorm2.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution
weights = parameters.conv3.Weights;
bias = parameters.conv3.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same');
```

```
% Batch normalization
offset = parameters.batchnorm3.Offset;
scale = parameters.batchnorm3.Scale;
trainedMean = state.batchnorm3.TrainedMean;
trainedVariance = state.batchnorm3.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm3.TrainedMean = trainedMean;
    state.batchnorm3.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

% Addition, ReLU
dlY = dlYSkip + dlY;
dlY = relu(dlY);

% Fully connect, softmax (labels)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

% Fully connect (angles)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

end
```

Model Gradients Function

The `modelGradients` function takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

```
function [gradients,state,loss] = modelGradients(parameters,dlX,T1,T2,state)

doTraining = true;
[dlY1,dlY2,state] = model(parameters,dlX,doTraining,state);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes the model parameters, state, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in

the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function [predictions1, predictions2] = modelPredictions(modelFcn,parameters,state,mbq,classes)

doTraining = false;
predictions1 = [];
predictions2 = [];

while hasdata(mbq)

    dlXTest = next(mbq);

    [dlYPred1,dlYPred2] = modelFcn(parameters,dlXTest,doTraining,state);

    YPred1 = onehotdecode(dlYPred1,classes,1)';
    YPred2 = extractdata(dlYPred2)';

    predictions1 = [predictions1; YPred1];
    predictions2 = [predictions2; YPred2];
end

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label and angle data from the incoming cell arrays and concatenate along the second dimension into a categorical array and a numeric array, respectively.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,angle] = preprocessMiniBatch(XCell,YCell,angleCell)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate
Y = cat(2,YCell{:});

% Extract angle data from cell and concatenate
angle = cat(2,angleCell{:});

% One-hot encode labels
Y = onehotencode(Y,1);

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and then concatenating them into a numeric array.

For grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

end
```

See Also

`dlaccelerate` | `AcceleratedFunction` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Related Examples

- “Deep Learning Function Acceleration for Custom Training Loops” on page 18-304
- “Check Accelerated Deep Learning Function Outputs” on page 18-338
- “Evaluate Performance of Accelerated Deep Learning Function” on page 18-323

Evaluate Performance of Accelerated Deep Learning Function

This example shows how to evaluate the performance gains of using an accelerated function.

When using the `dlfeval` function in a custom training loop, the software traces each input `dlarray` object of the model gradients function to determine the computation graph used for automatic differentiation. This tracing process can take some time and can spend time recomputing the same trace. By optimizing, caching, and reusing the traces, you can speed up gradient computation in deep learning functions. You can also optimize, cache, and reuse traces to accelerate other deep learning functions that do not require automatic differentiation, for example you can also accelerate model functions and functions used for prediction.

To speed up calls to deep learning functions, use the `dlaccelerate` function to create an `AcceleratedFunction` object that automaticallyoptimizes, caches, and reuses the traces. You can use the `dlaccelerate` function to accelerate model functions and model gradients functions directly, or to accelerate subfunctions used by these functions. The performance gains are most noticeable for deeper networks and training loops with many epochs and iterations.

The returned `AcceleratedFunction` object caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` object, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

This example compares training and prediction times when using and not using acceleration.

Load Training and Test Data

The `digitTrain4DArrayData` function loads the images, their digit labels, and their angles of rotation from the vertical. Create `arrayDatastore` objects for the images, labels, and angles, and then use the `combine` function to make a single datastore that contains all of the training data. Extract the class names and number of nondiscrete responses.

```
[imagesTrain,labelsTrain,anglesTrain] = digitTrain4DArrayData;

dsImagesTrain = arrayDatastore(imagesTrain,'IterationDimension',4);
dsLabelsTrain = arrayDatastore(labelsTrain);
dsAnglesTrain = arrayDatastore(anglesTrain);

dsTrain = combine(dsImagesTrain,dsLabelsTrain,dsAnglesTrain);

classNames = categories(labelsTrain);
numClasses = numel(classNames);
numResponses = size(anglesTrain,2);
numObservations = numel(labelsTrain);
```

Create a datastore containing the test data given by the `digitTest4DArrayData` function using the same steps.

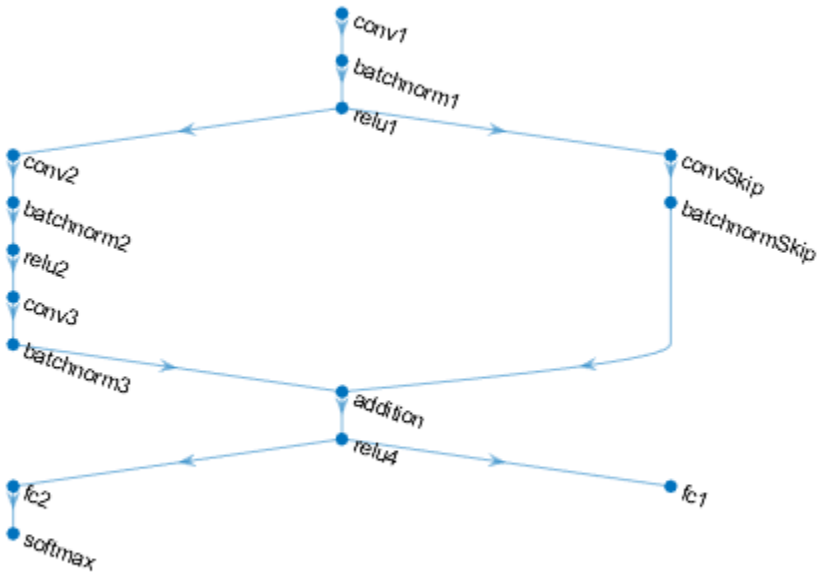
```
[imagesTest,labelsTest,anglesTest] = digitTest4DArrayData;

dsImagesTest = arrayDatastore(imagesTest,'IterationDimension',4);
```

```
dsLabelsTest = arrayDatastore(labelsTest);
dsAnglesTest = arrayDatastore(anglesTest);

dsTest = combine(dsImagesTest,dsLabelsTest,dsAnglesTest);
```

Define Deep Learning Model

Define the following network that predicts both labels and angles of rotation.

- A convolution-batchnorm-ReLU block with 16 5-by-5 filters.
- A branch of two convolution-batchnorm blocks each with 32 3-by-3 filters with a ReLU operation between
- A skip connection with a convolution-batchnorm block with 32 1-by-1 convolutions.
- Combine both branches using addition followed by a ReLU operation
- For the regression output, a branch with a fully connected operation of size 1 (the number of responses).
- For classification output, a branch with a fully connected operation of size 10 (the number of classes) and a softmax operation.



Define and Initialize Model Parameters and State

Create a struct `parametersBaseline` containing the model parameters using the `modelParameters` function, listed at the end of the example. The `modelParameters` function creates structures `parameters` and `state` that contain the initialized model parameters and state, respectively.

The output uses the format `parameters.OperationName.ParameterName` where `parameters` is the structure, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

```
[parametersBaseline,stateBaseline] = modelParameters(numClasses,numResponses);
```

Create a copy of the parameters and state for the baseline model to use for the accelerated model.

```
parametersAccelerated = parametersBaseline;
stateAccelerated = stateBaseline;
```

Define Model Function

Create the function `model`, listed at the end of the example, that computes the outputs of the deep learning model described earlier.

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

Specify Training Options

Specify the training options. Train for 20 epochs with a mini-batch size of 32. Displaying the plot can make training take longer to complete. Disable the plot by setting the `plots` variable to `"none"`. To enable the plot, set this variable to `"training-progress"`.

```
numEpochs = 20;
miniBatchSize = 32;
plots = "none";
```

Train Baseline Model

Use `minibatchqueue` to process and manage the mini-batches of images. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels or angles.
- Discard any partial mini-batches returned at the end of an epoch.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
mbq = minibatchqueue(dsTrain,...
    'MiniBatchSize',miniBatchSize,...
```

```
    'MiniBatchFcn',@preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB','',''}, ...
    'PartialMiniBatch','discard');
```

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

If required, initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.
- Update the training progress plot.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,dlT1,dlT2] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % model gradients function.
        [gradients,stateBaseline,loss] = dlfeval(@modelGradients,parametersBaseline,dlX,dlT1,dlT

        % Update the network parameters using the Adam optimizer.
        [parametersBaseline,trailingAvg,trailingAvgSq] = adamupdate(parametersBaseline,gradients,
            trailingAvg,trailingAvgSq,iteration);

        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            loss = double(gather(extractdata(loss)));
            addpoints(lineLossTrain,iteration,loss)
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
```

```
            drawnow
        end
    end
end
elapsedBaseline = toc(start)
```

```
elapsedBaseline = 285.8978
```

Train Accelerated Model

Accelerate the model gradients function using the `dlaccelerate` function.

```
accfun = dlaccelerate(@modelGradients);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun)
```

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

If required, initialize the training progress plot.

```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    ylim([0 inf])
    xlabel("Iteration")
    ylabel("Loss")
    grid on
end
```

Train the model using the accelerated model gradients function in the call to the `dlfeval` function.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches
    while hasdata(mbq)

        iteration = iteration + 1;

        [dlX,dlT1,dlT2] = next(mbq);

        % Evaluate the model gradients, state, and loss using dlfeval and the
        % accelerated function.
        [gradients,stateAccelerated,loss] = dlfeval(accfun, parametersAccelerated, dlX, dlT1, dl

        % Update the network parameters using the Adam optimizer.
        [parametersAccelerated,trailingAvg,trailingAvgSq] = adamupdate(parametersAccelerated,grad
            trailingAvg,trailingAvgSq,iteration);
```

```
        % Display the training progress.
        if plots == "training-progress"
            D = duration(0,0,toc(start),'Format','hh:mm:ss');
            loss = double(gather(extractdata(loss)));
            addpoints(lineLossTrain,iteration,loss)
            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end
end
elapsedAccelerated = toc(start)
```

```
elapsedAccelerated = 188.5316
```

Check the efficiency of the accelerated function by inspecting the `HitRate` property. The `HitRate` property contains the percentage of function calls that reuse a cached trace.

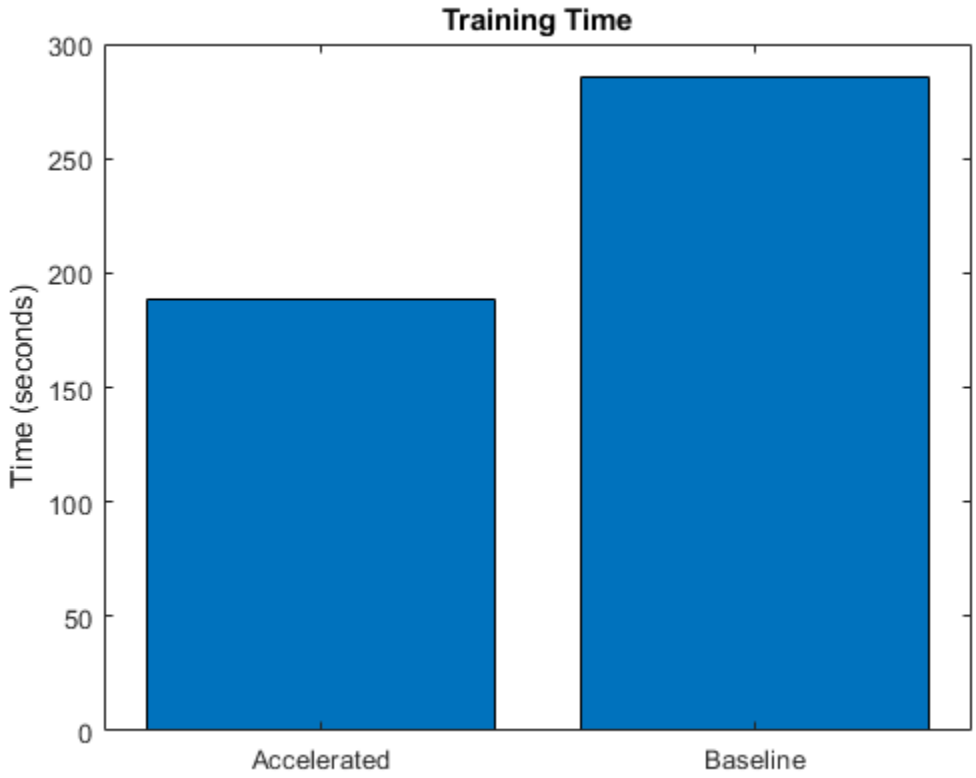
```
accfun.HitRate
```

```
ans = 99.9679
```

Compare Training Times

Compare the training times in a bar chart.

```
figure
bar(categorical(["Baseline" "Accelerated"]),[elapsedBaseline elapsedAccelerated]);
ylabel("Time (seconds)")
title("Training Time")
```



Calculate the speedup of acceleration.

```
speedup = elapsedBaseline / elapsedAccelerated
```

```
speedup = 1.5164
```

Time Baseline Predictions

Measure the time required to make predictions using the test data set.

After training, making predictions on new data does not require the labels. Create `minibatchqueue` object containing only the predictors of the test data:

- To ignore the labels for testing, set the number of outputs of the mini-batch queue to 1.
- Specify the same mini-batch size used for training.
- Preprocess the predictors using the `preprocessMiniBatchPredictors` function, listed at the end of the example.
- For the single output of the datastore, specify the mini-batch format `'SSCB'` (spatial, spatial, channel, batch).

```
numOutputs = 1;
mbqTest = minibatchqueue(dsTest,numOutputs, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFcn',@preprocessMiniBatchPredictors, ...
    'MiniBatchFormat','SSCB');
```

Loop over the mini-batches and classify the images using the `modelPredictions` function, listed at the end of the example and measure the elapsed time.

```
tic
[labelsPred,anglesPred] = modelPredictions(@model,parametersBaseline,stateBaseline,mbqTest,classN
elapsedPredictionBaseline = toc
```

```
elapsedPredictionBaseline = 5.5070
```

Time Accelerated Predictions

Because the model predictions function requires a mini-batch queue as input, the function does not support acceleration. To speed up prediction, accelerate the model function.

Accelerate the model function using the `dlaccelerate` function.

```
accfun2 = dlaccelerate(@model);
```

Clear any previously cached traces of the accelerated function using the `clearCache` function.

```
clearCache(accfun2)
```

Reset the mini-batch queue.

```
reset(mbqTest)
```

Loop over the mini-batches and classify the images using the `modelPredictions` function, listed at the end of the example and measure the elapsed time.

```
tic
[labelsPred,anglesPred] = modelPredictions(accfun2,parametersBaseline,stateBaseline,mbqTest,class
elapsedPredictionAccelerated = toc
```

```
elapsedPredictionAccelerated = 4.3057
```

Check the efficiency of the accelerated function by inspecting the `HitRate` property. The `HitRate` property contains the percentage of function calls that reuse a cached trace.

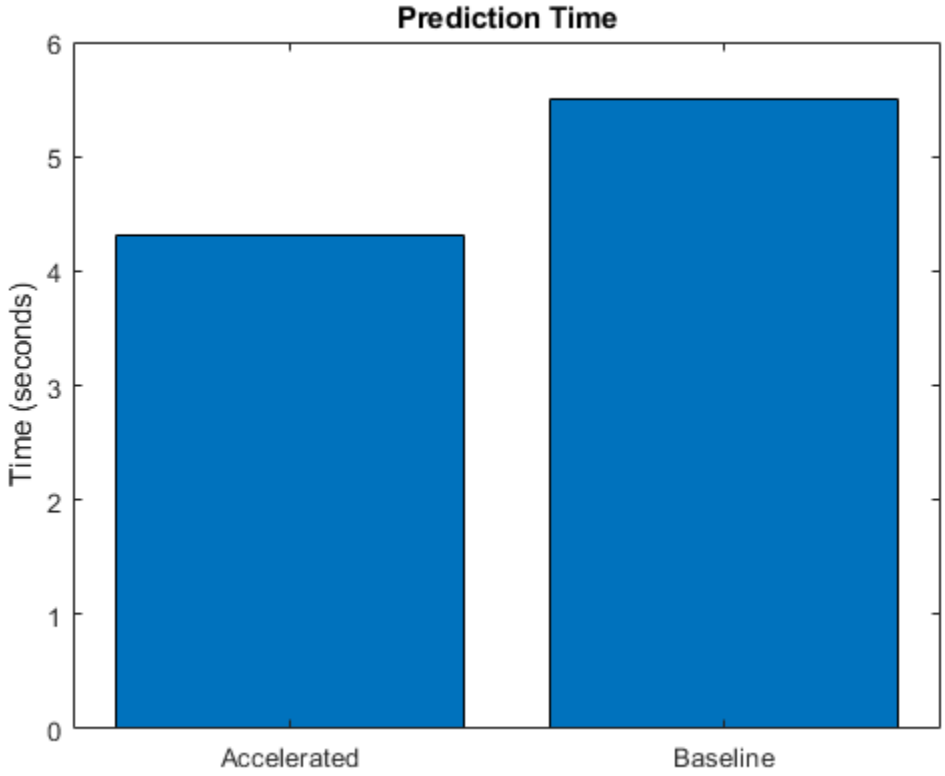
```
accfun2.HitRate
```

```
ans = 98.7261
```

Compare Prediction Times

Compare the prediction times in a bar chart.

```
figure
bar(categorical(["Baseline" "Accelerated"]),[elapsedPredictionBaseline elapsedPredictionAccelerat
ylabel("Time (seconds)")
title("Prediction Time")
```



Calculate the speedup of acceleration.

```
speedup = elapsedPredictionBaseline / elapsedPredictionAccelerated
```

```
speedup = 1.2790
```

Model Parameters Function

The `modelParameters` function creates structures `parameters` and `state` that contain the initialized model parameters and state, respectively for the model described in the Define Deep Learning Model on page 18-0 section. The function takes as input the number of classes and the number of responses and initializes the learnable parameters. The function:

- initializes the layer weights using the `initializeGlorot` function
- initializes the layer biases using the `initializeZeros` function
- initializes the batch normalization offset and scale parameters with the `initializeZeros` function
- initializes the batch normalization scale parameters with the `initializeOnes` function
- initializes the batch normalization state trained mean with the `initializeZeros` function
- initializes the batch normalization state trained variance with the `initializeOnes` example function

The initialization example functions are attached to this example as supporting files. To access these files, open the example as a live script. To learn more about initializing learnable parameters for deep learning models, see “Initialize Learnable Parameters for Model Function” on page 18-292.

The output uses the format `parameters.OperationName.ParameterName` where `parameters` is the structure, `OperationName` is the name of the operation (for example "conv1") and `ParameterName` is the name of the parameter (for example, "Weights").

```
function [parameters,state] = modelParameters(numClasses,numResponses)

% First convolutional layer.
filterSize = [5 5];
numChannels = 1;
numFilters = 16;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv1.Bias = initializeZeros([numFilters 1]);

% First batch normalization layer.
parameters.batchnorm1.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm1.Scale = initializeOnes([numFilters 1]);
state.batchnorm1.TrainedMean = initializeZeros([numFilters 1]);
state.batchnorm1.TrainedVariance = initializeOnes([numFilters 1]);

% Second convolutional layer.
filterSize = [3 3];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv2.Bias = initializeZeros([numFilters 1]);

% Second batch normalization layer.
parameters.batchnorm2.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm2.Scale = initializeOnes([numFilters 1]);
state.batchnorm2.TrainedMean = initializeZeros([numFilters 1]);
state.batchnorm2.TrainedVariance = initializeOnes([numFilters 1]);

% Third convolutional layer.
filterSize = [3 3];
numChannels = 32;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.conv3.Weights = initializeGlorot(sz,numOut,numIn);
parameters.conv3.Bias = initializeZeros([numFilters 1]);

% Third batch normalization layer.
parameters.batchnorm3.Offset = initializeZeros([numFilters 1]);
parameters.batchnorm3.Scale = initializeOnes([numFilters 1]);
state.batchnorm3.TrainedMean = initializeZeros([numFilters 1]);
```

```
state.batchnorm3.TrainedVariance = initializeOnes([numFilters 1]);

% Convolutional layer in the skip connection.
filterSize = [1 1];
numChannels = 16;
numFilters = 32;

sz = [filterSize numChannels numFilters];
numOut = prod(filterSize) * numFilters;
numIn = prod(filterSize) * numFilters;

parameters.convSkip.Weights = initializeGlorot(sz,numOut,numIn);
parameters.convSkip.Bias = initializeZeros([numFilters 1]);

% Batch normalization layer in the skip connection.
parameters.batchnormSkip.Offset = initializeZeros([numFilters 1]);
parameters.batchnormSkip.Scale = initializeOnes([numFilters 1]);

state.batchnormSkip.TrainedMean = initializeZeros([numFilters 1]);
state.batchnormSkip.TrainedVariance = initializeOnes([numFilters 1]);

% Fully connected layer corresponding to the classification output.
sz = [numClasses 6272];
numOut = numClasses;
numIn = 6272;
parameters.fc1.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc1.Bias = initializeZeros([numClasses 1]);

% Fully connected layer corresponding to the regression output.
sz = [numResponses 6272];
numOut = numResponses;
numIn = 6272;
parameters.fc2.Weights = initializeGlorot(sz,numOut,numIn);
parameters.fc2.Bias = initializeZeros([numResponses 1]);

end
```

Model Function

The function `model` takes the model parameters `parameters`, the input data `dlX`, the flag `doTraining` which specifies whether to model should return outputs for training or prediction, and the network state `state`. The network outputs the predictions for the labels, the predictions for the angles, and the updated network state.

```
function [dlY1,dlY2,state] = model(parameters,dlX,doTraining,state)

% Convolution
weights = parameters.conv1.Weights;
bias = parameters.conv1.Bias;
dlY = dlconv(dlX,weights,bias,'Padding','same');

% Batch normalization, ReLU
offset = parameters.batchnorm1.Offset;
scale = parameters.batchnorm1.Scale;
trainedMean = state.batchnorm1.TrainedMean;
trainedVariance = state.batchnorm1.TrainedVariance;

if doTraining
```

```
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm1.TrainedMean = trainedMean;
    state.batchnorm1.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution, batch normalization (Skip connection)
weights = parameters.convSkip.Weights;
bias = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,weights,bias,'Stride',2);

offset = parameters.batchnormSkip.Offset;
scale = parameters.batchnormSkip.Scale;
trainedMean = state.batchnormSkip.TrainedMean;
trainedVariance = state.batchnormSkip.TrainedVariance;

if doTraining
    [dlYSkip,trainedMean,trainedVariance] = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVa

    % Update state
    state.batchnormSkip.TrainedMean = trainedMean;
    state.batchnormSkip.TrainedVariance = trainedVariance;
else
    dlYSkip = batchnorm(dlYSkip,offset,scale,trainedMean,trainedVariance);
end

% Convolution
weights = parameters.conv2.Weights;
bias = parameters.conv2.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same','Stride',2);

% Batch normalization, ReLU
offset = parameters.batchnorm2.Offset;
scale = parameters.batchnorm2.Scale;
trainedMean = state.batchnorm2.TrainedMean;
trainedVariance = state.batchnorm2.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

dlY = relu(dlY);

% Convolution
weights = parameters.conv3.Weights;
bias = parameters.conv3.Bias;
dlY = dlconv(dlY,weights,bias,'Padding','same');
```

```
% Batch normalization
offset = parameters.batchnorm3.Offset;
scale = parameters.batchnorm3.Scale;
trainedMean = state.batchnorm3.TrainedMean;
trainedVariance = state.batchnorm3.TrainedVariance;

if doTraining
    [dlY,trainedMean,trainedVariance] = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);

    % Update state
    state.batchnorm3.TrainedMean = trainedMean;
    state.batchnorm3.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,offset,scale,trainedMean,trainedVariance);
end

% Addition, ReLU
dlY = dlYSkip + dlY;
dlY = relu(dlY);

% Fully connect, softmax (labels)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

% Fully connect (angles)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

end
```

Model Gradients Function

The `modelGradients` function, takes the model parameters, a mini-batch of input data `dlX` with corresponding targets `T1` and `T2` containing the labels and angles, respectively, and returns the gradients of the loss with respect to the learnable parameters, the updated network state, and the corresponding loss.

```
function [gradients,state,loss] = modelGradients(parameters,dlX,T1,T2,state)

doTraining = true;
[dlY1,dlY2,state] = model(parameters,dlX,doTraining,state);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,parameters);

end
```

Model Predictions Function

The `modelPredictions` function takes the model parameters, state, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in

the `minibatchqueue` object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function [predictions1, predictions2] = modelPredictions(modelFcn,parameters,state,mbq,classes)

doTraining = false;
predictions1 = [];
predictions2 = [];

while hasdata(mbq)

    dlXTest = next(mbq);

    [dlYPred1,dlYPred2] = modelFcn(parameters,dlXTest,doTraining,state);

    YPred1 = onehotdecode(dlYPred1,classes,1)';
    YPred2 = extractdata(dlYPred2)';

    predictions1 = [predictions1; YPred1];
    predictions2 = [predictions2; YPred2];
end

end
```

Mini-Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label and angle data from the incoming cell arrays and concatenate along the second dimension into a categorical array and a numeric array, respectively.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y,angle] = preprocessMiniBatch(XCell,YCell,angleCell)

% Preprocess predictors.
X = preprocessMiniBatchPredictors(XCell);

% Extract label data from cell and concatenate
Y = cat(2,YCell{:});

% Extract angle data from cell and concatenate
angle = cat(2,angleCell{:});

% One-hot encode labels
Y = onehotencode(Y,1);

end
```

Mini-Batch Predictors Preprocessing Function

The `preprocessMiniBatchPredictors` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For

grayscale input, concatenating over the fourth dimension adds a third dimension to each image, to use as a singleton channel dimension.

```
function X = preprocessMiniBatchPredictors(XCell)

% Concatenate.
X = cat(4,XCell{1:end});

end
```

See Also

`dlaccelerate` | `AcceleratedFunction` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Related Examples

- “Deep Learning Function Acceleration for Custom Training Loops” on page 18-304
- “Accelerate Custom Training Loop Functions” on page 18-311
- “Check Accelerated Deep Learning Function Outputs” on page 18-338

Check Accelerated Deep Learning Function Outputs

This example shows how to check that the outputs of accelerated functions match the outputs of the underlying function.

In some cases, the outputs of accelerated functions differ to the outputs of the underlying function. For example, you must take care when accelerating functions that use random number generation, such as a function that generates random noise to add to the network input. When caching the trace of a function that generates random numbers that are not `dlarray` objects, the accelerated function caches resulting random numbers in the trace. When reusing the trace, the accelerated function uses the cached random values. The accelerated function does not generate new random values.

To check that the outputs of the accelerated function match the outputs of the underlying function, use the `CheckMode` property of the accelerated function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ by more than a specified tolerance, the accelerated function throws a warning.

Accelerate the function `myUnsupportedFun`, listed at the end of the example using the `dlaccelerate` function. The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
accfun = dlaccelerate(@myUnsupportedFun)
```

```
accfun =
  AcceleratedFunction with properties:

         Function: @myUnsupportedFun
          Enabled: 1
        CacheSize: 50
          HitRate: 0
        Occupancy: 0
        CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun.CheckMode = 'tolerance'
```

```
accfun =
  AcceleratedFunction with properties:

         Function: @myUnsupportedFun
          Enabled: 1
        CacheSize: 50
          HitRate: 0
        Occupancy: 0
        CheckMode: 'tolerance'
    CheckTolerance: 1.0000e-04
```

Evaluate the accelerated function with an array of ones as input, specified as a `dlarray` input.

```
dlX = dlarray(ones(3,3));
dlY = accfun(dlX)
```

```
dlY =
  3x3 dlarray

    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Evaluate the accelerated function again with the same input. Because the accelerated function reuses the cached random noise values instead of generating new random values, the outputs of the reused trace differs from the outputs of the underlying function. When the `CheckMode` property of the accelerated function is `'tolerance'` and the outputs differ, the accelerated function throws a warning.

```
dlY = accfun(dlX)
```

```
Warning: Accelerated outputs differ from underlying function outputs.
```

```
dlY =
  3x3 dlarray

    1.8147    1.9134    1.2785
    1.9058    1.6324    1.5469
    1.1270    1.0975    1.9575
```

Random number generation using the `'like'` option of the `rand` function with a `dlarray` object supports acceleration. To use random number generation in an accelerated function, ensure that the function uses the `rand` function with the `'like'` option set to a traced `dlarray` object (a `dlarray` object that depends on an input `dlarray` object).

Accelerate the function `mySupportedFun`, listed at the end of the example. The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
accfun2 = dlaccelerate(@mySupportedFun);
```

Cleare any previously cached traces using the `clearCache` function.

```
clearCache(accfun2)
```

To check that the outputs of reused cached traces match the outputs of the underlying function, set the `CheckMode` property to `'tolerance'`.

```
accfun2.CheckMode = 'tolerance';
```

Evaluate the accelerated function twice with the same input as before. Because the outputs of the reused cache match the outputs of the underlying function, the accelerated function does not throw a warning.

```
dlY = accfun2(dlX)
```

```
dlY =
  3x3 dlarray
```

```
    1.7922    1.0357    1.6787
    1.9595    1.8491    1.7577
    1.6557    1.9340    1.7431


dlY = accfun2(dlX)

dlY =
  3x3 dlarray

    1.3922    1.7060    1.0462
    1.6555    1.0318    1.0971
    1.1712    1.2769    1.8235
```

Checking the outputs match requires extra processing and increases the time required for function evaluation. After checking the outputs, set the `CheckMode` property to `'none'`.

```
accfun1.CheckMode = 'none';
accfun2.CheckMode = 'none';
```

Example Functions

The function `myUnsupportedFun` generates random noise and adds it to the input. This function does not support acceleration because the function generates random numbers that are not `dlarray` objects.

```
function out = myUnsupportedFun(dlX)

sz = size(dlX);
noise = rand(sz);
out = dlX + noise;

end
```

The function `mySupportedFun` adds noise to the input by generating noise using the `'like'` option with a traced `dlarray` object.

```
function out = mySupportedFun(dlX)

sz = size(dlX);
noise = rand(sz,'like',dlX);
out = dlX + noise;

end
```

See Also

`dlaccelerate` | `AcceleratedFunction` | `clearCache` | `dlarray` | `dlgradient` | `dlfeval`

Related Examples

- “Deep Learning Function Acceleration for Custom Training Loops” on page 18-304
- “Accelerate Custom Training Loop Functions” on page 18-311
- “Evaluate Performance of Accelerated Deep Learning Function” on page 18-323

Solve Partial Differential Equations Using Deep Learning

This example shows how to solve Burger's equation using deep learning.

The Burger's equation is a partial differential equation (PDE) that arises in different areas of applied mathematics. In particular, fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flows.

Given the computational domain$[-1,1] \times [0,1]$, this examples uses a physics informed neural network (PINN) [1] and trains a multilayer perceptron neural network that takes samples (x,t) as input, where $x \in [-1,1]$ is the spatial variable, and $t \in [0,1]$ is the time variable, and returns $u(x,t)$, where u is the solution of the Burger's equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2} = 0,$$

with $u(x, t = 0) = -sin(\pi x)$as the initial condition, and $u(x = -1, t) = 0$ and $u(x = 1, t) = 0$ as the boundary conditions.

The example trains the model by enforcing that given an input (x,t), the output of the network $u(x,t)$ fulfills the Burger's equation, the boundary conditions, and the initial condition.

Training this model does not require collecting data in advance. You can generate data using the definition of the PDE and the constraints.

Generate Training Data

Training the model requires a data set of collocation points that enforce the boundary conditions, enforce the initial conditions, and fulfill the Burger's equation.

Select 25 equally spaced time points to enforce each of the boundary conditions $u(x = -1, t) = 0$ and $u(x = 1, t) = 0$.

```
numBoundaryConditionPoints = [25 25];

x0BC1 = -1*ones(1,numBoundaryConditionPoints(1));
x0BC2 = ones(1,numBoundaryConditionPoints(2));

t0BC1 = linspace(0,1,numBoundaryConditionPoints(1));
t0BC2 = linspace(0,1,numBoundaryConditionPoints(2));

u0BC1 = zeros(1,numBoundaryConditionPoints(1));
u0BC2 = zeros(1,numBoundaryConditionPoints(2));
```

Select 50 equally spaced spatial points to enforce the initial condition $u(x, t = 0) = -sin(\pi x)$.

```
numInitialConditionPoints  = 50;

x0IC = linspace(-1,1,numInitialConditionPoints);
t0IC = zeros(1,numInitialConditionPoints);
u0IC = -sin(pi*x0IC);
```

Group together the data for initial and boundary conditions.

```
X0 = [x0IC x0BC1 x0BC2];
T0 = [t0IC t0BC1 t0BC2];
U0 = [u0IC u0BC1 u0BC2];
```

Select 10,000 points to enforce the output of the network to fulfill the Burger's equation.

```
numInternalCollocationPoints = 10000;

pointSet = sobolset(2);
points = net(pointSet,numInternalCollocationPoints);

dataX = 2*points(:,1)-1;
dataT = points(:,2);
```

Create an array datastore containing the training data.

```
ds = arrayDatastore([dataX dataT]);
```

Define Deep Learning Model

Define a multilayer perceptron architecture with 9 fully connect operations with 20 hidden neurons. The first fully connect operation has two input channels corresponding to the inputs x and t. The last fully connect operation has one output $u(x,t)$.

Define and Initialize Model Parameters

Define the parameters for each of the operations and include them in a struct. Use the format `parameters.OperationName.ParameterName` where `parameters` is the struct, `OperationName` is the name of the operation (for example "fc1") and `ParameterName` is the name of the parameter (for example, "Weights").

Specify the number of layers and the number of neurons for each layer.

```
numLayers = 9;
numNeurons = 20;
```

Initialize the parameters for the first fully connect operation. The first fully connect operation has two input channels.

```
parameters = struct;

sz = [numNeurons 2];
parameters.fc1.Weights = initializeHe(sz,2);
parameters.fc1.Bias = initializeZeros([numNeurons 1]);
```

Initialize the parameters for each of the remaining intermediate fully connect operations.

```
for layerNumber=2:numLayers-1
    name = "fc"+layerNumber;

    sz = [numNeurons numNeurons];
    numIn = numNeurons;
    parameters.(name).Weights = initializeHe(sz,numIn);
    parameters.(name).Bias = initializeZeros([numNeurons 1]);
end
```

Initialize the parameters for the final fully connect operation. The final fully connect operation has one output channel.

```
sz = [1 numNeurons];
numIn = numNeurons;
```

```
parameters.("fc" + numLayers).Weights = initializeHe(sz,numIn);
parameters.("fc" + numLayers).Bias = initializeZeros([1 1]);
```

View the network parameters.

```
parameters
```

```
parameters = struct with fields:
    fc1: [1×1 struct]
    fc2: [1×1 struct]
    fc3: [1×1 struct]
    fc4: [1×1 struct]
    fc5: [1×1 struct]
    fc6: [1×1 struct]
    fc7: [1×1 struct]
    fc8: [1×1 struct]
    fc9: [1×1 struct]
```

View the parameters of the first fully connected layer.

```
parameters.fc1
```

```
ans = struct with fields:
    Weights: [20×2 dlarray]
       Bias: [20×1 dlarray]
```

Define Model and Model Gradients Functions

Create the function `model`, listed in the Model Function on page 18-0 section at the end of the example, that computes the outputs of the deep learning model. The function `model` takes as input the model parameters and the network inputs, and returns the model output.

Create the function `modelGradients`, listed in the Model Gradients Function on page 18-0 section at the end of the example, that takes as input the model parameters, the network inputs, and the initial and boundary conditions, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Specify Training Options

Train the model for 3000 epochs with a mini-batch size of 1000.

```
numEpochs = 3000;
miniBatchSize = 1000;
```

To train on a GPU if one is available, specify the execution environment `"auto"`. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox) (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Specify ADAM optimization options.

```
initialLearnRate = 0.01;
decayRate = 0.005;
```

Train Network

Train the network using a custom training loop.

Create a `minibatchqueue` object that processes and manages mini-batches of data during training. For each mini-batch:

- Format the data with the dimension labels `'BC'` (batch, channel). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.
- Train on a GPU according to the value of the `executionEnvironment` variable. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available.

```
mbq = minibatchqueue(ds, ...
    'MiniBatchSize',miniBatchSize, ...
    'MiniBatchFormat','BC', ...
    'OutputEnvironment',executionEnvironment);
```

Convert the initial and boundary conditions to `dlarray`. For the input data points, specify format with dimensions `'CB'` (channel, batch).

```
dlX0 = dlarray(X0,'CB');
dlT0 = dlarray(T0,'CB');
dlU0 = dlarray(U0);
```

If training using a GPU, convert the initial and conditions to `gpuArray`.

```
if (executionEnvironment == "auto" && canUseGPU) || (executionEnvironment == "gpu")
    dlX0 = gpuArray(dlX0);
    dlT0 = gpuArray(dlT0);
    dlU0 = gpuArray(dlU0);
end
```

Initialize the parameters for the Adam solver.

```
averageGrad = [];
averageSqGrad = [];
```

Accelerate the model gradients function using the `dlaccelerate` function. To learn more, see “Accelerate Custom Training Loop Functions” on page 18-311.

```
accfun = dlaccelerate(@modelGradients);
```

Initialize the training progress plot.

```
figure
C = colororder;
lineLoss = animatedline('Color',C(2,:));
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

Train the network.

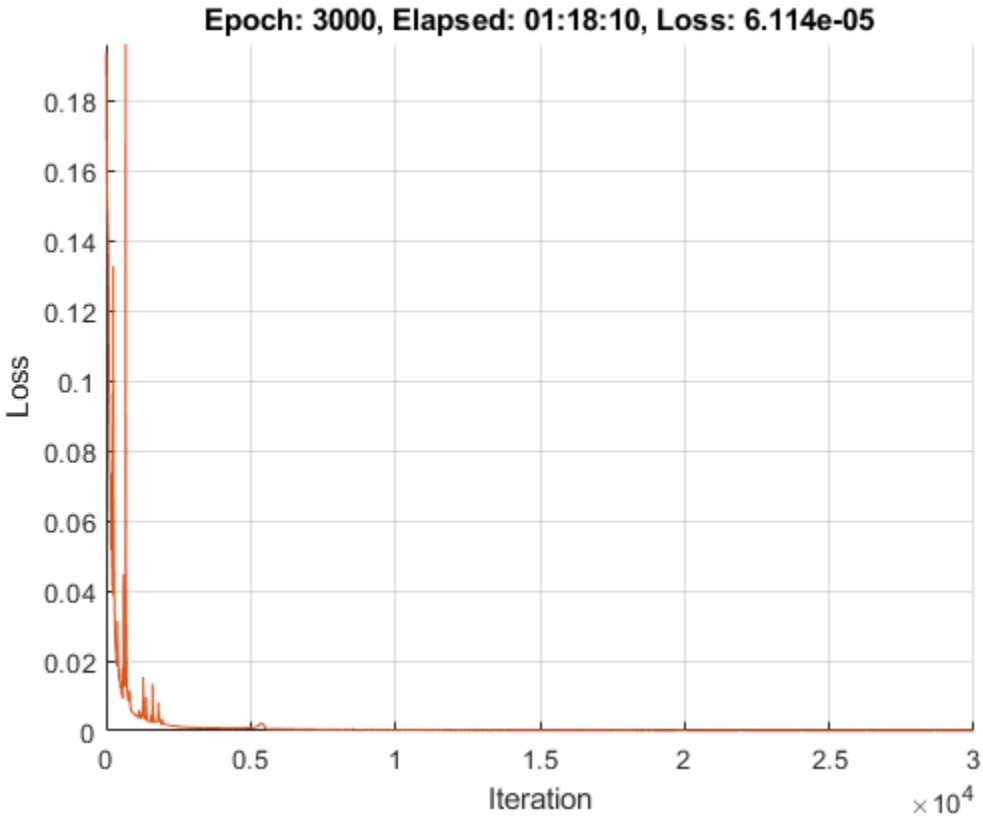
For each iteration:

- Read a mini-batch of data from the mini-batch queue

- Evaluate the model gradients and loss using the accelerated model gradients and `dlfeval` functions.
- Update the learning rate.
- Update the learnable parameters using the `adamupdate` function.

At the end of each epoch, update the training plot with the loss values.

```
start = tic;

iteration = 0;

for epoch = 1:numEpochs
    reset(mbq);

    while hasdata(mbq)
        iteration = iteration + 1;

        dlXT = next(mbq);
        dlX = dlXT(1,:);
        dlT = dlXT(2,:);

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss] = dlfeval(accfun,parameters,dlX,dlT,dlX0,dlT0,dlU0);

        % Update learning rate.
        learningRate = initialLearnRate / (1+decayRate*iteration);

        % Update the network parameters using the adamupdate function.
        [parameters,averageGrad,averageSqGrad] = adamupdate(parameters,gradients,averageGrad, ..
            averageSqGrad,iteration,learningRate);
    end

    % Plot training progress.
    loss = double(gather(extractdata(loss)));
    addpoints(lineLoss,iteration, loss);

    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    title("Epoch: " + epoch + ", Elapsed: " + string(D) + ", Loss: " + loss)
    drawnow
end
```



Check the effectiveness of the accelerated function by checking the hit and occupancy rate.

```
accfun
```

```
accfun =
  AcceleratedFunction with properties:

          Function: @modelGradients
           Enabled: 1
         CacheSize: 50
           HitRate: 99.9984
         Occupancy: 2
         CheckMode: 'none'
    CheckTolerance: 1.0000e-04
```

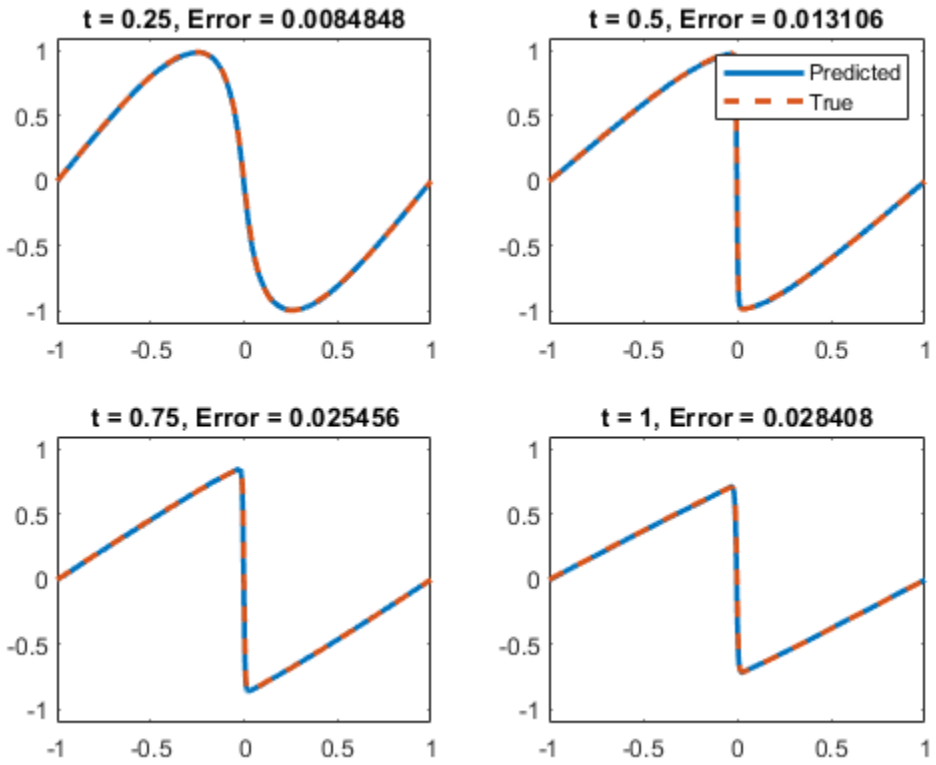
Evaluate Model Accuracy

For values of t at 0.25, 0.5, 0.75, and 1, compare the predicted values of the deep learning model with the true solutions of the Burger's equation using the l^2 error.

Set the target times to test the model at. For each time, calculate the solution at 1001 equally spaced points in the range [-1,1].

```
tTest = [0.25 0.5 0.75 1];
numPredictions = 1001;
XTest = linspace(-1,1,numPredictions);
```

```
figure

for i=1:numel(tTest)
    t = tTest(i);
    TTest = t*ones(1,numPredictions);

    % Make predictions.
    dlXTest = dlarray(XTest,'CB');
    dlTTest = dlarray(TTest,'CB');
    dlUPred = model(parameters,dlXTest,dlTTest);

    % Calcualte true values.
    UTest = solveBurgers(XTest,t,0.01/pi);

    % Calculate error.
    err = norm(extractdata(dlUPred) - UTest) / norm(UTest);

    % Plot predictions.
    subplot(2,2,i)
    plot(XTest,extractdata(dlUPred),'-','LineWidth',2);
    ylim([-1.1, 1.1])

    % Plot true values.
    hold on
    plot(XTest, UTest, '--','LineWidth',2)
    hold off

    title("t = " + t + ", Error = " + gather(err));
end

subplot(2,2,2)
legend('Predicted','True')
```



The plots show how close the predictions are to the true values.

Solve Burger's Equation Function

The `solveBurgers` function returns the true solution of Burger's equation at times `t` as outlined in [2].

```
function U = solveBurgers(X,t,nu)

% Define functions.
f = @(y) exp(-cos(pi*y)/(2*pi*nu));
g = @(y) exp(-(y.^2)/(4*nu*t));

% Initialize solutions.
U = zeros(size(X));

% Loop over x values.
for i = 1:numel(X)
    x = X(i);

    % Calculate the solutions using the integral function. The boundary
    % conditions in x = -1 and x = 1 are known, so leave 0 as they are
    % given by initialization of U.
    if abs(x) ~= 1
        fun = @(eta) sin(pi*(x-eta)) .* f(x-eta) .* g(eta);
        uxt = -integral(fun,-inf,inf);
        fun = @(eta) f(x-eta) .* g(eta);
```

```
        U(i) = uxt / integral(fun,-inf,inf);
    end
end

end
```

Model Gradients Function

The model is trained by enforcing that given an input (x,t) the output of the network $u(x,t)$ fulfills the Burger's equation, the boundary conditions, and the intial condition. In particular, two quantities contribute to the loss to be minimized:

loss = $\text{MSE}_f + \text{MSE}_u$,

where $\text{MSE}_f = \frac{1}{N_f}\sum_{i=1}^{N_f}\left|f(x_f^i, t_f^i)\right|^2$ and $\text{MSE}_u = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u(x_u^i, t_u^i) - u^i\right|^2$.

Here, $\{x_u^i, t_u^i\}_{i=1}^{N_u}$ correspond to collocation points on the boundary of the computational domain and account for both boundary and initial condition. $\{x_f^i, t_f^i\}_{i=1}^{N_f}$ are points in the interior of the domain.

Calculating MSE_f requires the derivatives $\frac{\partial u}{\partial t}, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}$ of the output u of the model.

The function `modelGradients` takes as input, the model parameters `parameters`, the network inputs `dlX` and `dlT`, the initial and boundary conditions `dlX0`, `dlT0`, and `dlU0`, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

```
function [gradients,loss] = modelGradients(parameters,dlX,dlT,dlX0,dlT0,dlU0)

% Make predictions with the initial conditions.
U = model(parameters,dlX,dlT);

% Calculate derivatives with respect to X and T.
gradientsU = dlgradient(sum(U,'all'),{dlX,dlT},'EnableHigherDerivatives',true);
Ux = gradientsU{1};
Ut = gradientsU{2};

% Calculate second-order derivatives with respect to X.
Uxx = dlgradient(sum(Ux,'all'),dlX,'EnableHigherDerivatives',true);

% Calculate lossF. Enforce Burger's equation.
f = Ut + U.*Ux - (0.01./pi).*Uxx;
zeroTarget = zeros(size(f), 'like', f);
lossF = mse(f, zeroTarget);

% Calculate lossU. Enforce initial and boundary conditions.
dlU0Pred = model(parameters,dlX0,dlT0);
lossU = mse(dlU0Pred, dlU0);

% Combine losses.
loss = lossF + lossU;

% Calculate gradients with respect to the learnable parameters.
gradients = dlgradient(loss,parameters);
```

```
end
```

Model Function

The model trained in this example consists of a series of fully connect operations with a tanh operation between each one.

The model function takes as input the model parameters `parameters` and the network inputs `dlX` and `dlT`, and returns the model output `dlU`.

```
function dlU = model(parameters,dlX,dlT)

dlXT = [dlX;dlT];
numLayers = numel(fieldnames(parameters));

% First fully connect operation.
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlU = fullyconnect(dlXT,weights,bias);

% tanh and fully connect operations for remaining layers.
for i=2:numLayers
    name = "fc" + i;

    dlU = tanh(dlU);

    weights = parameters.(name).Weights;
    bias = parameters.(name).Bias;
    dlU = fullyconnect(dlU, weights, bias);
end

end
```

References

1. Maziar Raissi, Paris Perdikaris, and George Em Karniadakis, Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations https://arxiv.org/abs/1711.10561
2. C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, A. Patera, Spectral and finite difference solutions of the Burgers equation, Computers & fluids 14 (1986) 23–41.

See Also

`dlarray` | `dlfeval` | `dlgradient` | `minibatchqueue`

More About

- “Solve Partial Differential Equation with LBFGS Method and Deep Learning” on page 18-351
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216

Solve Partial Differential Equation with LBFGS Method and Deep Learning

This example shows how to train a Physics Informed Neural Network (PINN) to numerically compute the solution of the Burger's equation by using the limited-memory BFGS (LBFGS) algorithm.

The Burger's equation is a partial differential equation (PDE) that arises in different areas of applied mathematics. In particular, fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flows.

Given the computational domain$[-1,1] \times [0,1]$, this examples uses a physics informed neural network (PINN) [1] and trains a multilayer perceptron neural network that takes samples (x,t) as input, where $x \in [-1,1]$ is the spatial variable, and $t \in [0,1]$ is the time variable, and returns $u(x,t)$, where u is the solution of the Burger's equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{0.01}{\pi}\frac{\partial^2 u}{\partial x^2} = 0,$$

with $u(x, t=0) = -sin(\pi x)$as the initial condition, and $u(x=-1,t) = 0$ and $u(x=1,t) = 0$ as the boundary conditions.

Instead of training the network using the `trainNetwork` function, or using a custom training loop that updates parametes using `sgdmupdate` or similar functions, this example estimates the learnable parameters by using the `fmincon` function (requires Optimization Toolbox™). The `fmincon` function finds the minimum of constrained nonlinear multivariable functions.

The example trains the model by enforcing that given an input (x,t), the output of the network $u(x,t)$ fulfills the Burger's equation, the boundary conditions, and the initial condition. To train the model, the exmaple uses the limited-memory BFGS (LBFGS) algorithm which is a quasi-Newton method that approximates the Broyden-Fletcher-Goldfarb-Shanno algorithm.

Training this model does not require collecting data in advance. You can generate data using the definition of the PDE and the constraints.

Generate Training Data

Training the model requires a data set of collocation points that enforce the boundary conditions, enforce the initial conditions, and fulfill the Burger's equation.

Select 25 equally spaced time points to enforce each of the boundary conditions $u(x=-1,t) = 0$ and $u(x=1,t) = 0$.

```
numBoundaryConditionPoints = [25 25];

x0BC1 = -1*ones(1,numBoundaryConditionPoints(1));
x0BC2 = ones(1,numBoundaryConditionPoints(2));

t0BC1 = linspace(0,1,numBoundaryConditionPoints(1));
t0BC2 = linspace(0,1,numBoundaryConditionPoints(2));

u0BC1 = zeros(1,numBoundaryConditionPoints(1));
u0BC2 = zeros(1,numBoundaryConditionPoints(2));
```

Select 50 equally spaced spatial points to enforce the initial condition $u(x, t=0) = -sin(\pi x)$.

```
numInitialConditionPoints  = 50;

x0IC = linspace(-1,1,numInitialConditionPoints);
t0IC = zeros(1,numInitialConditionPoints);
u0IC = -sin(pi*x0IC);
```

Group together the data for initial and boundary conditions.

```
X0 = [x0IC x0BC1 x0BC2];
T0 = [t0IC t0BC1 t0BC2];
U0 = [u0IC u0BC1 u0BC2];
```

Select 10,000 points to enforce the output of the network to fulfill the Burger's equation.

```
numInternalCollocationPoints = 10000;

pointSet = sobolset(2);
points = net(pointSet,numInternalCollocationPoints);

dataX = 2*points(:,1)-1;
dataT = points(:,2);
```

Create an array datastore containing the training data.

```
ds = arrayDatastore([dataX dataT]);
```

Define Deep Learning Model

Define a multilayer perceptron architecture with 9 fully connect operations with 20 hidden neurons. The first fully connect operation has two input channels corresponding to the inputs x and t. The last fully connect operation has one output $u(x,t)$.

Define and Initialize Model Parameters

Define the parameters for each of the operations and include them in a structure. Use the format `parameters.OperationName_ParameterName` where `parameters` is the structure, `OperationName` is the name of the operation (for example "fc1") and `ParameterName` is the name of the parameter (for example, "Weights").

The algorithm in this example requires learnable parameters to be in the first level of the stucture, so do not use nested structures in this step. The `fmincon` function requires the learnable to be doubles.

Specify the number of layers and the number of neurons for each layer.

```
numLayers = 9;
numNeurons = 20;
```

Initialize the parameters for the first fully connect operation. The first fully connect operation has two input channels.

```
parameters = struct;

sz = [numNeurons 2];
parameters.fc1_Weights = initializeHe(sz,2,'double');
parameters.fc1_Bias = initializeZeros([numNeurons 1],'double');
```

Initialize the parameters for each of the remaining intermediate fully connect operations.

```
for layerNumber=2:numLayers-1
    name = "fc"+layerNumber;

    sz = [numNeurons numNeurons];
    numIn = numNeurons;
    parameters.(name + "_Weights") = initializeHe(sz,numIn,'double');
    parameters.(name + "_Bias") = initializeZeros([numNeurons 1],'double');
end
```

Initialize the parameters for the final fully connect operation. The final fully connect operation has one output channel.

```
sz = [1 numNeurons];
numIn = numNeurons;
parameters.("fc" + numLayers + "_Weights") = initializeHe(sz,numIn,'double');
parameters.("fc" + numLayers + "_Bias") = initializeZeros([1 1],'double');
```

View the network parameters.

```
parameters
```

```
parameters = struct with fields:
    fc1_Weights: [20×2 dlarray]
       fc1_Bias: [20×1 dlarray]
    fc2_Weights: [20×20 dlarray]
       fc2_Bias: [20×1 dlarray]
    fc3_Weights: [20×20 dlarray]
       fc3_Bias: [20×1 dlarray]
    fc4_Weights: [20×20 dlarray]
       fc4_Bias: [20×1 dlarray]
    fc5_Weights: [20×20 dlarray]
       fc5_Bias: [20×1 dlarray]
    fc6_Weights: [20×20 dlarray]
       fc6_Bias: [20×1 dlarray]
    fc7_Weights: [20×20 dlarray]
       fc7_Bias: [20×1 dlarray]
    fc8_Weights: [20×20 dlarray]
       fc8_Bias: [20×1 dlarray]
    fc9_Weights: [1×20 dlarray]
       fc9_Bias: [1×1 dlarray]
```

Define Model and Model Gradients Functions

Create the function `model`, listed in the Model Function on page 18-0 section at the end of the example, that computes the outputs of the deep learning model. The function `model` takes as input the model parameters and the network inputs, and returns the model output.

Create the function `modelGradients`, listed in the Model Gradients Function on page 18-0 section at the end of the example, that takes as input the model parameters, the network inputs, and the initial and boundary conditions, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Define `fmincon` Objective Function

Create the function `objectiveFunction`, listed in the `fmincon` Objective Function section of the example that returns the loss and gradients of the model. The function `objectiveFunction` takes as input, a vector of learnable parameters, the network inputs, the initial conditions, and the names

and sizes of the learnable parameters, and returns the loss to be minimized by the `fmincon` function and the gradients of the loss with respect to the learnable parameters.

Specify Optimization Options

Specify the optimization options:

- Optimize using the `fmincon` optmizer with the LBFGS algorithm for no more than 7500 iterations and function evaluations.
- Evaluate with optimality tolerance `1e-5`.
- Provide the gradients to the algorithm.

```
options = optimoptions('fmincon', ...
    'HessianApproximation','lbfgs', ...
    'MaxIterations',7500, ...
    'MaxFunctionEvaluations',7500, ...
    'OptimalityTolerance',1e-5, ...
    'SpecifyObjectiveGradient',true);
```

Train Network Using `fmincon`

Train the network using the `fmincon` function.

The `fmincon` function requires the learnable parameters to be specified as a vector. Convert the parameters to a vector using the `paramsStructToVector` function (attached to this example as a supporting file). To convert back to a structure of parameters, also return the parameter names and sizes.

```
[parametersV,parameterNames,parameterSizes] = parameterStructToVector(parameters);
parametersV = extractdata(parametersV);
```

Convert the training data to `dlarray` objects with format `'CB'` (channel, batch).

```
dlX = dlarray(dataX','CB');
dlT = dlarray(dataT','CB');
dlX0 = dlarray(X0,'CB');
dlT0 = dlarray(T0,'CB');
dlU0 = dlarray(U0,'CB');
```

Create a function handle with one input that defines the objective function.

```
objFun = @(parameters) objectiveFunction(parameters,dlX,dlT,dlX0,dlT0,dlU0,parameterNames,parame
```

Update the learnable parameters using the `fmincon` function. Depending on the number of iterations, this can take a while to run. To enable a detailed verbose output, set the `'Display'` optimization option to `'iter-detailed'`.

```
parametersV = fmincon(objFun,parametersV,[],[],[],[],[],[],[],options);
```

```
Solver stopped prematurely.

fmincon stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 7.500000e+03.
```

For prediction, convert the vector of parameters to a structure using the `parameterVectorToStruct` function (attached to this example as a supporting file).

```
parameters = parameterVectorToStruct(parametersV,parameterNames,parameterSizes);
```

Evaluate Model Accuracy

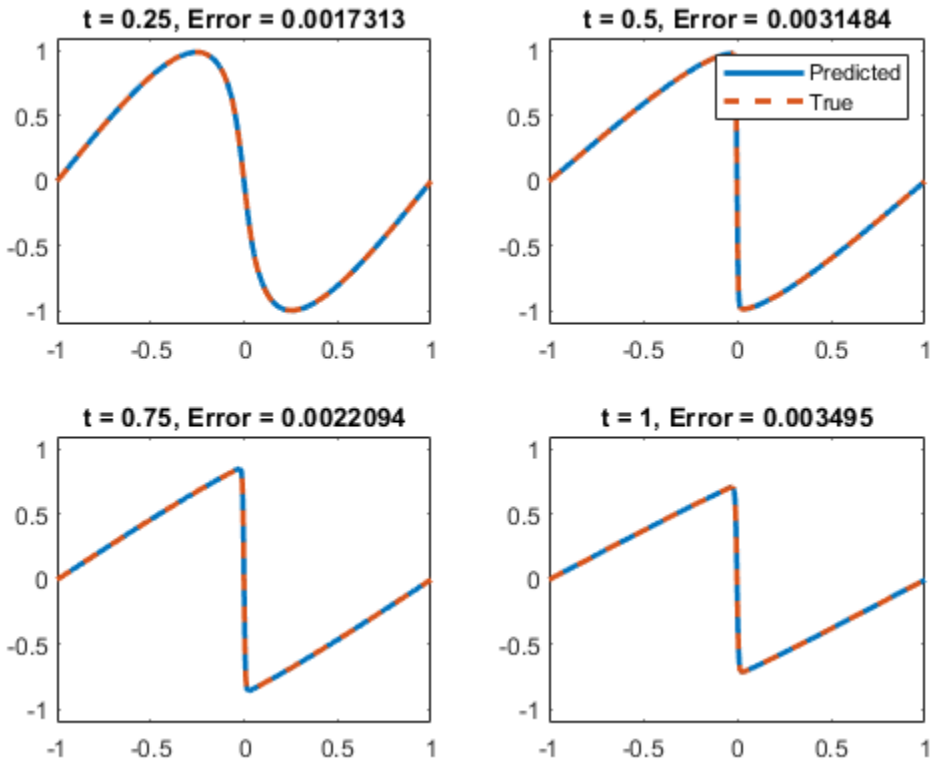
For values of t at 0.25, 0.5, 0.75, and 1, compare the predicted values of the deep learning model with the true solutions of the Burger's equation using the relative l^2 error.

Set the target times to test the model at. For each time, calculate the solution at 1001 equally spaced points in the range [-1,1].

```
tTest = [0.25 0.5 0.75 1];
numPredictions = 1001;
XTest = linspace(-1,1,numPredictions);
```

Test the model.

```
figure
for i=1:numel(tTest)
    t = tTest(i);
    TTest = t*ones(1,numPredictions);

    % Make predictions.
    dlXTest = dlarray(XTest,'CB');
    dlTTest = dlarray(TTest,'CB');
    dlUPred = model(parameters,dlXTest,dlTTest);

    % Calcualte true values.
    UTest = solveBurgers(XTest,t,0.01/pi);

    % Calculate error.
    err = norm(extractdata(dlUPred) - UTest) / norm(UTest);

    % Plot predictions.
    subplot(2,2,i)
    plot(XTest,extractdata(dlUPred),'-','LineWidth',2);
    ylim([-1.1, 1.1])

    % Plot true values.
    hold on
    plot(XTest, UTest, '--','LineWidth',2)
    hold off

    title("t = " + t + ", Error = " + gather(err));
end

subplot(2,2,2)
legend('Predicted','True')
```



The plots show how close the predictions are to the true values.

Solve Burger's Equation Function

The `solveBurgers` function returns the true solution of Burger's equation at times `t` as outlined in [2].

```
function U = solveBurgers(X,t,nu)

% Define functions.
f = @(y) exp(-cos(pi*y)/(2*pi*nu));
g = @(y) exp(-(y.^2)/(4*nu*t));

% Initialize solutions.
U = zeros(size(X));

% Loop over x values.
for i = 1:numel(X)
    x = X(i);

    % Calculate the solutions using the integral function. The boundary
    % conditions in x = -1 and x = 1 are known, so leave 0 as they are
    % given by initialization of U.
    if abs(x) ~= 1
        fun = @(eta) sin(pi*(x-eta)) .* f(x-eta) .* g(eta);
        uxt = -integral(fun,-inf,inf);
        fun = @(eta) f(x-eta) .* g(eta);
```

```
        U(i) = uxt / integral(fun,-inf,inf);
    end
end

end
```

`fmincon` Objective Function

The `objectiveFunction` function defines the objective function to be used by the LBFGS algorithm.

This `objectiveFunction` funtion takes as input, a vector of learnable parameters `parametersV`, the network inputs, `dlX` and `dlT`, the initial and boundary conditions `dlX0, dlT0`, and `dlU0`, and the names and sizes of the learnable parameters `parameterNames` and `parameterSizes`, respectively, and returns the loss to be minimized by the `fmincon` function `loss` and a vector containing the gradients of the loss with respect to the learnable parameters `gradientsV`.

```
function [loss,gradientsV] = objectiveFunction(parametersV,dlX,dlT,dlX0,dlT0,dlU0,parameterNames,

% Convert parameters to structure of dlarray objects.
parametersV = dlarray(parametersV);
parameters = parameterVectorToStruct(parametersV,parameterNames,parameterSizes);

% Evaluate model gradients and loss.
[gradients,loss] = dlfeval(@modelGradients,parameters,dlX,dlT,dlX0,dlT0,dlU0);

% Return loss and gradients for fmincon.
gradientsV = parameterStructToVector(gradients);
gradientsV = extractdata(gradientsV);
loss = extractdata(loss);

end
```

Model Gradients Function

The model is trained by enforcing that given an input (x, t) the output of the network $u(x, t)$ fulfills the Burger's equation, the boundary conditions, and the intial condition. In particular, two quantities contribute to the loss to be minimized:

loss = $\text{MSE}_f + \text{MSE}_u$,

where $\text{MSE}_f = \frac{1}{N_f}\sum_{i=1}^{N_f}\left|f(x_f^i, t_f^i)\right|^2$ and $\text{MSE}_u = \frac{1}{N_u}\sum_{i=1}^{N_u}\left|u(x_u^i, t_u^i) - u^i\right|^2$.

Here, $\{x_u^i, t_u^i\}_{i=1}^{N_u}$ correspond to collocation points on the boundary of the computational domain and account for both boundary and initial condition. $\{x_f^i, t_f^i\}_{i=1}^{N_f}$ are points in the interior of the domain.

Calculating MSE_f requires the derivatives $\frac{\partial u}{\partial t}, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}$ of the output u of the model.

The function `modelGradients` takes as input, the model parameters `parameters`, the network inputs `dlX` and `dlT`, the initial and boundary conditions `dlX0`, `dlT0`, and `dlU0`, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

```
function [gradients,loss] = modelGradients(parameters,dlX,dlT,dlX0,dlT0,dlU0)
```

```
% Make predictions with the initial conditions.
U = model(parameters,dlX,dlT);

% Calculate derivatives with respect to X and T.
gradientsU = dlgradient(sum(U,'all'),{dlX,dlT},'EnableHigherDerivatives',true);
Ux = gradientsU{1};
Ut = gradientsU{2};

% Calculate second-order derivatives with respect to X.
Uxx = dlgradient(sum(Ux,'all'),dlX,'EnableHigherDerivatives',true);

% Calculate lossF. Enforce Burger's equation.
f = Ut + U.*Ux - (0.01./pi).*Uxx;
zeroTarget = zeros(size(f), 'like', f);
lossF = mse(f, zeroTarget);

% Calculate lossU. Enforce initial and boundary conditions.
dlU0Pred = model(parameters,dlX0,dlT0);
lossU = mse(dlU0Pred, dlU0);

% Combine losses.
loss = lossF + lossU;

% Calculate gradients with respect to the learnable parameters.
gradients = dlgradient(loss,parameters);

end
```

Model Function

The model trained in this example consists of a series of fully connect operations with a tanh operation between each one.

The model function takes as input the model parameters `parameters` and the network inputs `dlX` and `dlT`, and returns the model output `dlU`.

```
function dlU = model(parameters,dlX,dlT)

dlXT = [dlX;dlT];
numLayers = numel(fieldnames(parameters))/2;

% First fully connect operation.
weights = parameters.fc1_Weights;
bias = parameters.fc1_Bias;
dlU = fullyconnect(dlXT,weights,bias);

% tanh and fully connect operations for remaining layers.
for i=2:numLayers
    name = "fc" + i;

    dlU = tanh(dlU);

    weights = parameters.(name + "_Weights");
    bias = parameters.(name + "_Bias");
    dlU = fullyconnect(dlU, weights, bias);
end

end
```

References

1. Maziar Raissi, Paris Perdikaris, and George Em Karniadakis, Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations https://arxiv.org/abs/1711.10561
2. C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, A. Patera, Spectral and finite difference solutions of the Burgers equation, Computers & fluids 14 (1986) 23–41.

See Also

`dlarray` | `dlfeval` | `dlgradient`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216

Solve Ordinary Differential Equation Using Neural Network

This example shows how to solve an ordinary differential equation (ODE) using a neural network.

Not all differential equations have a closed-form solution. To find approximate solutions to these types of equations, many traditional numerical algorithms are available. However, you can also solve an ODE by using a neural network. This approach comes with several advantages, including that it provides differentiable approximate solutions in a closed analytic form [1].

This example shows you how to:

1. Generate training data in the range $x \in [0, 2]$.
2. Define a neural network that takes x as input and returns the approximate solution to the ODE $\dot{y} = -2xy$, evaluated at x, as output.
3. Train the network with a custom loss function.
4. Compare the network predictions with the analytic solution.

ODE and Loss Function

In this example, you solve the ODE

$$\dot{y} = -2xy,$$

with the initial condition $y(0) = 1$. This ODE has the analytic solution

$$y(x) = e^{-x^2}.$$

Define a custom loss function that penalizes deviations from satisfying the ODE and the initial condition. In this example, the loss function is a weighted sum of the ODE loss and the initial condition loss:

$$L_\theta(x) = \|\dot{y}_\theta + 2xy_\theta\|^2 + k\|y_\theta(0) - 1\|^2$$

θ is the network parameters, k is a constant coefficient, y_θ is the solution predicted by the network, and $\dot{y_\theta}$ is the derivative of the predicted solution computed using automatic differentiation. The term $\| \dot{y_\theta} + 2xy_\theta \|^2$ is the ODE loss and it quantifies how much the predicted solution deviates from satisfying the ODE definition. The term $\|y_\theta(0) - 1\|^2$ is the initial condition loss and it quantifies how much the predicted solution deviates from satisfying the initial condition.

Generate Input Data and Define Network

Generate 10,000 training data points in the range $x \in [0, 2]$.

```
x = linspace(0,2,10000)';
```

Define the network for solving the ODE. As the input is a real number $x \in \mathbb{R}$, specify an input size of 1.

```
inputSize = 1;
layers = [
    featureInputLayer(inputSize,Normalization="none")
    fullyConnectedLayer(10)
```

```
    sigmoidLayer
    fullyConnectedLayer(1)
    sigmoidLayer];
```

Create a `dlnetwork` object from the layer array.

```
dlnet = dlnetwork(layers)
```

```
dlnet =
  dlnetwork with properties:

         Layers: [5×1 nnet.cnn.layer.Layer]
    Connections: [4×2 table]
     Learnables: [4×3 table]
          State: [0×3 table]
     InputNames: {'input'}
    OutputNames: {'layer_2'}
    Initialized: 1
```

Define Model Gradients Function

Create the function `modelGradients` on page 18-0 , listed at the end of the example, which takes as inputs a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX`, and the coefficient associated with the initial condition loss `icCoeff`. This function returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the corresponding loss.

Specify Training Options

Train for 15 epochs with a mini-batch size of 100.

```
numEpochs = 15;
miniBatchSize = 100;
```

Specify the options for SGDM optimization. Specify a learning rate of 0.5, a learning rate drop factor of 0.5, a learning rate drop period of 5, and a momentum of 0.9.

```
initialLearnRate = 0.5;
learnRateDropFactor = 0.5;
learnRateDropPeriod = 5;
momentum = 0.9;
```

Specify the coefficient of the initial condition term in the loss function as 7. This coefficient specifies the relative contribution of the initial condition to the loss. Tweaking this parameter can help training converge faster.

```
icCoeff = 7;
```

Train Model

To use mini-batches of data during training:

1. Create an `arrayDatastore` object from the training data.
2. Create a `minibatchqueue` object that takes the `arrayDatastore` object as input, specify a mini-batch size, and format the training data with the dimension labels `'BC'` (batch, channel).

```
ads = arrayDatastore(x,IterationDimension=1);
mbq = minibatchqueue(ads,MiniBatchSize=miniBatchSize,MiniBatchFormat="BC");
```

By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`.

Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Support by Release" (Parallel Computing Toolbox).

Initialize the training progress plot.

```
figure
set(gca,YScale="log")
lineLossTrain = animatedline(Color=[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss (log scale)")
grid on
```

Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

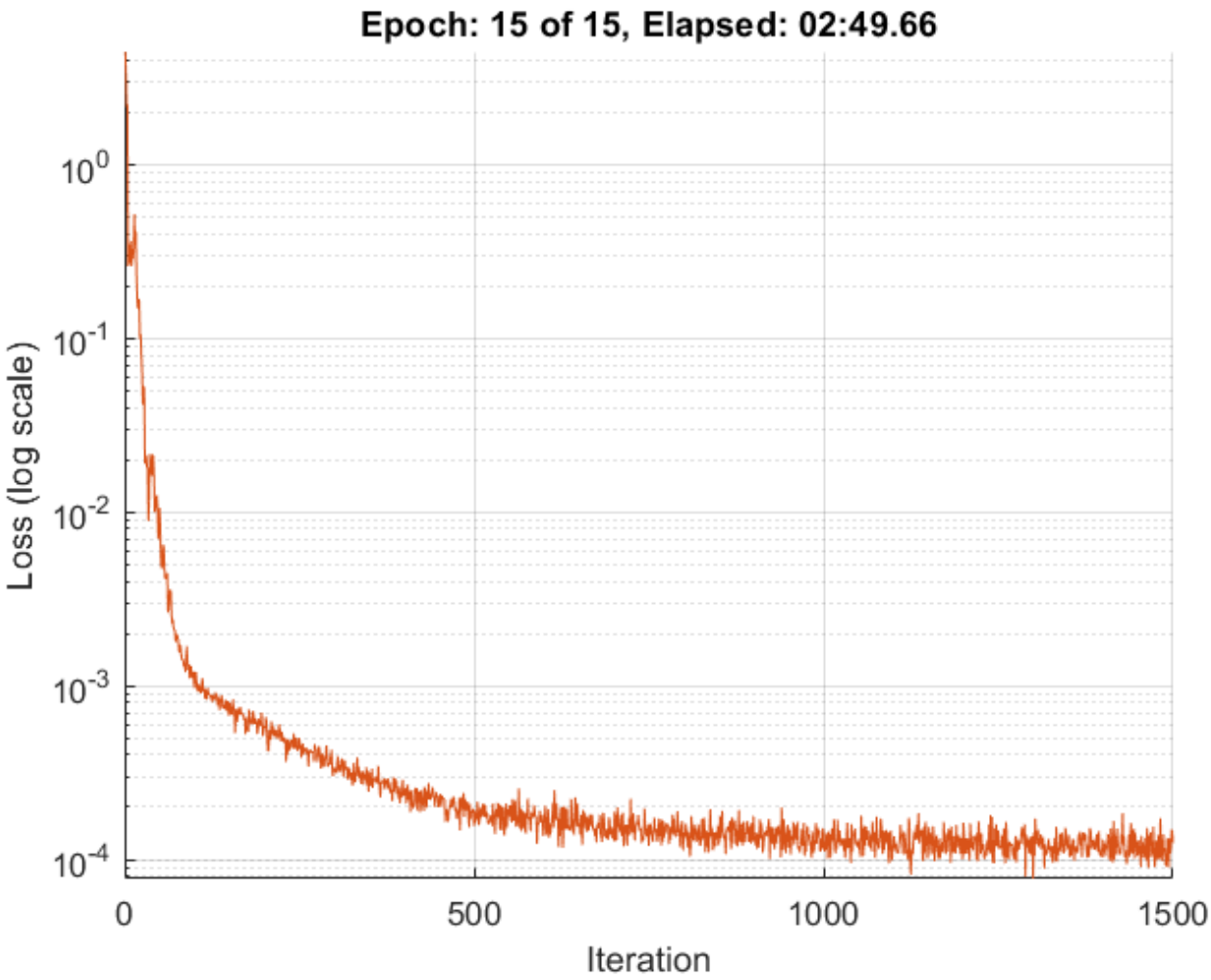
Train the network using a custom training loop. For each epoch, shuffle the data and loop over mini-batches of data. For each mini-batch:

- Evaluate the model gradients and loss using the `dlfeval` and `modelGradients` functions.
- Update the network parameters using the `sgdmupdate` function.
- Display the training progress.

Every `learnRateDropPeriod` epochs, multiply the learning rate by `learnRateDropFactor`.

```
iteration = 0;
learnRate = initialLearnRate;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    mbq.shuffle

    % Loop over mini-batches.
    while hasdata(mbq)

        iteration = iteration + 1;

        % Read mini-batch of data.
        dlX = next(mbq);

        % Evaluate the model gradients and loss using dlfeval and the modelGradients function.
        [gradients,loss] = dlfeval(@modelGradients, dlnet, dlX, icCoeff);

        % Update network parameters using the SGDM optimizer.
        [dlnet,velocity] = sgdmupdate(dlnet,gradients,velocity,learnRate,momentum);

        % To plot, convert the loss to double.
        loss = double(gather(extractdata(loss)));
```

```
        % Display the training progress.
        D = duration(0,0,toc(start),Format="mm:ss.SS");
        addpoints(lineLossTrain,iteration,loss)
        title("Epoch: " + epoch + " of " + numEpochs + ", Elapsed: " + string(D))
        drawnow

    end
    % Reduce the learning rate.
    if mod(epoch,learnRateDropPeriod)==0
        learnRate = learnRate*learnRateDropFactor;
    end
end
```



Test Model

Test the accuracy of the network by comparing its predictions with the analytic solution.

Generate test data in the range $x \in [0, 4]$ to see if the network is able to extrapolate outside the training range $x \in [0, 2]$.

```
xTest = linspace(0,4,1000)';
```

To use mini-batches of data during testing:

1. Create an `arrayDatastore` object from the testing data.
2. Create a `minibatchqueue` object that takes the `arrayDatastore` object as input, specify a mini-batch size of 100, and format the training data with the dimension labels `'BC'` (batch, channel).

```
adsTest = arrayDatastore(xTest,IterationDimension=1);
mbqTest = minibatchqueue(adsTest,MiniBatchSize=100,MiniBatchFormat="BC");
```

Loop over the mini-batches and make predictions using the `modelPredictions` function, listed at the end of the example.

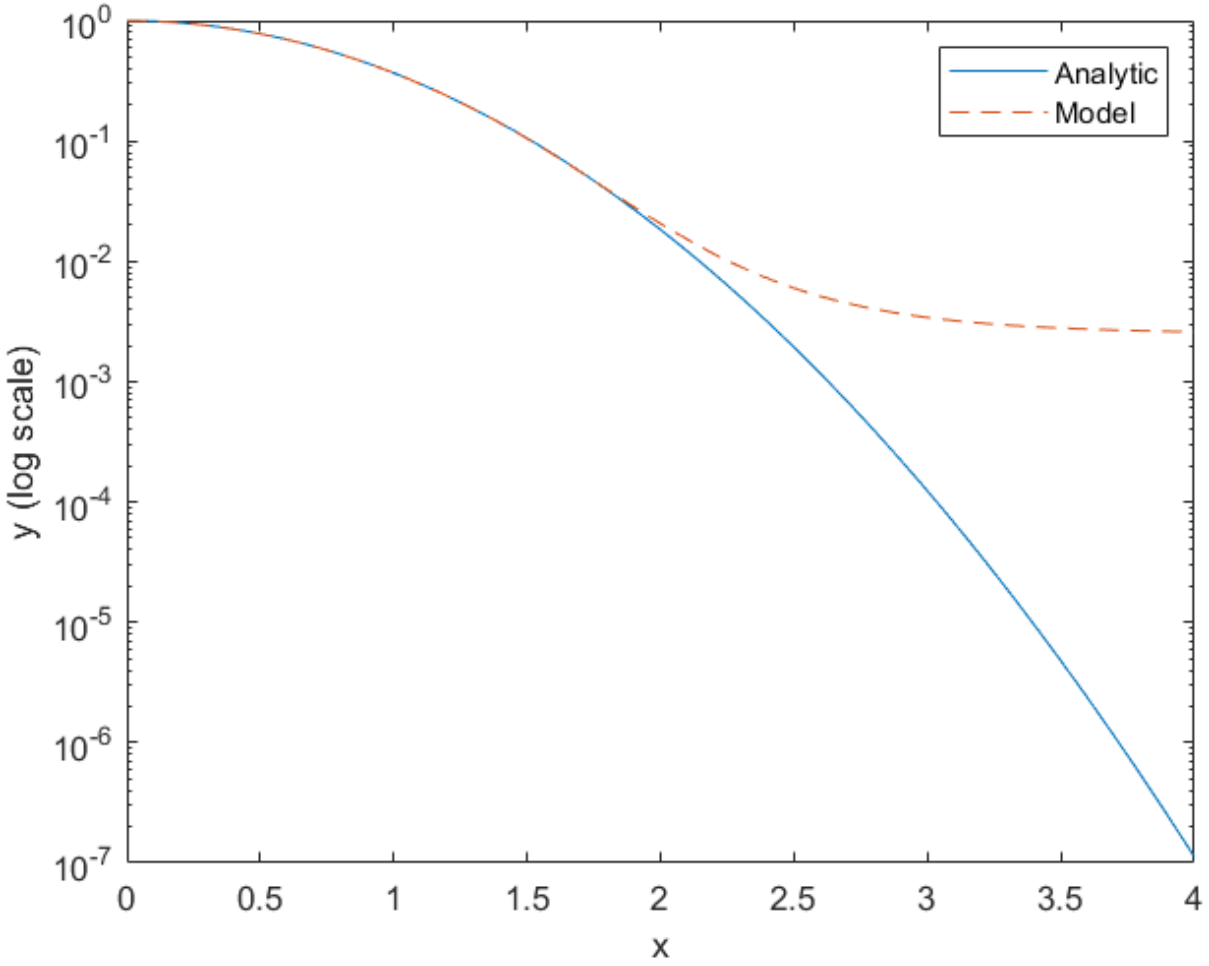
```
yModel = modelPredictions(dlnet,mbqTest);
```

Evaluate the analytic solution.

```
yAnalytic = exp(-xTest.^2);
```

Compare the analytic solution and the model prediction by plotting them on a logarithmic scale.

```
figure;
plot(xTest,yAnalytic,"-")
hold on
plot(xTest,yModel,"--")
legend("Analytic","Model")
xlabel("x")
ylabel("y (log scale)")
set(gca,YScale="log")
```



The model approximates the analytic solution accurately in the training range $x \in [0, 2]$ and it extrapolates in the range $x \in (2, 4]$ with lower accuracy.

Calculate the mean squared relative error in the training range $x \in [0, 2]$.

```
yModelTrain = yModel(1:500);
yAnalyticTrain = yAnalytic(1:500);

errorTrain = mean(((yModelTrain-yAnalyticTrain)./yAnalyticTrain).^2)
```

```
errorTrain = single
    4.3454e-04
```

Calculate the mean squared relative error in the extrapolated range $x \in (2, 4]$.

```
yModelExtra = yModel(501:1000);
yAnalyticExtra = yAnalytic(501:1000);

errorExtra = mean(((yModelExtra-yAnalyticExtra)./yAnalyticExtra).^2)
```

```
errorExtra = single
    17576612
```

Notice that the mean squared relative error is higher in the extrapolated range than it is in the training range.

Model Gradients Function

The `modelGradients` function takes as inputs a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX`, and the coefficient associated with the initial condition loss `icCoeff`. This function returns the gradients of the loss with respect to the learnable parameters in `dlnet` and the corresponding loss. The loss is defined as a weighted sum of the ODE loss and the initial condition loss. The evaluation of this loss requires second order derivatives. To enable second order automatic differentiation, use the function `dlgradient` and set the `EnableHigherDerivatives` name-value argument to `true`.

```
function [gradients,loss] = modelGradients(dlnet, dlX, icCoeff)
y = forward(dlnet,dlX);

% Evaluate the gradient of y with respect to x.
% Since another derivative will be taken, set EnableHigherDerivatives to true.
dy = dlgradient(sum(y,"all"),dlX,EnableHigherDerivatives=true);

% Define ODE loss.
eq = dy + 2*y.*dlX;

% Define initial condition loss.
ic = forward(dlnet,dlarray(0,"CB")) - 1;

% Specify the loss as a weighted sum of the ODE loss and the initial condition loss.
loss = mean(eq.^2,"all") + icCoeff * ic.^2;

% Evaluate model gradients.
gradients = dlgradient(loss, dlnet.Learnables);

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet` and a `minibatchqueue` of input data `mbq` and computes the model predictions `y` by iterating over all data in the `minibatchqueue` object.

```
function Y = modelPredictions(dlnet,mbq)

Y = [];

while hasdata(mbq)

    % Read mini-batch of data.
    dlXTest = next(mbq);

    % Predict output using trained network.
    dlY = predict(dlnet,dlXTest);
    YPred = gather(extractdata(dlY));
    Y = [Y; YPred'];

end

end
```

References

1 Lagaris, I. E., A. Likas, and D. I. Fotiadis. “Artificial Neural Networks for Solving Ordinary and Partial Differential Equations.” *IEEE Transactions on Neural Networks* 9, no. 5 (September 1998): 987–1000. https://doi.org/10.1109/72.712178.

See Also

`dlarray` | `dlfeval` | `dlgradient`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Dynamical System Modeling Using Neural ODE” on page 18-368
- “Solve Partial Differential Equations Using Deep Learning” on page 18-341

Dynamical System Modeling Using Neural ODE

This example shows how to train a neural network with neural ordinary differential equations (ODEs) to learn the dynamics of a physical system.

Neural ODEs [1] are deep learning operations defined by the solution of an ODE. More specifically, neural ODE is an operation that can be used in any architecture and, given an input, defines its output as the numerical solution of the ODE

$$y' = f(t, y, \theta)$$

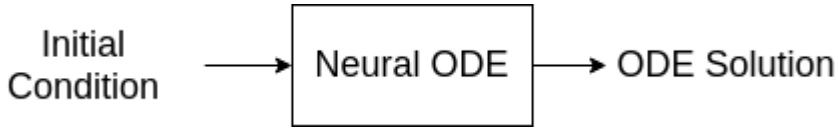
for the time horizon (t_0, t_1) and the initial condition $y(t_0) = y_0$. The right-hand side $f(t, y, \theta)$ of the ODE depends on a set of trainable parameters θ, which the model learns during the training process. In this example, $f(t, y, \theta)$ is modeled with a model function containing fully connected operations and nonlinear activations. The initial condition y_0 is either the input of the entire architecture, as in the case of this example, or is the output of a previous operation.

This example shows how to train a neural network with neural ODEs to learn the dynamics x of a given physical system, described by the following ODE:

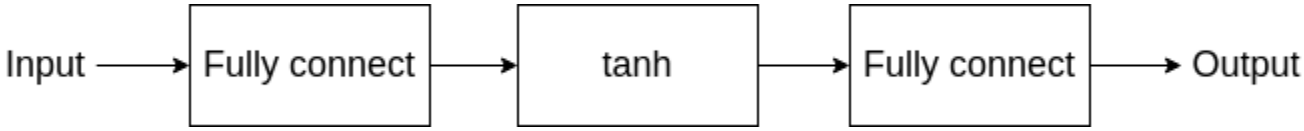
$x' = Ax$,

where A is a 2-by-2 matrix.

The neural network of this example takes as input an initial condition and computes the ODE solution through the learned neural ODE model.



The neural ODE operation, given an initial condition, outputs the solution of an ODE model. In this example, specify a block with a fully connected layer, a tanh layer, and another fully connected layer as the ODE model.



In this example, the ODE that defines the model is solved numerically with the explicit Runge-Kutta (4,5) pair of Dormand and Prince [2]. The backward pass uses automatic differentiation to learn the trainable parameters θ by backpropagating through each operation of the ODE solver.

The learned function $f(t, y, \theta)$ is used as the right-hand side for computing the solution of the same model for additional initial conditions.

Synthesize Data of Target Dynamics

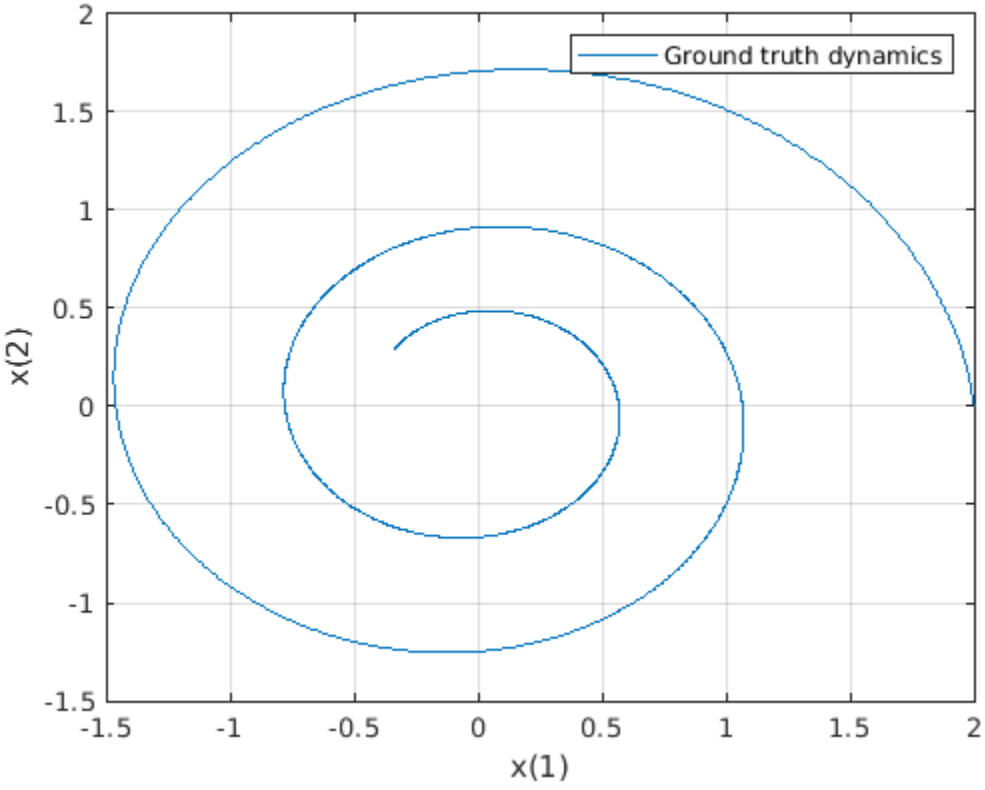
Define the target dynamics as a linear ODE model $x' = Ax$, with `x0` as its initial condition, and compute its numerical solution `xTrain` with `ode45` in the time interval `[0 15]`. To compute an accurate ground truth data, set the relative tolerance of the `ode45` numerical solver to 10^{-7}. Later,

you use the value of `xTrain` as ground truth data for learning an approximated dynamics with a neural ODE model.

```
x0 = [2; 0];
A = [-0.1 -1; 1 -0.1];
trueModel = @(t,y) A*y;

numTimeSteps = 2000;
T = 15;
odeOptions = odeset(RelTol=1.e-7);
t = linspace(0, T, numTimeSteps);
[~, xTrain] = ode45(trueModel, t, x0, odeOptions);
xTrain = xTrain';
```

Visualize the training data in a plot.

```
plot(xTrain(1,:),xTrain(2,:))
legend("Ground truth dynamics")
xlabel("x(1)")
ylabel("x(2)")
grid on
```



Define and Initialize Model Parameters

The model function consists of a single call to `dlode45` to solve the ODE defined by the approximated dynamics $f(t, y, \theta)$ for 40 time steps.

```
neuralOdeTimesteps = 40;
dt = t(2);
timesteps = (0:neuralOdeTimesteps)*dt;
```

Define the learnable parameters to use in the call to `dlode45` and collect them in the variable `neuralOdeParameters`. The function `initializeGlorot` takes as input the size of the learnable parameters `sz` and the number of outputs and number of inputs of the fully connected operations, and returns a `dlarray` object with underlying type `'single'` with values set using Glorot initialization. The function `initializeZeros` takes as input the size of the learnable parameters, and returns the parameters as a `dlarray` object with underlying type `'single'`. The initialization example functions are attached to this example as supporting files. To access these functions, open this example as a live script. For more information about initializing learnable parameters for model functions, see “Initialize Learnable Parameters for Model Function” on page 18-292.

Initialize the parameters structure.

```
neuralOdeParameters = struct;
```

Initialize the parameters for the fully connected operations in the ODE model. The first fully connected operation takes as input a vector of size `stateSize` and increases its length to `hiddenSize`. Conversely, the second fully connected operation takes as input a vector of length `hiddenSize` and decreases its length to `stateSize`.

```
stateSize  = size(xTrain,1);
hiddenSize = 20;

neuralOdeParameters.fc1 = struct;
sz = [hiddenSize stateSize];
neuralOdeParameters.fc1.Weights = initializeGlorot(sz, hiddenSize, stateSize);
neuralOdeParameters.fc1.Bias    = initializeZeros([hiddenSize 1]);

neuralOdeParameters.fc2 = struct;
sz = [stateSize hiddenSize];
neuralOdeParameters.fc2.Weights = initializeGlorot(sz, stateSize, hiddenSize);
neuralOdeParameters.fc2.Bias    = initializeZeros([stateSize 1]);
```

Display the learnable parameters of the model.

```
neuralOdeParameters.fc1
```

```
ans = struct with fields:
    Weights: [20×2 dlarray]
       Bias: [20×1 dlarray]
```

```
neuralOdeParameters.fc2
```

```
ans = struct with fields:
    Weights: [2×20 dlarray]
       Bias: [2×1 dlarray]
```

Define Neural ODE Model

Create the function `odeModel`, listed in the ODE Model on page 18-0 section of the example, which takes as input the time input (unused), the corresponding solution, and the ODE function parameters. The function applies a fully connected operation, a tanh operation, and another fully connected operation to the input data using the weights and biases given by the parameters.

Define Model Function

Create the function `model`, listed in the Model Function on page 18-0 section of the example, which computes the outputs of the deep learning model. The function `model` takes as input the model parameters and the input data. The function outputs the solution of the neural ODE.

Define Model Gradients Function

Create the function `modelGradients`, listed in the Model Gradients Function on page 18-0 section of the example, which takes as input the model parameters, a mini-batch of input data with corresponding targets, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Specify Training Options

Specify options for Adam optimization.

```
gradDecay = 0.9;
sqGradDecay = 0.999;
learnRate = 0.002;
```

Train for 1200 iterations with a mini-batch-size of 200.

```
numIter = 1200;
miniBatchSize = 200;
```

Every 50 iterations, solve the learned dynamics and display them against the ground truth in a phase diagram to show the training path.

```
plotFrequency = 50;
```

Train Model Using Custom Training Loop

Train the network using a custom training loop.

For each iteration:

- Construct a mini-batch of data from the synthesized data with the `createMiniBatch` function, listed in the Create Mini-Batches Function on page 18-0 section of the example.
- Evaluate the model gradients and loss using the `dlfeval` function and the `modelGradients` function, listed in the Model Gradients Function on page 18-0 section of the example.
- Update the model parameters using the `adamupdate` function.
- Update the training progress plot.

Initialize the training progress plot.

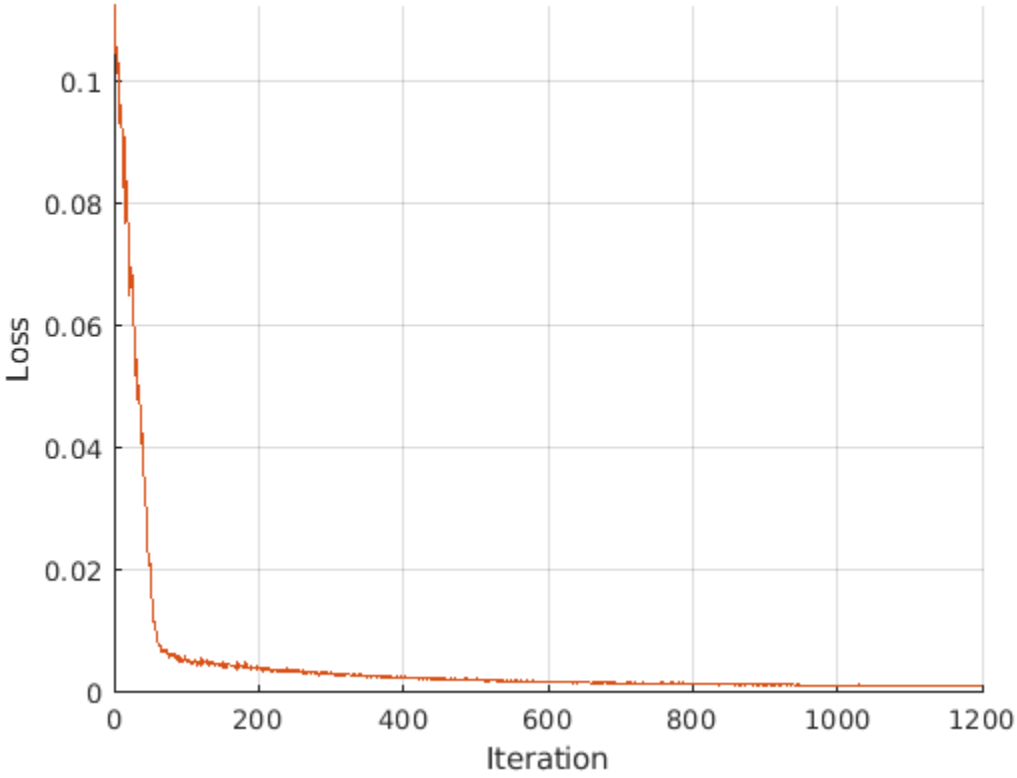
```
figure(2)
lineLossTrain = animatedline(Color=[0.85 0.325 0.098]);
ylim([0 inf])
xlabel("Iteration")
ylabel("Loss")
grid on
```

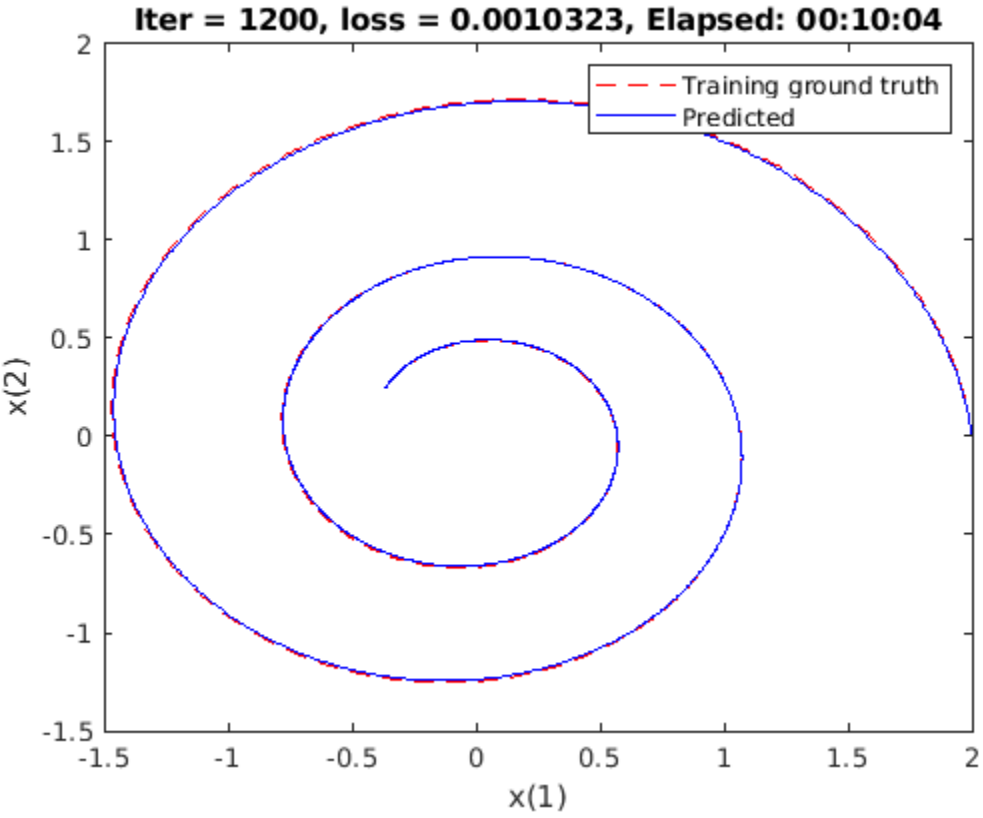
Initialize the `averageGrad` and `averageSqGrad` parameters for the Adam solver.

```
averageGrad   = [];
averageSqGrad = [];
```

Initialize the `lossHistory` array to record the evolution of the training loss.

```
lossHistory = [];
numTrainingTimesteps = numTimeSteps;
trainingTimesteps = 1:numTrainingTimesteps;
plottingTimesteps = 2:numTimeSteps;

start = tic;

for iter=1:numIter

    % Create batch
    [dlx0, targets] = createMiniBatch(numTrainingTimesteps, neuralOdeTimesteps, miniBatchSize, x

    % Evaluate network and compute gradients
    [grads,loss] = dlfeval(@modelGradients,timesteps,dlx0,neuralOdeParameters,targets);

    % Update network
    [neuralOdeParameters,averageGrad,averageSqGrad] = adamupdate(neuralOdeParameters,grads,averag
        learnRate,gradDecay,sqGradDecay);

    % Plot loss
    currentLoss = double(extractdata(loss));

    figure(2)
    addpoints(lineLossTrain, iter, currentLoss);
    drawnow

    % Plot predicted vs. real dynamics
    if mod(iter,plotFrequency) == 0  || iter == 1
        figure(3)
        clf

        % Use ode45 to compute the solution
        y = dlode45(@odeModel,t,dlarray(x0),neuralOdeParameters,DataFormat="CB");
        y = extractdata(y);

        plot(xTrain(1,plottingTimesteps),xTrain(2,plottingTimesteps),"r--")
        hold on
        plot(y(1,:),y(2,:),"b-")
        xlabel("x(1)")
        ylabel("x(2)")
        hold off
        D = duration(0,0,toc(start),Format="hh:mm:ss");
        title("Iter = " + iter + ", loss = " + num2str(currentLoss) + ", Elapsed: " + string(D))
        legend("Training ground truth", "Predicted")
    end
end
```


0.1
0.08
0.06
0.04
0.02
0
Loss
0
200
400
600
800
1000
1200
Iteration



Evaluate Model

Use the model to compute approximated solutions with different initial conditions.

Define four new initial conditions different from the one used for training the model.

```
tPred = t;

x0Pred1 = sqrt([2;2]);
x0Pred2 = [-1;-1.5];
x0Pred3 = [0;2];
x0Pred4 = [-2;0];
```

Numerically solve the ODE true dynamics with `ode45` for the four new initial conditions.

```
[~, xTrue1] = ode45(trueModel, tPred, x0Pred1, odeOptions);

[~, xTrue2] = ode45(trueModel, tPred, x0Pred2, odeOptions);

[~, xTrue3] = ode45(trueModel, tPred, x0Pred3, odeOptions);

[~, xTrue4] = ode45(trueModel, tPred, x0Pred4, odeOptions);
```

Numerically solve the ODE with the learned neural ODE dynamics.

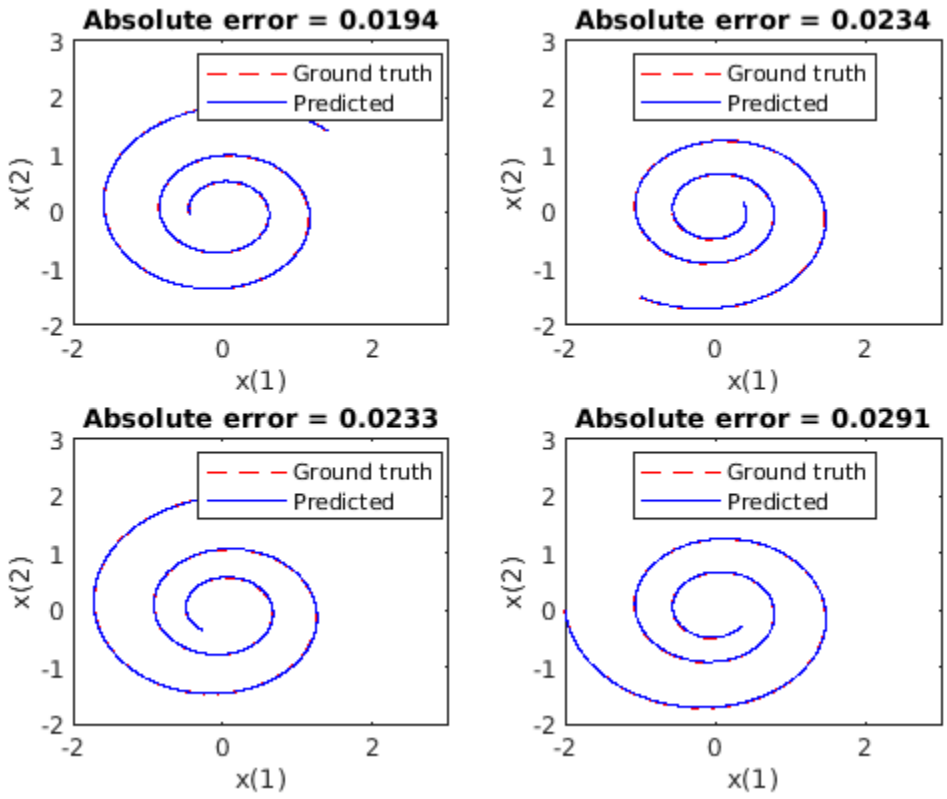
```
xPred1 = dlode45(@odeModel,tPred,dlarray(x0Pred1),neuralOdeParameters,DataFormat="CB");
xPred1 = extractdata(squeeze(xPred1))';
```

```
xPred2 = dlode45(@odeModel,tPred,dlarray(x0Pred2),neuralOdeParameters,DataFormat="CB");
xPred2 = extractdata(squeeze(xPred2))';

xPred3 = dlode45(@odeModel,tPred,dlarray(x0Pred3),neuralOdeParameters,DataFormat="CB");
xPred3 = extractdata(squeeze(xPred3))';

xPred4 = dlode45(@odeModel,tPred,dlarray(x0Pred4),neuralOdeParameters,DataFormat="CB");
xPred4 = extractdata(squeeze(xPred4))';
```

Visualize Predictions

Visualize the predicted solutions for different initial conditions against the ground truth solutions with the function `plotTrueAndPredictedSolutions`, listed in the Plot True and Predicted Solutions Function on page 18-0 section of the example.

```
subplot(2,2,1)
plotTrueAndPredictedSolutions(xTrue1, xPred1);

subplot(2,2,2)
plotTrueAndPredictedSolutions(xTrue2, xPred2);

subplot(2,2,3)
plotTrueAndPredictedSolutions(xTrue3, xPred3);

subplot(2,2,4)
plotTrueAndPredictedSolutions(xTrue4, xPred4);
```



Helper Functions

Model Function

The `model` function, which defines the neural network used to make predictions, is composed of a single neural ODE call. For each observation, this function takes a vector of length `stateSize`, which is used as initial condition for solving numerically the ODE with the function `odeModel` on page 18-0 , which represents the learnable right-hand side $f(t, y, \theta)$ of the ODE to be solved, as right hand side and a vector of time points `tspan` defining the time at which the numerical solution is output. The function uses the vector `tspan` for each observation, regardless of the initial condition, since the learned system is autonomous. That is, the `odeModel` function does not explicitly depend on time.

```
function X = model(tspan,X0,neuralOdeParameters)

X = dlode45(@odeModel,tspan,X0,neuralOdeParameters,DataFormat="CB");

end
```

ODE Model

The `odeModel` function is the learnable right-hand side used in the call to `dlode45`. It takes as input a vector of size `stateSize`, enlarges it so that it has length `hiddenSize`, and applies a nonlinearity function `tanh`. Then the function compresses the vector again to have length `stateSize`.

```
function y = odeModel(~,y,theta)

y = tanh(theta.fc1.Weights*y + theta.fc1.Bias);
y = theta.fc2.Weights*y + theta.fc2.Bias;

end
```

Model Gradients Function

This function takes as inputs a vector `tspan`, a set of initial conditions `dlX0`, the learnable parameters `neuralOdeParameters`, and target sequences `targets`. It computes the predictions with the `model` function, and compares them with the given targets sequences. Finally, it computes the gradient with respect to the learnable parameters of the neural ODE.

```
function [gradients,loss] = modelGradients(tspan,dlX0,neuralOdeParameters,targets)

% Compute predictions.
dlX = model(tspan,dlX0,neuralOdeParameters);

% Compute L1 loss.
loss = l1loss(dlX,targets,NormalizationFactor="all-elements",DataFormat="CBT");

% Compute gradients.
gradients = dlgradient(loss,neuralOdeParameters);
end
```

Create Mini-Batches Function

The `createMiniBatch` function creates a batch of observations of the target dynamics. It takes as input the total number of time steps of the ground truth data `numTimesteps`, the number of consecutive time steps to be returned for each observation `numTimesPerObs`, the number of observations `miniBatchSize`, and the ground truth data `X`.

```
function [x0, targets] = createMiniBatch(numTimesteps,numTimesPerObs,miniBatchSize,X)

% Create batches of trajectories.
s = randperm(numTimesteps - numTimesPerObs, miniBatchSize);

x0 = dlarray(X(:, s));
targets = zeros([size(X,1) miniBatchSize numTimesPerObs]);

for i = 1:miniBatchSize
    targets(:, i, 1:numTimesPerObs) = X(:, s(i) + 1:(s(i) + numTimesPerObs));
end

end
```

Plot True and Predicted Solutions Function

The `plotTrueAndPredictedSolutions` function takes as input the true solution `xTrue`, the approximated solution `xPred` computed with the learned neural ODE model, and the corresponding initial condition `x0Str`. It computes the error between the true and predicted solutions and plots it in a phase diagram.

```
function plotTrueAndPredictedSolutions(xTrue,xPred)

err = mean(abs(xTrue(2:end,:) - xPred), "all");

plot(xTrue(:,1),xTrue(:,2),"r--",xPred(:,1),xPred(:,2),"b-",LineWidth=1)

title("Absolute error = " + num2str(err,'%.4f') )
xlabel("x(1)")
ylabel("x(2)")

xlim([-2 3])
ylim([-2 3])

legend("Ground truth","Predicted",Location="best")
end
```

[1] Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. “Neural Ordinary Differential Equations.” Preprint, submitted December 13, 2019. https://arxiv.org/abs/1806.07366.

[2] Shampine, Lawrence F., and Mark W. Reichelt. “The MATLAB ODE Suite.” SIAM Journal on Scientific Computing 18, no. 1 (January 1997): 1–22. https://doi.org/10.1137/S1064827594276424.

See Also

`dlode45` | `dlarray` | `dlgradient` | `dlfeval` | `adamupdate`

More About

- “Train Neural ODE Network” on page 3-156
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Train Network Using Custom Training Loop” on page 18-225
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “List of Functions with dlarray Support” on page 18-423

Node Classification Using Graph Convolutional Network

This example shows how to classify nodes in a graph using Graph Convolutional Network (GCN).

The node classification task is one where an algorithm, in this example, a GCN [1], has to predict the labels of unlabelled nodes in a graph. In this example, a graph is represented by a molecule. Atoms in the molecule represent nodes in the graph and the chemical bonds between atoms represent edges in the graph. Node labels are the types of atom, for example, Carbon. As such, input to the GCN are molecules and the outputs are predictions of the type of atom of each unlabelled atom in the molecule.

To assign a categorical label to each node of a graph, the GCN models a function $f(X, A)$ on a graph $G = (V, E)$, where V denotes the set of nodes and E denotes the set of edges, such that$f(X, A)$ takes as input:

- X: A feature matrix of dimension $N \times C$, where $N = |V|$ is the number of nodes in G and C is number of input channels/features per node.
- A: An adjacency matrix of dimension $N \times N$ representing E and describing the structure of G.

and returns an output:

- Z: An Embedding or feature matrix of dimension $N \times F$, where F is number of output features per node. In other words, Z is the predictions of the network and F is the number of classes.

The model $f(X, A)$ is based on spectral graph convolution, with weights/filter parameters shared over all locations in G. The model can be represented as a layer-wise propagation model, such that the output of layer $l + 1$ is expressed as

$$Z_{l+1} = \sigma\left(\widehat{D}^{-1/2}\widehat{A}\widehat{D}^{-1/2}Z_l W_l\right),$$

where

- σ is an activation function.
- Z_l is the activation matrix of layer l, with $Z_1 = X$.
- W_l is the weight matrix of layer l.
- $\widehat{A} = A + I_N$ is the adjacency matrix of graph G with added self-connections. I_N is the identity matrix.
- $\widehat{D}$ is the degree matrix of $\widehat{A}$.

Expression $\widehat{D}^{-1/2}\widehat{A}\widehat{D}^{-1/2}$ can be referred to as the *normalized* adjacency matrix of the graph.

The GCN model in this example is a variant of the standard GCN model described above. The variant uses residual connections between layers [1]. The residual connections enable the model to carry over information from previous layer's input. Therefore, the output of layer $l + 1$of the GCN model in this example is

$$Z_{l+1} = \sigma\left(\widehat{D}^{-1/2}\widehat{A}\widehat{D}^{-1/2}Z_l W_l\right) + Z_l,$$

See [1] for more details about the GCN model.

This example uses the QM7 dataset [2] [3], which is a molecular dataset consisting of 7165 molecules composed of up to 23 atoms. That is, the molecule with the highest number of atoms has 23 atoms. Overall, the dataset consists of 5 unique atoms: Carbon, Hydrogen, Nitrogen, Oxygen, and Sulphur.

Download and Load QM7 data

Download the QM7 dataset from the following URL:

```
dataURL = 'http://quantum-machine.org/data/qm7.mat';
outputFolder = fullfile(tempdir,'qm7Data');
dataFile = fullfile(outputFolder,'qm7.mat');

if ~exist(dataFile, 'file')
    mkdir(outputFolder);
    fprintf('Downloading file ''%s'' ...\n', dataFile);
    websave(dataFile, dataURL);
end
```

Load QM7 data.

```
data = load(dataFile)
```

```
data = struct with fields:
    X: [7165×23×23 single]
    R: [7165×23×3 single]
    Z: [7165×23 single]
    T: [1×7165 single]
    P: [5×1433 int64]
```

The data consists of five different arrays. This example uses the arrays in fields `X` and `Z` of struct `data`. The array in `X` represents the Coulomb matrix [3] representation of each molecule, totalling 7165 molecules, and the array in `Z` represents the atomic charge/number of each atom in the molecules. The adjacency matrices of the graphs representing the molecules, and the feature matrices of the graphs, are extracted from the Coulomb matrices. The categorical array of labels is extracted from the array in `Z`.

Note that the data, for any molecule that does not have up to 23 atoms, contains padded zeros. For example, the data representing the atomic numbers of atoms in the molecule at index 1 is

```
data.Z(1,:)
```

```
ans = 1×23 single row vector

     6     1     1     1     1     0     0     0     0     0     0     0     0     0     0     0
```

This shows that this molecule is composed of five atoms; one atom with atomic number 6 and four atoms with atomic number 1, and the data is padded with 18 zeros.

Extract and Preprocess Graph Data

To extract graph data, get the Coulomb matrices and atomic numbers. Permute the data representing the Coulomb matrices and change the datatype to double. Sort the data representing the atomic charges so that it matches the data representing the Coulomb matrices.

```
coulombData = double(permute(data.X, [2 3 1]));
atomicNumber = sort(data.Z,2,'descend');
```

Reformat the Coulomb matrix representation of the molecules to binary adjacency matrices using the `coloumb2Adjacency` function attached to this example as a supporting file.

```
adjacencyData = coloumb2Adjacency(coulombData, atomicNumber);
```

Note that the `coloumb2Adjacency` function does not remove padded zeros from the data. They are left intentionally to make it easier to split the data into separate molecules for training, validation and inference. Therefore, ignoring the padded zeros, the adjacency matrix of the graph representing the molecule at index 1, which consists of 5 atoms, is

```
adjacencyData(1:5,1:5,1)
```

```
ans = 5×5

     0     1     1     1     1
     1     0     0     0     0
     1     0     0     0     0
     1     0     0     0     0
     1     0     0     0     0
```

Before preprocessing the data, use the `splitData` on page 18-0 function, provided at the end of the example, to randomly select and split the data into training, validation and test data. The function uses the ratio 80:10:10 to split the data.

The `adjacencyDataSplit` output of the `splitData` function is the `adjacencyData` input data split into three different arrays. Likewise, the `coulombDataSplit` and `atomicNumberSplit` outputs are the `coulombData` and `atomicNumber` input data split into three different arrays respectively.

```
[adjacencyDataSplit, coulombDataSplit, atomicNumberSplit] = splitData(adjacencyData, coulombData
```

Use the `preprocessData` on page 18-0 function, provided at the end of the example, to process the `adjacencyDataSplit`, `coulombDataSplit,` and `atomicNumberSplit` and return the adjacency matrix `adjacency`, feature matrix `features`, and categorical array `labels`.

The `preprocessData` function builds a sparse block-diagonal matrix of the adjacency matrices of different graph instances, such that, each block in the matrix corresponds to the adjacency matrix of one graph instance. This preprocessing is required because GCN accepts a single adjacency matrix as input, whereas this example deals with multiple graph instances. The function takes the non-zero diagonal elements of the Coulomb matrices and assigns them as features. Therefore, the number of input features per node in the example is 1.

```
[adjacency, features, labels] = cellfun(@preprocessData, adjacencyDataSplit, coulombDataSplit, at
```

View the adjacency matrices of the training, validation, and test data.

```
adjacency
```

```
adjacency=1×3 cell array
    {88722×88722 double}    {10942×10942 double}    {10986×10986 double}
```

This shows that there are 88722 nodes in the training data, 10942 nodes in the validation data, and 10986 nodes in the test data.

Normalize the feature array using the `normalizeFeatures` on page 18-0 function provided at the end of the example.

```
features = normalizeFeatures(features);
```

Get the training and the validation data.

```
featureTrain = features{1};
adjacencyTrain = adjacency{1};
targetTrain = labels{1};

featureValidation = features{2};
adjacencyValidation = adjacency{2};
targetValidation = labels{2};
```

Visualize Data and Data Statistics

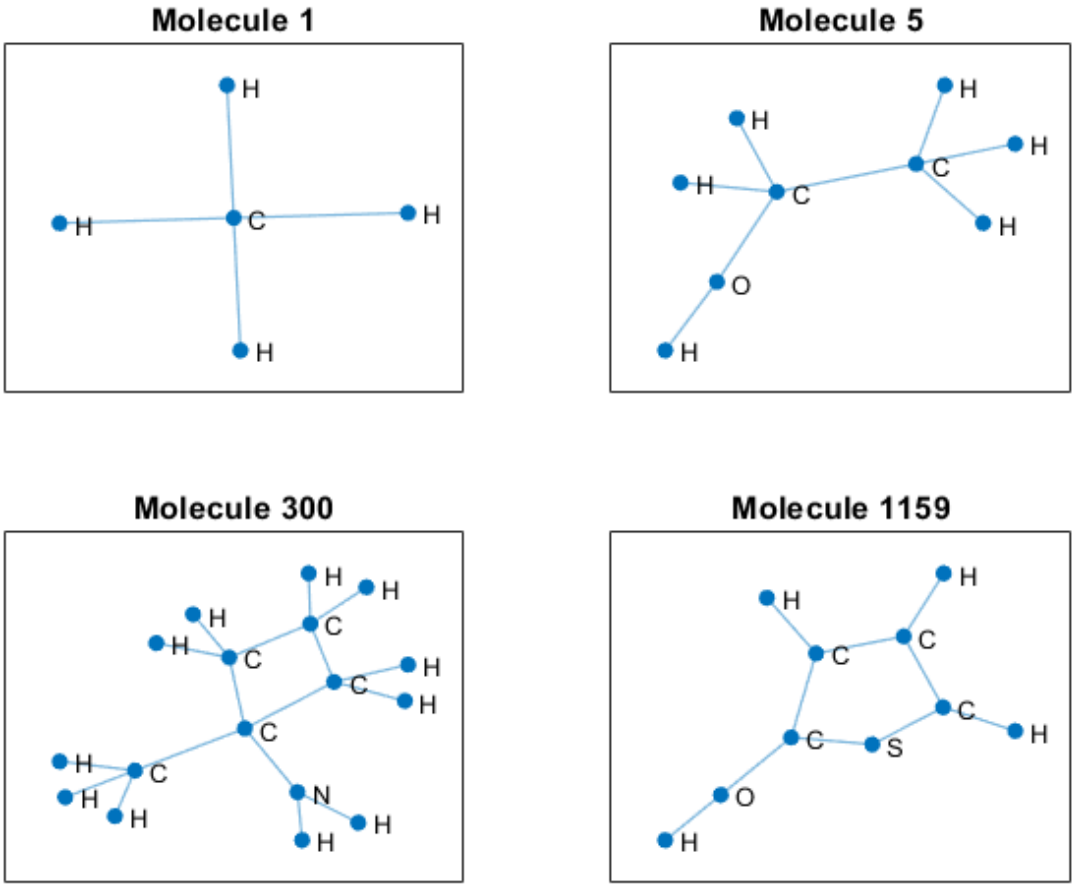
Sample and specify indices of molecules to visualize.

For each specified index

- Remove padded zeros from the data representing unprocessed atomic numbers `atomicNumber` and unprocessed adjacency matrix `adjacencyData` of the sampled molecule. The unprocessed data are used here for easy sampling.
- Convert the adjacency matrix to graph using the `graph` function.
- Convert the atomic numbers to symbols.
- Plot the graph using the atomic symbols as node labels.

```
idx = [1 5 300 1159];
for j = 1:numel(idx)
    % Remove padded zeros from the data
    atomicNum = nonzeros(atomicNumber(idx(j),:));
    numOfNodes = numel(atomicNum);
    adj = adjacencyData(1:numOfNodes,1:numOfNodes,idx(j));

    % Convert adjacency matrix to graph
    compound = graph(adj);

    % Convert atomic numbers to symbols
    symbols = cell(numOfNodes, 1);
    for i = 1:numOfNodes
        if atomicNum(i) == 1
            symbols{i} = 'H';
        elseif atomicNum(i) == 6
            symbols{i} = 'C';
        elseif atomicNum(i) == 7
            symbols{i} = 'N';
        elseif atomicNum(i) == 8
            symbols{i} = 'O';
        else
            symbols{i} = 'S';
        end
    end

    % Plot graph
    subplot(2,2,j)
    plot(compound, 'NodeLabel', symbols, 'LineWidth', 0.75, ...
```

```
    'Layout', 'force')
    title("Molecule " + idx(j))
end
```



Get all the labels and the classes.

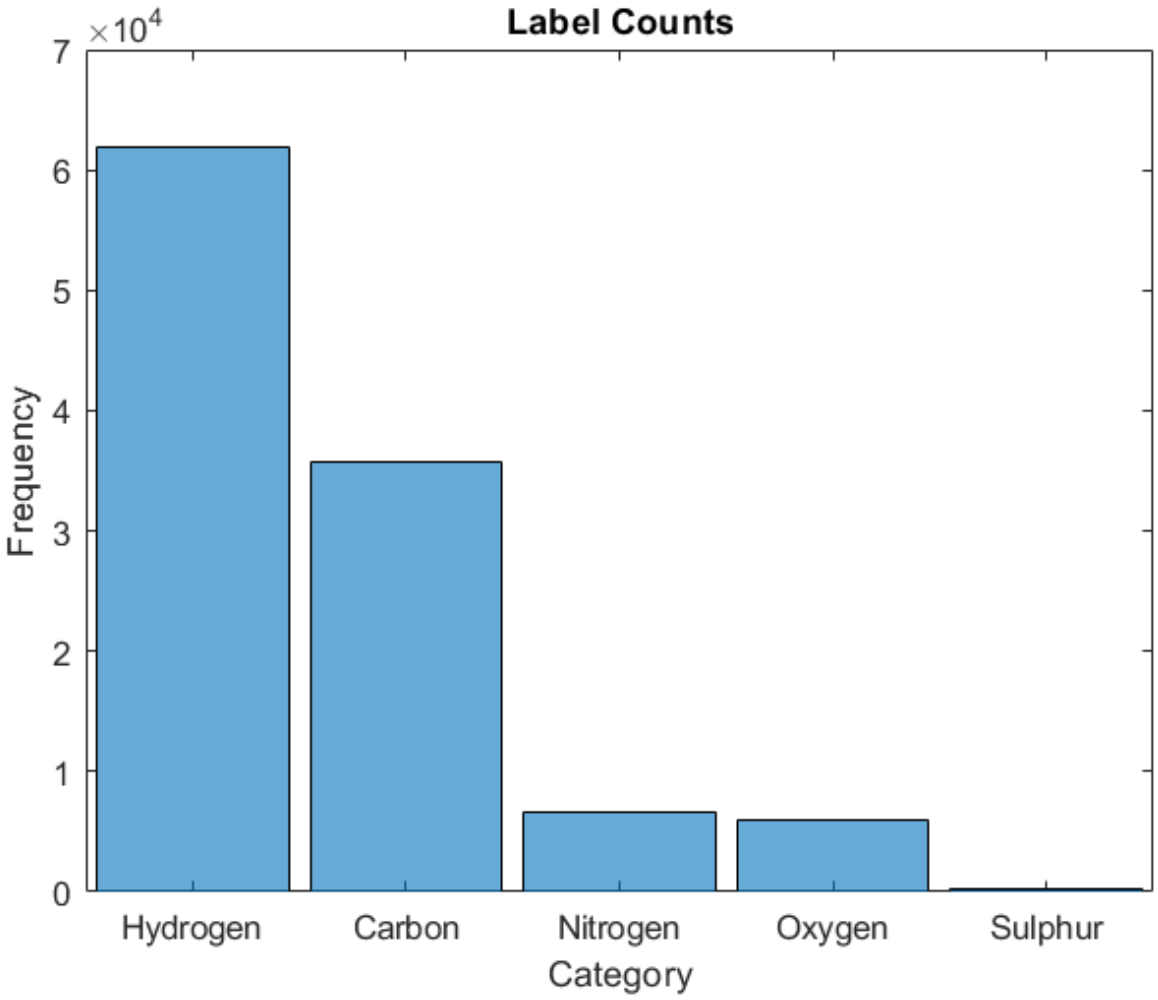
```
labelsAll = cat(1,labels{:});
classes = categories(labelsAll)
```

```
classes = 5×1 cell
    {'Hydrogen'}
    {'Carbon'  }
    {'Nitrogen'}
    {'Oxygen'  }
    {'Sulphur' }
```

Visualize frequency of each label category using a histogram.

```
figure
histogram(labelsAll)
```

```
xlabel('Category')
ylabel('Frequency')
title('Label Counts')
```



Define Model Function

Create the function model on page 18-0 , provided at the end of the example, that takes the feature data `dlX`, the adjacency matrix `A`, and the model parameters `parameters` as input and returns predictions for the label.

Initialize Model Parameters

Set the number of input features per node. This is the column length of the feature matrix.

```
numInputFeatures = size(featureTrain,2)
```

```
numInputFeatures = 1
```

Set the number of feature maps for the hidden layers.

```
numHiddenFeatureMaps = 32;
```

Set the number of output features as the number of categories.

```
numOutputFeatures = numel(classes)
```

```
numOutputFeatures = 5
```

Create a struct `parameters` containing the model weights. Initialize the weights using the `initializeGlorot` function attached to this example as a supporting file.

```
sz = [numInputFeatures numHiddenFeatureMaps];
numOut = numHiddenFeatureMaps;
numIn = numInputFeatures;
parameters.W1 = initializeGlorot(sz,numOut,numIn,'double');

sz = [numHiddenFeatureMaps numHiddenFeatureMaps];
numOut = numHiddenFeatureMaps;
numIn = numHiddenFeatureMaps;
parameters.W2 = initializeGlorot(sz,numOut,numIn,'double');

sz = [numHiddenFeatureMaps numOutputFeatures];
numOut = numOutputFeatures;
numIn = numHiddenFeatureMaps;
parameters.W3 = initializeGlorot(sz,numOut,numIn,'double');
parameters
```

```
parameters = struct with fields:
    W1: [1×32 dlarray]
    W2: [32×32 dlarray]
    W3: [32×5 dlarray]
```

Define Model Gradients Function

Create the function modelGradients on page 18-0 , provided at the end of the example, that takes the feature data `dlX`, the adjacency matrix `adjacencyTrain`, the one-hot encoded targets `T` of the labels, and the model parameters `parameters` as input and returns the gradients of the loss with respect to the parameters, the corresponding loss, and the network predictions.

Specify Training Options

Train for 1500 epochs and set the learn rate for Adam solver to 0.01.

```
numEpochs = 1500;
learnRate = 0.01;
```

Validate the network after every 300 epochs.

```
validationFrequency = 300;
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

To train on a GPU if one is available, specify the execution environment `"auto"`. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox) (Parallel Computing Toolbox).

```
executionEnvironment = "auto";
```

Train Model

Train the model using a custom training loop. The training uses full-batch gradient descent.

Initialize the training progress plot.

```
if plots == "training-progress"
    figure

    % Accuracy.
    subplot(2,1,1)
    lineAccuracyTrain = animatedline('Color',[0 0.447 0.741]);
    lineAccuracyValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 1])
    xlabel("Epoch")
    ylabel("Accuracy")
    grid on

    % Loss.
    subplot(2,1,2)
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
    lineLossValidation = animatedline( ...
        'LineStyle','--', ...
        'Marker','o', ...
        'MarkerFaceColor','black');
    ylim([0 inf])
    xlabel("Epoch")
    ylabel("Loss")
    grid on
end
```

Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Convert training and validation feature data to dlarray.

```
dlX = dlarray(featureTrain);
dlXValidation = dlarray(featureValidation);
```

For GPU training, convert data to `gpuArray` objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end
```

Encode training and validation label data using `onehotencode`.

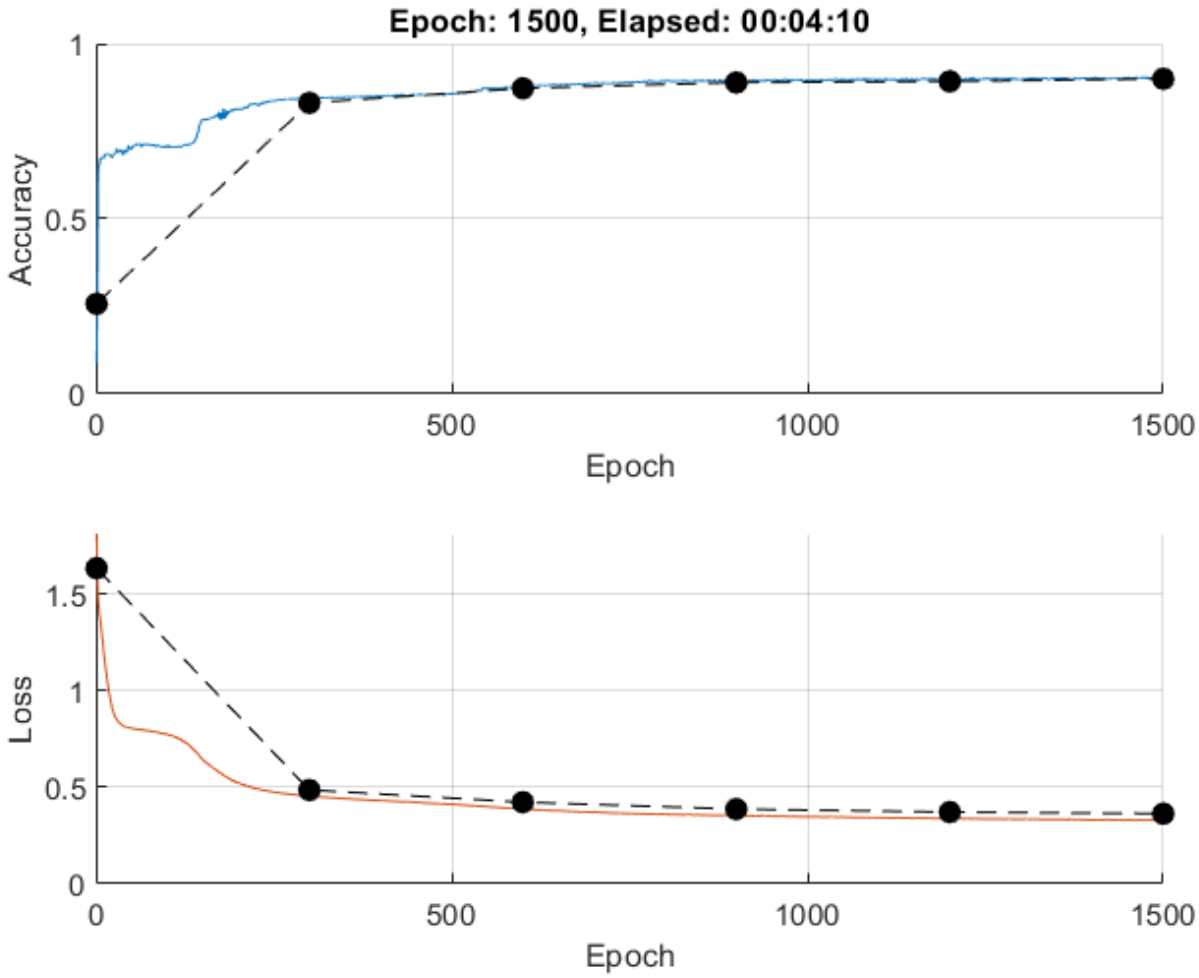
```
T = onehotencode(targetTrain, 2, 'ClassNames', classes);
TValidation = onehotencode(targetValidation, 2, 'ClassNames', classes);
```

Train the model.

For each epoch

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the network parameters using `adamupdate`.
- Compute the training accuracy score using the accuracy on page 18-0 function provided at the end of the example. The function takes the network predictions, the target containing the labels, and the categories `classes` as inputs and returns the accuracy score.
- If required, validate the network by making predictions using the `model` function and computing the validation loss and the validation accuracy score using `crossentropy` and the `accuracy` function.
- Update the training plot.

```
start = tic;
% Loop over epochs.
for epoch = 1:numEpochs

    % Evaluate the model gradients and loss using dlfeval and the
    % modelGradients function.
    [gradients, loss, dlYPred] = dlfeval(@modelGradients, dlX, adjacencyTrain, T, parameters);

    % Update the network parameters using the Adam optimizer.
    [parameters,trailingAvg,trailingAvgSq] = adamupdate(parameters,gradients, ...
        trailingAvg,trailingAvgSq,epoch,learnRate);

    % Display the training progress.
    if plots == "training-progress"
        subplot(2,1,1)
        D = duration(0,0,toc(start),'Format','hh:mm:ss');
        title("Epoch: " + epoch + ", Elapsed: " + string(D))

        % Loss.
        addpoints(lineLossTrain,epoch,double(gather(extractdata(loss))))

        % Accuracy score.
        score = accuracy(dlYPred, targetTrain, classes);
        addpoints(lineAccuracyTrain,epoch,double(gather(score)))

        drawnow

        % Display validation metrics.
        if epoch == 1 || mod(epoch,validationFrequency) == 0
            % Loss.
            dlYPredValidation = model(dlXValidation, adjacencyValidation, parameters);
            lossValidation = crossentropy(dlYPredValidation, TValidation, 'DataFormat', 'BC');
            addpoints(lineLossValidation,epoch,double(gather(extractdata(lossValidation))))

            % Accuracy score.
            scoreValidation = accuracy(dlYPredValidation, targetValidation, classes);
            addpoints(lineAccuracyValidation,epoch,double(gather(scoreValidation)))

            drawnow
        end
    end
end
```



Test Model

Test the model using the test data.

```
featureTest = features{3};
adjacencyTest = adjacency{3};
targetTest = labels{3};
```

Convert the test feature data to dlarray.

```
dlXTest = dlarray(featureTest);
```

Make predictions on the data.

```
dlYPredTest = model(dlXTest, adjacencyTest, parameters);
```

Calculate the accuracy score using the `accuracy` function. The `accuracy` function also returns a decoded network predictions `predTest` as class labels. The network predictions are decoded using `onehotdecode`.

```
[scoreTest, predTest] = accuracy(dlYPredTest, targetTest, classes);
```

View the accuracy score.

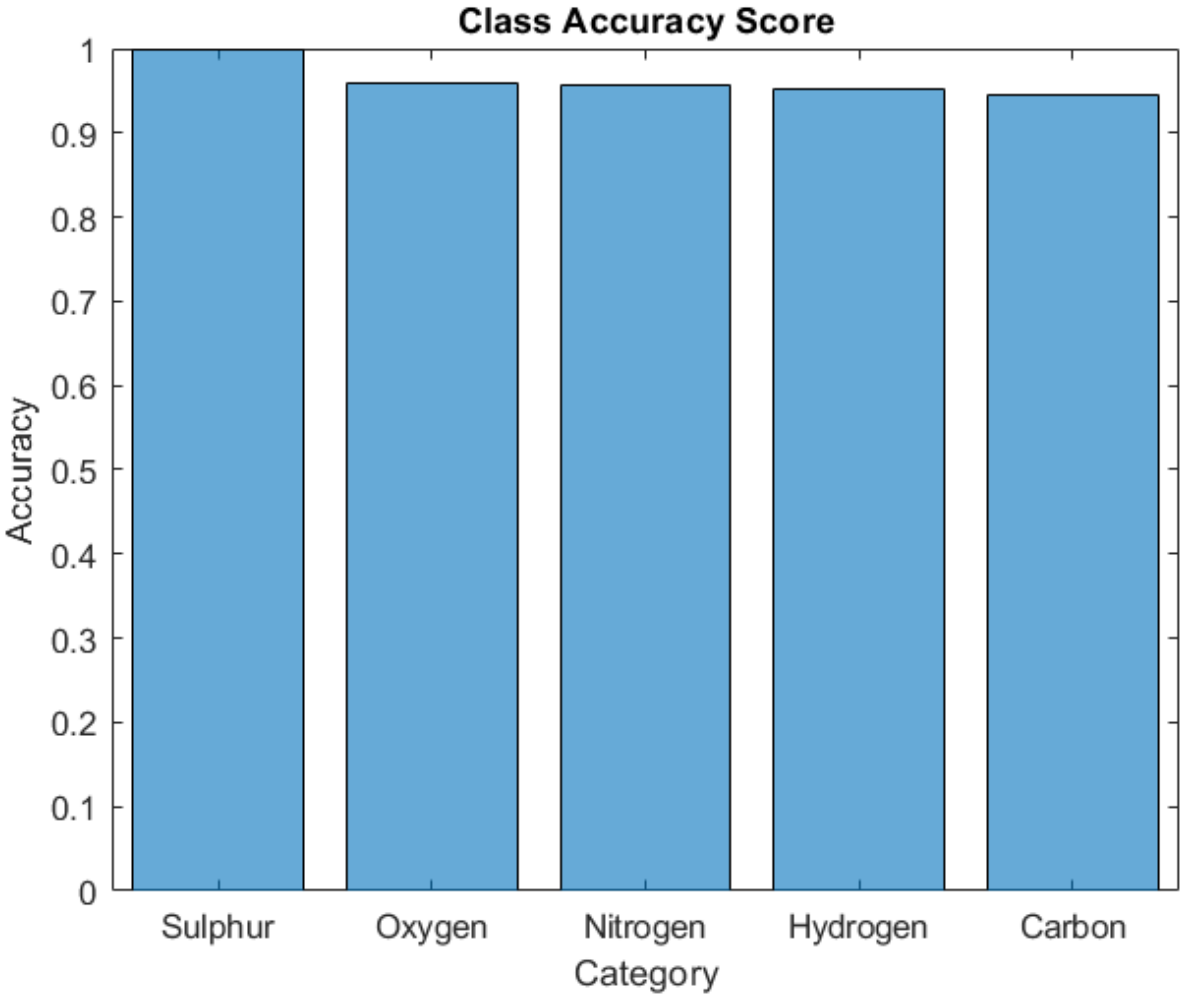
```
scoreTest
```

```
scoreTest = 0.9053
```

Visualize Predictions

To visualize the accuracy score for each category, compute the class-wise accuracy scores and visualize them using a histogram.

```
numOfSamples = numel(targetTest);
classTarget = zeros(numOfSamples, numOutputFeatures);
classPred = zeros(numOfSamples, numOutputFeatures);
for i = 1:numOutputFeatures
    classTarget(:,i) = targetTest==categorical(classes(i));
    classPred(:,i) = predTest==categorical(classes(i));
end

% Compute class-wise accuracy score
classAccuracy = sum(classPred == classTarget)./numOfSamples;

% Visualize class-wise accuracy score
figure
[~,idx] = sort(classAccuracy,'descend');
histogram('Categories',classes(idx), ...
    'BinCounts',classAccuracy(idx), ...
    'Barwidth',0.8)
xlabel("Category")
ylabel("Accuracy")
title("Class Accuracy Score")
```



The class-wise accuracy scores show how the model makes correct predictions using both the true positives and the true negatives. A true positive is an outcome where the model correctly predicts a class as present in an observation. A true negative is an outcome where the model correctly predicts a class as absent in an observation.

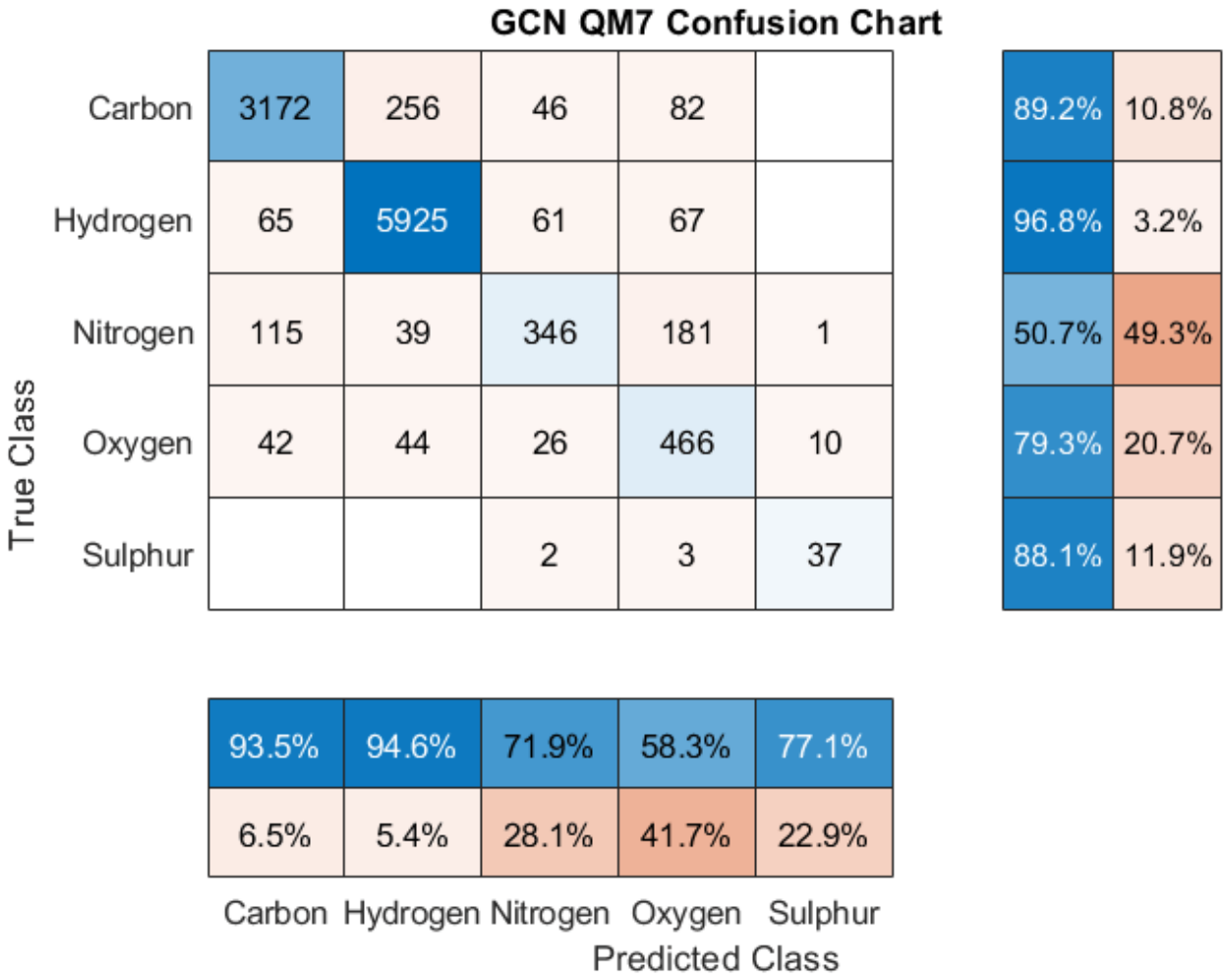
To visualize how the model makes incorrect predictions and evaluate the model based on class-wise precision and class-wise recall, calculate the confusion matrix using `confusionmat` and visualize the results using `confusionchart`.

Class-wise precision is the ratio of true positives to total positive predictions for a class. The total positive predictions include the true positives and false positives. A false positive is an outcome where the model incorrectly predicts a class as present in an observation.

Class-wise recall, also known as true positive rates, is the ratio of true positives to total positive observations for a class. The total positive observation includes the true positives and false negatives. A false negative is an outcome where the model incorrectly predicts a class as absent in an observation.

```
[confusionMatrix, order] = confusionmat(targetTest, predTest);
figure
```

```
cm = confusionchart(confusionMatrix, classes, ...
    'ColumnSummary','column-normalized', ...
    'RowSummary','row-normalized', ...
    'Title', 'GCN QM7 Confusion Chart');
```



The class-wise precision are the scores in the first row of the 'column summary' of the chart and the class-wise recall are the scores in the first column of the 'row summary' of the chart.

Split Data Function

The `splitData` function takes the `adjacencyData`, `coloumbData`, and `atomicNumber` data and randomly splits them into training, validation and test data in ratio 80:10:10. The function returns the corresponding split data `adjacencyDataSplit, coulombDataSplit, atomicNumberSplit` as cell arrays.

```
function [adjacencyDataSplit, coulombDataSplit, atomicNumberSplit] = splitData(adjacencyData, cou

adjacencyDataSplit = cell(1,3);
coulombDataSplit = cell(1,3);
atomicNumberSplit = cell(1,3);
```

```
numMolecules = size(adjacencyData, 3);

% Set initial random state for example reproducibility.
rng(0);

% Get training data
idx = randperm(size(adjacencyData, 3), floor(0.8*numMolecules));
adjacencyDataSplit{1} = adjacencyData(:,:,idx);
coulombDataSplit{1} = coulombData(:,:,idx);
atomicNumberSplit{1} = atomicNumber(idx,:);
adjacencyData(:,:,idx) = [];
coulombData(:,:,idx) = [];
atomicNumber(idx,:) = [];

% Get validation data
idx = randperm(size(adjacencyData, 3), floor(0.1*numMolecules));
adjacencyDataSplit{2} = adjacencyData(:,:,idx);
coulombDataSplit{2} = coulombData(:,:,idx);
atomicNumberSplit{2} = atomicNumber(idx,:);
adjacencyData(:,:,idx) = [];
coulombData(:,:,idx) = [];
atomicNumber(idx,:) = [];

% Get test data
adjacencyDataSplit{3} = adjacencyData;
coulombDataSplit{3} = coulombData;
atomicNumberSplit{3} = atomicNumber;

end
```

Preprocess Data Function

The `preprocessData` function preprocesses the input data as follows:

For each graph/molecule

- Remove padded zeros from `atomicNumber`.
- Concatenate the atomic number data with the atomic number data of other graph instances. It is necessary to concatenate the data since the example deals with multiple graph instances.
- Remove padded zeros from `adjacencyData`.
- Build a sparse block-diagonal matrix of the adjacency matrices of different graph instances. Each block in the matrix corresponds to the adjacency matrix of one graph instance. This step is also necessary because there are multiple graph instances in the example.
- Extract feature array from `coulombData`. The feature array is the non-zero diagonal elements of the Coulomb matrix in `coulombData`.
- Concatenate the feature array with feature arrays of other graph instances.

The function then converts the atomic number data to categorical arrays.

```
function [adjacency, features, labels] = preprocessData(adjacencyData, coulombData, atomicNumber

adjacency = sparse([]);
features = [];
labels = [];
for i = 1:size(adjacencyData, 3)
```

```
    % Remove padded zeros from atomicNumber
    tmpLabels = nonzeros(atomicNumber(i,:));
    labels = [labels; tmpLabels];

    % Get the indices of the un-padded data
    validIdx = 1:numel(tmpLabels);

    % Use the indices for un-padded data to remove padded zeros
    % from the adjacency data
    tmpAdjacency = adjacencyData(validIdx, validIdx, i);

    % Build the adjacency matrix into a block diagonal matrix
    adjacency = blkdiag(adjacency, tmpAdjacency);

    % Remove padded zeros from coulombData and extract the
    % feature array
    tmpFeatures = diag(coulombData(validIdx, validIdx, i));
    features = [features; tmpFeatures];
end

% Convert labels to categorical array
atomicNumbers = unique(labels);
atomNames = ["Hydrogen","Carbon","Nitrogen","Oxygen","Sulphur"];
labels = categorical(labels, atomicNumbers, atomNames);

end
```

Normalize Features Function

The `normalizeFeatures` function standardizes the input training, validation, and test feature data `features` using the mean and variance of the training data.

```
function features = normalizeFeatures(features)

% Get the mean and variance from the training data
meanFeatures = mean(features{1});
varFeatures = var(features{1}, 1);

% Standardize training, validation and test data
for i = 1:3
    features{i} = (features{i} - meanFeatures)./sqrt(varFeatures);
end

end
```

Model Function

The `model` function takes the feature matrix `dlX`, the adjacency matrix `A`, and the model parameters `parameters` and returns the network predictions. In a preprocessing step, the `model` function calculates the *normalized* adjacency matrix described earlier using the `normalizeAdjacency` function provided.

```
function dlY = model(dlX, A, parameters)

% Normalize adjacency matrix
L = normalizeAdjacency(A);

Z1 = dlX;
```

```
Z2 = L * Z1 * parameters.W1;
Z2 = relu(Z2) + Z1;

Z3 = L * Z2 * parameters.W2;
Z3 = relu(Z3) + Z2;

Z4 = L * Z3 * parameters.W3;
dlY = softmax(Z4, 'DataFormat', 'BC');

end
```

Normalize Adjacency Function

The `normalizeAdjacency` function calculates and returns the *normalized* adjacency matrix `normAdjacency` of the input adjacency matrix `adjacency`.

```
function normAdjacency = normalizeAdjacency(adjacency)

% Add self connections to adjacency matrix
adjacency = adjacency + speye(size(adjacency));

% Compute degree of nodes
degree = sum(adjacency, 2);

% Compute inverse square root of degree
degreeInvSqrt = sparse(sqrt(1./degree));

% Normalize adjacency matrix
normAdjacency = diag(degreeInvSqrt) * adjacency * diag(degreeInvSqrt);

end
```

Model Gradients Function

The `modelGradients` function takes the feature matrix `dlX`, the adjacency matrix `adjacencyTrain`, the one-hot encoded target data `T`, and the model parameters `parameters`, and returns the gradients of the loss with respect to the model parameters, the corresponding loss, and the network predictions.

```
function [gradients, loss, dlYPred] = modelGradients(dlX, adjacencyTrain, T, parameters)

dlYPred = model(dlX, adjacencyTrain, parameters);

loss = crossentropy(dlYPred, T, 'DataFormat', 'BC');

gradients = dlgradient(loss, parameters);

end
```

Accuracy Function

The accuracy function decodes the network predictions `YPred` and calculates accuracy using the decoded predictions and the target data `target`. The function returns the computed accuracy score and the decoded predictions `prediction`.

```
function [score, prediction] = accuracy(YPred, target, classes)
```

```
% Decode probability vectors into class labels
prediction = onehotdecode(YPred, classes, 2);
score = sum(prediction == target)/numel(target);

end
```

References

1. T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In ICLR, 2016.
2. L. C. Blum, J. -L. Reymond, 970 Million Druglike Small Molecules for Virtual Screening in the Chemical Universe Database GDB-13, J. Am. Chem. Soc., 131:8732, 2009.
3. M. Rupp, A. Tkatchenko, K.-R. Müller, O. A. von Lilienfeld: Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning, Physical Review Letters, 108(5):058301, 2012.

See Also

`dlarray` | `dlfeval` | `dlgradient` | `minibatchqueue`

More About

- “Solve Partial Differential Equation with LBFGS Method and Deep Learning” on page 18-351
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216

Train Network Using Cyclical Learning Rate for Snapshot Ensembling

This example shows how to train a network to classify images of objects using a cyclical learning rate schedule and snapshot ensembling for better test accuracy. In the example, you learn how to use a cosine function for the learning rate schedule, take snapshots of the network during training to create a model ensemble, and add L2-norm regularization (weight decay) to the training loss.

This example trains a residual network [1] on the CIFAR-10 data set [2] with a custom cyclical learning rate: for each iteration, the solver uses the learning rate given by a shifted cosine function [3] `alpha(t) = (alpha0/2)*cos(pi*mod(t-1,T/M)/(T/M)+1)`, where `t` is the iteration number, `T` is the total number of training iterations, `alpha0` is the initial learning rate, and `M` is the number of cycles/snapshots. This learning rate schedule effectively splits the training process into `M` cycles. Each cycle begins with a large learning rate that decays monotonically, forcing the network to explore different local minima. At the end of each training cycle, you take a snapshot of the network (that is, you save the model at this iteration) and later average the predictions of all the snapshot models, also known as snapshot ensembling [4], to improve the final test accuracy.

Prepare Data

Download the CIFAR-10 data set [2]. The data set contains 60,000 images. Each image is 32-by-32 in size and has three color channels (RGB). The size of the data set is 175 MB. Depending on your internet connection, the download process can take time.

```
datadir = tempdir;
downloadCIFARData(datadir);
```

Load the CIFAR-10 training and test images as 4-D arrays. The training set contains 50,000 images and the test set contains 10,000 images.

```
[XTrain,YTrain,XTest,YTest] = loadCIFARData(datadir);
classes = categories(YTrain);
numClasses = numel(classes);
```

You can display a random sample of the training images using the following code.

```
figure;
idx = randperm(size(XTrain,4),20);
im = imtile(XTrain(:,:,:,idx),'ThumbnailSize',[96,96]);
imshow(im)
```

Create an `augmentedImageDatastore` object to use for network training. During training, the datastore randomly flips the training images along the vertical axis and randomly translates them up to four pixels horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(imageSize,XTrain,YTrain, ...
    'DataAugmentation',imageAugmenter);
```

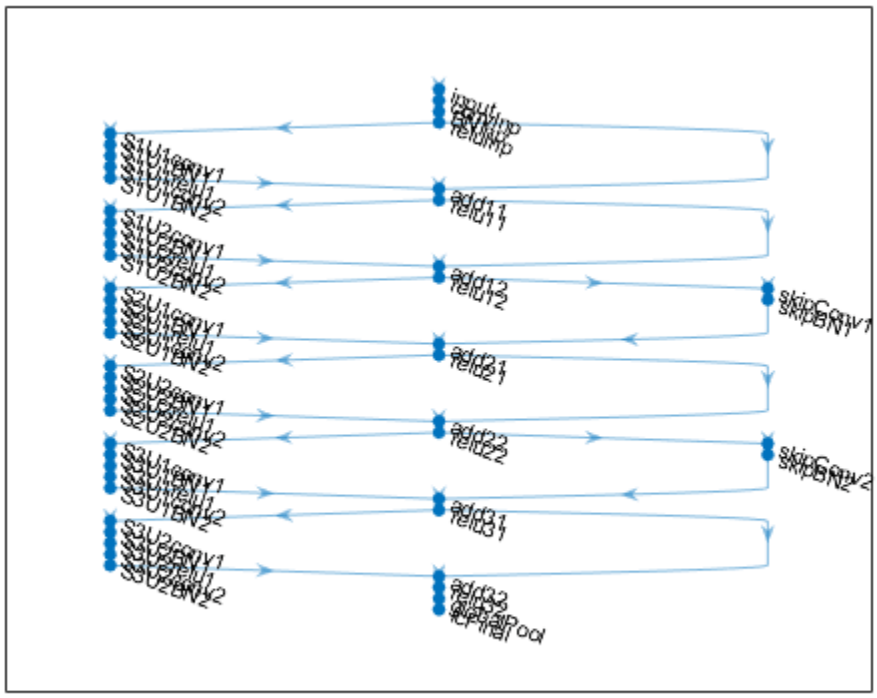
Define Network Architecture

Create a residual network [1] with six standard convolutional units (two units per stage) and a width of 16. The total network depth is 2*6+2 = 14. In addition, specify the average image using the `'Mean'` option in the image input layer.

```
netWidth = 16;
layers = [
    imageInputLayer(imageSize,'Name','input','Mean', mean(XTrain,4))
    convolution2dLayer(3,netWidth,'Padding','same','Name','convInp')
    batchNormalizationLayer('Name','BNInp')
    reluLayer('Name','reluInp')

    convolutionalUnit(netWidth,1,'S1U1')
    additionLayer(2,'Name','add11')
    reluLayer('Name','relu11')
    convolutionalUnit(netWidth,1,'S1U2')
    additionLayer(2,'Name','add12')
    reluLayer('Name','relu12')

    convolutionalUnit(2*netWidth,2,'S2U1')
    additionLayer(2,'Name','add21')
    reluLayer('Name','relu21')
    convolutionalUnit(2*netWidth,1,'S2U2')
    additionLayer(2,'Name','add22')
    reluLayer('Name','relu22')

    convolutionalUnit(4*netWidth,2,'S3U1')
    additionLayer(2,'Name','add31')
    reluLayer('Name','relu31')
    convolutionalUnit(4*netWidth,1,'S3U2')
    additionLayer(2,'Name','add32')
    reluLayer('Name','relu32')

    averagePooling2dLayer(8,'Name','globalPool')
    fullyConnectedLayer(10,'Name','fcFinal')
    ];

lgraph = layerGraph(layers);
lgraph = connectLayers(lgraph,'reluInp','add11/in2');
lgraph = connectLayers(lgraph,'relu11','add12/in2');
skip1 = [
    convolution2dLayer(1,2*netWidth,'Stride',2,'Name','skipConv1')
    batchNormalizationLayer('Name','skipBN1')];
lgraph = addLayers(lgraph,skip1);
lgraph = connectLayers(lgraph,'relu12','skipConv1');
lgraph = connectLayers(lgraph,'skipBN1','add21/in2');
lgraph = connectLayers(lgraph,'relu21','add22/in2');
skip2 = [
    convolution2dLayer(1,4*netWidth,'Stride',2,'Name','skipConv2')
    batchNormalizationLayer('Name','skipBN2')];
lgraph = addLayers(lgraph,skip2);
lgraph = connectLayers(lgraph,'relu22','skipConv2');
lgraph = connectLayers(lgraph,'skipBN2','add31/in2');
lgraph = connectLayers(lgraph,'relu31','add32/in2');
```

Plot the ResNet architecture.

```
figure;
plot(lgraph)
```



Create a `dlnetwork` object from the layer graph.

```
dlnet = dlnetwork(lgraph);
```

Define Model Gradients Function

Create the helper function `modelGradients`, listed at the end of the example. The function takes in a `dlnetwork` object `dlnet and` a mini-batch of input data `dlX` with corresponding labels `Y,` and returns the gradients of the loss with respect to the learnable parameters in `dlnet`. This function also returns the loss and the state of the nonlearnable parameters of the network at a given iteration.

Specify Training Options

Specify the training options. Train for 200 epochs with a mini-batch size of 64.

```
numEpochs = 200;
miniBatchSize = 64;

numObservations = numel(YTrain);

velocity = [];
momentum = 0.9;
weightDecay = 1e-4;
```

Specify the training options specific to the cyclical learning rate. `Alpha0` is the initial learning rate and `numSnapshots` is the number of cycles or snapshots taken during training.

```
alpha0 = 0.1;
numSnapshots = 5;
epochsPerSnapshot = numEpochs./numSnapshots;
iterationsPerSnapshot = ceil(numObservations./miniBatchSize)*numEpochs./numSnapshots;
modelPrefix = "SnapshotEpoch";
```

Visualize the training progress in a plot.

```
plots = "training-progress";
```

Initialize the training figure.

```
if plots == "training-progress"
    [lossLine,learnRateLine] = plotLossAndLearnRate();
end
```

Train Model

Use `minibatchqueue` to process and manage mini-batches of images during training. For each mini-batch:

- Use the custom mini-batch preprocessing function `preprocessMiniBatch` (defined at the end of this example) to one-hot encode the class labels.
- Format the image data with the dimension labels `'SSCB'` (spatial, spatial, channel, batch). By default, the `minibatchqueue` object converts the data to `dlarray` objects with underlying type `single`. Do not add a format to the class labels.
- Train on a GPU if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
augimdsTrain.MiniBatchSize = miniBatchSize;

mbqTrain = minibatchqueue(augimdsTrain,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

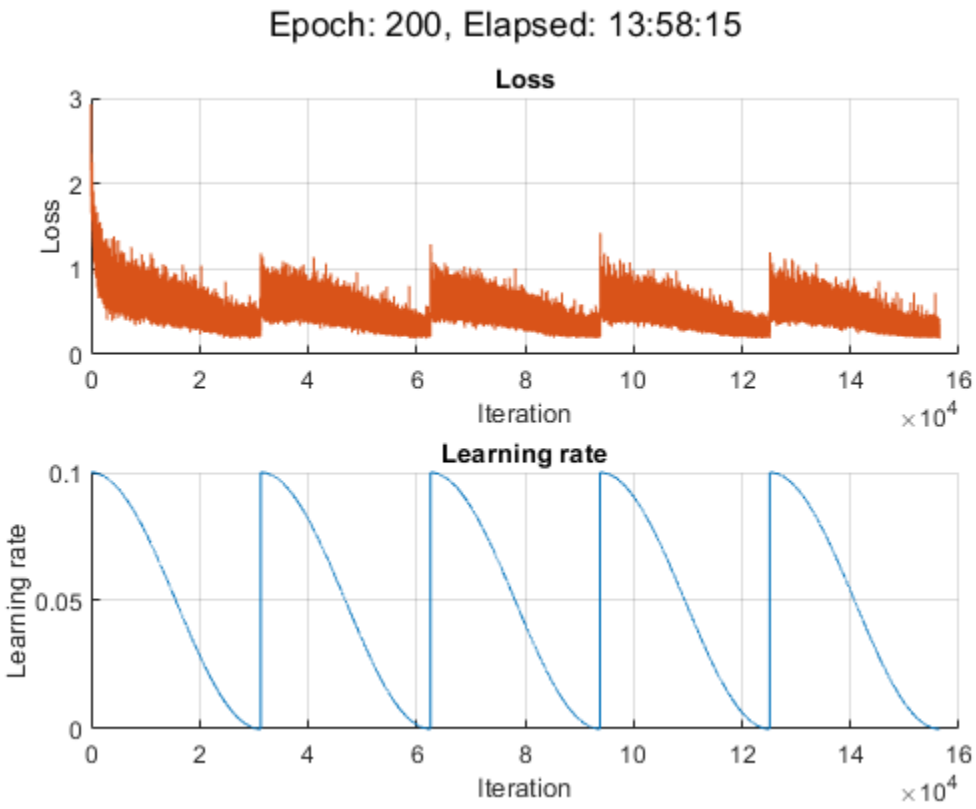
Train the model using a custom training loop.For each epoch, shuffle the datastore, loop over mini-batches of data, and save the model (snapshot) if the current epoch is a multiple of `epochsPerSnapshot`. At the end of each epoch, display the training progress. For each mini-batch:

- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the state of the nonlearnable parameters of the network.
- Determine the learning rate for the cyclical learning rate schedule.
- Update the network parameters using the `sgdmupdate` function.
- Plot the loss and learning rate at each iteration.

For this example, the training took approximately 14 hours on a NVIDIA™ TITAN RTX.

```
iteration = 0;
start = tic;
```

```
% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    shuffle(mbqTrain);

    % Save snapshot model.
    if ~mod(epoch,epochsPerSnapshot)
        save(modelPrefix + epoch + ".mat",'dlnet');
    end

    % Loop over mini-batches.
    while hasdata(mbqTrain)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [dlX,dlY] = next(mbqTrain);

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients, loss, state] = dlfeval(@modelGradients,dlnet,dlX,dlY,weightDecay);

        % Update the state of nonlearnable parameters.
        dlnet.State = state;

        % Determine learning rate for cyclical learning rate schedule.
        learnRate = 0.5*alpha0*(cos((pi*mod(iteration-1,iterationsPerSnapshot)./iterationsPerSnap

        % Update the network parameters using the SGDM optimizer.
        [dlnet.Learnables, velocity] = sgdmupdate(dlnet.Learnables, gradients, velocity, learnRa

        % Display the training progress.
        if plots == "training-progress"
                D = duration(0,0,toc(start),'Format','hh:mm:ss');
                addpoints(lossLine,iteration,double(gather(extractdata(loss))))
                addpoints(learnRateLine, iteration, learnRate);
                sgtitle("Epoch: " + epoch + ", Elapsed: " + string(D))
                drawnow
        end

    end

end
```



Create Snapshot Ensemble and Test Model

Combine the M snapshots of the network taken during training to form a final ensemble and test the classification accuracy of the model. The ensemble predictions correspond to the average of the output of the fully connected layer from all M individual models.

Test the model on the test data provided with the CIFAR-10 data set. Manage the test data set using a `minibatchqueue` object with the same setting as the training data.

```
augimdsTest = augmentedImageDatastore(imageSize,XTest,YTest);
augimdsTest.MiniBatchSize = miniBatchSize;

mbqTest = minibatchqueue(augimdsTest,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFcn', @preprocessMiniBatch,...
    'MiniBatchFormat',{'SSCB',''});
```

Evaluate the accuracy of each snapshot network. Use the `modelPredictions` function defined at the end of this example to iterate over all the data in the test data set. The function returns the output of the fully connected layer from the model, the predicted classes, and the comparison with the true class.

```
modelName = cell(numSnapshots+1,1);
fcOutput = zeros(numClasses,numel(YTest),numSnapshots+1);
classPredictions = cell(1,numSnapshots+1);
modelAccuracy = zeros(numSnapshots+1,1);
```

```
for m = 1:numSnapshots
    modelName{m} = modelPrefix + m*epochsPerSnapshot;
    load(modelName{m} + ".mat");

    reset(mbqTest);
    [fcOutputTest,classPredTest,classCorrTest] = modelPredictions(dlnet,mbqTest,classes);

    fcOutput(:,:,m) = fcOutputTest;
    classPredictions{m} = classPredTest;
    modelAccuracy(m) = 100*mean(classCorrTest);

    disp(modelName{m} + " accuracy: " + modelAccuracy(m) + "%")
end
```

```
SnapshotEpoch40 accuracy: 88.35%
SnapshotEpoch80 accuracy: 89.93%
SnapshotEpoch120 accuracy: 90.51%
SnapshotEpoch160 accuracy: 90.33%
SnapshotEpoch200 accuracy: 90.63%
```

To determine the output of the ensemble networks, compute the average of the fully connected output of each snapshot network. Find the predicted classes from the ensemble network using the `onehotdecode` function. Compare with the true classes to evaluate the accuracy of the ensemble.

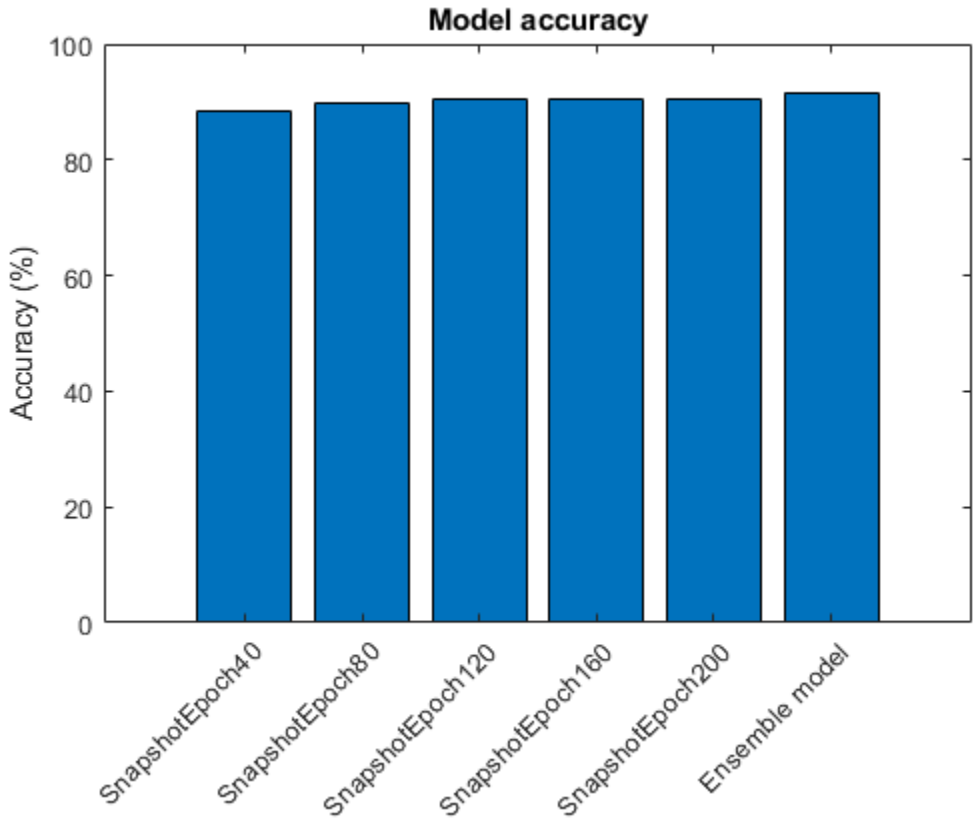
```
fcOutput(:,:,end) = mean(fcOutput(:,:,1:end-1),3);
classPredictions{end} = onehotdecode(softmax(fcOutput(:,:,end)),classes,1,'categorical');

classCorrEnsemble = classPredictions{end} == YTest';
modelAccuracy(end) = 100*mean(classCorrEnsemble);

modelName{end} = "Ensemble model";
disp("Ensemble accuracy: " + modelAccuracy(end) + "%")
```

```
Ensemble accuracy: 91.59%
```

Plot Accuracy

Plot the accuracy on the test data set for all snapshot models and the ensemble model.

```
figure;bar(modelAccuracy);
ylabel('Accuracy (%)');
xticklabels(modelName)
xtickangle(45)
title('Model accuracy')
```



Helper Functions

Model Gradients Function

The `modelGradients` function takes in a `dlnetwork` object `dlnet`, a mini-batch of input data `dlX`, the labels `Y`, and the parameter for weight decay. The function returns the gradients, the loss, and the state of the nonlearnable parameters. To compute the gradients automatically, use the `dlgradient` function.

```
function [gradients,loss,state] = modelGradients(dlnet,dlX,Y,weightDecay)

    [dlYPred,state] = forward(dlnet,dlX);
    dlYPred = softmax(dlYPred);

    loss = crossentropy(dlYPred, Y);

    % L2-regularization (weight decay)
    allParams = dlnet.Learnables(dlnet.Learnables.Parameter == "Weights" | dlnet.Learnables.Paran
    l2Norm = cellfun(@(x) sum(x.^2,'All'),allParams,'UniformOutput',false);
    l2Norm = sum(cat(1,l2Norm{:}));
    loss = loss + weightDecay*0.5*l2Norm;

    gradients = dlgradient(loss,dlnet.Learnables);
end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and computes the model predictions by iterating over all data in the `minibatchqueue`. The function uses the `onehotdecode` function to find the predicted class with the highest score and then compares the prediction with the true class. The function returns the network output, the class predictions, and a vector of ones and zeros that represents correct and incorrect predictions.

```
function [rawPredictions,classPredictions,classCorr] = modelPredictions(dlnet,mbq,classes)
    rawPredictions = [];
    classPredictions = [];
    classCorr = [];

    while hasdata(mbq)
        [dlX,dlY] = next(mbq);

        % Make predictions
        dlYPred = predict(dlnet,dlX);
        rawPredictions = [rawPredictions extractdata(gather(dlYPred))];

        % Convert network output to probabilities and determine predicted
        % classes
        dlYPred = softmax(dlYPred);
        YPredBatch = onehotdecode(dlYPred,classes,1);
        classPredictions = [classPredictions YPredBatch];

        % Compare predicted and true classes
        Y = onehotdecode(dlY,classes,1);
        classCorr = [classCorr YPredBatch == Y];
    end

end
```

Plot Loss and Learning Rate Function

The `plotLossAndLearnRate` function initiliaizes the plots for displaying the loss and learning rate at each iteration during training.

```
function [lossLine, learnRateLine] = plotLossAndLearnRate()
    figure

    subplot(2,1,1);
    lossLine = animatedline('Color',[0.85 0.325 0.098]);
    title('Loss');
    xlabel('Iteration')
    ylabel('Loss')
    grid on

    subplot(2,1,2);
    learnRateLine = animatedline('Color',[0 0.447 0.741]);
    title('Learning rate');
    xlabel('Iteration')
    ylabel('Learning rate')
    grid on
end
```

Convolutional Unit Function

The `convolutionalUnit(numF,stride,tag)` function creates an array of layers with two convolutional layers and corresponding batch normalization and ReLU layers. `numF` is the number of convolutional filters, `stride` is the stride of the first convolutional layer, and `tag` is a tag that is prepended to all layer names.

```
function layers = convolutionalUnit(numF,stride,tag)
    layers = [
        convolution2dLayer(3,numF,'Padding','same','Stride',stride,'Name',[tag,'conv1'])
        batchNormalizationLayer('Name',[tag,'BN1'])
        reluLayer('Name',[tag,'relu1'])
        convolution2dLayer(3,numF,'Padding','same','Name',[tag,'conv2'])
        batchNormalizationLayer('Name',[tag,'BN2'])];
end
```

Data Preprocessing Function

The `preprocessMiniBatch` function preprocesses the data using the following steps:

1. Extract the image data from the incoming cell array and concatenate into a numeric array. Concatenating the image data over the fourth dimension adds a third dimension to each image, to be used as a singleton channel dimension.
2. Extract the label data from the incoming cell arrays and concatenate into a categorical array along the second dimension.
3. One-hot encode the categorical labels into numeric arrays. Encoding into the first dimension produces an encoded array that matches the shape of the network output.

```
function [X,Y] = preprocessMiniBatch(XCell,YCell)
    % Extract image data from cell and concatenate
    X = cat(4,XCell{:});
    % Extract label data from cell and concatenate
    Y = cat(2,YCell{:});

    % One-hot encode labels
    Y = onehotencode(Y,1);
end
```

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

[2] Krizhevsky, Alex. "Learning multiple layers of features from tiny images." (2009). https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[3] Loshchilov, Ilya, and Frank Hutter. "Sgdr: Stochastic gradient descent with warm restarts." (2016). *arXiv preprint arXiv:1608.03983*.

[4] Huang, Gao, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. "Snapshot ensembles: Train 1, get m for free." (2017). *arXiv preprint arXiv:1704.00109*.

See Also

`dlnetwork` | `layerGraph` | `dlarray` | `sgdmupdate` | `dlfeval` | `dlgradient` | `sigmoid` | `minibatchqueue` | `onehotencode` | `onehotdecode`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-76
- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Make Predictions Using Model Function” on page 18-286
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Automatic Differentiation Background” on page 18-200

Deploy Imported Network with MATLAB Compiler

This topic shows how to import a pretrained network and then deploy the imported network using MATLAB Compiler. You can import a pretrained TensorFlow-Keras or ONNX (Open Neural Network Exchange) network using `importKerasNetwork` or `importONNXNetwork`, respectively. These functions require the corresponding support package: Deep Learning Toolbox Converter for TensorFlow Models or Deep Learning Toolbox Converter for ONNX Model Format. If the required support package is not installed, then `importKerasNetwork` or `importONNXNetwork` provides a download link.

The imported network might include Keras or ONNX layers that MATLAB Coder does not support for deployment. For a list of supported layers, see “Networks and Layers Supported for Code Generation” (MATLAB Coder). In this case, you can deploy the imported network as a standalone application using MATLAB Compiler. The standalone executable you create with MATLAB Compiler is independent of MATLAB; therefore, you can deploy it to users who do not have access to MATLAB.

In the deployment workflow, you first define a classification function that loads the imported network and predicts class labels. Then, you compile the classification function into a standalone application either programmatically, using the `mcc` function, or interactively, using the **Application Compiler** app.

- Use `mcc` if you prefer to work at the command line. You must manually specify the path to the Keras or ONNX layers folder that MATLAB Compiler includes in the standalone application. For an example of deploying an imported Keras network, see “Deploy Imported Pretrained Network Using mcc” on page 18-406. You can use the same workflow to deploy a network imported from ONNX using `mcc`.
- Use the **Application Compiler** app if you prefer an interactive workflow. You can access the app using the `deploytool` function or the apps gallery. The app suggests the support packages that MATLAB Compiler can include in the standalone application. Then, the app automatically includes the paths to the Keras or ONNX layers folder in the selected support package. For an example of deploying an imported Keras network, see “Deploy Imported Pretrained Network Using Application Compiler App” on page 18-409. You can use the same workflow to deploy a network imported from ONNX using the **Application Compiler** app.

Deploy Imported Pretrained Network Using mcc

Import a pretrained Keras network to classify an image, and then compile the classification function into a standalone application using `mcc`. This example uses a helper function that imports the network with `importKerasNetwork`, specifies the class names, and saves the imported network. To view the code for this function, see Helper Function on page 18-0 .

Download Required Support Package

The function `importKerasNetwork` requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, `importKerasNetwork` provides a download link to the required support package in the Add-On Explorer. A recommended practice is to download the support package to the default location for the version of MATLAB you are running. However, you can specify a different location during installation.

Display the support package root and the release number for the version of MATLAB you are running. You need this information when specifying the path to the Keras layers folder later in the example. The support package is located in the default location for MATLAB R2020b.

```
supportPKGFolder = matlabshared.supportpkg.getSupportPackageRoot
```

```
supportPKGFolder =
'C:\ProgramData\MATLAB\SupportPackages\R2020b'
```

```
version('-release')
```

```
ans =
'2020b'
```

Import Pretrained Network

Import and save the pretrained network `digitsDAGnet`, which contains a DAG (directed acyclic graph) convolutional neural network that classifies images of digits.

```
net = importDAGnet
```

```
net =
  DAGNetwork with properties:

         Layers: [13×1 nnet.cnn.layer.Layer]
    Connections: [13×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

Read and Save Image

Read and save the image to classify.

```
digitDatasetPath = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');
I = imread(fullfile(digitDatasetPath,'5','image4009.png'));
imwrite(I,'testImg.png')
```

Display the image.

```
imshow(I)
```

Define Classification Function

Define a classification function named `KerasNetClassify` that accepts a digit image, loads the imported Keras network, and predicts the class label using the loaded network.

```
type KerasNetClassify.m
```

```
function KerasNetClassify(imFile)
% KERASNETCLASSIFY Classify image using imported network
% KERASNETCLASSIFY loads the imported Keras pretrained network
% 'digitsDAGnet.mat', reads the image in imFile, and predicts the image
```

```
% label using the imported network.
load('digitsDAGnet.mat','net');
I = imread(imFile);
label = classify(net, I);
disp(label)
end
```

Create Executable File

To deploy the imported network using `mcc`, you must manually specify the path to the Keras layers folder. The layers folder is located in the support package folder. Display the path to the Keras layers folder.

```
fullfile(supportPKGFolder,'\toolbox\nnet\supportpackages\keras_importer\+nnet\+keras\+layer')
```

```
ans =
'C:\ProgramData\MATLAB\SupportPackages\R2020b\toolbox\nnet\supportpackages\keras_importer\+nnet\+
```

Compile the classification function into the standalone executable `KerasNetClassify.exe` by using the `mcc` function. Specify the path to the Keras layers folder that MATLAB Compiler includes in the standalone application by using `-a path`.

```
mcc -m KerasNetClassify.m...
    -a 'C:\ProgramData\MATLAB\SupportPackages\R2020b\toolbox\nnet\supportpackages\keras_importer\
    -n
```

Classify Image

Compare the labels classified using `classify`, `KerasNetClassify.m`, and `KerasNetClassify.exe`.

```
classify(net,I)
```

```
ans = categorical
     5
```

```
KerasNetClassify('testImg.png')
```

```
     5
```

```
!KerasNetClassify.exe 'testImg.png'
```

```
     5
```

All three ways to classify the image return the same label.

Helper Function

This section provides the code of the `importDAGnet` helper function. The `importDAGnet` function imports the pretrained network in the file `digitsDAGnet.h5`, specifies the class names, and saves the imported network to `digitsDAGnet.mat`.

```
function net = importDAGnet

    % Specify the model file.
    modelfile = 'digitsDAGnet.h5';
```

```
    % Specify the class names.
    classNames = {'0','1','2','3','4','5','6','7','8','9'};

    % Import the Keras network with the class names.
    net = importKerasNetwork(modelfile,'Classes',classNames);

    % Save the imported network to a MAT file.
    save('digitsDAGnet.mat', 'net');

end
```

Deploy Imported Pretrained Network Using Application Compiler App

Import the pretrained Keras network `digitsDAGnet` to classify an image, and then compile the classification function into a standalone application using the **Application Compiler** app.

Import Pretrained Network

Use `importKerasNetwork` to import the `digitsDAGnet` network, and then save it to a MAT file. The function `importKerasNetwork` requires the Deep Learning Toolbox Converter for TensorFlow Models support package. If this support package is not installed, `importKerasNetwork` provides a download link to the required support package in the Add-On Explorer. (For more details on how to import the pretrained network and save the image to classify, see “Deploy Imported Pretrained Network Using mcc” on page 18-406.)

Define Classification Function

Define a classification function named `KerasNetClassify` that accepts a digit image, loads the imported Keras network, and predicts the class label using the loaded network.

```
function KerasNetClassify(imFile)
% KERASNETCLASSIFY Classify image using imported network
% KERASNETCLASSIFY loads the imported Keras pretrained network
% 'digitsDAGnet.mat', reads the image in imFile, and predicts the image
% label using the imported network.
load('digitsDAGnet.mat','net');
I = imread(imFile);
label = classify(net, I);
disp(label)
end
```

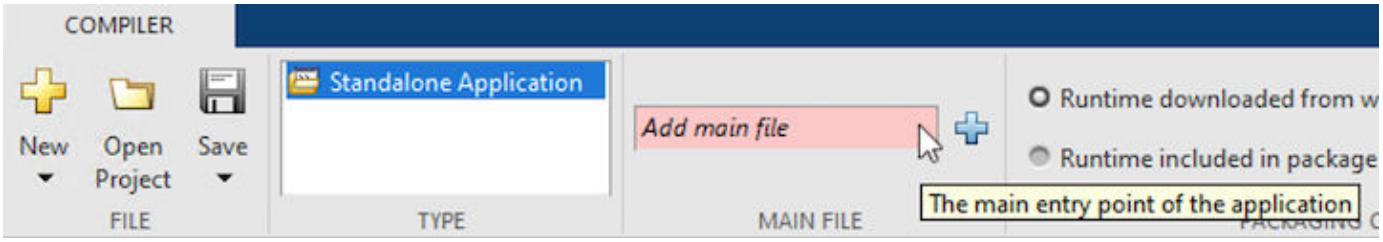
Create Executable File

Open a list of application deployment apps by using the `deploytool` function.

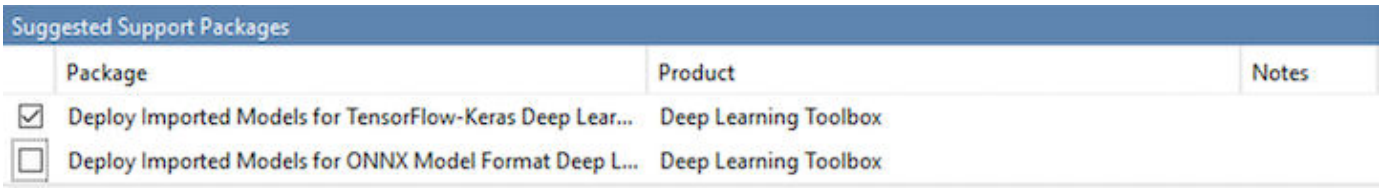
```
deploytool
```

In the **MATLAB Compiler** window, click **Application Compiler**. (You can also open the app by selecting it from the apps gallery, available from the **Apps** tab.)

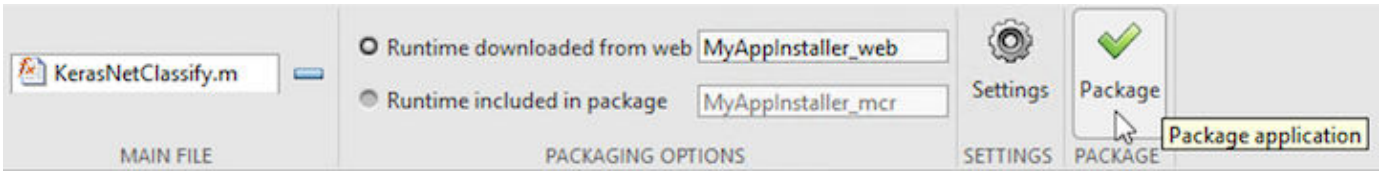
In the **Main File** section of the **Compiler** tab, add the main file of the application by clicking the plus sign. In the **Add Files** dialog box, specify the main file as the classification function `KerasNetClassify.m`.



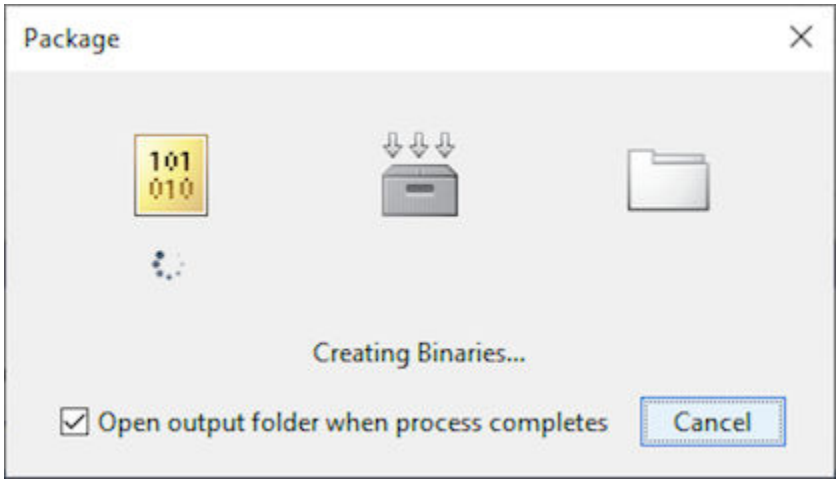
The app suggests software support packages from the installed support packages, which the executable can include. Because you have installed the Deep Learning Toolbox Converter for TensorFlow Models and Deep Learning Toolbox Converter for ONNX Model Format support packages, the app displays both. You must select the Deep Learning Toolbox Converter for TensorFlow Models support package. Selecting the Deep Learning Toolbox Converter for ONNX Model Format support package does not influence the execution of the application, but unnecessarily increases the application footprint.



In the **Package** section, click **Package** to save the standalone application.



The software compiles the standalone application. The default name for the output folder is `KerasNetClassify`, and the executable file `KerasNetClassify.exe` is located in the subfolder `for_redistribution_files_only`.



Classify Image

Copy the image file `testImg.png` (image of digit 5) to the folder containing the executable file. Change the current folder to the folder containing the executable file.

```
copyfile('testImg.png','KerasNetClassify\for_redistribution_files_only')
cd('KerasNetClassify\for_redistribution_files_only')
```

Run the executable file `KerasNetClassify.exe`, which you created with the **Application Compiler** app, to classify the image `testImg.png`.

```
!KerasNetClassify.exe testImg.png
```

```
     5
```

The classification label returned by `KerasNetClassify.exe` is correct.

See Also

`importKerasNetwork` | `importONNXNetwork` | `mcc` | `deploytool` | `importKerasLayers` | `importONNXLayers`

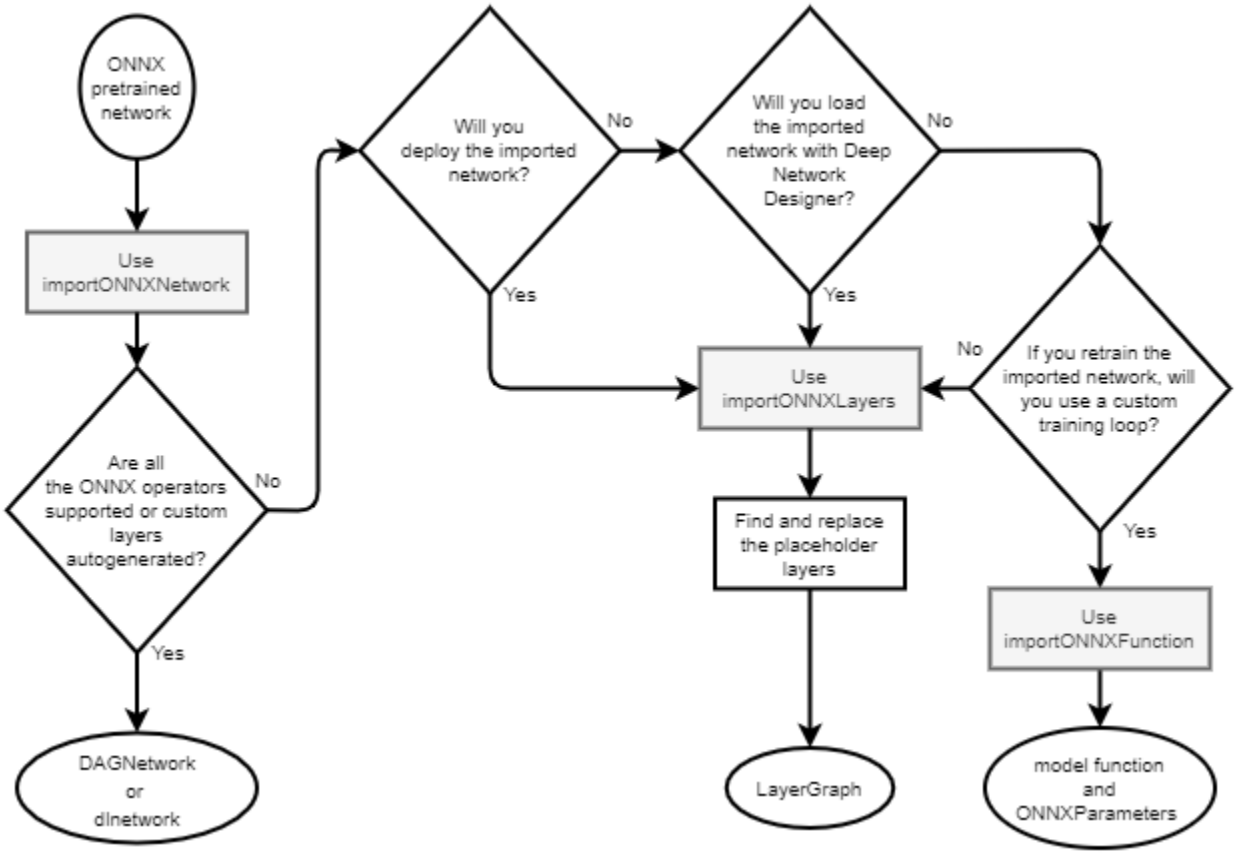
Related Examples

- “Pretrained Deep Neural Networks” on page 1-8
- “Load Pretrained Networks for Code Generation” (MATLAB Coder)
- “Networks and Layers Supported for Code Generation” (MATLAB Coder)
- “Create Standalone Application Using Application Compiler App” (MATLAB Compiler)

Select Function to Import ONNX Pretrained Network

Deep Learning Toolbox Converter for ONNX Model Format provides three functions to import a pretrained ONNX (Open Neural Network Exchange) network: `importONNXNetwork`, `importONNXLayers`, and `importONNXFunction`.

This flow chart illustrates which import function best suits different scenarios.



Note By default, `importONNXNetwork` and `importONNXLayers` try to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. For a list of operators for which the software supports conversion, see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers”.

`importONNXNetwork` and `importONNXLayers` save the generated custom layers in the package `+PackageName` in the current folder.

`importONNXNetwork` and `importONNXLayers` do not automatically generate a custom layer for each ONNX operator that is not supported for conversion into a built-in MATLAB layer.

Decisions

This table describes each decision in the workflow for selecting an ONNX import function.

Decision	Description
Are all the ONNX operators supported for conversion into equivalent built-in MATLAB layers or can the software automatically generate custom layers?	• If the imported network contains an ONNX operator not supported for conversion into a built-in MATLAB layer (see “ONNX Operators Supported for Conversion into Built-In MATLAB Layers”) and `importONNXNetwork` does not generate a custom layer, then `importONNXNetwork` returns an error. • If the imported network contains an ONNX operator not supported for conversion into a built-in MATLAB layer and `importONNXLayers` does not generate a custom layer, then `importONNXLayers` inserts a placeholder layer in place of the unsupported layer. • `importONNXFunction` supports most ONNX operators. For more information, see “ONNX Operators That importONNXFunction Supports”.
Will you deploy the imported network?	If you use `importONNXNetwork` or `importONNXLayers`, you can generate code for the imported network. To create a `DAGNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).
Will you load the imported network with Deep Network Designer?	If you use `importONNXNetwork` or `importONNXLayers`, you can load the imported network with the **Deep Network Designer** app.
If you retrain the imported network, will you use a custom training loop?	• If you use `importONNXFunction`, you can retrain the imported network only with a custom training loop. For an example, see “Train Imported ONNX Function Using Custom Training Loop”. • Use `importONNXNetwork` with `TargetNetwork` specified as `"dlnetwork"` to import the network as a `dlnetwork` object. A `dlnetwork` enables support for custom training loops using automatic differentiation. • Use `importONNXLayers` with `TargetNetwork` specified as `"dlnetwork"` to import the network as a `LayerGraph` object compatible with a `dlnetwork` object. Then convert the layer graph to a `dlnetwork` by using `dlnetwork`. • For more information about training options, see “Train Deep Learning Model in MATLAB” on page 18-3.

Actions

This table describes each action in the workflow for selecting an ONNX import function.

Action	Description
Use `importONNXNetwork`	`importONNXNetwork` returns a `DAGNetwork` or `dlnetwork` object that is ready to use for prediction (for more information, see the `TargetNetwork` name-value argument). Predict class labels by using the `classify` function on the `DAGNetwork` object or the `predict` function on the `dlnetwork` object.
Use `importONNXLayers`	`importONNXLayers` returns a `LayerGraph` object compatible with a `DAGNetwork` or `dlnetwork` object (for more information, see the `TargetNetwork` name-value argument). `importONNXLayers` inserts placeholder layers in the place of unsupported layers. Find and replace the placeholder layers. Then, you can assemble the layer graph by using `assembleNetwork`, which returns a `DAGNetwork` object, or convert the layer graph to a `dlnetwork` object by using `dlnetwork`.
Use `importONNXFunction`	`importONNXFunction` returns an `ONNXParameters` object, which contains the network parameters, and a model function (see “Imported ONNX Model Function”), which contains the network architecture. The `ONNXParameters` object and the model function are ready to use for prediction. For an example, see “Predict Using Imported ONNX Function”.
Find and replace the placeholder layers	To find the names and indices of the placeholder layers in the imported network, use the `findPlaceholderLayers` function. You then can replace a placeholder layer with a new layer that you define. To replace a layer, use `replaceLayer`.

See Also

`importONNXNetwork` | `importONNXLayers` | `importONNXFunction` | `DAGNetwork` | `dlnetwork` | `layerGraph` | `ONNXParameters`

More About

- “Pretrained Deep Neural Networks” on page 1-8
- “Train Deep Learning Model in MATLAB” on page 18-3
- “Assemble Network from Pretrained Keras Layers” on page 18-187
- “Train Network Using Custom Training Loop” on page 18-225

- “Define Custom Deep Learning Layers” on page 18-9
- “Load Pretrained Networks for Code Generation” (MATLAB Coder)

Classify Sequence of Images in Simulink with Imported TensorFlow Network

This example shows how to import a pretrained TensorFlow™ network in the saved model format by using `importTensorFlowNetwork`, and then use the Predict block to classify a sequence of images in Simulink®. The imported network contains layers that are not supported for conversion into built-in MATLAB® layers. `importTensorFlowNetwork` automatically generates custom layers when you import these layers. The Predict block predicts responses for the data at the input by using the trained network that you specify using the block parameters.

`importTensorFlowNetwork` requires the Deep Learning Toolbox™ Converter for TensorFlow Models support package. If this support package is not installed, then `importTensorFlowNetwork` provides a download link.

Load Image Data

Load the digit sample data as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object.

```
digitDatasetPath = fullfile(matlabroot,"toolbox","nnet","nndemos", ...
    "nndatasets","DigitDataset");
imds = imageDatastore(digitDatasetPath, ...
    IncludeSubfolders=true,LabelSource="foldernames");
```

For reproducibility, specify the seed for the MATLAB random number generator. Randomly select eight images from the image datastore, create the array of images `inputIms`, and display the selected images by using `montage` (Image Processing Toolbox).

```
rng("default")
perm = randperm(10000,8);
for i = 1:8
    inputIms(:,:,:,i) = imread(imds.Files{perm(i)});
end
montage(inputIms,size=[1 NaN]);
```

Import Pretrained TensorFlow Network

Specify the model folder that contains the pretrained network `digitsDAGnetwithnoise` in the saved model format. `digitsDAGnetwithnoise` can classify images of digits.

```
if ~exist("digitsDAGnetwithnoise","dir")
    unzip("digitsDAGnetwithnoise.zip")
end
modelFolder = "./digitsDAGnetwithnoise";
```

Specify the class names.

```
classNames = {'0','1','2','3','4','5','6','7','8','9'};
```

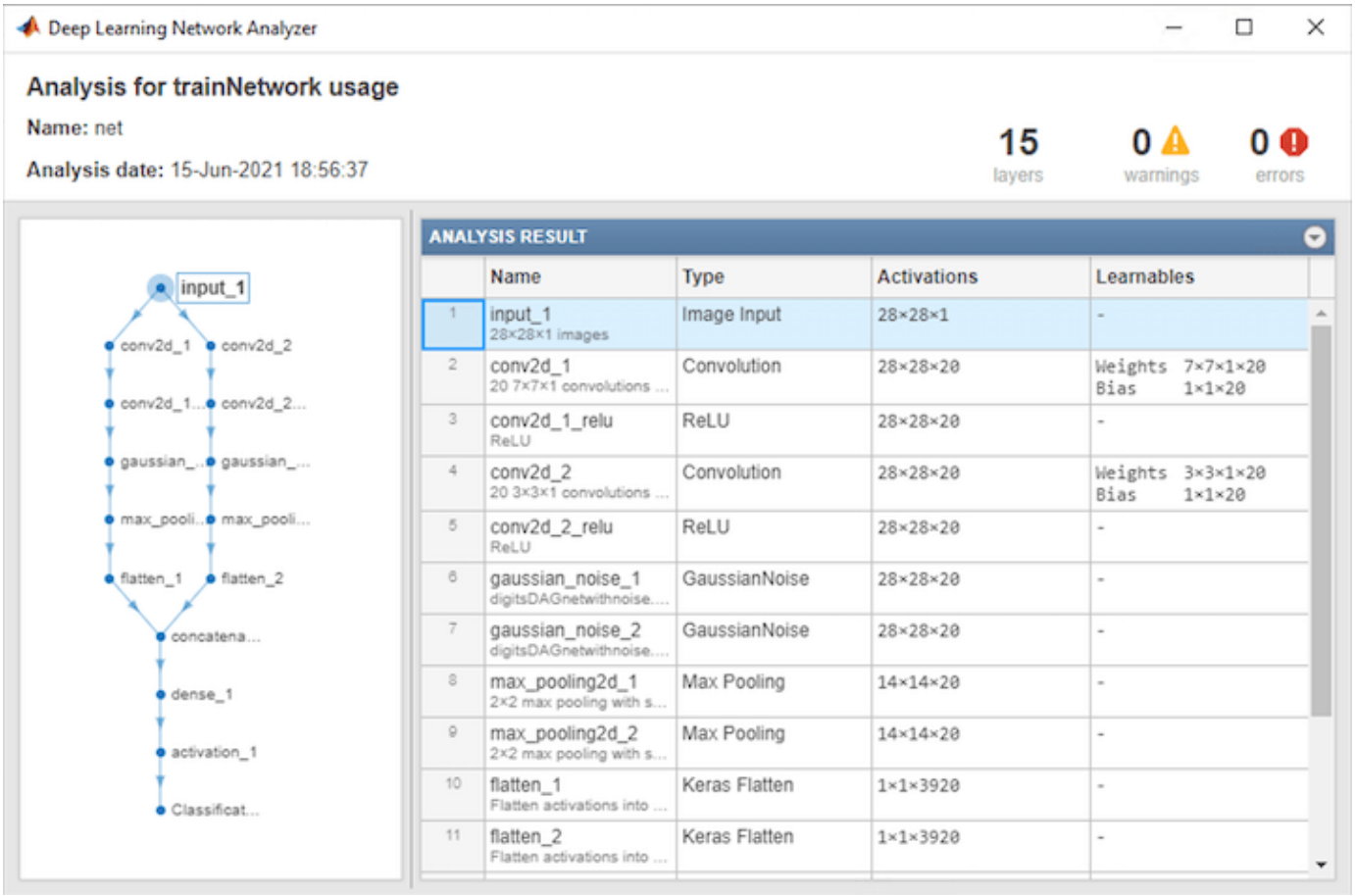
Import a TensorFlow network in the saved model format. By default, `importTensorFlowNetwork` imports the network as a `DAGNetwork` object.

```
net = importTensorFlowNetwork(modelFolder,Classes=classNames);
```

```
Importing the saved model...
Translating the model, this may take a few minutes...
Finished translation. Assembling network...
Import finished.
```

Analyze the imported network. `analyzeNetwork` displays an interactive plot of the network architecture and a table containing information about the network layers.

```
analyzeNetwork(net)
```



The imported network contains layers that are not supported for conversion into built-in MATLAB layers. The software automatically generates the custom layers `gaussian_noise_1` and `gaussian_noise_2`. The function `importTensorFlowNetwork` saves each generated custom layer to a separate `.m` file in the package `+digitsDAGnetwithnoise` in the current folder. For more information on these generated custom layers, see “Import TensorFlow Network with Autogenerated Custom Layers”.

Save the imported network in a MAT file.

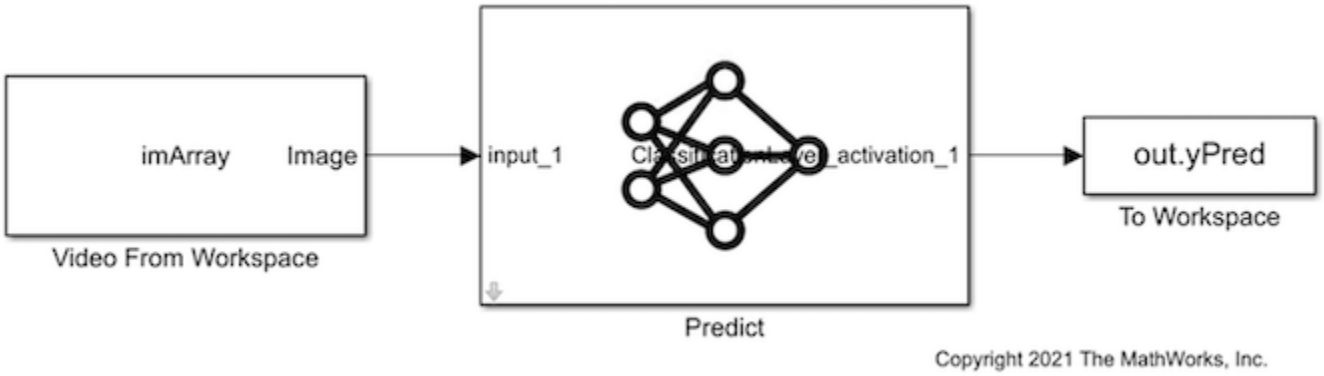
```
filename = "digitsNet.mat";
save(filename,"net")
```

Create Simulink Model

This example provides the Simulink model `slexDigitsImportedNetworkPredictExample.slx`. You can open the Simulink model (provided in this example) or create a new model by following the steps described in this section.

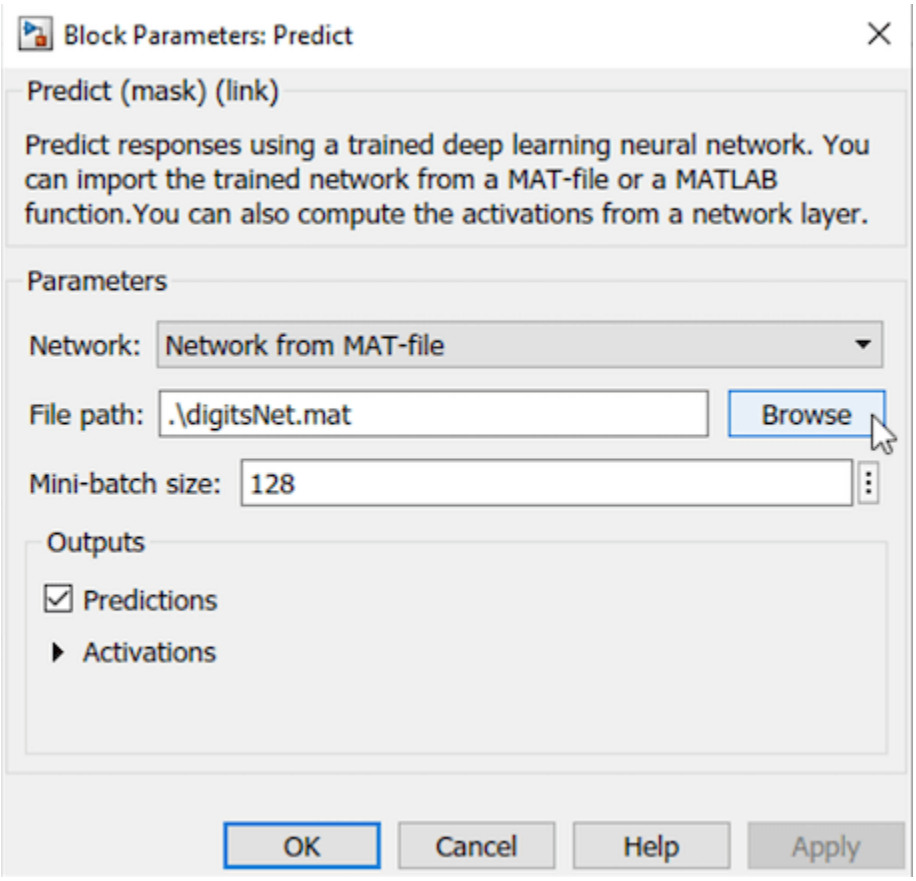
Open the Simulink model `slexDigitsImportedNetworkPredictExample.slx`.

```
SimMdlName = "slexDigitsImportedNetworkPredictExample";
open_system(SimMdlName)
```

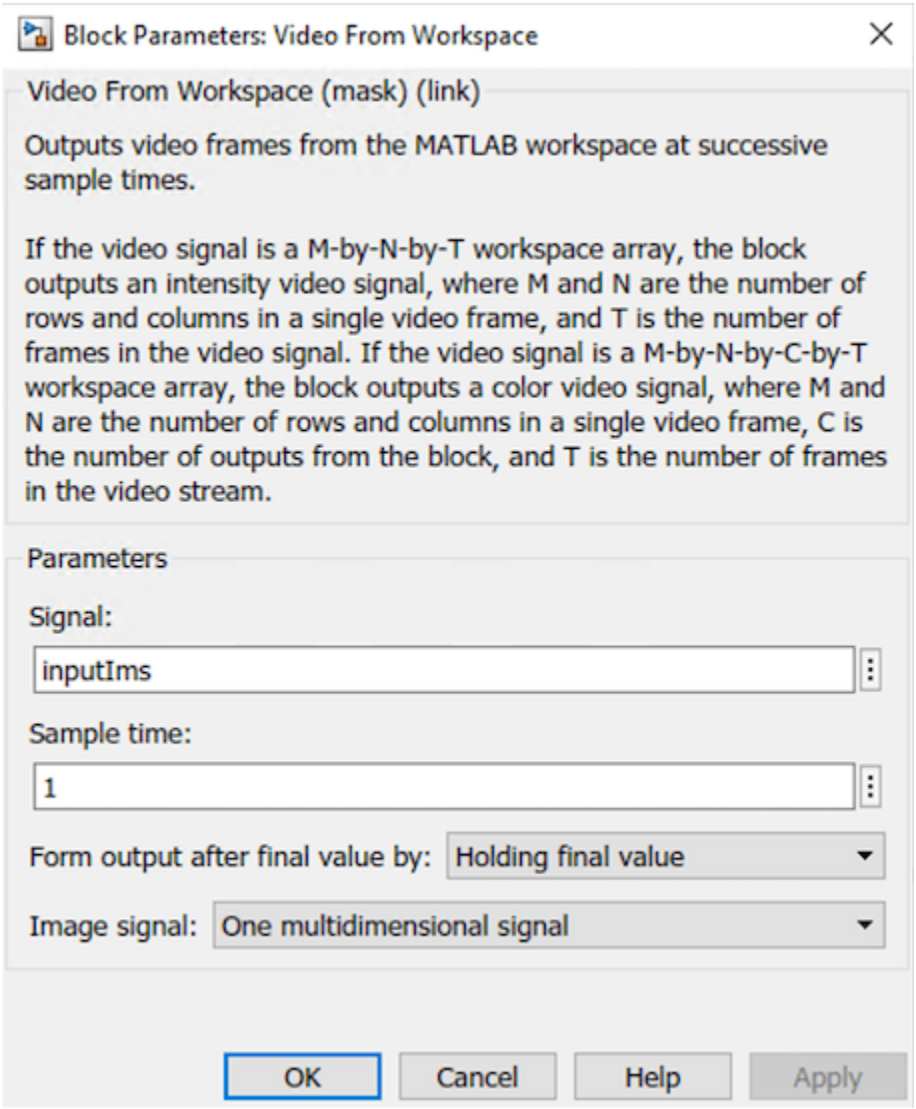


1. To create a new Simulink model, open the **Blank Model** template and add the Predict block from the Deep Learning Toolbox™ library. The Predict block predicts responses for the data at the input by using the trained network that you specify using the block parameters. The input to the block can be an *h*-by-*w*-by-*c*-by-*N* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the images, respectively, and *N* is the number of images.

Double-click the Predict block to open the Block Parameters dialog box. Select **`Network from MAT-file`** for the **Network** parameter. Click **Browse** in the **File Path** section to specify the network as the `digitsNet.mat` network in the current folder.



2. Insert the Video from Workspace block from the Computer Vision Toolbox™ library. Double-click the Video from Workspace block to open the Block Parameters dialog box. Specify **Signal** as `inputIms`, **Sample time** as 1, and **Form output after final value by** as **Holding final value**.



3. Check if the size of the input images matches the network input size. If they do not match, you must resize the input data by adding the Resize block from the Computer Vision Toolbox library to the model.

Display the size of the image and the input size of the network.

```
size(inputIms)
```

```
ans = 1×4

    28    28     1     8
```

```
netInputSize = net.Layers(1).InputSize
```

```
netInputSize = 1×3

    28    28     1
```

The input is a sequence of eight grayscale (one-channel) images that are 28-by-28 pixels in size. The image size matches the network input size.

4. Add a To Workspace block to the model and change the variable name to `yPred`. Connect the Video from Workspace block to the input of the Predict block and the To Workspace block to the output of the Predict block.

5. Open the Configuration Parameters dialog box. On the **Modeling** tab, click **Model Settings**. Under **Solver selection**, set **Type** to `Fixed-step`, and set **Solver** to `discrete (no continuous states)`.

Predict Using Simulink Model

Simulate the model and save the simulation output to `modelOutput`. The field `modelOutput.yPred.Data` contains the classification results.

```
modelOutput = sim(SimMdlName)
```

```
modelOutput =
  Simulink.SimulationOutput:

                    tout: [8x1 double]
                   yPred: [1x1 timeseries]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```

Display the sequence of images and the classification results.

```
tiledlayout(1,12,TileSpacing="None");
for i = 1:size(inputIms,4)
    nexttile
    imshow(inputIms(:,:,:,i))
    label = modelOutput.yPred.Data(:,:,i)==1;
    title([classNames{label}],FontSize=20)
end
```

See Also

`importTensorFlowNetwork` | `importKerasNetwork` | `importONNXNetwork` | `importCaffeNetwork`

Related Examples

- “GPU Code Generation for Blocks from the Deep Neural Networks Library” (GPU Coder)

- “Deploy Imported Network with MATLAB Compiler” on page 18-406
- “Assemble Network from Pretrained Keras Layers” on page 18-187

List of Functions with dlarray Support

Deep Learning Toolbox Functions with dlarray Support

These tables list and briefly describe the Deep Learning Toolbox functions that operate on `dlarray` objects.

Deep Learning Operations

Function	Description
`avgpool`	The average pooling operation performs downsampling by dividing the input into pooling regions and computing the average value of each region.
`batchnorm`	The batch normalization operation normalizes the input data across all observations for each channel independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use batch normalization between convolution and nonlinear operations such as `relu`.
`crossentropy`	The cross-entropy operation computes the cross-entropy loss between network predictions and target values for single-label and multi-label classification tasks.
`crosschannelnorm`	The cross-channel normalization operation uses local responses in different channels to normalize each activation. Cross-channel normalization typically follows a `relu` operation. Cross-channel normalization is also known as local response normalization.
`ctc`	The CTC operation computes the connectionist temporal classification (CTC) loss between unaligned sequences.
`dlconv`	The convolution operation applies sliding filters to the input data. Use the `dlconv` function for deep learning convolution, grouped convolution, and channel-wise separable convolution.
`dlode45`	The neural ordinary differential equation (ODE) operation returns the solution of a specified ODE.
`dltranspconv`	The transposed convolution operation upsamples feature maps.
`embed`	The embed operation converts numeric indices to numeric vectors, where the indices correspond to discrete data. Use embeddings to map discrete data such as categorical values or words to numeric vectors.

Function	Description
`fullyconnect`	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
`groupnorm`	The group normalization operation normalizes the input data across grouped subsets of channels for each observation independently. To speed up training of the convolutional neural network and reduce the sensitivity to network initialization, use group normalization between convolution and nonlinear operations such as `relu`.
`gru`	The gated recurrent unit (GRU) operation allows a network to learn dependencies between time steps in time series and sequence data.
`huber`	The Huber operation computes the Huber loss between network predictions and target values for regression tasks. When the `'TransitionPoint'` option is 1, this is also known as *smooth L_1 loss*.
`instancenorm`	The instance normalization operation normalizes the input data across each channel for each observation independently. To improve the convergence of training the convolutional neural network and reduce the sensitivity to network hyperparameters, use instance normalization between convolution and nonlinear operations such as `relu`.
`l1loss`	The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean absolute error (MAE).
`l2loss`	The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computed value is known as the mean squared error (MSE).
`layernorm`	The layer normalization operation normalizes the input data across all channels for each observation independently. To speed up training of recurrent and multilayer perceptron neural networks and reduce the sensitivity to network initialization, use layer normalization after the learnable operations, such as LSTM and fully connect operations.

Function	Description
`leakyrelu`	The leaky rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is multiplied by a fixed scale factor.
`lstm`	The long short-term memory (LSTM) operation allows a network to learn long-term dependencies between time steps in time series and sequence data.
`maxpool`	The maximum pooling operation performs downsampling by dividing the input into pooling regions and computing the maximum value of each region.
`maxunpool`	The maximum unpooling operation unpools the output of a maximum pooling operation by upsampling and padding with zeros.
`mse`	The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression tasks.
`onehotdecode`	The one-hot decode operation decodes probability vectors, such as the output of a classification network, into classification labels. The input `A` can be a `dlarray`. If `A` is formatted, the function ignores the data format.
`relu`	The rectified linear unit (ReLU) activation operation performs a nonlinear threshold operation, where any input value less than zero is set to zero.
`sigmoid`	The sigmoid activation operation applies the sigmoid function to the input data.
`softmax`	The softmax activation operation applies the softmax function to the channel dimension of the input data.

dlarray-Specific Functions

Function	Description
`dims`	This function returns the data format of a `dlarray`.
`dlfeval`	This function evaluates a `dlarray` function using automatic differentiation.
`dlgradient`	This function computes gradients using automatic differentiation.
`extractdata`	This function extracts the data from a `dlarray`.

Function	Description
`finddim`	This function finds the indices of `dlarray` dimensions with a given dimension label.
`stripdims`	This function removes the data format from a `dlarray`.

Domain-Specific Functions with dlarray Support

These tables list and briefly describe the domain-specific functions that operate on `dlarray` objects.

Computer Vision

Function	Description
`focalCrossEntropy`	Calculate the focal cross-entropy loss between two `dlarray` objects that represent predicted and target classification labels.
`generalizedDice`	Measure the similarity between two `dlarray` objects that represent segmented images, using a generalized Dice metric that accounts for class weighting.
`roialign`	Perform ROI pooling of `dlarray` data.

Image Processing

Function	Description
`depthToSpace`	Rearrange `dlarray` data from the depth dimension into spatial blocks.
`dlresize`	Resize the spatial dimensions of a `dlarray`.
`multissim`	Measure the similarity between two `dlarray` objects that represent 2-D images, using the multiscale structural similarity (MS-SSIM) metric.
`multissim3`	Measure the similarity between two `dlarray` objects that represent 3-D images, using the 3-D MS-SSIM metric.
`psnr`	Measure the similarity between two `dlarray` objects that represent images using the peak signal-to-noise ratio (PSNR) metric.
`spaceToDepth`	Rearrange spatial blocks of `dlarray` data into the depth dimension.
`ssim`	Measure the similarity between two `dlarray` objects that represent images using the structural similarity (SSIM) metric.

Signal Processing

Function	Description
`dlstft`	Compute short-time Fourier transform.

MATLAB Functions with dlarray Support

Many MATLAB functions operate on `dlarray` objects. These tables list the usage notes and limitations for these functions when you use `dlarray` arguments.

Unary Element-wise Functions

Function	Notes and Limitations
`abs`	The output `dlarray` has the same data format as the input `dlarray`.
`acos`	
`acosh`	
`acot`	
`acsc`	
`angle`	
`asec`	
`asin`	
`asinh`	
`atan`	
`atan2`	
`atanh`	
`conj`	
`cos`	
`cosh`	
`cot`	
`csc`	
`exp`	
`imag`	
`log`	
`real`	
`reallog`	
`realsqrt`	
`sec`	
`sign`	
`sin`	
`sinh`	

Function	Notes and Limitations
sqrt	
tan	
tanh	
uminus, -	
uplus, +	

Binary Element-wise Operators

Function	Notes and Limitations
complex	For the one-input syntax, the output dlarray has the same data format as the input dlarray For the two-input syntax, if dlarray inputs are formatted, their data formats must match.
minus, -	If the two dlarray inputs are formatted, then the output dlarray is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” on page 18-436.
plus, +	
power, .^	
rdivide, ./	
realpow	
times, .*	

Reduction Functions

Function	Notes and Limitations
mean	• The output dlarray has the same data format as the input dlarray. • The 'omitnan' option is not supported. • If the input dlarray is on the GPU, the 'native' option is not supported.
prod	• The output dlarray has the same data format as the input dlarray. • The 'omitnan' option is not supported.
sum	

Extrema Functions

Function	Notes and Limitations
ceil	The output dlarray has the same data format as the input dlarray.
eps	• The output dlarray has the same data format as the input dlarray. • Use eps(ones(‘like’, x)) to get a scalar epsilon value based on the data type of a dlarray x.

Function	Notes and Limitations
`fix`	The output `dlarray` has the same data format as the input `dlarray`.
`floor`	The output `dlarray` has the same data format as the input `dlarray`.
`max` `min`	• When you find the maximum or minimum elements of a single `dlarray`, the output `dlarray` has the same data format as the input `dlarray`. • When you find the maximum or minimum elements between two formatted `dlarray` inputs, the output `dlarray` has a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” on page 18-436. • The index output argument is not traced and cannot be used with automatic differentiation. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox” on page 18-205.
`rescale`	• If the first input `dlarray` `A` is unformatted, all additional inputs must be unformatted. • If the first input `dlarray` `A` is formatted, all additional inputs must either be unformatted scalars, or have data formats that are a subset of the data format of `A`. In this case, each dimension must either be singleton or match the length of the corresponding dimension of `A`.
`round`	• Only the syntax `Y = round(X)` is supported. • The output `dlarray` has the same data format as the input `dlarray`.

Fourier Transforms

Function	Notes and Limitations
`fft`	Only unformatted input arrays are supported.

Function	Notes and Limitations
`ifft`	• Only unformatted input arrays are supported. • When you use the `'symmetric'` option, `ifft` treats the input `Y` as exactly symmetric. If you compute the derivative using automatic differentiation, then the derivative is also exactly symmetric. If `Y` is non-symmetric, then the function and gradient behavior might not match. To ensure that function and gradient behavior match for non-symmetric inputs, explicitly symmetrize `Y`.

Other Math Operations

Function	Notes and Limitations

`colon`, `:`	• The supported operations are:
	  • `a:b`
	  • `a:b:c`
	  For information on indexing into a `dlarray`, see “Indexing” on page 18-438.
	• All inputs must be real scalars. The output `dlarray` is unformatted.

`interp1`	• Sample points input `x` must be a finite, increasing vector without repeating elements. • `method` must be `'linear'` or `'nearest'`. • The piecewise polynomial syntax (`'pp'`) is not supported. • Only the sample values input `v` can be a formatted `dlarray`. All other inputs must be unformatted. If `v` is a formatted `dlarray`, query points input `xq` must be a vector, and the output `vq` has the same data format as `v`.
`mrdivide`, `/`	The second `dlarray` input must be a scalar. The output `dlarray` has the same data format as the first `dlarray` input.
`mtimes`, `*`	• One input can be a formatted `dlarray` only when the other input is an unformatted scalar. In this case, the output `dlarray` has the same data format as the formatted `dlarray` input. • Multiplying a `dlarray` with a non-`dlarray` sparse matrix is supported only when both inputs are non-scalar.

Function	Notes and Limitations
`ode45`	The supported syntaxes are: • `[t,y] = ode45(odefun,tspan,y0)` • `[t,y] = ode45(odefun,tspan,y0,options)` At least one of `y0` and `tspan` must be an unformatted `dlarray` object. If `tspan` is a `dlarray` object, then the output `t` is an unformatted `dlarray` object. If `y0` is a `dlarray` object, then the output `y` is an unformatted `dlarray` object. For `dlarray` input, the `ode45` function does not support the `OutputFcn`, `Mass`, `NonNegative`, and `Events` options. For `dlarray` input, the `ode45` function does not support acceleration using `dlaccelerate`. **Tip** For neural ODE workflows, use `dlode45`.
`pagemtimes`	One input can be a formatted `dlarray` only when the other input is unformatted, with scalar pages. In this case, the output `dlarray` has the same data format as the formatted `dlarray` input.

Logical Operations

Function	Notes and Limitations
`all`	The output `dlarray` has the same data format as the input `dlarray`.
`and`, `&`	If the two `dlarray` inputs are formatted, then the output `dlarray` is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” on page 18-436.
`any`	The output `dlarray` has the same data format as the input `dlarray`.
`eq`, `==`	If the two `dlarray` inputs are formatted, then the output `dlarray` is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” on page 18-436.
`ge`, `>=`	
`gt`, `>`	
`le`, `<=`	
`lt`, `<`	
`ne`, `~=`	

Function	Notes and Limitations
`not`, `~`	The output `dlarray` has the same data format as the input `dlarray`.
`or`, `|`	If the two `dlarray` inputs are formatted, then the output `dlarray` is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” on page 18-436.
`xor`	

Size Manipulation Functions

Function	Notes and Limitations
`reshape`	The output `dlarray` is unformatted, even if the input `dlarray` is formatted.
`squeeze`	Two-dimensional `dlarray` objects are unaffected by `squeeze`. If the input `dlarray` is formatted, the function removes dimension labels belonging to singleton dimensions. If the input `dlarray` has more than two dimensions and its third and above dimensions are singleton, then the function discards these dimensions and their labels.

Transposition Operations

Function	Notes and Limitations
`ctranspose`, `'`	If the input `dlarray` is formatted, then the labels of both dimensions must be the same. The function performs transposition implicitly, and transposes directly only if necessary for other operations.
`permute`	If the input `dlarray` is formatted, then the permutation must be among only those dimensions that have the same label. The function performs permutations implicitly, and permutes directly only if necessary for other operations.
`transpose`, `.'`	If the input `dlarray` is formatted, then the labels of both dimensions must be the same. The function performs transposition implicitly, and transposes directly only if necessary for other operations.

Concatenation Functions

Function	Notes and Limitations
`cat`	The `dlarray` inputs must have matching formats or be unformatted. Mixed formatted and unformatted inputs are supported. If any
`horzcat`	

Function	Notes and Limitations
`vertcat`	`dlarray` inputs are formatted, then the output `dlarray` is formatted with the same data format.

Conversion Functions

Function	Notes and Limitations
`cast`	• `cast(dlA,newdatatype)` copies the data in the `dlarray` `dlA` into a `dlarray` of the underlying data type `newdatatype`. The `newdatatype` option must be `'double'`, `'single'`, or `'logical'`. The output `dlarray` is formatted with the same data format as `dlA`. • `cast(A,'like',Y)` returns an array of the same type as `Y`. If `Y` is a `dlarray`, then the output is a `dlarray` that has the same underlying data type as `Y`. If `Y` is on the GPU, then the output is on the GPU. If both `A` and `Y` are `dlarray` objects, then the output `dlarray` is formatted with the same data format as the input `A`.
`double`	The output is a `dlarray` that contains data of type `double`.

`gather`	• The supported syntaxes are:
	  • `dlX = gather(dlA)`
	  • `[dlX,dlY,dlZ,...] = gather(dlA,dlB,dlC,...)`
	• `gather(dlA)` returns a `dlarray` containing numeric or logical data. This function applies `gather` to the underlying data in the `dlarray` `dlA`. If `dlA` is on the GPU, then `dlX` is in the local workspace, not on the GPU. If `dlA` is in the local workspace (not on the GPU), then `dlX` is equal to `dlA`.
	• `gather(dlA,dlB,dlC,...)` gathers multiple arrays.

`gpuArray`	• This function requires Parallel Computing Toolbox. • `gpuArray` returns a `dlarray` containing a `gpuArray`. This function applies `gpuArray` to the underlying data. If the input `dlarray` is in the local workspace, then its data is moved to the GPU and internally represented as a `gpuArray`. If the input `dlarray` is on the GPU, then the output `dlarray` is equal to the input `dlarray`.

Function	Notes and Limitations
`logical`	The output is a `dlarray` that contains data of type `logical`.
`single`	The output is a `dlarray` that contains data of type `single`.

Comparison Functions

Function	Notes and Limitations
`isequal`	• The syntax with more than two input arguments is not supported. • Two `dlarray` inputs are equal if the numeric data they represent are equal and if they both are either formatted with the same data format or unformatted.
`isequaln`	• The syntax with more than two input arguments is not supported. • Two `dlarray` inputs are equal if the numeric data they represent are equal (treating `NaN`s as equal) and if they both are either formatted with the same data format or unformatted.

Data Type and Value Identification Functions

Function	Notes and Limitations
`isdlarray`	N/A
`isfinite`	The software applies the function to the underlying data of an input `dlarray`.
`isfloat`	
`isgpuarray`	
`isinf`	
`islogical`	
`isnan`	
`isnumeric`	
`isreal`	
`isUnderlyingType`	N/A
`mustBeUnderlyingType`	
`underlyingType`	
`validateattributes`	If input array `A` is a formatted `dlarray`, its dimensions are permuted to match the order `"SCBTU"`. Size validation is applied after permutation.

Size Identification Functions

Function	Notes and Limitations
`iscolumn`	This function returns `true` for a `dlarray` that is a column vector, where each dimension except the first is a singleton. For example, a 3-by-1-by-1 `dlarray` is a column vector.
`ismatrix`	This function returns `true` for `dlarray` objects with only two dimensions and for `dlarray` objects where each dimension except the first two is a singleton. For example, a 3-by-4-by-1 `dlarray` is a matrix.
`isrow`	This function returns `true` for a `dlarray` that is a row vector, where each dimension except the second is a singleton. For example, a 1-by-3-by-1 `dlarray` is a row vector.
`isscalar`	N/A
`isvector`	This function returns `true` for a `dlarray` that is a row vector or column vector. Note that `isvector` does not consider a 1-by-1-by-3 `dlarray` to be a vector.
`length`	N/A
`ndims`	If the input `dlarray` `dlX` is formatted, then `ndims(dlX)` returns the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.
`numel`	N/A
`size`	If the input `dlarray` `dlX` is formatted, then `size(dlX)` returns a vector of length equal to the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.

Creator Functions

Function	Notes and Limitations
`false`	Only the `'like'` syntax is supported for `dlarray`.
`inf`	
`nan`	
`ones`	
`rand`	
`randi`	
`randn`	
`true`	
`zeros`	

String and Character Functions

Function	Notes and Limitations
compose	N/A
fprintf	
int2str	
mat2str	
num2str	
sprintf	

Notable dlarray Behaviors

Implicit Expansion with Data Formats

Some functions use implicit expansion to combine two formatted `dlarray` inputs. The function introduces labeled singleton dimensions (dimensions of size 1) into the inputs, as necessary, to make their formats match. The function inserts singleton dimensions at the end of each block of dimensions with the same label.

To see an example of this behavior, enter the following code.

```
X = ones(2,3,2);
dlX = dlarray(X,'SCB')
Y = 1:3;
dlY = dlarray(Y,'C')
dlZ = dlX.*dlY
```

```
dlX =

  2(S) × 3(C) × 2(B) dlarray


(:,:,1) =

     1     1     1
     1     1     1


(:,:,2) =

     1     1     1
     1     1     1


dlY =

  3(C) × 1(U) dlarray

     1
     2
     3
```

```
dlZ =

  2(S) × 3(C) × 2(B) dlarray


(:,:,1) =

     1     2     3
     1     2     3


(:,:,2) =

     1     2     3
     1     2     3
```

In this example, `dlZ(i,j,k) = dlX(i,j,k).*dlY(j)` for indices `i`, `j`, and `k`. The second dimension of `dlZ` (labeled `'C'`) corresponds to the second dimension of `dlX` and the first dimension of `dlY`.

In general, the format of one `dlarray` input does not need to be a subset of the format of another `dlarray` input. For example, if `dlX` and `dlY` are input arguments with `dims(dlX) = 'SCB'` and `dims(dlY) = 'SSCT'`, then the output `dlZ` has `dims(dlZ) = 'SSCBT'`. The `'S'` dimension of `dlX` maps to the first `'S'` dimension of `dlY`.

Special 'U' Dimension Behavior

The `'U'` dimension of a `dlarray` behaves differently from other labeled dimensions in that it exhibits the standard MATLAB singleton dimension behavior. You can think of a formatted `dlarray` as having infinitely many `'U'` dimensions of size 1 following the dimensions returned by `size`.

The software discards a `'U'` label unless the dimension is nonsingleton or it is one of the first two dimensions of the `dlarray`.

To see an example of this behavior, enter the following code.

```
X = ones(2,2);
dlX = dlarray(X,'SC')
dlX(:,:,2) = 2

dlX =

  2(S) × 2(C) dlarray

     1     1
     1     1


dlX =

  2(S) × 2(C) × 2(U) dlarray


(:,:,1) =

     1     1
     1     1
```

```
(:,:,2) =

     2     2
     2     2
```

In this example, the software expands a formatted two-dimensional `dlarray` to a three-dimensional `dlarray`, and labels the third dimension with `'U'` by default. For an example of how the `'U'` dimension is used in implicit expansion, see “Implicit Expansion with Data Formats” on page 18-436.

Indexing

Indexing with a `dlarray` is supported and exhibits the following behaviors:

- `dlX(idx1,...,idxn)` returns a `dlarray` with the same data format as `dlX` if `n` is greater than or equal to `ndims(dlX)`. Otherwise, it returns an unformatted `dlarray`.
- If you set `dlY(idx1,...,idxn) = dlX`, then the data format of `dlY` is preserved, although the software might add or remove trailing `'U'` dimension labels. The data format of `dlX` has no impact on this operation.
- If you delete parts of a `dlarray` using `dlX(idx1,…,idxn) = []`, then the data format of `dlX` is preserved if `n` is greater than or equal to `ndims(dlX)`. Otherwise, `dlX` is returned unformatted.

Round-off Error

When you use a function with a `dlarray` input, the order of the operations within the function can change based on the internal storage order of the `dlarray`. This change can result in differences on the order of round-off for two `dlarray` objects that are otherwise equal.

See Also

`dlarray` | `dlgradient` | `dlfeval` | `dlnetwork`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 18-209
- “Train Network Using Custom Training Loop” on page 18-225
- “Specify Training Options in Custom Training Loop” on page 18-216
- “Define Model Gradients Function for Custom Training Loop” on page 18-231
- “Update Batch Normalization Statistics in Custom Training Loop” on page 18-236
- “Make Predictions Using dlnetwork Object” on page 18-255
- “Train Network Using Model Function” on page 18-259
- “Update Batch Normalization Statistics Using Model Function” on page 18-272
- “Make Predictions Using Model Function” on page 18-286
- “Initialize Learnable Parameters for Model Function” on page 18-292

Deep Learning Data Preprocessing

Datastores for Deep Learning

Datastores in MATLAB are a convenient way of working with and representing collections of data that are too large to fit in memory at one time. Because deep learning often requires large amounts of data, datastores are an important part of the deep learning workflow in MATLAB.

Select Datastore

For many applications, the easiest approach is to start with a built-in datastore. For more information about the available built-in datastores, see “Select Datastore for File Format or Application”. However, only some types of built-in datastores can be used directly as input for network training, validation, and inference. These datastores are:

Datastore	Description	Additional Toolbox Required
`ImageDatastore`	Datastore for image data	none
`AugmentedImageDatastore`	Datastore for resizing and augmenting training images Datastore is nondeterministic	none
`PixelLabelDatastore`	Datastore for pixel label data	Computer Vision Toolbox
`PixelLabelImageDatastore`	Datastore for training semantic segmentation networks Datastore is nondeterministic	Computer Vision Toolbox
`boxLabelDatastore`	Datastore for bounding box label data	Computer Vision Toolbox
`RandomPatchExtractionDatastore`	Datastore for extracting random patches from image-based data Datastore is nondeterministic	Image Processing Toolbox
`blockedImageDatastore`	Datastore for blockwise reading and processing of image data, including large images that do not fit in memory	Image Processing Toolbox
`DenoisingImageDatastore`	Datastore to train an image denoising deep neural network Datastore is nondeterministic	Image Processing Toolbox

Other built-in datastores can be used as input for deep learning, but the data read from these datastores must be preprocessed into a format required by a deep learning network. For more information on the required format of read data, see “Input Datastore for Training, Validation, and Inference” on page 19-3. For more information on how to preprocess data read from datastores, see “Transform and Combine Datastores” on page 19-6.

For some applications, there may not be a built-in datastore type that fits your data well. For these problems, you can create a custom datastore. For more information, see “Develop Custom Datastore”. All custom datastores are valid inputs to deep learning interfaces as long as the `read` function of the custom datastore returns data in the required form.

Input Datastore for Training, Validation, and Inference

Datastores are valid inputs in Deep Learning Toolbox for training, validation, and inference.

Training and Validation

You can use an image datastore or other types of datastore as a source of training data when training using the `trainNetwork` function. To use a datastore for validation, use the `'ValidationData'` name-value pair argument in `trainingOptions`.

To be a valid input for training or validation, the `read` function of a datastore must return data as either a cell array or a table (with the exception of `ImageDatastore` objects which can output numeric arrays and custom mini-batch datastores which must output tables).

For networks with a single input, the table or cell array returned by the datastore must have two columns. The first column of data represents inputs to the network and the second column of data represents responses. Each row of data represents a separate observation. For `ImageDatastore` only, `trainNetwork` and `trainingOptions` support data returned as integer arrays and single-column cell array of integer arrays.

To use a datastore for networks with multiple input layers, use the `combine` and `transform` functions to create a datastore that outputs a cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs. In this case, the first `numInputs` columns specify the predictors for each input and the last column specifies the responses. The order of inputs is given by the `InputNames` property of the layer graph `layers`.

The following table shows example outputs of calling the `read` function for datastore `ds`.

Network Architecture	Datastore Output	Example Output

Single input layer	Table or cell array with two columns.
	
	The first and second columns specify the predictors and responses, respectively.
	
	Table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.
	
	Custom mini-batch datastores must output tables.
		Table for network with one input and one output:
		
		`data = read(ds)`
		
		`data =`
		
		`4×2 table`
		
		`Predictors  Response`
		`{224×224×3 double}  2`
		`{224×224×3 double}  7`
		`{224×224×3 double}  9`
		`{224×224×3 double}  9`

Network Architecture	Datastore Output	Example Output

		Cell array for network with one input and one output:
		`data = read(ds)`
		`data =`
		`4×2 cell array`
		`{224×224×3 double} {[2]}`
		`{224×224×3 double} {[7]}`
		`{224×224×3 double} {[9]}`
		`{224×224×3 double} {[9]}`

Multiple input layers	Cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs.
	The first `numInputs` columns specify the predictors for each input and the last column specifies the responses.
	The order of inputs is given by the `InputNames` property of the layer graph `layers`.
		Cell array for network with two inputs and one output.
		`data = read(ds)`
		`data =`
		`4×3 cell array`
		`{224×224×3 double} {128×128×3 do`
		`{224×224×3 double} {128×128×3 do`
		`{224×224×3 double} {128×128×3 do`
		`{224×224×3 double} {128×128×3 do`

The format of the predictors depend on the type of data.

Data	Format of Predictors
2-D image	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the image, respectively.
3-D image	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the image, respectively.
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.
1-D image sequence	*h*-by-*c*-by-*s* array, where *h* and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.

Data	Format of Predictors
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
Features	*c*-by-1 column vector, where *c* is the number of features.

For predictors returned in tables, the elements must contain a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

The `trainNetwork` function does not support networks with multiple sequence input layers.

The format of the responses depend on the type of task.

Task	Format of Responses
Classification	Categorical scalar
Regression	• Scalar • Numeric vector • 3-D numeric array representing an image
Sequence-to-sequence classification	1-by-*s* sequence of categorical labels, where *s* is the sequence length of the corresponding predictor sequence.
Sequence-to-sequence regression	*R*-by-*s* matrix, where *R* is the number of responses and *s* is the sequence length of the corresponding predictor sequence.

For responses returned in tables, the elements must be a categorical scalar, a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

Prediction

For inference using `predict`, `classify`, and `activations`, a datastore is only required to yield the columns corresponding to the predictors. The inference functions use the first `NumInputs` columns and ignores the subsequent layers, where `NumInputs` is the number of network input layers.

Specify Read Size and Mini-Batch Size

A datastore may return any number of rows (observations) for each call to `read`. Functions such as `trainNetwork`, `predict`, `classify`, and `activations` that accept datastores and support specifying a `'MiniBatchSize'` call `read` as many times as is necessary to form complete mini-batches of data. As these functions form mini-batches, they use internal queues in memory to store read data. For example, if a datastore consistently returns 64 rows per call to `read` and `MiniBatchSize` is `128`, then to form each mini-batch of data requires two calls to `read`.

For best runtime performance, it is recommended to configure datastores such that the number of observations returned by `read` is equal to the `'MiniBatchSize'`. For datastores that have a

'ReadSize' property, set the 'ReadSize' to change the number of observations returned by the datastore for each call to read.

Transform and Combine Datastores

Deep learning frequently requires the data to be preprocessed and augmented before data is in an appropriate form to input to a network. The transform and combine functions of datastore are useful in preparing data to be fed into a network.

To use a datastore for networks with multiple input layers, use the combine and transform functions to create a datastore that outputs a cell array with (numInputs + 1) columns, where numInputs is the number of network inputs. In this case, the first numInputs columns specify the predictors for each input and the last column specifies the responses. The order of inputs is given by the InputNames property of the layer graph layers.

Transform Datastores

A transformed datastore applies a particular data transformation to an underlying datastore when reading data. To create a transformed datastore, use the transform function and specify the underlying datastore and the transformation.

- For complex transformations involving several preprocessing operations, define the complete set of transformations in your own function. Then, specify a handle to your function as the @fcn argument of transform. For more information, see “Create Functions in Files”.
- For simple transformations that can be expressed in one line of code, you can specify a handle to an anonymous function as the @fcn argument of transform. For more information, see “Anonymous Functions”.

The function handle provided to transform must accept input data in the same format as returned by the read function of the underlying datastore.

Example: Transform Image Datastore to Train Digit Classification Network

This example uses the transform function to create a training set in which randomized 90 degree rotation is added to each image within an image datastore. Pass the resulting TransformedDatastore to trainNetwork to train a simple digit classification network.

Create an image datastore containing digit images.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Set the mini-batch size equal to the ReadSize of the image datastore.

```
miniBatchSize = 128;
imds.ReadSize = miniBatchSize;
```

Transform images in the image datastore by adding randomized 90 degree rotation. The transformation function, preprocessForTraining, is defined at the end of this example.

```
dsTrain = transform(imds,@preprocessForTraining,'IncludeInfo',true)
```

```
dsTrain =

  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 matlab.io.datastore.ImageDatastore]
             Transforms: {@preprocessForTraining}
            IncludeInfo: 1
```

Specify layers of the network and training options, then train the network using the transformed datastore `dsTrain` as a source of data.

```
layers = [
    imageInputLayer([28 28 1],'Normalization','none')
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,'Stride',2)
    fullyConnectedLayer(10);
    softmaxLayer
    classificationLayer];

options = trainingOptions('adam', ...
    'Plots','training-progress', ...
    'MiniBatchSize',miniBatchSize);

net = trainNetwork(dsTrain,layers,options);
```

Define the transformation function, `preprocessForTraining`. The input to the function is a batch of data, `data`, read from the underlying datastore. The function in this example loops through each read image and performs randomized rotation, then returns the transformed image and corresponding label as a cell array as expected by `trainNetwork`.

```
function [dataOut,info] = preprocessForTraining(data,info)

numRows = size(data,1);
dataOut = cell(numRows,2);

for idx = 1:numRows

    % Randomized 90 degree rotation
    imgOut = rot90(data{idx,1},randi(4)-1);

    % Return the label from info struct as the
    % second column in dataOut.
    dataOut(idx,:) = {imgOut,info.Label(idx)};

end

end
```

Combine Datastores

The `combine` function associates multiple datastores. Operating on the resulting `CombinedDatastore`, such as resetting the datastore, performs the same operation on all of the underlying datastores. Calling the `read` function of a combined datastore reads one batch of data from all of the *N* underlying datastores, which must return the same number of observations. Reading from a combined datastore returns the horizontally concatenated results in an *N*-column cell array that is suitable for training and validation. Shuffling a combined datastore results in an identical randomized ordering of files in the underlying datastores.

For example, if you are training an image-to-image regression network, then you can create the training data set by combining two image datastores. This sample code demonstrates combining two image datastores named `imdsX` and `imdsY`. The combined datastore `imdsTrain` returns data as a two-column cell array.

```
imdsX = imageDatastore(___);
imdsY = imageDatastore(___);
imdsTrain = combine(imdsX,imdsY)
```

```
imdsTrain =

  CombinedDatastore with properties:

    UnderlyingDatastores: {1×2 cell}
```

If you have Image Processing Toolbox, then the `randomPatchExtractionDatastore` provides an alternate solution to associating image-based data in `ImageDatastore`s, `PixelLabelDatastore`s, and `TransformedDatastore`s. A `randomPatchExtractionDatastore` has several advantages over associating data using the `combine` function. Specifically, a random patch extraction datastore:

- Provides an easy way to extract patches from both 2-D and 3-D data without requiring you to implement a custom cropping operation using `transform` and `combine`
- Provides an easy way to generate multiple patches per image per mini-batch without requiring you to define a custom concatenation operation using `transform`.
- Supports efficient conversion between categorical and numeric data when applying image transforms to categorical data
- Supports parallel training
- Improves performance by caching images

Use Datastore for Parallel Training and Background Dispatching

Datastores used for parallel training or multi-GPU training must be partitionable. To determine if a datastore is partitionable, use the function `isPartitionable`. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training in parallel or using single or multiple GPUs requires Parallel Computing Toolbox.

Many built-in datastores are already partitionable because they support the `partition` function. Using the `transform` and `combine` functions with built-in datastores frequently maintains support for parallel and multi-GPU training.

If you need to create a custom datastore that supports parallel or multi-GPU training, then your datastore must implement the `matlab.io.datastore.Partitionable` class.

Partitionable datastores support reading training data using background dispatching. Background dispatching queues data in memory while the GPU is working. Specify background dispatching using the `'DispatchInBackground'` name-value pair argument of `trainingOptions`. Background dispatching requires Parallel Computing Toolbox.

When training in parallel, datastores do not support specifying the `'Shuffle'` name-value pair argument of `trainingOptions` as `'none'`.

See Also

`transform` | `combine` | `trainNetwork` | `trainingOptions` | `read`

Related Examples

- “Prepare Datastore for Image-to-Image Regression” on page 19-91
- “Classify Text Data Using Convolutional Neural Network” on page 4-97

More About

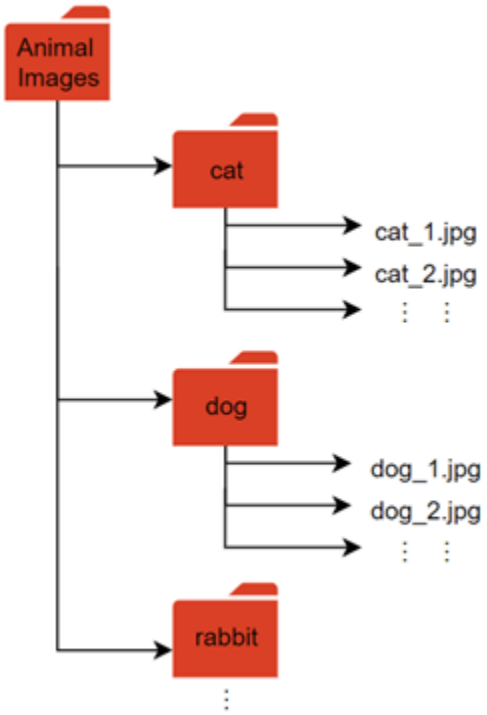
- “Getting Started with Datastore”
- “Select Datastore for File Format or Application”
- “Develop Custom Datastore”

Create and Explore Datastore for Image Classification

This example shows how to create, read, and augment an image datastore for use in training a deep learning network. In particular, this example shows how to create an `ImageDatastore` object from a collection of images, read and extract the properties of the datastore, and create an `augmentedImageDatastore` for use during training.

Create Image Datastore

Use an `imageDatastore` object to manage a large collection of images that cannot altogether fit in memory. Large collections of images are common in deep learning applications, which regularly involve training on thousands of labeled images. These images are often stored in a folder, with subfolders containing images for each class.

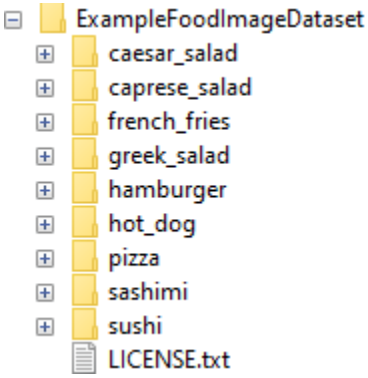


Download Data Set

This example uses the Example Food Images data set, which contains 978 photographs of food in nine classes and is approximately 77 MB in size. Download the `ExampleFoodImageDataset.zip` file from the MathWorks website, then unzip the file.

```
zipFile = matlab.internal.examples.downloadSupportFile('nnet','data/ExampleFoodImageDataset.zip'
filepath = fileparts(zipFile);
dataFolder = fullfile(filepath,'ExampleFoodImageDataset');
unzip(zipFile,dataFolder);
```

The images in this data set are separated into subfolders for each class.



Create an image datastore from the images in the path and their subfolders. Use the folder names as label names.

```
foodImds = imageDatastore(dataFolder, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Properties of Datastore

Extract the properties of the datastore.

Find the total number of observations. This data set has 978 observations split into nine classes.

```
numObs = length(foodImds.Labels)
```

```
numObs = 978
```

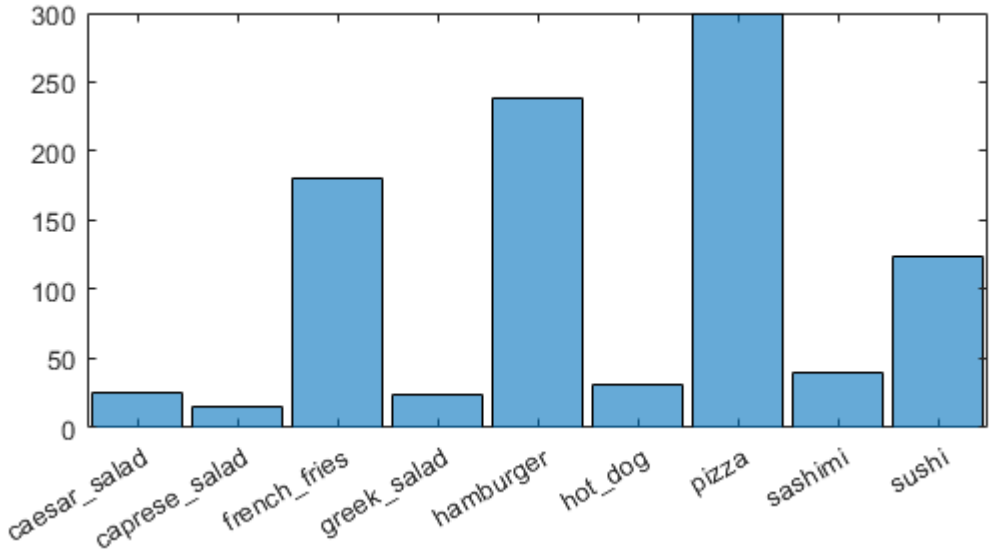
Find the number of observations per class. You can see that this data set does not contain an equal number of observations in each class.

```
numObsPerClass = countEachLabel(foodImds)
```

```
numObsPerClass=9×2 table
        Label        Count
    _____________    _____

    caesar_salad       26
    caprese_salad      15
    french_fries      181
    greek_salad        24
    hamburger         238
    hot_dog            31
    pizza             299
    sashimi            40
    sushi             124
```

You can also visualize the distribution of the class labels using a histogram.

```
histogram(foodImds.Labels)
set(gca,'TickLabelInterpreter','none')
```



Explore Datastore

Check that the data is as expected by viewing a random selection of images from the datastore.

```
numObsToShow = 8;
idx = randperm(numObs,numObsToShow);
imshow(imtile(foodImds.Files(idx),'GridSize',[2 4],'ThumbnailSize',[100 100]))
```

You can also view images that belong to a specific class.

```
class = pizza;
idxClass = find(foodImds.Labels == class);
idx = randsample(idxClass,numObsToShow);
imshow(imtile(foodImds.Files(idx),'GridSize',[2 4],'ThumbnailSize',[100 100]));
```

To take a closer look at individual images in your datastore or folder, use the Image Browser (Image Processing Toolbox) app.

Image Augmentation

Augmentation enables you to train networks to be invariant to distortions in image data. For example, you can add randomized rotations to input images so that a network is invariant to the presence of rotation. An `augmentedImageDatastore` object provides a convenient way to apply a limited set of augmentations to 2-D images for classification problems.

Define an augmentation scheme. This scheme applies a random rotation between [–90,90] degrees and a random scaling between [1,2]. The augmented datastore automatically resizes the images to the `inputSize` value during training.

```
imageAugmenter = imageDataAugmenter( ...
    'RandRotation',[-90 90], ...
    'RandScale',[1 2]);

inputSize = [100 100];
```

Using the augmentation scheme, define the augmented image datastore.

```
augFoodImds = augmentedImageDatastore(inputSize,foodImds, ...
    'DataAugmentation',imageAugmenter);
```

The augmented datastore contains the same number of images as the original image datastore.

```
augFoodImds.NumObservations
```

```
ans = 978
```

When you use an augmented image datastore as a source of training images, the datastore randomly perturbs the training data for each epoch, where an epoch is a full pass of the training algorithm over the entire training data set. Therefore, each epoch uses a slightly different data set, but the actual number of training images in each epoch does not change.

Visualize Augmented Data

Visualize the augmented image data that you want to use to train the network.

Shuffle the datastore.

```
augFoodImds = shuffle(augFoodImds);
```

The `augmentedImageDatastore` object applies the transformations when reading the datastore and does not store the transformed images in memory. Consequently, each time you read the same images, you see a random combination of the augmentations defined.

Use the `read` function to read a subset of the augmented datastore.

```
subset1 = read(augFoodImds);
```

Reset the datastore to its state before calling read and read a subset of the datastore again.

```
reset(augFoodImds)
subset2 = read(augFoodImds);
```

Display the two subsets of the augmented images.

```
imshow(imtile(subset1.input,'GridSize',[2 4]))
```

```
imshow(imtile(subset2.input,'GridSize',[2 4]))
```

You can see that both instances show the same images with different transformations. Applying transformations to images is useful in deep learning applications, as you can train the network on randomly altered versions of an image. Doing so exposes the network to different variations of images from that class and enables it to learn to classify images even if they have different visual properties.

After creating your datastore object, use the Deep Network Designer app or `trainNetwork` function to train an image classification network. For an example, see "Transfer Learning Using Pretrained Network" on page 3-33.

For more information on preprocessing images for deep learning applications, see "Preprocess Images for Deep Learning" on page 19-16. You can also apply more advanced augmentations, such as varying levels of brightness or saturation, by using the `transform` and `combine` functions. For more information, see "Datastores for Deep Learning" on page 19-2.

See Also

`trainNetwork` | **Deep Network Designer** | `augmentedImageDatastore` | `imageDatastore`

Related Examples

- "Transfer Learning with Deep Network Designer" on page 2-2
- "Transfer Learning Using Pretrained Network" on page 3-33
- "Data Sets for Deep Learning" on page 19-118
- "Datastores for Deep Learning" on page 19-2
- "Preprocess Images for Deep Learning" on page 19-16

Preprocess Images for Deep Learning

To train a network and make predictions on new data, your images must match the input size of the network. If you need to adjust the size of your images to match the network, then you can rescale or crop your data to the required size.

You can effectively increase the amount of training data by applying randomized augmentation to your data. Augmentation also enables you to train networks to be invariant to distortions in image data. For example, you can add randomized rotations to input images so that a network is invariant to the presence of rotation in input images. An `augmentedImageDatastore` provides a convenient way to apply a limited set of augmentations to 2-D images for classification problems.

For more advanced preprocessing operations, to preprocess images for regression problems, or to preprocess 3-D volumetric images, you can start with a built-in datastore. You can also preprocess images according to your own pipeline by using the `transform` and `combine` functions.

Resize Images Using Rescaling and Cropping

You can store image data as a numeric array, an `ImageDatastore` object, or a table. An `ImageDatastore` enables you to import data in batches from image collections that are too large to fit in memory. You can use an augmented image datastore or a resized 4-D array for training, prediction, and classification. You can use a resized 3-D array for prediction and classification only.

There are two ways to resize image data to match the input size of a network.

- Rescaling multiplies the height and width of the image by a scaling factor. If the scaling factor is not identical in the vertical and horizontal directions, then rescaling changes the spatial extents of the pixels and the aspect ratio.
- Cropping extracts a subregion of the image and preserves the spatial extent of each pixel. You can crop images from the center or from random positions in the image.

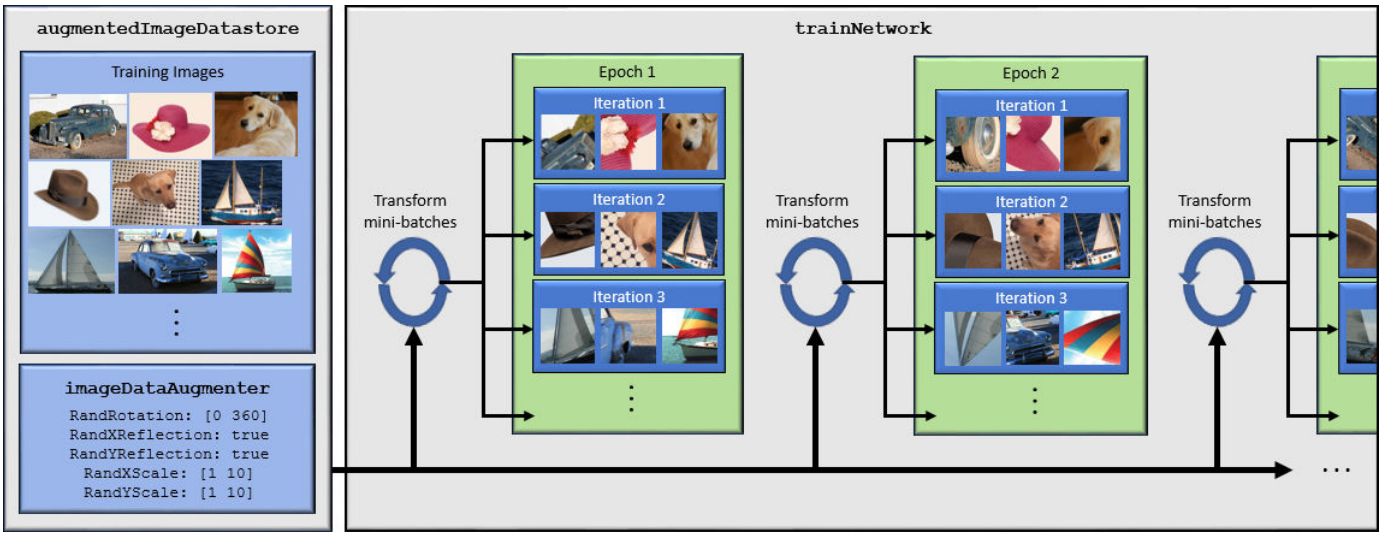
Resizing Option	Data Format	Resizing Function	Sample Code
Rescaling	• 3-D array representing a single color or multispectral image • 3-D array representing a stack of grayscale images • 4-D array representing a stack of images	`imresize`	`im = imresize(I,outputSize);` `outputSize` specifies the dimensions of the rescaled image.
	• 4-D array representing a stack of images • `ImageDatastore` • table	`augmentedImageDatastore`	`auimds = augmentedImageDatastore(outpu` `outputSize` specifies the dimensions of the rescaled image.
Cropping	• 3-D array representing a single color or multispectral image	`imcrop`	`im = imcrop(I,rect);` `rect` specifies the size and position of the 2-D cropping window.

Resizing Option	Data Format	Resizing Function	Sample Code
	• 3-D array representing a stack of grayscale images • 4-D array representing a stack of color or multispectral images	`imcrop3`	`im = imcrop3(I,cuboid);` `cuboid` specifies the size and position of the 3-D cropping window.
	• 4-D array representing a stack of images • `ImageDatastore` • table	`augmentedImageDatastore`	`auimds = augmentedImageDatastore(outpu` Specify `m` as `'centercrop'` to crop from the center of the input image. Specify `m` as `'randcrop'` to crop from a random location in the input image.

Augment Images for Training with Random Geometric Transformations

For image classification problems, you can use an `augmentedImageDatastore` to augment images with a random combination of resizing, rotation, reflection, shear, and translation transformations.

The diagram shows how `trainNetwork` uses an augmented image datastore to transform training data for each epoch. When you use data augmentation, one randomly augmented version of each image is used during each epoch of training. For an example of the workflow, see “Train Network with Augmented Images”.



1. Specify training images.
2. Configure image transformation options, such as the range of rotation angles and whether to apply reflection at random, by creating an `imageDataAugmenter`.

Tip To preview the transformations applied to sample images, use the `augment` function.

3. Create an `augmentedImageDatastore`. Specify the training images, the size of output images, and the `imageDataAugmenter`. The size of output images must be compatible with the size of the `imageInputLayer` of the network.
4. Train the network, specifying the augmented image datastore as the data source for `trainNetwork`. For each iteration of training, the augmented image datastore applies a random combination of transformations to images in the mini-batch of training data.

   When you use an augmented image datastore as a source of training images, the datastore randomly perturbs the training data for each epoch, so that each epoch uses a slightly different data set. The actual number of training images at each epoch does not change. The transformed images are not stored in memory.

Perform Additional Image Processing Operations Using Built-In Datastores

Some datastores perform specific and limited image preprocessing operations when they read a batch of data. These application-specific datastores are listed in the table. You can use these datastores as a source of training, validation, and test data sets for deep learning applications that use Deep Learning Toolbox. All of these datastores return data in a format supported by `trainNetwork`.

Datastore	Description
`augmentedImageDatastore`	Apply random affine geometric transformations, including resizing, rotation, reflection, shear, and translation, for training deep neural networks. For an example, see “Transfer Learning Using Pretrained Network” on page 3-33.
`pixelLabelImageDatastore`	Apply identical affine geometric transformations to images and corresponding ground truth labels for training semantic segmentation networks (requires Computer Vision Toolbox). For an example, see “Semantic Segmentation Using Deep Learning” on page 8-126.
`randomPatchExtractionDatastore`	Extract multiple pairs of random patches from images or pixel label images (requires Image Processing Toolbox). You optionally can apply identical random affine geometric transformations to the pairs of patches. For an example, see “Single Image Super-Resolution Using Deep Learning” on page 9-8.
`denoisingImageDatastore`	Apply randomly generated Gaussian noise for training denoising networks (requires Image Processing Toolbox).

Apply Custom Image Processing Pipelines Using Combine and Transform

To perform more general and complex image preprocessing operations than offered by the application-specific datastores, you can use the `transform` and `combine` functions. For more information, see “Datastores for Deep Learning” on page 19-2.

Transform Datastores with Image Data

The `transform` function creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore according to a transformation function that you define.

The custom transformation function must accept data in the format returned by the `read` function of the underlying datastore. For image data in an `ImageDatastore`, the format depends on the `ReadSize` property .

- When `ReadSize` is 1, the transformation function must accept an integer array. The size of the array is consistent with the type of images in the `ImageDatastore`. For example, a grayscale image has dimensions *m*-by-*n*, a truecolor image has dimensions *m*-by-*n*-by-3, and a multispectral image with *c* channels has dimensions *m*-by-*n*-by-*c*.
- When `ReadSize` is greater than 1, the transformation function must accept a cell array of image data. Each element corresponds to an image in the batch.

The `transform` function must return data that matches the input size of the network. The `transform` function does not support one-to-many observation mappings.

Tip The `transform` function supports prefetching when the underlying `ImageDatastore` reads a batch of JPG or PNG image files. For these image types, do not use the `readFcn` argument of `ImageDatastore` to apply image preprocessing, as this option is usually significantly slower. If you use a custom read function, then `ImageDatastore` does not prefetch.

Combine Datastores with Image Data

The `combine` function concatenates the data read from multiple datastores and maintains parity between the datastores.

- Concatenate data into a two-column table or two-column cell array for training networks with a single input, such as image-to-image regression networks.
- Concatenate data to a (`numInputs`+1)-column cell array for training networks with multiple inputs.

See Also

`trainNetwork` | `imresize` | `transform` | `combine` | `ImageDatastore`

Related Examples

- “Train Network with Augmented Images”
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Create and Explore Datastore for Image Classification” on page 19-10
- “Prepare Datastore for Image-to-Image Regression” on page 19-91

More About

- “Datastores for Deep Learning” on page 19-2
- “Preprocess Volumes for Deep Learning” on page 19-20
- “Deep Learning in MATLAB” on page 1-2

Preprocess Volumes for Deep Learning

Read Volumetric Data

Supported file formats for volumetric image data include MAT-files, Digital Imaging and Communications in Medicine (DICOM) files, and Neuroimaging Informatics Technology Initiative (NIfTI) files.

Read volumetric image data into an `ImageDatastore`. Read volumetric pixel label data into a `PixelLabelDatastore`. For more information, see “Datastores for Deep Learning” on page 19-2.

The table shows typical usages of `imageDatastore` and `pixelLabelDatastore` for each of the supported file formats. When you create the datastore, specify the `'FileExtensions'` argument as the file extensions of your data. Specify the `ReadFcn` property as a function handle that reads data of the file format. The `filepath` argument specifies the path to the files or folder containing image data. For pixel label images, the additional `classNames` and `pixelLabelID` arguments specify the mapping of voxel label values to class names.

Image File Format	Create Image Datastore or Pixel Label Datastore

MAT	`volds = imageDatastore(filepath, ...`
	`  'FileExtensions','.mat','ReadFcn',@(x) fcn(x));`
	
	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ...`
	`  'FileExtensions','.mat','ReadFcn',@(x) fcn(x));`
	
	`fcn` is a custom function that reads data from a MAT file. For example, this code defines a function called `matRead` that loads volume data from the first variable of a MAT file. Save the function in a file called `matRead.m`.
	
	`function data = matRead(filename)`
	`  inp = load(filename);`
	`  f = fields(inp);`
	`  data = inp.(f{1});`
	`end`

DICOM volume in single file	`volds = imageDatastore(filepath, ...`
	`  'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));`
	
	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ...`
	`  'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));`
	
	For more information about reading DICOM files, see `dicomread`.

DICOM volume in multiple files	Follow these steps. For an example, see “Create Image Datastore Containing Single and Multi-File DICOM Volumes” (Image Processing Toolbox). • Aggregate the files into a single study by using the `dicomCollection` function. • Read the DICOM data in the study by using the `dicomreadVolume` function. • Write each volume as a MAT file. • Create the `ImageDatastore` or `PixelLabelDatastore` from the collection of MAT files by following the procedure for MAT files.

Image File Format	Create Image Datastore or Pixel Label Datastore

NIfTI	`volds = imageDatastore(filepath, ...`
	`  'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));`
	
	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID, ...`
	`  'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));`
	
	For more information about reading NIfTI files, see `niftiread`.

Pair Image and Label Data

To associate volumetric image and label data for semantic segmentation, or two volumetric image datastores for regression, use a `randomPatchExtractionDatastore`. A random patch extraction datastore extracts corresponding randomly-positioned patches from two datastores. Patching is a common technique to prevent running out of memory when training with arbitrarily large volumes. Specify a patch size that matches the input size of the network and, for memory efficiency, is smaller than the full size of the volume, such as 64-by-64-by-64 voxels.

You can also use the `combine` function to associate two datastores. However, associating two datastores using a `randomPatchExtractionDatastore` has some benefits over `combine`.

- `randomPatchExtractionDatastore` supports parallel training, multi-GPU training, and prefetch reading. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value argument of `trainingOptions`. Specify prefetch reading using the `'DispatchInBackground'` name-value argument of `trainingOptions`. Prefetch reading requires Parallel Computing Toolbox.
- `randomPatchExtractionDatastore` inherently supports patch extraction. In contrast, to extract patches from a `CombinedDatastore`, you must define your own function that crops images into patches, and then use the `transform` function to apply the cropping operations.
- `randomPatchExtractionDatastore` can generate several image patches from one test image. One-to-many patch extraction effectively increases the amount of available training data.

Preprocess Volumetric Data

Deep learning frequently requires the data to be preprocessed and augmented. For example, you may want to normalize image intensities, enhance image contrast, or add randomized affine transformations to prevent overfitting.

To preprocess volumetric data, use the `transform` function. `transform` creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore according to the set of operations you define in a custom function. Image Processing Toolbox provides several functions that accept volumetric input. For a full list of functions, see 3-D Volumetric Image Processing (Image Processing Toolbox). You can also preprocess volumetric images using functions in MATLAB that work on multidimensional arrays.

The custom transformation function must accept data in the format returned by the `read` function of the underlying datastore.

Underlying Datastore	Format of Input to Custom Transformation Function
`ImageDatastore`	The input to the custom transformation function depends on the `ReadSize` property. • When `ReadSize` is 1, the transformation function must accept an integer array. The size of the array is consistent with the type of images in the `ImageDatastore`. For example, a grayscale image has size *m*-by-*n*, a truecolor image has size *m*-by-*n*-by-3, and a multispectral image with *c* channels has size *m*-by-*n*-by-*c*. • When `ReadSize` is greater than 1, the transformation function must accept a cell array of image data corresponding to each image in the batch. For more information, see the `read` function of `ImageDatastore`.
`PixelLabelDatastore`	The input to the custom transformation function depends on the `ReadSize` property. • When `ReadSize` is 1, the transformation function must accept a categorical matrix. • When `ReadSize` is greater than 1, the transformation function must accept a cell array of categorical matrices. For more information, see the `read` function of `PixelLabelDatastore`.
`RandomPatchExtractionDatastore`	The input to the custom transformation function must be a table with two columns. For more information, see the `read` function of `RandomPatchExtractionDatastore`.

The `transform` function must return data that matches the input size of the network. The `transform` function does not support one-to-many observation mappings.

To apply random affine transformations to volumetric data in `RandomPatchExtractionDatastore`, you must use the `transform` function. The `DataAugmentation` property of this datastore does not support volumetric data.

Examples

Transform Batch of Volumetric Data in Image Datastore

This example shows how to transform volumetric data in an image datastore using a sample image preprocessing pipeline.

Specify a set of volumetric images saved at MAT files.

```
filepath = fullfile(matlabroot,"toolbox","images","imdata","mristack.mat");
files = [filepath; filepath; filepath];
```

Create an image datastore that stores multiple volumetric images. Specify that the `ReadSize` of the datastore is greater than 1. Specify a custom read function, `matRead`. This function is defined in the Supporting Functions section of this example.

```
volDS = imageDatastore(files,"FileExtensions",".mat", ...
    "ReadSize",3,"ReadFcn",@(x) matRead(x));
```

Specify the input size of the network.

```
inputSize = [128 128];
```

Preprocess the volumetric images in `volDS` using the custom preprocessing pipeline defined in the `preprocessVolumetricIMDS` supporting function.

```
dsTrain = transform(volDS,@(x) preprocessVolumetricIMDS(x,inputSize));
```

Read a batch of data.

```
minibatch = read(dsTrain)
```

```
minibatch=3×1 cell array
    {128x128x21 uint8}
    {128x128x21 uint8}
    {128x128x21 uint8}
```

Supporting Functions

The `matRead` function loads volume data from the first variable of a MAT file.

```
function data = matRead(filename)
    inp = load(filename);
    f = fields(inp);
    data = inp.(f{1});
end
```

The `preprocessVolumetricIMDS` function performs the desired transformations of data read from an underlying image datastore. Because the read size of the image datastore is greater than 1, the function must accept a cell array of image data. The function loops through each read image and transforms the data according to this preprocessing pipeline:

- Randomly rotate the image about the *z*-axis.
- Resize the volume to the size expected by the network.
- Create a noisy version of the image with Gaussian noise.
- Return the image in a cell array.

```
function batchOut = preprocessVolumetricIMDS(batchIn,inputSize)

numRows = size(batchIn,1);
batchOut = cell(numRows,1);

for idx = 1:numRows

    % Perform randomized 90 degree rotation about the z-axis
    imRotated = imrotate3(batchIn{idx,1},90*(randi(4)-1),[0 0 1]);

    % Resize the volume to the size expected by the network
    imResized = imresize(imRotated,inputSize);

    % Add zero-mean Gaussian noise with a normalized variance of 0.01
    imNoisy = imnoise(imResized,'gaussian',0.01);
```

```
    % Return the preprocessed data
    batchOut(idx) = {imNoisy};

end
end
```

Transform Volumetric Data in Random Patch Extraction Datastore

This example shows how to transform pairs of volumetric data in a random patch extraction datastore using a sample image preprocessing pipeline.

Specify two sets of volumetric images saved at MAT files. Each set contains five volumetric images.

```
dir = fullfile(matlabroot,"toolbox","images","imdata","BrainMRILabeled");
filesVol1 = fullfile(dir,"images");
filesVol2 = fullfile(dir,"labels");
```

Store each set of volumetric images in an image datastore. Specify a custom read function, `matRead`. This function is defined in the Supporting Functions section of this example. Use the default `ReadSize` of 1.

```
vol1DS = imageDatastore(filesVol1,"FileExtensions",".mat","ReadFcn",@(x) matRead(x));
vol2DS = imageDatastore(filesVol2,"FileExtensions",".mat","ReadFcn",@(x) matRead(x));
```

Specify the input size of the network.

```
inputSize = [128 128];
```

Create a random patch extraction datastore that extracts corresponding patches from the two datastores. Select three patches per image.

```
patchVolDS = randomPatchExtractionDatastore(vol1DS,vol2DS,inputSize,"PatchesPerImage",3);
```

Preprocess the volumetric images in `patchVolDS` using the custom preprocessing pipeline defined in the `preprocessVolumetricPatchDS` supporting function.

```
dsTrain = transform(patchVolDS,@(x) preprocessVolumetricPatchDS(x));
```

Read a batch of data.

```
minibatch = read(dsTrain)
```

```
minibatch=15×2 table
         InputImage            ResponseImage
    ___________________    ___________________

    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
```

```
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
    {128x128x155 uint16}    {128x128x155 uint8}
```

Supporting Functions

The `matRead` function loads volume data from the first variable of a MAT file.

```
function data = matRead(filename)
    inp = load(filename);
    f = fields(inp);
    data = inp.(f{1});
end
```

The `preprocessVolumetricPatchDS` function performs the desired transformations of data read from the underlying random patch extraction datastore. The function must accept a table. The function transforms the data according to this preprocessing pipeline:

- Randomly select one of five augmentations.
- Apply the same augmentation to the data in both columns of the table.
- Return the augmented image pair in a table.

```
function batchOut = preprocessVolumetricPatchDS(batchIn)

numRows = size(batchIn,1);
batchOut = batchIn;

% 5 augmentations: nil,rot90,fliplr,flipud,rot90(fliplr)
augType = {@(x) x,@rot90,@fliplr,@flipud,@(x) rot90(fliplr(x))};

for idx = 1:numRows

    img = batchIn{idx,1}{1};
    resp = batchIn{idx,2}{1};

    rndIdx = randi(5,1);
    imgAug = augType{rndIdx}(img);
    respAug = augType{rndIdx}(resp);

    batchOut(idx,:) = {imgAug,respAug};

end
end
```

See Also

`trainNetwork` | `imageDatastore` | `pixelLabelDatastore` | `randomPatchExtractionDatastore` | `transform`

Related Examples

- “Create Image Datastore Containing Single and Multi-File DICOM Volumes” (Image Processing Toolbox)
- “3-D Brain Tumor Segmentation Using Deep Learning” on page 8-171

More About

- “Datastores for Deep Learning” on page 19-2
- “Deep Learning in MATLAB” on page 1-2
- “Create Functions in Files”

Preprocess Data for Domain-Specific Deep Learning Applications

Data preprocessing is used for training, validation, and inference. Preprocessing consists of a series of deterministic operations that normalize or enhance desired data features. For example, you can normalize data to a fixed range or rescale data to the size required by the network input layer.

Preprocessing can occur at two stages in the deep learning workflow.

- Commonly, preprocessing occurs as a separate step that you complete before preparing the data to be fed to the network. You load your original data, apply the preprocessing operations, then save the result to disk. The advantage of this approach is that the preprocessing overhead is only required once, then the preprocessed images are readily available as a starting place for all future trials of training a network.
- If you load your data into a datastore, then you can also apply preprocessing during training by using the `transform` and `combine` functions. For more information, see “Datastores for Deep Learning” on page 19-2. The transformed images are not stored in memory. This approach is convenient to avoid writing a second copy of training data to disk if your preprocessing operations are not computationally expensive and do not noticeably impact the speed of training the network.

Data augmentation consists of randomized operations that are applied to the training data while the network is training. Augmentation increases the effective amount of training data and helps to make the network invariant to common distortion in the data. For example, you can add artificial noise to training data so that the network is invariant to noise.

To augment training data, start by loading your data into a datastore. For more information, see “Datastores for Deep Learning” on page 19-2. Some built-in datastores apply a specific and limited set of augmentation to data for specific applications. You can also apply your own set of augmentation operations on data in the datastore by using the `transform` and `combine` functions. During training, the datastore randomly perturbs the training data for each epoch, so that each epoch uses a slightly different data set.

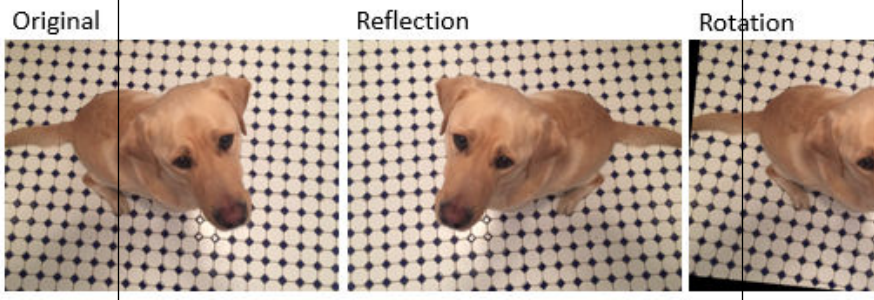
Image Processing Applications

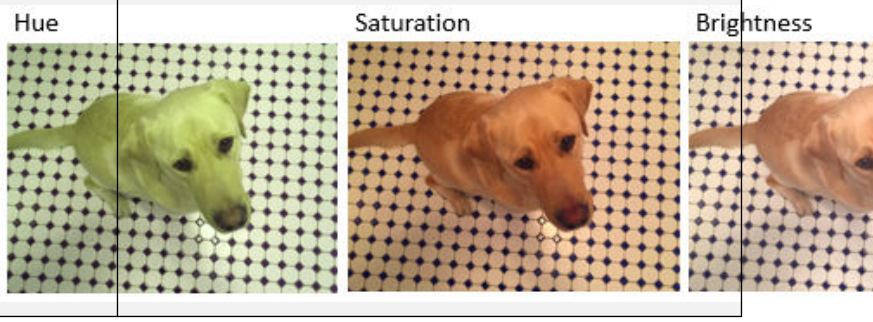
Augment image data to simulate variations in the image acquisition. For example, the most common type of image augmentation operations are geometric transformations such as rotation and translation, which simulate variations in the camera orientation with respect to the scene. Color jitter simulates variations of lighting conditions and color in the scene. Artificial noise simulates distortions caused by the electrical fluctuations in the sensor and analog-to-digital conversion errors. Blur simulates an out-of-focus lens or movement of the camera with respect to the scene.

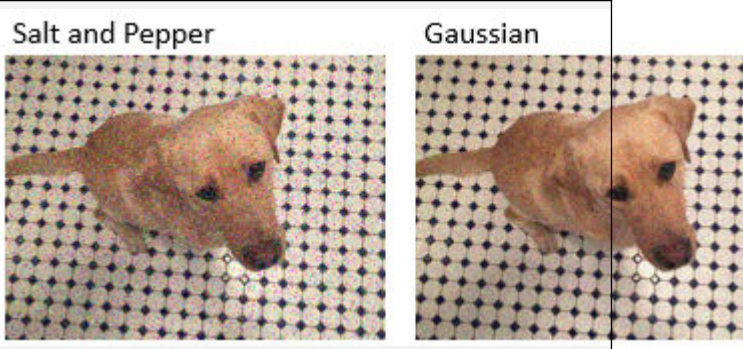
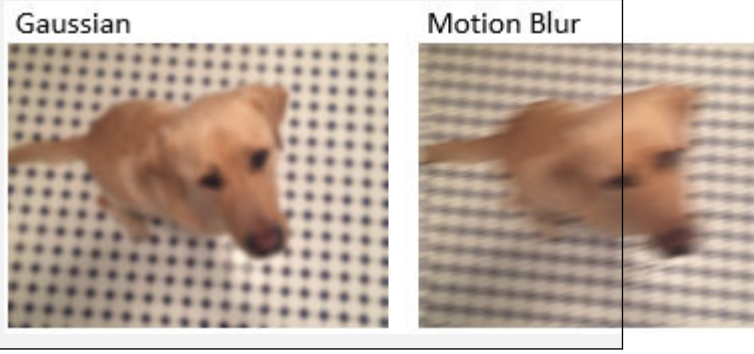
Common image preprocessing operations include noise removal, edge-preserving smoothing, color space conversion, contrast enhancement, and morphology.

If you have Image Processing Toolbox, then you can process data using these operations as well as any other functionality in the toolbox. For an example that shows how to create and apply these transformations, see “Augment Images for Deep Learning Workflows Using Image Processing Toolbox” on page 19-43.

Processing Type	Description	Sample Functions	Sample Output
Resize images	Resize images by a fixed scaling factor or to a target size	• `imresize`, `imresize3`	

Warp images	Apply random reflection, rotation, scale, shear, and translation to images	• `rand`
		`e2d`,
		`rand`
		`e3d`
			

Crop images	Crop an image to a target size from the center or a random position	• `centerCropWindow2d`, `centerCropWindow3d` • `randomWindow2d`, `randomCropWindow3d`	
Jitter color	Randomly adjust image hue, saturation, brightness, or contrast	• `jitt` `HSV`	

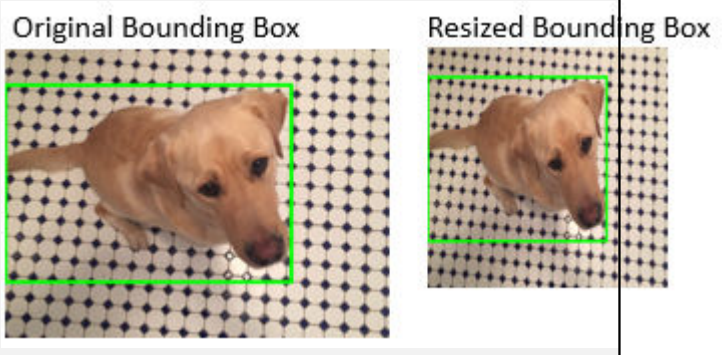
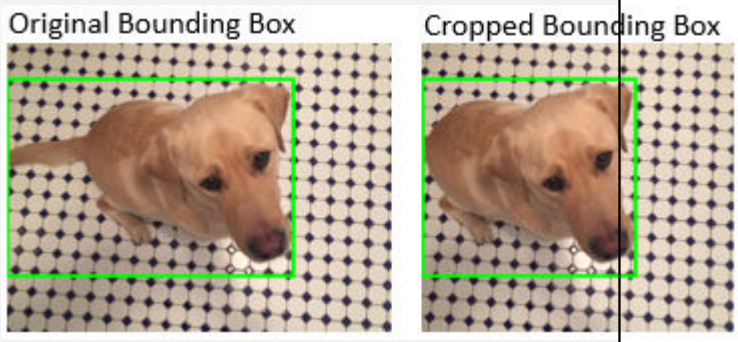
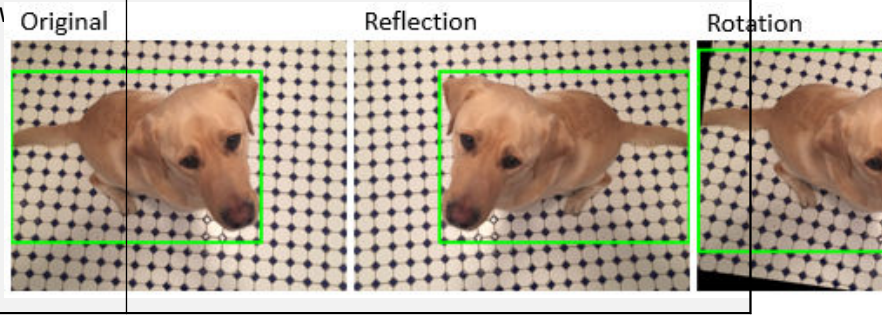
Processing Type	Description	Sample Functions	Sample Output
Simulate noise	Add random Gaussian, Poisson, salt and pepper, or multiplicative noise	• `imnoise`	
Simulate blur	Add Gaussian or directional motion blur	• `imgaussfilt`, `imgaussfilt3` • `imfilter`	

Object Detection

Object detection data consists of an image and bounding boxes that describe the location and characteristics of objects in the image.

If you have Computer Vision Toolbox, then you can use the **Image Labeler** and the **Video Labeler** apps to interactively label ROIs and export the label data for training a neural network. If you have Automated Driving Toolbox™, then you also use the **Ground Truth Labeler** app to create labeled ground truth training data.

When you transform an image, you must perform an identical transformation to the corresponding bounding boxes. If you have Computer Vision Toolbox, then you can process bounding box data using the operations in the table. For an example that shows how to create and apply these transformations, see “Augment Bounding Boxes for Object Detection” on page 19-77. For more information, see “Getting Started with Object Detection Using Deep Learning” (Computer Vision Toolbox).

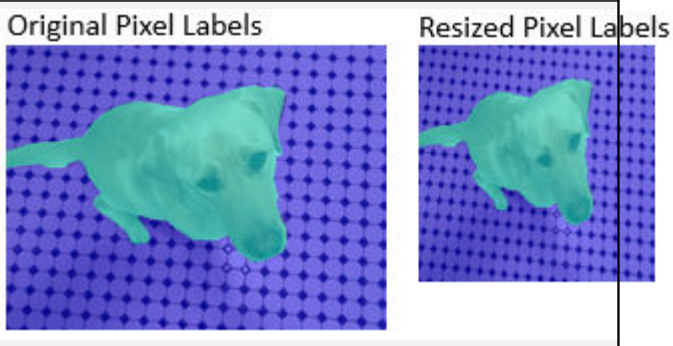
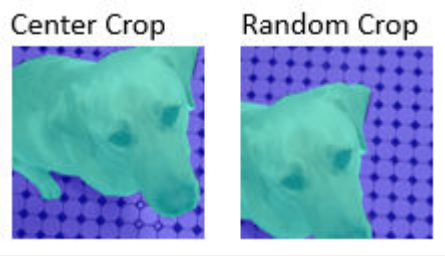
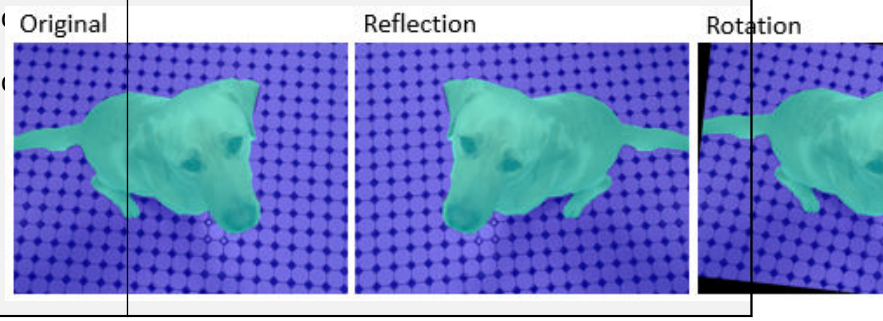
Processing Type	Description	Sample Functions	Sample Output
Resize bounding boxes	Resize bounding boxes by a fixed scaling factor or to a target size	• `bboxresize`	
Crop bounding boxes	Crop a bounding box to a target size from the center or a random position	• `bboxcrop`	
Warp bounding boxes	Apply reflection, rotation, scale, shear, and translation to bounding boxes	• `bbox`	

Semantic Segmentation

Semantic segmentation data consists of images and corresponding pixel labels represented as categorical arrays.

If you have Computer Vision Toolbox, then you can use the **Image Labeler** and the **Video Labeler** apps to interactively label pixels and export the label data for training a neural network. If you have Automated Driving Toolbox, then you also use the **Ground Truth Labeler** app to create labeled ground truth training data.

When you transform an image, you must perform an identical transformation to the corresponding pixel labeled image. If you have Image Processing Toolbox, then you can preprocess pixel label images using the functions in the table and any other toolbox function that supports categorical input. For an example that shows how to create and apply these transformations, see “Augment Pixel Labels for Semantic Segmentation” on page 19-67. For more information, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

Processing Type	Description	Sample Functions	Sample Output
Resize pixel labels	Resize pixel label images by a fixed scaling factor or to a target size	• `imresize`	
Crop pixel labels	Crop a pixel label image to a target size from the center or a random position	• `imcrop` • `centerCropWindow2d`, `centerCropWindow3d` • `randomWindow2d`, `randomCropWindow3d`	
Warp pixel labels	Apply random reflection, rotation, scale, shear, and translation to pixel label images	• `randomReflection2d`, `randomReflection3d`	

Signal Processing Applications

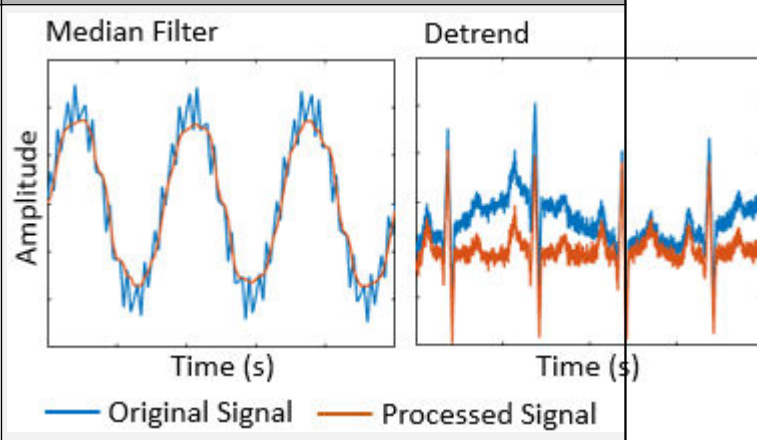
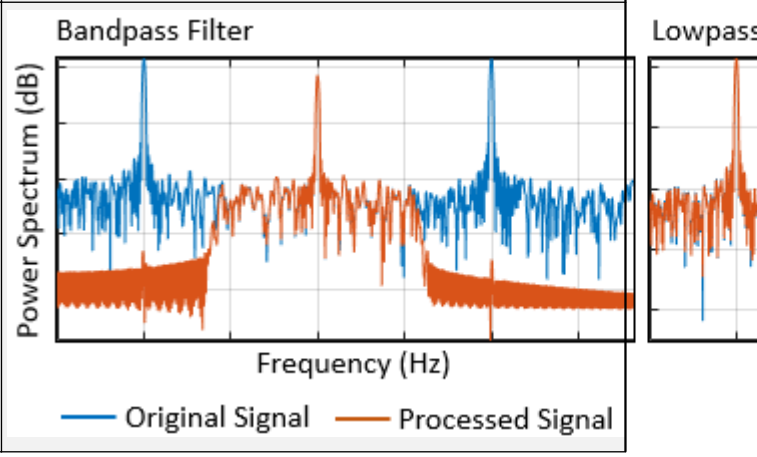
Signal Processing Toolbox™ enables you to denoise, smooth, detrend, and resample signals. You can augment training data with noise, multipath fading, and synthetic signals such as pulses and chirps. You can also create labeled sets of signals by using the **Signal Labeler** app and the `labeledSignalSet` object. For an example that shows how to create and apply these transformations, see “Waveform Segmentation Using Deep Learning” on page 12-41.

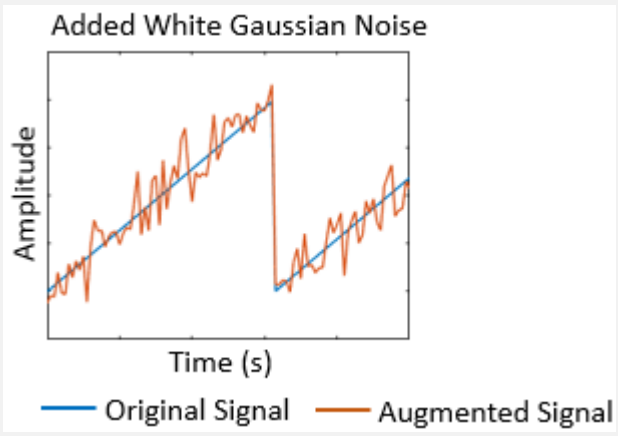
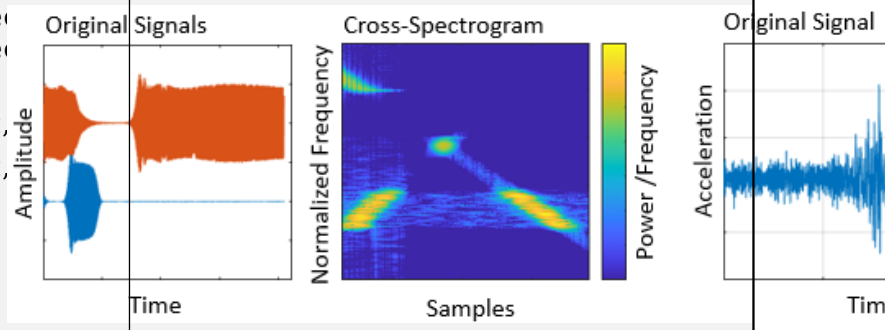
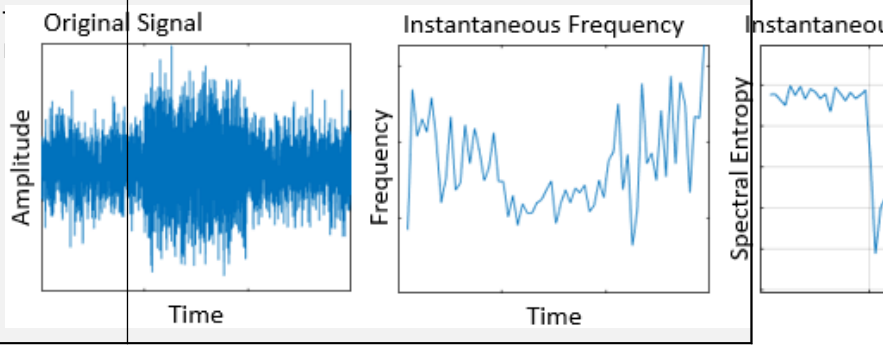
Wavelet Toolbox™ and Signal Processing Toolbox enable you to generate 2-D time-frequency representations of time series data that you can use as image inputs for signal classification applications. For an example, see “Classify Time Series Using Wavelet Analysis and Deep Learning” on page 12-94. Similarly, you can extract sequences from signal data to use as input for LSTM networks. For an example, see “Classify ECG Signals Using Long Short-Term Memory Networks” (Signal Processing Toolbox).

Communications Toolbox™ expands on signal processing functionality to enable you to perform error correction, interleaving, modulation, filtering, synchronization, and equalization of communication

systems. For an example that shows how to create and apply these transformations, see “Modulation Classification with Deep Learning” on page 13-38.

You can process signal data using the functions in the table as well as any other functionality in each toolbox.

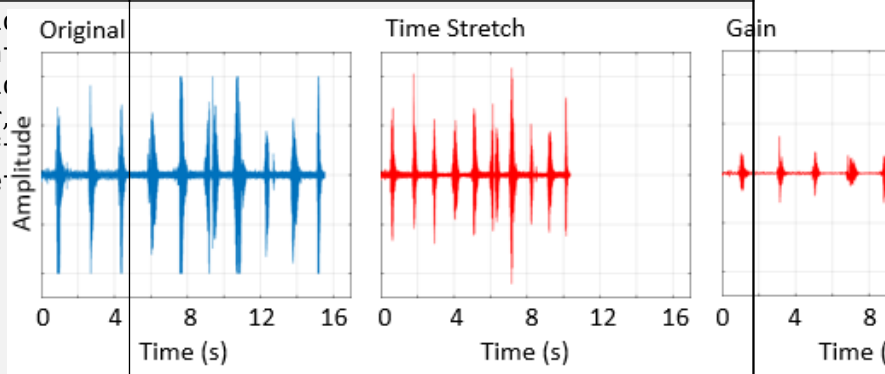
Processing Type	Description	Sample Functions	Sample Output
Clean signals	• Apply median filtering or moving average to signal • Remove polynomial trend • Resample signal to new fixed rate	• `medfilt1`, `smoothdata` • `detrend` • `downsample`, `interp`, `upsample`	
Filter signals	• Perform lowpass, highpass, and bandstop filtering of IIR and FIR signals • Design IIR and FIR filters • Apply IIR and FIR filters	• `bandpass`, `bandstop`, `highpass`, `lowpass` • `butter`, `designfilt`, `fir1`, `gaussdesign`, `rcosdesign` • `filter`	

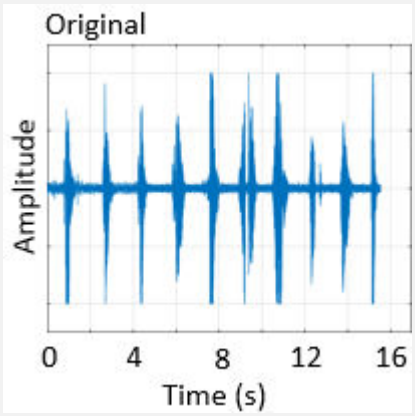
Processing Type	Description	Sample Functions	Sample Output
Augment signals	• Add white Gaussian noise to signal using Communications Toolbox • Adjust time information of the signal, and perform multipath fading using Communications Toolbox • Add synthetic chirps and waveforms	• `awgn` • `chirp`, `square`, `rectpuls`, `sawtooth`	
Create time-frequency representations	Create spectrograms, scalograms, and other 2-D representations of 1-D signals	• `pspe` `xspe` `m` • `fsst`, • `stft`, • `cwt`	
Extract features from signals	Estimate instantaneous frequency and spectral entropy	• `inst` `pent`	

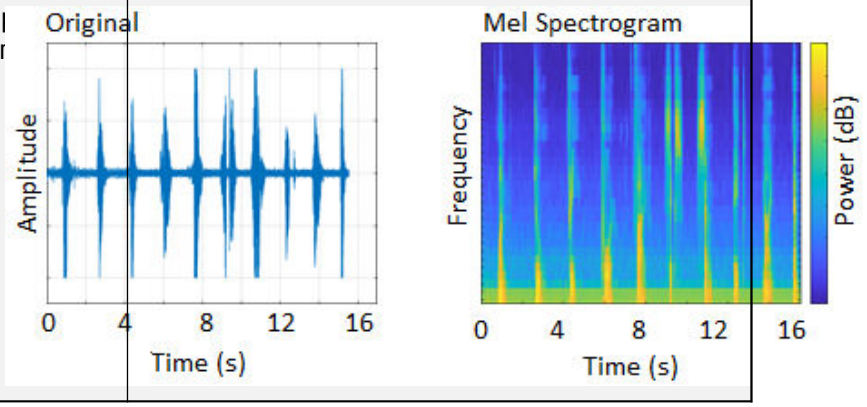
Audio Processing Applications

Audio Toolbox provides tools for audio processing, speech analysis, and acoustic measurement. Use these tools to extract auditory features and transform audio signals. Augment audio data with randomized or deterministic time scaling, time stretching, and pitch shifting. You can also create labeled ground truth training data by using the **Audio Labeler** app. You can process audio data using the functions in this table as well as any other functionality in the toolbox. For an example that shows how to create and apply these transformations, see “Augment Audio Dataset” (Audio Toolbox).

Processing Type	Description	Sample Functions	Sample Output

Augment audio data	Perform random or deterministic pitch shifting, time-scale modification, time shifting, noise addition, and volume control	• `audi`
		`gmen`
		`audi`
		`aler`,
		`shif`
		`stre`
		`o`
			

Extract audio features	Extract spectral parameters from audio segments	• `audioFeatur`
		`eExtractor`,
		`mfcc`
			
			Processed output:
			`ans = struct with fields:`
			`mfcc: [1 2 3 4 5 6 7 8 9 10 11 12 13`
			`mfccDelta: [14 15 16 17 18 19 20 21 22 23`
			`mfccDeltaDelta: [27 28 29 30 31 32 33 34 35 36`
			`spectralCentroid: 40`
			`pitch: 41`

Create time-frequency representations	Create mel spectrograms and other 2-D representations of audio signals	• `melS` `ram`,	

Text Analytics

Text Analytics Toolbox includes tools for processing raw text from sources such as equipment logs, news feeds, surveys, operator reports, and social media. Use these tools to extract text from popular file formats, preprocess raw text, extract individual words or multiword phrases (n-grams), convert text into numerical representations, and build statistical models. You can process text data using the functions in this table as well as any other functionality in the toolbox. For an example showing how to get started, see “Prepare Text Data for Analysis” (Text Analytics Toolbox).

Processing Type	Description	Sample Functions	Sample Output

Tokenize text	Parse text into words and punctuation	• `tokenizedDocument`	Original:
			`"A few tree limbs greater than 6 inches down on HWY 18 in Roseland."`
			Processed output:
			`15 tokens: A few tree limbs greater than 6 inches down on HWY 18 in Roseland .`

Clean text	• Remove variations in word forms and case
	• Remove punctuation
	• Remove stop words, short words, and long words
		• `normalizeWords`
		• `erasePunctuation`
		• `removeStopWords`, `removeShortWords`, `removeLongWords`
			Processed output:
			`15 tokens: a few tree limb great than 6 inch down on hwy 18 in roseland .`
			`14 tokens: a few tree limb great than 6 inch down on hwy 18 in roseland`
			`8 tokens: few tree limb great inch down hwy roseland`

See Also

`transform` | `combine` | `trainNetwork` | `trainingOptions` | `read`

More About

- “Datastores for Deep Learning” on page 19-2
- “Select Datastore for File Format or Application”

Develop Custom Mini-Batch Datastore

A *mini-batch datastore* is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications that use Deep Learning Toolbox.

To preprocess sequence, time series, or text data, build your own mini-batch datastore using the framework described here. For an example showing how to use a custom mini-batch datastore, see “Train Network Using Custom Mini-Batch Datastore for Sequence Data” on page 19-104.

Overview

Build your custom datastore interface using the custom datastore classes and objects. Then, use the custom datastore to bring your data into MATLAB.

Designing your custom mini-batch datastore involves inheriting from the `matlab.io.Datastore` and `matlab.io.datastore.MiniBatchable` classes, and implementing the required properties and methods. You optionally can add support for shuffling during training.

Processing Needs	Classes
Mini-batch datastore for training, validation, test, and prediction data sets in Deep Learning Toolbox	`matlab.io.Datastore` and `matlab.io.datastore.MiniBatchable` See “Implement MiniBatchable Datastore” on page 19-36.
Mini-batch datastore with support for shuffling during training	`matlab.io.Datastore`, `matlab.io.datastore.MiniBatchable`, and `matlab.io.datastore.Shuffleable` See “Add Support for Shuffling” on page 19-41.

Implement MiniBatchable Datastore

To implement a custom mini-batch datastore named `MyDatastore`, create a script `MyDatastore.m`. The script must be on the MATLAB path and should contain code that inherits from the appropriate class and defines the required methods. The code for creating a mini-batch datastore for training, validation, test, and prediction data sets in Deep Learning Toolbox must:

- Inherit from the classes `matlab.io.Datastore` and `matlab.io.datastore.MiniBatchable`.
- Define these properties: `MiniBatchSize` and `NumObservations`.
- Define these methods: `hasdata`, `read`, `reset`, and `progress`.

In addition to these steps, you can define any other properties or methods that you need to process and analyze your data.

Note If you are training a network and `trainingOptions` specifies `'Shuffle'` as `'once'` or `'every-epoch'`, then you must also inherit from the `matlab.io.datastore.Shuffleable` class. For more information, see “Add Support for Shuffling” on page 19-41.

The datastore read function must return data in a table. The table elements must be scalars, row vectors, or 1-by-1 cell arrays containing a numeric array.

For networks with a single input layer, the first and second columns specify the predictors and responses, respectively.

Tip To use a datastore for networks with multiple input layers, use the `combine` and `transform` functions to create a datastore that outputs a cell array with (`numInputs` + 1) columns, where `numInputs` is the number of network inputs. In this case, the first `numInputs` columns specify the predictors for each input and the last column specifies the responses. The order of inputs is given by the `InputNames` property of the layer graph `layers`.

The format of the predictors depend on the type of data.

Data	Format of Predictors
2-D image	*h*-by-*w*-by-*c* numeric array, where *h*, *w*, and *c* are the height, width, and number of channels of the image, respectively.
3-D image	*h*-by-*w*-by-*d*-by-*c* numeric array, where *h*, *w*, *d*, and *c* are the height, width, depth, and number of channels of the image, respectively.
Vector sequence	*c*-by-*s* matrix, where *c* is the number of features of the sequence and *s* is the sequence length.
1-D image sequence	*h*-by-*c*-by-*s* array, where *h* and *c* correspond to the height and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
2-D image sequence	*h*-by-*w*-by-*c*-by-*s* array, where *h*, *w*, and *c* correspond to the height, width, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
3-D image sequence	*h*-by-*w*-by-*d*-by-*c*-by-*s* array, where *h*, *w*, *d*, and *c* correspond to the height, width, depth, and number of channels of the image, respectively, and *s* is the sequence length. Each sequence in the mini-batch must have the same sequence length.
Features	*c*-by-1 column vector, where *c* is the number of features.

The table elements must contain a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

The `trainNetwork` function does not support networks with multiple sequence input layers.

The format of the responses depend on the type of task.

Task	Format of Responses
Classification	Categorical scalar
Regression	• Scalar • Numeric vector • 3-D numeric array representing an image
Sequence-to-sequence classification	1-by-*s* sequence of categorical labels, where *s* is the sequence length of the corresponding predictor sequence.
Sequence-to-sequence regression	*R*-by-*s* matrix, where *R* is the number of responses and *s* is the sequence length of the corresponding predictor sequence.

The table elements must contain a categorical scalar, a numeric scalar, a numeric row vector, or a 1-by-1 cell array containing a numeric array.

This example shows how to create a custom mini-batch datastore for processing sequence data. Save the script in a file called `MySequenceDatastore.m`.

Steps	Implementation

```
classdef MySequenceDatastore < matlab.io.Datastore & ...
                               matlab.io.datastore.MiniBatchable
```

1 Begin defining your class. Inherit from the base class `matlab.io.Datastore` and the `matlab.io.datastore.MiniBatchable` class.

2 Define properties.

- Redefine the `MiniBatchSize` and `NumObservations` properties. You optionally can assign additional property attributes to either property. For more information, see “Property Attributes”.
- You can also define properties unique to your custom mini-batch datastore.

```
    properties
        Datastore
        Labels
        NumClasses
        SequenceDimension
        MiniBatchSize
    end

    properties(SetAccess = protected)
        NumObservations
    end

    properties(Access = private)
        % This property is inherited from Datastore
        CurrentFileIndex
    end
```

3 Define methods.

- Implement the custom mini-batch datastore constructor.
- Implement the `hasdata` method.
- Implement the `read` method, which must return data as a table with the predictors in the first column and

```
    methods

        function ds = MySequenceDatastore(folder)
            % Construct a MySequenceDatastore object

            % Create a file datastore. The readSequence function is
            % defined following the class definition.
            fds = fileDatastore(folder, ...
                'ReadFcn',@readSequence, ...
                'IncludeSubfolders',true);
            ds.Datastore = fds;

            % Read labels from folder names
            numObservations = numel(fds.Files);
            for i = 1:numObservations
                file = fds.Files{i};
                filepath = fileparts(file);
                [~,label] = fileparts(filepath);
                labels{i,1} = label;
            end
            ds.Labels = categorical(labels);
            ds.NumClasses = numel(unique(labels));

            % Determine sequence dimension. When you define the LSTM
            % network architecture, you can use this property to
            % specify the input size of the sequenceInputLayer.
            X = preview(fds);
            ds.SequenceDimension = size(X,1);

            % Initialize datastore properties.
            ds.MiniBatchSize = 128;
            ds.NumObservations = numObservations;
            ds.CurrentFileIndex = 1;
        end

        function tf = hasdata(ds)
            % Return true if more data is available
            tf = ds.CurrentFileIndex + ds.MiniBatchSize - 1 ...
                <= ds.NumObservations;
        end

        function [data,info] = read(ds)
            % Read one mini-batch batch of data
            miniBatchSize = ds.MiniBatchSize;
            info = struct;

            for i = 1:miniBatchSize
                predictors{i,1} = read(ds.Datastore);
                responses(i,1) = ds.Labels(ds.CurrentFileIndex);
```

Steps	Implementation

responses in the second column.
 	
For sequence data, the sequences must be matrices of size *c*-by-*s*, where *c* is the number of features and *s* is sequence length. The value of *s* can vary between mini-batches.
• Implement the `reset` method.
• Implement the `progress` method.
• You can also define methods unique to your custom mini-batch datastore.
	`` ds.CurrentFileIndex = ds.CurrentFileIndex + 1;
	        end
	
	        data = preprocessData(ds,predictors,responses);
	    end
	
	    function data = preprocessData(ds,predictors,responses)
	        % data = preprocessData(ds,predictors,responses) preprocesses
	        % the data in predictors and responses and returns the table
	        % data
	
	        miniBatchSize = ds.MiniBatchSize;
	
	        % Pad data to length of longest sequence.
	        sequenceLengths = cellfun(@(X) size(X,2),predictors);
	        maxSequenceLength = max(sequenceLengths);
	        for i = 1:miniBatchSize
	            X = predictors{i};
	
	            % Pad sequence with zeros.
	            if size(X,2) < maxSequenceLength
	                X(:,maxSequenceLength) = 0;
	            end
	
	            predictors{i} = X;
	        end
	
	        % Return data as a table.
	        data = table(predictors,responses);
	    end
	
	    function reset(ds)
	        % Reset to the start of the data
	        reset(ds.Datastore);
	        ds.CurrentFileIndex = 1;
	    end
	
	end
	
	methods (Hidden = true)
	
	    function frac = progress(ds)
	        % Determine percentage of data read from datastore
	        frac = (ds.CurrentFileIndex - 1) / ds.NumObservations;
	    end
	
	end
	
	end % end class definition```

4 End the `classdef` section.	The implementation of the read method of your custom datastore uses a function called `readSequence`. You must create this function to read sequence data from a MAT-file.
	
	``function data = readSequence(filename)
	% data = readSequence(filename) reads the sequence X from the MAT-file
	% filename
	
	S = load(filename);
	data = S.X;
	end```

Add Support for Shuffling

To add support for shuffling, first follow the instructions in “Implement MiniBatchable Datastore” on page 19-36 and then update your implementation code in `MySequenceDatastore.m` to:

- Inherit from an additional class `matlab.io.datastore.Shuffleable`.
- Define the additional method `shuffle`.

This example code adds shuffling support to the `MySequenceDatastore` class. Vertical ellipses indicate where you should copy code from the `MySequenceDatastore` implementation.

Steps	Implementation

1 Update the class definition to also inherit from the `matlab.io.datastore.Shuffleable` class.	`classdef MySequenceDatastore < matlab.io.Datastore & ...`
	`  matlab.io.datastore.MiniBatchable & ...`
	`  matlab.io.datastore.Shuffleable`
	
	`  % previously defined properties`
	`  .`
	`  .`
	`  .`

2 Add the definition for `shuffle` to the existing `methods` section.	`  methods`
	
	`  % previously defined methods`
	`  .`
	`  .`
	`  .`
	
	`  function dsNew = shuffle(ds)`
	`  % dsNew = shuffle(ds) shuffles the files and the`
	`  % corresponding labels in the datastore.`
	
	`  % Create a copy of datastore`
	`  dsNew = copy(ds);`
	`  dsNew.Datastore = copy(ds.Datastore);`
	`  fds = dsNew.Datastore;`
	
	`  % Shuffle files and corresponding labels`
	`  numObservations = dsNew.NumObservations;`
	`  idx = randperm(numObservations);`
	`  fds.Files = fds.Files(idx);`
	`  dsNew.Labels = dsNew.Labels(idx);`
	`  end`
	
	`  end`
	
	`end`

Validate Custom Mini-Batch Datastore

If you have followed all the instructions presented here, then the implementation of your custom mini-batch datastore is complete. Before using this datastore, qualify it using the guidelines presented in “Testing Guidelines for Custom Datastores”.

See Also

`trainNetwork`

Related Examples

- “Train Network Using Custom Mini-Batch Datastore for Sequence Data” on page 19-104

More About

- “Datastores for Deep Learning” on page 19-2
- “Getting Started with Datastore”
- “Develop Custom Datastore”
- “Developing Classes That Work Together”
- “Testing Guidelines for Custom Datastores”
- “Deep Learning in MATLAB” on page 1-2

Augment Images for Deep Learning Workflows Using Image Processing Toolbox

This example shows how MATLAB® and Image Processing Toolbox™ can perform common kinds of image augmentation as part of deep learning workflows.

Image Processing Toolbox functions enable you to implement common styles of image augmentation. This example demonstrates five common types of transformations:

- Random Image Warping Transformations on page 19-0
- Cropping Transformations on page 19-0
- Color Transformations on page 19-0
- Synthetic Noise on page 19-0
- Synthetic Blur on page 19-0

The example then shows how to apply augmentation to image data in datastores on page 19-0 using a combination of multiple types of transformations.

You can use augmented training data to train a network. For an example of training a network using augmented images, see “Prepare Datastore for Image-to-Image Regression” on page 19-91.

Read and display a sample image. To compare the effect of the different types of image augmentation, each transformation uses the same input image.

```
imOriginal = imread('kobi.png');
imshow(imOriginal)
```

Random Image Warping Transformations

The `randomAffine2d` (Image Processing Toolbox) function creates a randomized 2-D affine transformation from a combination of rotation, translation, scale (resizing), reflection, and shear. You can specify which transformations to include and the range of transformation parameters. If you specify the range as a 2-element numeric vector, then `randomAffine2d` selects the value of a parameter from a uniform probability distribution over the specified interval. For more control of the range of parameter values, you can specify the range using a function handle.

Control the spatial bounds and resolution of the warped image created by `imwarp` (Image Processing Toolbox) by using the `affineOutputView` (Image Processing Toolbox) function.

Rotation

Create a randomized rotation transformation that rotates the input image by an angle selected randomly from the range [-45, 45] degrees.

```
tform = randomAffine2d('Rotation',[-45 45]);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Translation

Create a translation transformation that shifts the input image horizontally and vertically by a distance selected randomly from the range [-50, 50] pixels.

```
tform = randomAffine2d('XTranslation',[-50 50],'YTranslation',[-50 50]);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Scale

Create a scale transformation that resizes the input image using a scale factor selected randomly from the range [1.2, 1.5]. This transformation resizes the image by the same factor in the horizontal and vertical directions.

```
tform = randomAffine2d('Scale',[1.2,1.5]);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Reflection

Create a reflection transformation that flips the input image with 50% probability in each dimension.

```
tform = randomAffine2d('XReflection',true,'YReflection',true);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Shear

Create a horizontal shear transformation with the shear angle selected randomly from the range [-30, 30].

```
tform = randomAffine2d('XShear',[-30 30]);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Control Range of Transformation Parameters Using Custom Selection Function

In the preceding transformations, the range of transformation parameters was specified by two-element numeric vectors. For more control of the range of the transformation parameters, specify a function handle instead of a numeric vector. The function handle takes no input arguments and yields a valid value for each parameter.

For example, this code selects a rotation angle from a discrete set of 90 degree rotation angles.

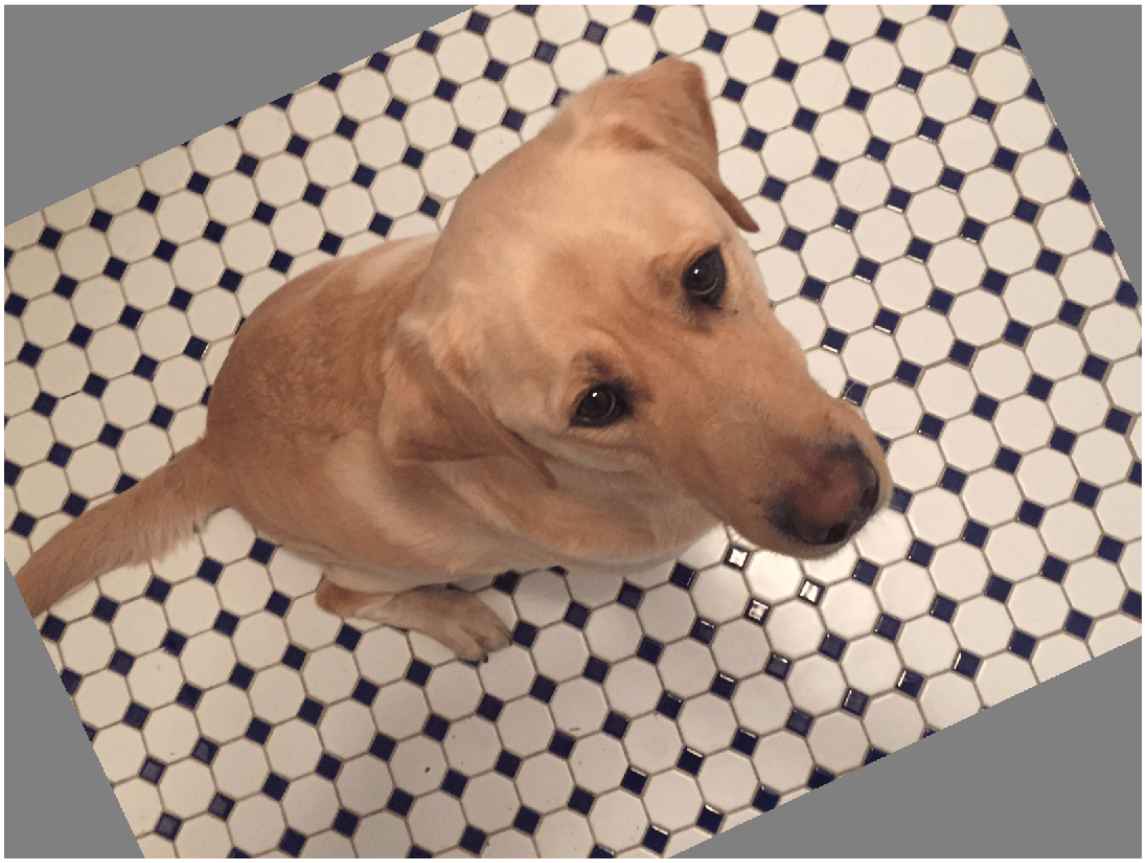
```
angles = 0:90:270;
tform = randomAffine2d('Rotation',@() angles(randi(4)));
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView);
imshow(imAugmented)
```

Control Fill Value

When you warp an image using a geometric transformation, pixels in the output image can map to a location outside the bounds of the input image. In that case, `imwarp` assigns a fill value to those pixels in the output image. By default, `imwarp` selects black as the fill value. You can change the fill value by specifying the `'FillValues'` name-value pair argument.

Create a random rotation transformation, then apply the transformation and specify a gray fill value.

```
tform = randomAffine2d('Rotation',[-45 45]);
outputView = affineOutputView(size(imOriginal),tform);
imAugmented = imwarp(imOriginal,tform,'OutputView',outputView,'FillValues',[128 128 128]);
imshow(imAugmented)
```



Cropping Transformations

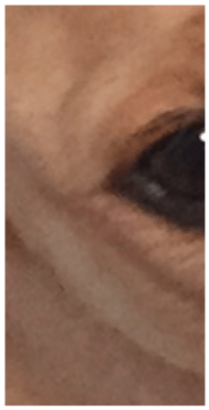
To create output images of a desired size, use the `randomWindow2d` (Image Processing Toolbox) and `centerCropWindow2d` (Image Processing Toolbox) functions. Be careful to select a window that includes the desired content in the image.

Specify the desired size of the cropped region as a 2-element vector of the form [*height, width*].

```
targetSize = [200,100];
```

Crop the image to the target size from the center of the image.

```
win = centerCropWindow2d(size(imOriginal),targetSize);
imCenterCrop = imcrop(imOriginal,win);
imshow(imCenterCrop)
```



Crop the image to the target size from a random location in the image.

```
win = randomWindow2d(size(imOriginal),targetSize);
imRandomCrop = imcrop(imOriginal,win);
imshow(imRandomCrop)
```

Color Transformations

You can randomly adjust the hue, saturation, brightness, and contrast of a color image by using the `jitterColorHSV` (Image Processing Toolbox) function. You can specify which color transformations are included and the range of transformation parameters.

You can randomly adjust the brightness and contrast of grayscale images by using basic math operations.

Hue Jitter

Hue specifies the shade of color, or a color's position on a color wheel. As hue varies from 0 to 1, colors vary from red through yellow, green, cyan, blue, purple, magenta, and back to red. Hue jitter shifts the apparent shade of colors in an image.

Adjust the hue of the input image by a small positive offset selected randomly from the range [0.05, 0.15]. Colors that were red now appear more orange or yellow, colors that were orange appear yellow or green, and so on.

```
imJittered = jitterColorHSV(imOriginal,'Hue',[0.05 0.15]);
montage({imOriginal,imJittered})
```

Saturation Jitter

Saturation is the purity of color. As saturation varies from 0 to 1, hues vary from gray (indicating a mixture of all colors) to a single pure color. Saturation jitter shifts how dull or vibrant colors are.

Adjust the saturation of the input image by an offset selected randomly from the range [-0.4, -0.1]. The colors in the output image appear more muted, as expected when the saturation decreases.

```
imJittered = jitterColorHSV(imOriginal,'Saturation',[-0.4 -0.1]);
montage({imOriginal,imJittered})
```

Brightness Jitter

Brightness is the amount of hue. As brightness varies from 0 to 1, colors go from black to white. Brightness jitter shifts the darkness and lightness of an input image.

Adjust the brightness of the input image by an offset selected randomly from the range [-0.3, -0.1]. The image appears darker, as expected when the brightness decreases.

```
imJittered = jitterColorHSV(imOriginal,'Brightness',[-0.3 -0.1]);
montage({imOriginal,imJittered})
```

Contrast Jitter

Contrast jitter randomly adjusts the difference between the darkest and brightest regions in an input image.

Adjust the contrast of the input image by a scale factor selected randomly from the range [1.2, 1.4]. The contrast increases, such that shadows become darker and highlights become brighter.

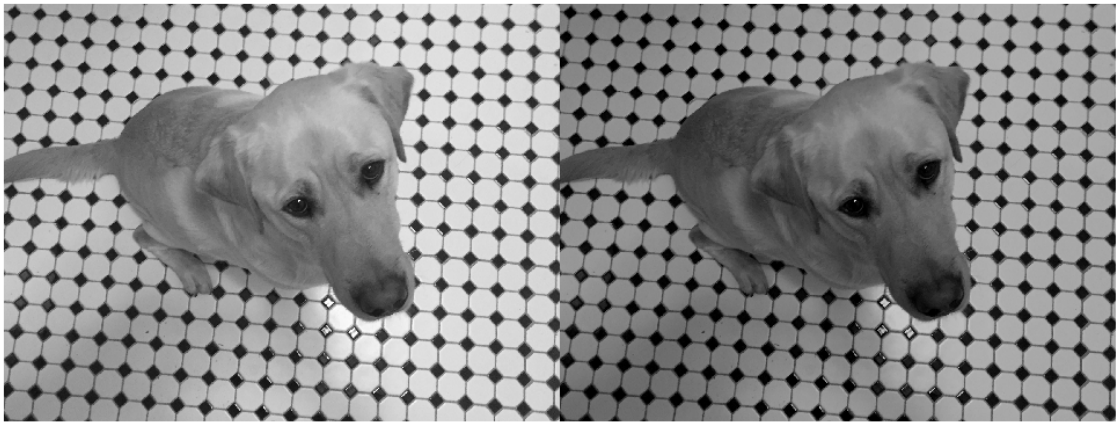
```
imJittered = jitterColorHSV(imOriginal,'Contrast',[1.2 1.4]);
montage({imOriginal,imJittered})
```

Brightness and Contrast Jitter of Grayscale Images

You can apply randomized brightness and contrast jitter to grayscale images by using basic math operations.

Convert the sample image to grayscale. Specify a random contrast scale factor in the range [0.8, 1] and a random brightness offset in the range [-0.15, 0.15]. Multiply the image by the contrast scale factor, then add the brightness offset.

```
imGray = rgb2gray(im2double(imOriginal));
contrastFactor = 1-0.2*rand;
brightnessOffset = 0.3*(rand-0.5);
imJittered = imGray.*contrastFactor + brightnessOffset;
imJittered = im2uint8(imJittered);
montage({imGray,imJittered})
```



Randomized Color-to-Grayscale

One type of color augmentation randomly drops the color information from an RGB image while preserving the number of channels expected by the network. This code shows a "random grayscale" transformation in which an RGB image is randomly converted with 80% probability to a three channel output image where R == G == B.

```
desiredProbability = 0.8;
if rand <= desiredProbability
    imJittered = repmat(rgb2gray(imOriginal),[1 1 3]);
end
imshow(imJittered)
```

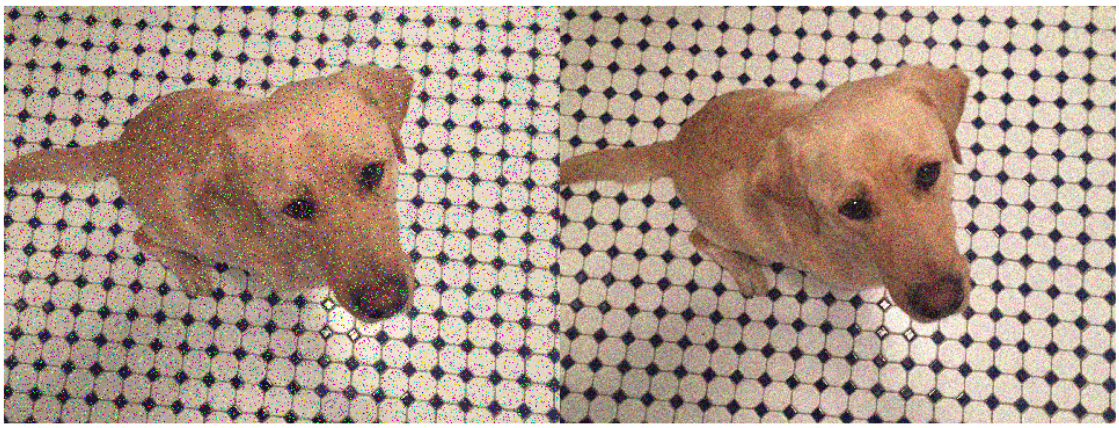
Other Image Processing Operations

Use the `transform` function to apply any combination of Image Processing Toolbox functions to input images. Adding noise and blur are two common image processing operations used in deep learning applications.

Synthetic Noise

To apply synthetic noise to an input image, use the `imnoise` (Image Processing Toolbox) function. You can specify which noise model to use, such as Gaussian, Poisson, salt and pepper, and multiplicative noise. You can also specify the strength of the noise.

```
imSaltAndPepperNoise = imnoise(imOriginal,'salt & pepper',0.1);
imGaussianNoise = imnoise(imOriginal,'gaussian');
montage({imSaltAndPepperNoise,imGaussianNoise})
```



Synthetic Blur

To apply randomized Gaussian blur to an image, use the `imgaussfilt` (Image Processing Toolbox) function. You can specify the amount of smoothing.

```
sigma = 1+5*rand;
imBlurred = imgaussfilt(imOriginal,sigma);
imshow(imBlurred)
```

Apply Augmentation to Image Data in Datastores

In practical deep learning problems, the image augmentation pipeline typically combines multiple operations. Datastores are a convenient way to read and augment collections of images.

This section of the example shows how to define data augmentation pipelines that augment datastores in the context of training image classification and image regression problems.

First, create an `imageDatastore` that contains unprocessed images. The image datastore in this example contains digit images with labels.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
imds.ReadSize = 6;
```

Image Classification

In image classification, the classifier should learn that a randomly altered version of an image still represents the same image class. To augment data for image classification, it is sufficient to augment the input images while leaving the corresponding categorical labels unchanged.

Augment images in the pristine image datastore with random Gaussian blur, salt and pepper noise, and randomized scale and rotation. These operations are defined in the helper function `classificationAugmentationPipeline` at the end of this example. Apply data augmentation to the training data by using the `transform` function.

```
dsTrain = transform(imds,@classificationAugmentationPipeline,'IncludeInfo',true);
```

Visualize a sample of the output coming from the augmented pipeline.

```
dataPreview = preview(dsTrain);
montage(dataPreview(:,1))
title("Augmented Images for Image Classification")
```

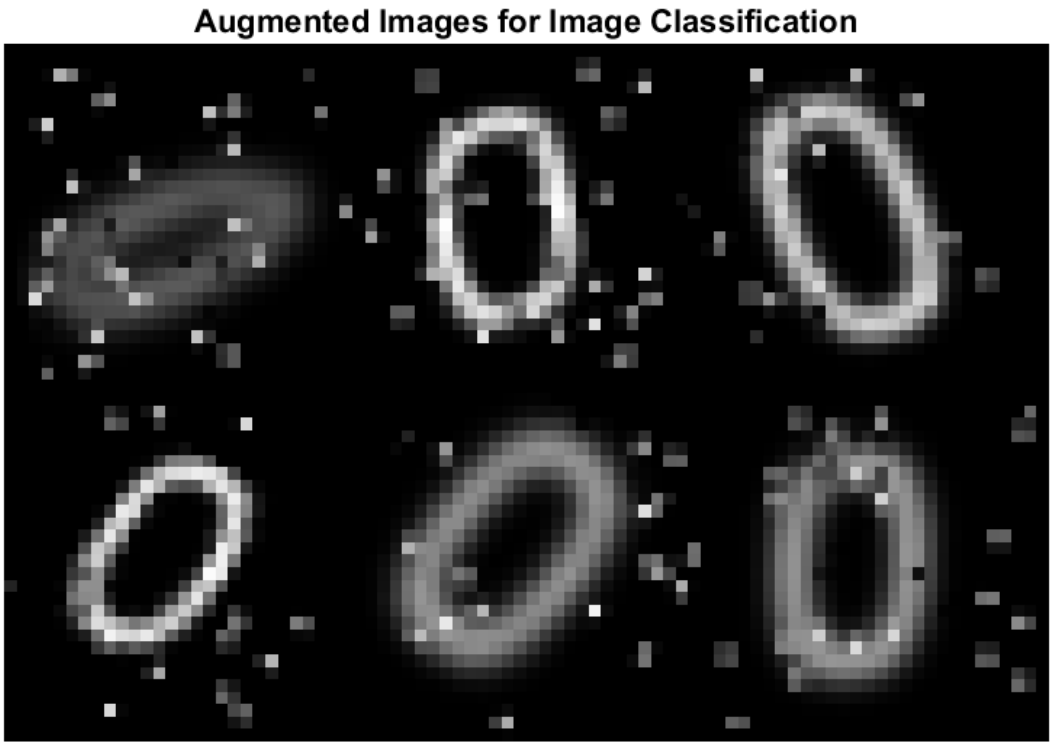
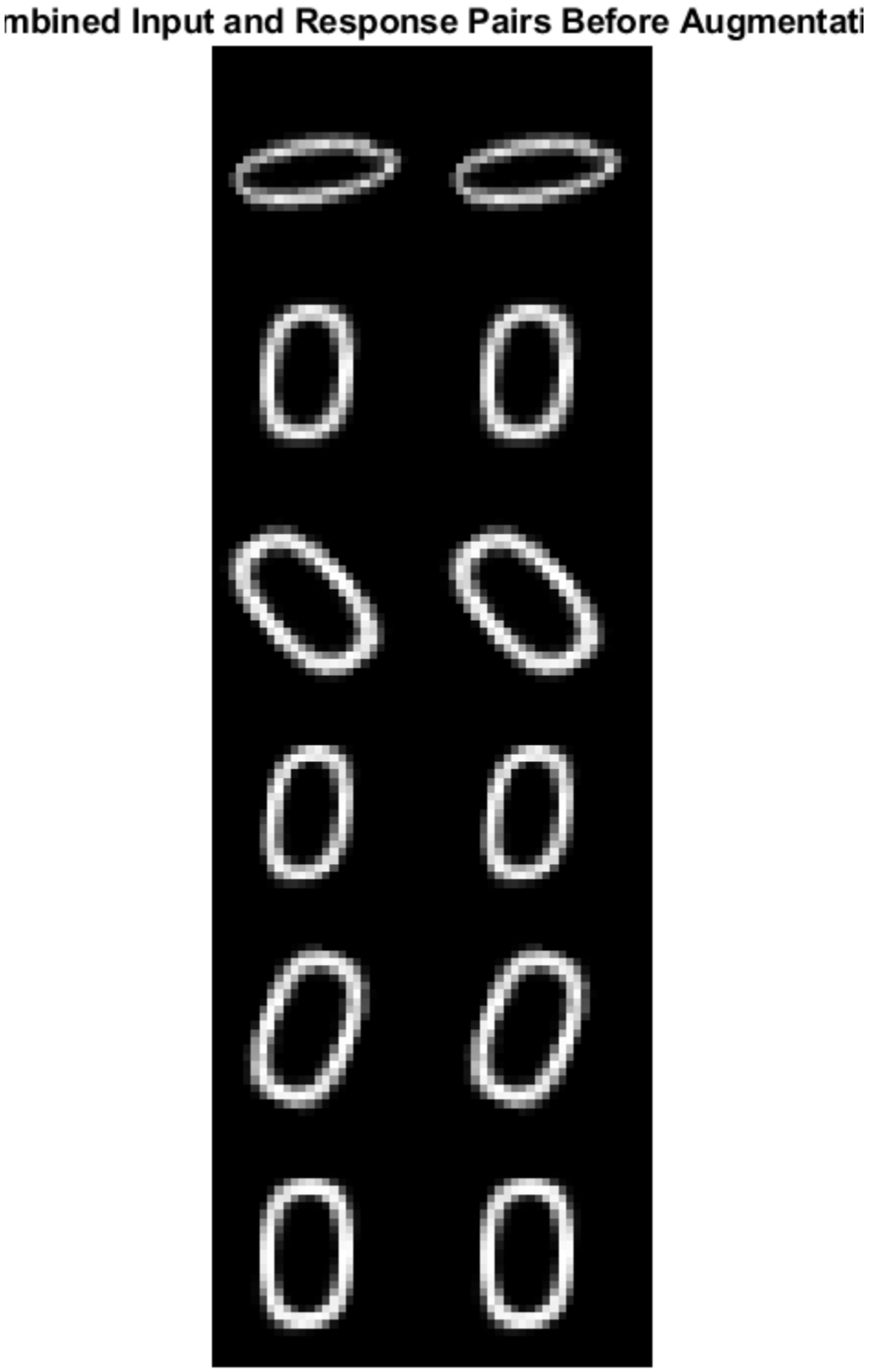


Image Regression

Image augmentation for image-to-image regression is more complicated because you must apply identical geometric transformations to the input and response images. Associate pairs of input and response images by using the `combine` function. Transform one or both images in each pair by using the `transform` function.

Combine two identical copies of the image datastore `imds`. When data is read from the combined datastore, image data is returned in a two-column cell array, where the first column represents network input images and the second column contains network responses.

```
dsCombined = combine(imds,imds);
montage(preview(dsCombined)','Size',[6 2])
title("Combined Input and Response Pairs Before Augmentation")
```

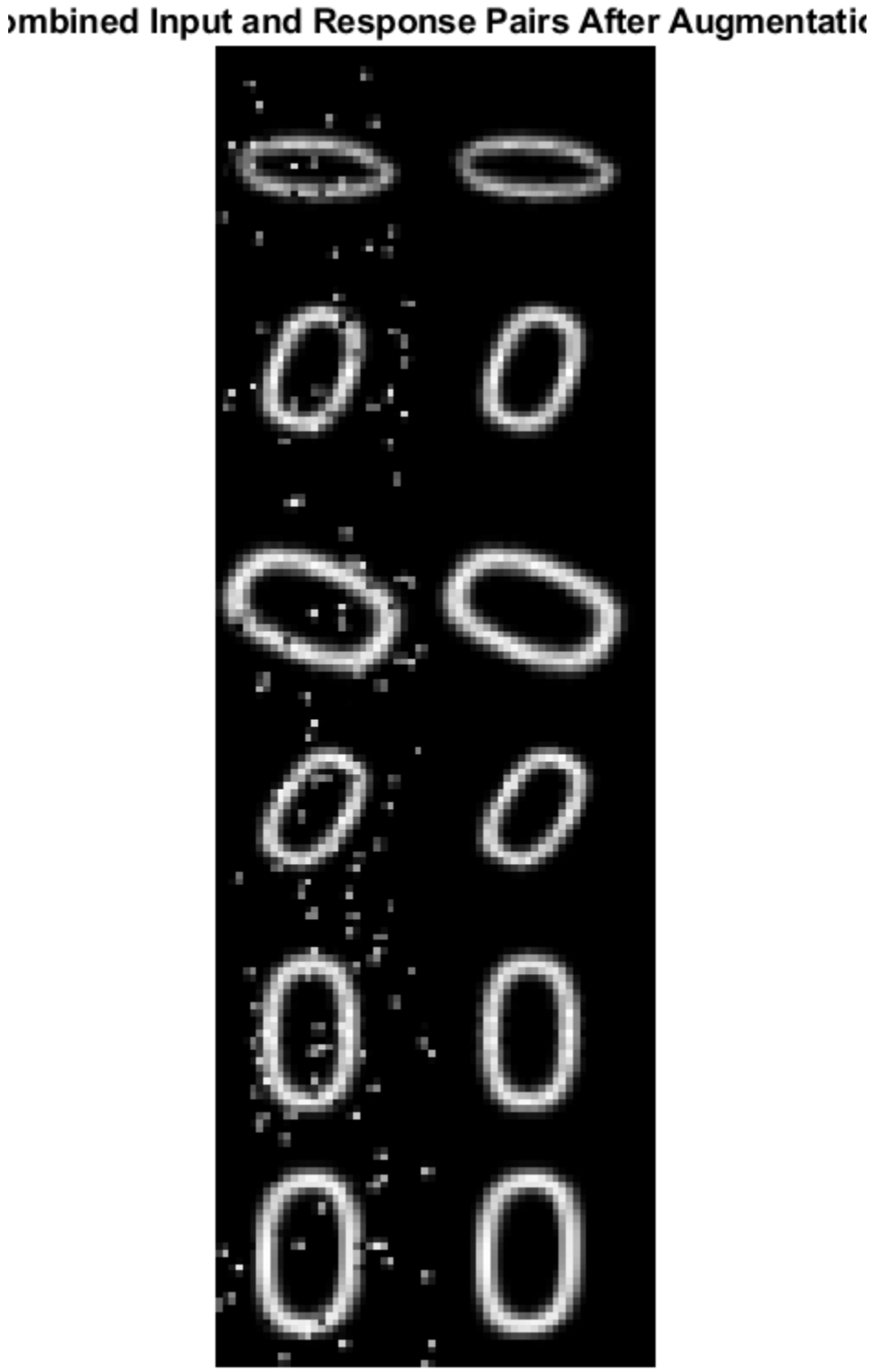

mbined Input and Response Pairs Before Augmentati

Augment each pair of training images with a series of image processing operations:

- Resize the input and response image to 32-by-32 pixels.
- Add salt and pepper noise to the input image only.
- Create a transformation that has randomized scale and rotation.
- Apply the same transformation to the input and response image.

These operations are defined in the helper function `imageRegressionAugmentationPipeline` at the end of this example. Apply data augmentation to the training data by using the `transform` function.

```
dsTrain = transform(dsCombined,@imageRegressionAugmentationPipeline);
montage(preview(dsTrain)','Size',[6 2])
title("Combined Input and Response Pairs After Augmentation")
```


ombined Input and Response Pairs After Augmentatic

For a complete example that includes training and evaluating an image-to-image regression network, see “Prepare Datastore for Image-to-Image Regression” on page 19-91.

Supporting Functions

The `classificationAugmentationPipeline` helper function augments images for classification. `dataIn` and `dataOut` are two-element cell arrays, where the first element is the network input image and the second element is the categorical label.

```
function [dataOut,info] = classificationAugmentationPipeline(dataIn,info)

dataOut = cell([size(dataIn,1),2]);

for idx = 1:size(dataIn,1)
    temp = dataIn{idx};

    % Add randomized Gaussian blur
    temp = imgaussfilt(temp,1.5*rand);

    % Add salt and pepper noise
    temp = imnoise(temp,'salt & pepper');

    % Add randomized rotation and scale
    tform = randomAffine2d('Scale',[0.95,1.05],'Rotation',[-30 30]);
    outputView = affineOutputView(size(temp),tform);
    temp = imwarp(temp,tform,'OutputView',outputView);

    % Form second column expected by trainNetwork which is the expected response,
    % the categorical label in this case
    dataOut(idx,:) = {temp,info.Label(idx)};
end

end
```

The `imageRegressionAugmentationPipeline` helper function augments images for image-to-image regression. `dataIn` and `dataOut` are two-element cell arrays, where the first element is the network input image and the second element is the network response image.

```
function dataOut = imageRegressionAugmentationPipeline(dataIn)

dataOut = cell([size(dataIn,1),2]);
for idx = 1:size(dataIn,1)

    % Resize images to 32-by-32 pixels and convert to data type single
    inputImage = im2single(imresize(dataIn{idx,1},[32 32]));
    targetImage = im2single(imresize(dataIn{idx,2},[32 32]));

    % Add salt and pepper noise
    inputImage = imnoise(inputImage,'salt & pepper');

    % Add randomized rotation and scale
    tform = randomAffine2d('Scale',[0.9,1.1],'Rotation',[-30 30]);
    outputView = affineOutputView(size(inputImage),tform);

    % Use imwarp with the same tform and outputView to augment both images
    % the same way
    inputImage = imwarp(inputImage,tform,'OutputView',outputView);
    targetImage = imwarp(targetImage,tform,'OutputView',outputView);
```

```
    dataOut(idx,:) = {inputImage,targetImage};
end

end
```

See Also

`transform` | `combine`

Related Examples

- “Prepare Datastore for Image-to-Image Regression” on page 19-91

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 19-27
- “Preprocess Images for Deep Learning” on page 19-16

Augment Pixel Labels for Semantic Segmentation

This example shows how to perform common kinds of image and pixel label augmentation as part of semantic segmentation workflows.

Semantic segmentation training data consists of images represented by numeric matrices and pixel label images represented by categorical matrices. When you augment training data, you must apply identical transformations to the image and associated pixel labels. This example demonstrates three common types of transformations:

- Resize Image and Pixel Labels on page 19-0
- Crop Image and Pixel Labels on page 19-0
- Warp Image and Pixel Labels on page 19-0

The example then shows how to apply augmentation to semantic segmentation training data in datastores on page 19-0 using a combination of multiple types of transformations.

You can use augmented training data to train a network. For an example showing how to train a semantic segmentation network, see “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

To demonstrate the effects of the different types of augmentation, each transformation in this example uses the same input image and pixel label image.

Read a sample image.

```
filenameImage = 'kobi.png';
I = imread(filenameImage);
```

Read the pixel label image. The image has two classes.

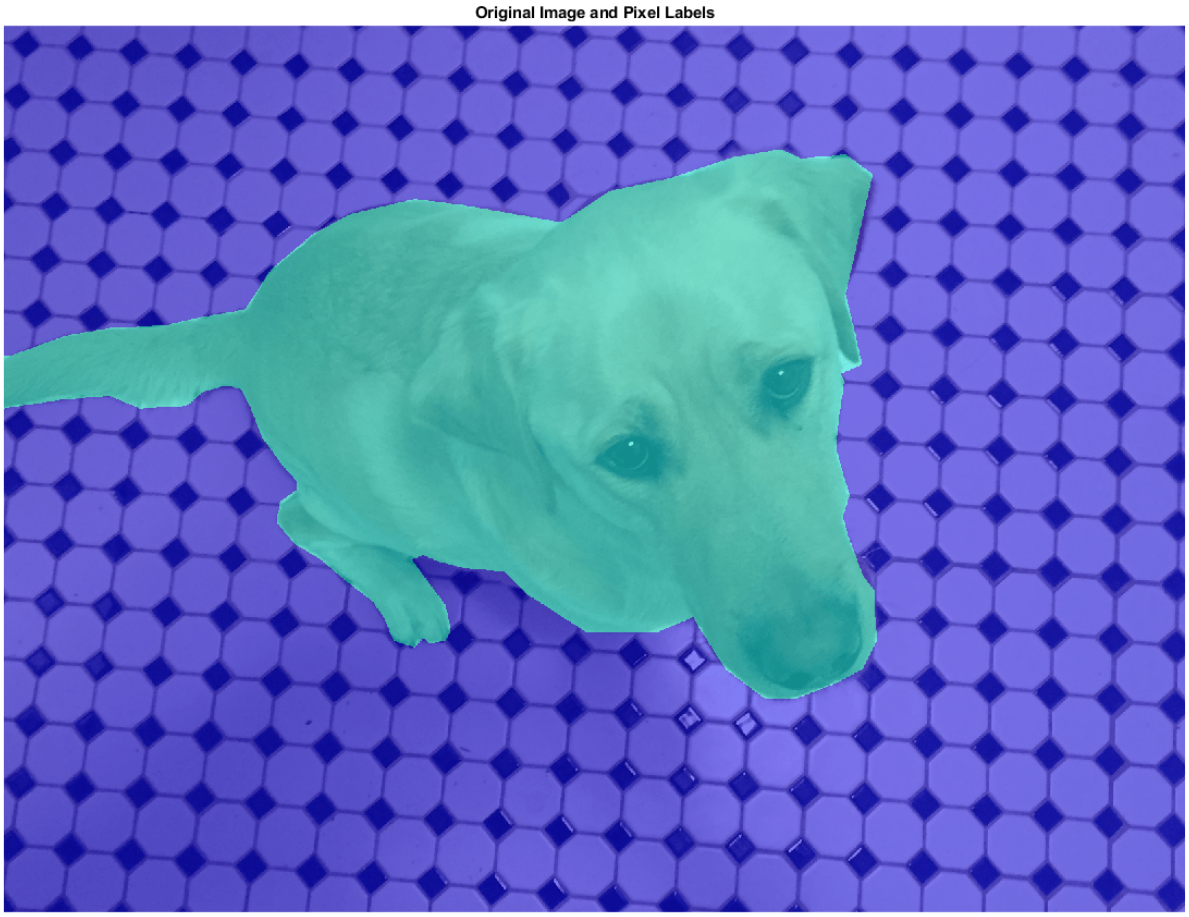
```
filenameLabels = 'kobiPixelLabeled.png';
L = imread(filenameLabels);
classes = ["floor","dog"];
ids = [1 2];
```

Convert the pixel label image to the categorical data type.

```
C = categorical(L,ids,classes);
```

Display the labels over the image by using the `labeloverlay` function. Pixels with the label "floor" have a blue tint and pixels with the label "dog" have a cyan tint.

```
B = labeloverlay(I,C);
imshow(B)
title('Original Image and Pixel Labels')
```



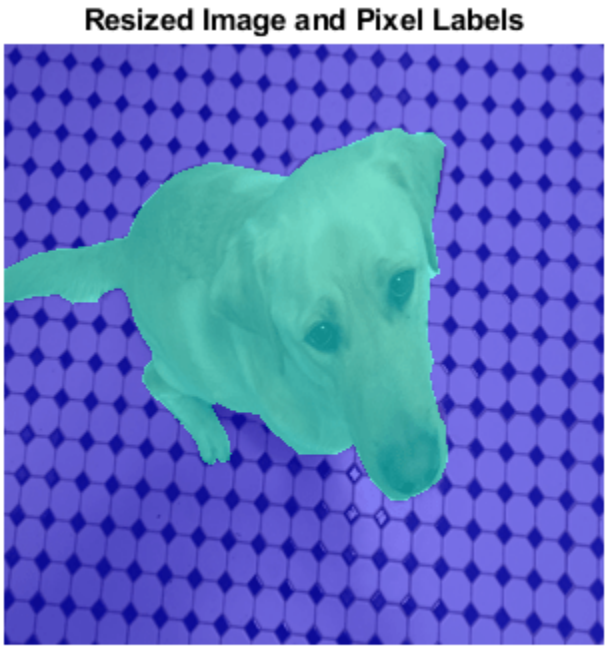
Resize Image and Pixel Labels

You can resize numeric and categorical images by using the `imresize` function. Resize the image and the pixel label image to the same size, and display the labels over the image.

```
targetSize = [300 300];
resizedI = imresize(I,targetSize);
resizedC = imresize(C,targetSize);
```

Display the resized labels over the resized image.

```
B = labeloverlay(resizedI,resizedC);
imshow(B)
title('Resized Image and Pixel Labels')
```



Crop Image and Pixel Labels

Cropping is a common preprocessing step to make the data match the input size of the network. To create output images of a desired size, first specify the size and position of the crop window by using the `randomWindow2d` (Image Processing Toolbox) and `centerCropWindow2d` (Image Processing Toolbox) functions. Make sure you select a cropping window that includes the desired content in the image. Then, crop the image and pixel label image to the same window by using `imcrop`.

Specify the desired size of the cropped region as a two-element vector of the form [*height, width*].

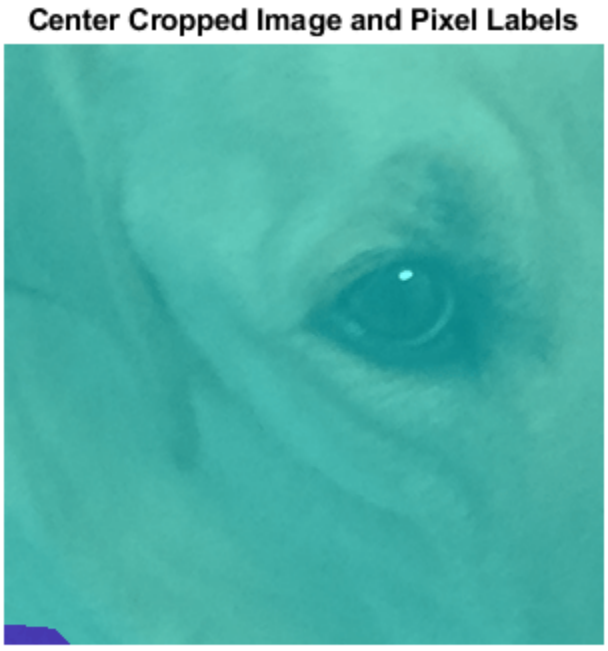
```
targetSize = [300 300];
```

Crop the image to the target size from the center of the image.

```
win = centerCropWindow2d(size(I),targetSize);
croppedI = imcrop(I,win);
croppedC = imcrop(C,win);
```

Display the cropped labels over the cropped image.

```
B = labeloverlay(croppedI,croppedC);
imshow(B)
title('Center Cropped Image and Pixel Labels')
```

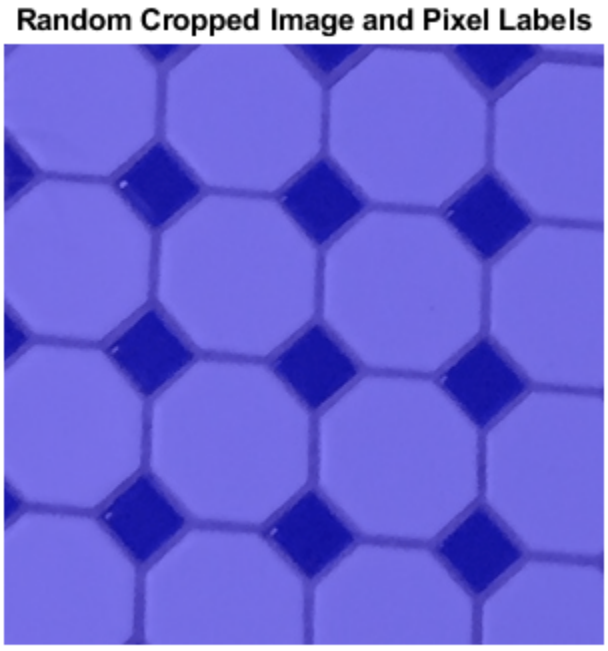


Crop the image to the target size from a random position in the image.

```
win = randomWindow2d(size(I),targetSize);
croppedI = imcrop(I,win);
croppedC = imcrop(C,win);
```

Display the cropped labels over the cropped image.

```
B = labeloverlay(croppedI,croppedC);
imshow(B)
title('Random Cropped Image and Pixel Labels')
```



Warp Image and Pixel Labels

The `randomAffine2d` (Image Processing Toolbox) function creates a randomized 2-D affine transformation from a combination of rotation, translation, scaling (resizing), reflection, and shearing. Apply the transformation to images and pixel label images by using `imwarp` (Image Processing Toolbox). Control the spatial bounds and resolution of the warped output by using the `affineOutputView` (Image Processing Toolbox) function.

Rotate the input image and pixel label image by an angle selected randomly from the range [-50,50] degrees.

```
tform = randomAffine2d("Rotation",[-50 50]);
```

Create an output view for the warped image and pixel label image.

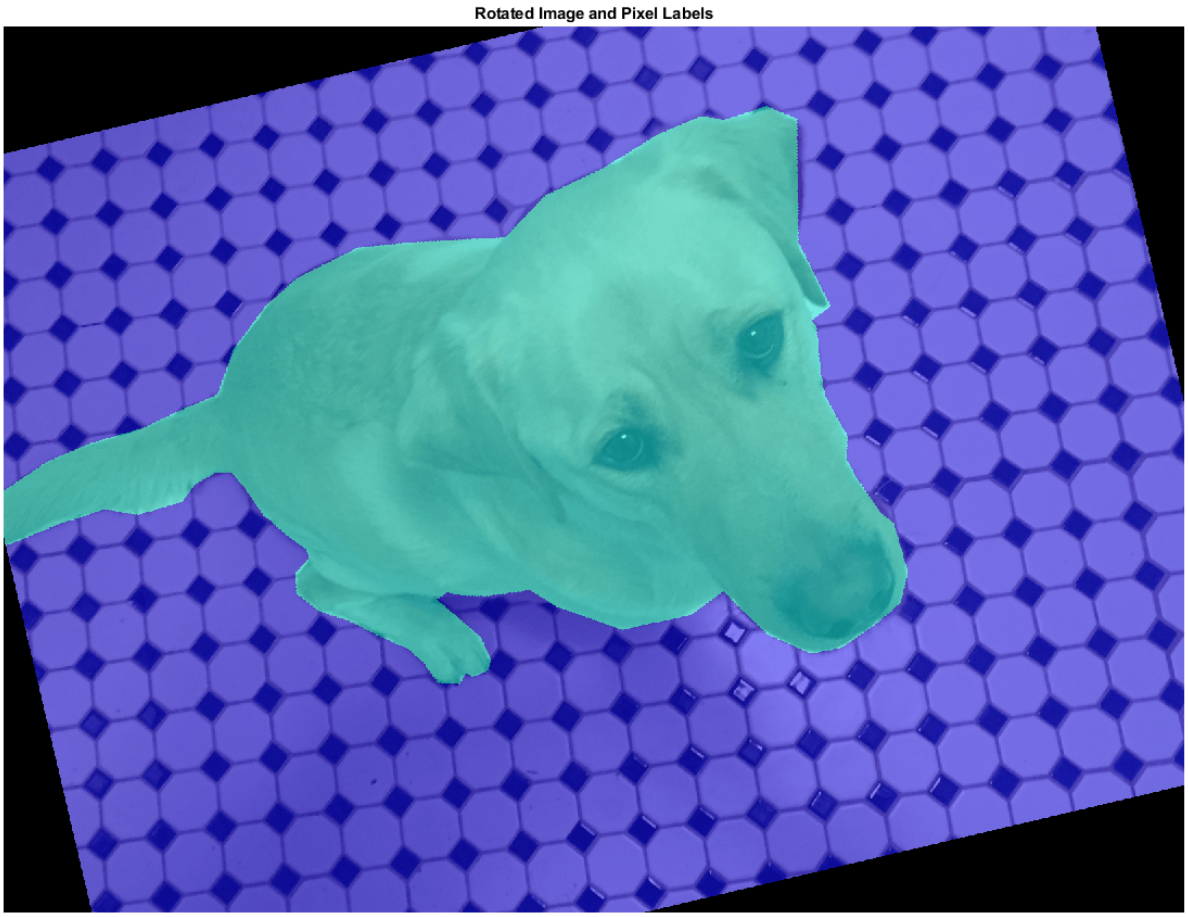
```
rout = affineOutputView(size(I),tform);
```

Use `imwarp` to rotate the image and pixel label image.

```
rotatedI = imwarp(I,tform,'OutputView',rout);
rotatedC = imwarp(C,tform,'OutputView',rout);
```

Display the rotated labels over the rotated image.

```
B = labeloverlay(rotatedI,rotatedC);
imshow(B)
title('Rotated Image and Pixel Labels')
```



Apply Augmentation to Semantic Segmentation Training Data in Datastores

Datastores are a convenient way to read and augment collections of images. Create a datastore that stores image and pixel label image data, and augment the data with a series of multiple operations.

Create Datastores Containing Image and Pixel Label Image Data

To increase the size of the sample datastores, replicate the filenames of the image and pixel label image.

```
numObservations = 4;
trainImages = repelem({filenameImage},numObservations,1);
trainLabels = repelem({filenameLabels},numObservations,1);
```

Create an `imageDatastore` from the training image files. Create a `pixelLabelDatastore` from the training pixel label files. The datastores contain multiple copies of the same data.

```
imds = imageDatastore(trainImages);
pxds = pixelLabelDatastore(trainLabels,classes,ids);
```

Associate the image and pixel label pairs by combining the image datastore and pixel label datastore.

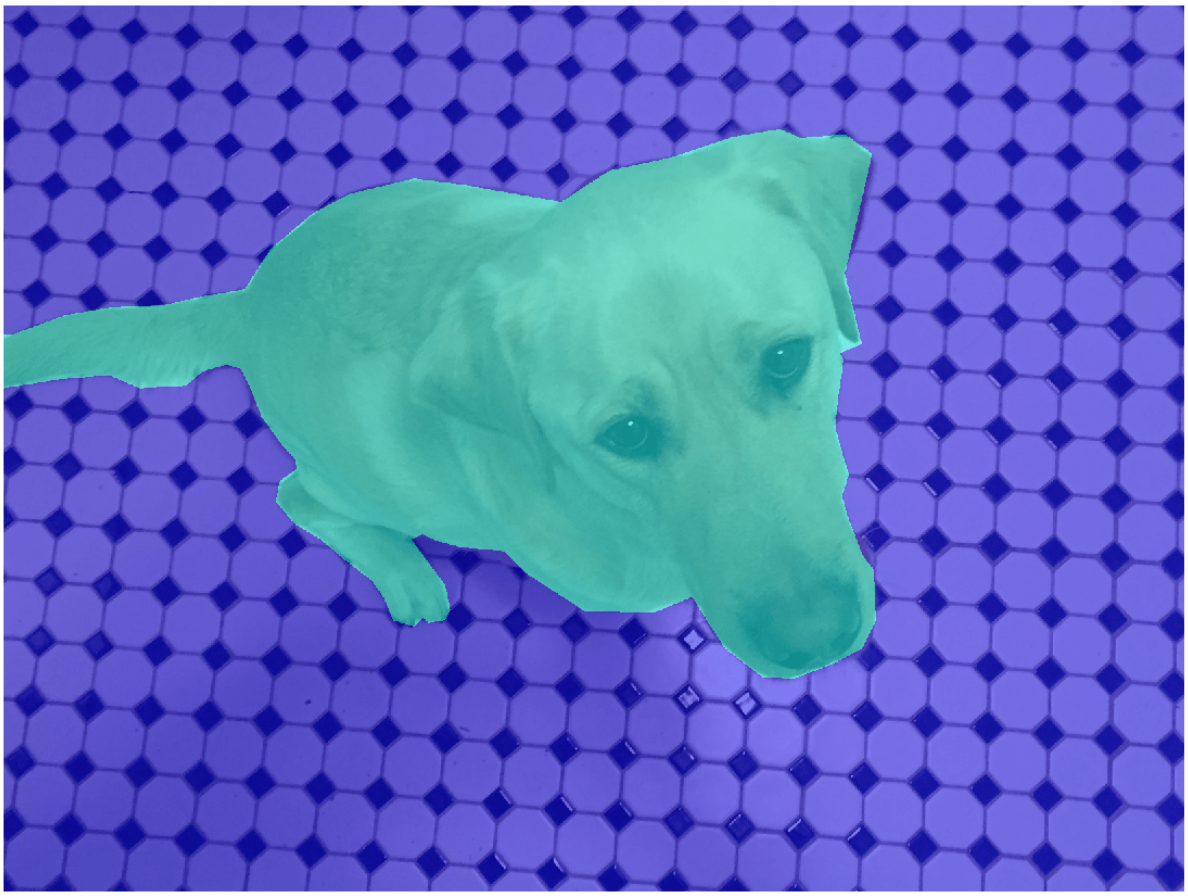
```
trainingData = combine(imds,pxds);
```

Read the first image and its associated pixel label image from the combined datastore.

```
data = read(trainingData);
I = data{1};
C = data{2};
```

Display the image and pixel label data.

```
B = labeloverlay(I,C);
imshow(B)
```



Apply Data Augmentation

Apply data augmentation to the training data by using the `transform` function. This example performs two separate augmentations to the training data.

The first augmentation jitters the color of the image and then performs identical random scaling, horizontal reflection, and rotation on the image and pixel label image pairs. These operations are defined in the `jitterImageColorAndWarp` helper function at the end of this example.

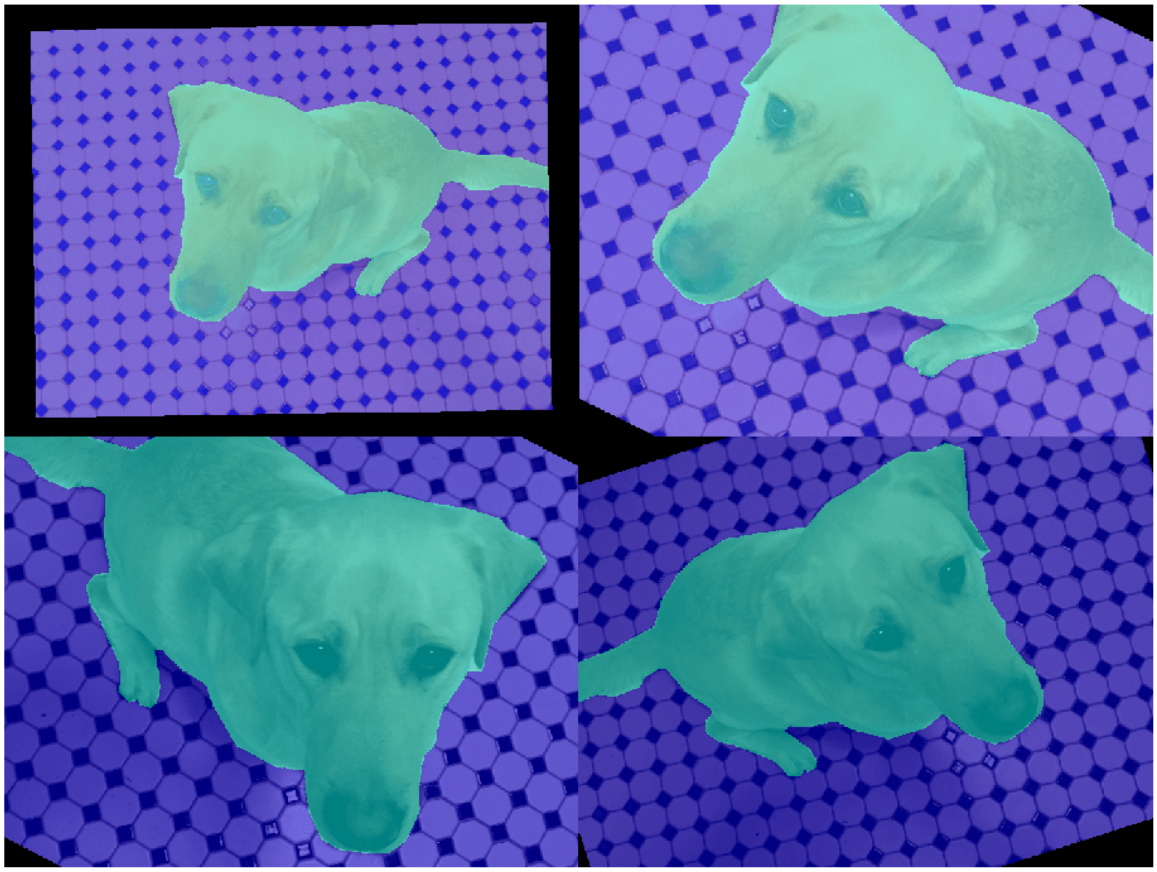
```
augmentedTrainingData = transform(trainingData,@jitterImageColorAndWarp);
```

Read all the augmented data.

```
data = readall(augmentedTrainingData);
```

Display the augmented image and pixel label data.

```
rgb = cell(numObservations,1);
for k = 1:numObservations
    I = data{k,1};
    C = data{k,2};
    rgb{k} = labeloverlay(I,C);
end
montage(rgb)
```



The second augmentation center crops the image and pixel label image to a target size. These operations are defined in the `centerCropImageAndLabel` helper function at the end of this example.

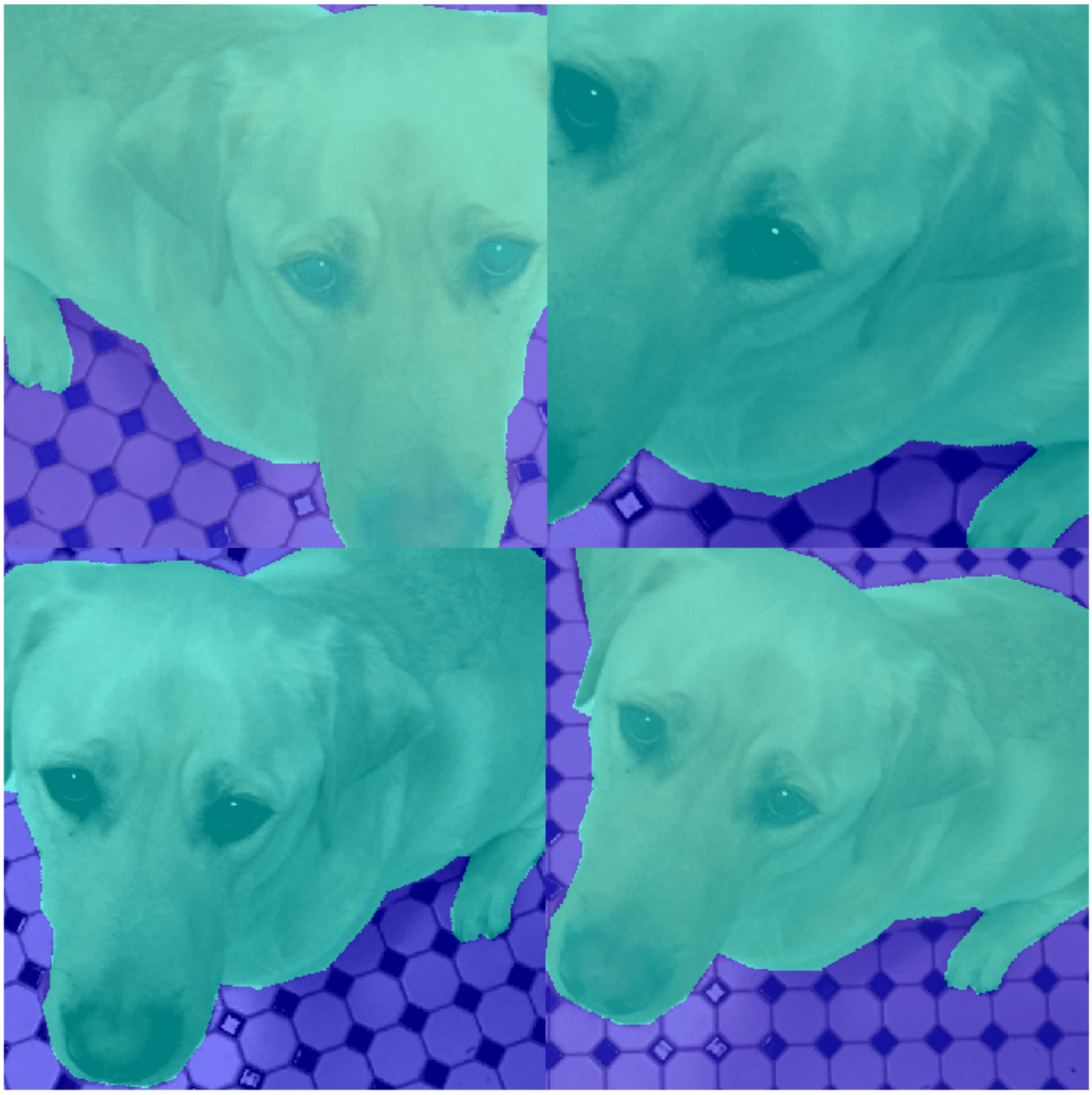
```
targetSize = [800 800];
preprocessedTrainingData = transform(augmentedTrainingData,...
    @(data)centerCropImageAndLabel(data,targetSize));
```

Read all of the preprocessed data.

```
data = readall(preprocessedTrainingData);
```

Display the preprocessed image and pixel label data.

```
rgb = cell(numObservations,1);
for k = 1:numObservations
    I = data{k,1};
    C = data{k,2};
    rgb{k} = labeloverlay(I,C);
end
montage(rgb)
```



Helper Functions for Augmentation

The `jitterImageColorAndWarp` helper function applies random color jitter to the image data, then applies an identical affine transformation to the image and pixel label image data. The transformation

consists of a random combination of scaling by a scale factor in the range [0.8 1.5], horizontal reflection, and rotation in the range [-30, 30] degrees. The input `data` and output `out` are two-element cell arrays, where the first element is the image data and the second element is the pixel label image data.

```
function out = jitterImageColorAndWarp(data)
% Unpack original data.
I = data{1};
C = data{2};

% Apply random color jitter.
I = jitterColorHSV(I,"Brightness",0.3,"Contrast",0.4,"Saturation",0.2);

% Define random affine transform.
tform = randomAffine2d("Scale",[0.8 1.5],"XReflection",true,'Rotation',[-30 30]);
rout = affineOutputView(size(I),tform);

% Transform image and bounding box labels.
augmentedImage = imwarp(I,tform,"OutputView",rout);
augmentedLabel = imwarp(C,tform,"OutputView",rout);

% Return augmented data.
out = {augmentedImage,augmentedLabel};
end
```

The `centerCropImageAndLabel` helper function creates a crop window centered on the image, then crops both the image and the pixel label image using the crop window. The input `data` and output `out` are two-element cell arrays, where the first element is the image data and the second element is the pixel label image data.

```
function out = centerCropImageAndLabel(data,targetSize)
win = centerCropWindow2d(size(data{1}),targetSize);
out{1} = imcrop(data{1},win);
out{2} = imcrop(data{2},win);
end
```

See Also

`randomAffine2d` | `centerCropWindow2d` | `randomWindow2d`

Related Examples

- “Augment Images for Deep Learning Workflows Using Image Processing Toolbox” on page 19-43
- “Semantic Segmentation Using Deep Learning” on page 8-126

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 19-27
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Augment Bounding Boxes for Object Detection

This example shows how to perform common kinds of image and bounding box augmentation as part of object detection workflows.

Object detector training data consists of images and associated bounding box labels. When you augment training data, you must apply identical transformations to the image and associated bounding boxes. This example demonstrates three common types of transformations:

- Resize Image and Bounding Box on page 19-0
- Crop Image and Bounding Box on page 19-0
- Warp Image and Bounding Box on page 19-0

The example then shows how to apply augmentation to training data in datastores on page 19-0 using a combination of multiple types of transformations.

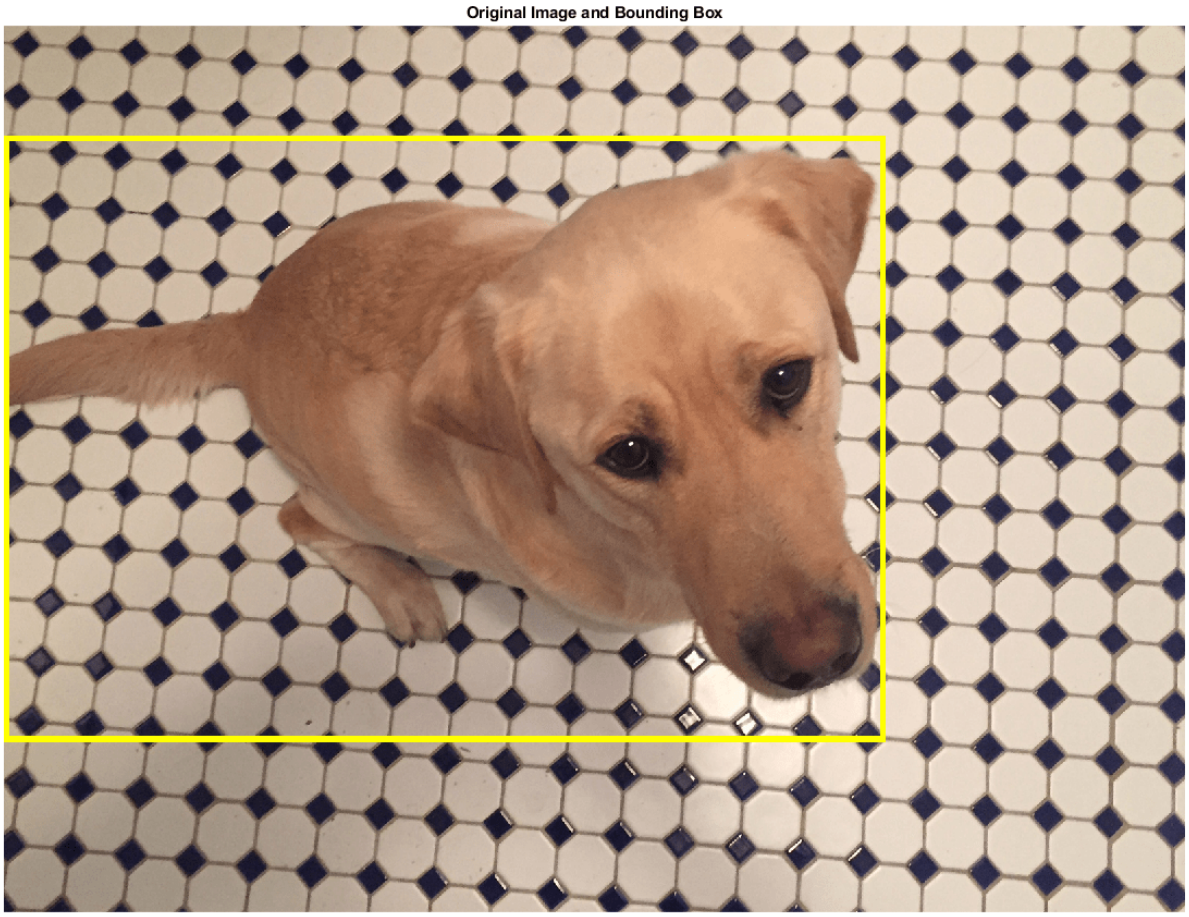
You can use augmented training data to train a network. For an example showing how to train an object detection network, see “Object Detection Using Faster R-CNN Deep Learning” (Computer Vision Toolbox).

Read and display a sample image and bounding box. To compare the effects of the different types of augmentation, each transformation in this example uses the same input image and bounding box.

```
filenameImage = 'kobi.png';
I = imread(filenameImage);
bbox = [4 156 1212 830];
label = "dog";
```

Display the image and bounding box.

```
annotatedImage = insertShape(I,"rectangle",bbox,"LineWidth",8);
imshow(annotatedImage)
title('Original Image and Bounding Box')
```



Resize Image and Bounding Box

Use `imresize` to scale down the image by a factor of 2.

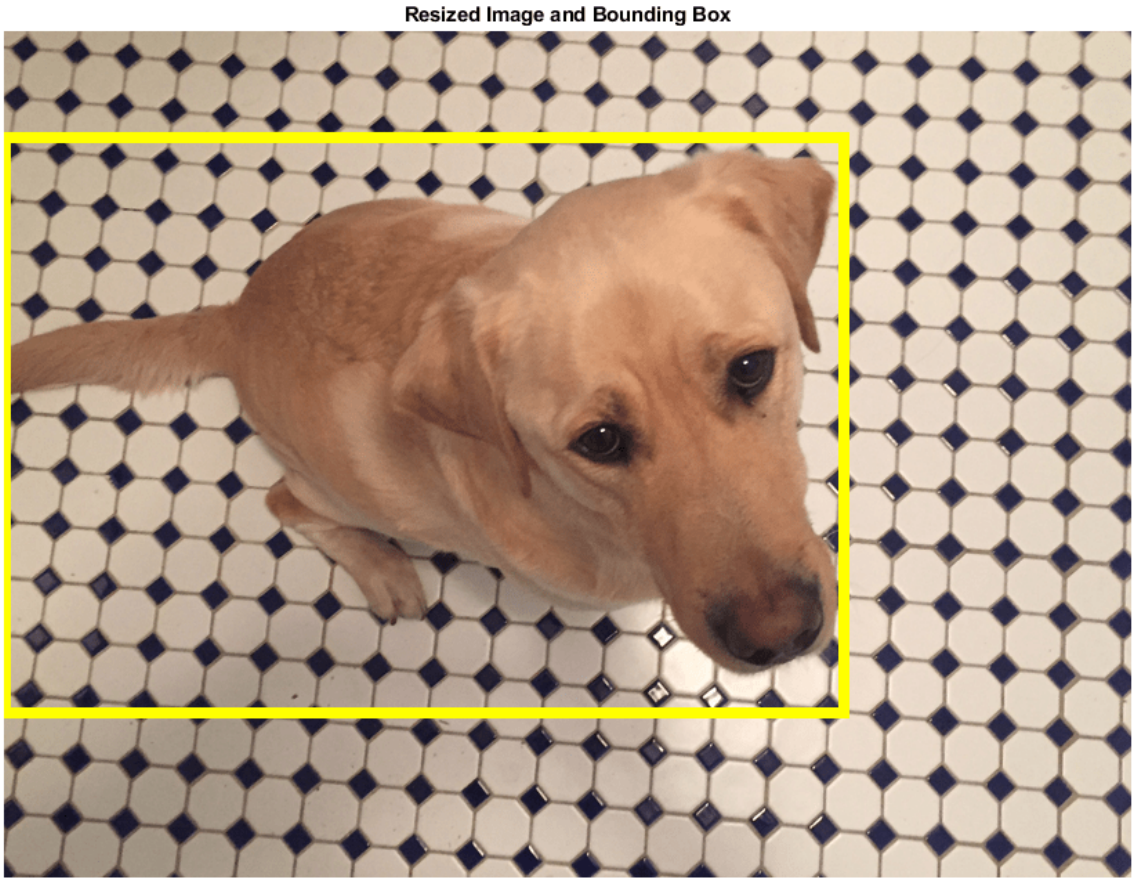
```
scale = 1/2;
J = imresize(I,scale);
```

Use `bboxresize` to apply the same scaling to the associated bounding box.

```
bboxResized = bboxresize(bbox,scale);
```

Display the resized image and bounding box.

```
annotatedImage = insertShape(J,"rectangle",bboxResized,"LineWidth",8);
imshow(annotatedImage)
title('Resized Image and Bounding Box')
```



Crop Image and Bounding Box

Cropping is a common preprocessing step to make the data match the input size of the network. To create output images of a desired size, first specify the size and position of the crop window by using the `randomWindow2d` (Image Processing Toolbox) or `centerCropWindow2d` (Image Processing Toolbox) function. Make sure you select a cropping window that includes the desired content in the image. Then, crop the image and pixel label image to the same window by using `imcrop`.

Specify the desired size of the cropped region as a two-element vector of the form [*height*, *width*].

```
targetSize = [1024 1024];
```

Crop the image to the target size from the center of the image by using `imcrop`.

```
win = centerCropWindow2d(size(I),targetSize);
J = imcrop(I,win);
```

Crop the bounding boxes using the same crop window by using `bboxcrop`. Specify `OverlapThreshold` as a value less than 1 so that the function clips the bounding boxes to the crop window instead of discarding them when the crop window does not completely enclose the bounding box. The overlap threshold enables you to control the amount of clipping that is tolerable for objects in your images. For example, clipping more than half a person is not useful for training a person detector, whereas clipping half a vehicle might be tolerable.

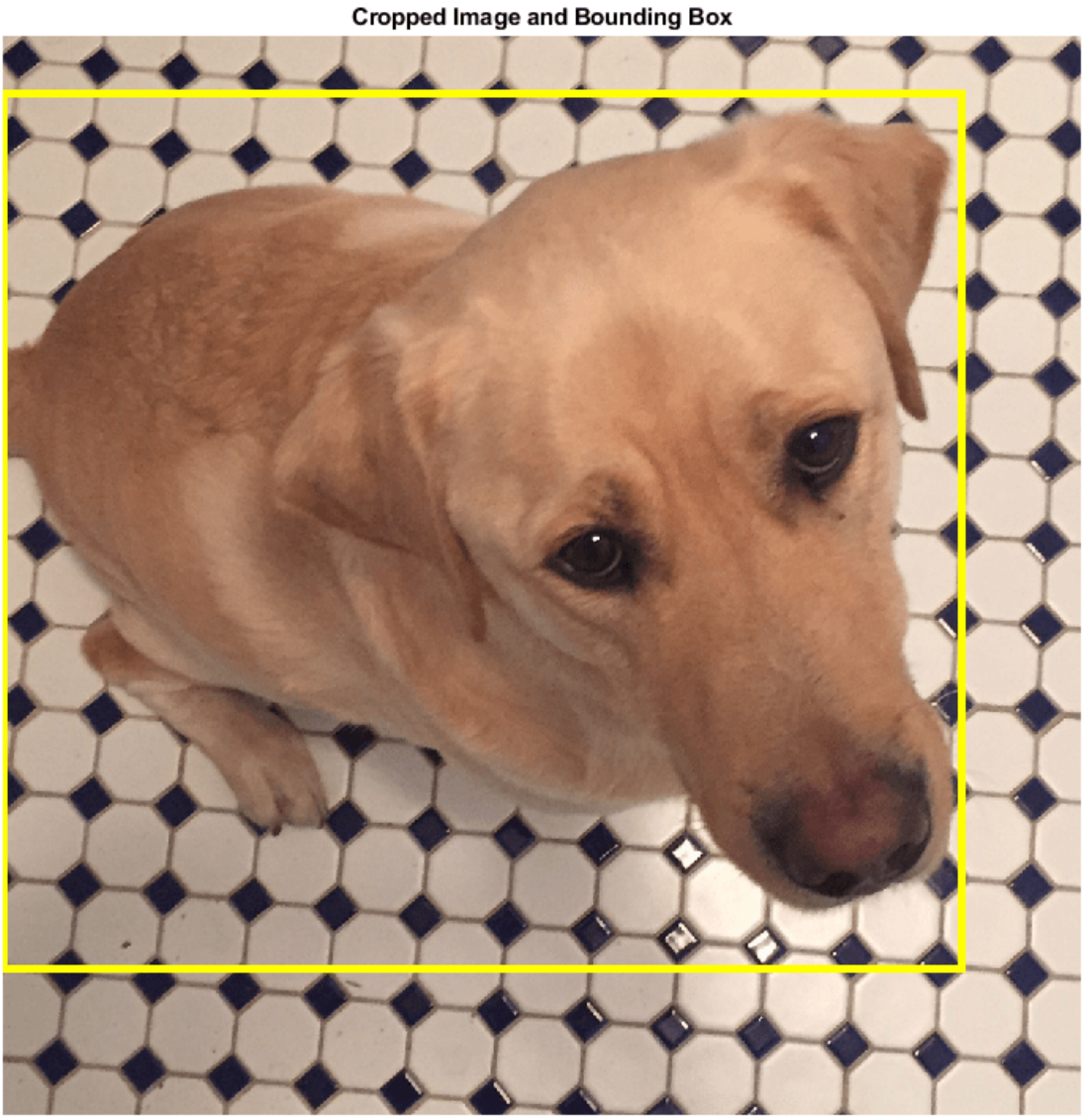
```
[bboxCropped,valid] = bboxcrop(bbox,win,"OverlapThreshold",0.7);
```

Keep labels that are inside the cropping window.

```
label = label(valid);
```

Display the cropped image and bounding box.

```
annotatedImage = insertShape(J,"rectangle",bboxCropped,"LineWidth",8);
imshow(annotatedImage)
title('Cropped Image and Bounding Box')
```



Crop and Resize Image and Bounding Box

Cropping and resizing are often performed together. You can use `bboxcrop` and `bboxresize` in series to implement the commonly used "*crop and resize*" transformation.

Create a crop window from a random position in the image. Crop the image and bounding box to the same crop window.

```
cropSize = [1024 1024];
win = randomWindow2d(size(I),cropSize);
J = imcrop(I,win);
croppedBox = bboxcrop(bbox,win,"OverlapThreshold",0.5);
```

Resize the image and box to a target size.

```
targetSize = [512 512];
J = imresize(J,targetSize);
croppedAndResizedBox = bboxresize(croppedBox,targetSize./cropSize);
```

Display the cropped and resized image and bounding box.

```
annotatedImage = insertShape(J,"rectangle",croppedAndResizedBox,"LineWidth",8);
imshow(annotatedImage)
title('Crop and Resized Image and Bounding Box')
```

Warp Image and Bounding Box

The `randomAffine2d` (Image Processing Toolbox) function creates a randomized 2-D affine transformation from a combination of rotation, translation, scaling (resizing), reflection, and shearing. Warp an image by using `imwarp` (Image Processing Toolbox). Warp bounding boxes by using `bboxwarp`. Control the spatial bounds and resolution of the warped output by using the `affineOutputView` (Image Processing Toolbox) function.

This example demonstrates two of the randomized affine transformations: scaling and rotation.

Random Scale

Create a scale transformation that resizes the input image and bounding box using a scale factor selected randomly from the range [1.5,1.8]. This transformation applies the same scale factor in the horizontal and vertical directions.

```
tform = randomAffine2d("Scale",[1.5 1.8]);
```

Create an output view for the affine transform.

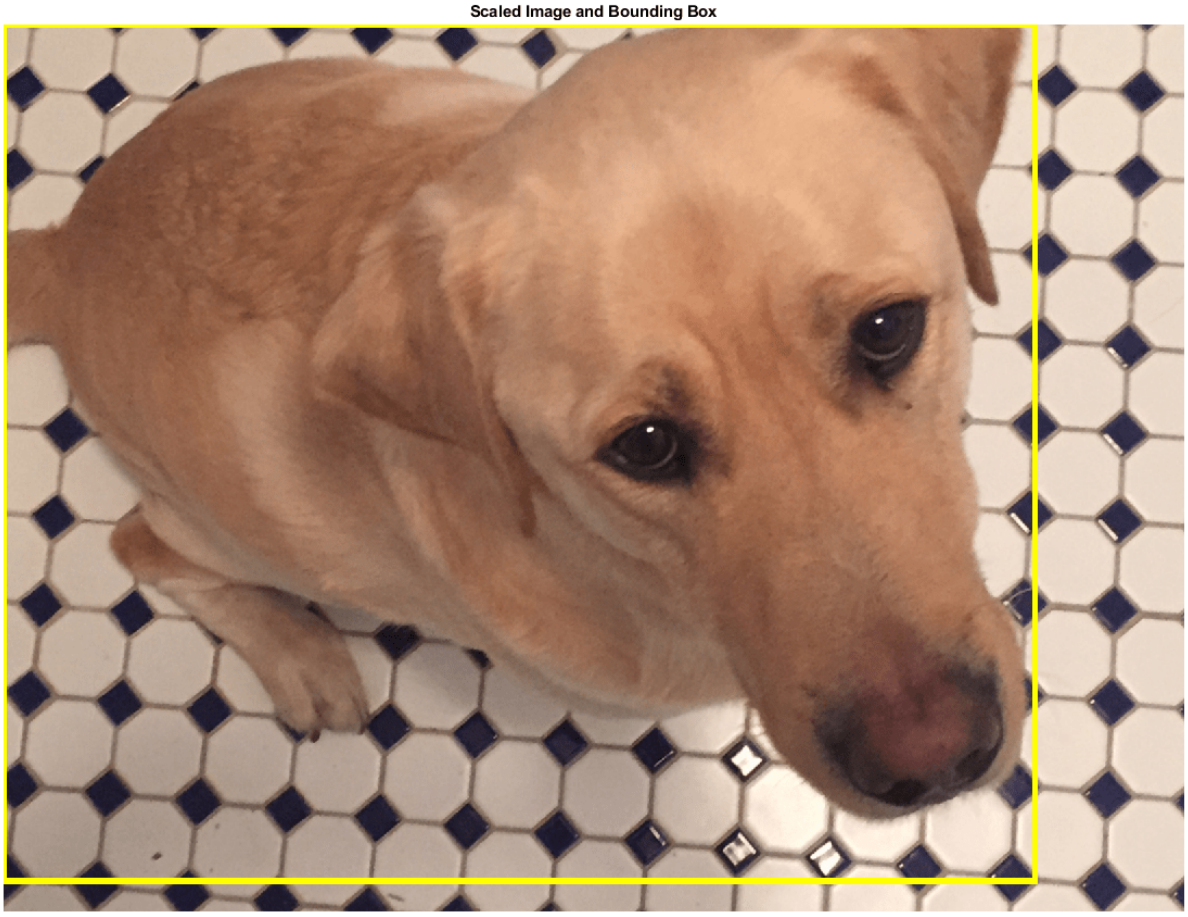
```
rout = affineOutputView(size(I),tform);
```

Rescale the image using `imwarp` and rescale the bounding box using `bboxwarp`. Specify an `OverlapThreshold` value of 0.5.

```
J = imwarp(I,tform,"OutputView",rout);
bboxScaled = bboxwarp(bbox,tform,rout,"OverlapThreshold",0.5);
```

Display the scaled image and bounding box.

```
annotatedImage = insertShape(J,"rectangle",bboxScaled,"LineWidth",8);
imshow(annotatedImage)
title('Scaled Image and Bounding Box')
```



Random Rotation

Create a randomized rotation transformation that rotates the image and box labels by an angle selected randomly from the range [-15,15] degrees.

```
tform = randomAffine2d("Rotation",[-15 15]);
```

Create an output view for `imwarp` and `bboxwarp`.

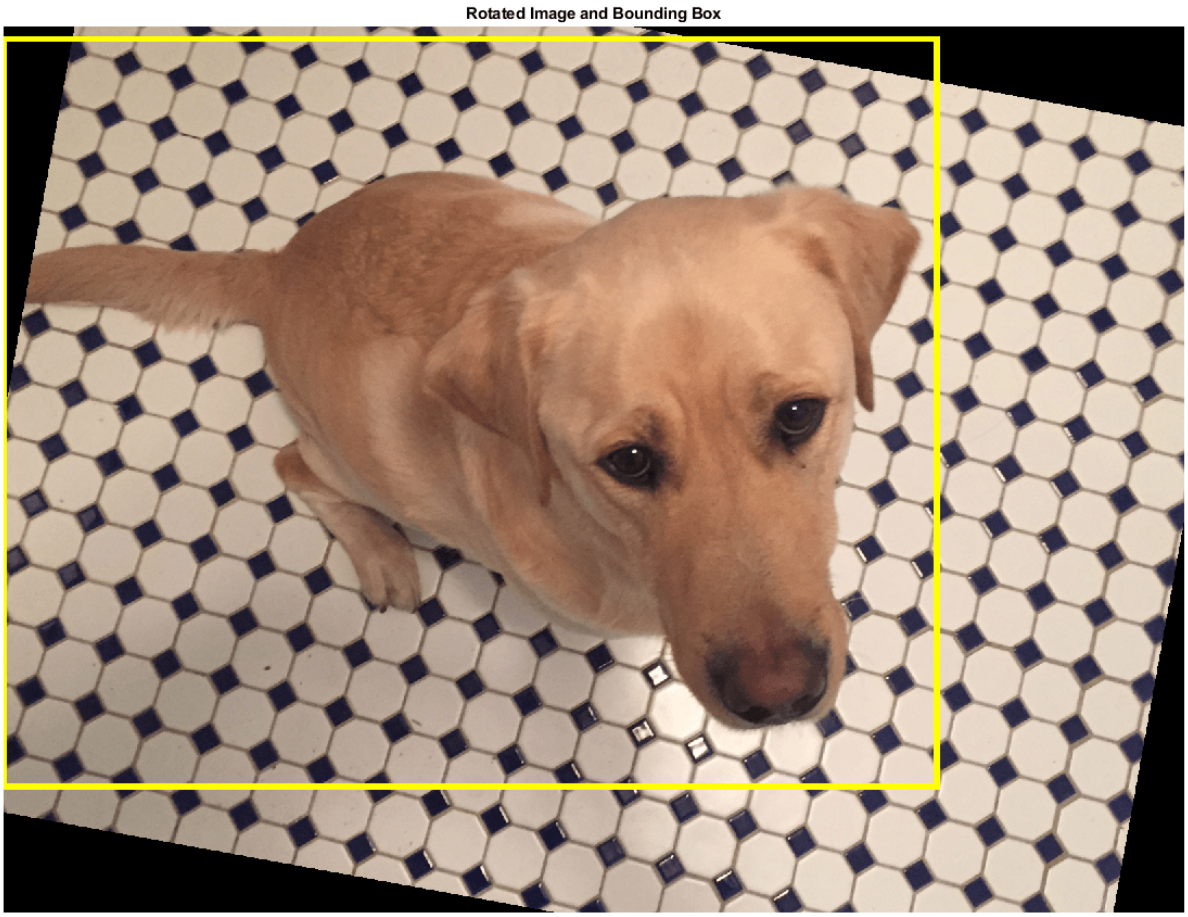
```
rout = affineOutputView(size(I),tform);
```

Rotate the image using `imwarp` and rotate the bounding box using `bboxwarp`. Specify an `OverlapThreshold` value of 0.5.

```
J = imwarp(I,tform,"OutputView",rout);
bboxRotated = bboxwarp(bbox,tform,rout,"OverlapThreshold",0.5);
```

Display the cropped image and bounding box. Note that the bounding box returned by `bboxwarp` is always aligned to the image axes. The size and aspect ratio of the bounding box changes to accommodate the rotated object.

```
annotatedImage = insertShape(J,"rectangle",bboxRotated,"LineWidth",8);
imshow(annotatedImage)
title('Rotated Image and Bounding Box')
```



Apply Augmentation to Training Data in Datastores

Datastores are a convenient way to read and augment collections of data. Create a datastore that stores image and bounding box data, and augment the data using a series of multiple operations.

Create Datastores Containing Image and Bounding Box Data

To increase the size of the sample datastores, replicate the file names of the image and the bounding box and labels.

```
numObservations = 4;
images = repelem({filenameImage},numObservations,1);
bboxes = repelem({bbox},numObservations,1);
labels = repelem({label},numObservations,1);
```

Create an `imageDatastore` from the training image files. Combine the bounding box and label data in a table, then create a `boxLabelDatastore` from the table.

```
imds = imageDatastore(images);

tbl = table(bboxes,labels);
blds = boxLabelDatastore(tbl);
```

Associate the image and box label pairs by combining the image datastore and box label datastore.

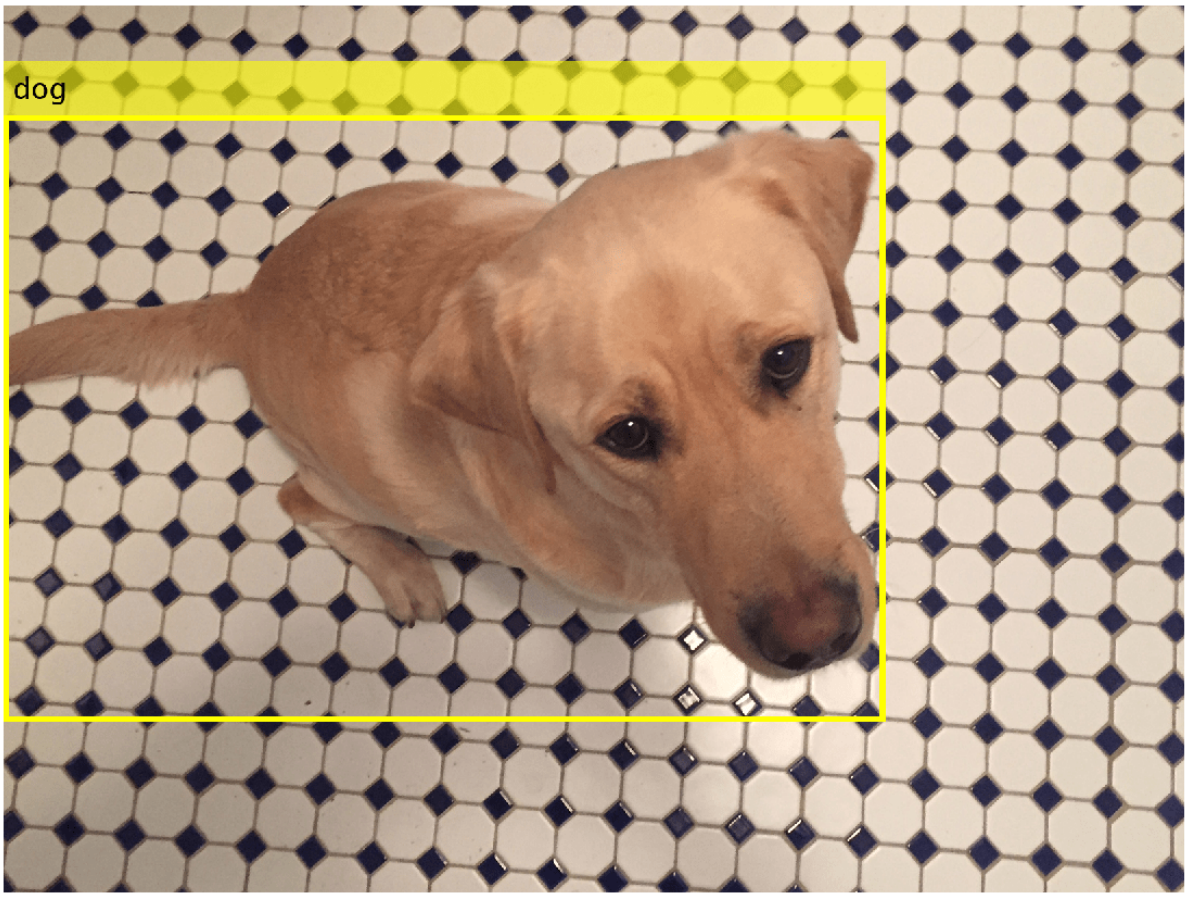
```
trainingData = combine(imds,blds);
```

Read the first image and its associated box label from the combined datastore.

```
data = read(trainingData);
I = data{1};
bboxes = data{2};
labels = data{3};
```

Display the image and box label data.

```
annotatedImage = insertObjectAnnotation(I,'rectangle',bbox,labels, ...
    'LineWidth',8,'FontSize',40);
imshow(annotatedImage)
```



Apply Data Augmentation

Apply data augmentation to the training data by using the `transform` function. This example performs two separate augmentations to the training data.

The first augmentation jitters the color of the image and then performs identical random horizontal reflection and rotation on the image and box label pairs. These operations are defined in the `jitterImageColorAndWarp` helper function at the end of this example.

```
augmentedTrainingData = transform(trainingData,@jitterImageColorAndWarp);
```

Read all the augmented data.

```
data = readall(augmentedTrainingData);
```

Display the augmented image and box label data.

```
rgb = cell(numObservations,1);
for k = 1:numObservations
    I = data{k,1};
    bbox = data{k,2};
    labels = data{k,3};
    rgb{k} = insertObjectAnnotation(I,'rectangle',bbox,labels,'LineWidth',8,'FontSize',40);
end
montage(rgb)
```

The second augmentation rescales the image and box label to a target size. These operations are defined in the `resizeImageAndLabel` helper function at the end of this example.

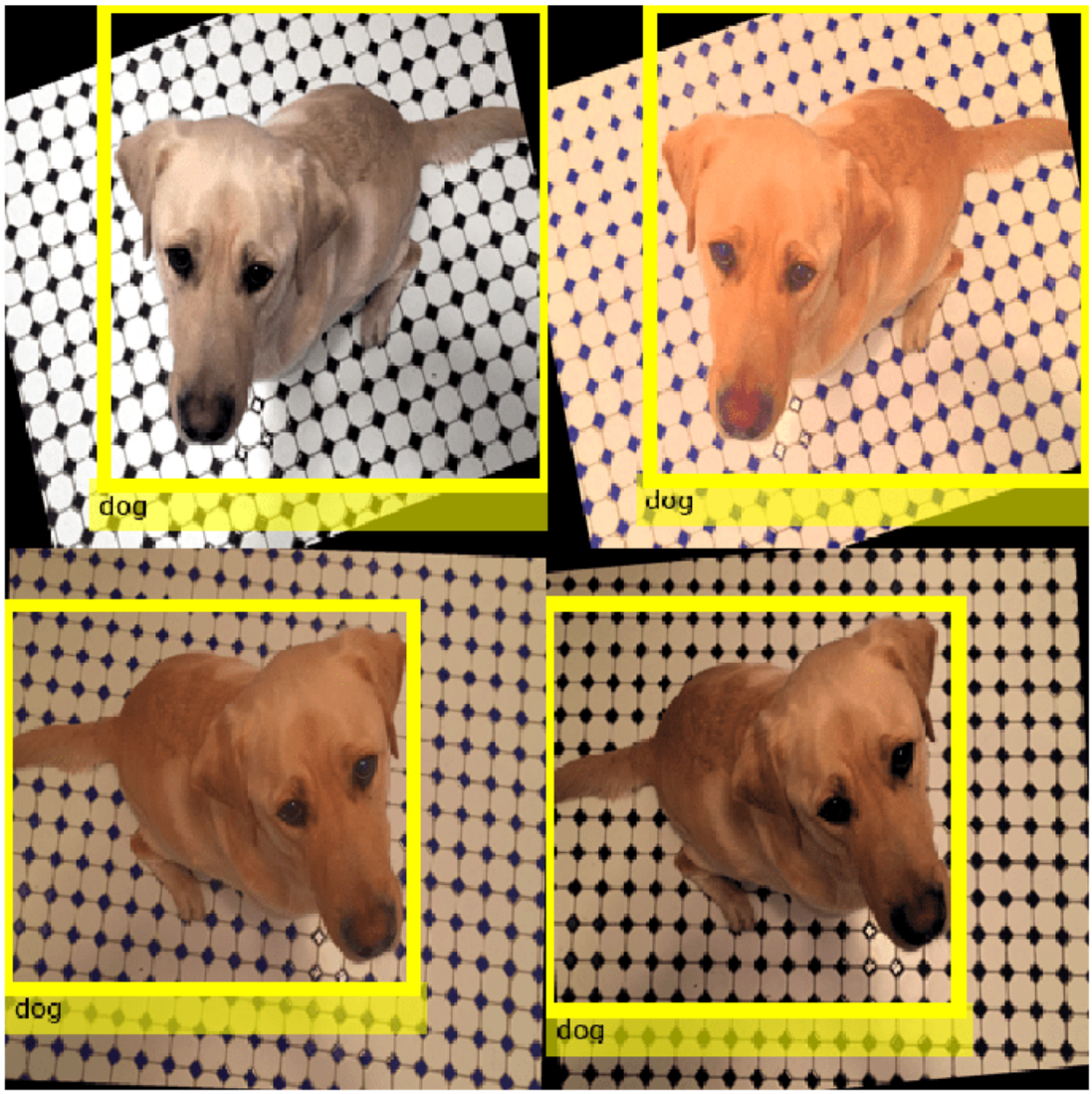
```
targetSize = [300 300];
preprocessedTrainingData = transform(augmentedTrainingData,...
    @(data)resizeImageAndLabel(data,targetSize));
```

Read all of the preprocessed data.

```
data = readall(preprocessedTrainingData);
```

Display the preprocessed image and box label data.

```
rgb = cell(numObservations,1);
for k = 1:numObservations
    I = data{k,1};
    bbox = data{k,2};
    labels = data{k,3};
    rgb{k} = insertObjectAnnotation(I,'rectangle',bbox,labels, ...
        'LineWidth',8,'FontSize',15);
end
montage(rgb)
```



Helper Functions for Augmentation

The `jitterImageColorAndWarp` helper function applies random color jitter to the image data, then applies an identical affine transformation to the image and box label data. The transformation consists of random horizontal reflection and rotation. The input `data` and output `out` are two-element cell arrays, where the first element is the image data and the second element is the box label data.

```
function out = jitterImageColorAndWarp(data)
% Unpack original data.
I = data{1};
boxes = data{2};
labels = data{3};
```

```
% Apply random color jitter.
I = jitterColorHSV(I,"Brightness",0.3,"Contrast",0.4,"Saturation",0.2);

% Define random affine transform.
tform = randomAffine2d("XReflection",true,'Rotation',[-30 30]);
rout = affineOutputView(size(I),tform);

% Transform image and bounding box labels.
augmentedImage = imwarp(I,tform,"OutputView",rout);
[augmentedBoxes, valid] = bboxwarp(boxes,tform,rout,'OverlapThreshold',0.4);
augmentedLabels = labels(valid);

% Return augmented data.
out = {augmentedImage,augmentedBoxes,augmentedLabels};
end
```

The `resizeImageAndLabel` helper function calculates the scale factor for the image to match a target size, then resizes the image using `imresize` and the box label using `bboxresize`. The input and output data are two-element cell arrays, where the first element is the image data and the second element is the box label data.

```
function data = resizeImageAndLabel(data,targetSize)
scale = targetSize./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize);
data{2} = bboxresize(data{2},scale);
end
```

See Also

`bboxresize` | `bboxcrop` | `bboxwarp` | `imresize` | `imcrop` | `centerCropWindow2d` | `randomWindow2d`

Related Examples

- “Augment Images for Deep Learning Workflows Using Image Processing Toolbox” on page 19-43
- “Train Object Detector Using R-CNN Deep Learning” on page 8-188

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 19-27
- “Getting Started with Object Detection Using Deep Learning” (Computer Vision Toolbox)

Prepare Datastore for Image-to-Image Regression

This example shows how to prepare a datastore for training an image-to-image regression network using the `transform` and `combine` functions of `ImageDatastore`.

This example shows how to preprocess data using a pipeline suitable for training a denoising network. This example then uses the preprocessed noise data to train a simple convolutional autoencoder network to remove image noise.

Prepare Data Using Preprocessing Pipeline

This example uses a salt and pepper noise model in which a fraction of input image pixels are set to either `0` or `1` (black and white, respectively). Noisy images act as the network input. Pristine images act as the expected network response. The network learns to detect and remove the salt and pepper noise.

Load the pristine images in the digit data set as an `imageDatastore`. The datastore contains 10,000 synthetic images of digits from 0 to 9. The images are generated by applying random transformations to digit images created with different fonts. Each digit image is 28-by-28 pixels. The datastore contains an equal number of images per category.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet', ...
    'nndemos','nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
```

Specify a large read size to minimize the cost of file I/O.

```
imds.ReadSize = 500;
```

Set the seed of the global random number generator to aid in the reproducibility of results.

```
rng(0)
```

Use the `shuffle` function to shuffle the digit data prior to training.

```
imds = shuffle(imds);
```

Use the `splitEachLabel` function to divide `imds` into three image datastores containing pristine images for training, validation, and testing.

```
[imdsTrain,imdsVal,imdsTest] = splitEachLabel(imds,0.95,0.025);
```

Use the `transform` function to create noisy versions of each input image, which will serve as the network input. The `transform` function reads data from an underlying datastore and processes the data using the operations defined in the helper function `addNoise` (defined at the end of this example). The output of the `transform` function is a `TransformedDatastore`.

```
dsTrainNoisy = transform(imdsTrain,@addNoise);
dsValNoisy = transform(imdsVal,@addNoise);
dsTestNoisy = transform(imdsTest,@addNoise);
```

Use the `combine` function to combine the noisy images and pristine images into a single datastore that feeds data to `trainNetwork`. This combined datastore reads batches of data into a two-column cell array as expected by `trainNetwork`. The output of the `combine` function is a `CombinedDatastore`.

```
dsTrain = combine(dsTrainNoisy,imdsTrain);
dsVal = combine(dsValNoisy,imdsVal);
dsTest = combine(dsTestNoisy,imdsTest);
```

Use the `transform` function to perform additional preprocessing operations that are common to both the input and response datastores. The `commonPreprocessing` helper function (defined at the end of this example) resizes input and response images to 32-by-32 pixels to match the input size of the network, and normalizes the data in each image to the range [0, 1].

```
dsTrain = transform(dsTrain,@commonPreprocessing);
dsVal = transform(dsVal,@commonPreprocessing);
dsTest = transform(dsTest,@commonPreprocessing);
```

Finally, use the `transform` function to add randomized augmentation to the training set. The `augmentImages` helper function (defined at the end of this example) applies randomized 90 degree rotations to the data. Identical rotations are applied to the network input and corresponding expected responses.

```
dsTrain = transform(dsTrain,@augmentImages);
```

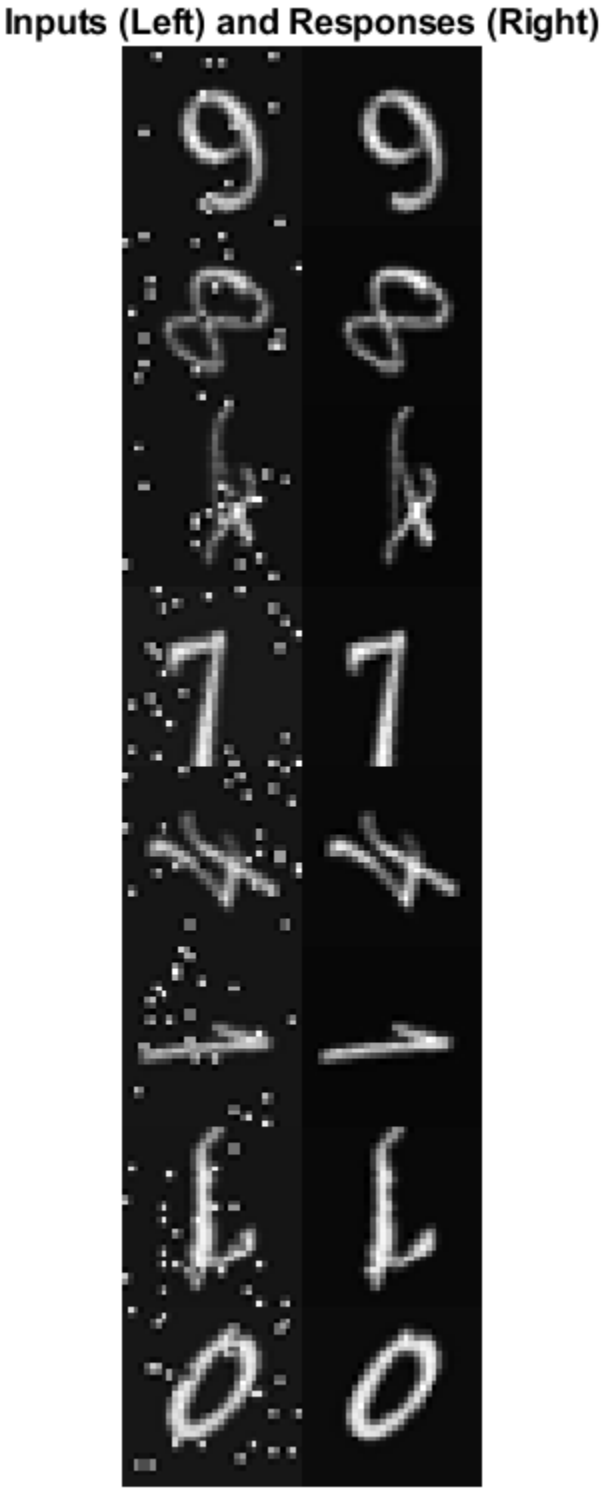
Augmentation reduces overfitting and adds robustness to the presence of rotations in the trained network. Randomized augmentation is not needed for the validation or test data sets.

Preview Preprocessed Data

Since there are several preprocessing operations necessary to prepare the training data, preview the preprocessed data to confirm it looks correct prior to training. Use the `preview` function to preview the data.

Visualize examples of paired noisy and pristine images using the `montage` (Image Processing Toolbox) function. The training data looks correct. Salt and pepper noise appears in the input images in the left column. Other than the addition of noise, the input image and response image are the same. Randomized 90 degree rotation is applied to both input and response images in the same way.

```
exampleData = preview(dsTrain);
inputs = exampleData(:,1);
responses = exampleData(:,2);
minibatch = cat(2,inputs,responses);
montage(minibatch','Size',[8 2])
title('Inputs (Left) and Responses (Right)')
```


Inputs (Left) and Responses (Right)

Define Convolutional Autoencoder Network

Convolutional autoencoders are a common architecture for denoising images. Convolutional autoencoders consist of two stages: an encoder and a decoder. The encoder compresses the original input image into a latent representation that is smaller in width and height, but deeper in the sense that there are many feature maps per spatial location than the original input image. The compressed latent representation loses some amount of spatial resolution in its ability to recover high frequency features in the original image, but it also learns to not include noisy artifacts in the encoding of the original image. The decoder repeatedly upsamples the encoded signal to move it back to its original width, height, and number of channels. Since the encoder removes noise, the decoded final image has fewer noise artifacts.

This example defines the convolutional autoencoder network using layers from Deep Learning Toolbox™, including:

- `imageInputLayer` - Image input layer
- `convolution2dLayer` - Convolution layer for convolutional neural networks
- `reluLayer` - Rectified linear unit layer
- `maxPooling2dLayer` - 2-D max pooling layer
- `transposedConv2dLayer` - Transposed convolution layer
- `clippedReluLayer` - Clipped rectified linear unit layer
- `regressionLayer` - Regression output layer

Create the image input layer. To simplify the padding concerns related to downsampling and upsampling by factors of two, choose a 32-by-32 input size because 32 is cleanly divisible by 2, 4, and 8.

```
imageLayer = imageInputLayer([32,32,1]);
```

Create the encoding layers. Downsampling in the encoder is achieved by max pooling with a pool size of 2 and a stride of 2.

```
encodingLayers = [ ...
    convolution2dLayer(3,16,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2), ...
    convolution2dLayer(3,8,'Padding','same'), ...
    reluLayer, ...
    maxPooling2dLayer(2,'Padding','same','Stride',2)];
```

Create the decoding layers. The decoder upsamples the encoded signal using a transposed convolution layer. Create the transposed convolution layer with the correct upsampling factor by using the `createUpsampleTransponseConvLayer` helper function. This function is defined at the end of this example.

The network uses a `clippedReluLayer` as the final activation layer to force outputs to be in the range [0, 1].

```
decodingLayers = [ ...
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
```

```
    createUpsampleTransponseConvLayer(2,8), ...
    reluLayer, ...
    createUpsampleTransponseConvLayer(2,16), ...
    reluLayer, ...
    convolution2dLayer(3,1,'Padding','same'), ...
    clippedReluLayer(1.0), ...
    regressionLayer];
```

Concatenate the image input layer, the encoding layers, and the decoding layers to form the convolutional autoencoder network architecture.

```
layers = [imageLayer,encodingLayers,decodingLayers];
```

Define Training Options

Train the network using the Adam optimizer. Specify the hyperparameter settings by using the `trainingOptions` function. Train for 100 epochs.

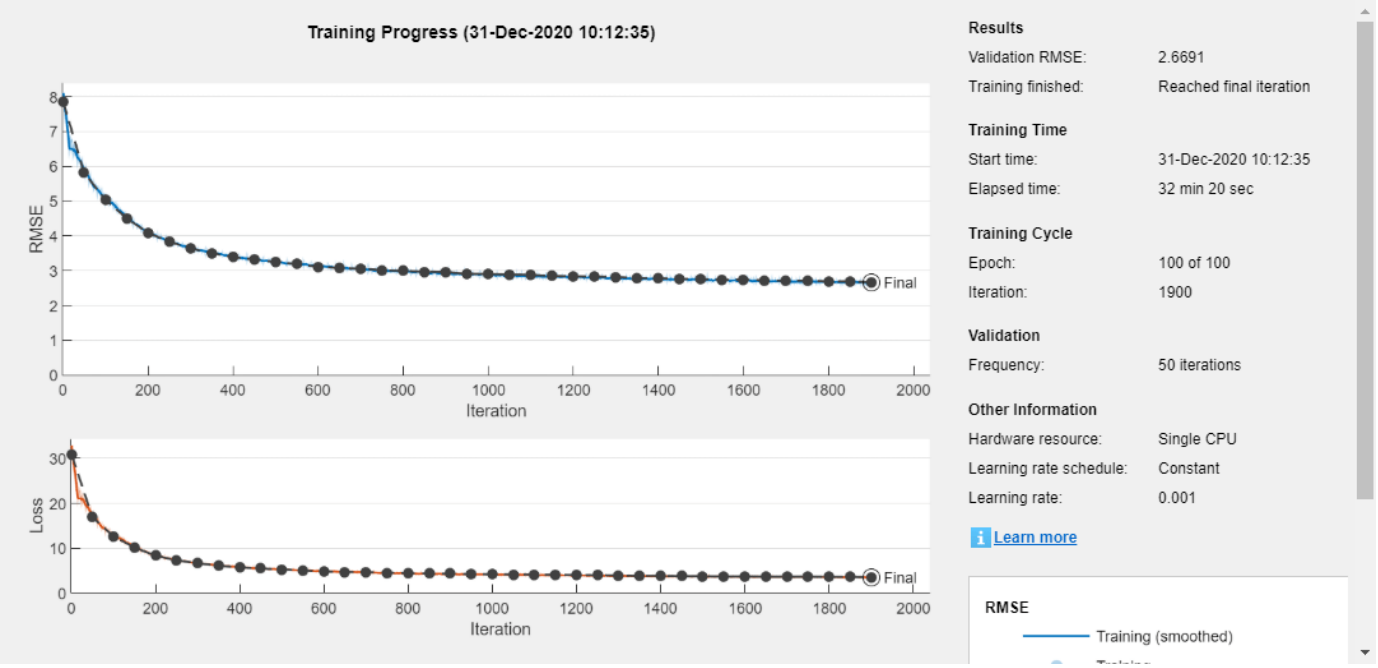
```
options = trainingOptions('adam', ...
    'MaxEpochs',100, ...
    'MiniBatchSize',imds.ReadSize, ...
    'ValidationData',dsVal, ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

Now that the data source and training options are configured, train the convolutional autoencoder network using the `trainNetwork` function.

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For more information, see “GPU Support by Release” (Parallel Computing Toolbox). Training takes about 25 minutes on an NVIDIA Titan XP.

```
net = trainNetwork(dsTrain,layers,options);
```



```
modelDateTime = string(datetime('now','Format',"yyyy-MM-dd-HH-mm-ss"));
save(strcat("trainedImageToImageRegressionNet-",modelDateTime,".mat"),'net');
```

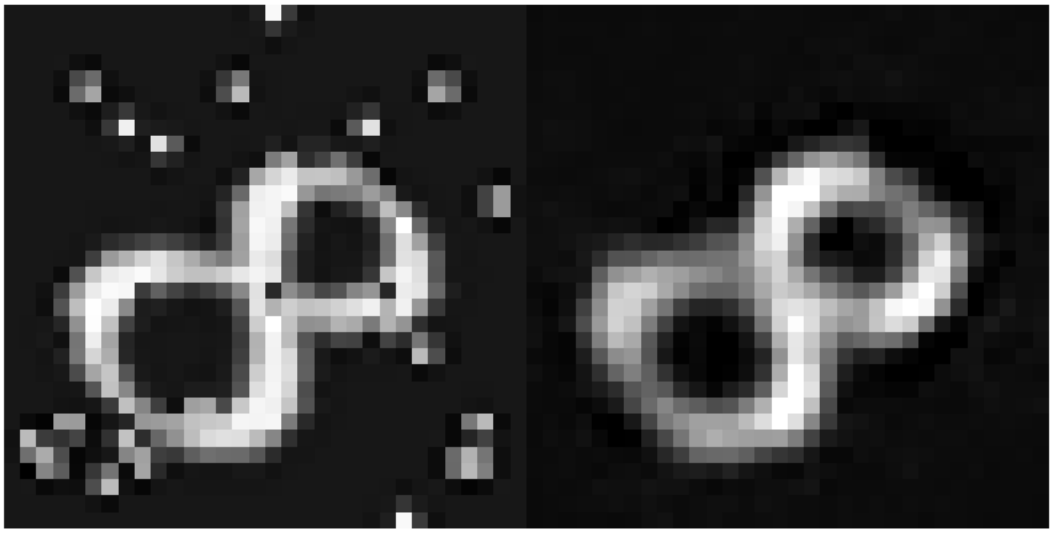
Evaluate the Performance of the Denoising Network

Obtain output images from the test set by using the `predict` function.

```
ypred = predict(net,dsTest);
```

Visualize a sample input image and the associated prediction output from the network to get a sense of how well denoising is working. As expected, the output image from the network has removed most of the noise artifacts from the input image. The denoised image is slightly blurry as a result of the encoding and decoding process.

```
inputImageExamples = preview(dsTest);
montage({inputImageExamples{1},ypred(:,:,:,1)});
```



Assess the performance of the network by analyzing the peak signal-to-noise ratio (PSNR).

```
ref = inputImageExamples{1,2};
originalNoisyImage = inputImageExamples{1,1};
psnrNoisy = psnr(originalNoisyImage,ref)
```

```
psnrNoisy = single
    17.8455
```

```
psnrDenoised = psnr(ypred(:,:,:,1),ref)
```

```
psnrDenoised = single
    21.8439
```

The PSNR of the output image is higher than the noisy input image, as expected.

Supporting Functions

The `addNoise` helper function adds salt and pepper noise to images by using the `imnoise` (Image Processing Toolbox) function. The `addNoise` function requires the format of the input data to be a cell array of image data, which matches the format of data returned by the `read` function of `ImageDatastore`.

```
function dataOut = addNoise(data)

    dataOut = data;
    for idx = 1:size(data,1)
       dataOut{idx} = imnoise(data{idx},'salt & pepper');
    end

end
```

The `commonPreprocessing` helper function defines the preprocessing that is common to the training, validation, and test sets. The helper function performs these preprocessing steps.

1. Convert the image data to data type `single`.
2. Resize image data to match the size of the input layer by using the `imresize` function.
3. Normalize data to the range [0, 1] by using the `rescale` function.

The helper function requires the format of the input data to be a two-column cell array of image data, which matches the format of data returned by the `read` function of `CombinedDatastore`.

```
function dataOut = commonPreprocessing(data)

    dataOut = cell(size(data));
    for col = 1:size(data,2)
        for idx = 1:size(data,1)
            temp = single(data{idx,col});
            temp = imresize(temp,[32,32]);
            temp = rescale(temp);
            dataOut{idx,col} = temp;
        end
    end
end
```

The `augmentImages` helper function adds randomized 90 degree rotations to the data by using the `rot90` function. Identical rotations are applied to the network input and corresponding expected responses. The function requires the format of the input data to be a two-column cell array of image data, which matches the format of data returned by the `read` function of `CombinedDatastore`.

```
function dataOut = augmentImages(data)

    dataOut = cell(size(data));
    for idx = 1:size(data,1)
        rot90Val = randi(4,1,1)-1;
        dataOut(idx,:) = {rot90(data{idx,1},rot90Val),rot90(data{idx,2},rot90Val)};
    end
end
```

The `createUpsampleTransposeConvLayer` helper function defines a transposed convolution layer that upsamples the layer input by the specified factor.

```
function out = createUpsampleTransponseConvLayer(factor,numFilters)

    filterSize = 2*factor - mod(factor,2);
    cropping = (factor-mod(factor,2))/2;
    numChannels = 1;

    out = transposedConv2dLayer(filterSize,numFilters, ...
        'NumChannels',numChannels,'Stride',factor,'Cropping',cropping);
end
```

See Also

`trainNetwork` | `trainingOptions` | `transform` | `combine` | `imageDatastore`

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

More About

- “Datastores for Deep Learning” on page 19-2

Train Network Using Out-of-Memory Sequence Data

This example shows how to train a deep learning network on out-of-memory sequence data by transforming and combining datastores.

A transformed datastore transforms or processes data read from an underlying datastore. You can use a transformed datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use transformed datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data. When you have separate datastores containing predictors and labels, you can combine them so you can input the data into a deep learning network.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. For in-memory data, the `trainingOptions` function provides options to pad and truncate input sequences, however, for out-of-memory data, you must pad and truncate the sequences manually.

Load Training Data

Load the Japanese Vowels data set as described in [1] and [2]. The zip file `japaneseVowels.zip` contains sequences of varying length. The sequences are divided into two folders, `Train` and `Test`, which contain training sequences and test sequences, respectively. In each of these folders, the sequences are divided into subfolders, which are numbered from 1 to 9. The names of these subfolders are the label names. A MAT file represents each sequence. Each sequence is a matrix with 12 rows, with one row for each feature, and a varying number of columns, with one column for each time step. The number of rows is the sequence dimension and the number of columns is the sequence length.

Unzip the sequence data.

```
filename = "japaneseVowels.zip";
outputFolder = fullfile(tempdir,"japaneseVowels");
unzip(filename,outputFolder);
```

For the training predictors, create a file datastore and specify the read function to be the `load` function. The `load` function, loads the data from the MAT-file into a structure array. To read files from the subfolders in the training folder, set the `'IncludeSubfolders'` option to `true`.

```
folderTrain = fullfile(outputFolder,"Train");
fdsPredictorTrain = fileDatastore(folderTrain, ...
    'ReadFcn',@load, ...
    'IncludeSubfolders',true);
```

Preview the datastore. The returned struct contains a single sequence from the first file.

```
preview(fdsPredictorTrain)
```

```
ans = struct with fields:
    X: [12×20 double]
```

For the labels, create a file datastore and specify the read function to be the `readLabel` function, defined at the end of the example. The `readLabel` function extracts the label from the subfolder name.

```
classNames = string(1:9);
fdsLabelTrain = fileDatastore(folderTrain, ...
```

```
    'ReadFcn',@(filename) readLabel(filename,classNames), ...
    'IncludeSubfolders',true);
```

Preview the datastore. The output corresponds to the label of the first file.

```
preview(fdsLabelTrain)
```

```
ans = categorical
     1
```

Transform and Combine Datastores

To input the sequence data from the datastore of predictors to a deep learning network, the mini-batches of the sequences must have the same length. Transform the datastore using the `padSequence` function, defined at the end of the datastore, that pads or truncates the sequences to have length 20.

```
sequenceLength = 20;
tdsTrain = transform(fdsPredictorTrain,@(data) padSequence(data,sequenceLength));
```

Preview the transformed datastore. The output corresponds to the padded sequence from the first file.

```
X = preview(tdsTrain)
```

```
X = 1×1 cell array
    {12×20 double}
```

To input both the predictors and labels from both datastores into a deep learning network, combine them using the `combine` function.

```
cdsTrain = combine(tdsTrain,fdsLabelTrain);
```

Preview the combined datastore. The datastore returns a 1-by-2 cell array. The first element corresponds to the predictors. The second element corresponds to the label.

```
preview(cdsTrain)
```

```
ans = 1×2 cell array
    {12×20 double}    {[1]}
```

Define LSTM Network Architecture

Define the LSTM network architecture. Specify the number of features of the input data as the input size. Specify an LSTM layer with 100 hidden units and to output the last element of the sequence. Finally, specify a fully connected layer with output size equal to the number of classes, followed by a softmax layer and a classification layer.

```
numFeatures = 12;
numClasses = numel(classNames);
numHiddenUnits = 100;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
```

```
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

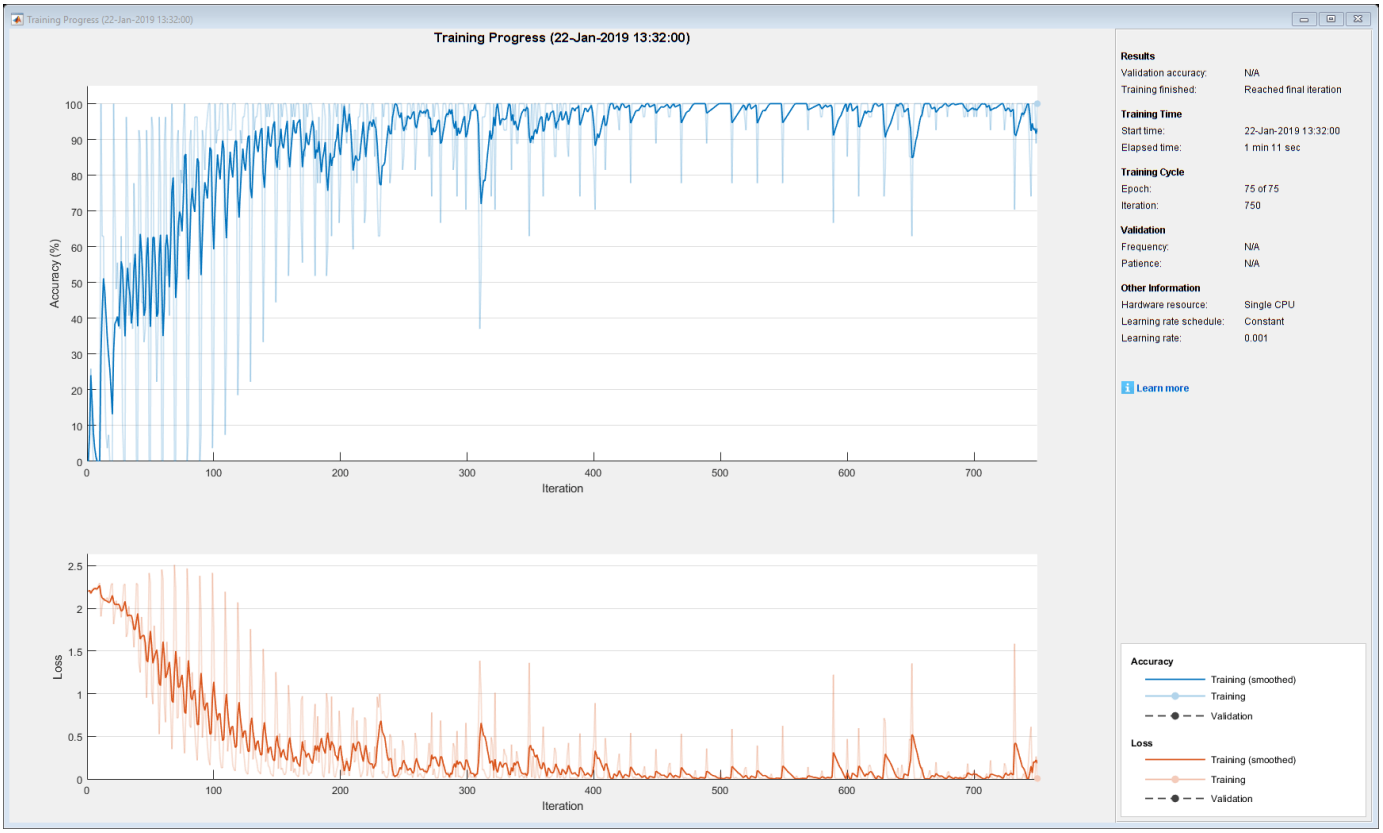
Specify the training options. Set the solver to `'adam'` and `'GradientThreshold'` to 2. Set the mini-batch size to 27 and set the maximum number of epochs to 75. The datastores do not support shuffling, so set `'Shuffle'` to `'never'`.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set `'ExecutionEnvironment'` to `'cpu'`. To train on a GPU, if available, set `'ExecutionEnvironment'` to `'auto'` (the default value).

```
miniBatchSize = 27;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',75, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'Shuffle','never',...
    'Verbose',0, ...
    'Plots','training-progress');
```

Train the LSTM network with the specified training options.

```
net = trainNetwork(cdsTrain,layers,options);
```



Test the Network

Create a transformed datastore containing the held-out test data using the same steps as for the training data.

```
folderTest = fullfile(outputFolder,"Test");

fdsPredictorTest = fileDatastore(folderTest, ...
    'ReadFcn',@load, ...
    'IncludeSubfolders',true);
tdsTest = transform(fdsPredictorTest,@(data) padSequence(data,sequenceLength));
```

Make predictions on the test data using the trained network.

```
YPred = classify(net,tdsTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy on the test data. To get the labels of the test set, create a file datastore with the read function `readLabel` and specify to include subfolders. Specify that the outputs are vertically concatenateable by setting the `'UniformRead'` option to `true`.

```
fdsLabelTest = fileDatastore(folderTest, ...
    'ReadFcn',@(filename) readLabel(filename,classNames), ...
    'IncludeSubfolders',true, ...
    'UniformRead',true);
YTest = readall(fdsLabelTest);

accuracy = mean(YPred == YTest)
```

```
accuracy = 0.9351
```

Functions

The `readLabel` function extracts the label from the specified filename over the categories in `classNames`.

```
function label = readLabel(filename,classNames)

filepath = fileparts(filename);
[~,label] = fileparts(filepath);

label = categorical(string(label),classNames);

end
```

The `padSequence` function pads or truncates the sequence in `data.X` to have the specified sequence length and returns the result in a 1-by-1 cell.

```
function sequence = padSequence(data,sequenceLength)

sequence = data.X;
[C,S] = size(sequence);

if S < sequenceLength
    padding = zeros(C,sequenceLength-S);
    sequence = [sequence padding];
else
    sequence = sequence(:,1:sequenceLength);
end
```

```
sequence = {sequence};

end
```

See Also

`lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer` | `combine` | `transform`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Train Network Using Custom Mini-Batch Datastore for Sequence Data

This example shows how to train a deep learning network on out-of-memory sequence data using a custom mini-batch datastore.

A mini-batch datastore is an implementation of a datastore with support for reading data in batches. Use mini-batch datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications.

This example uses the custom mini-batch datastore `sequenceDatastore.m`. You can adapt this datastore to your data by customizing the datastore functions. For an example showing how to create your own custom mini-batch datastore, see “Develop Custom Mini-Batch Datastore” on page 19-36.

Load Training Data

Load the Japanese Vowels data set as described in [1] and [2]. The zip file `japaneseVowels.zip` contains sequences of varying length. The sequences are divided into two folders, `Train` and `Test`, which contain training sequences and test sequences, respectively. In each of these folders, the sequences are divided into subfolders, which are numbered from `1` to `9`. The names of these subfolders are the label names. A MAT file represents each sequence. Each sequence is a matrix with 12 rows, with one row for each feature, and a varying number of columns, with one column for each time step. The number of rows is the sequence dimension and the number of columns is the sequence length.

Unzip the sequence data.

```
filename = "japaneseVowels.zip";
outputFolder = fullfile(tempdir,"japaneseVowels");
unzip(filename,outputFolder);
```

Create Custom Mini-Batch Datastore

Create a custom mini-batch datastore. The mini-batch datastore `sequenceDatastore` reads data from a folder and gets the labels from the subfolder names. To use this datastore, first save the file `sequenceDatastore.m` to the path.

Create a datastore containing the sequence data using `sequenceDatastore`.

```
folderTrain = fullfile(outputFolder,"Train");
dsTrain = sequenceDatastore(folderTrain)
```

```
dsTrain =
  sequenceDatastore with properties:

           Datastore: [1×1 matlab.io.datastore.FileDatastore]
              Labels: [270×1 categorical]
          NumClasses: 9
    SequenceDimension: 12
        MiniBatchSize: 128
      NumObservations: 270
```

Define LSTM Network Architecture

Define the LSTM network architecture. Specify the sequence dimension of the input data as the input size. Specify an LSTM layer with 100 hidden units and to output the last element of the sequence. Finally, specify a fully connected layer with output size equal to the number of classes, followed by a softmax layer and a classification layer.

```
inputSize = dsTrain.SequenceDimension;
numClasses = dsTrain.NumClasses;
numHiddenUnits = 100;
layers = [
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

Specify the training options. Specify `'adam'` as the solver and `'GradientThreshold'` as 1. Set the mini-batch size to 27 and set the maximum number of epochs to 75. To ensure that the datastore creates mini-batches of the size that the `trainNetwork` function expects, also set the mini-batch size of the datastore to the same value.

Because the mini-batches are small with short sequences, the CPU is better suited for training. Set `'ExecutionEnvironment'` to `'cpu'`. To train on a GPU, if available, set `'ExecutionEnvironment'` to `'auto'` (the default value).

```
miniBatchSize = 27;
options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'MaxEpochs',75, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',1, ...
    'Verbose',0, ...
    'Plots','training-progress');
dsTrain.MiniBatchSize = miniBatchSize;
```

Train the LSTM network with the specified training options.

```
net = trainNetwork(dsTrain,layers,options);
```

Test the Network

Create a sequence datastore from the test data.

```
folderTest = fullfile(outputFolder,"Test");
dsTest = sequenceDatastore(folderTest);
```

Classify the test data. Specify the same mini-batch size as for the training data. To ensure that the datastore creates mini-batches of the size that the `classify` function expects, also set the mini-batch size of the datastore to the same value.

```
dsTest.MiniBatchSize = miniBatchSize;
YPred = classify(net,dsTest,'MiniBatchSize',miniBatchSize);
```

Calculate the classification accuracy of the predictions.

```
YTest = dsTest.Labels;
acc = sum(YPred == YTest)./numel(YTest)
```

```
acc = 0.9432
```

References

[1] Kudo, M., J. Toyama, and M. Shimbo. "Multidimensional Curve Classification Using Passing-Through Regions." *Pattern Recognition Letters*. Vol. 20, No. 11–13, pp. 1103–1111.

[2] Kudo, M., J. Toyama, and M. Shimbo. *Japanese Vowels Data Set*. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

See Also

`trainNetwork` | `trainingOptions` | `lstmLayer` | `sequenceInputLayer`

Related Examples

- “Develop Custom Mini-Batch Datastore” on page 19-36
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47
- “Long Short-Term Memory Networks” on page 1-75
- “Deep Learning in MATLAB” on page 1-2

Classify Out-of-Memory Text Data Using Deep Learning

This example shows how to classify out-of-memory text data with a deep learning network using a transformed datastore.

A transformed datastore transforms or processes data read from an underlying datastore. You can use a transformed datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use transformed datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. The `trainingOptions` function provides options to pad and truncate input sequences, however, these options are not well suited for sequences of word vectors. Furthermore, this function does not support padding data in a custom datastore. Instead, you must pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

The “Classify Text Data Using Deep Learning” (Text Analytics Toolbox) example manually truncates and pads all the documents to the same length. This process adds lots of padding to very short documents and discards lots of data from very long documents.

Alternatively, to prevent adding too much padding or discarding too much data, create a transformed datastore that inputs mini-batches into the network. The datastore created in this example converts mini-batches of documents to sequences or word indices and left-pads each mini-batch to the length of the longest document in the mini-batch.

Load Pretrained Word Embedding

The datastore requires a word embedding to convert documents to sequences of vectors. Load a pretrained word embedding using `fastTextWordEmbedding`. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Load Data

Create a tabular text datastore from the data in `factoryReports.csv`. Specify to read the data from the `"Description"` and `"Category"` columns only.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
ttdsTrain = tabularTextDatastore(filenameTrain,'SelectedVariableNames',[textName labelName]);
```

View a preview of the datastore.

```
preview(ttdsTrain)
```

```
ans=8×2 table
                                Description                                                Category
    ____________________________________________________________________________    ____________________

    {'Items are occasionally getting stuck in the scanner spools.'        }    {'Mechanical Failu
    {'Loud rattling and banging sounds are coming from assembler pistons.'}    {'Mechanical Failu
    {'There are cuts to the power when starting the plant.'               }    {'Electronic Failu
```

```
    {'Fried capacitors in the assembler.'                                  }    {'Electronic Failu
    {'Mixer tripped the fuses.'                                            }    {'Electronic Failu
    {'Burst pipe in the constructing agent is spraying coolant.'           }    {'Leak'
    {'A fuse is blown in the mixer.'                                       }    {'Electronic Failu
    {'Things continue to tumble off of the belt.'                          }    {'Mechanical Failu
```

Transform Datastore

Create a custom transform function that converts data read from the datastore to a table containing the predictors and the responses. The `transformText` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are *C*-by-*S* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension and *S* is the sequence length. The responses are categorical labels over the classes.

To get the class names, read the labels from the training data using the `readLabels` function, listed and the end of the example, and find the unique class names.

```
labels = readLabels(ttdsTrain,labelName);
classNames = unique(labels);
numObservations = numel(labels);
```

Because tabular text datastores can read multiple rows of data in a single read, you can process a full mini-batch of data in the transform function. To ensure that the transform function processes a full mini-batch of data, set the read size of the tabular text datastore to the mini-batch size that will be used for training.

```
miniBatchSize = 64;
ttdsTrain.ReadSize = miniBatchSize;
```

To convert the output of the tabular text data to sequences for training, transform the datastore using the `transform` function.

```
tdsTrain = transform(ttdsTrain, @(data) transformText(data,emb,classNames))
```

```
tdsTrain =
  TransformedDatastore with properties:

       UnderlyingDatastore: [1×1 matlab.io.datastore.TabularTextDatastore]
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@(data)transformText(data,emb,classNames)}
               IncludeInfo: 0
```

Preview of the transformed datastore. The predictors are *C*-by-*S* arrays, where *S* is the sequence length and *C* is the number of features (the embedding dimension). The responses are the categorical labels.

```
preview(tdsTrain)
```

```
ans=8×2 table
      predictors           responses
    _______________    ________________

    {300×11 single}    Mechanical Failure
    {300×11 single}    Mechanical Failure
    {300×11 single}    Electronic Failure
    {300×11 single}    Electronic Failure
```

```
    {300×11 single}    Electronic Failure
    {300×11 single}    Leak
    {300×11 single}    Electronic Failure
    {300×11 single}    Mechanical Failure
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to the embedding dimension. Next, include an LSTM layer with 180 hidden units. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with output size equal to the number of classes, a softmax layer, and a classification layer.

```
numFeatures = emb.Dimension;
numHiddenUnits = 180;
numClasses = numel(classNames);
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

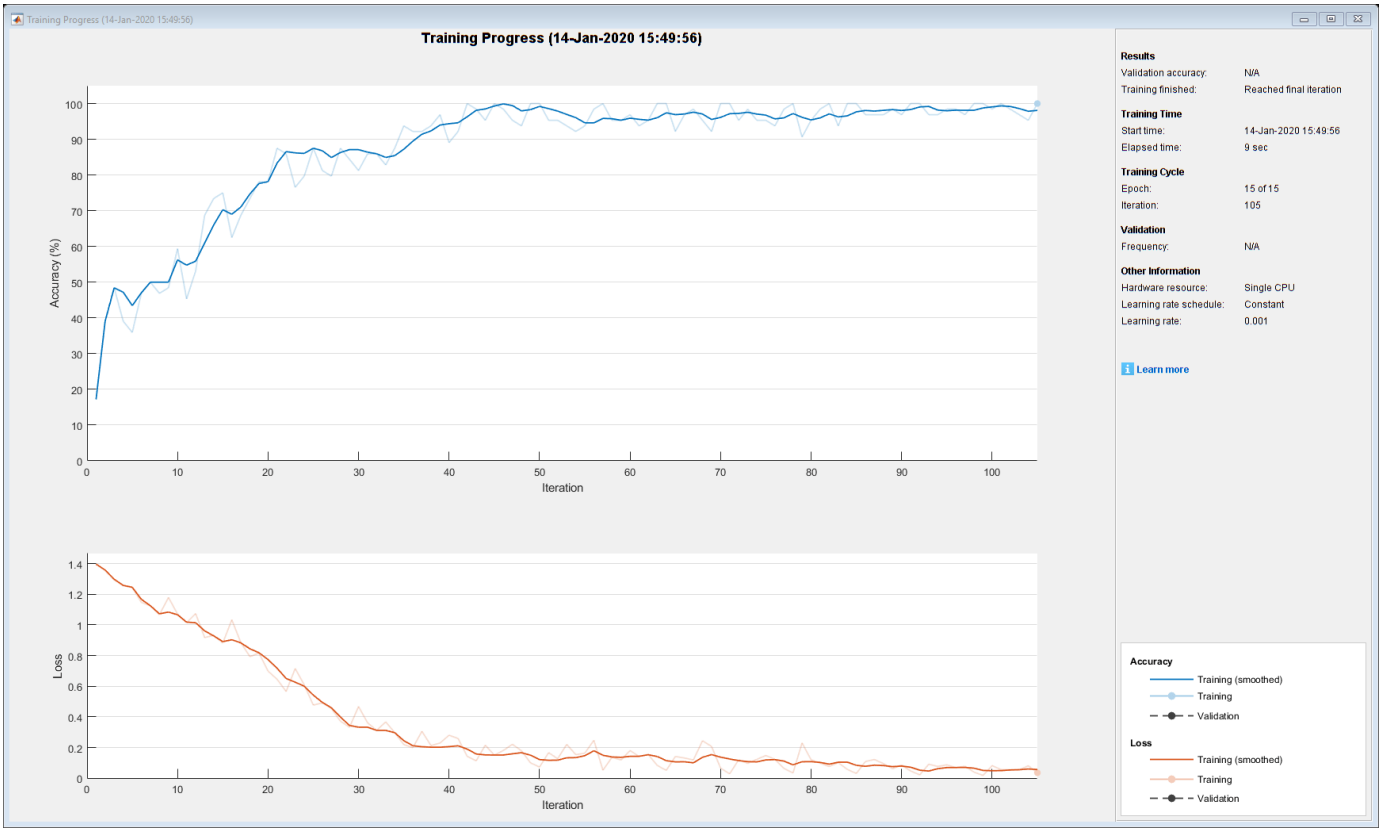
Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore does not support shuffling, so set `'Shuffle'`, to `'never'`. Validate the network once per epoch. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

By default, `trainNetwork` uses a GPU if one is available. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU. Training using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MaxEpochs',15, ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(tdsTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the training documents.

```
documentsNew = preprocessText(reportsNew);
```

Convert the text data to sequences of embedding vectors using `doc2sequence`.

```
XNew = doc2sequence(emb,documentsNew);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,XNew)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
     Mechanical Failure
```

Transform Text Function

The `transformText` function takes the data read from a `tabularTextDatastore` object and returns a table of predictors and responses. The predictors are *C*-by-*S* arrays of word vectors given by the word embedding `emb`, where *C* is the embedding dimension and *S* is the sequence length. The responses are categorical labels over the classes in `classNames`.

```
function dataTransformed = transformText(data,emb,classNames)

% Preprocess documents.
textData = data{:,1};
documents = preprocessText(textData);

% Convert to sequences.
predictors = doc2sequence(emb,documents);

% Read labels.
labels = data{:,2};
responses = categorical(labels,classNames);

% Convert data to table.
dataTransformed = table(predictors,responses);

end
```

Preprocessing Function

The function `preprocessText` performs these steps:

1. Tokenize the text using `tokenizedDocument`.
2. Convert the text to lowercase using `lower`.
3. Erase the punctuation using `erasePunctuation`.

```
function documents = preprocessText(textData)

documents = tokenizedDocument(textData);
documents = lower(documents);
documents = erasePunctuation(documents);

end
```

Read Labels Function

The `readLabels` function creates a copy of the `tabularTextDatastore` object `ttds` and reads the labels from the `labelName` column.

```
function labels = readLabels(ttds,labelName)

ttdsNew = copy(ttds);
ttdsNew.SelectedVariableNames = labelName;
tbl = readall(ttdsNew);
labels = tbl.(labelName);
```

```
end
```

See Also

`fastTextWordEmbedding` | `wordEmbeddingLayer` | `doc2sequence` | `tokenizedDocument` | `lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer` | `transform`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-15
- “Long Short-Term Memory Networks” on page 1-75
- “List of Deep Learning Layers” on page 1-21
- “Deep Learning Tips and Tricks” on page 1-67

Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore

This example shows how to classify out-of-memory text data with a deep learning network using a custom mini-batch datastore.

A mini-batch datastore is an implementation of a datastore with support for reading data in batches. You can use a mini-batch datastore as a source of training, validation, test, and prediction data sets for deep learning applications. Use mini-batch datastores to read out-of-memory data or to perform specific preprocessing operations when reading batches of data.

When training the network, the software creates mini-batches of sequences of the same length by padding, truncating, or splitting the input data. The `trainingOptions` function provides options to pad and truncate input sequences, however, these options are not well suited for sequences of word vectors. Furthermore, this function does not support padding data in a custom datastore. Instead, you must pad and truncate the sequences manually. If you *left-pad* and truncate the sequences of word vectors, then the training might improve.

The “Classify Text Data Using Deep Learning” (Text Analytics Toolbox) example manually truncates and pads all the documents to the same length. This process adds lots of padding to very short documents and discards lots of data from very long documents.

Alternatively, to prevent adding too much padding or discarding too much data, create a custom mini-batch datastore that inputs mini-batches into the network. The custom mini-batch datastore `textDatastore.m` converts mini-batches of documents to sequences or word indices and left-pads each mini-batch to the length of the longest document in the mini-batch. For sorted data, this datastore can help reduce the amount of padding added to the data since documents are not padded to a fixed length. Similarly, the datastore does not discard any data from the documents.

This example uses the custom mini-batch datastore `textDatastore`, attached to this example as a supporting file. To access this file, open the example as a live script. You can adapt this datastore to your data by customizing the functions. For an example showing how to create your own custom mini-batch datastore, see “Develop Custom Mini-Batch Datastore” on page 19-36.

Load Pretrained Word Embedding

The datastore `textDatastore` requires a word embedding to convert documents to sequences of vectors. Load a pretrained word embedding using `fastTextWordEmbedding`. This function requires Text Analytics Toolbox™ Model *for fastText English 16 Billion Token Word Embedding* support package. If this support package is not installed, then the function provides a download link.

```
emb = fastTextWordEmbedding;
```

Create Mini-Batch Datastore of Documents

Create a datastore that contains the data for training. The custom mini-batch datastore `textDatastore` reads predictors and labels from a CSV file. For the predictors, the datastore converts the documents into sequences of word indices and for the responses, the datastore returns a categorical label for each document. For more information about creating custom mini-batch datastores, see “Develop Custom Mini-Batch Datastore” on page 19-36.

For the training data, specify the CSV file `"factoryReports.csv"` and that the text and labels are in the columns `"Description"` and `"Category"` respectively.

```
filenameTrain = "factoryReports.csv";
textName = "Description";
labelName = "Category";
dsTrain = textDatastore(filenameTrain,textName,labelName,emb)
```

```
dsTrain =
  textDatastore with properties:

            ClassNames: ["Electronic Failure"    "Leak"    "Mechanical Failure"    "Software Fai
             Datastore: [1×1 matlab.io.datastore.TransformedDatastore]
    EmbeddingDimension: 300
             LabelName: "Category"
         MiniBatchSize: 128
            NumClasses: 4
       NumObservations: 480
```

Create and Train LSTM Network

Define the LSTM network architecture. To input sequence data into the network, include a sequence input layer and set the input size to the embedding dimension. Next, include an LSTM layer with 180 hidden units. To use the LSTM layer for a sequence-to-label classification problem, set the output mode to `'last'`. Finally, add a fully connected layer with output size equal to the number of classes, a softmax layer, and a classification layer.

```
numFeatures = dsTrain.EmbeddingDimension;
numHiddenUnits = 180;
numClasses = dsTrain.NumClasses;

layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

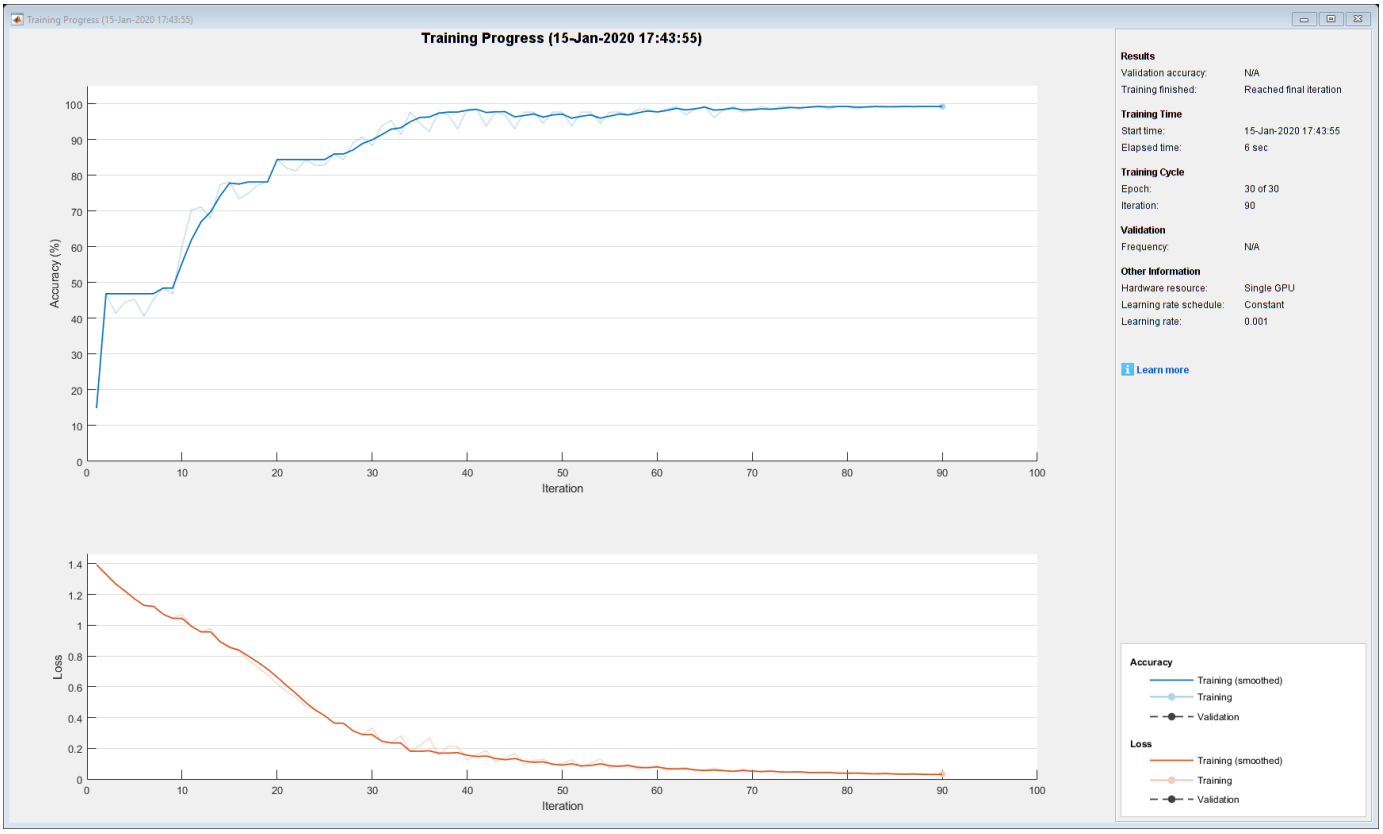
Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore `textDatastore.m` does not support shuffling, so set `'Shuffle'`, to `'never'`. For an example showing how to implement a datastore with support for shuffling, see “Develop Custom Mini-Batch Datastore” on page 19-36. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

By default, `trainNetwork` uses a GPU if one is available. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU. Training using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Support by Release” (Parallel Computing Toolbox).

```
miniBatchSize = 128;
numObservations = dsTrain.NumObservations;
numIterationsPerEpoch = floor(numObservations / miniBatchSize);

options = trainingOptions('adam', ...
    'MiniBatchSize',miniBatchSize, ...
    'GradientThreshold',2, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the LSTM network using the `trainNetwork` function.

```
net = trainNetwork(dsTrain,layers,options);
```



Predict Using New Data

Classify the event type of three new reports. Create a string array containing the new reports.

```
reportsNew = [
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."];
```

Preprocess the text data using the preprocessing steps as the datastore `textDatastore`.

```
documents = tokenizedDocument(reportsNew);
documents = lower(documents);
documents = erasePunctuation(documents);
predictors = doc2sequence(emb,documents);
```

Classify the new sequences using the trained LSTM network.

```
labelsNew = classify(net,predictors)
```

```
labelsNew = 3×1 categorical
     Leak
     Electronic Failure
```

```
Mechanical Failure
```

See Also

`wordEmbeddingLayer` | `doc2sequence` | `tokenizedDocument` | `lstmLayer` | `trainNetwork` | `trainingOptions` | `sequenceInputLayer` | `wordcloud` | `extractHTMLText` | `findElement` | `htmlTree`

Related Examples

- “Generate Text Using Deep Learning” on page 4-180
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Data Sets for Deep Learning

Use these data sets to get started with deep learning applications.

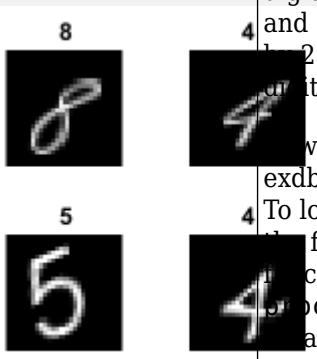
Image Data Sets

Data Set	Description	Task

Digits	The Digits data set consists of 10,000 synthetic grayscale images of handwritten digits. Each image is 28-by-28 pixels and has an associated label denoting which digit the image represents (0–9). Each image has been rotated by a certain angle. When loading the images as in-memory numeric arrays, you can also load the rotation angle of the image.
	
	Load the Digits data as in-memory numeric arrays using the `digitTrain4DArrayData` and `digitTest4DArrayData` functions.
	
	`[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;`
	`[XTest,YTest,anglesTest] = digitTest4DArrayData;`
	
	For examples showing how to process this data for deep learning, see “Monitor Deep Learning Training Progress” on page 5-115 and “Train Convolutional Neural Network for Regression” on page 3-53.
		Image classification and image regression

	Load the Digits data as an image datastore using the `imageDatastore` function and specify the folder containing the image data.
	
	`dataFolder = fullfile(toolboxdir('nnet'),'nndemos','nndatasets','DigitDataset');`
	`imds = imageDatastore(dataFolder, ...`
	`    'IncludeSubfolders',true,'LabelSource','foldernames');`
	
	For an example showing how to process this data for deep learning, see “Create Simple Deep Learning Network for Classification” on page 3-47.
		Image classification

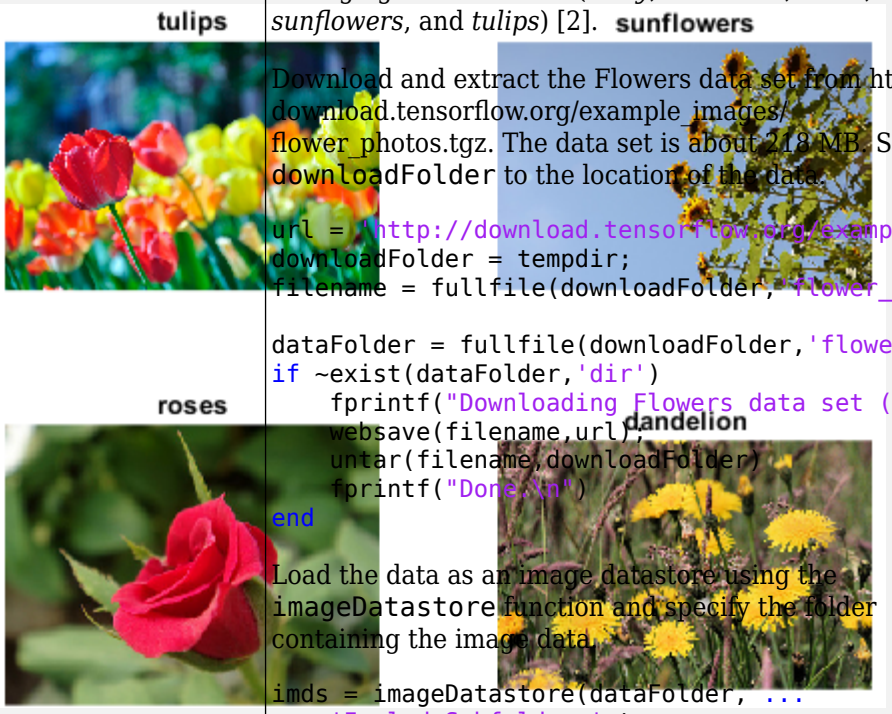
Data Set	Description	Task

MNIST
 
(Representative example)
	The MNIST data set consists of 70,000 handwritten digits split into training and test partitions of 60,000 and 10,000 images, respectively. Each image is 28-by-28 pixels and has an associated label denoting which digit the image represents (0–9).
	
	Download the MNIST files from http://yann.lecun.com/exdb/mnist/ and load the data set into the workspace. To load the data from the files as MATLAB arrays, place the files in the working directory, then use the helper functions `processImagesMNIST` and `processLabelsMNIST`, which are used in the example "Train Variational Autoencoder (VAE) to Generate Images" on page 3-167.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','nnet','main'));`
	`filenameImagesTrain = 'train-images-idx3-ubyte.gz';`
	`filenameLabelsTrain = 'train-labels-idx1-ubyte.gz';`
	`filenameImagesTest = 't10k-images-idx3-ubyte.gz';`
	`filenameLabelsTest = 't10k-labels-idx1-ubyte.gz';`
	
	`XTrain = processImagesMNIST(filenameImagesTrain);`
	`YTrain = processLabelsMNIST(filenameLabelsTrain);`
	`XTest = processImagesMNIST(filenameImagesTest);`
	`YTest = processLabelsMNIST(filenameLabelsTest);`
	
	For an example showing how to process this data for deep learning, see "Train Variational Autoencoder (VAE) to Generate Images" on page 3-167.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Image classification

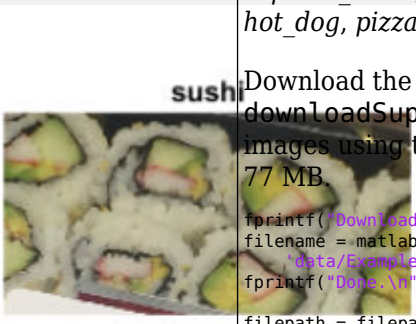
Data Set	Description	Task

Omniglot	The Omniglot data set contains character sets for 50 alphabets, divided into 30 sets for training and 20 sets for testing [1]. Each alphabet contains a number of characters, from 14 for Ojibwe (Canadian Aboriginal syllabics) to 55 for Tifinagh. Finally, each character has 20 handwritten observations.
	
	Download and extract the Omniglot data set from https://github.com/brendenlake/omniglot. Set `downloadFolder` to the location of the data.
	
	`downloadFolder = tempdir;`
	
	`url = "https://github.com/brendenlake/omniglot/raw/master/python";`
	`urlTrain = url + "/images_background.zip";`
	`urlTest = url + "/images_evaluation.zip";`
	
	`filenameTrain = fullfile(downloadFolder,"images_background.zip");`
	`filenameTest = fullfile(downloadFolder,"images_evaluation.zip");`
	
	`dataFolderTrain = fullfile(downloadFolder,"images_background");`
	`dataFolderTest = fullfile(downloadFolder,"images_evaluation");`
	
	`if ~exist(dataFolderTrain,"dir")`
	`  fprintf("Downloading Omniglot training data set (4.5 MB)... ")`
	`  websave(filenameTrain,urlTrain);`
	`  unzip(filenameTrain,downloadFolder);`
	`  fprintf("Done.\n")`
	`end`
	
	`if ~exist(dataFolderTest,"dir")`
	`  fprintf("Downloading Omniglot test data (3.2 MB)... ")`
	`  websave(filenameTest,urlTest);`
	`  unzip(filenameTest,downloadFolder);`
	`  fprintf("Done.\n")`
	`end`
	
	To load the training and test data as image datastores, use the `imageDatastore` function. Specify the labels manually by extracting the labels from the file names and setting the `Labels` property.
	
	`imdsTrain = imageDatastore(dataFolderTrain, ...`
	`  'IncludeSubfolders',true, ...`
	`  'LabelSource','none');`
	
	`files = imdsTrain.Files;`
	`parts = split(files,filesep);`
	`labels = join(parts(:,(end-2):(end-1)),'_');`
	`imdsTrain.Labels = categorical(labels);`
	
	`imdsTest = imageDatastore(dataFolderTest, ...`
	`  'IncludeSubfolders',true, ...`
	`  'LabelSource','none');`
	
	`files = imdsTest.Files;`
	`parts = split(files,filesep);`
	`labels = join(parts(:,(end-2):(end-1)),'_');`
	`imdsTest.Labels = categorical(labels);`
		Image similarity

Data Set	Description	Task
	For an example showing how to process this data for deep learning, see “Train a Siamese Network to Compare Images” on page 3-128.	

Flowers
 
 	
Image credits: [3] [4] [5] [6]
	The Flowers data set contains 3670 images of flowers belonging to five classes (*daisy*, *dandelion*, *roses*, *sunflowers*, and *tulips*) [2].
	
	Download and extract the Flowers data set from http://download.tensorflow.org/example_images/flower_photos.tgz. The data set is about 218 MB. Set `downloadFolder` to the location of the data.
	
	`url = 'http://download.tensorflow.org/example_images/flower_photos.tgz';`
	`downloadFolder = tempdir;`
	`filename = fullfile(downloadFolder,'flower_dataset.tgz');`
	
	`dataFolder = fullfile(downloadFolder,'flower_photos');`
	`if ~exist(dataFolder,'dir')`
	`  fprintf("Downloading Flowers data set (218 MB)... ")`
	`  websave(filename,url);`
	`  untar(filename,downloadFolder)`
	`  fprintf("Done.\n")`
	`end`
	
	Load the data as an image datastore using the `imageDatastore` function and specify the folder containing the image data.
	
	`imds = imageDatastore(dataFolder, ...`
	`  'IncludeSubfolders',true, ...`
	`  'LabelSource','foldernames');`
	
	For an example showing how to process this data for deep learning, see “Train Generative Adversarial Network (GAN)” on page 3-76.
		Image classification

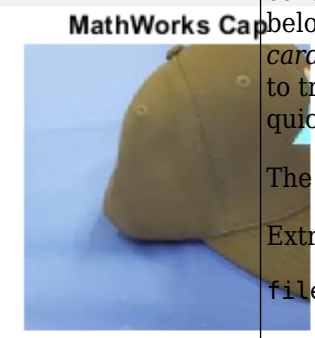
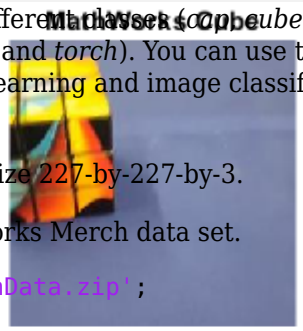
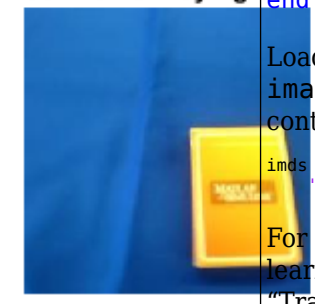
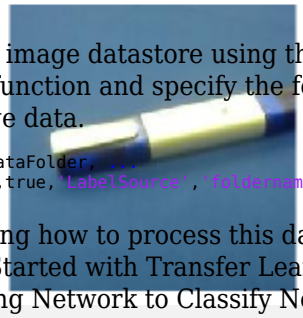
Data Set	Description	Task

Example Food Images	The Example Food Images data set contains 978 photographs of food in nine classes (*caesar_salad*, *caprese_salad*, *french_fries*, *greek_salad*, *hamburger*, *hot_dog*, *pizza*, *sashimi*, and *sushi*).
	 
	 
	Download the Example Food Images data set using the `downloadSupportFile` function and extract the images using the `unzip` function. This data set is about 77 MB.
	`fprintf('Downloading Example Food Image data set (77 MB)... ')`
	`filename = matlab.internal.examples.downloadSupportFile('nnet', ...`
	`    'data/ExampleFoodImageDataset.zip');`
	`fprintf("Done.\n")`
	`filepath = fileparts(filename);`
	`dataFolder = fullfile(filepath,'ExampleFoodImageDataset');`
	`unzip(filename,dataFolder);`
	For an example showing how to process this data for deep learning, see “View Network Behavior Using tsne” on page 5-129.
	 
	 
		Image classification

Data Set	Description	Task

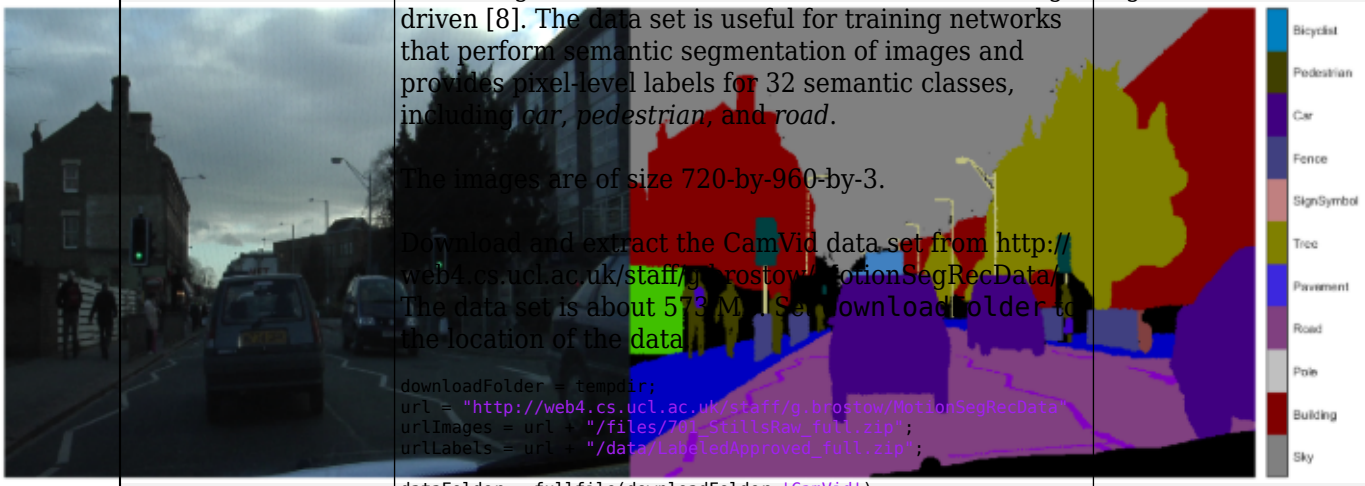
CIFAR-10
 
 	
(Representative example)
	The CIFAR-10 data set contains 60,000 color images of size 32-by-32 pixels, belonging to 10 classes (*airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*) [7]. There are 6,000 images per class.
	
	The data set is split into a training set with 50,000 images and a test set with 10,000 images. This data set is one of the most widely used data sets for testing new image classification models.
	
	Download and extract the CIFAR-10 data set from https://www.cs.toronto.edu/%7Ekriz/cifar-10-matlab.tar.gz. The data set is about 175 MB. Set `downloadFolder` to the location of the data.
	
	`url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';`
	`downloadFolder = tempdir;`
	`filename = fullfile(downloadFolder,'cifar-10-matlab.tar.gz');`
	
	`dataFolder = fullfile(downloadFolder,'cifar-10-batches-mat');`
	`if ~exist(dataFolder,'dir')`
	`  fprintf("Downloading CIFAR-10 dataset (175 MB)... ");`
	`  websave(filename,url);`
	`  untar(filename,downloadFolder);`
	`  fprintf("Done.\n")`
	`end`
	
	Convert the data to numeric arrays using the helper function `loadCIFARData`, which is used in the example “Train Residual Network for Image Classification” on page 3-13.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','nnet','main'));`
	`[XTrain,YTrain,XValidation,YValidation] = loadCIFARData(downloadFolder);`
	
	For an example showing how to process this data for deep learning, see “Train Residual Network for Image Classification” on page 3-13.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Image classification

Data Set	Description	Task

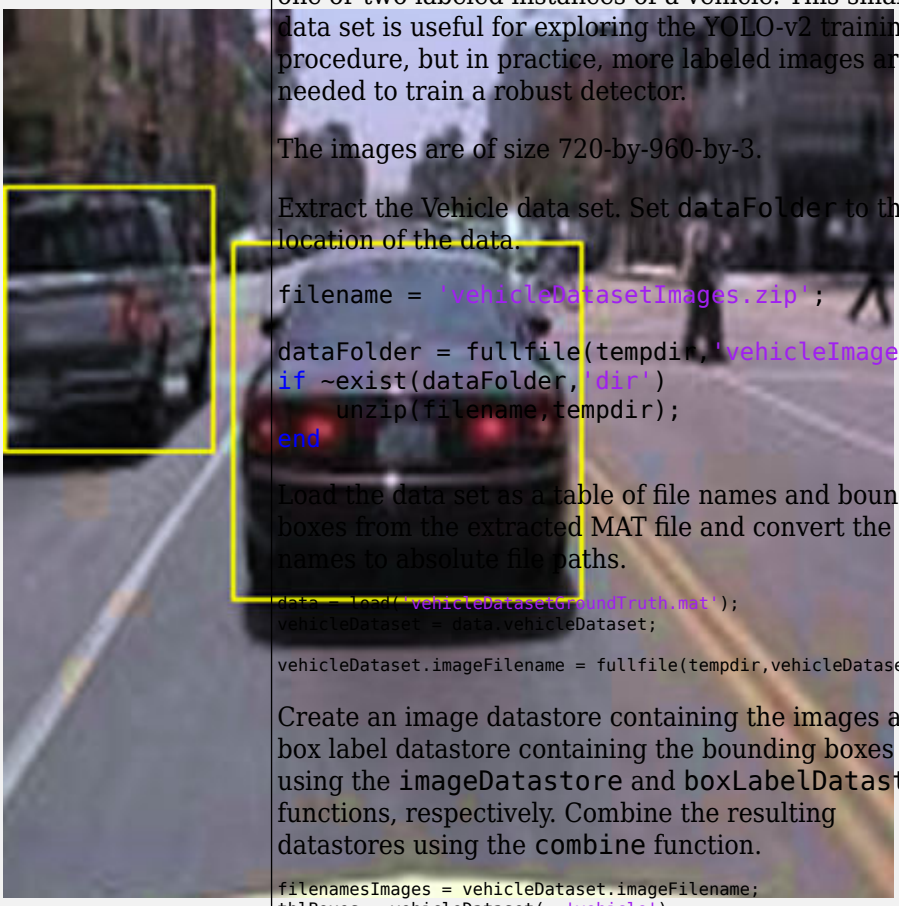
MathWorks Merch    	The MathWorks Merch data set is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*). You can use this data set to try out transfer learning and image classification quickly.
	
	The images are of size 227-by-227-by-3.
	
	Extract the MathWorks Merch data set.
	
	`filename = 'MerchData.zip';`
	
	`dataFolder = fullfile(tempdir,'MerchData');`
	`if ~exist(dataFolder,'dir')`
	`  unzip(filename,tempdir);`
	`end`
	
	Load the data as an image datastore using the `imageDatastore` function and specify the folder containing the image data.
	
	`imds = imageDatastore(dataFolder, ...`
	`  'IncludeSubfolders',true,'LabelSource','foldernames');`
	
	For examples showing how to process this data for deep learning, see “Get Started with Transfer Learning” and “Train Deep Learning Network to Classify New Images” on page 3-6.
		Image classification

Data Set	Description	Task

CamVid	The CamVid data set is a collection of images containing street-level views obtained from cars being driven [8]. The data set is useful for training networks that perform semantic segmentation of images and provides pixel-level labels for 32 semantic classes, including *car*, *pedestrian*, and *road*.
	
	The images are of size 720-by-960-by-3.
	
	Download and extract the CamVid data set from http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/. The data set is about 573 MB. Set `downloadFolder` to the location of the data.
	
	`downloadFolder = tempdir;`
	`url = "http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData";`
	`urlImages = url + "/files/701_StillsRaw_full.zip";`
	`urlLabels = url + "/data/LabeledApproved_full.zip";`
	`dataFolder = fullfile(downloadFolder,'CamVid');`
	`dataFolderImages = fullfile(dataFolder,'images');`
	`dataFolderLabels = fullfile(dataFolder,'labels');`
	
	`filenameLabels = fullfile(dataFolder,'labels.zip');`
	`filenameImages = fullfile(dataFolder,'images.zip');`
	
	`if ~exist(filenameLabels, 'file') || ~exist(imagesZip,'file')`
	`  mkdir(dataFolder)`
	
	`  fprintf("Downloading CamVid data set images (557 MB)... ");`
	`  websave(filenameImages, urlImages);`
	`  unzip(filenameImages, dataFolderImages);`
	`  fprintf("Done.\n")`
	
	`  fprintf("Downloading CamVid data set labels (16 MB)... ");`
	`  websave(filenameLabels, urlLabels);`
	`  unzip(filenameLabels, dataFolderLabels);`
	`  fprintf("Done.\n")`
	`end`
	
	Load the data as a pixel label datastore using the `pixelLabelDatastore` function and specify the folder containing the label data, the classes, and the label IDs. To make training easier, group some of the original classes to reduce the number of classes from 32 to 11. To get the label IDs, use the helper function `camvidPixelLabelIDs`, which is used in the example “Semantic Segmentation Using Deep Learning” on page 8-126.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','deeplearning_shared','main'));`
	`imds = imageDatastore(dataFolderImages,'IncludeSubfolders',true);`
	
	`classes = ["Sky" "Building" "Pole" "Road" "Pavement" "Tree" ...`
	`  "SignSymbol" "Fence" "Car" "Pedestrian" "Bicyclist"];`
	
	`labelIDs = camvidPixelLabelIDs;`
	
	`pxds = pixelLabelDatastore(dataFolderLabels,classes,labelIDs);`
	
	For an example showing how to process this data for deep learning, see “Semantic Segmentation Using Deep Learning” on page 8-126.
		Semantic segmentation



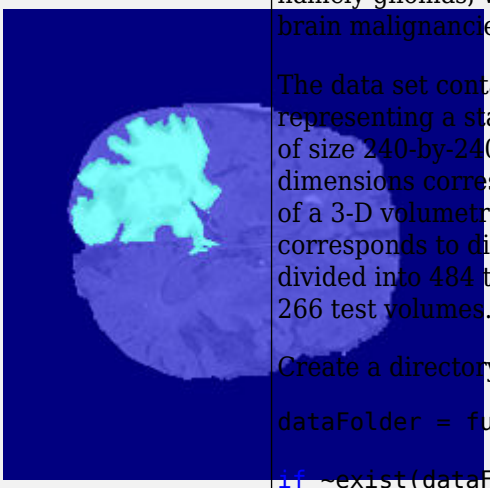
Data Set	Description	Task
	To restore the path, use the `path` function. `path(oldpath);`	

Vehicle
 	
 
	The Vehicle data set consists of 295 images containing one or two labeled instances of a vehicle. This small data set is useful for exploring the YOLO-v2 training procedure, but in practice, more labeled images are needed to train a robust detector.
	
	The images are of size 720-by-960-by-3.
	
	Extract the Vehicle data set. Set `dataFolder` to the location of the data.
	
	`filename = 'vehicleDatasetImages.zip';`
	
	`dataFolder = fullfile(tempdir,'vehicleImages');`
	`if ~exist(dataFolder,'dir')`
	`  unzip(filename,tempdir);`
	`end`
	
	Load the data set as a table of file names and bounding boxes from the extracted MAT file and convert the file names to absolute file paths.
	
	`data = load('vehicleDatasetGroundTruth.mat');`
	`vehicleDataset = data.vehicleDataset;`
	
	`vehicleDataset.imageFilename = fullfile(tempdir,vehicleDataset.imageFilename);`
	
	Create an image datastore containing the images and a box label datastore containing the bounding boxes using the `imageDatastore` and `boxLabelDatastore` functions, respectively. Combine the resulting datastores using the `combine` function.
	
	`filenamesImages = vehicleDataset.imageFilename;`
	`tblBoxes = vehicleDataset(:,'vehicle');`
	
	`imds = imageDatastore(filenamesImages);`
	`blds = boxLabelDatastore(tblBoxes);`
	
	`cds = combine(imds,blds);`
	
	For an example showing how to process this data for deep learning, see “Object Detection Using YOLO v2 Deep Learning” on page 8-115.
		Object detection

Data Set	Description	Task

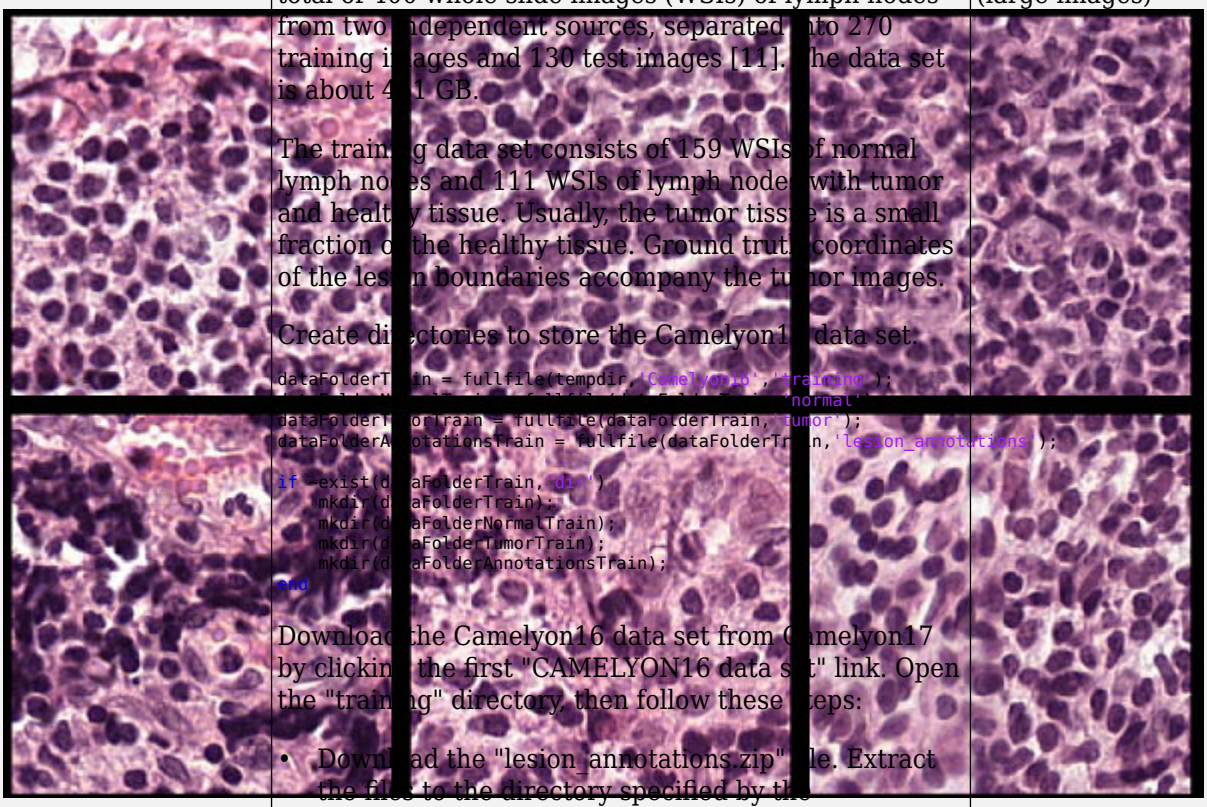
RIT-18	The RIT-18 data set contains image data captured by a drone over Hamlin Beach State Park in New York state [9]. The data contains labeled training, validation, and test sets, with 18 object class labels including *road markings*, *tree*, and *building*. The data set is about 3 GB.
	
	Download the RIT-18 data set from https://www.cis.rit.edu/%7Ermk6217/rit18_data.mat. Set `downloadFolder` to the location of the data.
	
	`downloadFolder = tempdir;`
	`url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';`
	`filename = fullfile(downloadFolder,'rit18_data.mat');`
	
	`if ~exist(filename,'file')`
	`  fprintf("Downloading Hamlin Beach data set (3 GB)... ");`
	`  websave(filename,url);`
	`  fprintf("Done.\n")`
	`end`
	
	For an example showing how to process this data for deep learning, see "Semantic Segmentation of Multispectral Images Using Deep Learning" on page 8-154.
		Semantic segmentation

Data Set	Description	Task

BraTS 	The BraTS data set contains MRI scans of brain tumors, namely gliomas, which are the most common primary brain malignancies [10].
	
	The data set contains 750 4-D volumes, each representing a stack of 3-D images. Each 4-D volume is of size 240-by-240-by-155-by-4, where the first three dimensions correspond to the height, width, and depth of a 3-D volumetric image. The fourth dimension corresponds to different scan modalities. The data set is divided into 484 training volumes with voxel labels and 266 test volumes. The data set is about 7 GB.
	
	Create a directory to store the BraTS data set.
	
	`dataFolder = fullfile(tempdir,'BraTS');`
	
	`if ~exist(dataFolder,'dir')`
	`  mkdir(dataFolder);`
	`end`
	
	Download the BraTS data from Medical Segmentation Decathlon by clicking the "Download Data" link. Download the "Task01_BrainTumour.tar" file.
	
	Extract the TAR file into the directory specified by the `dataFolder` variable. If the extraction is successful, then `dataFolder` contains a directory named `Task01_BrainTumour` that has three subdirectories: `imagesTr`, `imagesTs`, and `labelsTr`.
	
	For an example showing how to process this data for deep learning, see “3-D Brain Tumor Segmentation Using Deep Learning” on page 8-171.
		Semantic segmentation

Data Set	Description	Task

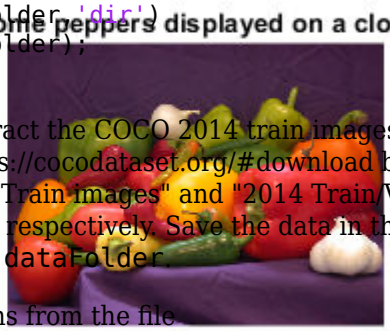
Camelyon16	The data from the Camelyon16 challenge contains a total of 400 whole-slide images (WSIs) of lymph nodes from two independent sources, separated into 270 training images and 130 test images [11]. The data set is about 4.1 GB.
	
	The training data set consists of 159 WSIs of normal lymph nodes and 111 WSIs of lymph nodes with tumor and healthy tissue. Usually, the tumor tissue is a small fraction of the healthy tissue. Ground truth coordinates of the lesion boundaries accompany the tumor images.
	
	Create directories to store the Camelyon16 data set:
	
	`dataFolderTrain = fullfile(tempdir,'Camelyon16','training');`
	`dataFolderNormalTrain = fullfile(dataFolderTrain,'normal');`
	`dataFolderTumorTrain = fullfile(dataFolderTrain,'tumor');`
	`dataFolderAnnotationsTrain = fullfile(dataFolderTrain,'lesion_annotations');`
	
	`if ~exist(dataFolderTrain,'dir')`
	`  mkdir(dataFolderTrain);`
	`  mkdir(dataFolderNormalTrain);`
	`  mkdir(dataFolderTumorTrain);`
	`  mkdir(dataFolderAnnotationsTrain);`
	`end`
	
	Download the Camelyon16 data set from Camelyon17 by clicking the first "CAMELYON16 data set" link. Open the "training" directory, then follow these steps:
	
	• Download the "lesion_annotations.zip" file. Extract the files to the directory specified by the `dataFolderAnnotationsTrain` variable.
	• Open the "normal" directory. Download the images to the directory specified by the `dataFolderNormalTrain` variable.
	• Open the "tumor" directory. Download the images to the directory specified by the `dataFolderTumorTrain` variable.
	
	For an example showing how to process this data for deep learning, see “Preprocess Multiresolution Images for Training Classification Network” (Image Processing Toolbox).
		Image classification (large images)



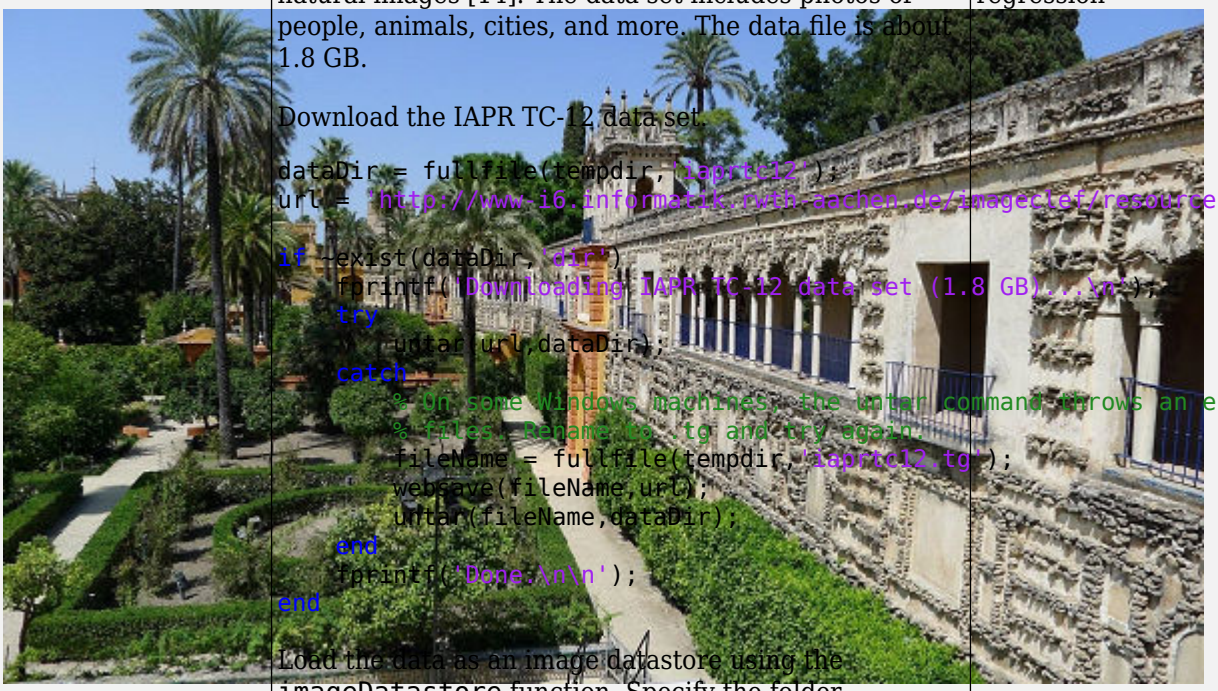
Data Set	Description	Task

Low Dose CT Grand Challenge	The Low Dose CT Grand Challenge includes pairs of regular-dose CT images and simulated low-dose CT images for 99 head scans (labeled N for neuro), 100 chest scans (labeled C for chest), and 100 abdomen scans (labeled L for liver) [12] [13]. The full data set is about 1.2 TB.
	
	Create a directory to store the chest files from the Low Dose CT Grand Challenge data set.
	
	`dataDir = fullfile(tempdir,"LDCT","LDCT-and-Projection-data");`
	`if ~exist(dataDir,'dir')`
	`    mkdir(dataDir);`
	`end`
	
	To download the data, go to The Cancer Imaging Archive website. Download the chest files from the "Images (DICOM, 952 GB)" data set using the NBIA Data Retriever. Specify the `dataDir` variable as the location of the downloaded data. When the data is downloaded successfully, `dataDir` contains 50 subfolders with names such as "C002" and "C004", ending with "C296".
	
	For an example showing how to process this data for deep learning, see “Unsupervised Medical Image Denoising Using CycleGAN” on page 9-130.
		Image-to-image regression

Data Set	Description	Task

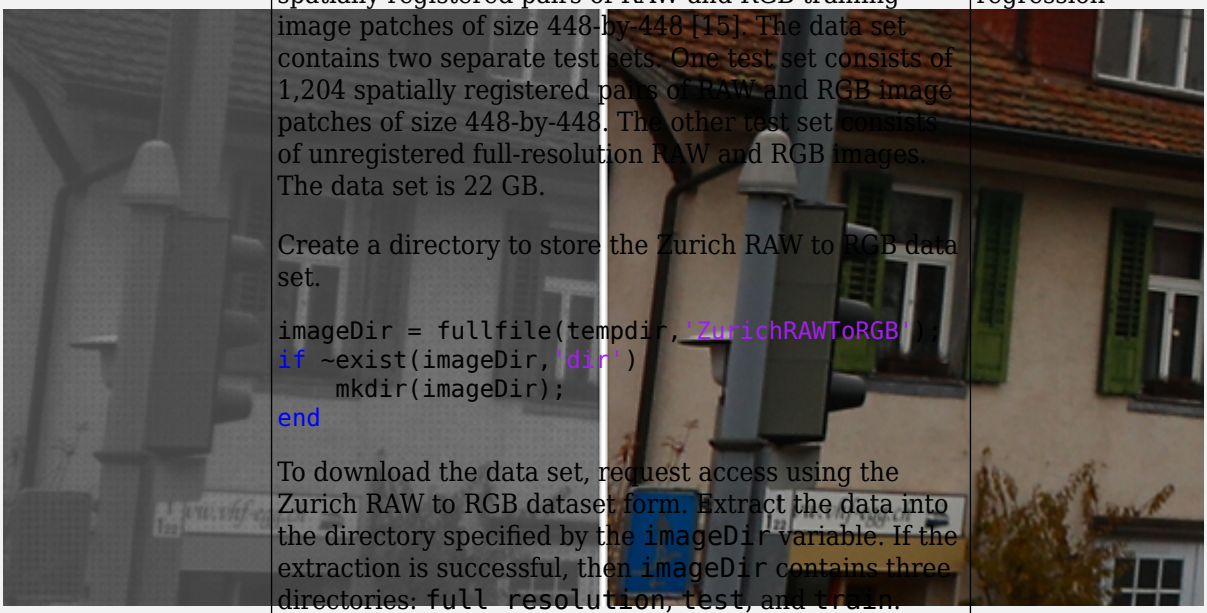
Common Objects in Context (COCO)
 
 
(Representative example)
	The COCO 2014 train images data set consists of 82,783 images. The annotations data contains at least five captions corresponding to each image.
	
	Create directories to store the COCO data set.
	
	`dataFolder = fullfile(tempdir,"coco");`
	`if ~exist(dataFolder,'dir')`
	`  mkdir(dataFolder);`
	`end`
	
	Download and extract the COCO 2014 train images and captions from https://cocodataset.org/#download by clicking the "2014 Train images" and "2014 Train/Val annotations" links, respectively. Save the data in the folder specified by `dataFolder`.
	
	Extract the captions from the file `captions_train2014.json` using the `jsondecode` function.
	
	`filename = fullfile(dataFolder,"annotations_trainval2014","annotations", ...`
	`  "captions_train2014.json");`
	`str = fileread(filename);`
	`data = jsondecode(str);`
	
	The `annotations` field of the structure contains the data required for image captioning.
	
	For an example showing how to process this data for deep learning, see “Image Captioning Using Attention” on page 4-198.
		Image captioning

Data Set	Description	Task

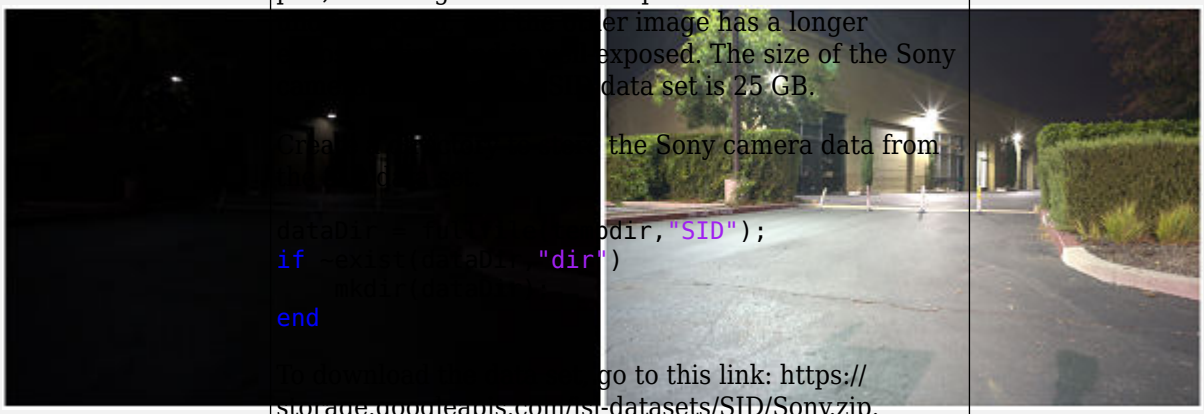
IAPR TC-12
 
(Representative example)
	The IAPR TC-12 Benchmark consists of 20,000 still natural images [14]. The data set includes photos of people, animals, cities, and more. The data file is about 1.8 GB.
	
	Download the IAPR TC-12 data set.
	
	`dataDir = fullfile(tempdir,'iaprtc12');`
	`url = 'http://www-i6.informatik.rwth-aachen.de/imageclef/resources/iaprtc12.t`
	
	`if ~exist(dataDir,'dir')`
	`    fprintf('Downloading IAPR TC-12 data set (1.8 GB)...\n');`
	`    try`
	`        untar(url,dataDir);`
	`    catch`
	`        % On some Windows machines, the untar command throws an error for .tg`
	`        % files. Rename to .tg and try again.`
	`        fileName = fullfile(tempdir,'iaprtc12.tg');`
	`        websave(fileName,url);`
	`        untar(fileName,dataDir);`
	`    end`
	`    fprintf('Done.\n\n');`
	`end`
	
	Load the data as an image datastore using the `imageDatastore` function. Specify the folder containing the image data and the image file extensions.
	
	`imageDir = fullfile(dataDir,'images')`
	`exts = {'.jpg','.bmp','.png'};`
	`imds = imageDatastore(imageDir, ...`
	`    'IncludeSubfolders',true,'FileExtensions',exts);`
	
	For an example showing how to process this data for deep learning, see “Single Image Super-Resolution Using Deep Learning” on page 9-8.
		Image-to-image regression

Data Set	Description	Task

Zurich RAW to RGB	The Zurich RAW to RGB data set contains 48,043 spatially registered pairs of RAW and RGB training image patches of size 448-by-448 [15]. The data set contains two separate test sets. One test set consists of 1,204 spatially registered pairs of RAW and RGB image patches of size 448-by-448. The other test set consists of unregistered full-resolution RAW and RGB images. The data set is 22 GB.
	
	Create a directory to store the Zurich RAW to RGB data set.
	
	`imageDir = fullfile(tempdir,'ZurichRAWToRGB');`
	`if ~exist(imageDir,'dir')`
	`    mkdir(imageDir);`
	`end`
	
	To download the data set, request access using the Zurich RAW to RGB dataset form. Extract the data into the directory specified by the `imageDir` variable. If the extraction is successful, then `imageDir` contains three directories: `full_resolution`, `test`, and `train`.
	
	For an example showing how to process this data for deep learning, see “Develop RAW Camera Processing Pipeline Using Deep Learning” on page 9-51.
		Image-to-image regression

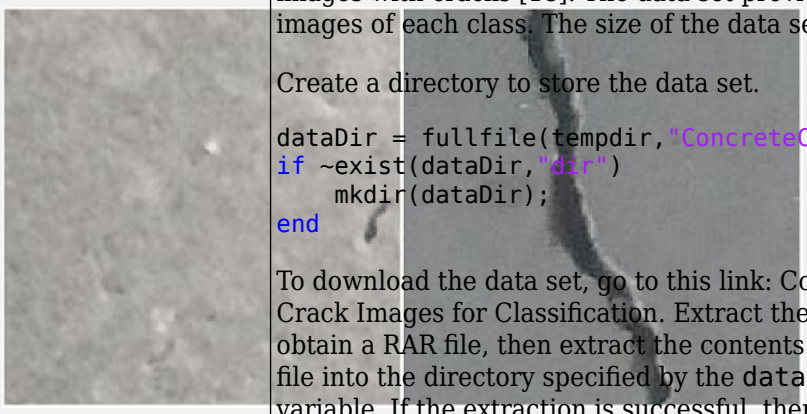


Data Set	Description	Task

See-In-The-Dark (SID)	The See-In-The-Dark (SID) data set provides registered pairs of RAW images of the same scene [16]. In each pair, one image has a short exposure time and is
	 
	er image has a longer
	exposed. The size of the Sony
	data set is 25 GB.
	
	the Sony camera data from
	
	`dataDir = fullfile(tempdir,"SID");`
	`if ~exist(dataDir,"dir")`
	`  mkdir(dataDir)`
	`end`
	
	go to this link: https://storage.googleapis.com/isl-datasets/SID/Sony.zip. Extract the data into the directory specified by the `dataDir` variable. When extracted successfully, `dataDir` contains the directory `Sony` with two subdirectories: `long` and `short`. The files in the `long` subdirectory have a long exposure and are well-exposed. The files in the `short` subdirectory have a short exposure and are quite underexposed and dark.
	
	The data set also provides text files that describe how to partition the files into training, validation, and test data sets. Move the files "Sony_train_list.txt", "Sony_val_list.txt", and "Sony_test_list.txt" to the directory specified by the `dataDir` variable.
	
	For an example showing how to process this data for deep learning, see “Recover Images from Extreme Low-Light Conditions Using Deep Learning” on page 9-73.
		Image-to-image regression

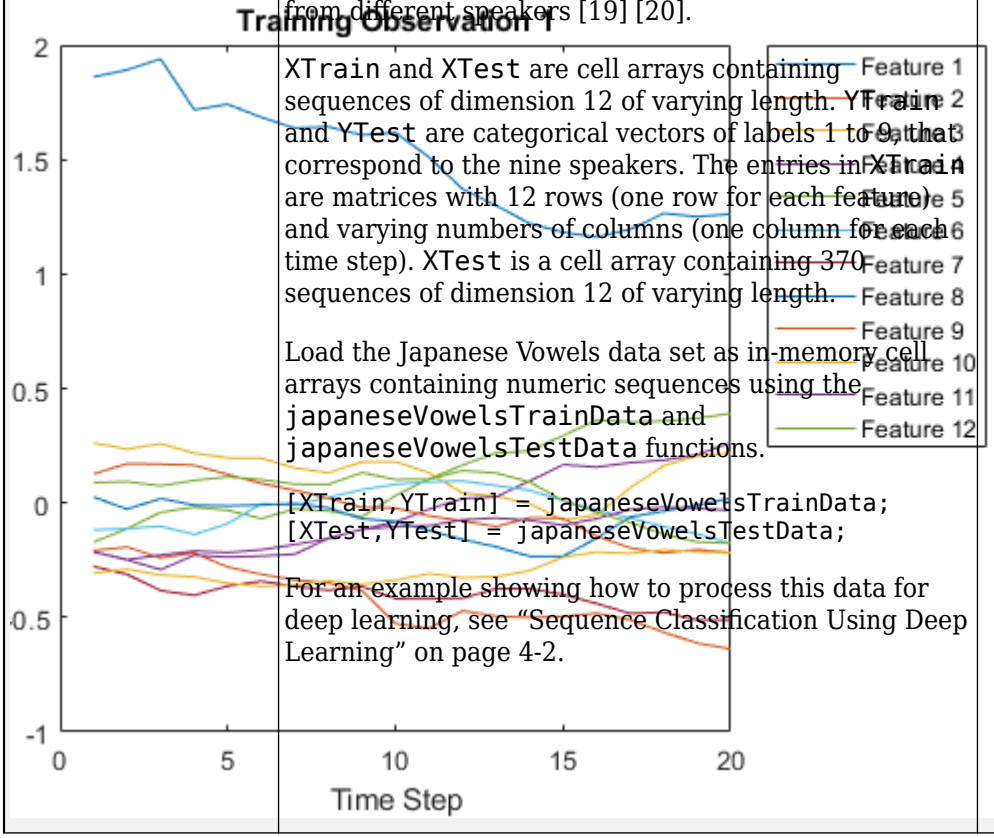
Data Set	Description	Task

LIVE In the Wild
 
	The LIVE In the Wild data set consists of 1,162 photos captured by mobile devices, with seven additional training images [17]. Each image is rated by an average of 175 individuals on a scale of [1, 100]. The data set provides the mean and standard deviation of the subjective scores for each image.
	
	Create a directory to store the LIVE In the Wild data set.
	
	`imageDir = fullfile(tempdir,"LIVEInTheWild");`
	`if ~exist(imageDir,'dir')`
	`    mkdir(imageDir);`
	`end`
	
	Download the data set by following the instructions outlined in LIVE In the Wild Image Quality Challenge Database. Extract the data into the directory specified by the `imageDir` variable. When extracted successfully, `imageDir` contains two directories: `Data` and `Images`.
	
	For an example showing how to process this data for deep learning, see “Quantify Image Quality Using Neural Image Assessment” on page 9-108.
		Image classification

Concrete Crack Images for Classification
 
	The Concrete Crack Images for Classification data set contains images of two classes: "Negative" images without cracks present in the road and "Positive" images with cracks [18]. The data set provides 20,000 images of each class. The size of the data set is 235 MB.
	
	Create a directory to store the data set.
	
	`dataDir = fullfile(tempdir,"ConcreteCracks");`
	`if ~exist(dataDir,"dir")`
	`    mkdir(dataDir);`
	`end`
	
	To download the data set, go to this link: Concrete Crack Images for Classification. Extract the ZIP file to obtain a RAR file, then extract the contents of the RAR file into the directory specified by the `dataDir` variable. If the extraction is successful, then `dataDir` contains two subdirectories: `Negative` and `Positive`.
	
	For an example showing how to process this data for deep learning, see “Detect Image Anomalies Using Explainable One-Class Classification Neural Network” on page 9-157.
		Image classification

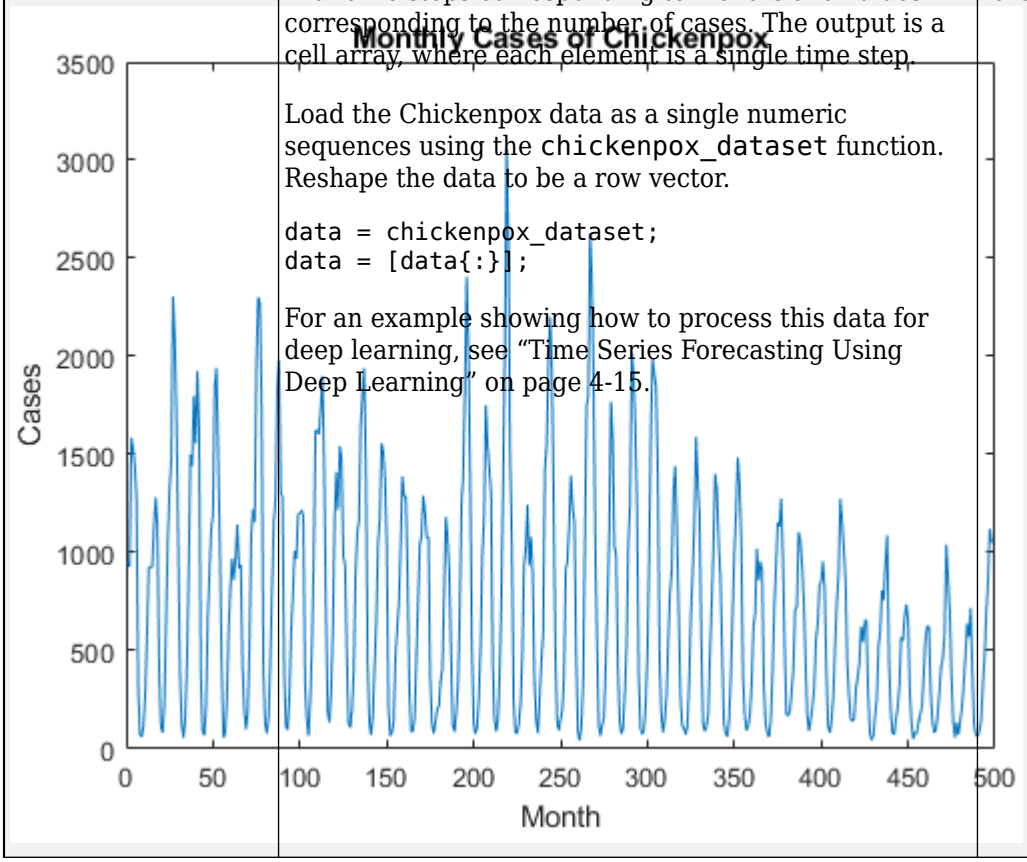
Time Series and Signal Data Sets

Data	Description	Task

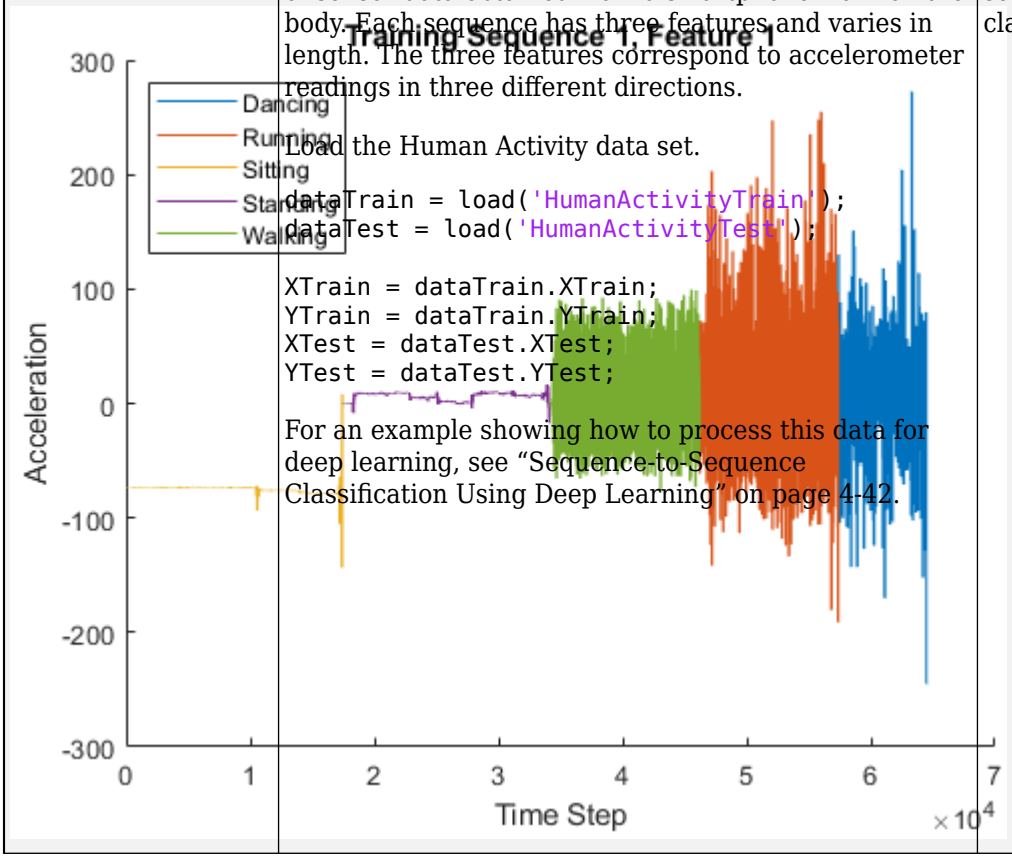
Japanese Vowels	The Japanese Vowels data set contains preprocessed sequences representing utterances of Japanese vowels from different speakers [19] [20].
	 
	`XTrain` and `XTest` are cell arrays containing sequences of dimension 12 of varying length. `YTrain` and `YTest` are categorical vectors of labels 1 to 9, that correspond to the nine speakers. The entries in `XTrain` are matrices with 12 rows (one row for each feature) and varying numbers of columns (one column for each time step). `XTest` is a cell array containing 370 sequences of dimension 12 of varying length.
	
	Load the Japanese Vowels data set as in-memory cell arrays containing numeric sequences using the `japaneseVowelsTrainData` and `japaneseVowelsTestData` functions.
	
	`[XTrain,YTrain] = japaneseVowelsTrainData;`
	`[XTest,YTest] = japaneseVowelsTestData;`
	
	For an example showing how to process this data for deep learning, see “Sequence Classification Using Deep Learning” on page 4-2.
		Sequence-to-label classification

Data	Description	Task

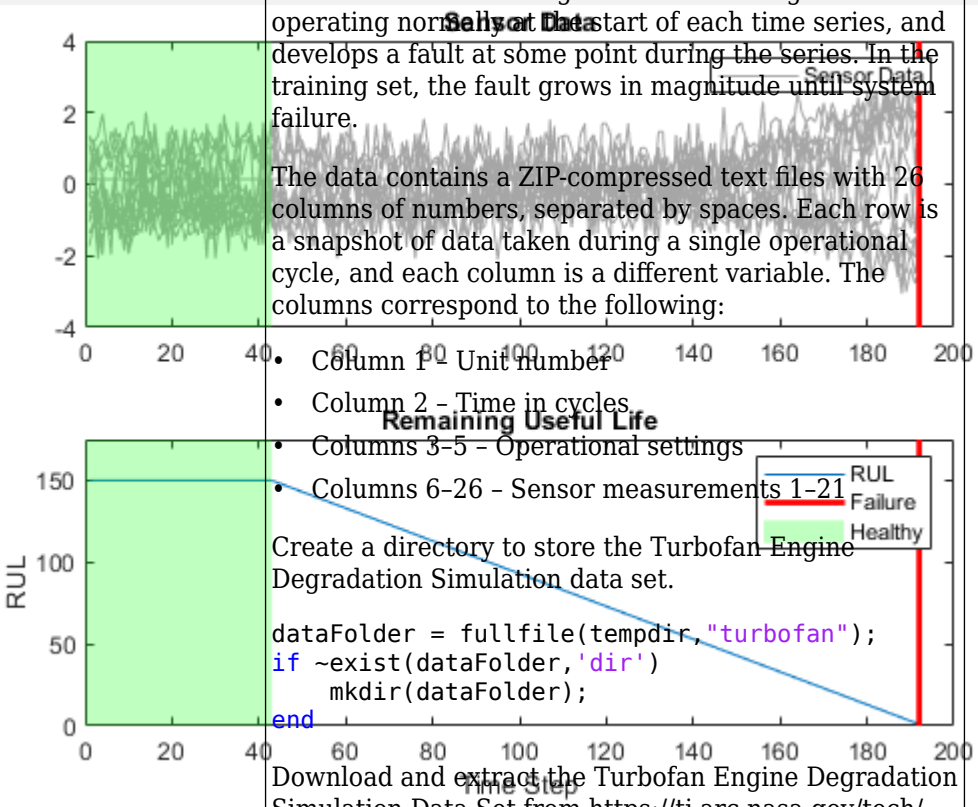
Chickenpox	The Chickenpox data set contains a single time series, with time steps corresponding to months and values corresponding to the number of cases. The output is a cell array, where each element is a single time step.
	
	Load the Chickenpox data as a single numeric sequences using the `chickenpox_dataset` function. Reshape the data to be a row vector.
	
	`data = chickenpox_dataset;`
	`data = [data{:}];`
	
	For an example showing how to process this data for deep learning, see “Time Series Forecasting Using Deep Learning” on page 4-15.
		Time-series forecasting



Data	Description	Task

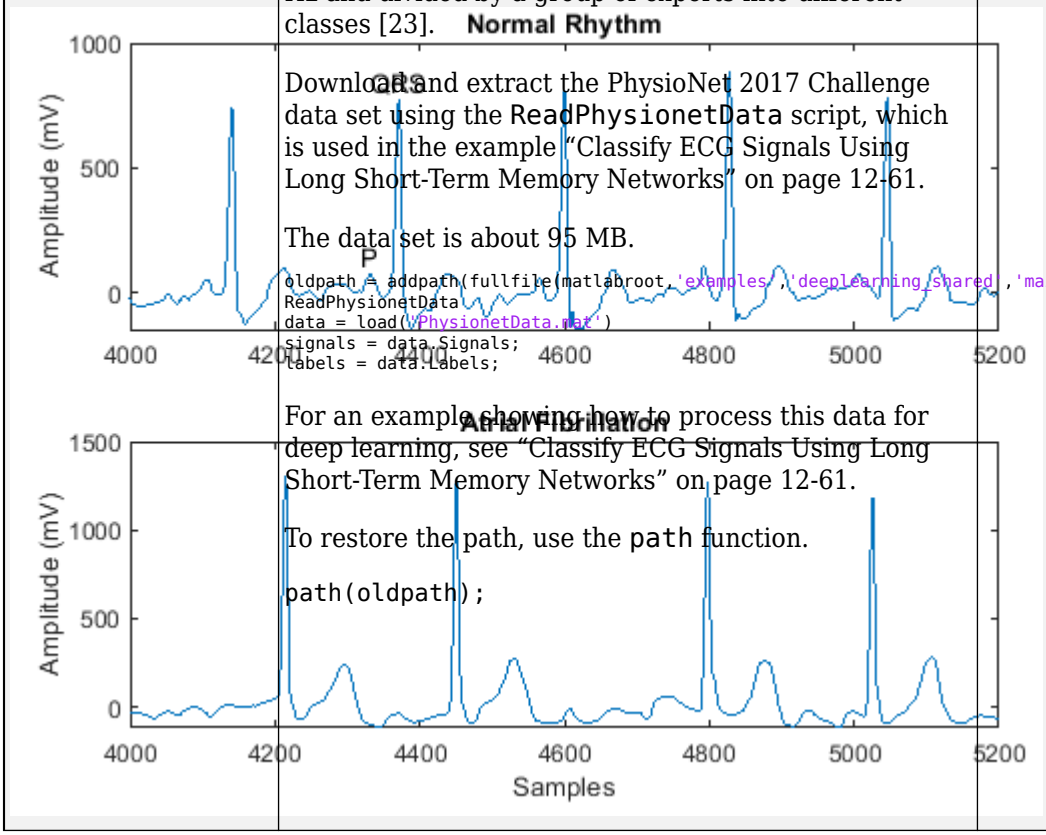
Human Activity 	The Human Activity data set contains seven time series of sensor data obtained from a smartphone worn on the body. Each sequence has three features and varies in length. The three features correspond to accelerometer readings in three different directions.
	
	Load the Human Activity data set.
	
	`dataTrain = load('HumanActivityTrain');`
	`dataTest = load('HumanActivityTest');`
	
	`XTrain = dataTrain.XTrain;`
	`YTrain = dataTrain.YTrain;`
	`XTest = dataTest.XTest;`
	`YTest = dataTest.YTest;`
	
	For an example showing how to process this data for deep learning, see “Sequence-to-Sequence Classification Using Deep Learning” on page 4-42.
		Sequence-to-sequence classification

Data	Description	Task

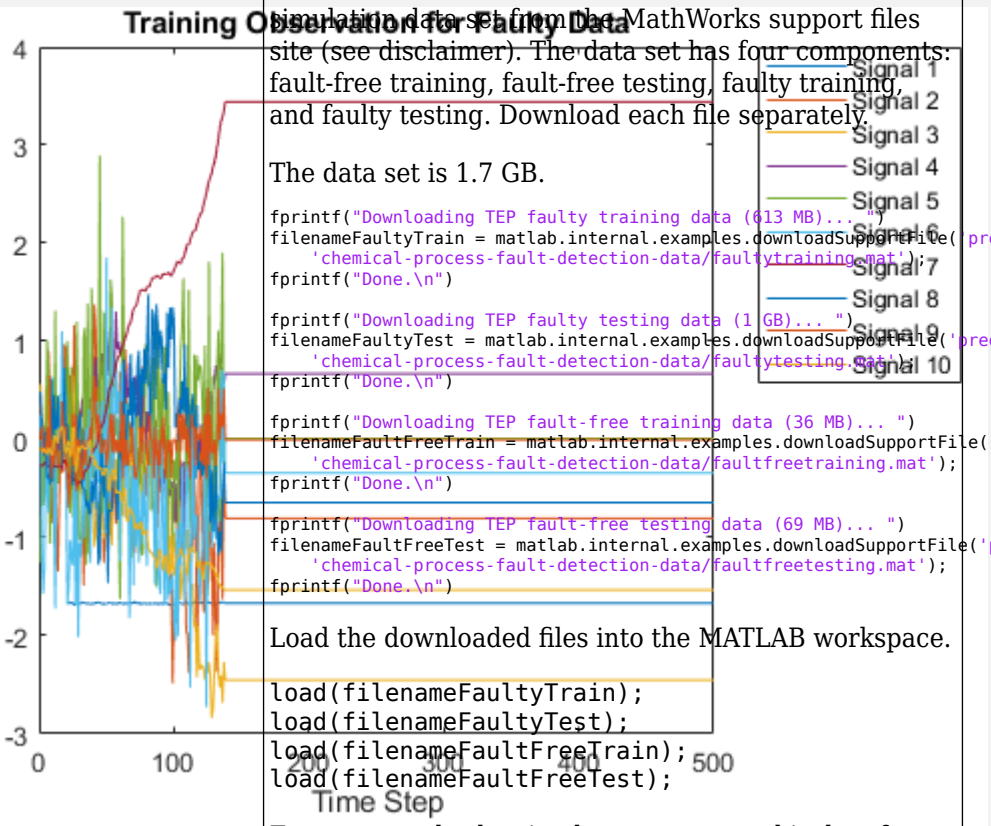
Turbofan Engine Degradation Simulation	Each time series of the Turbofan Engine Degradation Simulation data set represents a different engine [21]. Each engine starts with unknown degrees of initial wear and manufacturing variation. The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure.
	
	 
	
	The data contains a ZIP-compressed text files with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, and each column is a different variable. The columns correspond to the following:
	
	• Column 1 – Unit number
	• Column 2 – Time in cycles
	• Columns 3–5 – Operational settings
	• Columns 6–26 – Sensor measurements 1–21
	
	Create a directory to store the Turbofan Engine Degradation Simulation data set.
	
	`dataFolder = fullfile(tempdir,"turbofan");`
	`if ~exist(dataFolder,'dir')`
	`  mkdir(dataFolder);`
	`end`
	
	Download and extract the Turbofan Engine Degradation Simulation Data Set from https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/.
	
	Unzip the data from the file `CMAPSSData.zip`.
	
	`filename = "CMAPSSData.zip";`
	`unzip(filename,dataFolder)`
	
	Load the training and test data using the helper functions `processTurboFanDataTrain` and `processTurboFanDataTest`, respectively. These functions are used in the example “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','nnet','main'));`
	`filenamePredictors = fullfile(dataFolder,"train_FD001.txt");`
	`[XTrain,YTrain] = processTurboFanDataTrain(filenamePredictors);`
	
	`filenamePredictors = fullfile(dataFolder,"test_FD001.txt");`
	`filenameResponses = fullfile(dataFolder,"RUL_FD001.txt");`
	`[XTest,YTest] = processTurboFanDataTest(filenamePredictors,filenameResponses);`
		Sequence-to-sequence regression, predictive maintenance

Data	Description	Task
	For an example showing how to process this data for deep learning, see “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47. To restore the path, use the `path` function. `path(oldpath);`	

PhysioNet 2017 Challenge	The PhysioNet 2017 Challenge data set consists of a set of electrocardiogram (ECG) recordings sampled at 300 Hz and divided by a group of experts into different classes [23].
	
	Download and extract the PhysioNet 2017 Challenge data set using the `ReadPhysionetData` script, which is used in the example “Classify ECG Signals Using Long Short-Term Memory Networks” on page 12-61.
	
	The data set is about 95 MB.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','deeplearning_shared','main'));`
	`ReadPhysionetData`
	`data = load('PhysionetData.mat')`
	`signals = data.Signals;`
	`labels = data.Labels;`
	
	For an example showing how to process this data for deep learning, see “Classify ECG Signals Using Long Short-Term Memory Networks” on page 12-61.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Sequence-to-label classification

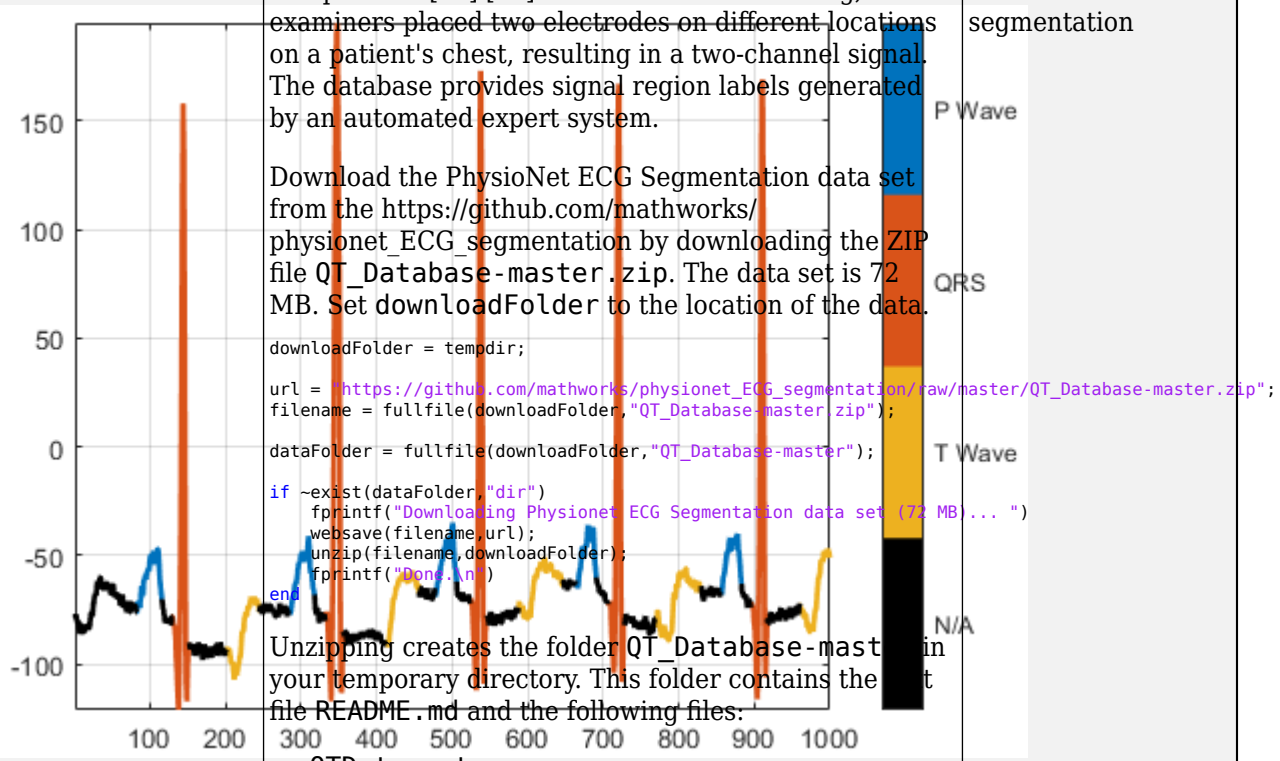


Data	Description	Task

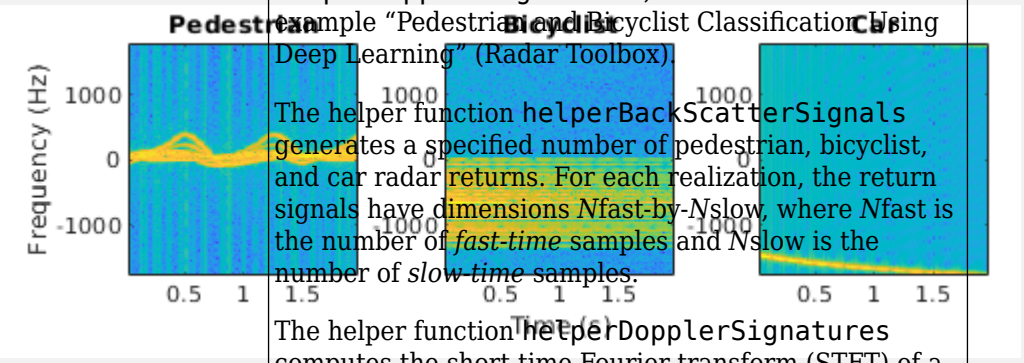
Tennessee Eastman Process (TEP) simulation	This data set consists of MAT files converted from the Tennessee Eastman Process (TEP) simulation data [22].
	
	Download the Tennessee Eastman Process (TEP) simulation data set from the MathWorks support files site (see disclaimer). The data set has four components: fault-free training, fault-free testing, faulty training, and faulty testing. Download each file separately.
	
	The data set is 1.7 GB.
	
	 
	`fprintf("Downloading TEP faulty training data (613 MB)... ")`
	`filenameFaultyTrain = matlab.internal.examples.downloadSupportFile('predmaint', ...`
	`'chemical-process-fault-detection-data/faultytraining.mat');`
	`fprintf("Done.\n")`
	
	`fprintf("Downloading TEP faulty testing data (1 GB)... ")`
	`filenameFaultyTest = matlab.internal.examples.downloadSupportFile('predmaint', ...`
	`'chemical-process-fault-detection-data/faultytesting.mat');`
	`fprintf("Done.\n")`
	
	`fprintf("Downloading TEP fault-free training data (36 MB)... ")`
	`filenameFaultFreeTrain = matlab.internal.examples.downloadSupportFile('predmaint', ...`
	`'chemical-process-fault-detection-data/faultfreetraining.mat');`
	`fprintf("Done.\n")`
	
	`fprintf("Downloading TEP fault-free testing data (69 MB)... ")`
	`filenameFaultFreeTest = matlab.internal.examples.downloadSupportFile('predmaint', ...`
	`'chemical-process-fault-detection-data/faultfreetesting.mat');`
	`fprintf("Done.\n")`
	
	Load the downloaded files into the MATLAB workspace.
	
	`load(filenameFaultyTrain);`
	`load(filenameFaultyTest);`
	`load(filenameFaultFreeTrain);`
	`load(filenameFaultFreeTest);`
	
	For an example showing how to process this data for deep learning, see “Chemical Process Fault Detection Using Deep Learning” on page 16-2.
		Sequence-to-label classification

Data	Description	Task

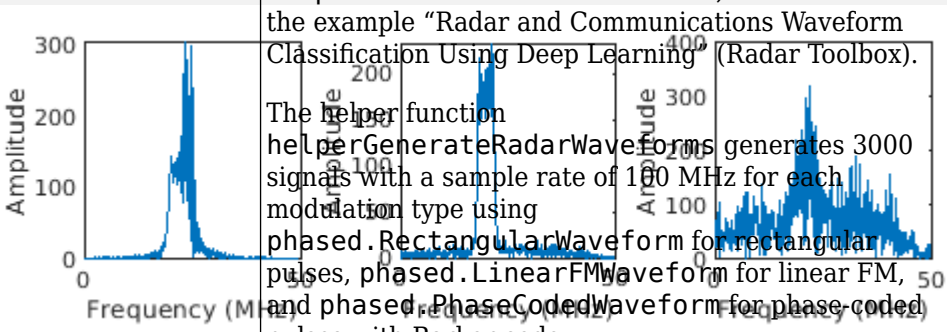
PhysioNet ECG Segmentation	The PhysioNet ECG Segmentation data set consists of roughly 15 minutes of ECG recordings from a total of 105 patients [23] [24]. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, resulting in a two-channel signal. The database provides signal region labels generated by an automated expert system.
	
	Download the PhysioNet ECG Segmentation data set from the https://github.com/mathworks/physionet_ECG_segmentation by downloading the ZIP file `QT_Database-master.zip`. The data set is 72 MB. Set `downloadFolder` to the location of the data.
	
	`downloadFolder = tempdir;`
	
	`url = "https://github.com/mathworks/physionet_ECG_segmentation/raw/master/QT_Database-master.zip";`
	`filename = fullfile(downloadFolder,"QT_Database-master.zip");`
	
	`dataFolder = fullfile(downloadFolder,"QT_Database-master");`
	
	`if ~exist(dataFolder,"dir")`
	`  fprintf("Downloading Physionet ECG Segmentation data set (72 MB)... ")`
	`  websave(filename,url);`
	`  unzip(filename,downloadFolder);`
	`  fprintf("Done.\n")`
	`end`
	
	Unzipping creates the folder `QT_Database-master` in your temporary directory. This folder contains the text file `README.md` and the following files:
	
	• `QTData.mat`
	• `Modified_physionet_data.txt`
	• `License.txt`
	
	`QTData.mat` contains the PhysioNet ECG Segmentation data. The file `Modified_physionet_data.txt` provides the source attributions for the data and a description of the operations applied to each raw ECG recording. Load the PhysioNet ECG Segmentation data from the MAT file.
	
	`load(fullfile(dataFolder,'QTData.mat'))`
	
	For an example showing how to process this data for deep learning, see “Waveform Segmentation Using Deep Learning” on page 12-41.
		Sequence-to-label classification, waveform segmentation



Data	Description	Task

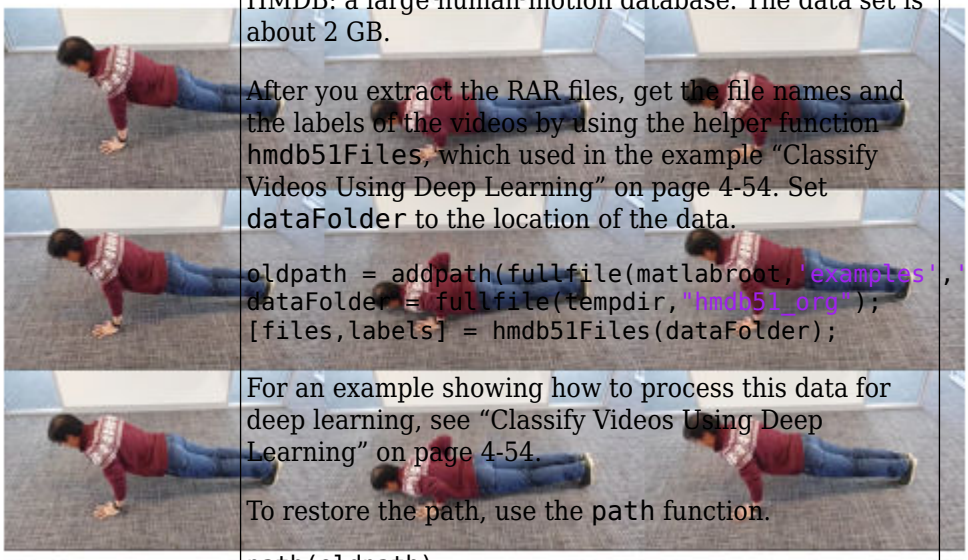
Synthetic pedestrian, car, and bicyclist backscattering	Generate a synthetic pedestrian, car, and bicyclist backscattering data set using the helper functions `helperBackScatterSignals` and `helperDopplerSignatures`, which are used in the example "Pedestrian and Bicyclist Classification Using Deep Learning" (Radar Toolbox).
	
	 
	The helper function `helperBackScatterSignals` generates a specified number of pedestrian, bicyclist, and car radar returns. For each realization, the return signals have dimensions *N*fast-by-*N*slow, where *N*fast is the number of *fast-time* samples and *N*slow is the number of *slow-time* samples.
	
	The helper function `helperDopplerSignatures` computes the short-time Fourier transform (STFT) of a radar return to generate the micro-Doppler signature. To obtain the micro-Doppler signatures, use the helper functions to apply the STFT and a preprocessing method to each signal.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','phased','main'));`
	`numPed = 1; % Number of pedestrian realizations`
	`numBic = 1; % Number of bicyclist realizations`
	`numCar = 1; % Number of car realizations`
	`[xPedRec,xBicRec,xCarRec,Tsamp] = helperBackScatterSignals(numPed,numBic,numCar);`
	
	`[SPed,T,F] = helperDopplerSignatures(xPedRec,Tsamp);`
	`[SBic,~,~] = helperDopplerSignatures(xBicRec,Tsamp);`
	`[SCar,~,~] = helperDopplerSignatures(xCarRec,Tsamp);`
	
	For an example showing how to process this data for deep learning, see "Pedestrian and Bicyclist Classification Using Deep Learning" (Radar Toolbox).
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Sequence-to-label classification

Data	Description	Task

Generated waveforms 	Generate rectangular, linear FM, and phase coded waveforms using the helper function `helperGenerateRadarWaveforms`, which is used in the example “Radar and Communications Waveform Classification Using Deep Learning” (Radar Toolbox).
	
	The helper function `helperGenerateRadarWaveforms` generates 3000 signals with a sample rate of 100 MHz for each modulation type using `phased.RectangularWaveform` for rectangular pulses, `phased.LinearFMWaveform` for linear FM, and `phased.PhaseCodedWaveform` for phase-coded pulses with Barker code.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','phased','main'));`
	`[wav, modType] = helperGenerateRadarWaveforms;`
	
	For an example showing how to process this data for deep learning, see “Radar and Communications Waveform Classification Using Deep Learning” (Radar Toolbox).
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Sequence-to-label classification

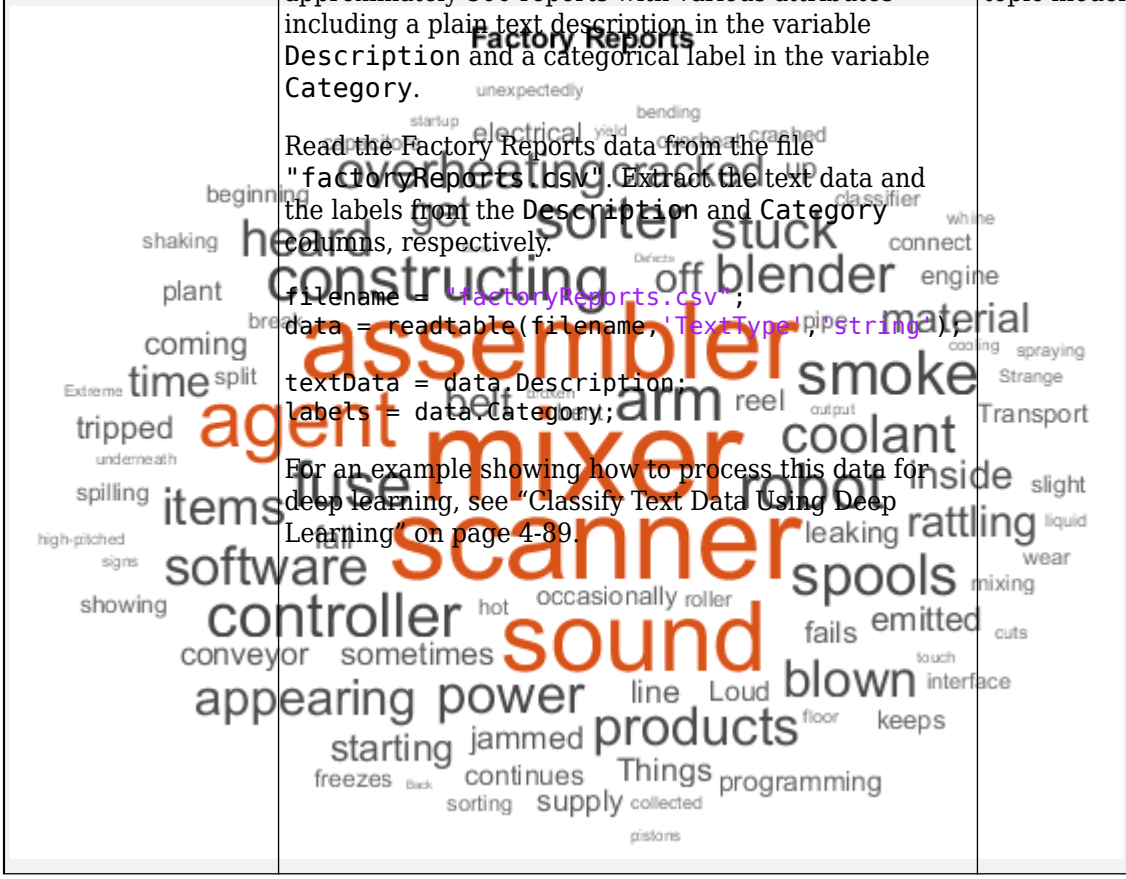
Video Data Sets

Data	Description	Task

HMDB: a large human motion database
 
(Representative example)
	The HMBD51 data set contains about 2 GB of video data for 7000 clips from 51 classes, such as *drink*, *run*, and *pushup*.
	
	Download and extract the HMBD51 data set from HMDB: a large human motion database. The data set is about 2 GB.
	
	After you extract the RAR files, get the file names and the labels of the videos by using the helper function `hmdb51Files`, which used in the example "Classify Videos Using Deep Learning" on page 4-54. Set `dataFolder` to the location of the data.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','nnet','main'));`
	`dataFolder = fullfile(tempdir,"hmdb51_org");`
	`[files,labels] = hmdb51Files(dataFolder);`
	
	For an example showing how to process this data for deep learning, see "Classify Videos Using Deep Learning" on page 4-54.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Video classification

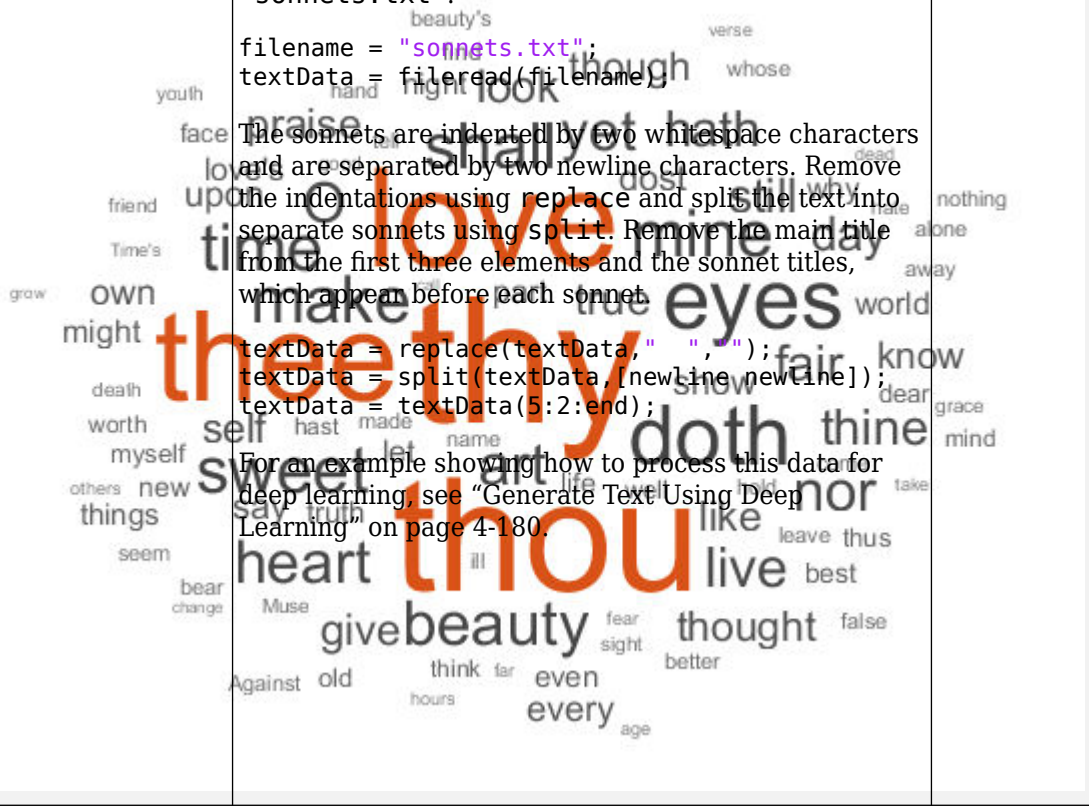
Text Data Sets

Data	Description	Task

Factory Reports	The Factory Reports data set is a table containing approximately 500 reports with various attributes including a plain text description in the variable `Description` and a categorical label in the variable `Category`.
	 
	Read the Factory Reports data from the file `"factoryReports.csv"`. Extract the text data and the labels from the `Description` and `Category` columns, respectively.
	`filename = "factoryReports.csv";`
	`data = readtable(filename,'TextType','string');`
	`textData = data.Description;`
	`labels = data.Category;`
	For an example showing how to process this data for deep learning, see “Classify Text Data Using Deep Learning” on page 4-89.
		Text classification, topic modeling

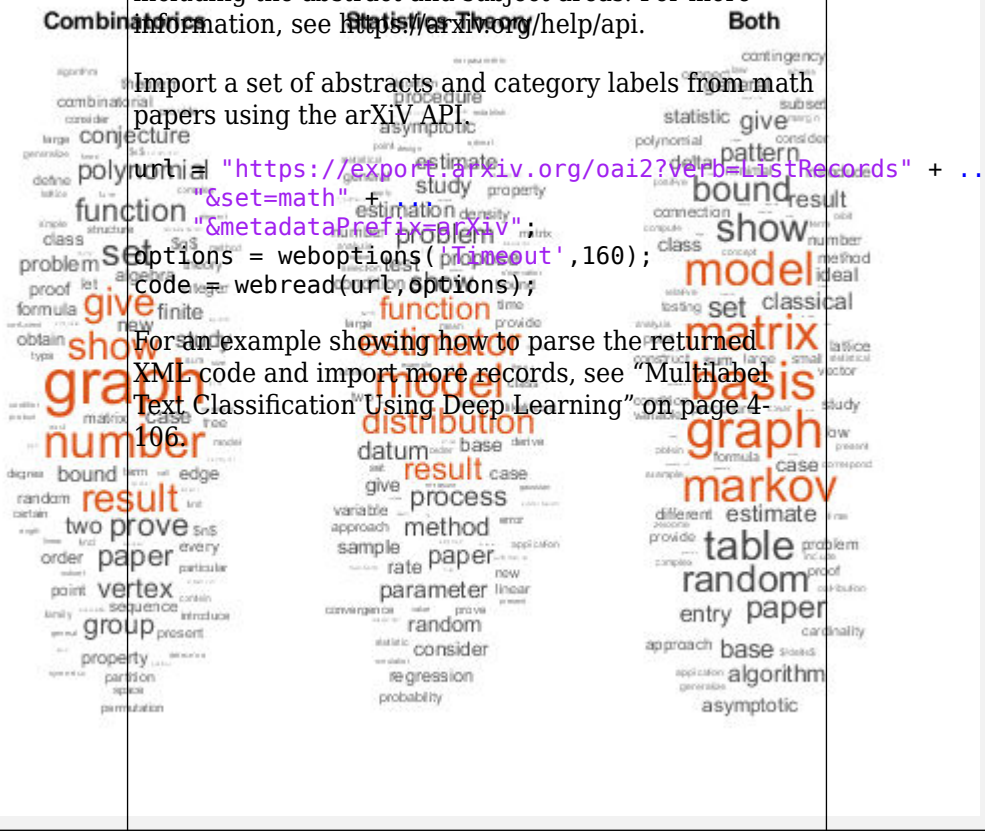
Data	Description	Task

Shakespeare's Sonnets	The file `sonnets.txt` contains all of Shakespeare's sonnets in a single text file.
	
	Read the Shakespeare's Sonnets data from the file `"sonnets.txt"`.
	
	`filename = "sonnets.txt";`
	`textData = fileread(filename);`
	
	The sonnets are indented by two whitespace characters and are separated by two newline characters. Remove the indentations using `replace` and split the text into separate sonnets using `split`. Remove the main title from the first three elements and the sonnet titles, which appear before each sonnet.
	
	`textData = replace(textData,"  ","");`
	`textData = split(textData,[newline newline]);`
	`textData = textData(5:2:end);`
	
	For an example showing how to process this data for deep learning, see "Generate Text Using Deep Learning" on page 4-180.
		Topic modeling, text generation



Data	Description	Task

ArXiv Metadata	The ArXiv API allows you to access the metadata of scientific e-prints submitted to https://arxiv.org including the abstract and subject areas. For more information, see https://arxiv.org/help/api.
	
	Import a set of abstracts and category labels from math papers using the arXiV API.
	
	`url = "https://export.arxiv.org/oai2?verb=ListRecords" + ... "&set=math" + ... "&metadataPrefix=arXiv";`
	`options = weboptions('Timeout',160);`
	`code = webread(url,options);`
	
	For an example showing how to parse the returned XML code and import more records, see “Multilabel Text Classification Using Deep Learning” on page 4-106.
		Text classification, topic modeling



Data	Description	Task

Books from Project Gutenberg	You can download many books from Project Gutenberg. For example, download the text from Alice's Adventures in Wonderland by Lewis Carroll from https://www.gutenberg.org/files/11/11-h/11-h.htm using the `webread` function.
	
	`url = "https://www.gutenberg.org/files/11/11-h/11-h.htm";`
	`code = webread(url);`
	
	The HTML code contains the relevant text inside `<p>` (paragraph) elements. Extract the relevant text by parsing the HTML code using the `htmlTree` function and then finding all the elements with the element name `"p"`.
	
	`tree = htmlTree(code);`
	`selector = "p";`
	`subtrees = findElement(tree,selector);`
	
	Extract the text data from the HTML subtrees using the `extractHTMLText` function and remove the empty elements.
	
	`textData = extractHTMLText(subtrees);`
	`textData(textData == "") = [];`
	
	For an example showing how to process this data for deep learning, see "Word-By-Word Text Generation Using Deep Learning" on page 4-192.
		Topic modeling, text generation

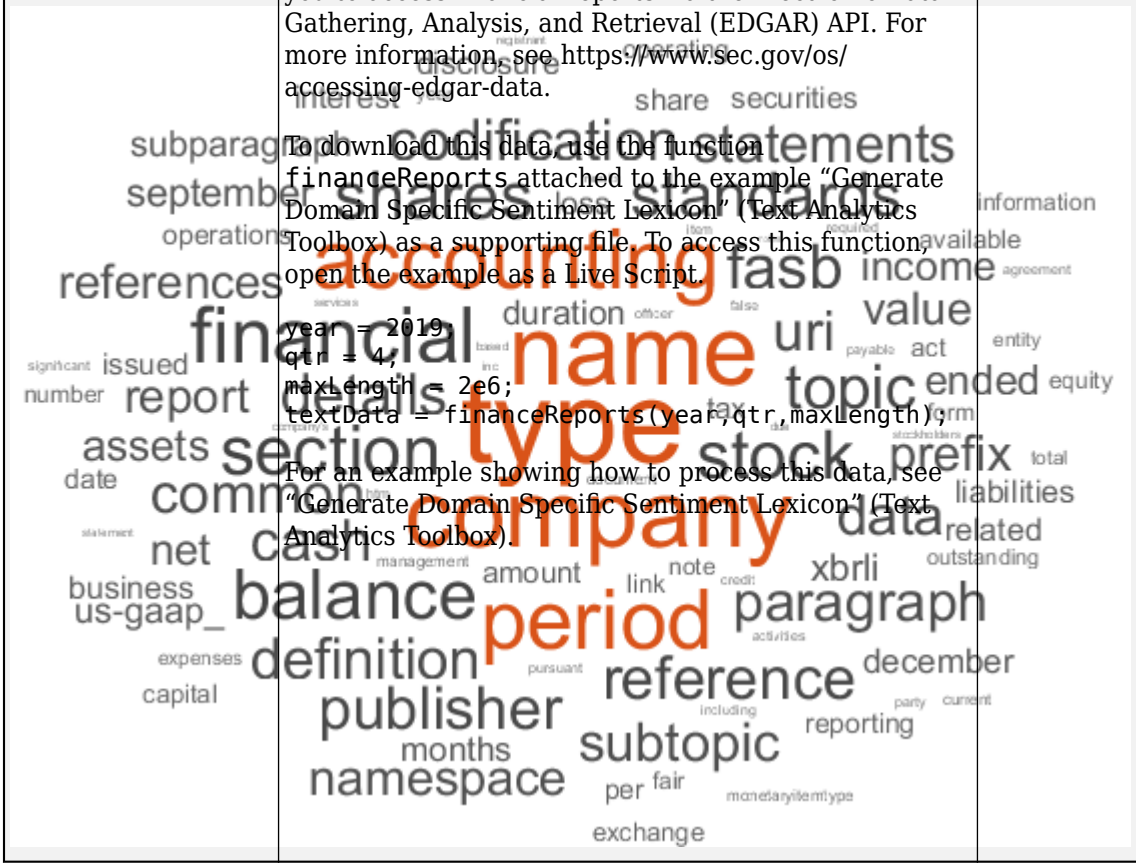
Data	Description	Task

Weekend updates	The file `weekendUpdates.xlsx` contains example social media status updates containing the hashtags "#weekend" and "#vacation". This data set requires Text Analytics Toolbox.
	
	Extract the text data from the file `weekendUpdates.xlsx` using the `readtable` function and extract the text data from the variable `TextData`.
	
	`filename = "weekendUpdates.xlsx";`
	`tbl = readtable(filename,'TextType','string');`
	`textData = tbl.TextData;`
	
	For an example showing how to process this data, see "Analyze Sentiment in Text" (Text Analytics Toolbox).
		Sentiment analysis

Roman Numerals	The CSV file `"romanNumerals.csv"` contains the decimal numbers 1–1000 in the first column and the corresponding Roman numerals in the second column.
	
	Load the decimal-Roman numeral pairs from the CSV file `"romanNumerals.csv"`.
	
	`filename = fullfile("romanNumerals.csv");`
	`options = detectImportOptions(filename, ...`
	`    'TextType','string', ...`
	`    'ReadVariableNames',false);`
	`options.VariableNames = ["Source" "Target"];`
	`options.VariableTypes = ["string" "string"];`
	
	`data = readtable(filename,options);`
	
	For an example showing how to process this data for deep learning, see "Sequence-to-Sequence Translation Using Attention" on page 4-164.
		Sequence-to-sequence translation

Data	Description	Task

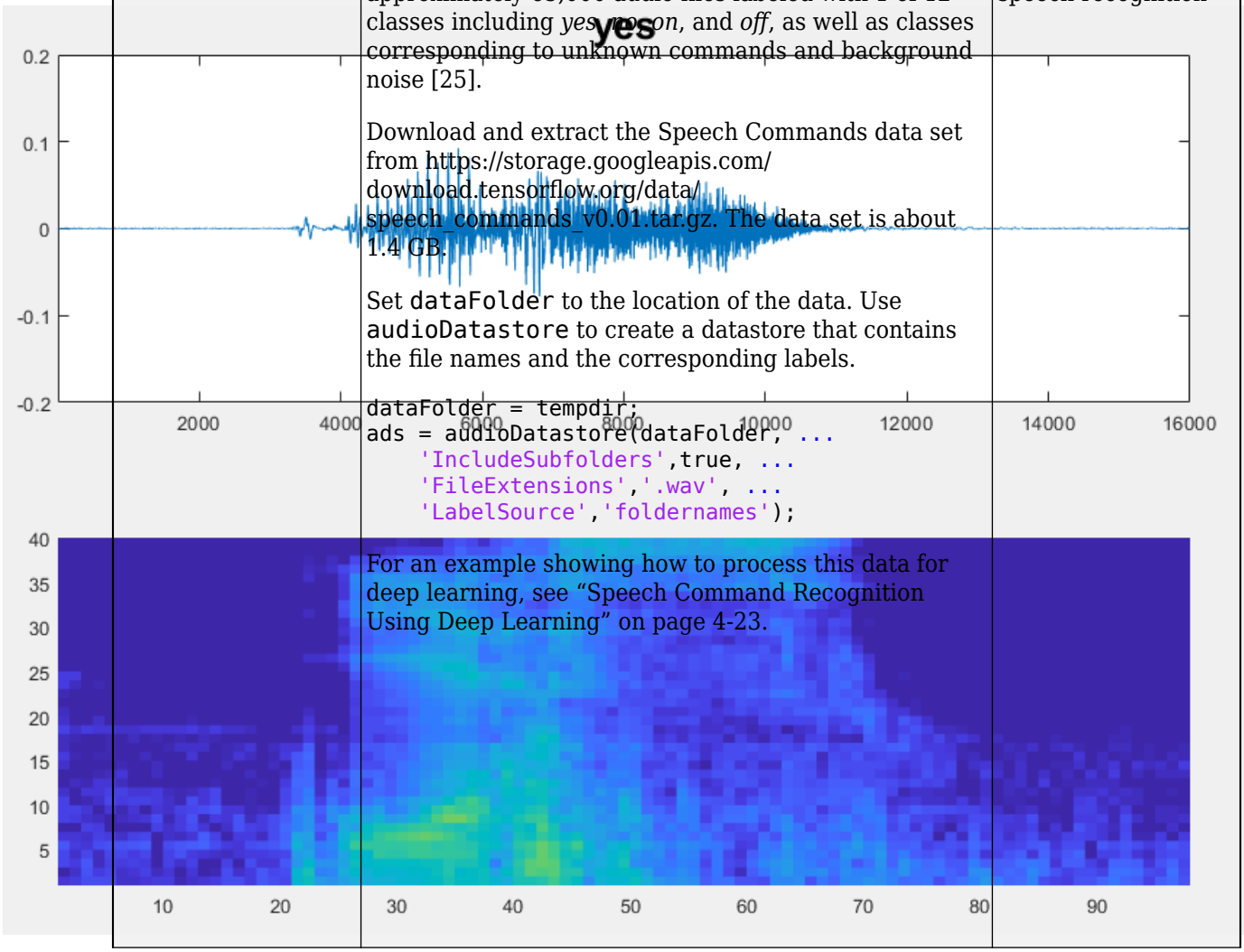
Finance Reports	The Securities and Exchange Commission (SEC) allows you to access financial reports via the Electronic Data Gathering, Analysis, and Retrieval (EDGAR) API. For more information, see https://www.sec.gov/os/accessing-edgar-data.
	
	To download this data, use the function `financeReports` attached to the example "Generate Domain Specific Sentiment Lexicon" (Text Analytics Toolbox) as a supporting file. To access this function, open the example as a Live Script.
	
	`year = 2019;`
	`qtr = 4;`
	`maxLength = 2e6;`
	`textData = financeReports(year,qtr,maxLength);`
	
	For an example showing how to process this data, see "Generate Domain Specific Sentiment Lexicon" (Text Analytics Toolbox).
		Sentiment analysis



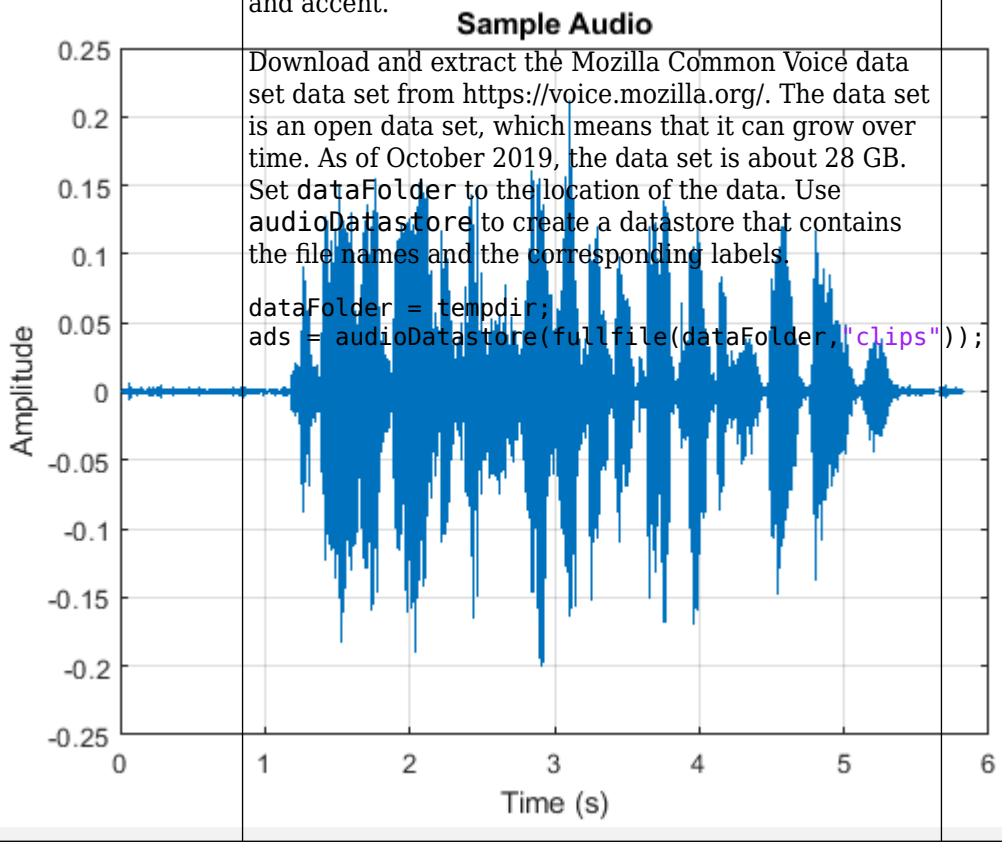
Audio Data Sets

Data	Description	Task

Speech Commands	The Speech Commands data set consists of approximately 65,000 audio files labeled with 1 of 12 classes including *yes*, *no*, *on*, and *off*, as well as classes corresponding to unknown commands and background noise [25].
	
	Download and extract the Speech Commands data set from https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz. The data set is about 1.4 GB.
	
	Set `dataFolder` to the location of the data. Use `audioDatastore` to create a datastore that contains the file names and the corresponding labels.
	
	`dataFolder = tempdir;`
	`ads = audioDatastore(dataFolder, ...`
	`    'IncludeSubfolders',true, ...`
	`    'FileExtensions','.wav', ...`
	`    'LabelSource','foldernames');`
	
	For an example showing how to process this data for deep learning, see “Speech Command Recognition Using Deep Learning” on page 4-23.
		Audio classification, speech recognition

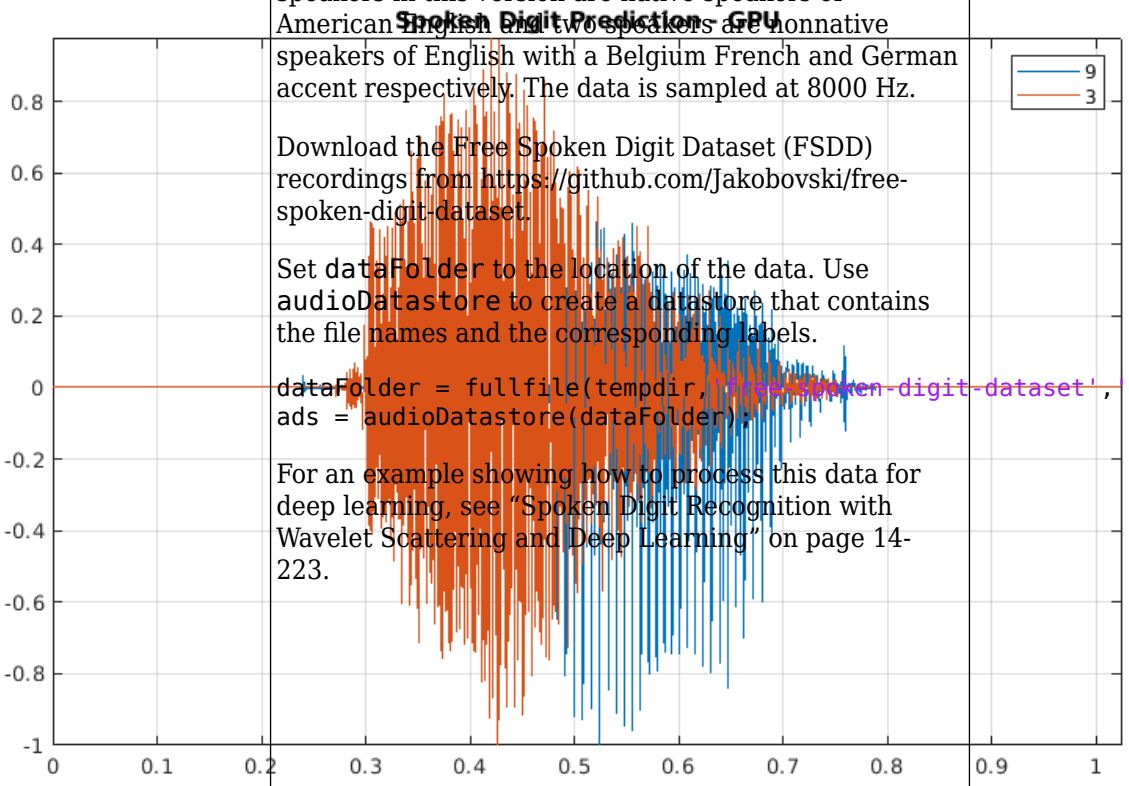


Data	Description	Task

Mozilla Common Voice	The Mozilla Common Voice data set consists of audio recordings of speech and corresponding text files. The data also includes demographic metadata such as age and accent.
	 
	Download and extract the Mozilla Common Voice data set data set from https://voice.mozilla.org/. The data set is an open data set, which means that it can grow over time. As of October 2019, the data set is about 28 GB. Set `dataFolder` to the location of the data. Use `audioDatastore` to create a datastore that contains the file names and the corresponding labels.
	`dataFolder = tempdir;`
	`ads = audioDatastore(fullfile(dataFolder,"clips"));`
		Audio classification, speech recognition.

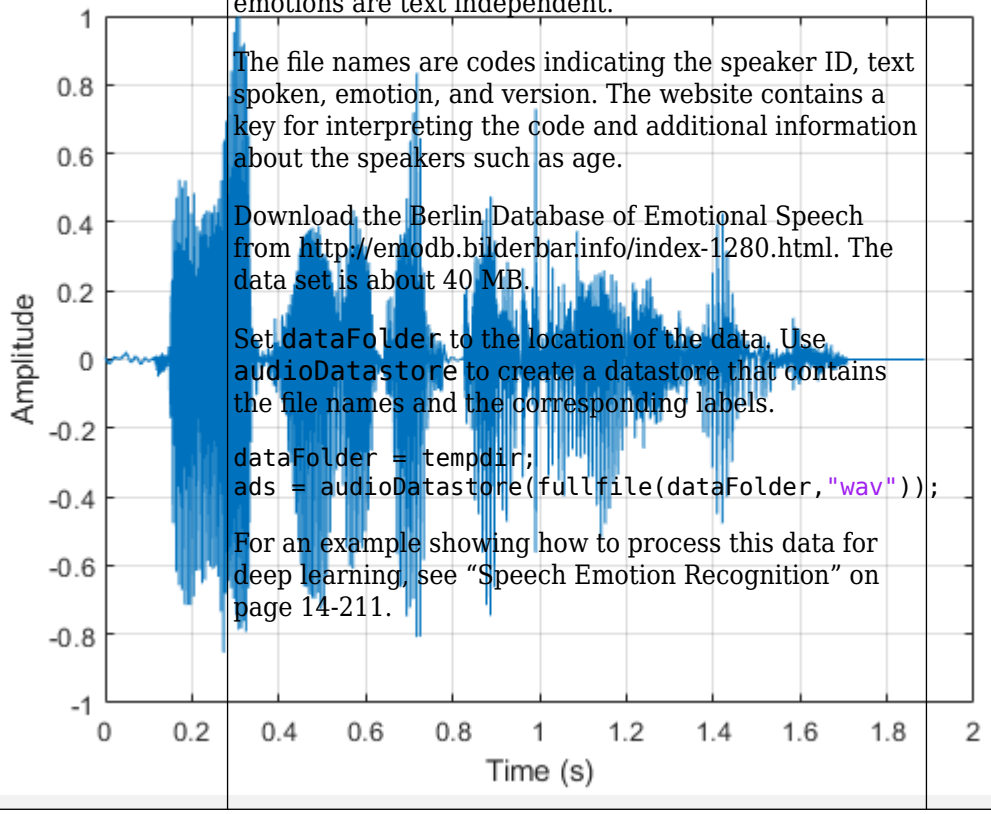
Data	Description	Task

Free Spoken Digit Dataset	The Free Spoken Digit Dataset, as of January 29, 2019, consists of 2000 recordings of the English digits 0 through 9 obtained from four speakers. Two of the speakers in this version are native speakers of American English and two speakers are nonnative speakers of English with a Belgium French and German accent respectively. The data is sampled at 8000 Hz.
	
	Download the Free Spoken Digit Dataset (FSDD) recordings from https://github.com/Jakobovski/free-spoken-digit-dataset.
	
	Set `dataFolder` to the location of the data. Use `audioDatastore` to create a datastore that contains the file names and the corresponding labels.
	
	`dataFolder = fullfile(tempdir,'free-spoken-digit-dataset','recordings');`
	`ads = audioDatastore(dataFolder);`
	
	For an example showing how to process this data for deep learning, see "Spoken Digit Recognition with Wavelet Scattering and Deep Learning" on page 14-223.
		Audio classification, speech recognition.

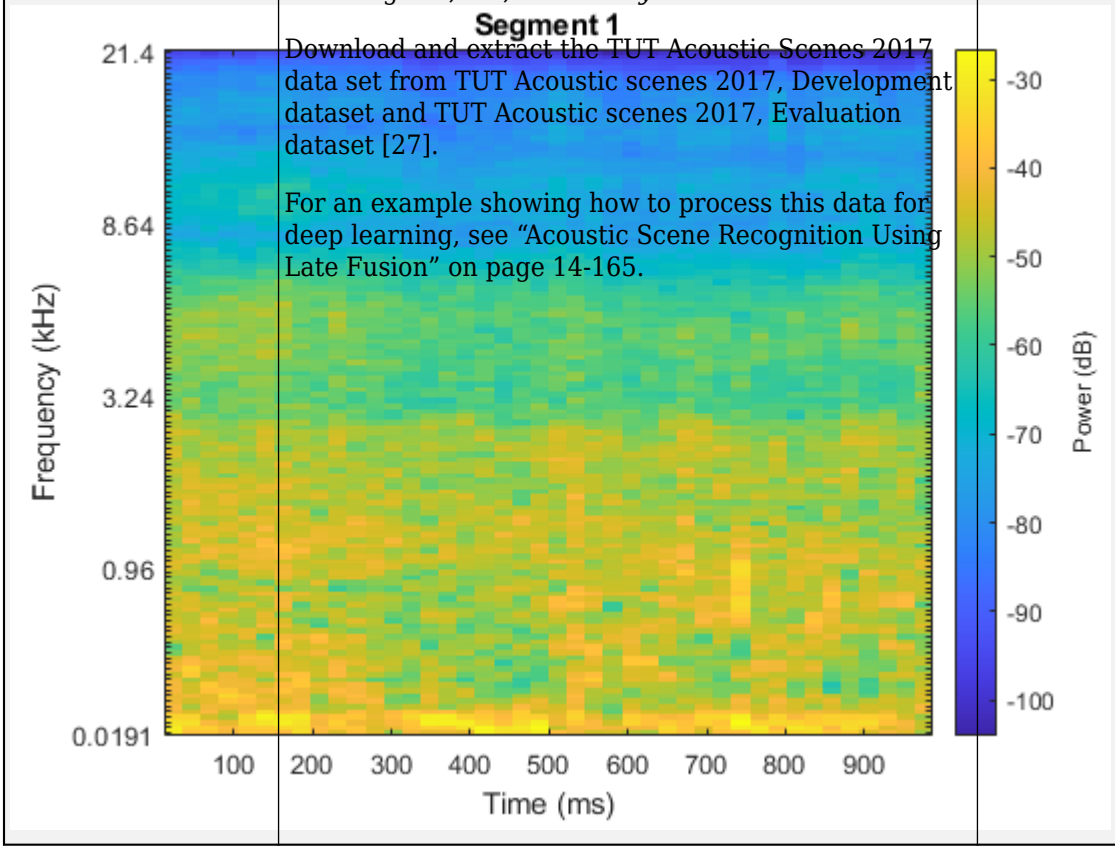


Data	Description	Task

Berlin Database of Emotional Speech	The Berlin Database of Emotional Speech contains 535 utterances spoken by 10 actors intended to convey one of the following emotions: anger, boredom, disgust, anxiety/fear, happiness, sadness, or neutral [26]. The emotions are text independent.
	
	The file names are codes indicating the speaker ID, text spoken, emotion, and version. The website contains a key for interpreting the code and additional information about the speakers such as age.
	
	Download the Berlin Database of Emotional Speech from http://emodb.bilderbar.info/index-1280.html. The data set is about 40 MB.
	
	Set `dataFolder` to the location of the data. Use `audioDatastore` to create a datastore that contains the file names and the corresponding labels.
	
	`dataFolder = tempdir;`
	`ads = audioDatastore(fullfile(dataFolder,"wav"));`
	
	For an example showing how to process this data for deep learning, see “Speech Emotion Recognition” on page 14-211.
		Audio classification, speech recognition.

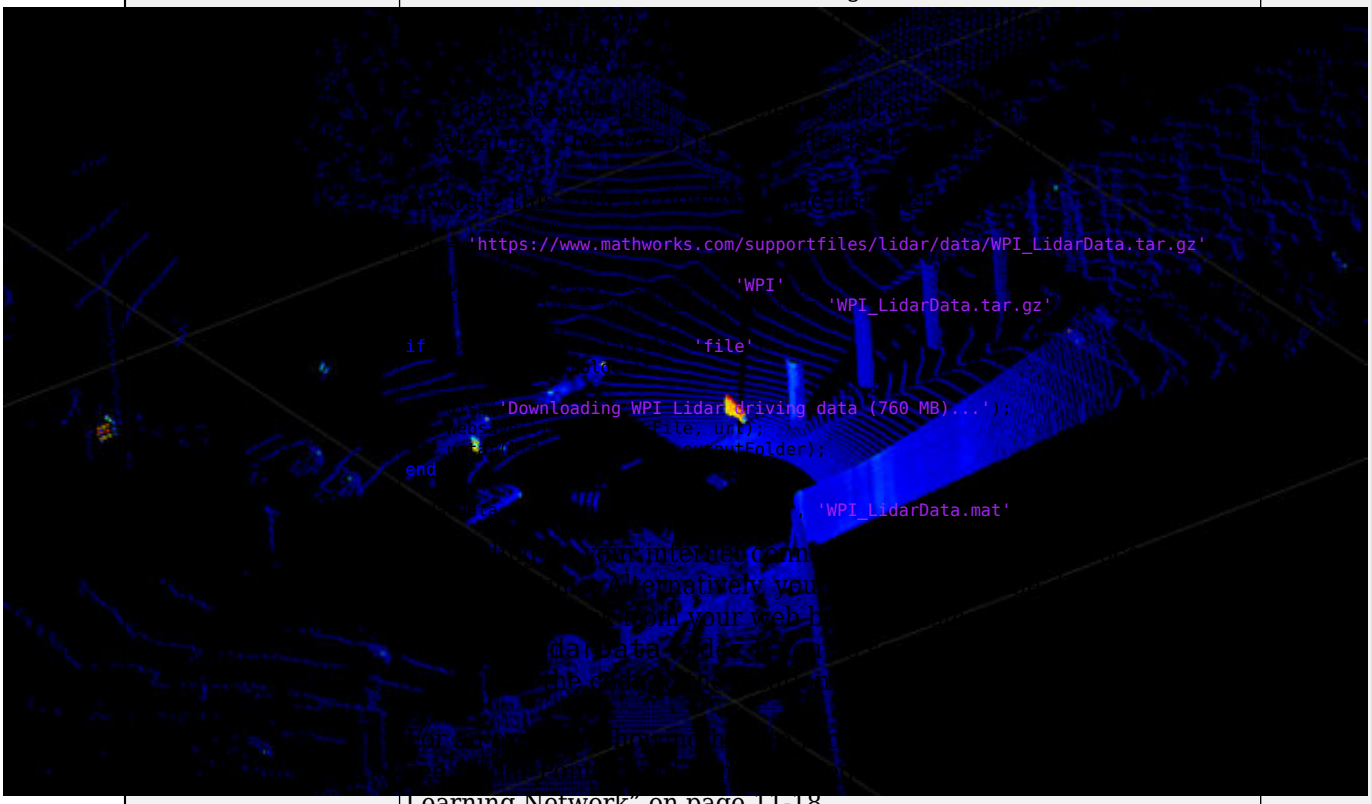


Data	Description	Task
TUT Acoustic Scenes 2017	The TUT Acoustic Scenes 2017 data set consists of 10-second audio segments from 15 acoustic scenes including *bus*, *car*, and *library*. Download and extract the TUT Acoustic Scenes 2017 data set from TUT Acoustic scenes 2017, Development dataset and TUT Acoustic scenes 2017, Evaluation dataset [27]. For an example showing how to process this data for deep learning, see "Acoustic Scene Recognition Using Late Fusion" on page 14-165.	Acoustic scene classification

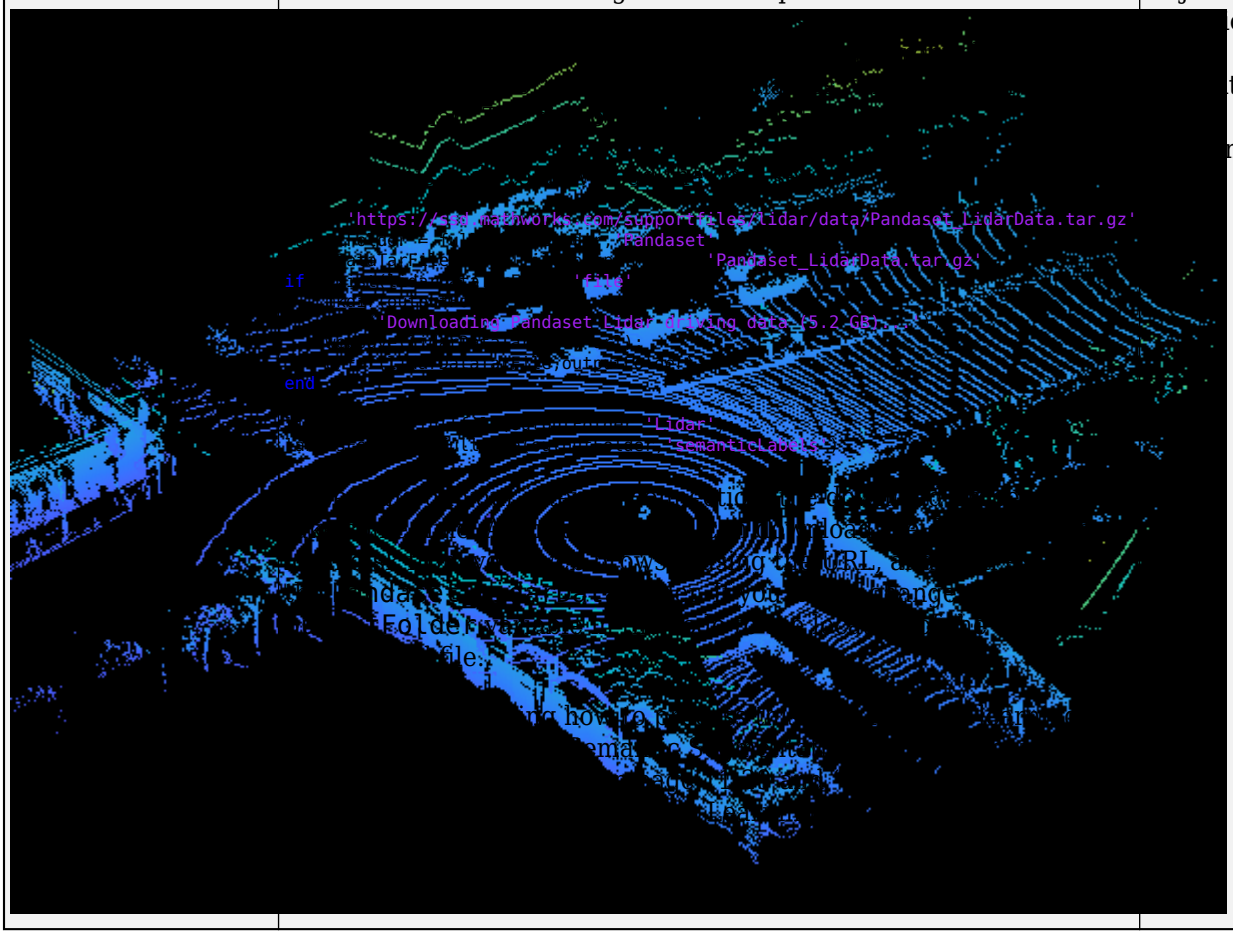


Point Cloud Data Sets

Data	Description	Task
WPI Lidar Data	The WPI Lidar data is collected using an Ouster OS1 sensor. It	Semant
	Learning Network” on page 11-18.	



Data	Description	Task
PandaSet Data	PandaSet contains 2560 organized lidar point cloud scans of various	Object



References

[1] Lake, Brenden M., Ruslan Salakhutdinov, and Joshua B. Tenenbaum. “Human-Level Concept Learning through Probabilistic Program Induction.” *Science* 350, no. 6266 (December 11, 2015): 1332–38. https://doi.org/10.1126/science.aab3050.

[2] The TensorFlow Team. "Flowers" https://www.tensorflow.org/datasets/catalog/tf_flowers.

[3] Kat, *Tulips*, image, https://www.flickr.com/photos/swimparallel/3455026124. Creative Commons License (CC BY).

[4] Rob Bertholf, *Sunflowers*, image, https://www.flickr.com/photos/robbertholf/20777358950. Creative Commons 2.0 Generic License.

[5] Parvin, *Roses*, image, https://www.flickr.com/photos/55948751@N00. Creative Commons 2.0 Generic License.

[6] John Haslam, *Dandelions*, image, https://www.flickr.com/photos/foxypar4/645330051. Creative Commons 2.0 Generic License.

[7] Krizhevsky, Alex. "Learning Multiple Layers of Features from Tiny Images." MSc thesis, University of Toronto, 2009. https://www.cs.toronto.edu/%7Ekriz/learning-features-2009-TR.pdf.

[8] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. “Semantic Object Classes in Video: A High-Definition Ground Truth Database.” *Pattern Recognition Letters* 30, no. 2 (January 2009): 88–97. https://doi.org/10.1016/j.patrec.2008.04.005.

[9] Kemker, Ronald, Carl Salvaggio, and Christopher Kanan. “High-Resolution Multispectral Dataset for Semantic Segmentation.” *ArXiv:1703.01918 [Cs]*, March 6, 2017. https://arxiv.org/abs/1703.01918.

[10] Isensee, Fabian, Philipp Kickingereder, Wolfgang Wick, Martin Bendszus, and Klaus H. Maier-Hein. “Brain Tumor Segmentation and Radiomics Survival Prediction: Contribution to the BRATS 2017 Challenge.” In *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*, edited by Alessandro Crimi, Spyridon Bakas, Hugo Kuijf, Bjoern Menze, and Mauricio Reyes, 10670: 287–97. Cham, Switzerland: Springer International Publishing, 2018. https://doi.org/10.1007/978-3-319-75238-9_25.

[11] Ehteshami Bejnordi, Babak, Mitko Veta, Paul Johannes van Diest, Bram van Ginneken, Nico Karssemeijer, Geert Litjens, Jeroen A. W. M. van der Laak, et al. “Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer.” *JAMA* 318, no. 22 (December 12, 2017): 2199. https://doi.org/10.1001/jama.2017.14585.

[12] McCollough, C.H., Chen, B., Holmes, D., III, Duan, X., Yu, Z., Yu, L., Leng, S., Fletcher, J. (2020). Data from Low Dose CT Image and Projection Data [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/9npb-2637.

[13] Grants EB017095 and EB017185 (Cynthia McCollough, PI) from the National Institute of Biomedical Imaging and Bioengineering.

[14] Grubinger, Michael, Paul Clough, Henning Müller, and Thomas Deselaers. "The IAPR TC-12 Benchmark: A New Evaluation Resource for Visual Information Systems." *Proceedings of the OntoImage 2006 Language Resources For Content-Based Image Retrieval*. Genoa, Italy. Vol. 5, May 2006, p. 10.

[15] Ignatov, Andrey, Luc Van Gool, and Radu Timofte. “Replacing Mobile Camera ISP with a Single Deep Learning Model.” *ArXiv:2002.05509 [Cs, Eess]*, February 13, 2020. https://arxiv.org/abs/2002.05509. Project Website.

[16] Chen, Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. “Learning to See in the Dark.” *ArXiv:1805.01934 [Cs]*, May 4, 2018. https://arxiv.org/abs/1805.01934.

[17] LIVE: Laboratory for Image and Video Engineering. https://live.ece.utexas.edu/research/ChallengeDB/index.html.

[18] Liznerski, Philipp, Lukas Ruff, Robert A. Vandermeulen, Billy Joe Franks, Marius Kloft, and Klaus-Robert Müller. "Explainable Deep One-Class Classification." *ArXiv:2007.01760 [Cs, Stat]*, March 18, 2021. http://arxiv.org/abs/2007.01760.

[19] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. "Multidimensional Curve Classification Using Passing-through Regions." *Pattern Recognition Letters* 20, no. 11–13 (November 1999): 1103–11. https://doi.org/10.1016/S0167-8655(99)00077-X.

[20] Kudo, Mineichi, Jun Toyama, and Masaru Shimbo. *Japanese Vowels Data Set*. Distributed by UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels

[21] Saxena, Abhinav, Kai Goebel. "Turbofan Engine Degradation Simulation Data Set." *NASA Ames Prognostics Data Repository* https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/, NASA Ames Research Center, Moffett Field, CA.

[22] Rieth, Cory A., Ben D. Amsel, Randy Tran, and Maia B. Cook. "Additional Tennessee Eastman Process Simulation Data for Anomaly Detection Evaluation." *Harvard Dataverse*, Version 1, 2017. https://doi.org/10.7910/DVN/6C3JR1.

[23] Goldberger, Ary L., Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation* 101, No. 23, 2000, pp. e215–e220. https://circ.ahajournals.org/content/101/23/e215.full.

[24] Laguna, Pablo, Roger G. Mark, Ary L. Goldberger, and George B. Moody. "A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG." *Computers in Cardiology* 24, 1997, pp. 673–676.

[25] Warden, Pete. "Speech Commands: A public dataset for single-word speech recognition", 2017. Available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz. Copyright Google 2017. The Speech Commands Dataset is licensed under the Creative Commons Attribution 4.0 license, available here: https://creativecommons.org/licenses/by/4.0/legalcode.

[26] Burkhardt, Felix, Astrid Paeschke, Melissa A. Rolfes, Walter F. Sendlmeier, and Benjamin Weiss. "A Database of German Emotional Speech." *Proceedings of Interspeech 2005*. Lisbon, Portugal: International Speech Communication Association, 2005.

[27] Mesaros, Annamaria, Toni Heittola, and Tuomas Virtanen. "Acoustic scene classification: an overview of DCASE 2017 challenge entries." In *2018 16th International Workshop on Acoustic Signal Enhancement (IWAENC)*, pp. 411-415. IEEE, 2018.

[28] Hesai and Scale. PandaSet. https://scale.com/open-datasets/pandaset

See Also

`trainingOptions` | `trainNetwork`

More About

- Deep Network Designer
- “Pretrained Deep Neural Networks” on page 1-8
- “Create Simple Deep Learning Network for Classification” on page 3-47
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Deep Learning in MATLAB” on page 1-2

Choose an App to Label Ground Truth Data

You can use Computer Vision Toolbox, Automated Driving Toolbox, Lidar Toolbox™, Audio Toolbox, and Signal Processing Toolbox apps to label ground truth data. Use this labeled data to validate or train algorithms such as image classifiers, object detectors, semantic segmentation networks, instance segmentation networks, and deep learning applications. The choice of labeling app depends on several factors, including the supported data sources, labels, and types of automation.

One key consideration is the type of data that you want to label.

- If your data is an image collection, use the **Image Labeler** app. An image collection is an unordered set of images that can vary in size. For example, you can use the app to label images of books for training a classifier. The **Image Labeler** can also handle very large images (at least one dimension >8K).
- If your data is a single video or image sequence, use the **Video Labeler** app. An image sequence is an ordered set of images that resembles a video. For example, you can use this app to label a video or image sequence of cars driving on a highway for training an object detector.
- If your data includes multiple time-overlapped signals, such as videos, image sequences, or lidar signals, use the **Ground Truth Labeler** app. For example, you can label data for a single scene captured by multiple sensors mounted on a vehicle.
- If your data is only a lidar signal, use the **Lidar Labeler**. For example, you can use this app to label data captured from a point cloud sensor.
- If your data consists of single-channel or multichannel one-dimensional signals, use the **Signal Labeler**. For example, you can label biomedical, speech, communications, or vibration data. To perform audio-specific tasks, such as speech detection, speech-to-text transcription, and recording new audio, use the **Audio Labeler** app.

This table summarizes the key features of the labeling apps.

Labeling App	Data Sources	Label Support	Automation	Additional Features
Image Labeler	• Image collections • Very large images (at least one dimension >8K)	• Rectangle regions of interest (ROIs) • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms • Custom automation algorithms • Blocked image automation algorithms	• View visual summary of labeled data

Labeling App	Data Sources	Label Support	Automation	Additional Features
Video Labeler	• Videos • Image sequences • Custom image data sources	• Rectangle ROIs • Projected cuboid (ROIs) • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms • Custom automation algorithms • Temporal automation algorithms	• View visual summary of labeled data
Ground Truth Labeler	• Videos • Image sequences • Custom image data sources • Point cloud sequences (PCD or PLY files) • Velodyne® lidar files • Rosbags (requires ROS Toolbox)	• Rectangle ROIs • Projected cuboid (ROIs) • Cuboid ROIs • Line ROIs • Pixel ROIs • Polygon ROIs • Sublabels • Attributes • Scenes	• Built-in automation algorithms, including vehicle and lane detection algorithms and a point cloud temporal interpolation algorithm • Custom automation algorithms • Temporal automation algorithms • Multisignal automation	• View visual summary of labeled data • Connect external tool to app for displaying time-synchronized signals, such as lidar or CAN bus data • Customize loading interface to support additional data sources

Labeling App	Data Sources	Label Support	Automation	Additional Features
Lidar Labeler	• Point cloud sequences (PCD or PLY files) • Velodyne lidar files • LAS/LAZ file sequences • Rosbags (requires ROS Toolbox)	• Cuboid ROIs • Attributes • Scenes	• Built-in automation algorithms, including a lidar object tracker and point cloud temporal interpolator • Custom automation algorithms • Temporal automation algorithms	• View the cuboid labels in top, side, and front views • Save and reuse custom camera views • Connect to external tool to display time-synchronized signals for ease of labeling, such as videos, to use as a reference while labeling
Signal Labeler	• Numeric arrays, MATLAB timetables, and `labeledSignalSet` objects in the MATLAB workspace • MAT-files and CSV files • Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC)	• Time-based ROIs • Time-based points • Attributes • Sublabels	• Built-in peak labeling • Custom automation algorithms	• Expand, collapse, and browse details of labeled data • View signal spectra and spectrograms • Label ROIs and points using the spectrogram • Label signals in bulk • Use Label Viewer to view and compare labels
Audio Labeler	• Audio files (WAVE, OGG, FLAC, AU, AIFF, AIFC, MP3, MPEG-4 AAC) • `labeledSignalSet` objects in the MATLAB workspace or in MAT-files	• Time-based ROIs • File-level labels	• Speech detection • Speech-to-text transcription (requires Audio Toolbox extended functionality for speech2text)	• Audio playback • Audio recording • Inspect audio file information

See Also

More About

- “Get Started with the Image Labeler” (Computer Vision Toolbox)
- “Get Started with the Video Labeler” (Computer Vision Toolbox)
- “Get Started with the Ground Truth Labeler” (Automated Driving Toolbox)
- “Get Started with the Lidar Labeler” (Lidar Toolbox)
- “Using Signal Labeler App” (Signal Processing Toolbox)
- “Label Audio Using Audio Labeler” (Audio Toolbox)

Deep Learning Code Generation

Code Generation for Deep Learning Networks

This example shows how to perform code generation for an image classification application that uses deep learning. It uses the `codegen` command to generate a MEX function that runs prediction by using image classification networks such as MobileNet-v2, ResNet, and GoogLeNet.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

`mobilenetv2_predict` Entry-Point Function

MobileNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 155 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The network has an image input size of 224-by-224. Use the `analyzeNetwork` function to display an interactive visualization of the deep learning network architecture.

```
net = mobilenetv2();
analyzeNetwork(net);
```

The `mobilenetv2_predict.m` entry-point function takes an image input and runs prediction on the image using the pretrained MobileNet-v2 convolutional neural network. The function uses a persistent object *mynet* to load the series network object and reuses the persistent object for prediction on subsequent calls.

```
type('mobilenetv2_predict.m')
```

```
% Copyright 2017-2019 The MathWorks, Inc.

function out = mobilenetv2_predict(in)
```

```
%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('mobilenetv2','mobilenetv2');
end

% pass in input
out = mynet.predict(in);
```

Run MEX Code Generation

To generate CUDA code for the `mobilenetv2_predict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command and specify an input size of [224,224,3]. This value corresponds to the input layer size of the MobileNet-v2 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg mobilenetv2_predict -args {ones(224,224,3)} -report

Code generation successful: View report
```

Generated Code Description

The series network is generated as a C++ class containing an array of 155 layer classes and functions to set up, call predict, and clean up the network.

```
class b_mobilenetv2_0
{
   ....
   public:
     b_mobilenetv2_0();
     void setup();
     void predict();
     void cleanup();
     ~b_mobilenetv2_0();
};
```

The `setup()` method of the class sets up handles and allocates memory for each layer of the network object. The `predict()` method performs prediction for each of the 155 layers in the network.

The entry-point function `mobilenetv2_predict()` in the generated code file `mobilenetv2_predict.cu` constructs a static object of *b_mobilenetv2* class type and invokes setup and predict on this network object.

```
static b_mobilenetv2_0 mynet;
static boolean_T mynet_not_empty;

/* Function Definitions */
void mobilenetv2_predict(const real_T in[150528], real32_T out[1000])
{
  if (!mynet_not_empty) {
    DeepLearningNetwork_setup(&mynet);
```

```
    mynet_not_empty = true;
  }

   /*  pass in input    */
   DeepLearningNetwork_predict(&mynet, in, out);
 }
```

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For instance, files cnn_mobilenetv2_conv*_w and cnn_mobilenetv2_conv*_b correspond to weights and bias parameters for the convolution layers in the network. To see a list of the generated files, use:

```
dir(fullfile(pwd, 'codegen', 'mex', 'mobilenetv2_predict'))
```

Run Generated MEX

Load an input image.

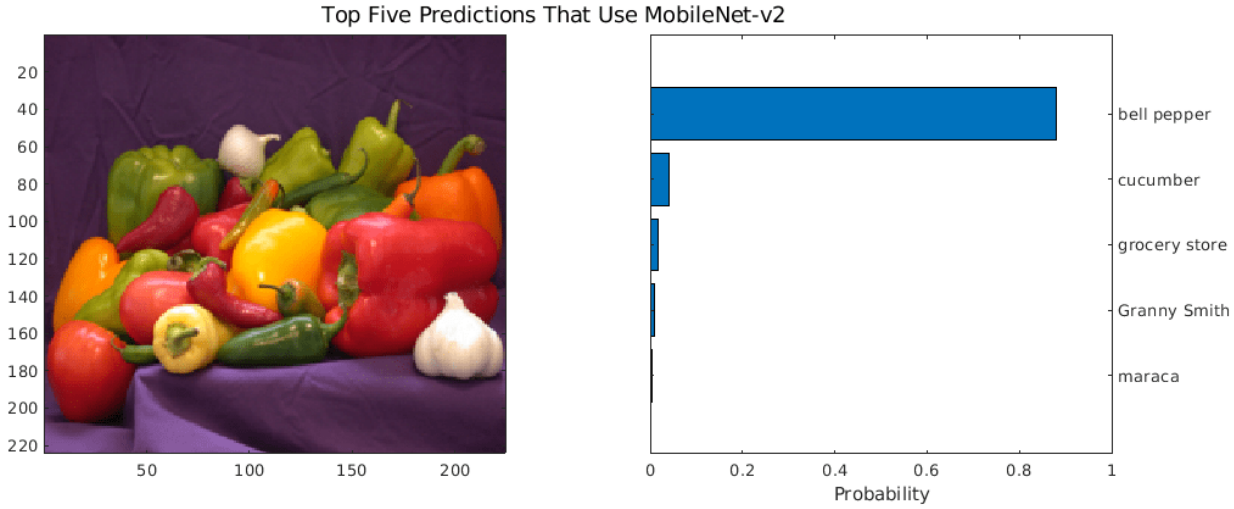
```
im = imread('peppers.png');
imshow(im);
```

Call `mobilenetv2_predict_mex` on the input image.

```
im = imresize(im, [224,224]);
predict_scores = mobilenetv2_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');
classNames = net.Layers(end).ClassNames;
classNamesTop = classNames(indx(1:5));

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top Five Predictions That Use MobileNet-v2')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

Classification of Images by Using ResNet-50 network

You can also use the DAG network ResNet-50 for image classification. A pretrained ResNet-50 model for MATLAB is available in the ResNet-50 support package of Deep Learning Toolbox. To download and install the support package, use the Add-On Explorer. To learn more about finding and installing add-ons, see “Get and Manage Add-Ons”.

```
net = resnet50;
disp(net)
```

```
  DAGNetwork with properties:

         Layers: [177×1 nnet.cnn.layer.Layer]
    Connections: [192×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_fc1000'}
```

Run MEX Code Generation

To generate CUDA code for the `resnet_predict.m` entry-point function,create a GPU code configuration object for a MEX target and set the target language to C++. This entry-point function calls the `resnet50` function to load the network and perform prediction on the input image.

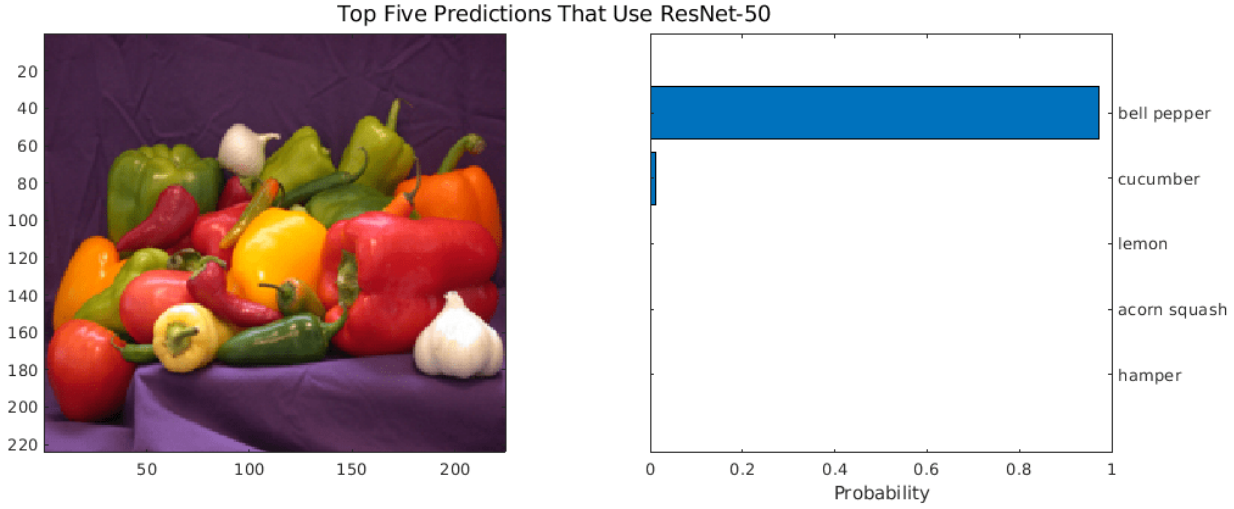
```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg resnet_predict -args {ones(224,224,3)} -report

Code generation successful: View report
```

Call `resnet_predict_mex` on the input image.

```
predict_scores = resnet_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');
classNames = net.Layers(end).ClassNames;
classNamesTop = classNames(indx(1:5));

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top Five Predictions That Use ResNet-50')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

Classification of Images by Using GoogLeNet (Inception) network

A pretrained GoogLeNet model for MATLAB is available in the GoogLeNet support package of Deep Learning Toolbox. To download and install the support package, use the Add-On Explorer. To learn more about finding and installing add-ons, see “Get and Manage Add-Ons”.

```
net = googlenet;
disp(net)
```

```
  DAGNetwork with properties:

         Layers: [144×1 nnet.cnn.layer.Layer]
    Connections: [170×2 table]
     InputNames: {'data'}
    OutputNames: {'output'}
```

Run MEX Code Generation

Generate CUDA code for the `googlenet_predict.m` entry-point function. This entry-point function calls the `googlenet` function to load the network and perform prediction on the input image. To generate code for this entry-point function, create a GPU configuration object for MEX target.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg googlenet_predict -args {ones(224,224,3)} -report
```

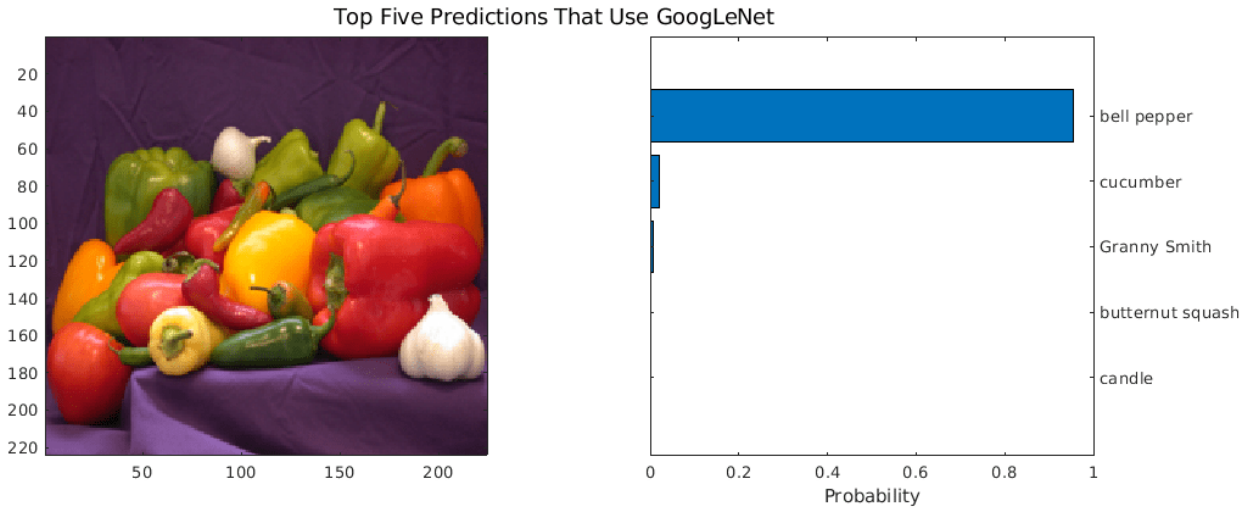
```
Code generation successful: View report
```

Call `googlenet_predict_mex` on the input image.

```
im = imresize(im, [224,224]);
predict_scores = googlenet_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');
classNames = net.Layers(end).ClassNames;
classNamesTop = classNames(indx(1:5));

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top Five Predictions That Use GoogLeNet')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Code Generation for Semantic Segmentation Network

This example shows code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction on a DAG Network object for SegNet [1], a deep learning network for image segmentation.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Segmentation Network

SegNet [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. It is a deep encoder-decoder multi-class pixel-wise segmentation network trained on the CamVid [2] dataset and imported into MATLAB® for inference. The SegNet [1] is trained to segment pixels belonging to 11 classes that include Sky, Building, Pole, Road, Pavement, Tree, SignSymbol, Fence, Car, Pedestrian, and Bicyclist.

For information regarding training a semantic segmentation network in MATLAB by using the CamVid [2] dataset, see “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

The `segnet_predict` Entry-Point Function

The `segnet_predict.m` entry-point function takes an image input and performs prediction on the image by using the deep learning network saved in the `SegNet.mat` file. The function loads the network object from the `SegNet.mat` file into a persistent variable *mynet* and reuses the persistent variable on subsequent prediction calls.

```
type('segnet_predict.m')
```

```
function out = segnet_predict(in)
%#codegen
% Copyright 2018-2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('SegNet.mat');
end

% pass in input
out = predict(mynet,in);
```

Get Pretrained SegNet DAG Network Object

```
net = getSegNet();
```

The DAG network contains 91 layers including convolution, batch normalization, pooling, unpooling, and the pixel classification output layers. Use the `analyzeNetwork` function to display an interactive visualization of the deep learning network architecture.

```
analyzeNetwork(net);
```

Run MEX Code Generation

To generate CUDA code for the `segnet_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [360,480,3]. This value corresponds to the input layer size of `SegNet`.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg segnet_predict -args {ones(360,480,3,'uint8')} -report

Code generation successful: View report
```

Run Generated MEX

Load and display an input image. Call `segnet_predict_mex` on the input image.

```
im = imread('gpucoder_segnet_image.png');
imshow(im);
```

```
predict_scores = segnet_predict_mex(im);
```

The *predict_scores* variable is a three-dimensional matrix that has 11 channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get pixel-wise labels.

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the input image and display the segmented region.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];

cmap = camvidColorMap();
```

```
SegmentedImage = labeloverlay(im,argmax,'ColorMap',cmap);
figure
imshow(SegmentedImage);
pixelLabelColorbar(cmap,classes);
```

References

[1] Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." *arXiv preprint arXiv:1511.00561,* 2015.

[2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters* Vol 30, Issue 2, 2009, pp 88-97.

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Lane Detection Optimized with GPU Coder

This example shows how to generate CUDA® code from a deep learning network, represented by a `SeriesNetwork` object. In this example, the series network is a convolutional neural network that can detect and output lane marker boundaries from an image.

Prerequisites

- CUDA enabled NVIDIA® GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- OpenCV libraries for video read and image display operations.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained SeriesNetwork

```
[laneNet, coeffMeans, coeffStds] = getLaneDetectionNetworkGPU();
```

This network takes an image as an input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by the parabolic equation: $y = ax^2 + bx + c$, where y is the lateral offset and x is the longitudinal distance from the vehicle. The network outputs the three parameters a, b, and c per lane. The network architecture is similar to `AlexNet` except that the last few layers are replaced by a smaller fully connected layer and regression output layer. To view the network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(laneNet)
```

Examine Main Entry-Point Function

```
type detect_lane.m
```

```
function [laneFound, ltPts, rtPts] = detect_lane(frame, laneCoeffMeans, laneCoeffStds)
% From the networks output, compute left and right lane points in the image
% coordinates. The camera coordinates are described by the caltech mono
% camera model.

%#codegen

% A persistent object mynet is used to load the series network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
```

```
% reloading the network object.
persistent lanenet;

if isempty(lanenet)
    lanenet = coder.loadDeepLearningNetwork('laneNet.mat', 'lanenet');
end

lanecoeffsNetworkOutput = lanenet.predict(permute(frame, [2 1 3]));

% Recover original coeffs by reversing the normalization steps

params = lanecoeffsNetworkOutput .* laneCoeffStds + laneCoeffMeans;

isRightLaneFound = abs(params(6)) > 0.5; %c should be more than 0.5 for it to be a right lane
isLeftLaneFound =  abs(params(3)) > 0.5;

vehicleXPoints = 3:30; %meters, ahead of the sensor
ltPts = coder.nullcopy(zeros(28,2,'single'));
rtPts = coder.nullcopy(zeros(28,2,'single'));

if isRightLaneFound && isLeftLaneFound
    rtBoundary = params(4:6);
    rt_y = computeBoundaryModel(rtBoundary, vehicleXPoints);
    ltBoundary = params(1:3);
    lt_y = computeBoundaryModel(ltBoundary, vehicleXPoints);

    % Visualize lane boundaries of the ego vehicle
    tform = get_tformToImage;
    % map vehicle to image coordinates
    ltPts =  tform.transformPointsInverse([vehicleXPoints', lt_y']);
    rtPts =  tform.transformPointsInverse([vehicleXPoints', rt_y']);
    laneFound = true;
else
    laneFound = false;
end

end

function yWorld = computeBoundaryModel(model, xWorld)
    yWorld = polyval(model, xWorld);
end

function tform = get_tformToImage
% Compute extrinsics based on camera setup
yaw = 0;
pitch = 14; % pitch of the camera in degrees
roll = 0;

translation = translationVector(yaw, pitch, roll);
rotation    = rotationMatrix(yaw, pitch, roll);

% Construct a camera matrix
focalLength    = [309.4362, 344.2161];
principalPoint = [318.9034, 257.5352];
Skew = 0;

camMatrix = [rotation; translation] * intrinsicMatrix(focalLength, ...
    Skew, principalPoint);
```

```
% Turn camMatrix into 2-D homography
tform2D = [camMatrix(1,:); camMatrix(2,:); camMatrix(4,:)]; % drop Z

tform = projective2d(tform2D);
tform = tform.invert();
end

function translation = translationVector(yaw, pitch, roll)
SensorLocation = [0 0];
Height = 2.1798;    % mounting height in meters from the ground
rotationMatrix = (...
    rotZ(yaw)*... % last rotation
    rotX(90-pitch)*...
    rotZ(roll)... % first rotation
    );


% Adjust for the SensorLocation by adding a translation
sl = SensorLocation;

translationInWorldUnits = [sl(2), sl(1), Height];
translation = translationInWorldUnits*rotationMatrix;
end

%--------------------------------------------------------------------
% Rotation around X-axis
function R = rotX(a)
a = deg2rad(a);
R = [...
    1   0        0;
    0   cos(a)  -sin(a);
    0   sin(a)   cos(a)];

end

%--------------------------------------------------------------------
% Rotation around Y-axis
function R = rotY(a)
a = deg2rad(a);
R = [...
    cos(a)  0 sin(a);
    0       1 0;
    -sin(a) 0 cos(a)];

end

%--------------------------------------------------------------------
% Rotation around Z-axis
function R = rotZ(a)
a = deg2rad(a);
R = [...
    cos(a) -sin(a) 0;
    sin(a)  cos(a) 0;
    0       0      1];
end

%--------------------------------------------------------------------
```

```
% Given the Yaw, Pitch, and Roll, determine the appropriate Euler angles
% and the sequence in which they are applied to align the camera's
% coordinate system with the vehicle coordinate system. The resulting
% matrix is a Rotation matrix that together with the Translation vector
% defines the extrinsic parameters of the camera.
function rotation = rotationMatrix(yaw, pitch, roll)

rotation = (...
    rotY(180)*...            % last rotation: point Z up
    rotZ(-90)*...            % X-Y swap
    rotZ(yaw)*...            % point the camera forward
    rotX(90-pitch)*...       % "un-pitch"
    rotZ(roll)...            % 1st rotation: "un-roll"
    );
end

function intrinsicMat = intrinsicMatrix(FocalLength, Skew, PrincipalPoint)
intrinsicMat = ...
    [FocalLength(1)  , 0                    , 0; ...
     Skew            , FocalLength(2)   , 0; ...
     PrincipalPoint(1), PrincipalPoint(2), 1];
end
```

Generate Code for Network and Post-Processing Code

The network computes parameters a, b, and c that describe the parabolic equation for the left and right lane boundaries.

From these parameters, compute the x and y coordinates corresponding to the lane positions. The coordinates must be mapped to image coordinates. The function `detect_lane.m` performs all these computations. Generate CUDA code for this function by creating a GPU code configuration object for a `'lib'` target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command.

```
cfg = coder.gpuConfig('lib');
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
cfg.TargetLang = 'C++';
inputs = {ones(227,227,3,'single'),ones(1,6,'double'),ones(1,6,'double')};
codegen -args inputs -config cfg detect_lane

Code generation successful: View report
```

Generated Code Description

The series network is generated as a C++ class containing an array of 23 layer classes.

```
class c_lanenet {
 public:
  int32_T batchSize; int32_T numLayers; real32_T *inputData; real32_T
  *outputData; MWCNNLayer *layers[23];
 public:
  c_lanenet(void); void setup(void); void predict(void); void
  cleanup(void); ~c_lanenet(void);
};
```

The `setup()` method of the class sets up handles and allocates memory for each layer object. The `predict()` method invokes prediction for each of the 23 layers in the network.

The cnn_lanenet_conv*_w and cnn_lanenet_conv*_b files are the binary weights and bias file for convolution layer in the network. The cnn_lanenet_fc*_w and cnn_lanenet_fc*_b files are the binary weights and bias file for fully connected layer in the network.

```
codegendir = fullfile('codegen', 'lib', 'detect_lane');
dir(codegendir)
```

```
.                                       MWReLULayer.o
..                                      MWReLULayerImpl.cu
.gitignore                              MWReLULayerImpl.hpp
DeepLearningNetwork.cu                  MWReLULayerImpl.o
DeepLearningNetwork.h                   MWTargetNetworkImpl.cu
DeepLearningNetwork.o                   MWTargetNetworkImpl.hpp
MWCNNLayer.cpp                          MWTargetNetworkImpl.o
MWCNNLayer.hpp                          MWTensor.hpp
MWCNNLayer.o                            MWTensorBase.cpp
MWCNNLayerImpl.cu                       MWTensorBase.hpp
MWCNNLayerImpl.hpp                      MWTensorBase.o
MWCNNLayerImpl.o                        _clang-format
MWCUSOLVERUtils.cpp                     buildInfo.mat
MWCUSOLVERUtils.hpp                     cnn_lanenet0_0_conv1_b.bin
MWCUSOLVERUtils.o                       cnn_lanenet0_0_conv1_w.bin
MWCudaDimUtility.hpp                    cnn_lanenet0_0_conv2_b.bin
MWCustomLayerForCuDNN.cpp               cnn_lanenet0_0_conv2_w.bin
MWCustomLayerForCuDNN.hpp               cnn_lanenet0_0_conv3_b.bin
MWCustomLayerForCuDNN.o                 cnn_lanenet0_0_conv3_w.bin
MWElementwiseAffineLayer.cpp            cnn_lanenet0_0_conv4_b.bin
MWElementwiseAffineLayer.hpp            cnn_lanenet0_0_conv4_w.bin
MWElementwiseAffineLayer.o              cnn_lanenet0_0_conv5_b.bin
MWElementwiseAffineLayerImpl.cu         cnn_lanenet0_0_conv5_w.bin
MWElementwiseAffineLayerImpl.hpp        cnn_lanenet0_0_data_offset.bin
MWElementwiseAffineLayerImpl.o          cnn_lanenet0_0_data_scale.bin
MWElementwiseAffineLayerImplKernel.cu   cnn_lanenet0_0_fc6_b.bin
MWElementwiseAffineLayerImplKernel.o    cnn_lanenet0_0_fc6_w.bin
MWFCLayer.cpp                           cnn_lanenet0_0_fcLane1_b.bin
MWFCLayer.hpp                           cnn_lanenet0_0_fcLane1_w.bin
MWFCLayer.o                             cnn_lanenet0_0_fcLane2_b.bin
MWFCLayerImpl.cu                        cnn_lanenet0_0_fcLane2_w.bin
MWFCLayerImpl.hpp                       cnn_lanenet0_0_responseNames.txt
MWFCLayerImpl.o                         codeInfo.mat
MWFusedConvReLULayer.cpp                codedescriptor.dmr
MWFusedConvReLULayer.hpp                compileInfo.mat
MWFusedConvReLULayer.o                  defines.txt
MWFusedConvReLULayerImpl.cu             detect_lane.a
MWFusedConvReLULayerImpl.hpp            detect_lane.cu
MWFusedConvReLULayerImpl.o              detect_lane.h
MWInputLayer.cpp                        detect_lane.o
MWInputLayer.hpp                        detect_lane_data.cu
MWInputLayer.o                          detect_lane_data.h
MWInputLayerImpl.hpp                    detect_lane_data.o
MWKernelHeaders.hpp                     detect_lane_initialize.cu
MWMaxPoolingLayer.cpp                   detect_lane_initialize.h
MWMaxPoolingLayer.hpp                   detect_lane_initialize.o
MWMaxPoolingLayer.o                     detect_lane_internal_types.h
MWMaxPoolingLayerImpl.cu                detect_lane_rtw.mk
```

```
MWMaxPoolingLayerImpl.hpp             detect_lane_terminate.cu
MWMaxPoolingLayerImpl.o               detect_lane_terminate.h
MWNormLayer.cpp                       detect_lane_terminate.o
MWNormLayer.hpp                       detect_lane_types.h
MWNormLayer.o                         examples
MWNormLayerImpl.cu                    gpu_codegen_info.mat
MWNormLayerImpl.hpp                   html
MWNormLayerImpl.o                     interface
MWOutputLayer.cpp                     mean.bin
MWOutputLayer.hpp                     predict.cu
MWOutputLayer.o                       predict.h
MWOutputLayerImpl.cu                  predict.o
MWOutputLayerImpl.hpp                 rtw_proj.tmw
MWOutputLayerImpl.o                   rtwtypes.h
MWReLULayer.cpp
MWReLULayer.hpp
```

Generate Additional Files for Post-Processing the Output

Export mean and std values from the trained network for use during execution.

```
codegendir = fullfile(pwd, 'codegen', 'lib','detect_lane');
fid = fopen(fullfile(codegendir,'mean.bin'), 'w');
A = [coeffMeans coeffStds];
fwrite(fid, A, 'double');
fclose(fid);
```

Main File

Compile the network code by using a main file. The main file uses the OpenCV `VideoCapture` method to read frames from the input video. Each frame is processed and classified until no more frames are read. Before displaying the output for each frame, the outputs are post-processed by using the `detect_lane` function generated in `detect_lane.cu`.

```
type main_lanenet.cu
```

```
/* Copyright 2016 The MathWorks, Inc. */

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/core/types.hpp>
#include <opencv2/highgui.hpp>
#include <list>
#include <cmath>
#include "detect_lane.h"

using namespace cv;
void readData(float *input, Mat& orig, Mat & im)
{
    Size size(227,227);
    resize(orig,im,size,0,0,INTER_LINEAR);
    for(int j=0;j<227*227;j++)
    {
        //BGR to RGB
```

```
        input[2*227*227+j]=(float)(im.data[j*3+0]);
        input[1*227*227+j]=(float)(im.data[j*3+1]);
        input[0*227*227+j]=(float)(im.data[j*3+2]);
    }
}

void addLane(float pts[28][2], Mat & im, int numPts)
{
    std::vector<Point2f> iArray;
    for(int k=0; k<numPts; k++)
    {
        iArray.push_back(Point2f(pts[k][0],pts[k][1]));
    }
    Mat curve(iArray, true);
    curve.convertTo(curve, CV_32S); //adapt type for polylines
    polylines(im, curve, false, CV_RGB(255,255,0), 2, LINE_AA);
}


void writeData(float *outputBuffer, Mat & im, int N, double means[6], double stds[6])
{
    // get lane coordinates
    boolean_T laneFound = 0;
    float ltPts[56];
    float rtPts[56];
    detect_lane(outputBuffer, means, stds, &laneFound, ltPts, rtPts);

    if (!laneFound)
    {
        return;
    }

    float ltPtsM[28][2];
    float rtPtsM[28][2];
    for(int k=0; k<28; k++)
    {
        ltPtsM[k][0] = ltPts[k];
        ltPtsM[k][1] = ltPts[k+28];
        rtPtsM[k][0] = rtPts[k];
        rtPtsM[k][1] = rtPts[k+28];
    }

    addLane(ltPtsM, im, 28);
    addLane(rtPtsM, im, 28);
}

void readMeanAndStds(const char* filename, double means[6], double stds[6])
{
    FILE* pFile = fopen(filename, "rb");
    if (pFile==NULL)
    {
        fputs ("File error",stderr);
        return;
    }

    // obtain file size
    fseek (pFile , 0 , SEEK_END);
    long lSize = ftell(pFile);
```

```
    rewind(pFile);

    double* buffer = (double*)malloc(lSize);

    size_t result = fread(buffer,sizeof(double),lSize,pFile);
    if (result*sizeof(double) != lSize) {
        fputs ("Reading error",stderr);
        return;
    }

    for (int k = 0 ; k < 6; k++)
    {
        means[k] = buffer[k];
        stds[k] = buffer[k+6];
    }
    free(buffer);
}


// Main function
int main(int argc, char* argv[])
{

    float *inputBuffer = (float*)calloc(sizeof(float),227*227*3);
    float *outputBuffer = (float*)calloc(sizeof(float),6);

    if ((inputBuffer == NULL) || (outputBuffer == NULL)) {
        printf("ERROR: Input/Output buffers could not be allocated!\n");
        exit(-1);
    }

    // get ground truth mean and std
    double means[6];
    double stds[6];
    readMeanAndStds("mean.bin", means, stds);

    if (argc < 2)
    {
        printf("Pass in input video file name as argument\n");
        return -1;
    }

    VideoCapture cap(argv[1]);
    if (!cap.isOpened()) {
        printf("Could not open the video capture device.\n");
        return -1;
    }

    cudaEvent_t start, stop;
    float fps = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    Mat orig, im;
    namedWindow("Lane detection demo",WINDOW_NORMAL);
    while(true)
    {
        cudaEventRecord(start);
        cap >> orig;
```

```
        if (orig.empty()) break;
        readData(inputBuffer, orig, im);

        writeData(inputBuffer, orig, 6, means, stds);

        cudaEventRecord(stop);
        cudaEventSynchronize(stop);

        char strbuf[50];
        float milliseconds = -1.0;
        cudaEventElapsedTime(&milliseconds, start, stop);
        fps = fps*.9+1000.0/milliseconds*.1;
        sprintf (strbuf, "%.2f FPS", fps);
        putText(orig, strbuf, Point(200,30), FONT_HERSHEY_DUPLEX, 1, CV_RGB(0,0,0), 2);
        imshow("Lane detection demo", orig);
        if( waitKey(50)%256 == 27 ) break; // stop capturing by pressing ESC    */
    }
    destroyWindow("Lane detection demo");

    free(inputBuffer);
    free(outputBuffer);

    return 0;
}
```

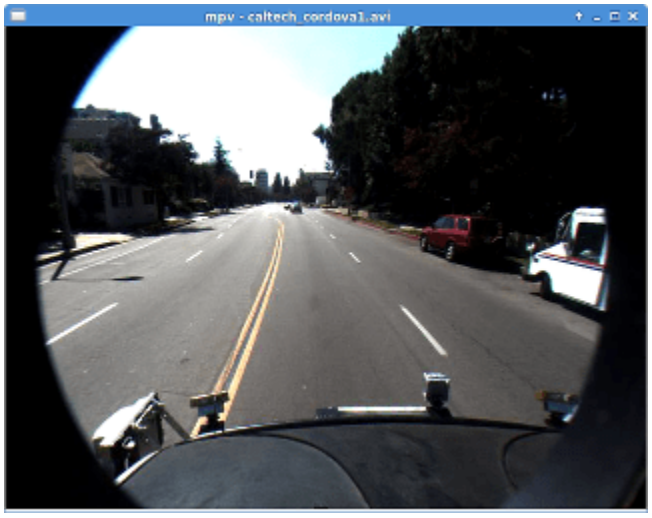
Download Example Video

```
if ~exist('./caltech_cordova1.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/caltech_cordova1.avi';
    websave('caltech_cordova1.avi', url);
end
```

Build Executable

```
if ispc
    setenv('MATLAB_ROOT', matlabroot);
    vcvarsall = mex.getCompilerConfigurations('C++').Details.CommandLineShell;
    setenv('VCVARSALL', vcvarsall);
    system('make_win_lane_detection.bat');
    cd(codegendir);
    system('lanenet.exe ..\..\..\caltech_cordova1.avi');
else
    setenv('MATLAB_ROOT', matlabroot);
    system('make -f Makefile_lane_detection.mk');
    cd(codegendir);
    system('./lanenet ../../../caltech_cordova1.avi');
end
```

Input Screenshot



Output Screenshot

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Code Generation for a Sequence-to-Sequence LSTM Network

This example demonstrates how to generate CUDA® code for a long short-term memory (LSTM) network. The example generates a MEX application that makes predictions at each step of an input timeseries. Two methods are demonstrated: a method using a standard LSTM network, and a method leveraging the stateful behavior of the same LSTM network. This example uses accelerometer sensor data from a smartphone carried on the body and makes predictions on the activity of the wearer. User movements are classified into one of five categories, namely dancing, running, sitting, standing, and walking. The example uses a pretrained LSTM network. For more information on training, see the "Sequence Classification Using Deep Learning" on page 4-2 example from Deep Learning Toolbox™.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see "Third-Party Hardware" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

The `lstmnet_predict` Entry-Point Function

A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence. The `lstmnet_predict.m` entry-point function takes an input sequence and passes it to a trained LSTM network for prediction. Specifically, the function uses the LSTM network trained in the *Sequence to Sequence Classification Using Deep Learning* example. The function loads the network object from the `lstmnet_predict.mat` file into a persistent variable and reuses the persistent object on subsequent prediction calls.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` function.

```
type('lstmnet_predict.m')
```

```
function out = lstmnet_predict(in) %#codegen
```

```
% Copyright 2019-2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('lstmnet.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX

To generate CUDA MEX for the `lstmnet_predict.m` entry-point function, create a GPU configuration object and specify the target to be MEX. Set the target language to C++. Create a deep learning configuration object that specifies the target library as cuDNN. Attach this deep learning configuration object to the GPU configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

At compile time, GPU Coder™ must know the data types of all the inputs to the entry-point function. Specify the type and size of the input argument to the `codegen` (MATLAB Coder) command by using the `coder.typeof` (MATLAB Coder) function. For this example, the input is of double data type with a feature dimension value of three and a variable sequence length. Specifying the sequence length as variable-sized enables us to perform prediction on an input sequence of any length.

```
matrixInput = coder.typeof(double(0),[3 Inf],[false true]);
```

Run the codegen command.

```
codegen -config cfg lstmnet_predict -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run Generated MEX on Test Data

Load the `HumanActivityValidate` MAT-file. This MAT-file stores the variable `XValidate` that contains sample timeseries of sensor readings on which you can test the generated code. Call `lstmnet_predict_mex` on the first observation.

```
load HumanActivityValidate
YPred1 = lstmnet_predict_mex(XValidate{1});
```

`YPred1` is a 5-by-53888 numeric matrix containing the probabilities of the five classes for each of the 53888 time steps. For each time step, find the predicted class by calculating the index of the maximum probability.

```
[~, maxIndex] = max(YPred1, [], 1);
```

Associate the indices of max probability to the corresponding label. Display the first ten labels. From the results, you can see that the network predicted the human to be sitting for the first ten time steps.

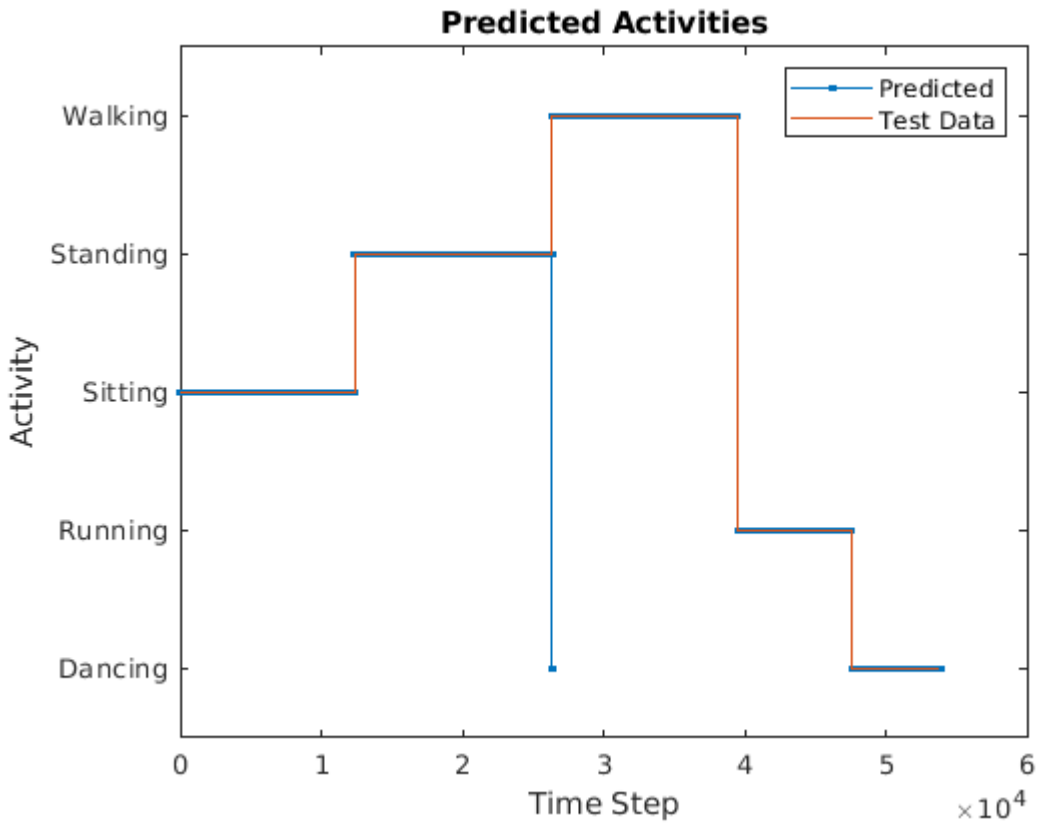
```
labels = categorical({'Dancing', 'Running', 'Sitting', 'Standing', 'Walking'});
predictedLabels1 = labels(maxIndex);
disp(predictedLabels1(1:10)')
```

```
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
```

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
figure
plot(predictedLabels1,'.-');
hold on
plot(YValidate{1});
hold off

xlabel("Time Step")
ylabel("Activity")
title("Predicted Activities")
legend(["Predicted" "Test Data"])
```



Call Generated MEX on an Observation with a Different Sequence Length

Call `lstmnet_predict_mex` on the second observation with a different sequence length. In this example, `XValidate{2}` has a sequence length of 64480 whereas `XValidate{1}` had a sequence length of 53888. The generated code handles prediction correctly because we specified the sequence length dimension to be variable-size.

```
YPred2 = lstmnet_predict_mex(XValidate{2});
[~, maxIndex] = max(YPred2, [], 1);
predictedLabels2 = labels(maxIndex);
disp(predictedLabels2(1:10)')
```

```
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
     Sitting
```

Generate MEX that takes in Multiple Observations

If you want to perform prediction on many observations at once, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell array, and each cell must contain one observation. Each observation must have the same feature dimension, but the sequence lengths may vary. In this example, `XValidate` contains five observations. To generate a MEX that can take `XValidate` as input, specify the input type to be a 5-by-1 cell array. Further, specify that each cell be of the same type as `matrixInput`, the type you specified for the single observation in the previous `codegen` command.

```
matrixInput = coder.typeof(double(0),[3 Inf],[false true]);
cellInput = coder.typeof({matrixInput}, [5 1]);

codegen -config cfg lstmnet_predict -args {cellInput} -report

Code generation successful: View report


YPred3 = lstmnet_predict_mex(XValidate);
```

The output is a 5-by-1 cell array of predictions for the five observations passed in.

```
disp(YPred3)
```

```
    {5×53888 single}
    {5×64480 single}
    {5×53696 single}
    {5×56416 single}
    {5×50688 single}
```

Generate MEX with Stateful LSTM

Instead of passing the entire timeseries to predict in one step, we can run prediction on an input by streaming in one timestep at a time, making use of the function `predictAndUpdateState` This

function takes in an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account.

The entry-point function `lstmnet_predict_and_update.m` takes in a single-timestep input and processes the input using the `predictAndUpdateState` function. `predictAndUpdateState` outputs a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps one at a time, the resulting output is the same as if all timesteps were passed in as a single input.

```
type('lstmnet_predict_and_update.m')
```

```
function out = lstmnet_predict_and_update(in) %#codegen

% Copyright 2019-2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('lstmnet.mat');
end

% pass in input
[mynet, out] = predictAndUpdateState(mynet,in);
```

Run codegen on this new design file. Since we are taking in a single timestep each call, we specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[3 1]);
codegen -config cfg lstmnet_predict_and_update -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run the generated MEX on the first validation sample's first timestep.

```
firstSample = XValidate{1};
firstTimestep = firstSample(:,1);
YPredStateful = lstmnet_predict_and_update_mex(firstTimestep);
[~, maxIndex] = max(YPredStateful, [], 1);
predictedLabelsStateful1 = labels(maxIndex)
```

```
predictedLabelsStateful1 = categorical
     Sitting
```

Compare the output label with the ground truth.

```
YValidate{1}(1)
```

```
ans = categorical
     Sitting
```

Deep Learning Prediction on ARM Mali GPU

This example shows how to use the `cnncodegen` function to generate code for an image classification application that uses deep learning on ARM® Mali GPUs. The example uses the `MobileNet-v2` DAG network to perform image classification. The generated code takes advantage of the ARM Compute library for computer vision and machine learning.

Prerequisites

- ARM Mali GPU based hardware. For example, HiKey960 is one of the target platforms that contains a Mali GPU.
- ARM Compute Library on the target ARM hardware built for the Mali GPU.
- Open source Computer Vision Library (OpenCV v2.4.9) on the target ARM hardware.
- Environment variables for the compilers and libraries. Ensure that the `ARM_COMPUTE` and the `LD_LIBRARY_PATH` variables are set on the target platform. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).

Get Pretrained DAGNetwork

Load the pretrained `MobileNet-v2` network available in the `Deep Learning Toolbox Model for MobileNet-v2 Network`.

```
net = mobilenetv2
```

```
net =
  DAGNetwork with properties:

         Layers: [154×1 nnet.cnn.layer.Layer]
    Connections: [163×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_Logits'}
```

The network contains 155 layers including convolution, batch normalization, softmax, and the classification output layers. The `analyzeNetwork()` function displays an interactive plot of the network architecture and a table containing information about the network layers.

```
analyzeNetwork(net);
```

Generate Code

For deep learning on ARM targets, you generate code on the host development computer. To build and run the executable program, move the generated code to the ARM target platform. The target platform must have an ARM Mali GPU. For example, HiKey960 is one of the target platforms on which you can execute the code generated in this example.

Call the `cnncodegen` function, specifying the target library as `arm-compute-mali`.

```
cnncodegen(net,'targetlib','arm-compute-mali');
```

Copy Generated Files to the Target

Move the generated codegen folder and other required files from the host development computer to the target platform by using your preferred SCP (Secure Copy Protocol) or Secure Shell File Transfer Protocol (SSH) client.

For example, on the Linux® platform, to transfer the files to the HiKey960, use the scp command with the format:

```
system('sshpass -p [password] scp (sourcefile) [username]@[targetname]:~/');
```

```
system('sshpass -p password scp main_mobilenet_arm_generic.cpp username@targetname:~/');
system('sshpass -p password scp peppers_mobilenet.png username@targetname:~/');
system('sshpass -p password scp makefile_mobilenet_arm_generic.mk username@targetname:~/');
system('sshpass -p password scp synsetWords.txt username@targetname:~/');
system('sshpass -p password scp -r codegen username@targetname:~/');
```

On the Windows® platform, you can use the `pscp` tool that comes with a PuTTY installation. For example:

```
system('pscp -pw password-r codegen username@targetname:/home/username');
```

PSCP utilities must be either on your PATH or in your current folder.

Build Executable

To build the library on the target platform, use the generated makefile `cnnbuild_rtw.mk`.

For example, to build the library on the HiKey960:

```
system('sshpass -p password ssh username@targetname' ...
' "make -C /home/username/codegen -f cnnbuild_rtw.mk"');
```

On the Windows platform, you can use the `putty` command with `-ssh` argument to log in and run the make command. For example:

```
system('putty -ssh username@targetname -pw password');
```

To build and run the executable on the target platform, use the command with the format: `make -C /home/$(username)` and `./execfile -f makefile_mobilenet_arm_generic.mk`

For example, on the HiKey960:

```
make -C /home/usrname arm_mobilenet -f makefile_mobilenet_arm_generic.mk
```

Run the executable on the ARM platform specifying an input image file.

```
./mobilenet_exe peppers_mobilenet.png
```

The top five predictions for the input image file are:

```
Top 5 Predictions:
------------------------------
88.976% bell pepper
4.907% cucumber
1.390% grocery store
0.512% Granny Smith
0.256% lemon
```

Code Generation for Object Detection by Using YOLO v2

This example shows how to generate CUDA® MEX for a you only look once (YOLO) v2 object detector. A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. This example generates code for the network trained in the *Object Detection Using YOLO v2 Deep Learning* example from Computer Vision Toolbox™. For more information, see "Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox). You can modify this example to generate CUDA® MEX for the network imported in the *Import Pretrained ONNX YOLO v2 Object Detector* example from Computer Vision Toolbox™. For more information, see "Import Pretrained ONNX YOLO v2 Object Detector" (Computer Vision Toolbox).

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see "Third-Party Hardware" (GPU Coder) and "Setting Up the Prerequisite Products" (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAGNetwork

This example uses the `yolov2ResNet50VehicleExample` MAT-file containing the pretrained network. The file is approximately 98MB in size. Download the file from the MathWorks website.

```
matFile = matlab.internal.examples.downloadSupportFile('vision/data','yolov2ResNet50VehicleExamp
vehicleDetector = load(matFile);
net = vehicleDetector.detector.Network
```

```
net =
  DAGNetwork with properties:

         Layers: [150×1 nnet.cnn.layer.Layer]
    Connections: [162×2 table]
     InputNames: {'input_1'}
```

```
    OutputNames: {'yolov2OutputLayer'}
```

The DAG network contains 150 layers including convolution, ReLU, and batch normalization layers and the YOLO v2 transform and YOLO v2 output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(net);
```

The `yolov2_detect` Entry-Point Function

The `yolov2_detect.m` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `yolov2ResNet50VehicleExample.mat` file. The function loads the network object from the `yolov2ResNet50VehicleExample.mat` file into a persistent variable *yolov2Obj* and reuses the persistent object on subsequent detection calls.

```
type('yolov2_detect.m')
```

```
function outImg = yolov2_detect(in,matFile)

%   Copyright 2018-2021 The MathWorks, Inc.

persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork(matFile);
end

% Call to detect method
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);

% Convert categorical labels to cell array of charactor vectors
labels = cellstr(labels);

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
```

Run MEX Code Generation

To generate CUDA code for the entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of 224-by-224-by-3. This value corresponds to the input layer size of YOLOv2.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
inputArgs = {ones(224,224,3,'uint8'),coder.Constant(matFile)};

codegen -config cfg yolov2_detect -args inputArgs
```

```
Code generation successful: View report
```

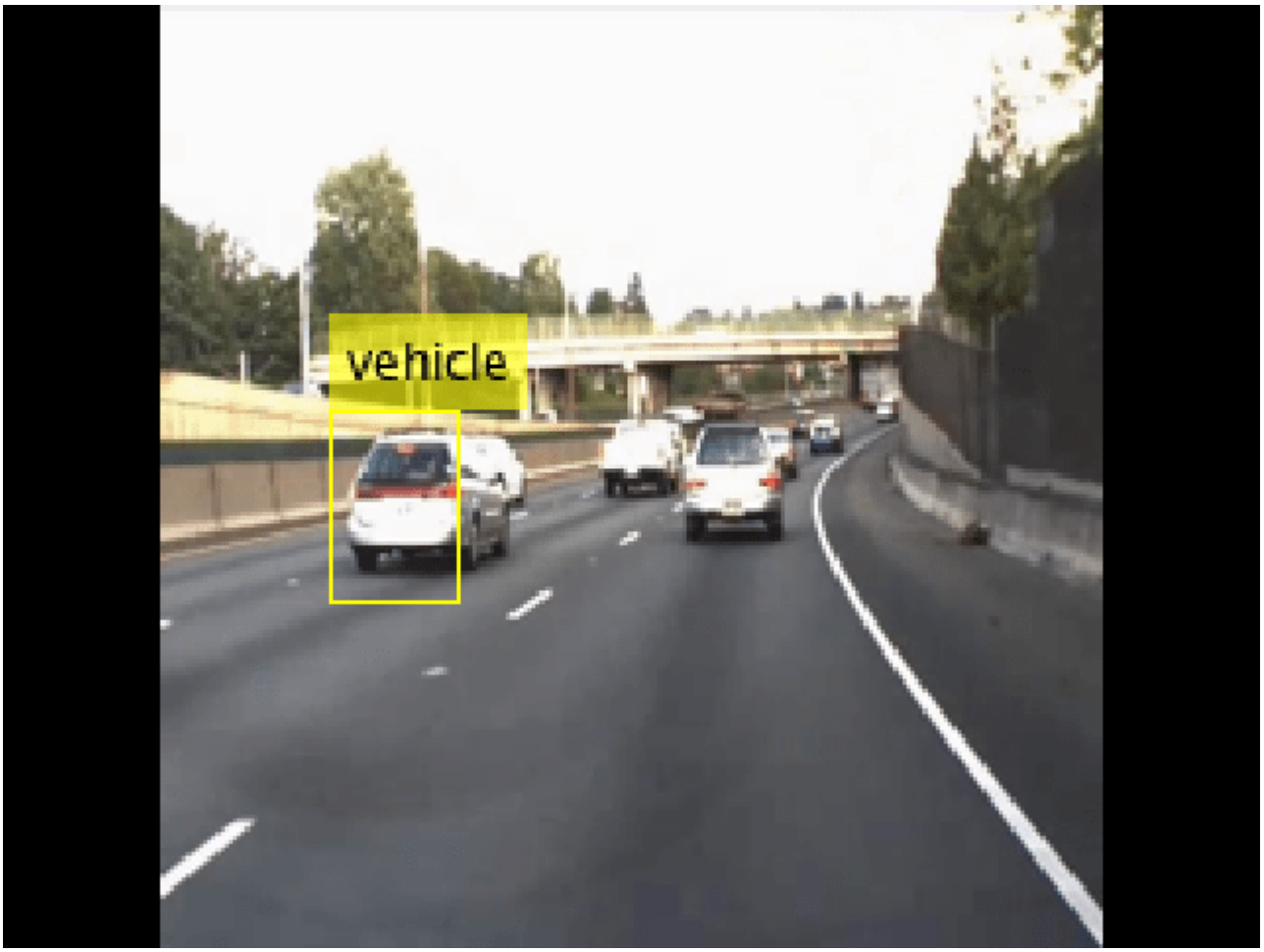
Run Generated MEX

Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[224,224]);
    out = yolov2_detect_mex(in,matFile);
    step(depVideoPlayer, out);
    % Exit the loop if the video player figure window is closed
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer);
end
```



References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Code Generation For Object Detection Using YOLO v3 Deep Learning

This example shows how to generate CUDA® MEX for a you only look once (YOLO) v3 object detector with custom layers. YOLO v3 improves upon YOLO v2 by adding detection at multiple scales to help detect smaller objects. Moreover, the loss function used for training is separated into mean squared error for bounding box regression and binary cross-entropy for object classification to help improve detection accuracy. The YOLO v3 network used in this example was trained from the *Object Detection Using YOLO v3 Deep Learning* example in the Computer Vision Toolbox (TM). For more information, see “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

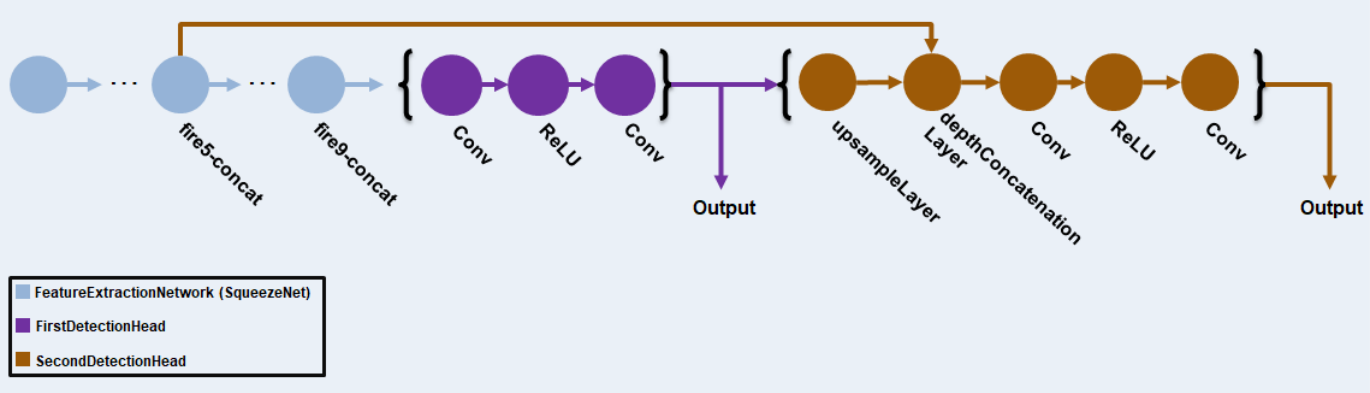
To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` (GPU Coder) function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

YOLO v3 Network

The YOLO v3 network in this example is based on `squeezenet`, and uses the feature extraction network in SqueezeNet with the addition of two detection heads at the end. The second detection head is twice the size of the first detection head, so it is better able to detect small objects. Note that any number of detection heads of different sizes can be specified based on the size of the objects to be detected. The YOLO v3 network uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the network learn to predict the boxes accurately. For information about anchor boxes, see “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

The YOLO v3 network in this example is illustrated in the following diagram.



Each detection head predicts the bounding box coordinates (x, y, width, height), object confidence, and class probabilities for the respective anchor box masks. Therefore, for each detection head, the number of output filters in the last convolution layer is the number of anchor box mask times the number of prediction elements per anchor box. The detection heads comprise the output layer of the network.

Pretrained YOLO v3 Network

This example uses the `yolov3SqueezeNetVehicleExample_21a.zip` file containing the pretrained YOLO v3 network. This file is approximately 9MB in size. Download the file from the MathWorks website, then unzip the file.

```
fileName = matlab.internal.examples.downloadSupportFile('vision/data/','yolov3SqueezeNetVehicleEx
unzip(fileName);
```

The YOLO v3 network used in this example was trained using the steps described in “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox).

```
matFile = 'yolov3SqueezeNetVehicleExample_21a.mat';
pretrained = load(matFile);
net = pretrained.net;
```

YOLO v3 network uses a `resize2dLayer` (Image Processing Toolbox) to resize the 2-D input image by replicating the neighboring pixel values by a scaling factor of 2. The resize2DLayer is implemented as a custom layer supported for code generation. For more information, see “Define Custom Deep Learning Layer for Code Generation” on page 18-142.

Note: You can also use the pretrained detector network availabe through the Computer Vision Toolbox™ Model for YOLO v3 Object Detection support package.

To use this pretrained network, you must first install the Computer Vision Toolbox Model for YOLO v3 Object Detection from the Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Then, save the network from the `yolov3ObjectDetector` object to a MAT-file and proceed. For example,

```
detector = yolov3ObjectDetector('darknet53-coco');
net = detector.Network;
matFile = 'pretrainedYOLOv3Detector.mat';
save(matFile,'net');
```

The `yolov3Detect` Entry-Point Function

The `yolov3Detect` entry-point function takes an input image and passes it to a trained network for prediction through the `yolov3Predict` function. The `yolov3Predict` function loads the network object from the MAT-file into a persistent variable and reuses the persistent object for subsequent prediction calls. Specifically, the function uses the `dlnetwork` representation of the network trained in the “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example. The predictions from the YOLO v3 grid cell coordinates obtained from the `yolov3Predict` calls are then converted to bounding box coordinates by using the supporting functions `generateTiledAnchors` and `applyAnchorBoxOffsets`.

```
type('yolov3Detect.m')
```

```
function [bboxes,scores,labelsIndex] = yolov3Detect(matFile, im,...
    networkInputSize, networkOutputs, confidenceThreshold,...
    overlapThreshold, classes)
% The yolov3Detect function detects the bounding boxes, scores, and
% labelsIndex in an image.
%#codegen

%   Copyright 2020-2021 The MathWorks, Inc.

%% Preprocess Data
% This example applies all the preprocessing transforms to the data set
% applied during training, except data augmentation. Because the example
% uses a pretrained YOLO v3 network, the input data must be representative
% of the original data and left unmodified for unbiased evaluation.

% Specifically the following preprocessing operations are applied to the
% input data.
%     1. Resize the images to the network input size, as the images are
%     bigger than networkInputSize. 2. Scale the image pixels in the range
%     [0 1]. 3. Convert the resized and rescaled image to a dlarray object.

im = dlarray(preprocessData(im, networkInputSize), "SSCB");
imageSize = size(im,[1,2]);

%% Define Anchor Boxes
% Specify the anchor boxes estimated on the basis of the preprocessed
% training data used when training the YOLO v3 network. These anchor box
% values are same as mentioned in "Object Detection Using YOLO v3 Deep
% Learning" example. For details on estimating anchor boxes, see "Anchor
% Boxes for Object Detection".

anchors = [
    41    34;
   163   130;
    98    93;
   144   125;
    33    24;
    69    66];

% Specify anchorBoxMasks to select anchor boxes to use in both the
% detection heads of the YOLO v3 network. anchorBoxMasks is a cell array of
% size M-by-1, where M denotes the number of detection heads. Each
% detection head consists of a 1-by-N array of row index of anchors in
% anchorBoxes, where N is the number of anchor boxes to use. Select anchor
```

```
% boxes for each detection head based on size-use larger anchor boxes at
% lower scale and smaller anchor boxes at higher scale. To do so, sort the
% anchor boxes with the larger anchor boxes first and assign the first
% three to the first detection head and the next three to the second
% detection head.

area = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxMasks = {[1,2,3],[4,5,6]};

%% Predict on Yolov3
% Predict and filter the detections based on confidence threshold.
predictions = yolov3Predict(matFile,im,networkOutputs,anchorBoxMasks);

%% Generate Detections
% indices corresponding to x,y,w,h predictions for bounding boxes
anchorIndex = 2:5;
tiledAnchors = generateTiledAnchors(predictions,anchors,anchorBoxMasks,...
    anchorIndex);
predictions = applyAnchorBoxOffsets(tiledAnchors, predictions,...
    networkInputSize, anchorIndex);
[bboxes,scores,labelsIndex] = generateYOLOv3DetectionsForCodegen(predictions,...
    confidenceThreshold, overlapThreshold, imageSize, classes);

end

function YPredCell = yolov3Predict(matFile,im,networkOutputs,anchorBoxMask)
% Predict the output of network and extract the confidence, x, y,
% width, height, and class.

% load the deep learning network for prediction
persistent net;

if isempty(net)
    net = coder.loadDeepLearningNetwork(matFile);
end

YPredictions = cell(coder.const(networkOutputs), 1);
[YPredictions{:}] = predict(net, im);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);

% Apply activation to the predicted cell array.
YPredCell = applyActivations(YPredCell);
end
```

Evaluate the Entry-Point Function for Object Detection

Follow these steps to evaluate the entry-point function on an image from the test data.

- Specify the confidence threshold as 0.5 to keep only detections with confidence scores above this value.
- Specify the overlap threshold as 0.5 to remove overlapping detections.
- Read an image from the input data.
- Use the entry-point function `yolov3Detect` to get the predicted bounding boxes, confidence scores, and class labels.

- Display the image with bounding boxes and confidence scores.

Define the desired thresholds.

```
confidenceThreshold = 0.5;
overlapThreshold = 0.5;
```

Specify the network input size of the trained network and the number of network outputs.

```
networkInputSize = [227 227 3];
networkOutputs = numel(net.OutputNames);
```

Read the example image data obtained from the labeled data set from the “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example. This image contains one instance of an object of type vehicle.

```
I = imread('vehicleImage.jpg');
```

Specify the class names.

```
classNames = {'vehicle'};
```

Invoke the detect method on YOLO v3 network and display the results.

```
[bboxes,scores,labelsIndex] = yolov3Detect(matFile,I,...
networkInputSize,networkOutputs,confidenceThreshold,overlapThreshold,classNames);
labels = classNames(labelsIndex);

% Display the detections on the image
IAnnotated = insertObjectAnnotation(I,'rectangle',bboxes,strcat(labels,{' - '},num2str(scores)));
figure
imshow(IAnnotated)
```

Generate CUDA MEX

To generate CUDA® code for the `yolov3Detect` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the

`coder.DeepLearningConfig` (GPU Coder) function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig(TargetLibrary='cudnn');

args = {coder.Constant(matFile),I,coder.Constant(networkInputSize),...
coder.Constant(networkOutputs),confidenceThreshold,...
overlapThreshold,classNames};

codegen -config cfg yolov3Detect -args args -report

Code generation successful: View report
```

To generate CUDA® code for TensorRT target create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object. Similarly, to generate code for MKLDNN target, create a CPU code configuration object and use MKLDNN deep learning configuration object as its `DeepLearningConfig` property.

Run the Generated MEX

Call the generated CUDA MEX with the same image input `I` as before and display the results.

```
[bboxes,scores,labelsIndex] = yolov3Detect_mex(matFile,I,...
networkInputSize,networkOutputs,confidenceThreshold,...
overlapThreshold,classNames);
labels = classNames(labelsIndex);

figure;
IAnnotated = insertObjectAnnotation(I,'rectangle',bboxes,strcat(labels,{' - '},num2str(scores)));
imshow(IAnnotated);
```

Utility Functions

The utillity functions listed below are based on the ones used in “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example and modified to make the utility functions suitable for code generation.

```
type('applyActivations.m')
```

```
function YPredCell = applyActivations(YPredCell)    %#codegen

%   Copyright 2020-2021 The MathWorks, Inc.


numCells = size(YPredCell, 1);
for iCell = 1:numCells
    for idx = 1:3
        YPredCell{iCell, idx} = sigmoidActivation(YPredCell{iCell,idx});
    end
end
for iCell = 1:numCells
    for idx = 4:5
        YPredCell{iCell, idx} = exp(YPredCell{iCell, idx});
    end
end
for iCell = 1:numCells
    YPredCell{iCell, 6} = sigmoidActivation(YPredCell{iCell, 6});
end
end

function out = sigmoidActivation(x)
out = 1./(1+exp(-x));
end
```

```
type('extractPredictions.m')
```

```
function predictions = extractPredictions(YPredictions, anchorBoxMask)
%#codegen

%   Copyright 2020-2021 The MathWorks, Inc.

numPredictionHeads = size(YPredictions, 1);
predictions = cell(numPredictionHeads,6);
for ii = 1:numPredictionHeads
    % Get the required info on feature size.
    numChannelsPred = size(YPredictions{ii},3);
    numAnchors = size(anchorBoxMask{ii},2);
    numPredElemsPerAnchors = numChannelsPred/numAnchors;
    allIds = (1:numChannelsPred);

    stride = numPredElemsPerAnchors;
    endIdx = numChannelsPred;

    YPredictionsData = extractdata(YPredictions{ii});

    % X positions.
    startIdx = 1;
    predictions{ii,2} = YPredictionsData(:,:,startIdx:stride:endIdx,:);
    xIds = startIdx:stride:endIdx;
```

```
    % Y positions.
    startIdx = 2;
    predictions{ii,3} = YPredictionsData(:,:,startIdx:stride:endIdx,:);
    yIds = startIdx:stride:endIdx;

    % Width.
    startIdx = 3;
    predictions{ii,4} = YPredictionsData(:,:,startIdx:stride:endIdx,:);
    wIds = startIdx:stride:endIdx;

    % Height.
    startIdx = 4;
    predictions{ii,5} = YPredictionsData(:,:,startIdx:stride:endIdx,:);
    hIds = startIdx:stride:endIdx;

    % Confidence scores.
    startIdx = 5;
    predictions{ii,1} = YPredictionsData(:,:,startIdx:stride:endIdx,:);
    confIds = startIdx:stride:endIdx;

    % Accumulate all the non-class indexes
    nonClassIds = [xIds yIds wIds hIds confIds];

    % Class probabilities.
    % Get the indexes which do not belong to the nonClassIds
    classIdx = setdiff(allIds, nonClassIds, 'stable');
    predictions{ii,6} = YPredictionsData(:,:,classIdx,:);
end
end
```

```
type('generateTiledAnchors.m')
```

```
function tiledAnchors = generateTiledAnchors(YPredCell,anchorBoxes,...
    anchorBoxMask,anchorIndex)
% Generate tiled anchor offset for converting the predictions from the YOLO
% v3 grid cell coordinates to bounding box coordinates
%#codegen

%   Copyright 2020-2021 The MathWorks, Inc.

numPredictionHeads = size(YPredCell,1);
tiledAnchors = cell(numPredictionHeads, size(anchorIndex, 2));
for i = 1:numPredictionHeads
    anchors = anchorBoxes(anchorBoxMask{i}, :);
    [h,w,~,n] = size(YPredCell{i,1});
    [tiledAnchors{i,2},tiledAnchors{i,1}] = ndgrid(0:h-1,0:w-1,...
        1:size(anchors,1),1:n);
    [~,~,tiledAnchors{i,3}] = ndgrid(0:h-1,0:w-1,anchors(:,2),1:n);
    [~,~,tiledAnchors{i,4}] = ndgrid(0:h-1,0:w-1,anchors(:,1),1:n);
end
end
```

```
type('applyAnchorBoxOffsets.m')
```

```
function YPredCell = applyAnchorBoxOffsets(tiledAnchors,YPredCell,...
    inputImageSize,anchorIndex)   %#codegen
% Convert the predictions from the YOLO v3 grid cell coordinates to
% bounding box coordinates
```

```
%   Copyright 2020-2021 The MathWorks, Inc.

for i = 1:size(YPredCell,1)
    [h,w,~,~] = size(YPredCell{i,1});
    YPredCell{i,anchorIndex(1)} = (tiledAnchors{i,1}+...
        YPredCell{i,anchorIndex(1)})./w;
    YPredCell{i,anchorIndex(2)} = (tiledAnchors{i,2}+...
        YPredCell{i,anchorIndex(2)})./h;
    YPredCell{i,anchorIndex(3)} = (tiledAnchors{i,3}.*...
        YPredCell{i,anchorIndex(3)})./inputImageSize(2);
    YPredCell{i,anchorIndex(4)} = (tiledAnchors{i,4}.*...
        YPredCell{i,anchorIndex(4)})./inputImageSize(1);
end
end

type('preprocessData.m')

function image = preprocessData(image, targetSize)
% Resize the images and scale the pixels to between 0 and 1.
%#codegen

%   Copyright 2020-2021 The MathWorks, Inc.


imgSize = size(image);

% Convert an input image with single channel to 3 channels.
if numel(imgSize) < 1
    image = repmat(image,1,1,3);
end

image = im2single(rescale(image));

image = iLetterBoxImage(image,coder.const(targetSize(1:2)));

end


function Inew = iLetterBoxImage(I,targetSize)
% LetterBoxImage returns a resized image by preserving the width and height
% aspect ratio of input Image I. 'targetSize' is a 1-by-2 vector consisting
% the target dimension.
%
% Input I can be uint8, uint16, int16, double, single, or logical, and must
% be real and non-sparse.

[Irow,Icol,Ichannels] = size(I);

% Compute aspect Ratio.
arI = Irow./Icol;

% Preserve the maximum dimension based on the aspect ratio.
if arI<1
    IcolFin = targetSize(1,2);
    IrowFin = floor(IcolFin.*arI);
else
    IrowFin = targetSize(1,1);
```

```
    IcolFin = floor(IrowFin./arI);
end

% Resize the input image.
Itmp = imresize(I,[IrowFin,IcolFin]);

% Initialize Inew with gray values.
Inew = ones([targetSize,Ichannels],'like',I).*0.5;

% Compute the offset.
if arI<1
    buff = targetSize(1,1)-IrowFin;
else
    buff = targetSize(1,2)-IcolFin;
end

% Place the resized image on the canvas image.
if (buff==0)
    Inew = Itmp;
else
    buffVal = floor(buff/2);
    if arI<1
        Inew(buffVal:buffVal+IrowFin-1,:,:) = Itmp;
    else
        Inew(:,buffVal:buffVal+IcolFin-1,:) = Itmp;
    end
end

end
```

References

1. Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” Preprint, submitted April 8, 2018. https://arxiv.org/abs/1804.02767.

Deep Learning Prediction by Using NVIDIA TensorRT

This example shows code generation for a deep learning application by using the NVIDIA TensorRT™ library. It uses the `codegen` command to generate a MEX file to perform prediction with a ResNet-50 image classification network by using TensorRT. A second example demonstrates usage of `codegen` command to generate a MEX file that performs 8-bit integer prediction by using TensorRT for a logo classification network.

Third-Party Prerequisites

Required

This example generates CUDA® MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN and TensorRT library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'tensorrt';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

The `resnet_predict` Entry-Point Function

This example uses the DAG network ResNet-50 to show image classification by using TensorRT. A pretrained ResNet-50 model for MATLAB® is available in the ResNet-50 support package of Deep Learning Toolbox. To download and install the support package, use the Add-On Explorer.

The `resnet_predict.m` function loads the ResNet-50 network into a persistent network object and reuses the persistent object on subsequent prediction calls.

```
type('resnet_predict.m')
```

```
% Copyright 2020 The MathWorks, Inc.

function out = resnet_predict(in)
%#codegen

% A persistent object mynet is used to load the series network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
```

```
% reused to call predict on inputs, avoiding reconstructing and reloading
% the network object.

persistent mynet;

if isempty(mynet)
    % Call the function resnet50 that returns a DAG network
    % for ResNet-50 model.
    mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
end

% pass in input
out = mynet.predict(in);
```

Run MEX Code Generation

To generate CUDA code for the `resnet_predict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `TensorRT` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [224,224,3]. This value corresponds to the input layer size of ResNet-50 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
codegen -config cfg resnet_predict -args {ones(224,224,3)} -report

Code generation successful: View report
```

Perform Prediction on Test Image

```
im = imread('peppers.png');
im = imresize(im, [224,224]);
predict_scores = resnet_predict_mex(double(im));
%
% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
net = resnet50;
classnames = net.Layers(end).ClassNames;
labels = classnames(indx(1:5));
```

Clear the static network object that was loaded in memory.

```
clear mex;
```

Generate TensorRT Code for INT8 Prediction

Generate TensorRT code that runs inference in int8 precision. Use a pretrained logo classification network to classify logos in images. Download the pretrained `LogoNet` network and save it as a `logonet.mat` file. The network was developed in MATLAB. This network can recognize 32 logos under various lighting conditions and camera angles. The network is pretrained in single precision floating-point format.

```
net = getLogonet();
```

Code generation by using the NVIDIA TensorRT Library with inference computation in 8-bit integer precision supports these additional networks:

- Object detector networks such as YOLOv2 and SSD.
- Regression and semantic segmentation networks.

TensorRT requires a calibration data set to calibrate a network that is trained in floating-point to compute inference in 8-bit integer precision. Set the data type to int8 and the path to the calibration data set by using the `DeepLearningConfig`. `logos_dataset` is a subfolder containing images grouped by their corresponding classification labels. For int8 support, GPU compute capability must be 6.1 or higher.

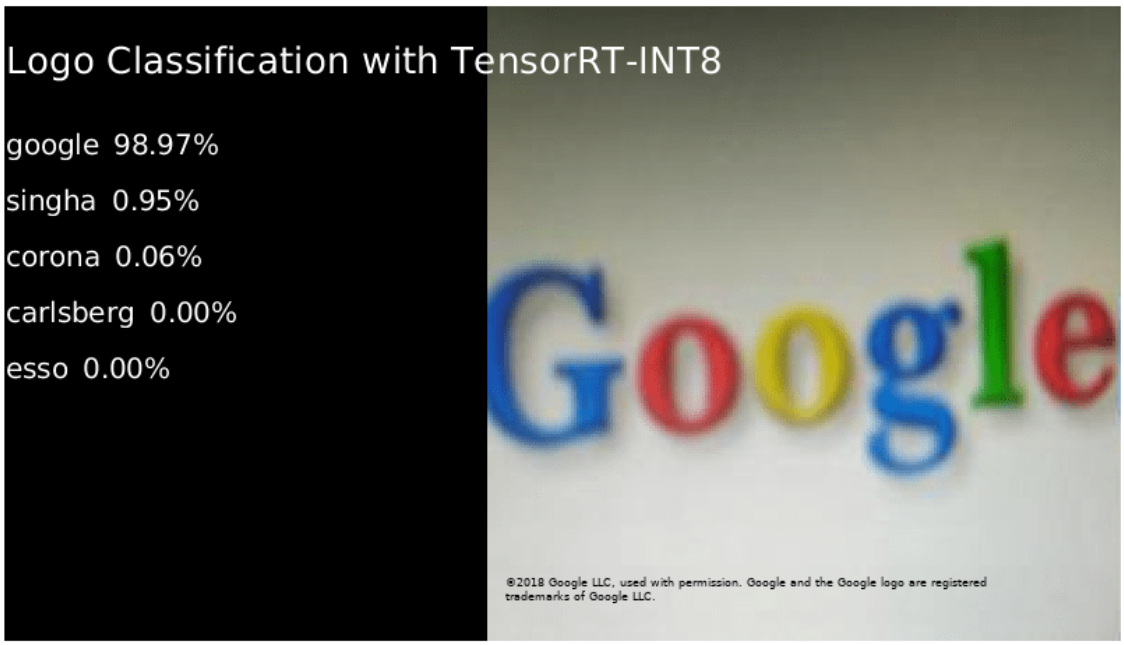
Note: For semantic segmentation networks, the calibration data images must be of a format supported by the `imread` function.

```
unzip('logos_dataset.zip');
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.GpuConfig.ComputeCapability = '6.1';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.DataPath = 'logos_dataset';
cfg.DeepLearningConfig.NumCalibrationBatches = 50;
codegen -config cfg logonet_predict -args {ones(227,227,3,'int8')} -report
```

```
Code generation successful: View report
```

Run INT8 Prediction on Test Image

```
im = imread('gpucoder_tensorrt_test.png');
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(int8(im));
%
% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
classnames = net.Layers(end).ClassNames;
labels = classnames(indx(1:5));
```

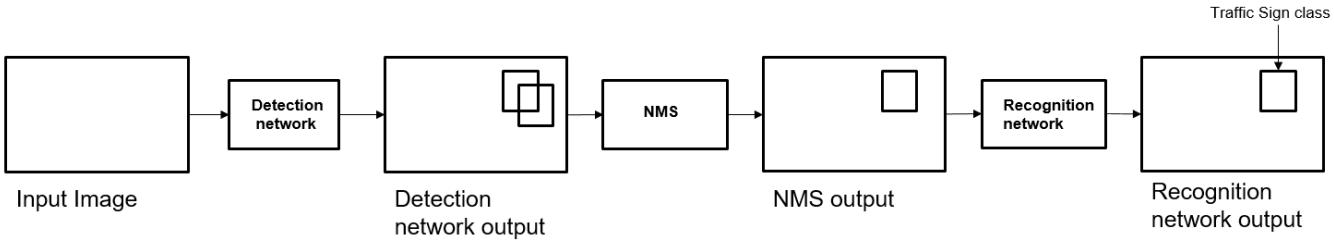


Clear the static network object that was loaded in memory.

```
clear mex;
```

Traffic Sign Detection and Recognition

This example shows how to generate CUDA® MEX code for a traffic sign detection and recognition application that uses deep learning. Traffic sign detection and recognition is an important application for driver assistance systems, aiding and providing information to the driver about road signs.



In this traffic sign detection and recognition example you perform three steps - detection, Non-Maximal Suppression (NMS), and recognition. First, the example detects the traffic signs on an input image by using an object detection network that is a variant of the You Only Look Once (YOLO) network. Then, overlapping detections are suppressed by using the NMS algorithm. Finally, the recognition network classifies the detected traffic signs.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Detection and Recognition Networks

The detection network is trained in the Darknet framework and imported into MATLAB® for inference. Because the size of the traffic sign is relatively small with respect to that of the image and the number of training samples per class are fewer in the training data, all the traffic signs are considered as a single class for training the detection network.

The detection network divides the input image into a 7-by-7 grid. Each grid cell detects a traffic sign if the center of the traffic sign falls within the grid cell. Each cell predicts two bounding boxes and confidence scores for these bounding boxes. Confidence scores indicate whether the box contains an object or not. Each cell predicts on probability for finding the traffic sign in the grid cell. The final score is product of the preceding scores. You apply a threshold of 0.2 on this final score to select the detections.

The recognition network is trained on the same images by using MATLAB.

The trainRecognitionnet.m helper script shows the recognition network training.

Get the Pretrained SeriesNetwork

Download the detection and recognition networks.

```
getTsdr();
```

The detection network contains 58 layers including convolution, leaky ReLU, and fully connected layers.

```
load('yolo_tsr.mat');
yolo
```

```
yolo =
  SeriesNetwork with properties:

         Layers: [58×1 nnet.cnn.layer.Layer]
     InputNames: {'input'}
    OutputNames: {'classoutput'}
```

To view the network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(yolo)
```

The recognition network contains 14 layers including convolution, fully connected, and the classification output layers.

```
load('RecognitionNet.mat');
convnet
```

```
convnet =
  SeriesNetwork with properties:

         Layers: [14×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

The `tsdr_predict` Entry-Point Function

The `tsdr_predict.m` entry-point function takes an image input and detects the traffic signs in the image by using the detection network. The function suppresses the overlapping detections (NMS) by using `selectStrongestBbox` and recognizes the traffic sign by using the recognition network. The function loads the network objects from `yolo_tsr.mat` into a persistent variable *detectionnet* and the `RecognitionNet.mat` into a persistent variable *recognitionnet*. The function reuses the persistent objects on subsequent calls.

```
type('tsdr_predict.m')
```

```
function [selectedBbox,idx] = tsdr_predict(img)
%#codegen

% This function detects the traffic signs in the image using Detection Network
% (modified version of Yolo) and recognizes(classifies) using Recognition Network
%
% Inputs :
%
% im            : Input test image
%
% Outputs :
%
% selectedBbox  : Detected bounding boxes
% idx           : Corresponding classes

% Copyright 2017-2021 The MathWorks, Inc.

coder.gpu.kernelfun;

% resize the image
img_rz = imresize(img,[448,448]);

% Converting into BGR format
img_rz = img_rz(:,:,3:-1:1);
img_rz = im2single(img_rz);

%% TSD
persistent detectionnet;
if isempty(detectionnet)
    detectionnet = coder.loadDeepLearningNetwork('yolo_tsr.mat','Detection');
end

predictions = detectionnet.activations(img_rz,56,'OutputAs','channels');


%% Convert predictions to bounding box attributes
classes = 1;
num = 2;
side = 7;
thresh = 0.2;
[h,w,~] = size(img);


boxes = single(zeros(0,4));
probs = single(zeros(0,1));
for i = 0:(side*side)-1
    for n = 0:num-1
```

```
            p_index = side*side*classes + i*num + n + 1;
            scale = predictions(p_index);
            prob = zeros(1,classes+1);
            for j = 0:classes
                class_index = i*classes + 1;
                tempProb = scale*predictions(class_index+j);
                if tempProb > thresh

                    row = floor(i / side);
                    col = mod(i,side);

                    box_index = side*side*(classes + num) + (i*num + n)*4 + 1;
                    bxX = (predictions(box_index + 0) + col) / side;
                    bxY = (predictions(box_index + 1) + row) / side;

                    bxW = (predictions(box_index + 2)^2);
                    bxH = (predictions(box_index + 3)^2);

                    prob(j+1) = tempProb;
                    probs = [probs;tempProb];

                    boxX = (bxX-bxW/2)*w+1;
                    boxY = (bxY-bxH/2)*h+1;
                    boxW = bxW*w;
                    boxH = bxH*h;
                    boxes = [boxes; boxX,boxY,boxW,boxH];
                end
            end
        end
    end

    %% Run Non-Maximal Suppression on the detected bounding boxess
    coder.varsize('selectedBbox',[98, 4],[1 0]);
    [selectedBbox,~] = selectStrongestBbox(round(boxes),probs);

    %% Recognition

    persistent recognitionnet;
    if isempty(recognitionnet)
        recognitionnet = coder.loadDeepLearningNetwork('RecognitionNet.mat','Recognition');
    end

    idx = zeros(size(selectedBbox,1),1);
    inpImg = coder.nullcopy(zeros(48,48,3,size(selectedBbox,1)));
    for i = 1:size(selectedBbox,1)

        ymin = selectedBbox(i,2);
        ymax = ymin+selectedBbox(i,4);
        xmin = selectedBbox(i,1);
        xmax = xmin+selectedBbox(i,3);


        % Resize Image
        inpImg(:,:,:,i) = imresize(img(ymin:ymax,xmin:xmax,:),[48,48]);
    end

    for i = 1:size(selectedBbox,1)
        output = recognitionnet.predict(inpImg(:,:,:,i));
```

```
    [~,idx(i)]=max(output);
end
```

Generate CUDA MEX for the `tsdr_predict` Function

Create a GPU configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. To generate CUDA MEX, use the `codegen` command and specify the input to be of size [480,704,3]. This value corresponds to the input image size of the `tsdr_predict` function.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg tsdr_predict -args {ones(480,704,3,'uint8')} -report
```

```
Code generation successful: View report
```

To generate code by using TensorRT, pass `coder.DeepLearningConfig('tensorrt')` as an option to the coder configuration object instead of `'cudnn'`.

Run Generated MEX

Load an input image.

```
im = imread('stop.jpg');
imshow(im);
```

Call `tsdr_predict_mex` on the input image.

```
im = imresize(im, [480,704]);
[bboxes,classes] = tsdr_predict_mex(im);
```

Map the class numbers to traffic sign names in the class dictionary.

```
classNames = {...
    'addedLane','slow','dip','speedLimit25','speedLimit35','speedLimit40',...
    'speedLimit45','speedLimit50','speedLimit55','speedLimit65',...
    'speedLimitUrdbl','doNotPass','intersection','keepRight','laneEnds',...
    'merge','noLeftTurn','noRightTurn','stop','pedestrianCrossing',...
    'stopAhead','rampSpeedAdvisory20','rampSpeedAdvisory45',...
    'truckSpeedLimit55','rampSpeedAdvisory50','turnLeft',...
    'rampSpeedAdvisoryUrdbl','turnRight','rightLaneMustTurn','yield',...
    'yieldAhead','school','schoolSpeedLimit25','zoneAhead45','signalAhead'};

classRec = classNames(classes);
```

Display the detected traffic signs.

```
outputImage = insertShape(im,'Rectangle',bboxes,'LineWidth',3);

for i = 1:size(bboxes,1)
    outputImage = insertText(outputImage,[bboxes(i,1)+ ...
        bboxes(i,3) bboxes(i,2)-20],classRec{i},'FontSize',20,...
        'TextColor','red');
end

imshow(outputImage);
```

Traffic Sign Detection and Recognition on a Video

The included helper file `tsdr_testVideo.m` grabs frames from the test video, performs traffic sign detection and recognition, and plots the results on each frame of the test video.

```
type tsdr_testVideo
```

```
function tsdr_testVideo

% Copyright 2017-2021 The MathWorks, Inc.

% Input video
v = VideoReader('stop.avi');


%% Integrated codegeneration for Traffic Sign Detection and Recognition

% Generate MEX
cfg = coder.config('mex');
cfg.GpuConfig = coder.gpu.config;
cfg.GpuConfig.Enabled = true;

cfg.GenerateReport = false;
cfg.TargetLang = 'C++';

% Create a GPU Configuration object for MEX target setting target language
% to C++. Run the |codegen| command specifying an input of input video
```

```
% frame size. This corresponds to the input image size of tsdr_predict
% function.
codegen -config cfg tsdr_predict -args {ones(480,704,3  ,'uint8')}

fps = 0;

while hasFrame(v)
    % Take a frame
    picture = readFrame(v);
    picture = imresize(picture,[480,704]);
    % Call MEX function for Traffic Sign Detection and Recognition
    tic;
    [bboxes,clases] = tsdr_predict_mex(picture);
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % display

        diplayDetections(picture,bboxes,clases,fps);
end


end

function diplayDetections(im,boundingBoxes,classIndices,fps)
% Function for inserting the detected bounding boxes and recognized classes
% and displaying the result
%
% Inputs :
%
% im            : Input test image
% boundingBoxes : Detected bounding boxes
% classIndices  : Corresponding classes
%

% Traffic Signs (35)
classNames = {'addedLane','slow','dip','speedLimit25','speedLimit35',...
    'speedLimit40','speedLimit45','speedLimit50','speedLimit55',...
    'speedLimit65','speedLimitUrdbl','doNotPass','intersection',...
    'keepRight','laneEnds','merge','noLeftTurn','noRightTurn','stop',...
    'pedestrianCrossing','stopAhead','rampSpeedAdvisory20',...
    'rampSpeedAdvisory45','truckSpeedLimit55','rampSpeedAdvisory50',...
    'turnLeft','rampSpeedAdvisoryUrdbl','turnRight','rightLaneMustTurn',...
    'yield','yieldAhead','school','schoolSpeedLimit25','zoneAhead45',...
    'signalAhead'};

outputImage = insertShape(im,'Rectangle',boundingBoxes,'LineWidth',3);

for i = 1:size(boundingBoxes,1)

     ymin = boundingBoxes(i,2);
     xmin = boundingBoxes(i,1);
     xmax = xmin+boundingBoxes(i,3);

    % inserting class as text at YOLO detection
    classRec = classNames{classIndices(i)};
```

```
    outputImage = insertText(outputImage,[xmax ymin-20],classRec,...
        'FontSize',20,'TextColor','red');

end
outputImage = insertText(outputImage,...
    round(([size(outputImage,1) 40]/2)-20),...
    ['Frame Rate: ',num2str(fps)],'FontSize',20,'TextColor','red');
imshow(outputImage);
end
```

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Logo Recognition Network

This example shows code generation for a logo classification application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction on a `SeriesNetwork` object called LogoNet.

Third-Party Prerequisites

Required

This example generates CUDA® MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

The Logo Recognition Network

Logos assist users in brand identification and recognition. Many companies incorporate their logos in advertising, documentation materials, and promotions. The logo recognition network (logonet) was developed in MATLAB® and can recognize 32 logos under various lighting conditions and camera motions. Because this network focuses only on recognition, you can use it in applications where localization is not required.

Training the Network

The network is trained in MATLAB by using training data that contains around 200 images for each logo. Because the number of images for training the network is small, data augmentation increases the number of training samples. Four types of data augmentation are used: contrast normalization, Gaussian blur, random flipping, and shearing. This data augmentation helps in recognizing logos in images captured by different lighting conditions and camera motions. The input size for logonet is 227-by-227-by-3. Standard SGDM trains by using a learning rate of 0.0001 for 40 epochs with a mini-batch size of 45. The trainLogonet.m helper script demonstrates the data augmentation on a sample image, architecture of the logonet, and training options.

Get Pretrained SeriesNetwork

Download the logonet network and save it to `LogoNet.mat`.

```
getLogonet();
```

The saved network contains 22 layers including convolution, fully connected, and the classification output layers.

```
load('LogoNet.mat');
convnet
```

```
convnet = SeriesNetwork with properties:
         Layers: [22×1 nnet.cnn.layer.Layer]
     InputNames: {'imageinput'}
    OutputNames: {'classoutput'}
```

To view the network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(convnet)
```

The `logonet_predict` Entry-Point Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image using the deep learning network saved in the `LogoNet.mat` file. The function loads the network object from `LogoNet.mat` into a persistent variable *logonet* and reuses the persistent variable on subsequent prediction calls.

```
type('logonet_predict.m')
```

```
function out = logonet_predict(in)
%#codegen

% Copyright 2017-2020 The MathWorks, Inc.

persistent logonet;

if isempty(logonet)

    logonet = coder.loadDeepLearningNetwork('LogoNet.mat','logonet');
end

out = logonet.predict(in);

end
```

Generate CUDA MEX for the `logonet_predict` Function

Create a GPU configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. To generate CUDA MEX, use the `codegen` command and specify the input to be of size [227,227,3]. This value corresponds to the input layer size of the logonet network.

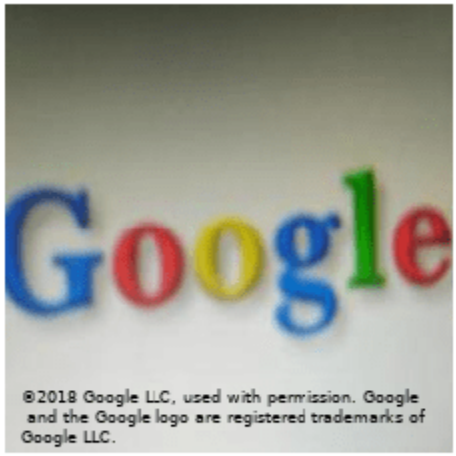
```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg logonet_predict -args {ones(227,227,3,'uint8')} -report
```

```
Code generation successful: View report
```

Run Generated MEX

Load an input image. Call `logonet_predict_mex` on the input image.

```
im = imread('test.png');
imshow(im);
```




```
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(im);
```

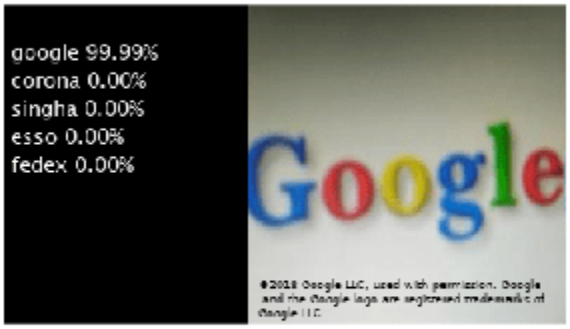
Map the top five prediction scores to words in the Wordnet dictionary synset (logos).

```
synsetOut = {'adidas', 'aldi', 'apple', 'becks', 'bmw', 'carlsberg', ...
    'chimay', 'cocacola', 'corona', 'dhl', 'erdinger', 'esso', 'fedex',...
    'ferrari', 'ford', 'fosters', 'google', 'guinness', 'heineken', 'hp',...
    'milka', 'nvidia', 'paulaner', 'pepsi', 'rittersport', 'shell',...
    'singha', 'starbucks', 'stellaartois', 'texaco', 'tsingtao', 'ups'};

[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
top5labels = synsetOut(indx(1:5));
```

Display the top five classification labels.

```
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:,:,k);
end

scol = 1;
srow = 20;

for k = 1:5
    outputImage = insertText(outputImage, [scol, srow],...
```

```
        [top5labels{k},' ',num2str(scores(k),'%2.2f'),'%'],...
        'TextColor', 'w','FontSize',15, 'BoxColor', 'black');
    srow = srow + 20;
end

 imshow(outputImage);
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Code Generation for Denoising Deep Neural Network

This example shows how to generate CUDA® MEX from MATLAB® code and denoise grayscale images by using the denoising convolutional neural network (DnCNN [1]). You can use the denoising network to estimate noise in a noisy image, and then remove it to obtain a denoised image.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

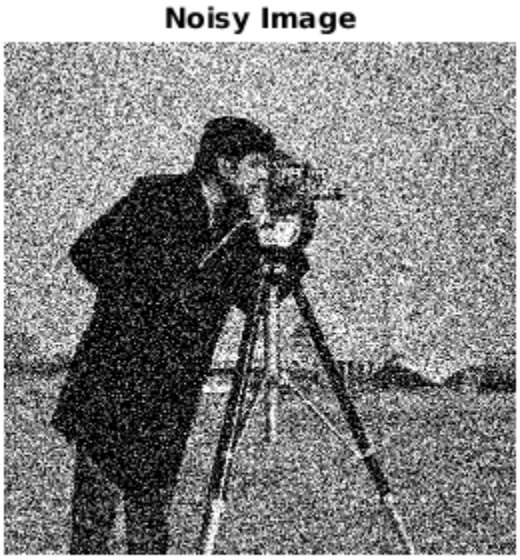
Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Load Noisy Image

Load a noisy grayscale image into the workspace and display the image.

```
noisyI = imread('noisy_cameraman.png');
figure
imshow(noisyI);
title('Noisy Image');
```



Get Pretrained Denoising Network

Call the `getDenoisingNetwork` helper function to get a pretrained image denoising deep neural network.

```
net = getDenoisingNetwork;
```

The `getDenoisingNetwork` function returns a pretrained DnCNN [1] that you can use to detect additive white Gaussian noise (AWGN) that has unknown levels. The network is a feed-forward denoising convolutional network that implements a residual learning technique to predict a residual image. In other words, DnCNN [1] computes the difference between a noisy image and the latent clean image.

The network contains 59 layers including convolution, batch normalization, and regression output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(net);
```

The `denoisenet_predict` Function

The `denoisenet_predict` entry-point function takes a noisy image input and returns a denoised image by using a pretrained denoising network.

The function loads the network object returned by `getDenoisingNetwork` into a persistent variable *mynet* and reuses the persistent object on subsequent prediction calls.

```
type denoisenet_predict
```

```
function I = denoisenet_predict(in)
%#codegen
% Copyright 2018-2021 The MathWorks, Inc.
```

```
persistent mynet;


if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('getDenoisingNetwork', 'DnCNN');
end

% The activations methods extracts the output from the last layer. The
% 'OutputAs' 'channels' name-value pair argument is used inorder to call
% activations on an image whose input dimensions are greater than or equal
% to the network's imageInputLayer.InputSize.

res = mynet.activations(in, 59,'OutputAs','channels');


% Once the noise is estimated, we subtract the noise from the original
% image to obtain a denoised image.

I = in - res;
```

Here, the `activations` method is called with the layer numeric index as 59 to extract the activations from the final layer of the network. The `'OutputAs' 'channels'` name-value pair argument computes activations on images larger than the `imageInputLayer.InputSize` of the network.

The `activations` method returns an estimate of the noise in the input image by using the pretrained denoising image.

Once the noise is estimated, subtract the noise from the original image to obtain a denoised image.

Run MEX Code Generation

To generate CUDA code for the `denoisenet_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [256,256]. This value corresponds to the size of the noisy image that you intend to denoise.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg denoisenet_predict -args {ones(256,256,'single')} -report

Code generation successful: View report
```

Run Generated MEX

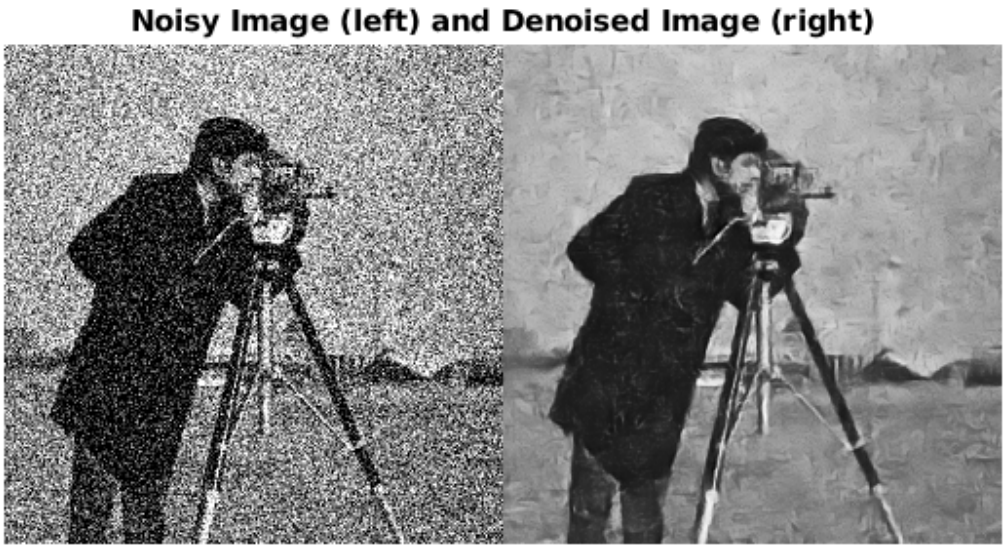
The DnCNN [1] is trained on input images having an input range [0,1]. Call the `im2single` (Image Processing Toolbox) function on `noisyI` to rescale the values from [0,255] to [0,1].

Call `denoisenet_predict_predict` on the rescaled input image.

```
denoisedI = denoisenet_predict_mex(im2single(noisyI));
```

View Denoised Image

```
figure
imshowpair(noisyI,denoisedI,'montage');
title('Noisy Image (left) and Denoised Image (right)');
```



References

[1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang. "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." IEEE Transactions on Image Processing. Vol. 26, Number 7, Feb. 2017, pp. 3142-3155.

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Train and Deploy Fully Convolutional Networks for Semantic Segmentation

This example shows how to train and deploy a fully convolutional semantic segmentation network on an NVIDIA® GPU by using GPU Coder™.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

To illustrate the training procedure, this example trains FCN-8s [1], one type of convolutional neural network (CNN) designed for semantic image segmentation. Other types of networks for semantic segmentation include fully convolutional networks, such as SegNet and U-Net. You can apply this training procedure to those networks too.

This example uses the CamVid dataset [2] from the University of Cambridge for training. This data set is a collection of images containing street-level views obtained while driving. The data set provides pixel-level labels for 32 semantic classes including car, pedestrian, and road.

Third-party Prerequisites

Required

- CUDA® enabled NVIDIA GPU and compatible driver.

Optional

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Setup

This example creates the fully convolutional semantic segmentation network with weights initialized from the VGG-16 network. The vgg16 function checks for the existence of the Deep Learning Toolbox Model for VGG-16 Network support package and returns a pretrained VGG-16 model.

```
vgg16();
```

Download a pretrained version of FCN. This pretrained model enables you to run the entire example without waiting for the training to complete. The doTraining flag controls whether the example uses the trained network of the example or the pretrained FCN network for code generation.

```
doTraining = false;
if ~doTraining
    pretrainedURL = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/fcn/FCN8sCamVid.m
    disp('Downloading pretrained FCN (448 MB)...');
    websave('FCN8sCamVid.mat',pretrainedURL);
end
```

```
Downloading pretrained FCN (448 MB)...
```

Download CamVid Dataset

Download the CamVid dataset from these URLs.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.zi
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.z

outputFolder = fullfile(pwd,'CamVid');

if ~exist(outputFolder, 'dir')

    mkdir(outputFolder)
    labelsZip = fullfile(outputFolder,'labels.zip');
    imagesZip = fullfile(outputFolder,'images.zip');

    disp('Downloading 16 MB CamVid dataset labels...');
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder,'labels'));

    disp('Downloading 557 MB CamVid dataset images...');
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder,'images'));
end
```

The data download time depends on your Internet connection. The example execution does not proceed until the download operation is complete. Alternatively, use your web browser to first download the data set to your local disk. Then, use the outputFolder variable to point to the location of the downloaded file.

Load CamVid Images

Use `imageDatastore` to load CamVid images. The imageDatastore enables you to efficiently load a large collection of images onto a disk.

```
imgDir = fullfile(outputFolder,'images','701_StillsRaw_full');
imds = imageDatastore(imgDir);
```

Display one of the images.

```
I = readimage(imds,25);
I = histeq(I);
imshow(I)
```

Load CamVid Pixel-Labeled Images

Use `pixelLabelDatastore` (Computer Vision Toolbox) to load CamVid pixel label image data. A pixelLabelDatastore encapsulates the pixel label data and the label ID to a class name mapping.

Following the training method described in the SegNet paper [3], group the 32 original classes in CamVid to 11 classes. Specify these classes.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];
```

To reduce 32 classes into 11 classes, multiple classes from the original data set are grouped together. For example, "Car" is a combination of "Car", "SUVPickupTruck", "Truck_Bus", "Train", and "OtherMoving". Return the grouped label IDs by using the camvidPixelLabelIDs supporting function.

```
labelIDs = camvidPixelLabelIDs();
```

Use the classes and label IDs to create the pixelLabelDatastore.

```
labelDir = fullfile(outputFolder,'labels');
pxds = pixelLabelDatastore(labelDir,classes,labelIDs);
```

Read and display one of the pixel-labeled images by overlaying it on top of an image.

```
C = readimage(pxds,25);
cmap = camvidColorMap;
B = labeloverlay(I,C,'ColorMap',cmap);
imshow(B)
pixelLabelColorbar(cmap,classes);
```

Areas with no color overlay do not have pixel labels and are not used during training.

Analyze Data Set Statistics

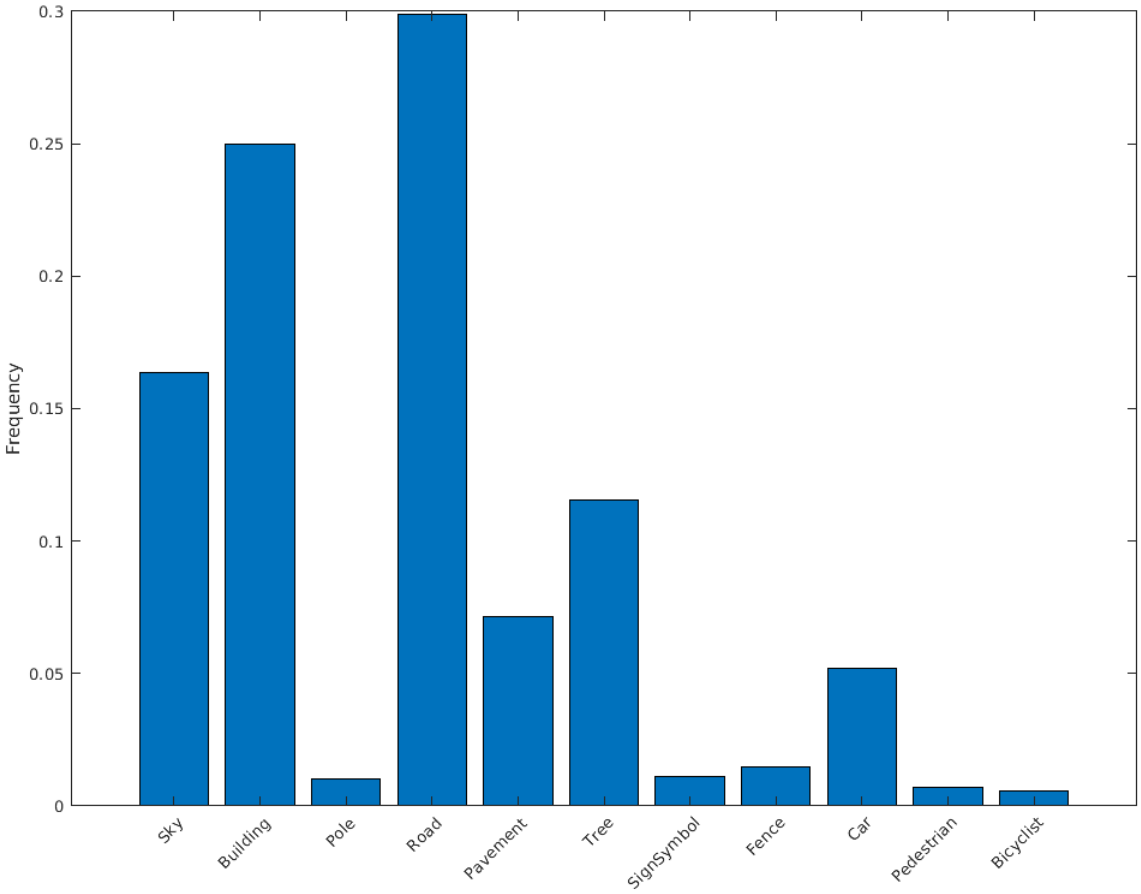
To see the distribution of class labels in the CamVid dataset, use `countEachLabel` (Computer Vision Toolbox). This function counts the number of pixels by class label.

```
tbl = countEachLabel(pxds)
```

```
tbl=11×3 table
         Name          PixelCount    ImagePixelCount
    _______________    __________    _______________

    {'Sky'       }    7.6801e+07      4.8315e+08
    {'Building'  }    1.1737e+08      4.8315e+08
    {'Pole'      }    4.7987e+06      4.8315e+08
    {'Road'      }    1.4054e+08      4.8453e+08
    {'Pavement'  }    3.3614e+07      4.7209e+08
    {'Tree'      }    5.4259e+07       4.479e+08
    {'SignSymbol'}    5.2242e+06      4.6863e+08
    {'Fence'     }    6.9211e+06       2.516e+08
    {'Car'       }    2.4437e+07      4.8315e+08
    {'Pedestrian'}    3.4029e+06      4.4444e+08
    {'Bicyclist' }    2.5912e+06      2.6196e+08
```

Visualize the pixel counts by class.

```
frequency = tbl.PixelCount/sum(tbl.PixelCount);

bar(1:numel(classes),frequency)
xticks(1:numel(classes))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes have an equal number of observations. The classes in CamVid are imbalanced, which is a common issue in automotive data sets of street scenes. Such scenes have more sky, building, and road pixels than pedestrian and bicyclist pixels because sky, buildings, and roads cover more area in the image. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes. Later on in this example, you use class weighting to handle this issue.

Resize CamVid Data

The images in the CamVid data set are 720-by-960. To reduce training time and memory usage, resize the images and pixel label images to 360-by-480 by using the resizeCamVidImages and resizeCamVidPixelLabels supporting functions.

```
imageFolder = fullfile(outputFolder,'imagesResized',filesep);
imds = resizeCamVidImages(imds,imageFolder);

labelFolder = fullfile(outputFolder,'labelsResized',filesep);
pxds = resizeCamVidPixelLabels(pxds,labelFolder);
```

Prepare Training and Test Sets

SegNet is trained by using 60% of the images from the dataset. The rest of the images are used for testing. The following code randomly splits the image and pixel label data into a training set and a test set.

```
[imdsTrain,imdsTest,pxdsTrain,pxdsTest] = partitionCamVidData(imds,pxds);
```

The 60/40 split results in the following number of training and test images:

```
numTrainingImages = numel(imdsTrain.Files)
```

```
numTrainingImages = 421
```

```
numTestingImages = numel(imdsTest.Files)
```

```
numTestingImages = 280
```

Create Network

Use `fcnLayers` (Computer Vision Toolbox) to create fully convolutional network layers initialized by using VGG-16 weights. The `fcnLayers` function performs the network transformations to transfer the weights from VGG-16 and adds the additional layers required for semantic segmentation. The output of the `fcnLayers` function is a LayerGraph object representing FCN. A LayerGraph object encapsulates the network layers and the connections between the layers.

```
imageSize = [360 480];
numClasses = numel(classes);
lgraph = fcnLayers(imageSize,numClasses);
```

The image size is selected based on the size of the images in the dataset. The number of classes is selected based on the classes in CamVid.

Balance Classes by Using Class Weighting

The classes in CamVid are not balanced. To improve training, you can use the pixel label counts computed earlier by the `countEachLabel` (Computer Vision Toolbox) function and calculate the median frequency class weights [3].

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;
```

Specify the class weights by using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights
```

```
pxLayer =
  PixelClassificationLayer with properties:

           Name: 'labels'
        Classes: [11×1 categorical]
    ClassWeights: [11×1 double]
      OutputSize: 'auto'

   Hyperparameters
    LossFunction: 'crossentropyex'
```

Update the SegNet network that has the new pixelClassificationLayer by removing the current pixelClassificationLayer and adding the new layer. The current pixelClassificationLayer is named 'pixelLabels'. Remove it by using the `removeLayers` function, add the new one by using the `addLayers` function, and connect the new layer to the rest of the network by using the `connectLayers` function.

```
lgraph = removeLayers(lgraph,'pixelLabels');
lgraph = addLayers(lgraph, pxLayer);
lgraph = connectLayers(lgraph,'softmax','labels');
```

Select Training Options

The optimization algorithm for training is Adam, which is derived from *adaptive moment estimation*. Use the `trainingOptions` function to specify the hyperparameters used for Adam.

```
options = trainingOptions('adam', ...
    'InitialLearnRate',1e-3, ...
    'MaxEpochs',100, ...
    'MiniBatchSize',4, ...
    'Shuffle','every-epoch', ...
    'CheckpointPath', tempdir, ...
    'VerboseFrequency',2);
```

A 'MiniBatchSize' of four reduces memory usage while training. You can increase or decrease this value based on the amount of GPU memory in your system.

'CheckpointPath' is set to a temporary location. This name-value pair enables the saving of network checkpoints at the end of every training epoch. If training is interrupted due to a system failure or power outage, you can resume training from the saved checkpoint. Make sure that the location specified by 'CheckpointPath' has enough space to store the network checkpoints.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without increasing the number of labeled training samples. To apply the same random transformation to both image and pixel label data use datastore combine and transform. First, combine `imdsTrain` and `pxdsTrain`.

```
dsTrain = combine(imdsTrain, pxdsTrain);
```

Next, use datastore transform to apply the desired data augmentation defined in the supporting function `augmentImageAndLabel`. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation.

```
xTrans = [-10 10];
yTrans = [-10 10];
dsTrain = transform(dsTrain, @(data)augmentImageAndLabel(data,xTrans,yTrans));
```

Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

Start Training

Start training using `trainNetwork` if the `doTraining` flag is true. Otherwise, load a pretrained network.

The training was verified on an NVIDIA™ Titan Xp with 12 GB of GPU memory. If your GPU has less memory, you might run out of memory. If you do not have enough memory in your system, try lowering the `MiniBatchSize` property in `trainingOptions` to 1. Training this network takes about 5 hours or longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(dsTrain,lgraph,options);
    save('FCN8sCamVid.mat','net');
end
```

Save the DAG network object as a MAT-file named `FCN8sCamVid.mat`. This MAT-file is used during code generation.

Perform MEX Code-generation

The fcn_predict.m function takes an image input and performs prediction on the image by using the deep learning network saved in `FCN8sCamVid.mat` file. The function loads the network object from `FCN8sCamVid.mat` into a persistent variable *mynet* and reuses the persistent object on subsequent prediction calls.

```
type('fcn_predict.m')
```

```
function out = fcn_predict(in)
%#codegen
% Copyright 2018-2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('FCN8sCamVid.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate a GPU Configuration object for MEX target setting target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a cuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` (MATLAB Coder) command specifying an input size [360, 480, 3]. This size corresponds to the input layer of FCN.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg fcn_predict -args {ones(360,480,3,'uint8')} -report
```

```
Code generation successful: View report
```

Run Generated MEX

Load and display an input image.

```
im = imread('testImage.png');
imshow(im);
```

Run prediction by calling `fcn_predict_mex` on the input image.

```
predict_scores = fcn_predict_mex(im);
```

The `predict_scores` variable is a three-dimensional matrix having 11 channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get pixel-wise labels.

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the input image and display the segmented region.

```
classes = [
    "Sky"
    "Building"
    "Pole"
    "Road"
    "Pavement"
    "Tree"
    "SignSymbol"
    "Fence"
    "Car"
    "Pedestrian"
    "Bicyclist"
    ];
```

```
cmap = camvidColorMap();
SegmentedImage = labeloverlay(im,argmax,'ColorMap',cmap);
figure
imshow(SegmentedImage);
pixelLabelColorbar(cmap,classes);
```

Cleanup

Clear the static network object that was loaded in memory.

```
clear mex;
```

Supporting Functions

```
function data = augmentImageAndLabel(data, xTrans, yTrans)
% Augment images and pixel label images using random reflection and
% translation.

for i = 1:size(data,1)

    tform = randomAffine2d(...
        'XReflection',true,...
        'XTranslation', xTrans, ...
        'YTranslation', yTrans);
```

```
    % Center the view at the center of image in the output space while
    % allowing translation to move the output image out of view.
    rout = affineOutputView(size(data{i,1}), tform, 'BoundsStyle', 'centerOutput');

    % Warp the image and pixel labels using the same transform.
    data{i,1} = imwarp(data{i,1}, tform, 'OutputView', rout);
    data{i,2} = imwarp(data{i,2}, tform, 'OutputView', rout);

end
end
```

References

[1] Long, J., E. Shelhamer, and T. Darrell. "Fully Convolutional Networks for Semantic Segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3431–3440.

[2] Brostow, G. J., J. Fauqueur, and R. Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.

[3] Badrinarayanan, V., A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." arXiv preprint arXiv:1511.00561, 2015.

Code Generation for Semantic Segmentation Network That Uses U-net

This example shows code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction on a DAG Network object for U-Net, a deep learning network for image segmentation.

For a similar example covering segmentation of images by using U-Net without the `codegen` command, see “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox).

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” (GPU Coder) and “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` (GPU Coder) function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Segmentation Network

U-Net [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. Combining these two series paths forms a U-shaped graph. The network was originally trained for and used to perform prediction on biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One challenge is differentiating classes that have similar

visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network in order to correctly classify each pixel.

The U-Net used is trained to segment pixels belonging to 18 classes which includes:

```
0. Other Class/Image Border      7. Picnic Table         14. Grass
1. Road Markings                 8. Black Wood Panel     15. Sand
2. Tree                          9. White Wood Panel     16. Water (Lake)
3. Building                     10. Orange Landing Pad   17. Water (Pond)
4. Vehicle (Car, Truck, or Bus) 11. Water Buoy           18. Asphalt (Parking Lot/Walkway)
5. Person                       12. Rocks
6. Lifeguard Chair              13. Other Vegetation
```

The `segmentImageUnet` Entry-Point Function

The `segmentImageUnet.m` entry-point function performs patchwise semantic segmentation on the input image by using the multispectralUnet network found in the `multispectralUnet.mat` file. The function loads the network object from the `multispectralUnet.mat` file into a persistent variable *mynet* and reuses the persistent variable on subsequent prediction calls.

```
type('segmentImageUnet.m')
```

```
%  OUT = segmentImageUnet(IM, PATCHSIZE) returns a semantically segmented
%  image, segmented using the network multispectralUnet. The segmentation
%  is performed over each patch of size PATCHSIZE.
%
% Copyright 2019-2021 The MathWorks, Inc.
function out = segmentImageUnet(im, patchSize)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
end

[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([patchSize, nChannel-1]));

% pad image to have dimensions as multiples of patchSize
padSize = zeros(1,2);
padSize(1) = patchSize(1) - mod(height, patchSize(1));
padSize(2) = patchSize(2) - mod(width, patchSize(2));

im_pad = padarray (im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);

out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');

for i = 1:patchSize(1):height_pad
    for j =1:patchSize(2):width_pad
        for p = 1:nChannel-1
```

```
            patch(:,:,p) = squeeze( im_pad( i:i+patchSize(1)-1,...
                                            j:j+patchSize(2)-1,...
                                            p));
        end

        % pass in input
        segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');

        % Takes the max of each channel (6 total at this point)
        [~,L] = max(segmentedLabels,[],3);
        patch_seg = uint8(L);

        % populate section of output
        out(i:i+patchSize(1)-1, j:j+patchSize(2)-1) = patch_seg;

    end
end

% Remove the padding
out = out(1:height, 1:width);
```

Get Pretrained U-Net DAG Network Object

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_url,pwd);
```

```
Downloading Pretrained U-net for Hamlin Beach dataset...
This will take several minutes to download...
done.
```

```
ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers including convolution, max pooling, depth concatenation, and the pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function. analyzeNetwork(net);

Prepare Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

```
Downloading Hamlin Beach dataset...
This will take several minutes to download...
done.
```

Load and examine the data in MATLAB.

```
load(fullfile(pwd,'data','rit18_data','rit18_data.mat'));

% Examine data
whos test_data
```

```
  Name           Size                         Bytes  Class     Attributes

  test_data      7x12446x7654            1333663576  uint16
```

The image has seven channels. The RGB color channels are the fourth, fifth, and sixth image channels. The first three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data  = switchChannelsToThirdPlane(test_data);

% Confirm data has the correct structure (channels last).
whos test_data
```

```
  Name               Size                         Bytes  Class     Attributes

  test_data      12446x7654x7            1333663576  uint16
```

Run MEX Code Generation

To generate CUDA code for `segmentImageUnet.m` entry-point function, create a GPU Configuration object for a MEX target setting the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create a `CuDNN` deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [12446,7654,7] and a patch size of [1024,1024]. These values correspond to the entire test_data size. The smaller patch sizes speed up inference. To see how the patches are calculated, see the `segmentImageUnet.m` entry-point function.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
inputArgs = {ones(size(test_data),'uint16'),coder.Constant([1024 1024])};
codegen -config cfg segmentImageUnet -args inputArgs -report
```

```
Code generation successful: View report
```

Run Generated MEX to Predict Results for test_data

This `segmentImageUnet` function takes in the data to test (test_data) and a vector containing the dimensions of the patch size to use. Take patches of the image, predict the pixels in a particular patch, then combine all the patches together. Due to the size of test_data (12446x7654x7), it is easier to process such a large image in patches.

```
segmentedImage = segmentImageUnet_mex(test_data,[1024 1024]);
```

To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the test data.

```
segmentedImage = uint8(test_data(:,:,7)~=0) .* segmentedImage;
```

Because the output of the semantic segmentation is noisy, remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImage = medfilt2(segmentedImage,[5,5]);
```

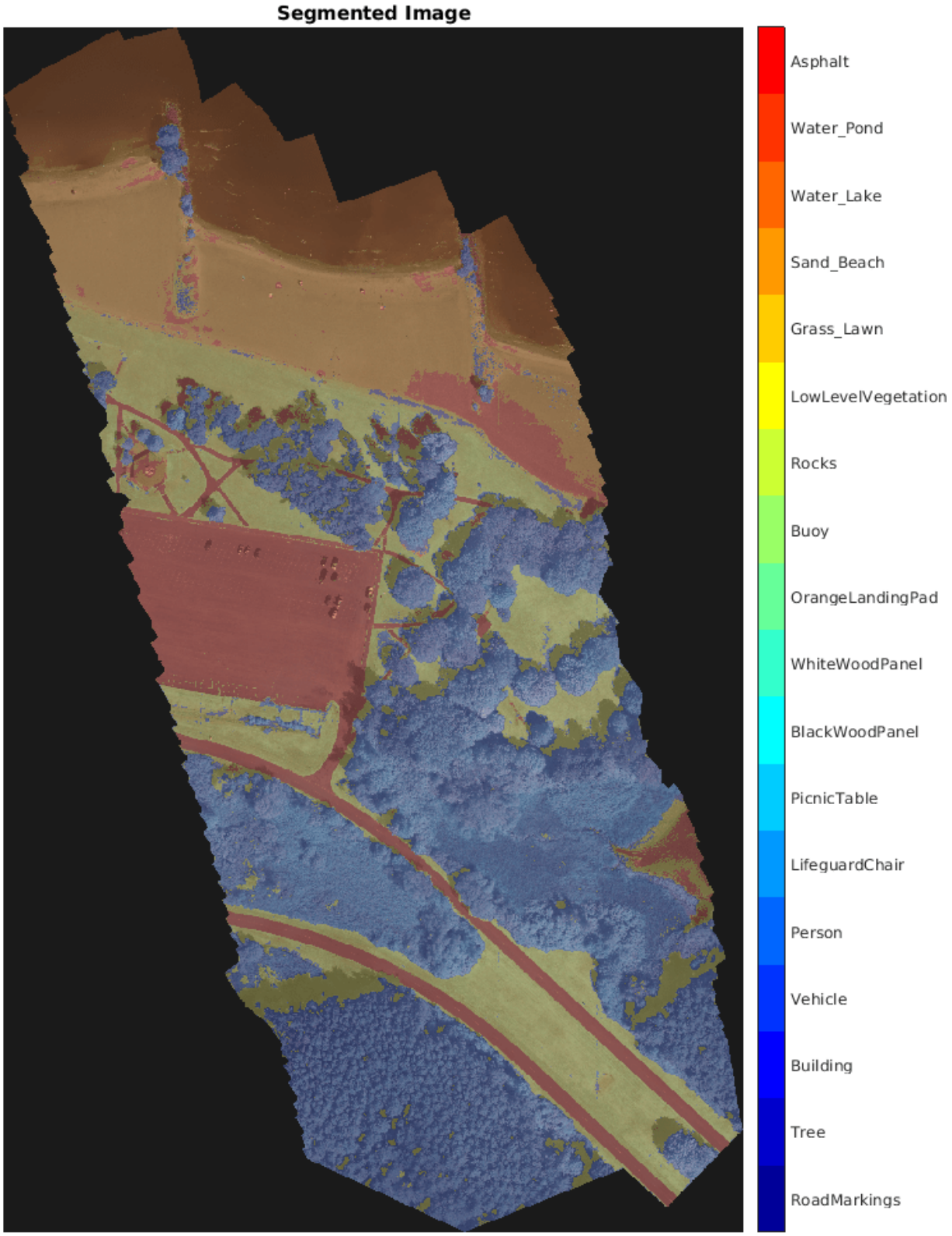
Display U-Net Segmented test_data

The following line of code creates a vector of the class names.

```
classNames = [ "RoadMarkings","Tree","Building","Vehicle","Person", ...
               "LifeguardChair","PicnicTable","BlackWoodPanel",...
               "WhiteWoodPanel","OrangeLandingPad","Buoy","Rocks",...
               "LowLevelVegetation","Grass_Lawn","Sand_Beach",...
               "Water_Lake","Water_Pond","Asphalt"];
```

Overlay the labels on the segmented RGB test image and add a color bar to the segmentation image.

```
cmap = jet(numel(classNames));
B = labeloverlay(imadjust(test_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),...
    segmentedImage,'Transparency',0.8,'Colormap',cmap);
figure
imshow(B)

N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,...
    'TickLabelInterpreter','none');
colormap(cmap)
title('Segmented Image');
```


Segmented Image
Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetation
Rocks
Buoy
OrangeLandingPad
WhiteWoodPanel
BlackWoodPanel
PicnicTable
LifeguardChair
Person
Vehicle
Building
Tree
RoadMarkings

References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597,* 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

Code Generation for Deep Learning on ARM Targets

This example shows how to generate and deploy code for prediction on an ARM®-based device without using a hardware support package.

When you generate code for prediction using the ARM Compute Library and a hardware support package, `codegen` generates code on the host computer, copies the generated files to the target hardware, and builds the executable on the target hardware. Without a hardware support package, `codegen` generates code on the host computer. You must run commands to copy the files and build the executable program on the target hardware.

This example uses the `packNGo` function to package all relevant files into a compressed zip file. Use this example to learn how to deploy the generated code on ARM Neon targets that do not have a hardware support package by using `packNGo`.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- Open Source Computer Vision Library(Open CV)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

This example is not supported for MATLAB Online.

squeezenet_predict Function

This example uses the DAG network SqueezeNet to show image classification with the ARM Compute Library. A pretrained SqueezeNet for MATLAB is available in the Deep Learning Toolbox. The `squeezenet_predict` function loads the SqueezeNet network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

```
type squeezenet_predict
```

```
% Copyright 2018 The MathWorks, Inc.

function out = squeezenet_predict(in)
%#codegen

% A persistent object mynet is used to load the DAG network object.
% At the first call to this function, the persistent object is constructed and
% set up. When the function is called subsequent times, the same object is reused
% to call predict on inputs, avoiding reconstructing and reloading the
% network object.

persistent mynet;
```

```
if isempty(mynet)
       mynet = coder.loadDeepLearningNetwork('squeezenet','squeezenet');
end

out = mynet.predict(in);
```

Set Up a Code Generation Configuration Object for a Static Library

When you generate code targeting an ARM-based device and do not use a hardware support package, create a configuration object for a library. Do not create a configuration object for an executable program.

Set up the configuration object for generation of C++ code and generation of code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Attach the Deep Learning Configuration Object to the Code Generation Configuration Object

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate Source C++ Code by Using codegen

```
codegen -config cfg squeezenet_predict -args {ones(227, 227, 3, 'single')} -d arm_compute
```

The code is generated in the arm_compute folder in the current working folder on the host computer.

Generate the Zip File using packNGo function

The packNGo function packages all relevant files in a compressed zip file.

```
zipFileName = 'arm_compute.zip';
bInfo = load(fullfile('arm_compute','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',t
```

The code is generated as zip file.

Copy the Generated Zip file to the Target Hardware

Copy the Zip file and extract into folder and remove the Zip file in the hardware

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy and extract zip file from Linux.

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetloc/arm_compute ];
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetloc/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetloc' zipFileName
```

Perform the steps below to copy and extract zip file from Windows.

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetloc/arm_compute ];
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetloc/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetloc' zipFileName
```

Copy Example Files to the Target Hardware

Copy these supporting files from the host computer to the target hardware:

- Input image, `coffeemug.png`
- Makefile for generating the library, `squeezenet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_squeezenet_arm_generic.mk`
- Synset dictionary, `synsetWords.txt`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy all the required files when running from Linux

```
if isunix, system('sshpass -p password scp squeezenet_predict_rtw.mk username@targetname:targetlo
if isunix, system('sshpass -p password scp coffeemug.png username@targetname:targetloc/arm_comput
if isunix, system('sshpass -p password scp makefile_squeezenet_arm_generic.mk username@targetname
if isunix, system('sshpass -p password scp synsetWords.txt username@targetname:targetloc/arm_comp
```

Perform the steps below to copy all the required files when running from Windows

```
if ispc, system('pscp.exe -pw password squeezenet_predict_rtw.mk username@targetname:targetloc/a
if ispc, system('pscp.exe -pw password coffeemug.png username@targetname:targetloc/arm_compute/'
if ispc, system('pscp.exe -pw password makefile_squeezenet_arm_generic.mk username@targetname:ta
if ispc, system('pscp.exe -pw password synsetWords.txt username@targetname:targetloc/arm_compute/
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables ARM_COMPUTELIB and LD_LIBRARY_PATH on the target hardware. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

ARM_ARCH variable is used in Makefile to pass compiler flags based on Arm Architecture. ARM_VER variable is used in Makefile to compile the code based on Arm Compute Version. Replace the hardware credentials and paths in similar to above steps.

Perform the below steps to build the library from Linux.

```
if isunix, system('sshpass -p password scp main_squeezenet_arm_generic.cpp username@targetname:ta
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -
```

Perform the below steps to build the library from windows.

```
if ispc, system('pscp.exe -pw password main_squeezenet_arm_generic.cpp username@targetname:targe
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -
```

Create Executable from the Library on the Target Hardware

Build the library with the source main wrapper file to create the executable. `main_squeezenet_arm_generic.cpp` is the C++ main wrapper file which invokes squeezenet_predict function to create the executable.

Run the below command to create the executable from Linux.

```
if isunix, system('sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -f
```

Run the below command to create the executable from Windows.

```
if ispc, system('plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -f
```

Run the Executable on the Target Hardware

Run the executable from Linux using below command.

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetloc/arm_compute/; ./sque
```

Run the executable from Windows using below command.

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetloc/arm_compute/; ./sque
```

```
Top 5 Predictions:
------------------------------
88.299% coffee mug
7.309% cup
1.098% candle
0.634% paper towel
0.591% water jug
```


MathWorks

Deep Learning Prediction with ARM Compute Using codegen

This example shows how to use `codegen` to generate code for a Logo classification application that uses deep learning on ARM® processors. The logo classification application uses the `LogoNet` series network to perform logo recognition from images. The generated code takes advantage of the ARM Compute library for computer vision and machine learning.

Prerequisites

- ARM processor that supports the NEON extension
- Open Source Computer Vision Library (OpenCV) v3.1
- Environment variables for ARM Compute and OpenCV libraries
- MATLAB® Coder™ for C++ code generation
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™ for using the `SeriesNetwork` object

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

This example is supported on Linux® and Windows® platforms and not supported for MATLAB Online.

Get the Pretrained SeriesNetwork

Download the pretrained `LogoNet` network and save it as `logonet.mat`, if it does not exist. The network was developed in MATLAB® and its architecture is similar to that of AlexNet. This network can recognize 32 logos under various lighting conditions and camera angles.

```
net = getLogonet();
```

The network contains 22 layers including convolution, fully connected, and the classification output layers.

```
net.Layers
```

```
ans =

  22×1 Layer array with layers:

     1   'imageinput'    Image Input             227×227×3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution             96 5×5×3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     Max Pooling             3×3 max pooling with stride [2  2] and padding
     5   'conv_2'        Convolution             128 3×3×96 convolutions with stride [1  1] and
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             3×3 max pooling with stride [2  2] and padding
     8   'conv_3'        Convolution             384 3×3×128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     Max Pooling             3×3 max pooling with stride [2  2] and padding
    11   'conv_4'        Convolution             128 3×3×384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     Max Pooling             3×3 max pooling with stride [2  2] and padding
```

```
    14   'fc_1'          Fully Connected         2048 fully connected layer
    15   'relu_5'        ReLU                    ReLU
    16   'dropout_1'     Dropout                 50% dropout
    17   'fc_2'          Fully Connected         2048 fully connected layer
    18   'relu_6'        ReLU                    ReLU
    19   'dropout_2'     Dropout                 50% dropout
    20   'fc_3'          Fully Connected         32 fully connected layer
    21   'softmax'       Softmax                 softmax
    22   'classoutput'   Classification Output   crossentropyex with 'adidas' and 31 other classe
```

Set Environment Variables

On the ARM target hardware, make sure that ARM_COMPUTELIB is set and that LD_LIBRARY_PATH contains the path to the ARM Compute Library folder.

See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

logonet_predict Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image using the deep learning network saved in the `LogoNet` MAT-file. The function loads the network object from `LogoNet.mat` into a persistent network variable *logonet*. On subsequent calls to the function, the persistent object is reused.

```
type logonet_predict
```

```
function out = logonet_predict(in)
%#codegen

% Copyright 2017-2020 The MathWorks, Inc.

persistent logonet;

if isempty(logonet)

    logonet = coder.loadDeepLearningNetwork('LogoNet.mat','logonet');
end

out = logonet.predict(in);

end
```

Set Up a Code Generation Configuration Object for a Static Library

When you generate code targeting an ARM-based device and do not use a hardware support package, create a configuration object for a library. Do not create a configuration object for an executable program.

Set up the configuration object for generation of C++ code and generation of code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Attach the Deep Learning Configuration Object to the Code Generation Configuration Object

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate Source C++ Code by Using codegen

```
codegen -config cfg logonet_predict -args {ones(227, 227, 3, 'single')} -d arm_compute
```

The code is generated in the `arm_compute` folder in the current working folder on the host computer.

Generate the Zip File Using the packNGo function

The packNGo function packages all relevant files in a compressed zip file.

```
zipFileName = 'arm_compute.zip';
bInfo = load(fullfile('arm_compute','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',t
```

Copy the Generated Zip File to the Target Hardware

Copy the Zip file and extract into a folder. Remove the Zip file from the target hardware.

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Run these commands to copy and extract zip file from Linux.

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetloc/arm_compute ];
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetloc/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetloc' zipFileName
```

Run these commands to copy and extract zip file from Windows.

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetloc/arm_compute ];
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetloc/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetloc' zipFileName
```

Copy Example Files to the Target Hardware

Copy these supporting files from the host computer to the target hardware:

- Input image, `coderdemo_google.png`
- Makefile for generating the library, `logonet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_arm_logo.mk`
- Synset dictionary, `synsetWordsLogoDet.txt`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetloc` with the destination folder for the files

Perform the steps below to copy all the required files when running from Linux

```
if isunix, system('sshpass -p password scp logonet_predict_rtw.mk username@targetname:targetloc/a
if isunix, system('sshpass -p password scp coderdemo_google.png username@targetname:targetloc/arm
if isunix, system('sshpass -p password scp makefile_arm_logo.mk username@targetname:targetloc/arm
if isunix, system('sshpass -p password scp synsetWordsLogoDet.txt username@targetname:targetloc/a
```

Perform the steps below to copy all the required files when running from Windows

```
if ispc, system('pscp.exe -pw password logonet_predict_rtw.mk username@targetname:targetloc/arm_c
if ispc, system('pscp.exe -pw password coderdemo_google.png username@targetname:targetloc/arm_com
if ispc, system('pscp.exe -pw password makefile_arm_logo.mk username@targetname:targetloc/arm_com
if ispc, system('pscp.exe -pw password synsetWordsLogoDet.txt username@targetname:targetloc/arm_c
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables ARM_COMPUTELIB and LD_LIBRARY_PATH on the target hardware. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder). The ARM_ARCH variable is used in the Makefile to pass compiler flags based on Arm Architecture. ARM_VER variable is used in the Makefile to compile the code based on Arm Compute Version. Replace the hardware credentials and paths in these commands similar to previous section.

Perform the below steps to build the library from Linux.

```
if isunix, system('sshpass -p password scp main_arm_logo.cpp username@targetname:targetloc/arm_co
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ 
```

Perform the below steps to build the library from windows.

```
if ispc, system('pscp.exe -pw password main_arm_logo.cpp username@targetname:targetloc/arm_comput
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ 
```

Create Executable from the Library on the Target Hardware

Build the library with the source main wrapper file to create the executable. `main_arm_logo.cpp` is the C++ main wrapper file which invokes the `logonet_predict` function.

Run the below command to create the executable from Linux.

```
if isunix, system('sshpass -p password ssh username@targetname "make -C targetloc/arm_compute/ -f
```

Run the below command to create the executable from Windows.

```
if ispc, system('plink.exe -l username -pw password targetname "make -C targetloc/arm_compute/ -f
```

Run the Executable on the Target Hardware

Run the executable from Linux using below command.

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetloc/arm_compute/; ./logo
```

Run the executable from Windows using below command.

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetloc/arm_compute/; ./logo
```

```
Top 5 Predictions:
------------------------------
99.992% google
0.003% corona
0.003% singha
0.001% esso
0.000% fedex
```

Deep Learning Code Generation on Intel Targets for Different Batch Sizes

This example shows how to use the `codegen` command to generate code for an image classification application that uses deep learning on Intel® processors. The generated code uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). This example consists of two parts:

- The first part shows how to generate a MEX function that accepts a batch of images as input.
- The second part shows how to generate an executable that accepts a batch of images as input.

Prerequisites

- Intel processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Environment variables for the compilers and libraries. For information on the supported versions of compilers, see Supported Compilers. For setting up the environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

This example is supported on Linux® , Windows® and Mac® platforms and not supported for MATLAB Online.

Download input video File

Download a sample video file.

```
if ~exist('./object_class.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/object_class.avi.zip';
    websave('object_class.avi.zip',url);
    unzip('object_class.avi.zip');
end
```

Define the `resnet_predict` Function

This example uses the DAG network ResNet-50 to show image classification on Intel desktops. A pretrained ResNet-50 model for MATLAB is available as part of the support package Deep Learning Toolbox Model for ResNet-50 Network.

The `resnet_predict` function loads the ResNet-50 network into a persistent network object and then performs prediction on the input. Subsequent calls to the function reuse the persistent network object.

```
type resnet_predict
```

```
% Copyright 2020 The MathWorks, Inc.

function out = resnet_predict(in)
%#codegen

% A persistent object mynet is used to load the series network object.
% At the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is reused
% to call predict on inputs, avoiding reconstructing and reloading the
% network object.
```

```
persistent mynet;

if isempty(mynet)
    % Call the function resnet50 that returns a DAG network
    % for ResNet-50 model.
    mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
end

% pass in input
out = mynet.predict(in);
```

Generate MEX for `resnet_predict`

To generate a MEX function for the `resnet_predict` function, use `codegen` with a deep learning configuration object for the MKL-DNN library. Attach the deep learning configuration object to the MEX code generation configuration object that you pass to `codegen`. Run the `codegen` command and specify the input as a 4D matrix of size [224,224,3,|batchSize|]. This value corresponds to the input layer size of the ResNet-50 network.

```
    batchSize = 5;
    cfg = coder.config('mex');
    cfg.TargetLang = 'C++';
    cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
    codegen -config cfg resnet_predict -args {ones(224,224,3,batchSize,'single')} -report

Code generation successful: To view the report, open('codegen\mex\resnet_predict\html\report.mlda
```

Perform Prediction on a Batch of Images

Presuming the Object_class.avi video file is already downloaded. Create the videoReader object and read five frames using videoReader read function.Since batchSize is set to 5 read 5 images .Resize the batch of input images to size needed by resnet50 size expected by ResNet50 network.

```
videoReader = VideoReader('Object_class.avi');
imBatch = read(videoReader,[1 5]);
imBatch = imresize(imBatch, [224,224]);
```

Call the generated `resnet_predict_mex` function which outputs classification results for the inputs that you provide.

```
predict_scores = resnet_predict_mex(single(imBatch));
```

Get top 5 probability scores and their labels for each image in the batch.

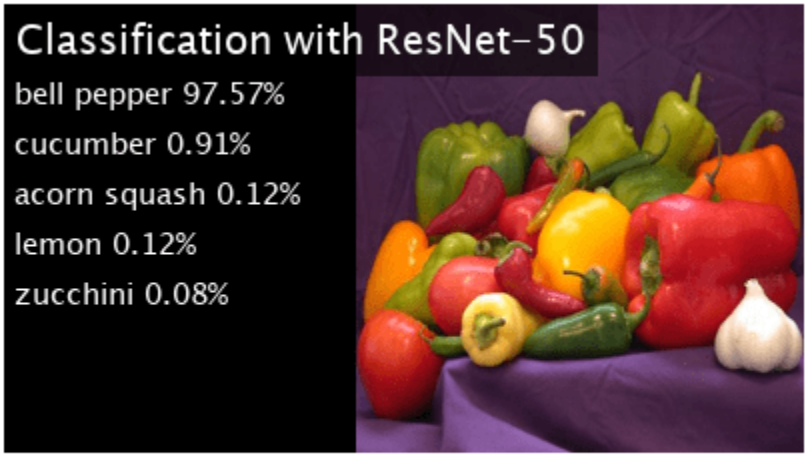
```
[val,indx] = sort(transpose(predict_scores), 'descend');
scores = val(1:5,:)*100;
net = resnet50;
classnames = net.Layers(end).ClassNames;
for i = 1:batchSize
    labels = classnames(indx(1:5,i));
    disp(['Top 5 predictions on image, ', num2str(i)]);
    for j=1:5
        disp([labels{j},' ',num2str(scores(j,i), '%2.2f'),'%'])
    end
end
```

For predictions on the first image, map the top five prediction scores to words in the `synset` dictionary.

```
fid = fopen('synsetWords.txt');
synsetOut = textscan(fid,'%s', 'delimiter', '\n');
synsetOut = synsetOut{1};
fclose(fid);
[val,indx] = sort(transpose(predict_scores), 'descend');
scores = val(1:5,1)*100;
top5labels = synsetOut(indx(1:5,1));
```

Display the top five classification labels on the image.

```
outputImage = zeros(224,400,3, 'uint8');
for k = 1:3
    outputImage(:,177:end,k) = imBatch(:,:,k,1);
end

scol = 1;
srow = 1;
outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50', 'TextCol
srow = srow + 30;
for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k},' ',num2str(scores(k),
    srow = srow + 25;
end

imshow(outputImage);
```



Clear the persistent network object from memory.

```
clear mex;
```

Define the `resnet_predict_exe` Entry-Point Function

To generate an executable from MATLAB code, define a new entry-point function `resnet_predict_exe`. This function is similar to the previous entry-point function `resent_predict` but, in addition, includes code for preprocessing and postprocessing. The API that `resnet_predict_exe` uses is platform independent. This function accepts a video and the batch size as input arguments. These arguments are compile-time constants.

```
type resnet_predict_exe
```

```
% Copyright 2020 The MathWorks, Inc.

function resnet_predict_exe(inputVideo,batchSize)
%#codegen

    % A persistent object mynet is used to load the series network object.
    % At the first call to this function, the persistent object is constructed and
    % setup. When the function is called subsequent times, the same object is reused
    % to call predict on inputs, avoiding reconstructing and reloading the
    % network object.
    persistent mynet;

    if isempty(mynet)
        % Call the function resnet50 that returns a DAG network
        % for ResNet-50 model.
        mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
    end

    % Create video reader and video player objects %
    videoReader = VideoReader(inputVideo);
    depVideoPlayer = vision.DeployableVideoPlayer;


    % Read the classification label names %
    synsetOut = readImageClassLabels('synsetWords.txt');

    i=1;
    % Read frames until end of video file %
    while ~(i+batchSize > (videoReader.NumFrames+1))
        % Read and resize batch of frames as specified by input argument%
        reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i);

        % run predict on resized input images %
        predict_scores = mynet.predict(reSizedImagesBatch);


        % overlay the prediction scores on images and display %
        overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depVideoPlay

        i = i+ batchSize;
    end
    release(depVideoPlayer);
end

function synsetOut = readImageClassLabels(classLabelsFile)
% Read the classification label names from the file
%
```

```
% Inputs :
% classLabelsFile - supplied by user
%
% Outputs :
% synsetOut       - cell array filled with 1000 image class labels

    synsetOut = cell(1000,1);
    fid = fopen(classLabelsFile);
    for i = 1:1000
        synsetOut{i} = fgetl(fid);
    end
    fclose(fid);
end

function reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i)
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% videoReader - Object used for reading the images from video file
% batchSize   - Number of images in batch to process. Supplied by user
% i           - index to track frames read from video file
%
% Outputs :
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize

    img = read(videoReader,[i (i+batchSize-1)]);
    reSizedImagesBatch = coder.nullcopy(ones(224,224,3,batchSize,'like',img));
    resizeTo  = coder.const([224,224]);
    reSizedImagesBatch(:,:,:,:) = imresize(img,resizeTo);
end


function overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depVideoPla
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% predict_scores  - classification results for given network
% synsetOut       - cell array filled with 1000 image class labels
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize
% batchSize       - Number of images in batch to process. Supplied by user
% depVideoPlayer  - Object for displaying results
%
% Outputs :
% Predicted results overlayed on input images

    % sort the predicted scores  %
    [val,indx] = sort(transpose(predict_scores), 'descend');

    for j = 1:batchSize
        scores = val(1:5,j)*100;
        outputImage = zeros(224,400,3, 'uint8');
        for k = 1:3
            outputImage(:,177:end,k) = reSizedImagesBatch(:,:,k,j);
        end

        % Overlay the results on image %
        scol = 1;
        srow = 1;
```

```
        outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50', 'Tex
        srow = srow + 30;
        for k = 1:5
            scoreStr = sprintf('%2.2f',scores(k));
            outputImage = insertText(outputImage, [scol, srow], [synsetOut{indx(k,j)},' ',scoreSt
            srow = srow + 25;
        end

        depVideoPlayer(outputImage);
    end
end
```

Structure of the `resnet_predict_exe` Function

The function `resnet_predict_exe` contains four subsections that perform these actions:

- Read the classification labels from supplied input text file
- Read the input batch of images and resize them as needed by the network
- Run inference on input image batch
- Overlay the results on the images

For more information each of these steps, see the subsequent sections.

The `readImageClassLabels` Function

This function accepts the `synsetWords.txt` file as an input argument. It reads the classification labels and populates a cell array.

```
function synsetOut = readImageClassLabels(classLabelsFile)
% Read the classification label names from the file
%
% Inputs :
% classLabelsFile - supplied by user
%
% Outputs :
% synsetOut       - cell array filled with 1000 image class labels

    synsetOut = cell(1000,1);
    fid = fopen(classLabelsFile);
    for i = 1:1000
        synsetOut{i} = fgetl(fid);
    end
    fclose(fid);
end
```

The `readImageInputBatch` Function

This function reads and resizes the images from the video input file that is passed to the function as an input argument. It reads the specified input images and resizes them to 224x224x3 which is the size the resnet50 network expects.

```
function reSizedImagesBatch = readImageInputBatch(videoReader,batchSize,i)
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
```

```
% videoReader - Object used for reading the images from video file
% batchSize   - Number of images in batch to process. Supplied by user
% i           - index to track frames read from video file
%
% Outputs :
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize

    img = read(videoReader,[i (i+batchSize-1)]);
    reSizedImagesBatch = coder.nullcopy(ones(224,224,3,batchSize,'like',img));
    resizeTo  = coder.const([224,224]);
    reSizedImagesBatch(:,:,:,:) = imresize(img,resizeTo);
end
```

The `mynet.predict` Function

This function accepts the resized batch of images as input and returns the prediction results.

```
% run predict on resized input images %
predict_scores = mynet.predict(reSizedImagesBatch);
```

The `overlayResultsOnImages` Function

This function accepts the prediction results and sorts them in descending order. It overlays these results on the input images and displays them.

```
function overlayResultsOnImages(predict_scores,synsetOut,reSizedImagesBatch,batchSize,depV
% Read and resize batch of frames as specified by input argument%
%
% Inputs :
% predict_scores  - classification results for given network
% synsetOut       - cell array filled with 1000 image class labels
% reSizedImagesBatch - Batch of images resized to 224x224x3xbatchsize
% batchSize       - Number of images in batch to process. Supplied by user
% depVideoPlayer  - Object for displaying results
%
% Outputs :
% Predicted results overlayed on input images

    % sort the predicted scores  %
    [val,indx] = sort(transpose(predict_scores), 'descend');

    for j = 1:batchSize
        scores = val(1:5,j)*100;
        outputImage = zeros(224,400,3, 'uint8');
        for k = 1:3
            outputImage(:,177:end,k) = reSizedImagesBatch(:,:,k,j);
        end

        % Overlay the results on image %
        scol = 1;
        srow = 1;
        outputImage = insertText(outputImage, [scol, srow], 'Classification with ResNet-50
        srow = srow + 30;
        for k = 1:5
            scoreStr = sprintf('%2.2f',scores(k));
            outputImage = insertText(outputImage, [scol, srow], [synsetOut{indx(k,j)},' '
            srow = srow + 25;
        end
```

```
            depVideoPlayer(outputImage);
        end
    end
```

Build and Run Executable

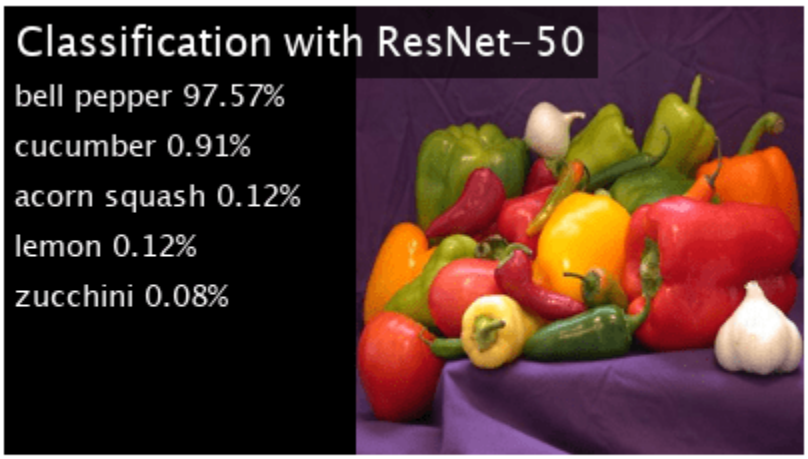
Create a code configuration object for generating an executable. Attach a deep learning configuration object to it. Set the `batchSize` and `inputVideoFile` variables.

If you do not intend to create a custom C++ main function and use the generated example C++ main instead, set the `GenerateExampleMain` parameter to `'GenerateCodeAndCompile'`. Also, disable cfg.EnableOpenMP to make sure there are no openmp library dependencies when you run your executable from the desktop terminal.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
batchSize = 5;
inputVideoFile = 'object_class.avi';
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';
cfg.EnableOpenMP = 0;
```

Run the `codegen` command to build the executable. Run the generated executable resnet_predict_exe either at the MATLAB command line or at the desktop terminal.

```
codegen -config cfg resnet_predict_exe -args {coder.Constant(inputVideoFile), coder.Consta
system('./resnet_predict_exe')
```



See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2

Generate C++ Code for Object Detection Using YOLO v2 and Intel MKL-DNN

This example shows how to generate C++ code for the YOLO v2 Object detection network on an Intel® processor. The generated code uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).

For more information, see "Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox).

Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Refer MKLDNN CPU Support to know the list of processors that supports MKL-DNN library
- MATLAB® Coder™ for C++ code generation
- MATLAB Coder Interface for Deep Learning support package
- Deep Learning Toolbox™ for using the `DAGNetwork` object
- Computer Vision Toolbox™ for video I/O operations

For more information on the supported versions of the compilers and libraries, see "Third-Party Hardware and Software" (MATLAB Coder).

This example is supported on Linux®, Windows®, and macOS platforms and not supported for MATLAB Online.

Get the Pretrained `DAGNetwork` Object

The DAG network contains 150 layers including convolution, ReLU, and batch normalization layers and the YOLO v2 transform and YOLO v2 output layers.

```
net = getYOLOv2();
```

```
Downloading pretrained detector (98 MB)...
```

Use the command `net.Layers` to see all the layers of the network.

```
net.Layers
```

Code Generation for `yolov2_detection` Function

The `yolov2_detection` function attached with the example takes an image input and runs the detector on the image using the network saved in `yolov2ResNet50VehicleExample.mat`. The function loads the network object from `yolov2ResNet50VehicleExample.mat` into a persistent variable `yolov2Obj`. Subsequent calls to the function reuse the persistent object for detection.

```
type('yolov2_detection.m')
```

```
function outImg = yolov2_detection(in)

%   Copyright 2018-2019 The MathWorks, Inc.

% A persistent object yolov2Obj is used to load the YOLOv2ObjectDetector object.
% At the first call to this function, the persistent object is constructed and
```

```
% set up. Subsequent calls to the function reuse the same object to call detection
% on inputs, thus avoiding having to reconstruct and reload the
% network object.
persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

% pass in input
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);
outImg = in;

% convert categorical labels to cell array of character vectors
labels = cellstr(labels);


if ~(isempty(bboxes) && isempty(labels))
% Annotate detections in the image.
    outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
end
```

To generate code, create a code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a MKL-DNN deep learning configuration object. Assign this object to the `DeepLearningConfig` property of the code configuration object. Specify the input size as an argument to the `codegen` command. In this example, the input layer size of the YOLO v2 network is `[224,224,3]`.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg yolov2_detection -args {ones(224,224,3,'uint8')} -report

Code generation successful: To view the report, open('codegen/mex/yolov2_detection/html/report.m
```

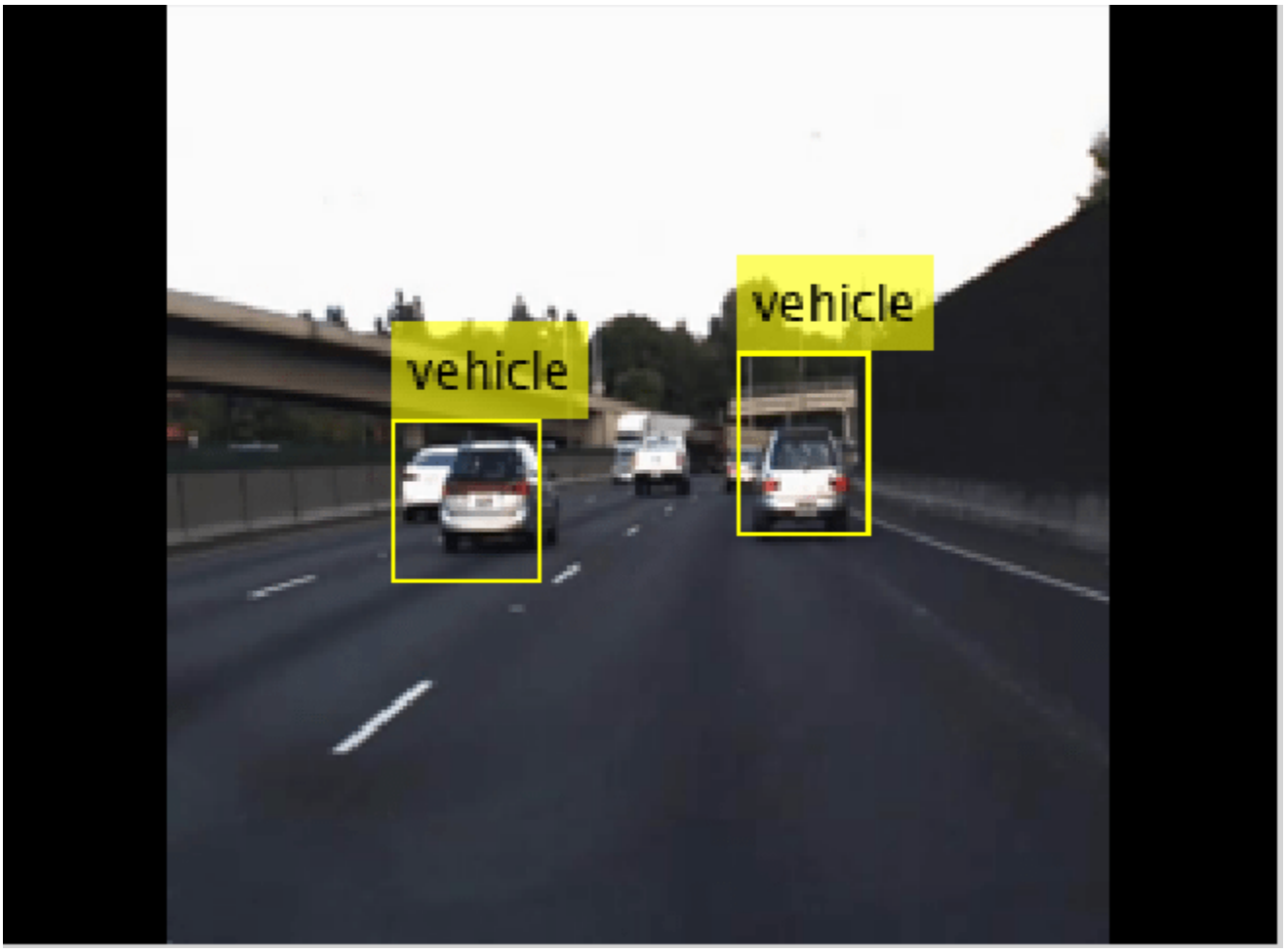
Run the Generated MEX Function on Example Input

Set up a video file reader and read the example input video `highway_lanechange.mp4`. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame by frame and detect the vehicles in the video by using the detector.

```
cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[224,224]);
    out = yolov2_detection_mex(in);
    depVideoPlayer(out);
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player
end
```



References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6517–25. Honolulu, HI: IEEE, 2017.

See Also

`coder.DeepLearningConfig` | `coder.hardware`

More About

- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” (MATLAB Coder)
- “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder)

Code Generation and Deployment of MobileNet-v2 Network to Raspberry Pi

This example shows how to generate and deploy C++ code that uses the MobileNet-v2 pretrained network for object prediction.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- Open Source Computer Vision Library(OpenCV) v2.4 (on the target ARM hardware)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- MATLAB Coder Interface for Deep Learning Libraries support package
- Deep Learning Toolbox™
- Deep Learning Toolbox Model for MobileNet-v2 Network support package
- Image Processing Toolbox™
- MATLAB Support Package for Raspberry Pi Hardware

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For supported versions of libraries and for information about setting up environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

This example is not supported for MATLAB online.

This example uses the DAG network MobileNet-v2 to perform image classification with the ARM® Compute Library. The pretrained MobileNet-v2 network for MATLAB is available in the Deep Learning Toolbox Model for MobileNet-v2 Network support package.

When you generate code that uses the ARM Compute Library and a hardware support package, `codegen` generates code on the host computer, copies the generated files to the target hardware, and builds the executable on the target hardware.

Configure Code Generation for the `mobilenet_predict` Function

The `mobilenet_predict` function calls the predict method of the MobileNet-v2 network object on an input image and returns the prediction score output. The function calls `coder.updateBuildInfo` to specify linking options for the generated makefile.

```
type mobilenet_predict
```

```
function out = mobilenet_predict(in)

persistent net;
opencv_linkflags = '`pkg-config --cflags --libs opencv`';
coder.updateBuildInfo('addLinkFlags',opencv_linkflags);
if isempty(net)
    net = coder.loadDeepLearningNetwork('mobilenetv2', 'mobilenet');
end
```

```
out = net.predict(in);

end
```

Create a C++ code generation configuration object.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Specify Use of the ARM Compute Library. The ARM Compute Library provides optimized functionality for the Raspberry Pi hardware. To generate code that uses the ARM Compute Library, create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute Library installed on your Raspberry Pi and the architecture of the Raspberry Pi. Attach the deep learning configuration object to the code generation configuration object.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
supportedVersions = dlcfg.getARMComputeSupportedVersions;
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
cfg.DeepLearningConfig = dlcfg;
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi Hardware function `raspi` to create a connection to the Raspberry Pi. In this code, replace:

- `raspiname` with the host name of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
r = raspi('raspiname','username','password');
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for the Raspberry Pi and attach it to the code generation configuration object.

```
hw = coder.hardware('Raspberry Pi');
cfg.Hardware = hw;
```

Specify a build folder on the Raspberry Pi:

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Provide a C++ Main File

Specify the main file `main_mobilenet.cpp` in the code generation configuration object. The file calls the generated C++ code for the `mobilenet_predict` function. The file reads the input image, passes the data to the generated function calls, retrieves the predictions on the image, and prints the prediction scores to a file.

```
cfg.CustomSource = 'main_mobilenet.cpp';
```

Generate the Executable Program on the Raspberry Pi

Generate C++ code. When you use `codegen` with the MATLAB Support Package for Raspberry PI Hardware, the executable is built on the Raspberry Pi.

For code generation, you must set the “Environment Variables” (MATLAB Coder) `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the Raspberry Pi.

```
codegen -config cfg mobilenet_predict -args {ones(224, 224, 3,'single')} -report
```

Fetch the Generated Executable Folder

To test the generated code on the Raspberry Pi, copy the input image to the generated code folder. You can find this folder manually or by using the `raspi.utils.getRemoteBuildDirectory` API. This function lists the folders of the binary files that are generated by using `codegen`. Assuming that the binary is found in only one folder, enter:

```
applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName','mobilenet_predict');
targetDirPath = applicationDirPaths{1}.directory;
```

Copy Example Files to the Raspberry Pi

To copy files required to run the executable program, use `putFile`.

```
r.putFile('peppers_raspi_mobilenet.png',targetDirPath);
```

Run the Executable Program on the Raspberry Pi

Run the executable program on the Raspberry Pi from MATLAB and direct the output back to MATLAB.

```
exeName = 'mobilenet_predict.elf';
argsforexe = ' peppers_raspi_mobilenet.png '; % Provide the input image;
command = ['cd ' targetDirPath ';sudo ./' exeName argsforexe];
output = system(r,command);
```

Get the Prediction Scores for the 1000 Output Classes of the Network

```
outputfile = [targetDirPath, '/output.txt'];
r.getFile(outputfile);
```

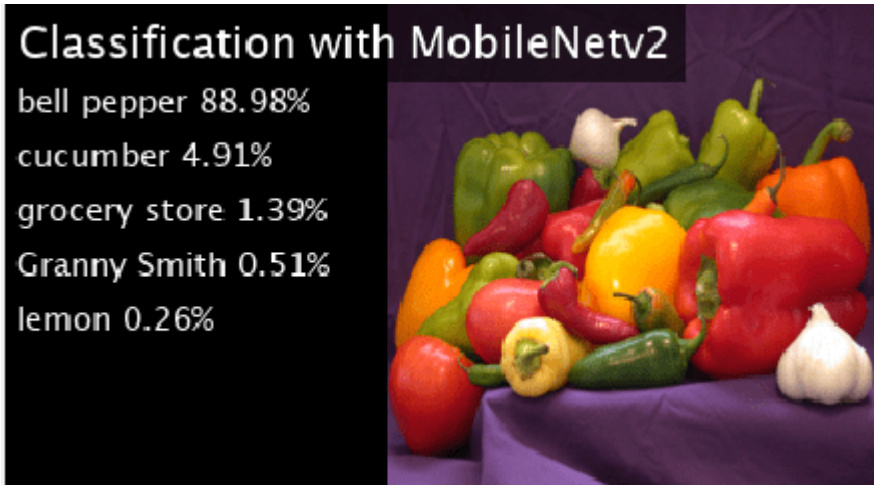
Map the Prediction Scores to Labels and Display Output

Map the top five prediction scores to the corresponding labels in the trained network, and display the output.

```
type mapPredictedScores_mobilenet
```

```
%% Map the Prediction Scores to Labels and Display Output
net = mobilenetv2;
ClassNames = net.Layers(end).ClassNames;

%% Read the classification
fid = fopen('output.txt') ;
S = textscan(fid,'%s');
fclose(fid) ;
S = S{1} ;
predict_scores = cellfun(@(x)str2double(x), S);

%% Remove NaN values that were strings
predict_scores(isnan(predict_scores))=[];
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
```

```
top5labels = ClassNames(indx(1:5));

%% Display classification labels on the image
im = imread('peppers_raspi_mobilenet.png');
im = imresize(im, [224 224]);
outputImage = zeros(224,400,3, 'uint8');
for k = 1:3
    outputImage(:,177:end,k) = im(:,:,k);
end
scol = 1;
srow = 1;
outputImage = insertText(outputImage, [scol, srow], 'Classification with MobileNetv2', 'TextColo
srow = srow + 30;
for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k},' ',num2str(scores(k), '%
    srow = srow + 25;
end
imshow(outputImage);
```



See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
- “Code Generation for Deep Learning on ARM Targets” (MATLAB Coder)

Code Generation for Semantic Segmentation Application on Intel CPUs That Uses U-Net

This example demonstrates code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction by using the deep learning network U-Net for image segmentation.

For a similar example that demonstrates segmentation of images by using U-Net but does not use the `codegen` command, see "Semantic Segmentation of Multispectral Images Using Deep Learning" (Image Processing Toolbox).

Third-Party Prerequisites

- Xeon processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions

This example is supported on Linux®, Windows®, and macOS platforms.

This example uses the Intel MKL-DNN library that ships with MATLAB and generates a MEX function for semantic segmentation.

This example is not supported in MATLAB Online.

Overview of U-Net

U-Net [1] is a type of convolutional neural network (CNN)that is designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. The combination of these two series paths forms a U-shaped graph. The network was originally trained to perform prediction for biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One of the challenges is differentiating classes that have similar visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network in order to correctly classify each pixel.

The U-Net this example uses is trained to segment pixels belonging to 18 classes which includes:

```
0. Other Class/Image Border      7. Picnic Table         14. Grass
1. Road Markings                 8. Black Wood Panel     15. Sand
2. Tree                          9. White Wood Panel     16. Water (Lake)
3. Building                     10. Orange Landing Pad   17. Water (Pond)
4. Vehicle (Car, Truck, or Bus) 11. Water Buoy           18. Asphalt (Parking Lot/Walkway)
5. Person                       12. Rocks
6. Lifeguard Chair              13. Other Vegetation
```

Get Pretrained U-Net DAG Network Object

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_url,pwd);
```

```
Downloading Pre-trained U-net for Hamlin Beach dataset...
This will take several minutes to download...
done.
```

```
ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers including convolution, max pooling, depth concatenation, and pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
%   analyzeNetwork(net);
```

The `segmentImageUnet` Entry-Point Function

The segmentImageUnet.m entry-point function performs semantic segmentation on the input image for each patch of a fixed size by using the multispectralUnet network contained in the `multispectralUnet.mat` file. This function loads the network object from the `multispectralUnet.mat` file into a persistent variable `mynet`. The function reuses this persistent variable in subsequent prediction calls.

```
type('segmentImageUnet.m')
```

```
%  OUT = segmentImageUnet(IM, PATCHSIZE) returns a semantically segmented
%  image, segmented using the network multispectralUnet. The segmentation
%  is performed over each patch of size PATCHSIZE.
%
% Copyright 2019-2020 The MathWorks, Inc.
function out = segmentImageUnet(im, patchSize)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
end

[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([patchSize, nChannel-1]));

% pad image to have dimensions as multiples of patchSize
padSize = zeros(1,2);
padSize(1) = patchSize(1) - mod(height, patchSize(1));
padSize(2) = patchSize(2) - mod(width, patchSize(2));

im_pad = padarray (im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);

out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');
```

```
for i = 1:patchSize(1):height_pad
    for j =1:patchSize(2):width_pad
        for p = 1:nChannel-1
            patch(:,:,p) = squeeze( im_pad( i:i+patchSize(1)-1,...
                                            j:j+patchSize(2)-1,...
                                            p));
        end

        % pass in input
        segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');

        % Takes the max of each channel (6 total at this point)
        [~,L] = max(segmentedLabels,[],3);
        patch_seg = uint8(L);

        % populate section of output
        out(i:i+patchSize(1)-1, j:j+patchSize(2)-1) = patch_seg;

    end
end

% Remove the padding
out = out(1:height, 1:width);
```

Prepare Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

```
Downloading Hamlin Beach dataset...
This will take several minutes to download...
done.
```

Load and examine the data in MATLAB.

```
load(fullfile(pwd,'data','rit18_data','rit18_data.mat'));

% Examine data
whos test_data
```

```
  Name           Size                         Bytes  Class     Attributes

  test_data      7x12446x7654            1333663576  uint16
```

The image has seven channels. The RGB color channels are the fourth, fifth, and sixth image channels. The first three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data  = switchChannelsToThirdPlane(test_data);
```

Confirm data has the correct structure (channels last).

```
whos test_data
```

```
  Name               Size                     Bytes  Class     Attributes

  test_data      12446x7654x7            1333663576  uint16
```

This example uses a cropped version of the full Hamlin Beach State Park dataset that the `test_data` variable contains. Crop the height and width of `test_data` to create the variable `input_data` that this example uses.

```
test_datacropRGB = imcrop(test_data(:,:,1:3),[2600, 3000, 2000, 2000]);
test_datacropInfrared = imcrop(test_data(:,:,4:6),[2600, 3000, 2000, 2000]);
test_datacropMask = imcrop(test_data(:,:,7),[2600, 3000, 2000, 2000]);
input_data(:,:,1:3) = test_datacropRGB;
input_data(:,:,4:6) = test_datacropInfrared;
input_data(:,:,7) = test_datacropMask;
```

Examine the input_data variable.

```
whos('input_data');
```

```
  Name               Size                       Bytes  Class     Attributes

  input_data      2001x2001x7                56056014  uint16
```

Generate MEX

To generate a MEX function for the segmentImageUnet.m entry-point function, create a code configuration object `cfg` for MEX code generation. Set the target language to C++. Use the `coder.DeepLearningConfig` (GPU Coder) function to create an MKL-DNN deep learning configuration object and assign it to the `DeepLearningConfig` property of `cfg`. Run the `codegen` command specifying an input size of [12446,7654,7] and a patch size of [1024,1024]. These values correspond to the size of the entire `input_data` variable. The smaller patch sizes speed up inference. To see how the patches are calculated, see the `segmentImageUnet` entry-point function.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg segmentImageUnet -args {ones(size(input_data),'uint16'),coder.Constant([1024
```

```
Code generation successful: To view the report, open('codegen\mex\segmentImageUnet\html\report.m
```

Run Generated MEX to Predict Results for `input_data`

The `segmentImageUnet` function accepts `input_data` and a vector containing the dimensions of the patch size as inputs. The function divides the image into patches, predicts the pixels in a particular patch, and finally combines all the patches. Because of the large size of `input_data` (12446x7654x7), it is easier to process the image in patches.

```
segmentedImage = segmentImageUnet_mex(input_data,[1024 1024]);
```

To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the test data.

```
segmentedImage = uint8(input_data(:,:,7)~=0) .* segmentedImage;
```

Remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImage = medfilt2(segmentedImage,[5,5]);
```

Display U-Net Segmented input_data

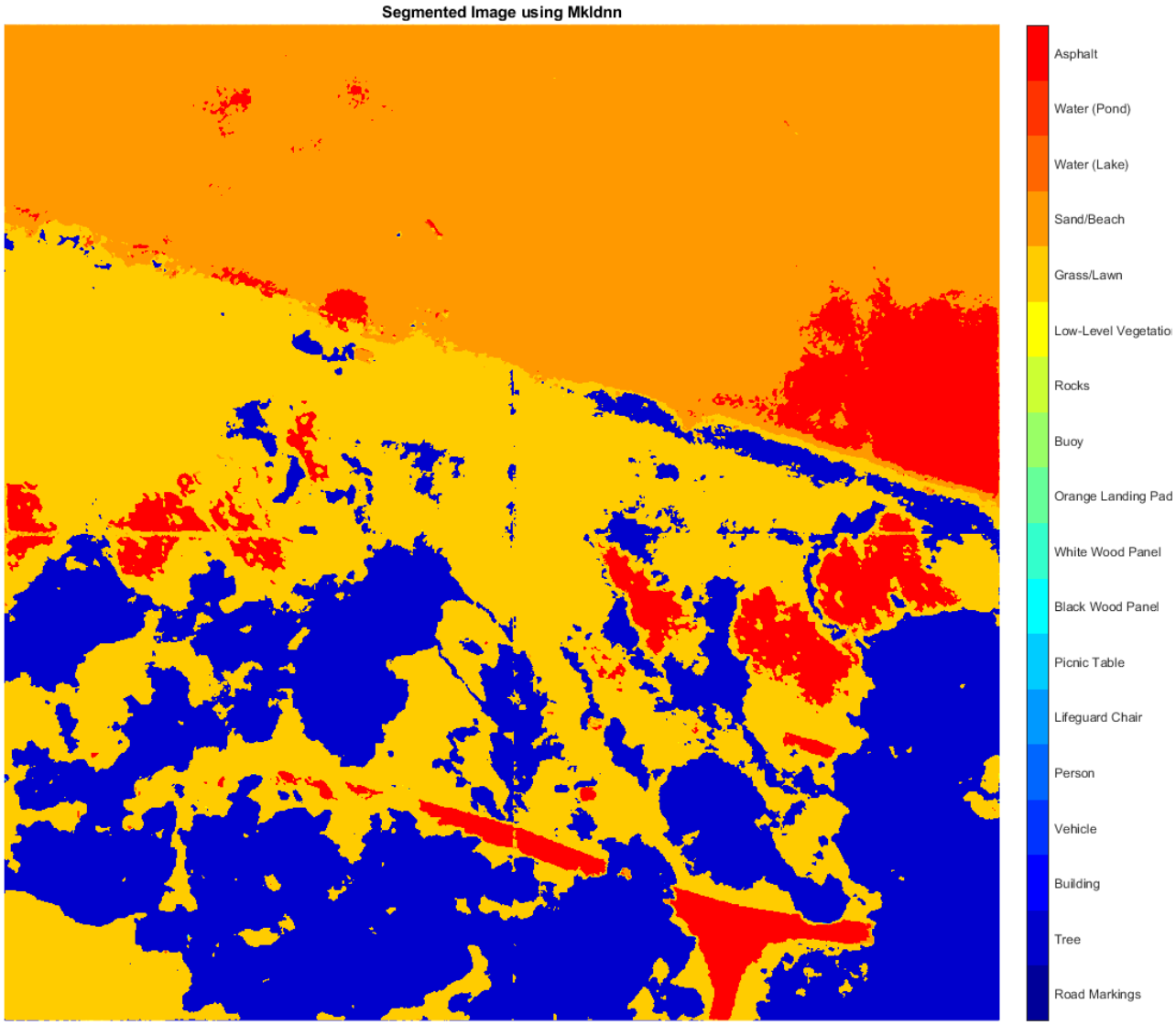
This line of code creates a vector of the class names:

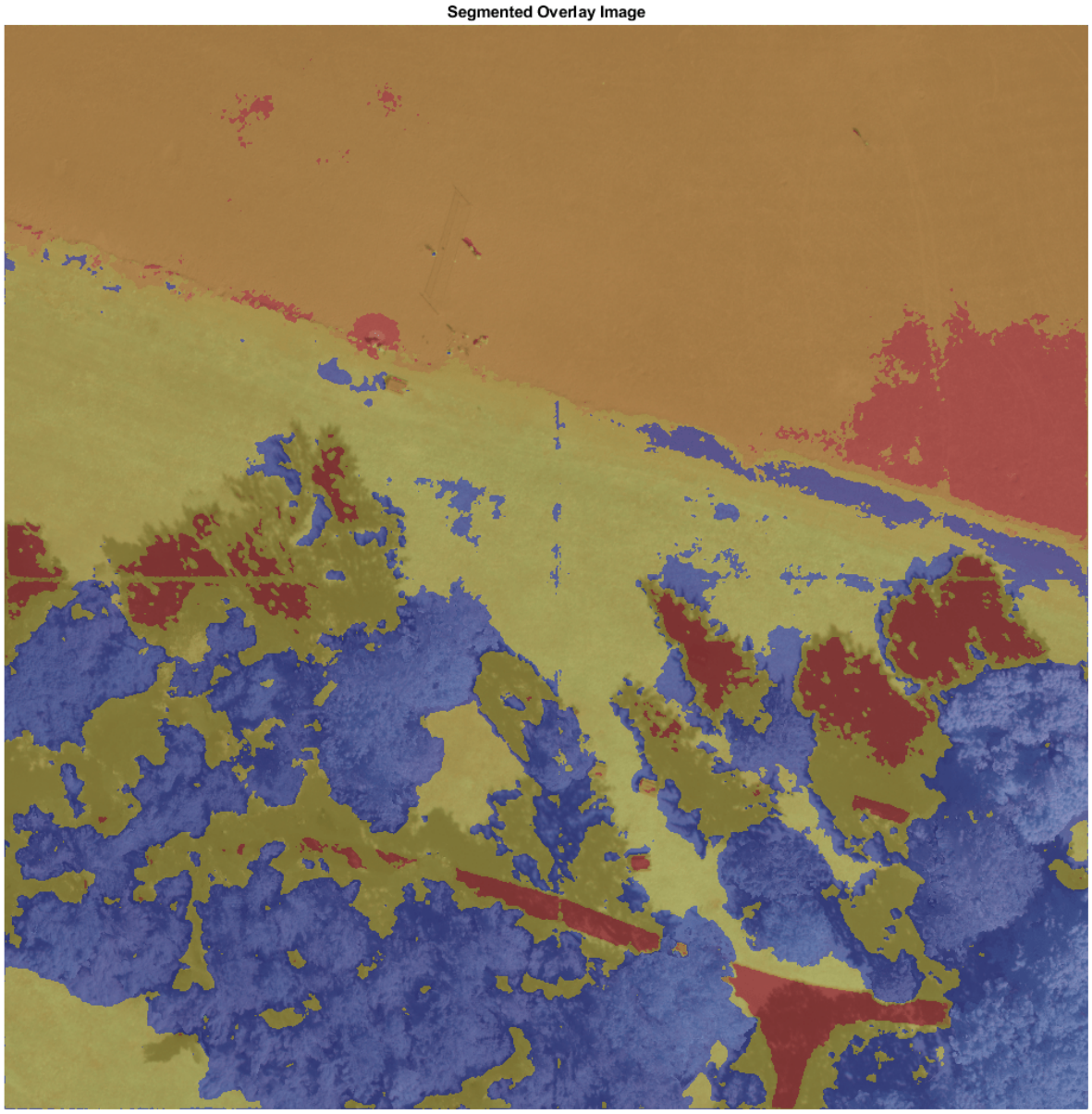
```
classNames = net.Layers(end).Classes;
```

Overlay the labels on the segmented RGB test image and add a color bar to the segmentation image.

```
% Display input data

figure(1);
imshow(histeq(input_data(:,:,1:3)));
title('Input Image');
cmap = jet(numel(classNames));
segmentedImageOut = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmentedI

% Display segmented data

figure(2);
imshow(segmentedImageOut);
title('Segmented Image Output');
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','n
colormap(cmap)
title('Segmented Image using Mkldnn');
segmentedImageOverlay = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmen
figure(3);
imshow(segmentedImageOverlay);
title('Segmented Overlay Image');
```


Input Image


Segmented Image using Mkldnn
Asphalt
Water (Pond)
Water (Lake)
Sand/Beach
Grass/Lawn
Low-Level Vegetatio
Rocks
Buoy
Orange Landing Pad
White Wood Panel
Black Wood Panel
Picnic Table
Lifeguard Chair
Person
Vehicle
Building
Tree
Road Markings



References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597,* 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

See Also

`coder.DeepLearningConfig` | `coder.hardware` | `analyzeNetwork`

More About

- “Deep Learning Code Generation on Intel Targets for Different Batch Sizes” (MATLAB Coder)
- “Workflow for Deep Learning Code Generation with MATLAB Coder” (MATLAB Coder)
- “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox)

Code Generation for Semantic Segmentation Application on ARM® Neon targets That Uses U-Net

This example shows how to generate code for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a static library that performs prediction on a DAG Network object for U-Net. U-Net is a deep learning network for image segmentation.

For a similar example that uses U-Net for image segmentation but does not use the `codegen` command, see "Semantic Segmentation of Multispectral Images Using Deep Learning" (Image Processing Toolbox).

Prerequisites

- ARM processor that supports the NEON extension and has a RAM of at least 3GB
- ARM Compute Library (on the target ARM hardware)
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- MATLAB Coder Interface for Deep Learning Libraries support package
- Deep Learning Toolbox™

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information about supported versions of libraries and about environment variables, see "Prerequisites for Deep Learning with MATLAB Coder" (MATLAB Coder).

This example is not supported in MATLAB Online.

Overview of U-Net

U-Net [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. The combination of these two series paths forms a U-shaped graph. The U-Net network was originally trained to perform prediction on biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep learning based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One of the challenges of such computation is to differentiating classes that have similar visual characteristics, such as classifying a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network to correctly classify each pixel.

The U-Net that this example uses is trained to segment pixels belonging to a set of 18 classes which includes:

```
0. Other Class/Image Border      7. Picnic Table         14. Grass
1. Road Markings                 8. Black Wood Panel     15. Sand
2. Tree                          9. White Wood Panel     16. Water (Lake)
3. Building                     10. Orange Landing Pad   17. Water (Pond)
4. Vehicle (Car, Truck, or Bus) 11. Water Buoy           18. Asphalt (Parking Lot/Walkway)
5. Person                       12. Rocks
6. Lifeguard Chair              13. Other Vegetation
```

The `segmentationUnetARM` Entry-Point Function

The segmentationUnetARM.m entry-point function performs patchwise semantic segmentation on the input image by using the multispectralUnet network contained in the `multispectralUnet.mat` file. The function loads the network object from the `multispectralUnet.mat` file into a persistent variable `mynet` and reuses the persistent variable on subsequent prediction calls.

```
type('segmentationUnetARM.m')
```

```
%  OUT = segmentationUnetARM(IM) returns a semantically segmented
%  image, which is segmented using the network multispectralUnet. This segmentation
%  is performed on the input image patchwise on patches of size 256,256.
%
% Copyright 2019-2020 The MathWorks, Inc.
function out = segmentationUnetARM(im)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
end

% The input data has to be padded to the size compatible
% with the network Input Size. This input_data is padded inorder to
% perform semantic segmentation on each patch of size (Network Input Size)
[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([256, 256, nChannel-1]));
%
padSize = zeros(1,2);
padSize(1) = 256 - mod(height, 256);
padSize(2) = 256 - mod(width, 256);
%
% Pad image must have have dimensions as multiples of network input dimensions
im_pad = padarray (im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);
%
out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');

for i = 1:256:height_pad
    for j =1:256:width_pad
        for p = 1:nChannel -1
            patch(:,:,p) = squeeze( im( i:i+255,...
                                        j:j+255,...
                                        p));
        end

        % pass in input
        segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');
```

```
        % Takes the max of each channel (6 total at this point)
        [~,L] = max(segmentedLabels,[],3);
        patch_seg = uint8(L);

        % populate section of output
        out(i:i+255, j:j+255) = patch_seg;

    end
end

% Remove the padding
out = out(1:height, 1:width);
```

Get Pretrained U-Net DAG Network Object

Download the `multispectralUnet.mat` file and load the U-Net DAG network object.

```
if ~exist('trainedUnet/multispectralUnet.mat','file')
    trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat'
    downloadUNet(trainedUnet_url,pwd);
end

ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers that include convolution, max pooling, depth concatenation, and pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(net);
```

Prepare Input Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

Load and examine the data in MATLAB.

```
load(fullfile(pwd,'data','rit18_data','rit18_data.mat'));
```

Examine data

```
whos test_data
```

The image has seven channels. The RGB color channels are the fourth, fifth, and sixth image channels. The first three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data = switchChannelsToThirdPlane(test_data);
```

Confirm data has the correct structure (channels last).

```
whos test_data
```

This example uses a cropped version of the full Hamlin Beach State Park dataset that the `test_data` variable contains. Crop the height and width of `test_data` to create the variable `input_data` that this example uses.

```
test_datacropRGB = imcrop(test_data(:,:,1:3),[2600, 3000, 2000, 2000]);
test_datacropInfrared = imcrop(test_data(:,:,4:6),[2600, 3000, 2000, 2000]);
test_datacropMask = imcrop(test_data(:,:,7),[2600, 3000, 2000, 2000]);

input_data(:,:,1:3) = test_datacropRGB;
input_data(:,:,4:6) = test_datacropInfrared;
input_data(:,:,7) = test_datacropMask;
```

Examine the `input_data` variable.

```
whos('input_data');
```

Write the input data into a text file that is passed as input to the generated executable.

```
WriteInputDatatoTxt(input_data);
[height, width, channels] = size(input_data);
```

Set Up a Code Generation Configuration Object for a Static Library

To generate code that targets an ARM-based device, create a configuration object for a library. Do not create a configuration object for an executable program. Set up the configuration object for generation of C++ source code only.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.GenCodeOnly = true;
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the library version and the architecture of the target ARM processor. For example, suppose that the target board is a HiKey/Rock960 board with ARMv8 architecture and ARM Compute Library version 19.05.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
dlcfg.ArmArchitecture = 'armv8';
```

Assign the `DeepLearningConfig` property of the code generation configuration object `cfg` to the deep learning configuration object `dlcfg`.

```
cfg.DeepLearningConfig = dlcfg;
```

Generate C++ Source Code by Using codegen

```
codegen -config cfg segmentationUnetARM -args {ones(size(input_data),'uint16')} -d unet_predict
```

The code gets generated in the `unet_predict` folder that is located in the current working directory on the host computer.

Generate Zip File by Using packNGo

The `packNGo` function packages all relevant files into a compressed zip file.

```
zipFileName = 'unet_predict.zip';
bInfo = load(fullfile('unet_predict','buildInfo.mat'));
packNGo(bInfo.buildInfo, {'fileName', zipFileName,'minimalHeaders', false, 'ignoreFileMissing',t
```

The name of the generated zip file is `unet_predict.zip`.

Copy Generated Zip file to the Target Hardware

Copy the zip file into the target hardware board. Extract the contents of the zip file into a folder and delete the zip file from the hardware.

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetDir` with the destination folder for the files

On the Linux® platform, to transfer and extract the zip file on the target hardware, run these commands:

```
if isunix, system(['sshpass -p password scp -r '  fullfile(pwd,zipFileName) ' username@targetname
if isunix, system('sshpass -p password ssh username@targetname "if [ -d targetDir/unet_predict ]
if isunix, system(['sshpass -p password ssh username@targetname "unzip targetDir/' zipFileName '
if isunix, system(['sshpass -p password ssh username@targetname "rm -rf  targetDir/' zipFileName
```

On the Windows® platform, to transfer and extract the zip file on the target hardware, run these commands:

```
if ispc, system(['pscp.exe -pw password -r '  fullfile(pwd,zipFileName) ' username@targetname:ta
if ispc, system('plink.exe -l username -pw password targetname "if [ -d targetDir/unet_predict ]
if ispc, system(['plink.exe -l username -pw password targetname "unzip targetDir/' zipFileName '
if ispc, system(['plink.exe -l username -pw password targetname "rm -rf  targetDir/' zipFileName
```

Copy Supporting Files to the Target Hardware

Copy these files from the host computer to the target hardware:

- Input data, `input_data.txt`
- Makefile for creating the library, `unet_predict_rtw.mk`
- Makefile for building the executable program, `makefile_unet_arm_generic.mk`

In the following commands, replace:

- `password` with your password
- `username` with your user name
- `targetname` with the name of your device
- `targetDir` with the destination folder for the files

On the Linux® platform, to transfer the supporting files to the target hardware, run these commands:

```
if isunix, system('sshpass -p password scp unet_predict_rtw.mk username@targetname:targetDir/unet
if isunix, system('sshpass -p password scp input_data.txt username@targetname:targetDir/unet_pred
if isunix, system('sshpass -p password scp makefile_unet_arm_generic.mk username@targetname:targe
```

On the Windows® platform, to transfer the supporting files to the target hardware, run these commands:

```
if ispc, system('pscp.exe -pw password unet_predict_rtw.mk username@targetname:targetDir/unet_pre
if ispc, system('pscp.exe -pw password input_data.txt username@targetname:targetDir/unet_predict/
if ispc, system('pscp.exe -pw password makefile_unet_arm_generic.mk username@targetname:targetDir
```

Build the Library on the Target Hardware

To build the library on the target hardware, execute the generated makefile on the ARM hardware.

Make sure that you set the environment variables `ARM_COMPUTELIB` and `LD_LIBRARY_PATH` on the target hardware. See “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder). The `ARM_ARCH` variable is used in Makefile to pass compiler flags based on the ARM Architecture. The `ARM_VER` variable is used in Makefile to compile the code based on the version of the ARM Compute library.

On the Linux host platform, run this command to build the library:

```
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetDir/unet_predict/
```

On the Windows host platform, run this command to build the library:

```
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetDir/unet_predict/
```

Create Executable on the Target

In these commands, replace `targetDir` with the destination folder where the library is generated. The variables `height`, `width`, and `channels` represent the dimensions of the input data.

`main_unet_arm_generic.cpp` is the C++ main wrapper file which invokes the segmentationUnetARM function and passes the input image to it. Build the library with the wrapper file to create the executable.

On the Linux host platform, to create the executable, run these commands:

```
if isunix, system('sshpass -p password scp main_unet_arm_generic.cpp username@targetname:targetDi
if isunix, system(['sshpass -p password ssh username@targetname "make -C targetDir/unet_predict/
```

On the Windows host platform, to create the executable, run these commands:

```
if ispc, system('pscp.exe -pw password main_unet_arm_generic.cpp username@targetname:targetDir/un
if ispc, system(['plink.exe -l username -pw password targetname "make -C targetDir/unet_predict/
```

Run the Executable on the Target Hardware

Run the Executable on the target hardware with the input image file `input_data.txt`.

On the Linux host platform, run this command:

```
if isunix, system('sshpass -p password ssh username@targetname "cd targetDir/unet_predict/; ./une
```

On the Windows host platform, run this command:

```
if ispc, system('plink.exe -l username -pw password targetname "cd targetDir/unet_predict/; ./une
```

The `unet` executable accepts the input data. Because of the large size of `input_data` (2001x2001x7), it is easier to process the input image in patches. The executable splits the input image into multiple patches, each corresponding to network input size. The executable performs prediction on the pixels in one particular patch at a time and then combines all the patches together.

Transfer the Output from Target Hardware to MATLAB

Copy the generated output file `output_data.txt` back to the current MATLAB session. On the Linux platform, run:

```
if isunix, system('sshpass -p password scp username@targetname:targetDir/unet_predict/output_data
```

To perform the same action on the Windows platform, run:

```
if ispc, system('pscp.exe -pw password username@targetname:targetDir/unet_predict/output_data.tx
```

Store the output data in the variable `segmentedImage`:

```
segmentedImage = uint8(importdata('output_data.txt'));
segmentedImage = reshape(segmentedImage,[height,width]);
```

To extract only the valid portion of the segmented image, multiply it by the mask channel of the input data.

```
segmentedImage = uint8(input_data(:,:,7)~=0) .* segmentedImage;
```

Remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImageCodegen = medfilt2(segmentedImage,[5,5]);
```

Display U-Net Segmented data

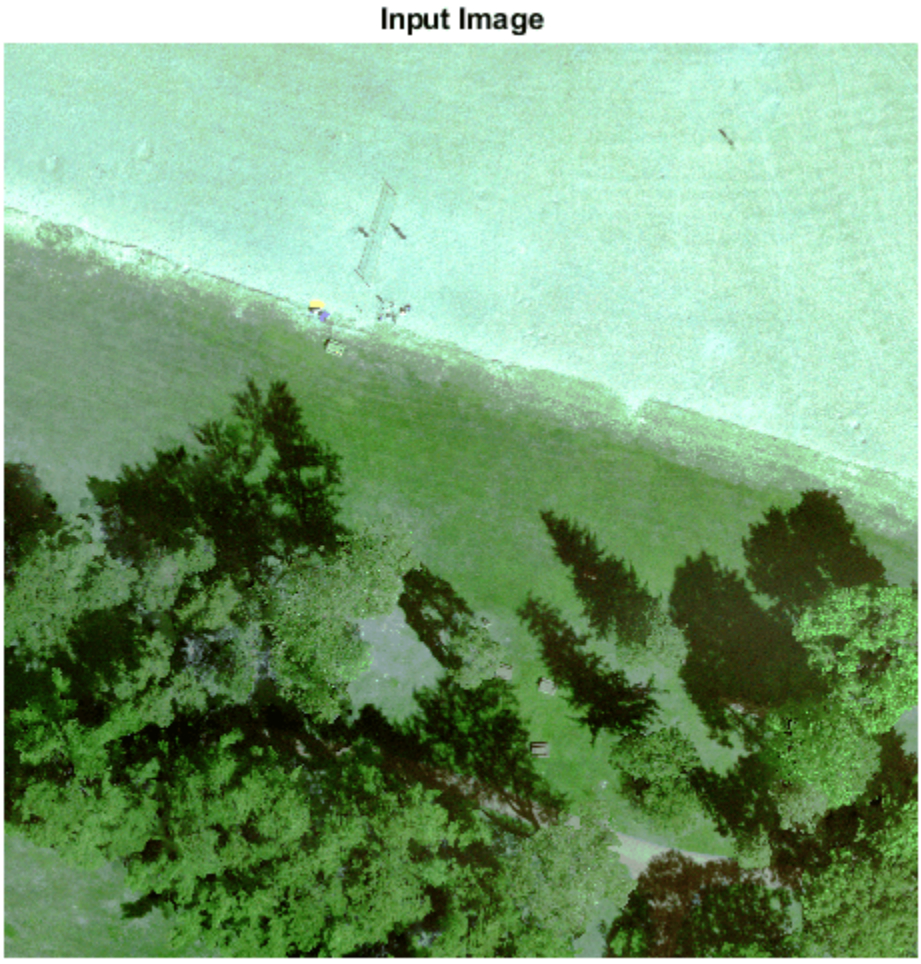
This line of code creates a vector of the class names.

```
classNames = net.Layers(end).Classes;
disp(classNames);
```

Overlay the labels on the segmented RGB test image and add a color bar to the segmented image.

Display input data

```
figure(1);
imshow(histeq(input_data(:,:,1:3)));
title('Input Image');
```



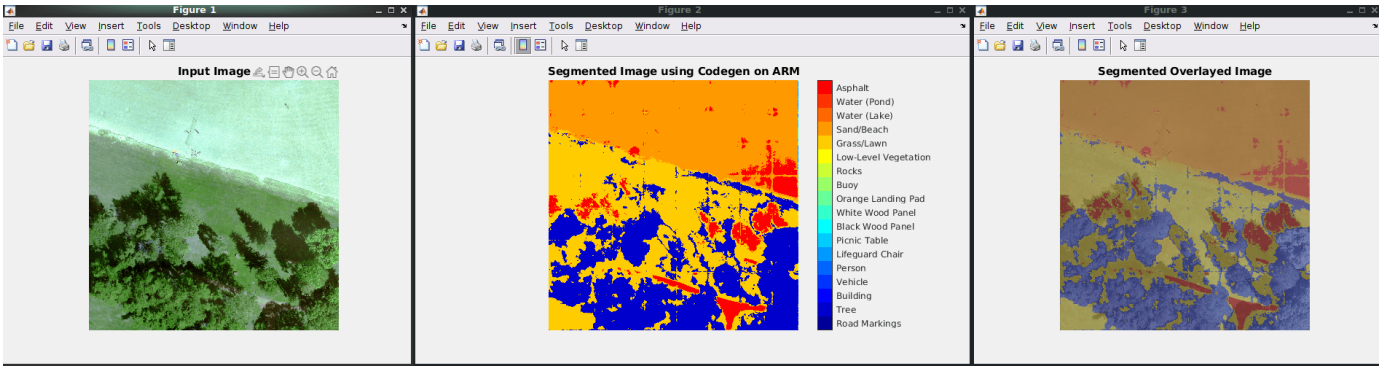

```
cmap = jet(numel(classNames));
segmentedImageOut = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmentedI
figure(2);
imshow(segmentedImageOut);
```

Display segmented data

```
title('Segmented Image using Codegen on ARM');
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','n
colormap(cmap)
```

Display segmented overlay Image

```
segmentedImageOverlay = labeloverlay(imadjust(input_data(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmen
figure(3);
imshow(segmentedImageOverlay);
title('Segmented Overlayed Image');
```



References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597,* 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

[3] Reference Input Data used is part of the Hamlin Beach State Park data. The following steps can be used to download the data for further evaluation.

```
if ~exist(fullfile(pwd,'data'))
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware` | `packNGo`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
- “Code Generation for Deep Learning on ARM Targets” (MATLAB Coder)
- “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox)

Code Generation for LSTM Network on Raspberry Pi

This example shows how to generate code for a pretrained long short-term memory (LSTM) network that uses the ARM® Compute Library and deploy the code on a Raspberry Pi™ target. In this example, the LSTM network predicts the Remaining Useful Life (RUL) of a machine. The network takes as input time series data sets that represent various sensors in the engine. The network returns the Remaining Useful Life of an engine, measured in cycles, as its output.

This example uses the Turbofan Engine Degradation Simulation Data Set as described in [1]. This data set contains 100 training observations and 100 test observations. The training data contains simulated time series data for 100 engines. Each sequence has 17 features, varies in length, and corresponds to a full run to failure (RTF) instance. The test data contains 100 partial sequences and corresponding values of the Remaining Useful Life at the end of each sequence.

This example uses a pretrained LSTM network. For more information on how to train an LSTM network, see the example “Sequence Classification Using Deep Learning” on page 4-2.

This example demonstrates two different approaches for performing prediction by using an LSTM network:

- The first approach uses a standard LSTM network and runs inference on a set of time series data.
- The second approach leverages the stateful behavior of the same LSTM network. In this method, you pass a single timestep of data at a time, and have the network update its state at each time step.

This example uses the PIL based workflow to generate a MEX function, which in turn calls the executable generated in the target hardware from MATLAB.

Notes:

- The code lines in this example are commented out. Uncomment them before you run the example.
- The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware and Software” (MATLAB Coder).
- This example is not supported in MATLAB Online.

Prerequisites

- MATLAB® Coder™
- Embedded Coder®
- Deep Learning Toolbox™
- MATLAB Coder Interface for Deep Learning Libraries. To install this support package, use the Add-On Explorer.
- MATLAB Support Package for Raspberry Pi Hardware. To install this support package, use the Add-On Explorer.
- Raspberry Pi hardware
- ARM Compute Library (on the target ARM hardware)
- Environment variables for the compilers and libraries. For setting up the environment variables, see “Environment Variables” (MATLAB Coder).

Set Up a Code Generation Configuration Object for a Static Library

To generate a PIL MEX function for a specified entry-point function, create a code configuration object for a static library and set the verification mode to 'PIL'. Set the target language to C++.

```
% cfg = coder.config('lib', 'ecoder', true);
% cfg.VerificationMode = 'PIL';
% cfg.TargetLang = 'C++';
```

Set Up a Configuration Object for Deep Learning Code Generation

Create a `coder.ARMNEONConfig` object. Specify the Compute Library version. For this example, suppose that the ARM Compute Library in the Raspberry Pi hardware is version 19.05.

```
% dlcfg = coder.DeepLearningConfig('arm-compute');
% dlcfg.ArmComputeVersion = '19.05';
```

Set the `DeepLearningConfig` property of the code generation configuration object to the deep learning configuration object.

```
% cfg.DeepLearningConfig = dlcfg;
```

Create a Connection to the Raspberry Pi

Use the MATLAB Support Package for Raspberry Pi Support Package function, `raspi`, to create a connection to the Raspberry Pi. In the following code, replace:

- `raspiname` with the name of your Raspberry Pi
- `username` with your user name
- `password` with your password

```
% r = raspi('raspiname','username','password');
```

Configure Code Generation Hardware Parameters for Raspberry Pi

Create a `coder.Hardware` object for Raspberry Pi and attach it to the code generation configuration object.

```
% hw = coder.hardware('Raspberry Pi');
% cfg.Hardware = hw;
```

First Approach: Generate PIL MEX Function for LSTM Network

In this approach, you generate code for the entry-point function `rul_lstmnet_predict`.

The rul_lstmnet_predict.m entry-point function takes an entire time series data set as an input and passes it to the network for prediction. Specifically, the function uses the LSTM network that is trained in the example “Sequence Classification Using Deep Learning” on page 4-2. The function loads the network object from the `rul_lstmnet.mat` file into a persistent variable and reuses this persistent object in subsequent prediction calls. A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` function.

```
type('rul_lstmnet_predict.m')
```

```
function out =  rul_lstmnet_predict(in) %#codegen

% Copyright 2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rul_lstmnet.mat');
end


out = mynet.predict(in);
```

To generate code by using the `codegen` (MATLAB Coder) command, use the `coder.typeof` (MATLAB Coder) function to specify the type and size of the input argument to the entry-point function. In this example, the input is of double data type with a feature dimension value of 17 and a variable sequence length. Specify the sequence length as variable-size to perform prediction on an input sequence of any length.

```
% matrixInput = coder.typeof(double(0),[17 Inf],[false true]);
```

Run the codegen command to generate a PIL based mex function `rul_lstmnet_predict_pil` on the host platform.

```
% codegen -config cfg rul_lstmnet_predict -args {matrixInput} -report
```

Run Generated PIL MEX Function on Test Data

Load the MAT-file `RULTestData`. This MAT-file stores the variables `XTest` and `YTest` that contain sample timeseries of sensor readings on which you can test the generated code. This test data is taken from the example “Sequence Classification Using Deep Learning” on page 4-2 after data pre-processing.

```
load RULTestData;
```

The `XTest` variable contains 100 input observations. Each observation has 17 features with varying sequence length.

```
XTest(1:5)
```

```
ans=5×1 cell array
    {17×31  double}
    {17×49  double}
    {17×126 double}
    {17×106 double}
    {17×98  double}
```

The `YTest` variable contains 100 output observations that correspond to the `XTest` input variable. Each output observation is a Remaining Useful Life (RUI) value, measured in cycles, for each time step data in entire sequence.

```
YTest(1:5)
```

```
ans=5×1 cell array
    {[
    {[
    {[150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150 150
```

```
    {[                                                                  150 150 150 150 150 150
    {[                                                                                          ]
```

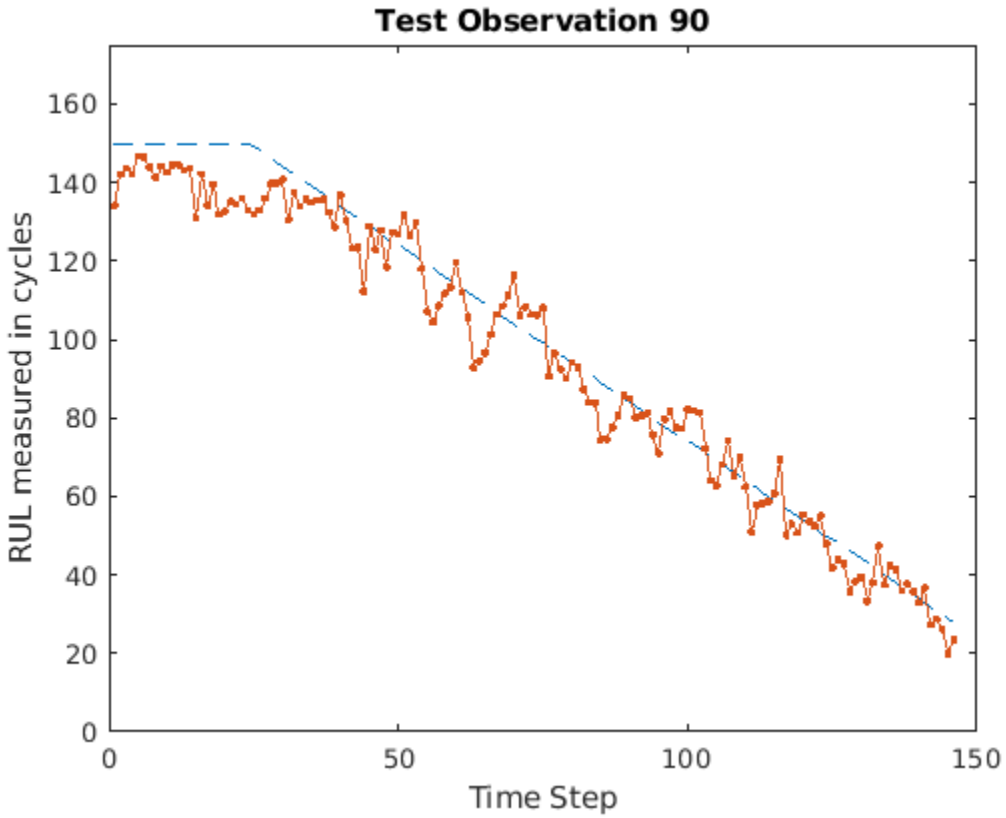
Run the generated MEX function `rul_lstmnet_predict_pil` on a random test data set.

```
% idx = randperm(numel(XTest), 1);
% inputData = XTest{idx};

% YPred1 = rul_lstmnet_predict_pil(inputData);
```

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
% figure('Name', 'Standard LSTM', 'NumberTitle', 'off');
%
% plot(YTest{idx},'--')
% hold on
% plot(YPred1,'.-')
% hold off
%
% ylim([0 175])
% title("Test Observation " + idx)
% xlabel("Time Step")
% ylabel("RUL measured in cycles")
```



Clear PIL

```
% clear rul_lstmnet_predict_pil;
```

Second Approach: Generate PIL MEX Function for Stateful LSTM Network

Instead of passing the entire timeseries data all at once to `predict`, you can run prediction by streaming the input data segment-wise by using the `predictAndUpdateState` function.

The entry-point function rul_lstmnet_predict_and_update.m accepts a single-timestep input and processes it by using the `predictAndUpdateState` function. `predictAndUpdateState` returns a prediction for the input timestep and updates the network so that subsequent parts of the input are treated as subsequent timesteps of the same sample.

```
type('rul_lstmnet_predict_and_update.m')
```

```
function out = rul_lstmnet_predict_and_update(in) %#codegen

% Copyright 2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rul_lstmnet.mat');
end

[mynet, out] = predictAndUpdateState(mynet, in);

end
```

Create the input type for the `codegen` command. Because `rul_lstmnet_predict_and_update` accepts a single timestep data in each call, specify the input type `matrixInput` to have a fixed sequence length of 1 instead of a variable sequence length.

```
% matrixInput = coder.typeof(double(0),[17 1]);
```

Run the `codegen` command to generate PIL based mex function `rul_lstmnet_predict_and_update_pil` on the host platform.

```
% codegen -config cfg rul_lstmnet_predict_and_update -args {matrixInput} -report
```

Run Generated PIL MEX Function on Test Data

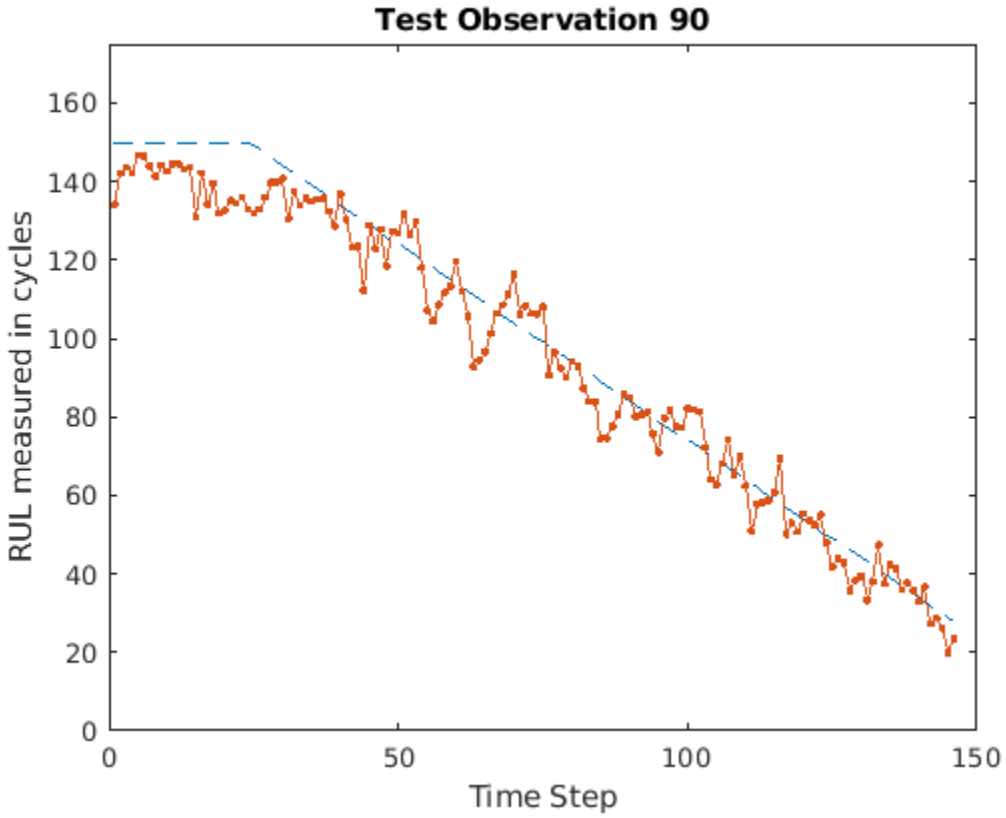
```
% Run generated MEX function(|rul_lstmnet_predict_and_update_pil|) for each
% time step data in the inputData sequence.

% sequenceLength = size(inputData,2);
% YPred2 = zeros(1, sequenceLength);
% for i=1:sequenceLength
%     inTimeStep = inputData(:,i);
%     YPred2(:, i) = rul_lstmnet_predict_and_update_pil(inTimeStep);
% end
```

After you pass all timesteps, one at a time, to the `rul_lstmnet_predict_and_update` function, the resulting output is the same as that in the first approach in which you passed all inputs at once.

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
% figure('Name', 'Statefull LSTM', 'NumberTitle', 'off');
%
%
% plot(YTest{idx},'--')
% hold on
% plot(YPred2,'.-')
% hold off
%
% ylim([0 175])
% title("Test Observation " + idx)
% xlabel("Time Step")
% ylabel("RUL measured in cycles")
```



Clear PIL

```
% clear rul_lstmnet_predict_and_update_pil;
```

References

[1] Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In Prognostics and Health Management, 2008. PHM 2008. International Conference on, pp. 1-9. IEEE, 2008.

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware` | `predictAndUpdateState`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
- “Code Generation for Deep Learning on ARM Targets” (MATLAB Coder)
- “Sequence Classification Using Deep Learning” on page 4-2

Code Generation for LSTM Network That Uses Intel MKL-DNN

This example shows how to generate code for a pretrained long short-term memory (LSTM) network that uses the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).This example generates a MEX function that makes predictions for each step of an input timeseries. The example demonstrates two approaches. The first approach uses a standard LSTM network. The second approach leverages the stateful behavior of the same LSTM network. This example uses textual descriptions of factory events that can be classified into one of these four categories: Electronic Failure, Leak, Mechanical Failure, and Software Failure. The example uses a pretrained LSTM network. For more information on training a network, see the “Classify Text Data Using Deep Learning” (Text Analytics Toolbox).

Third-Party Prerequisites

- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- For a list of processors that support the MKL-DNN library, see MKLDNN CPU Support
- For more information on the supported versions of the compilers and libraries, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)

This example is supported on Mac®, Linux® and Windows® platforms and not supported for MATLAB Online.

Prepare Input

Load the `wordEncoding` MAT-file. This MAT-file stores the words encoded as numerical indices. This encoding was performed during the training of the network. For more information, see “Classify Text Data Using Deep Learning” (Text Analytics Toolbox).

```
load("wordEncoding.mat");
```

Create a string array containing the new reports to classify the event type.

```
reportsNew = [ ...
    "Coolant is pooling underneath sorter."
    "Sorter blows fuses at start up."
    "There are some very loud rattling sounds coming from the assembler."
    "At times mechanical arrangement software freezes."
    "Mixer output is stuck."];
```

Tokenize the input string by using the `preprocessText` function.

```
documentsNew = preprocessText(reportsNew);
```

Use the `doc2sequence` (Text Analytics Toolbox) function to convert documents to sequences.

```
XNew = doc2sequence(enc,documentsNew);
labels = categorical({'Electronic Failure', 'Leak', 'Mechanical Failure', 'Software Failure'});
```

The `lstm_predict` Entry-Point Function

A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence. The `lstm_predict.m` entry-point function takes an input sequence and passes it to a trained LSTM network for prediction. Specifically, the function uses the LSTM network that is trained in the example “Classify Text Data Using Deep Learning” (Text Analytics Toolbox). The function loads the network object from the `textClassifierNetwork.mat` file into a

persistent variable and then performs prediction. On subsequent calls, the function reuses the persistent object.

```
type('lstm_predict.m')
```

```
function out = lstm_predict(in)
%#codegen

%   Copyright 2020 The MathWorks, Inc.

    persistent mynet;

    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork('textClassifierNetwork.mat');
    end

    out = predict(mynet, in);
end
```

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` function.

Generate MEX

To generate code, create a code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a MKL-DNN deep learning configuration object. Assign it to the `DeepLearningConfig` property of the code configuration object.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Use the `coder.typeof` (MATLAB Coder) function to specify the type and size of the input argument to the entry-point function. In this example, the input is of double data type with a feature dimension value of 1 and a variable sequence length.

```
matrixInput = coder.typeof(double(0),[1 Inf],[false true]);
```

Generate a MEX function by running the `codegen` (MATLAB Coder) command.

```
codegen -config cfg lstm_predict -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run Generated MEX

Call `lstm_predict_mex` on the first observation.

```
YPred1 = lstm_predict_mex(XNew{1});
```

`YPred1` contains the probabilities for the four classes. Find the predicted class by calculating the index of the maximum probability.

```
[~, maxIndex] = max(YPred1);
```

Associate the indices of max probability to the corresponding label. Display the classification. From the results, you can see that the network predicted the first event to be a Leak.

```
predictedLabels1 = labels(maxIndex);
disp(predictedLabels1)
```

```
     Leak
```

Generate MEX that Accepts Multiple Observations

If you want to perform prediction on many observations at once, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell array, and each cell must contain one observation. The sequence lengths of the inputs might vary. In this example, `XNew` contains five observations. To generate a MEX function that can accept `XNew` as input, specify the input type to be a 5-by-1 cell array. Specify that each cell be of the same type as `matrixInput`.

```
matrixInput = coder.typeof(double(0),[1 Inf],[false true]);
cellInput = coder.typeof({matrixInput}, [5 1]);
codegen -config cfg lstm_predict -args {cellInput} -report
```

```
Code generation successful: View report
```

Run the generated MEX function with `XNew` as input.

```
YPred2 = lstm_predict_mex(XNew);
```

`YPred2` is 5-by-4 cell array. Find the indices that have maximum probability for each of the five inputs and classify them.

```
[~, maxIndex] = max(YPred2, [], 2);
predictedLabels2 = labels(maxIndex);
disp(predictedLabels2)
```

```
     Leak      Mechanical Failure      Mechanical Failure      Software Failure      Electronic F
```

Generate MEX with Stateful LSTM

Instead of passing the entire timeseries to `predict` in a single step, you can run prediction on an input by streaming in one timestep at a time and using the function `predictAndUpdateState`. This function accepts an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account.

The entry-point function `lstm_predict_and_update.m` accepts a single-timestep input and processes the input using the `predictAndUpdateState` function. The `predictAndUpdateState` function returns a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps, one at a time, the resulting output is identical to the case where all timesteps were passed in as a single input.

```
type('lstm_predict_and_update.m')
```

```
function out = lstm_predict_and_update(in)
%#codegen

%   Copyright 2020 The MathWorks, Inc.

    persistent mynet;

    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork('textClassifierNetwork.mat');
```

```
    end

    [mynet, out] = predictAndUpdateState(mynet,in);
end
```

Generate code for `lstm_predict_and_update`. Because this function accepts a single timestep at each call, specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[1 1]);
codegen -config cfg lstm_predict_and_update -args {matrixInput} -report
```

```
Code generation successful: View report
```

Run the generated MEX on the first observation.

```
sequenceLength = size(XNew{1},2);
for i=1:sequenceLength
    inTimeStep = XNew{1}(:,i);
    YPred3 = lstm_predict_and_update_mex(inTimeStep);
end
clear mex;
```

Find the index that has the highest probability and map it to the labels.

```
[~, maxIndex] = max(YPred3);
predictedLabels3 = labels(maxIndex);
disp(predictedLabels3)
```

```
     Leak
```

See Also

`coder.DeepLearningConfig` | `doc2sequence` | `coder.typeof` | `codegen`

More About

- “Classify Text Data Using Deep Learning” (Text Analytics Toolbox)
- “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)

Cross Compile Deep Learning Code for ARM Neon Targets

This example shows how to cross-compile the generated deep learning code to create a library or an executable, and then deploy the library or executable on an ARM® target such as Hikey 960 or Rock 960. This example uses the `codegen` command.

Cross compiling the deep learning code for ARM® targets involves these steps:

- Configure the installed cross-compiler toolchain to perform compilation on the host MATLAB®. The compilation happens when you run the `codegen` command in MATLAB in the host computer.
- Use the `codegen` command to build the generated code and create a library or an executable on the host computer.
- Copy the generated library or executable and other supporting files to the target hardware. If you generate a library on the host computer, compile the copied makefile on the target to create an executable.
- Run the generated executable on the target ARM hardware.

You can use this workflow for any ARM Neon target that supportes the Neon|SIMD instruction set. This example is supported only for host Linux® platforms.

Prerequisites

- ARM processor that supports the Neon|SIMD extension
- ARM Compute Library (on the host computer)
- MATLAB® Coder™
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™
- The support package Deep Learning Toolbox Model for Inception-v3 Network
- Image Processing Toolbox™
- For deployment on armv7 (32 bit Arm Architecture) target, GNU/GCC `g++-arm-linux-gnueabihf` toolchain
- For deployment on armv8 (64 bit Arm Architecture) target, GNU/GCC `g++-aarch64-linux-gnu` toolchain
- Environment variables for the cross compilers and libraries

For information about how to install the cross-compiler toolchain and set up the associated environment variable, see “Cross-Compile Deep Learning Code That Uses ARM Compute Library” (MATLAB Coder).

The ARM Compute library version that this example uses might not be the latest version that code generation supports. For information about supported versions of libraries and about environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder).

The code lines in this example are commented out. Uncomment them before you run the example.

This example in not supported in MATLAB Online.

The `inception_predict_arm` Entry-Point Function

This example uses the Inception-V3 image classification network. A pretrained Inception-V3 network for MATLAB is available in the support package Deep Learning Toolbox Model for Inception-V3

Network. The `inception_predict_arm` entry-point function loads the Inception-V3 network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

```
type inception_predict_arm
```

```
function out = inception_predict_arm(in)

persistent net;
if isempty(net)
    net = coder.loadDeepLearningNetwork('inceptionv3','inceptionv3');
end

out = net.predict(in);

end
```

Set up a Deep Learning Configuration Object

Create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute library.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmComputeVersion = '19.05';
```

For classifying the input image `peppers.png`, convert the image to a text file.

```
% generateImagetoTxt('peppers.png');
```

First Approach: Create Static Library for Entry-Point Function on Host

In this approach, you first cross-compile the generated code to create a static library on the host computer. You then transfer the generated static library, the ARM Compute library files, the makefile, and other supporting files to the target hardware. You run the makefile on the target hardware to generate the executable. Finally, you run the executable on the target hardware.

Set Up a Code Generation Configuration Object

Create a code generation configuration object for a static library. Specify the target language as C++.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
```

Attach the deep learning configuration object to the code generation configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Configure the Cross-Compiler Toolchain

Configure the cross-compiler toolchain based on the ARM Architecture of the target device.

```
%   cfg.Toolchain =  'Linaro AArch64 Linux v6.3.1';% When the Arm Architecture is armv8
```

```
%   cfg.Toolchain =  'Linaro AArch32 Linux v6.3.1';% When the Arm Architecture is armv7
```

Generate Static Library on Host Computer by Using codegen

Use the `codegen` command to generate code for the entry-point function, build the generated code, and create static library for the target ARM architecture.

```
% codegen -config cfg inception_predict_arm -args {ones(299,299,3,'single')} -d arm_compute_cc_li
```

Copy the Generated Cross-Compiled Static Library to Target hardware

Copy the static library, the bin files, and the header files from the generated folder `arm_compute_cc_lib` to the target ARM hardware. In this code line and other code lines that follow, replace:

- password with your password
- username with your username
- hostname with the name of your device
- targetDir with the destination folder for the files

```
% system('sshpass -p password scp -r arm_compute_cc_lib/*.bin arm_compute_cc_lib/*.lib arm_comput
```

Copy the ARM Compute Library Files to Target Hardware

The executable uses the ARM Compute library files during runtime. The target board does not need header files while generating the executable and running the executable. Copy the library to the desired path.

```
% system(['sshpass -p password scp -r ' fullfile(getenv('ARM_COMPUTELIB'),'lib') ' username@hostn
```

Copy Supporting Files to Target Hardware

Copy these files to the target ARM hardware:

- Makefile `Makefile_Inceptionv3` to generate executable from static library.
- Input Image `inputimage.txt` that you want to classify.
- The text file `synsetWords.txt` that contains the ClassNames returned by `net.Layers(end).Classes`
- The main wrapper file `main_inception_arm.cpp` that calls the code generated for the `inception_predict_arm` function.

```
% system('sshpass -p password scp synsetWords.txt ./Makefile_Inceptionv3 ./inputimage.txt ./main_
```

Create the Executable on the Target

Compile the makefile on the target to generate the executable from the static library. This makefile links the static library with the main wrapper file `main_inception_arm.cpp` and generates the executable.

```
% system('sshpass -p password ssh username@hostname "make -C targetDir -f Makefile_Inceptionv3 ar
```

Run the Executable on the Target

Run the generated executable on the target. Make sure to export LD_LIBRARY_PATH that points to the ARM Compute library files while running executable.

```
% system('sshpass -p password ssh username@hostname "export LD_LIBRARY_PATH=targetDir/lib; cd tar
```

Second Approach: Create Executable for Entry-Point function on Host

In this approach, you first cross-compile the generated code to create an executable on the host computer. You then transfer the generated executable, the ARM Compute library files, and other supporting files to the target hardware. Finally, you run the executable on the target hardware.

Set Up a Code Generation Configuration Object

Create a code generation configuration object for an generating an executable. Set the target language as C++.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

Attach the deep learning configuration object to the code generation configuration object.

```
cfg.DeepLearningConfig = dlcfg;
```

Declare the main wrapper file `main_inception_arm.cpp` as the custom source file.

```
cfg.CustomSource = 'main_inception_arm.cpp';
```

Configure the Cross-Compiler Toolchain

Configure the cross-compiler toolchain based on the ARM Architecture of the target device.

```
%   cfg.Toolchain =  'Linaro AArch64 Linux v6.3.1'; % When the Arm Architecture is armv8,

%   cfg.Toolchain =  'Linaro AArch32 Linux v6.3.1';% When the Arm Architecture is armv7,
```

Generate Executable on the Host Computer by Using codegen

Use the `codegen` command to generate code for the entry-point function, build the generated code, and create an executablle for the target ARM architecture.

```
% codegen -config cfg inception_predict_arm -args {ones(299,299,3,'single')} -d arm_compute_cc_ex
```

Copy the Generated Executable to the Target Hardware

Copy the generated executable and the bin files to the target ARM hardware. In this code line and other code lines that follow, replace:

- password with your password
- username with your username
- hostname with the name of your device
- `targetDir` with the destination folder for the files

```
% system('sshpass -p password scp -r arm_compute_cc_exe/*.bin username@hostname:targetDir/');
% system('sshpass -p password scp inception_predict_arm.elf username@hostname:targetDir/');
```

Copy the ARM Compute Library Files to the Target Hardware

The executable uses the ARM Compute library files during runtime. It does not use header files at runtime. Copy the library files to the desired path.

```
% system(['sshpass -p password scp -r ' fullfile(getenv('ARM_COMPUTELIB'),'lib') ' username@hostn
```

Copy Supporting Files to the Target Hardware

Copy these files to the target ARM hardware:

- Input Image `inputimage.txt` that you want to classify.

- The text file `synsetWords.txt` that contains the ClassNames returned by `net.Layers(end).Classes`
- The main wrapper file `main_inception_arm.cpp` that calls the code generated for the `inception_predict_arm` function.

```
% system('sshpass -p password scp synsetWords.txt ./inputimage.txt ./main_inception_arm.cpp user
```

Run the Executable on the Target Hardware

Run the generated executable on the target. Make sure to export LD_LIBRARY_PATH that points to the ARM Compute library files while running executable.

```
% system('sshpass -p password ssh username@hostname "export LD_LIBRARY_PATH=targetDir/lib; cd ta
```

Transfer the Output Data from Target to MATLAB

Copy the generated output back to the current MATLAB session on the host computer.

```
% system('sshpass -p password scp username@hostname:targetDir/out.txt ./');
```

Map Prediction Scores to Labels

Map the top five prediction scores to corresponding labels in the trained network.

```
% outputImage = mapPredictionScores;

% Display the overlayed Image with Classification Scores.

% imshow(outputImage);
```

See Also

`coder.ARMNEONConfig` | `coder.DeepLearningConfig` | `coder.hardware`

More About

- “Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
- “Code Generation for Deep Learning on ARM Targets” (MATLAB Coder)
- “Cross-Compile Deep Learning Code That Uses ARM Compute Library” (MATLAB Coder)

Generate Generic C/C++ Code for Sequence-to-Sequence Regression That Uses Deep Learning

This example demonstrates how to generate plain C/C++ code that does not depend on any third-party deep learning libraries for a long short-term memory (LSTM) network. You generate a MEX function that accepts time series data representing various sensors in an engine. The MEX function then makes predictions for each step of the input timeseries to predict the remaining useful life (RUL) of the engine measured in cycles.

This example uses the Turbofan Engine Degradation Simulation Data Set as described in [1] and a pretrained LSTM network to predict the remaining useful life of an engine. The network was trained on simulated time series sequence data for 100 engines and corresponding values of the remaining useful life at the end of each sequence. Each sequence in this training data has a different length and corresponds to a full run to failure (RTF) instance. For more information on training the network, see the example “Sequence-to-Sequence Regression Using Deep Learning” on page 4-47

Define Entry-Point Function `rulPredict`

The `rulPredict` entry-point function takes an input sequence and passes it to a trained sequence-to-sequence LSTM network for prediction. The function loads the network object from the `rulNetwork.mat` file into a persistent variable and reuses the persistent object on subsequent prediction calls. The LSTM network makes predictions on the partial sequence one time step at a time. At each time step, the network predicts using the value at this time step, and the network state calculated from the previous time steps only. The network updates its state between each prediction. The `predict` function returns a sequence of these predictions. The last element of the prediction corresponds to the predicted RUL for the partial sequence.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` function.

```
type rulPredict.m
```

```
function out = rulPredict(in)
%#codegen

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rulNetwork.mat');
end

% pass in input to predict method
% To prevent the function from adding padding to the data, specify the mini-batch size 1.
out = predict(mynet,in,'MiniBatchSize',1);
```

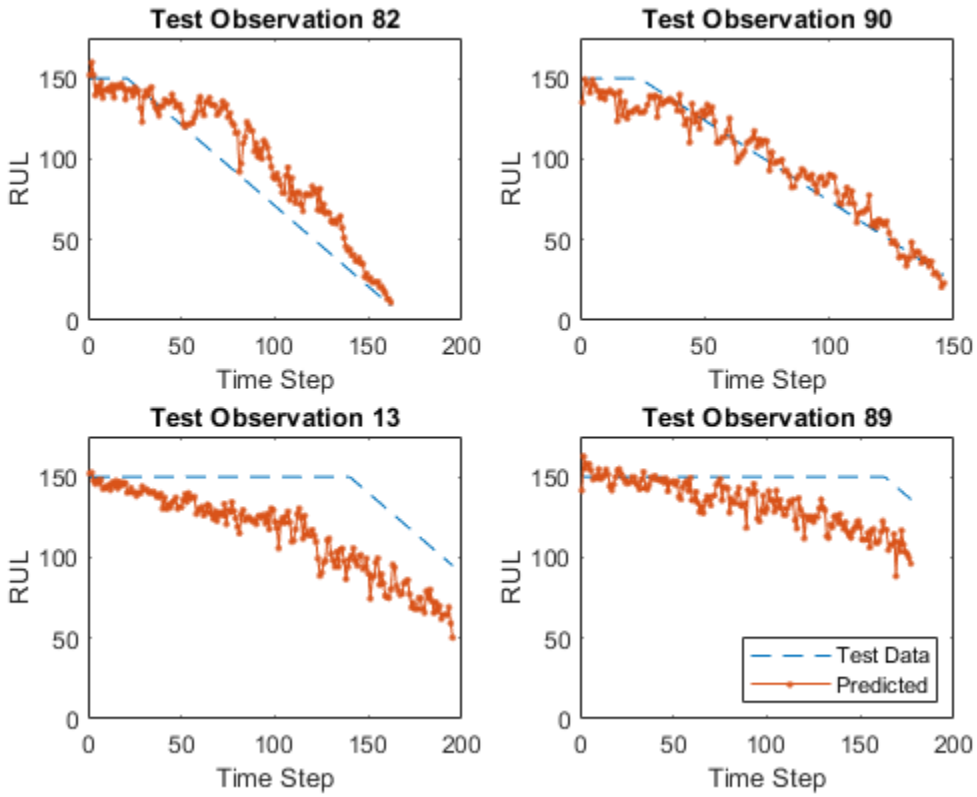
Run `rulPredict` on Test Data

Load the `TurboFanRULValidate` MAT-file. This MAT-file stores the variable `XValidate` that contains sample timeseries data for sensor readings the you use to test the entry-point function in MATLAB. Make predictions on the test data by calling the `rulPredict` method.

```
load TurboFanRULValidate.mat
YPred = rulPredict(XValidate);
```

Visualize some of the predictions in a plot.

```
idx = randperm(numel(YPred),4);
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPred{idx(i)},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted"],'Location','southeast')
```



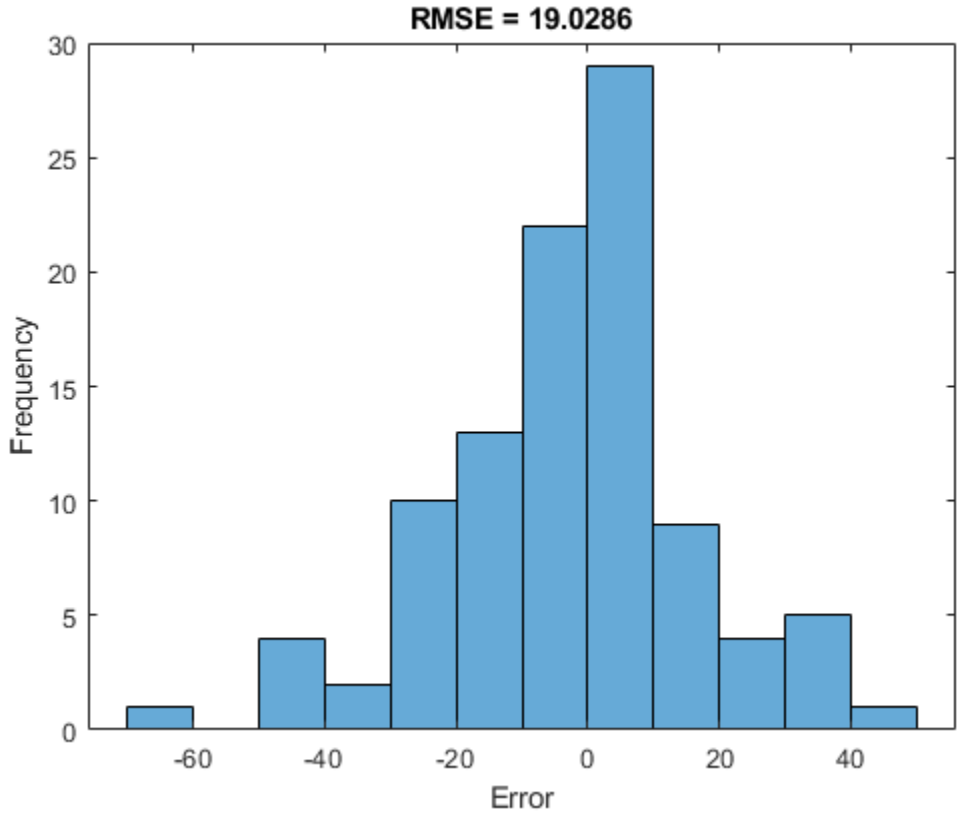
For a given partial sequence, the predicted current RUL is the last element of the predicted sequences. Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
YValidateLast = zeros(1, numel(YValidate));
YPredLast = zeros(1, numel(YValidate));
for i = 1:numel(YValidate)
    YValidateLast(i) = YValidate{i}(end);
    YPredLast(i) = YPred{i}(end);
```

```
end
figure
rmse = sqrt(mean((YPredLast - YValidateLast).^2))
```

```
rmse = 19.0286
```

```
histogram(YPredLast - YValidateLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```



Generate MEX function for `rulPredict`

To generate a MEX function for the `rulPredict` entry-point function, create a code generation configuration object `cfg` for MEX code generation. Create a deep learning configuration object that specifies that no target library is required and attach this deep learning configuration object to `cfg`.

```
cfg = coder.config('mex');
cfg.DeepLearningConfig = coder.DeepLearningConfig('TargetLibrary','none');
```

By default, the target language is set to C. If you want to generate C++ code, explicitly set the target language to C++.

Use the `coder.typeof` function to create the input type for the entry-point function `rulPredict` that you use with the `-args` option in the `codegen` command.

The data `XValidate` contains 100 observations where each observation is of double data type with a feature dimension value of 17 and a variable sequence length. In order to perform prediction on

several such observations in a single function call, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell array, and each cell must contain one observation. Each observation must have the same feature dimension, but the sequence lengths might vary as is the case for `XValidate`. Specifying the sequence length as variable-size enables us to perform prediction on an input sequence of any length.

```
matrixInput = coder.typeof(0, [17 Inf],[false true]); % input type for a single observation
cellInput = coder.typeof({matrixInput}, [100 1]); % input type for multiple observations
```

Run the codegen command. Specify the input type to be `cellInput`.

```
codegen -config cfg rulPredict -args {cellInput} -report
```

```
Code generation successful: To view the report, open('codegen\mex\rulPredict\html\report.mldatx')
```

By default for MEX code generation, the generated code calls into BLAS library for matrix operations and uses OpenMP library (if the compiler supports OpenMP) so that the any parallelizable for loops in the MEX can run on multiple threads leading to better execution performance. While OpenMP is enabled by default for standalone code generation, you will have to provide a custom BLAS callback to indicate to MATLAB Coder ™ that you want to generate BLAS calls for matrix operations following the steps mentioned in “Speed Up Matrix Operations in Generated Standalone Code by Using BLAS Calls” (MATLAB Coder).

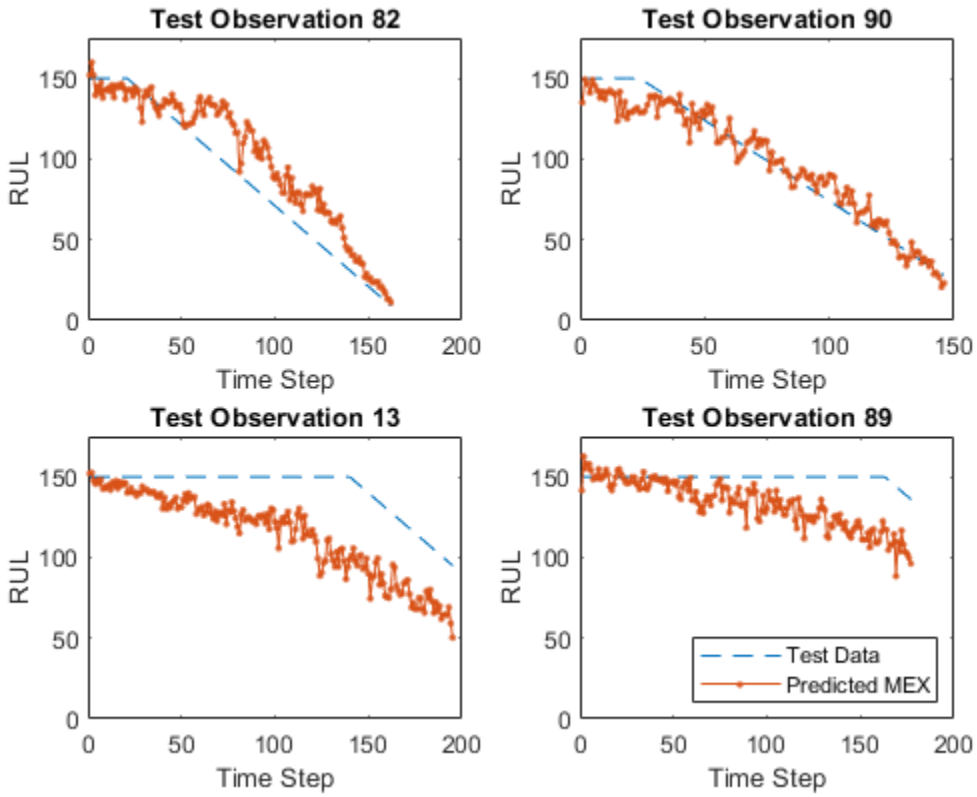
Run Generated MEX Function on Test Data

Make predictions on the test data by calling the generated MEX function `rulPredict_mex`.

```
YPredMex = rulPredict_mex(XValidate);
```

You can visualize the same predictions as before in a plot.

```
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPredMex{idx(i)},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted MEX"],'Location','southeast')
```

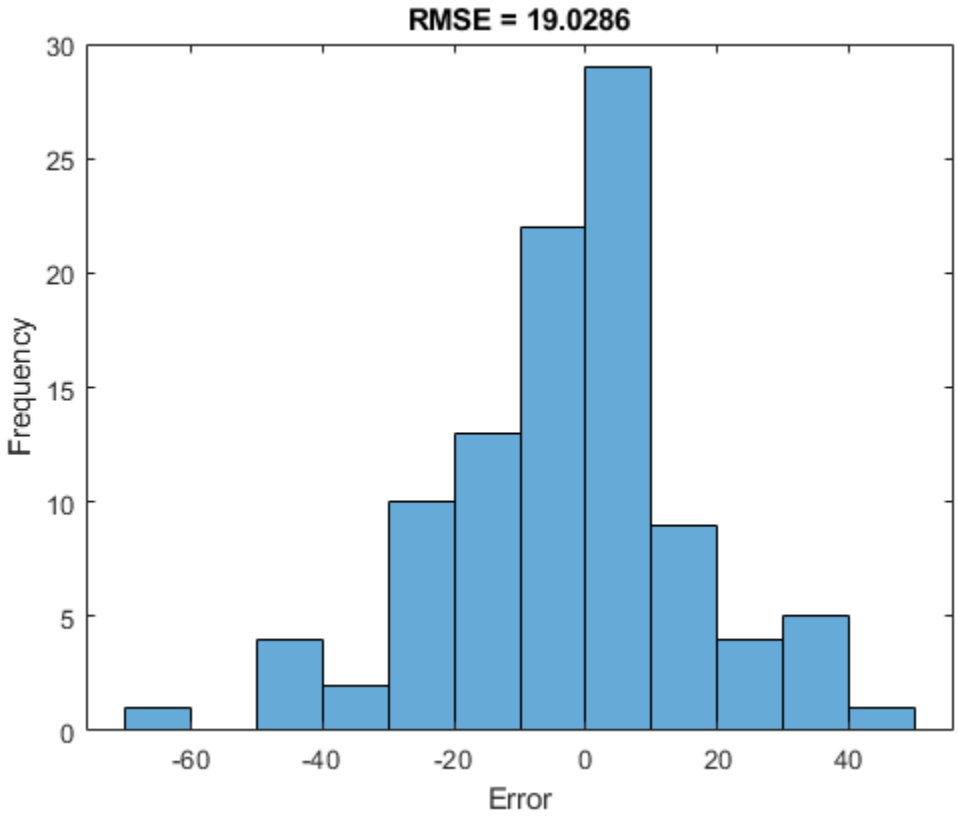


Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
YPredLastMex = zeros(1, numel(YValidate));
for i = 1:numel(YValidate)
    YPredLastMex(i) = YPredMex{i}(end);
end
figure
rmse = sqrt(mean((YPredLastMex - YValidateLast).^2))
```

```
rmse = 19.0286
```

```
histogram(YPredLastMex - YValidateLast)
title("RMSE = " + rmse)
ylabel("Frequency")
xlabel("Error")
```



Generate MEX function with Stateful LSTM

Instead of passing the entire timeseries to predict in one step, you can make predictions one time step at a time by using `predictAndUpdateState`. This is useful when you have the values of the time steps arriving in a stream. The `predictAndUpdateState` function takes in an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account. Usually, it is faster to make predictions on full sequences when compared to making predictions one time step at a time.

The entry-point function `rulPredictAndUpdate` takes in a single-timestep input and processes the input using the `predictAndUpdateState` function. `predictAndUpdateState` outputs a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps one at a time, the resulting output is the same as if all timesteps were passed in as a single input.

```
type rulPredictAndUpdate.m
```

```
function out = rulPredictAndUpdate(in)
%#codegen

% Copyright 2020 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('rulNetwork.mat');
end
```

```
% pass in input to predictAndUpdateState method
[mynet, out] = predictAndUpdateState(mynet, in);
```

Run codegen on this new entry-point function. Since we are taking in a single timestep each call, we specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[17 1]);
codegen -config cfg rulPredictAndUpdate -args {matrixInput} -report
```

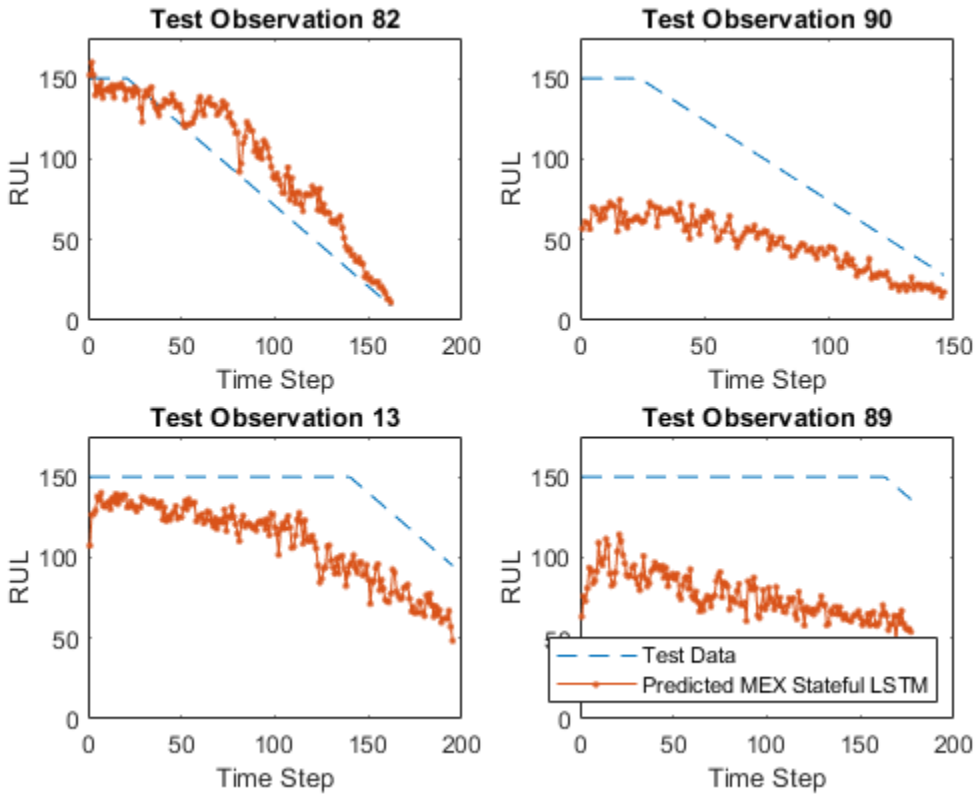
```
Code generation successful: To view the report, open('codegen\mex\rulPredictAndUpdate\html\report
```

Make predictions on the test data by calling the `rulPredictAndUpdate` function in MATLAB and the generated MEX function `rulPredictAndUpdate_mex`.

```
YPredStatefulMex = cell(numel(idx), 1);
for iSample = 1:numel(idx)
    sample = XValidate{idx(iSample)};
    numTimeStepsTest = size(sample, 2);
    for iStep = 1:numTimeStepsTest
        YPredStatefulMex{iSample}(1, iStep) = rulPredictAndUpdate_mex(sample(:, iStep));
    end
end
```

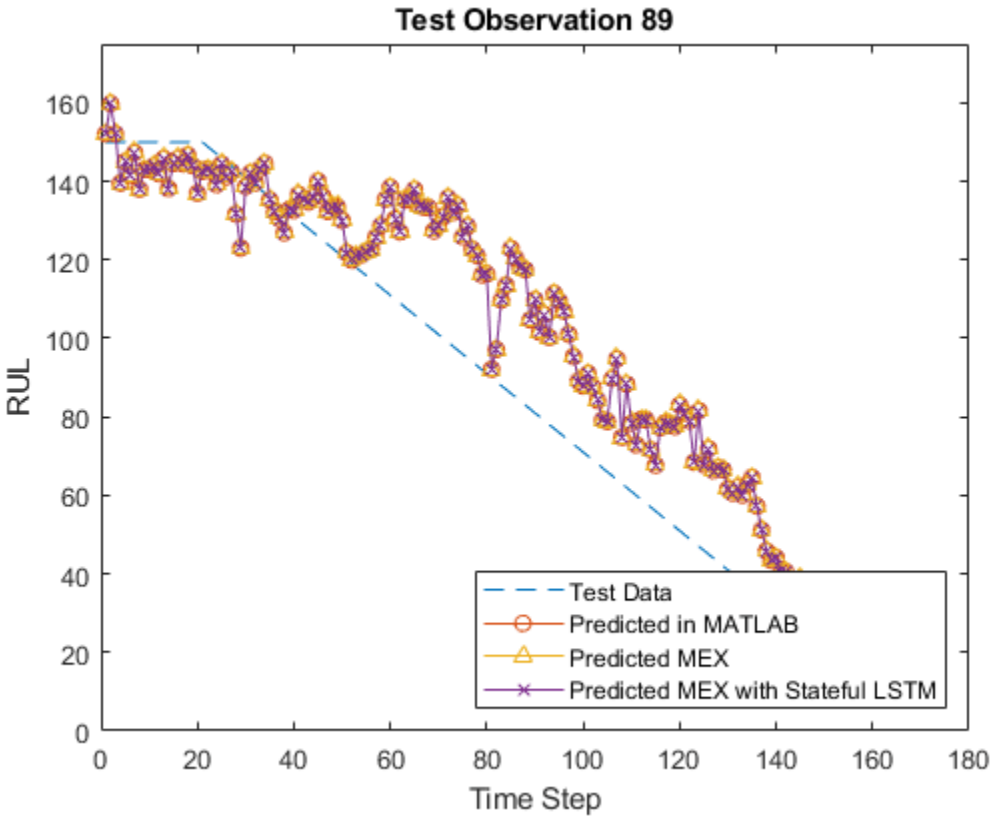
Once again you can visualize the predictions for stateful MEX as before in a plot.

```
figure
for i = 1:numel(idx)
    subplot(2,2,i)

    plot(YValidate{idx(i)},'--')
    hold on
    plot(YPredStatefulMex{i},'.-')
    hold off

    ylim([0 175])
    title("Test Observation " + idx(i))
    xlabel("Time Step")
    ylabel("RUL")
end
legend(["Test Data" "Predicted MEX Stateful LSTM"],'Location','southeast')
```



Finally you can also visualize the results for the two different MEX functions along with the MATLAB prediction in a plot for any particular sample.

```
figure()
sampleIdx = idx(1);
plot(YValidate{sampleIdx},'--')
hold on
plot(YPred{sampleIdx},'o-')
plot(YPredMex{sampleIdx},'^-')
plot(YPredStatefulMex{1},'x-')
hold off

ylim([0 175])
title("Test Observation " + idx(i))
xlabel("Time Step")
ylabel("RUL")
legend(["Test Data" "Predicted in MATLAB" "Predicted MEX" "Predicted MEX with Stateful LSTM"],'Lo
```



References

1 Saxena, Abhinav, Kai Goebel, Don Simon, and Neil Eklund. "Damage propagation modeling for aircraft engine run-to-failure simulation." In *Prognostics and Health Management, 2008. PHM 2008. International Conference on*, pp. 1-9. IEEE, 2008.

See Also

`coder.DeepLearningConfig` | `codegen` | `coder.config`

More About

- “Generate Generic C/C++ Code for Deep Learning Networks” (MATLAB Coder)

Quantize Residual Network Trained for Image Classification and Generate CUDA Code

This example shows how to quantize the learnable parameters in the convolution layers of a deep learning neural network that has residual connections and has been trained for image classification with CIFAR-10 data.

Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Most neural networks that you create and train using Deep Learning Toolbox™ use single-precision floating point data types. Even small networks require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and less memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

In this example, you use the Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types.

The network in this example has been trained for image classification with CIFAR-10 data.

Residual connections are a popular element in convolutional neural network architectures. A residual network is a type of DAG network that has residual (or shortcut) connections that bypass the main network layers. Residual connections enable the parameter gradients to propagate more easily from the output layer to the earlier layers of the network, which makes it possible to train deeper networks. This increased network depth can result in higher accuracies on more difficult tasks. For information on the network architecture and training, see Train Residual Network for Image Classification.

To run this example, you must have the products required to quantize and deploy a deep learning network to a GPU environment. For information on these products, see Quantization Workflow Prerequisites.

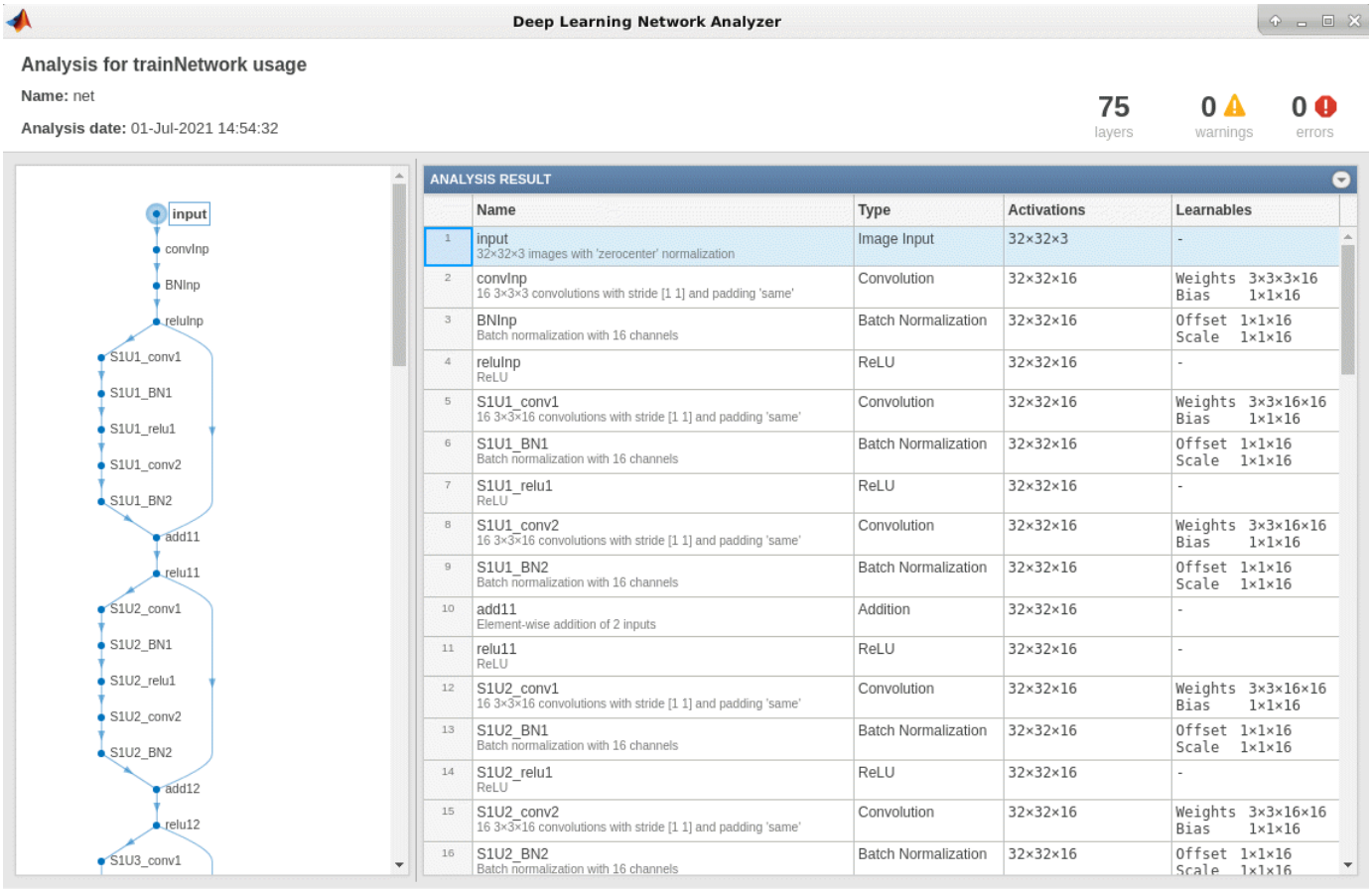
Load Pretrained Network

Load the pretrained network. For information on creating and training a network with residual connections for image classification yourself, see Train Residual Network for Image Classification.

```
load('CIFARNet-20-16.mat','trainedNet');
net = trainedNet;
```

You can use `analyzeNetwork` to analyze the deep learning network architecture.

```
analyzeNetwork(net)
```



Load Data

Download the CIFAR-10 data set [1] by executing the code below. The data set contains 60,000 images. Each image is 32-by-32 in size and has three color channels (RGB). The size of the data set is 175 MB. Depending on your internet connection, the download process can take some time.

```
datadir = tempdir;
downloadCIFARData(datadir);
```

```
Downloading CIFAR-10 dataset (175 MB). This can take a while...done.
```

Prepare Data for Calibration and Validation

Load the CIFAR-10 training and test images as 4-D arrays. The training set contains 50,000 images and the test set contains 10,000 images. Use the CIFAR-10 test images for network validation.

```
[XTrain,YTrain,XValidation,YValidation] = loadCIFARData(datadir);
```

You can display a random sample of the training images using the following code.

```
figure;
idx = randperm(size(XTrain,4),20);
im = imtile(XTrain(:,:,:,idx),'ThumbnailSize',[96,96]);
imshow(im)
```

Create an `augmentedImageDatastore` object to use for calibration and validation. Use 200 random images for calibration and 50 random images for validation.

```
inputSize = net.Layers(1).InputSize;

augimdsTrain = augmentedImageDatastore(inputSize,XTrain,YTrain);
augimdsCalibration = shuffle(augimdsTrain).subset(1:200);

augimdsValidation = augmentedImageDatastore(inputSize,XValidation,YValidation);
augimdsValidation = shuffle(augimdsValidation).subset(1:50);
```

Quantize the Network for GPU Deployment Using the Deep Network Quantizer App

This example uses a GPU execution environment. To learn about the products required to quantize and deploy the deep learning network to a GPU environment, see Quantization Workflow Prerequisites.

In the MATLAB® Command Window, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Select **New > Quantize a network**. The app automatically verifies your execution environment.

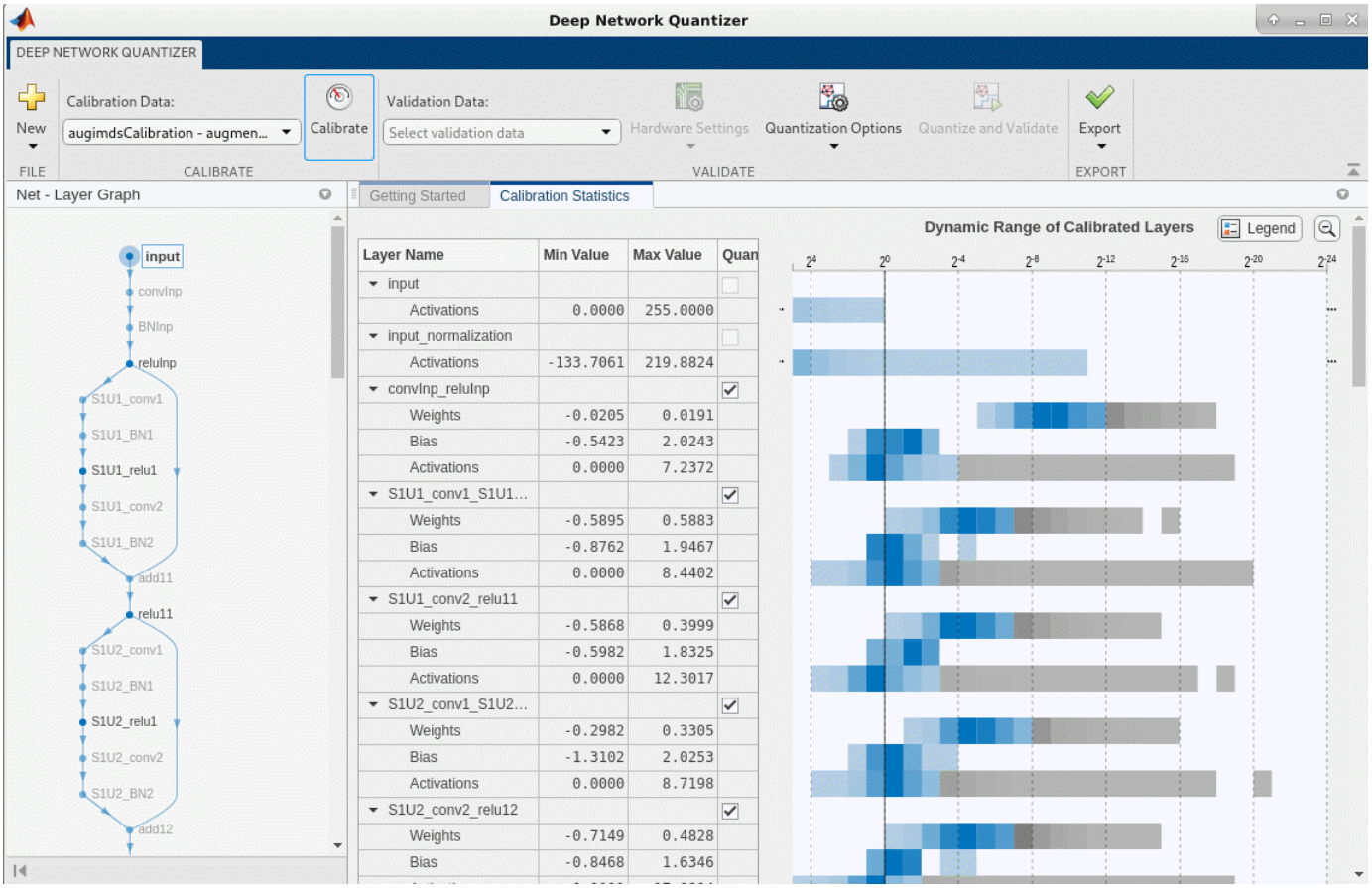
In the dialog, select the execution environment and the network to quantize from the base workspace. For this example, select a GPU execution environment and the DAG network `net`.

In the **Calibrate** section of the toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data `augimdsCalibration`.

Click **Calibrate**.

Deep Network Quantizer uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

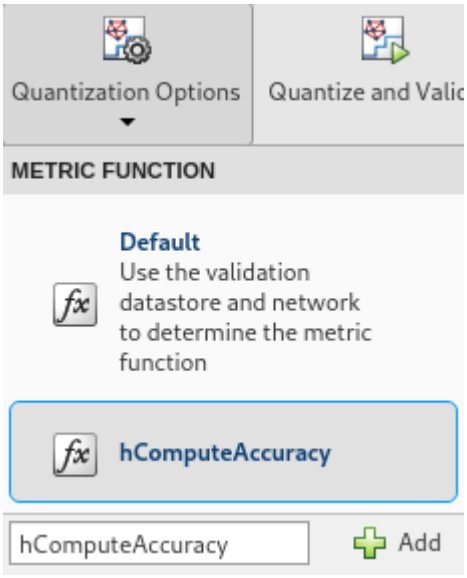
When the calibration is complete, the app displays a table containing the weights and biases in the convolution, as well as fully connected layers of the network and the dynamic ranges of the activations in all layers of the network with their minimum and maximum values during the calibration. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see Quantization of Deep Neural Networks.



In the **Quantize Layer** column of the table, indicate whether to quantize the learnable parameters in the layer. Layers that are not convolution layers cannot be quantized, and therefore cannot be selected. Layers that are not quantized remain in single precision after quantization.

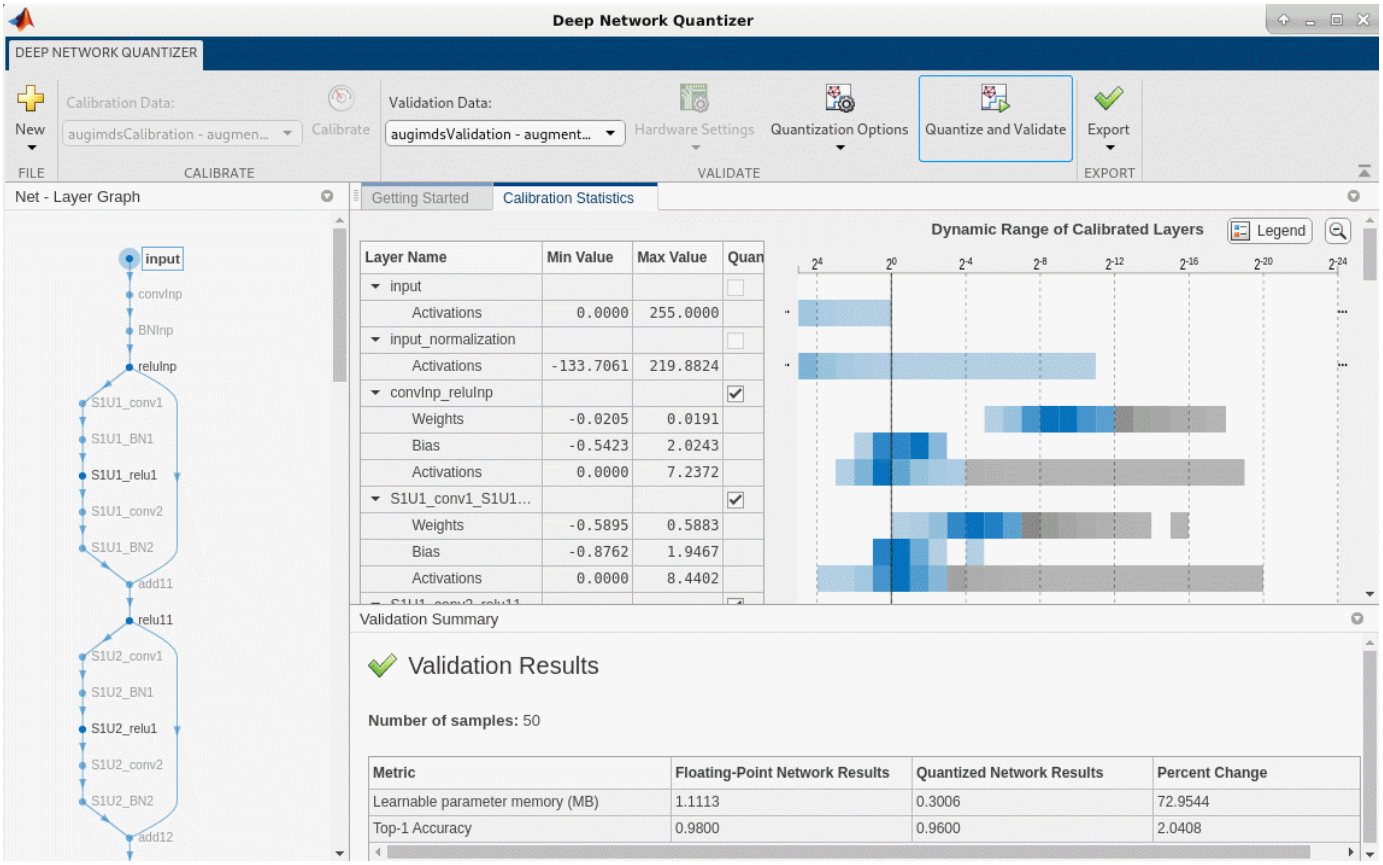
In the **Validate** section of the toolstrip, under **Validation Data**, select the `augmentedImageDatastore` object from the base workspace containing the validation data, `augimdsValidation`.

In the **Validate** section of the toolstrip, under **Quantization Options**, select the metric function to use for validation. The app determines a default metric function to use for validation based on the type of network that you quantize. You can also add additional custom metric functions to use for validation. For this example, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function to use for validation. This custom metric function compares the predicted label to the ground truth and returns the top-1 accuracy. The custom metric function must be on the path.



Click **Quantize and Validate**.

The app quantizes the network and displays a summary of the validation results. For this set of calibration and validation images, quantization of the network results in a 2% decrease in accuracy with a 73% reduction in learnable parameter memory for the set of 50 validation images.



After quantizing and validating the network, you can export the quantized network or generate code. To export the quantized network, select **Export > Export Quantizer** to create a `dlquantizer` object in the base workspace. To open the GPU Coder app and generate GPU code from the quantized neural network, select **Export > Generate Code**. To learn how to generate CUDA code for a quantized deep convolutional neural network using GPU Coder, see Code Generation for Quantized Deep Learning Networks.

Validate the Performance of the Quantized Network Using Multiple Metric Functions

You can use multiple metric functions to evaluate the performance of the quantized network simultaneously by using the `dlquantizer` function.

To begin, load the pretrained network and data, and prepare the data for calibration and validation, as described above.

Create a `dlquantizer` object. Specify the network to quantize and the execution environment to use. Use the `calibrate` function to exercise the network with sample inputs from `augimdsCalibration` and collect range information.

```
dq = dlquantizer(net,'ExecutionEnvironment','GPU');
calResults = calibrate(dq,augimdsCalibration)
```

```
calResults=72×5 table
          Optimized Layer Name           Network Layer Name    Learnables / Activations    MinVa
    _____________________________________    _________________    ________________________    _____

    {'convInp_reluInp_Weights'      }      {'reluInp'   }            "Weights"             -0.020
    {'convInp_reluInp_Bias'         }      {'reluInp'   }            "Bias"                 -0.54
    {'S1U1_conv1_S1U1_relu1_Weights'}      {'S1U1_relu1'}            "Weights"              -0.58
    {'S1U1_conv1_S1U1_relu1_Bias'   }      {'S1U1_relu1'}            "Bias"                 -0.87
    {'S1U1_conv2_relu11_Weights'    }      {'relu11'    }            "Weights"              -0.58
    {'S1U1_conv2_relu11_Bias'       }      {'relu11'    }            "Bias"                 -0.59
    {'S1U2_conv1_S1U2_relu1_Weights'}      {'S1U2_relu1'}            "Weights"              -0.29
    {'S1U2_conv1_S1U2_relu1_Bias'   }      {'S1U2_relu1'}            "Bias"                  -1.3
    {'S1U2_conv2_relu12_Weights'    }      {'relu12'    }            "Weights"               -0.7
    {'S1U2_conv2_relu12_Bias'       }      {'relu12'    }            "Bias"                 -0.84
    {'S1U3_conv1_S1U3_relu1_Weights'}      {'S1U3_relu1'}            "Weights"              -0.10
    {'S1U3_conv1_S1U3_relu1_Bias'   }      {'S1U3_relu1'}            "Bias"                  -1.3
    {'S1U3_conv2_relu13_Weights'    }      {'relu13'    }            "Weights"              -0.47
    {'S1U3_conv2_relu13_Bias'       }      {'relu13'    }            "Bias"                  -1.0
    {'S2U1_conv1_S2U1_relu1_Weights'}      {'S2U1_relu1'}            "Weights"              -0.14
    {'S2U1_conv1_S2U1_relu1_Bias'   }      {'S2U1_relu1'}            "Bias"                  -1.8
      ⋮
```

Specify the metric functions in a `dlquantizationOptions` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The `validate` function uses the metric functions defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization. For this example, use the top-1 accuracy and top-5 accuracy metrics are used to evaluate the performance of the network.

```
dqOpts = dlquantizationOptions('MetricFcn',...
    {@(x)hComputeAccuracy(x,net,augimdsValidation), ...
    @(x)hComputeTop_5(x,net,augimdsValidation)});

validationResults = validate(dq,augimdsValidation,dqOpts)
```

```
validationResults = struct with fields:
       NumSamples: 50
    MetricResults: [1×2 struct]
       Statistics: [2×2 table]
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network as measured by each metric function used.

```
validationResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}          0.98
     {'Quantized'     }          0.96


ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           1
     {'Quantized'     }           1
```

```
validationResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}               1.1113e+06
     {'Quantized'     }               3.0057e+05
```

To visualize the calibration statistics, first save the `dlquantizer` object `dq.`

```
save('dlquantObj.mat','dq')
```

Then import the `dlquantizer` object `dq` in the Deep Network Quantizer app by selecting **New > Import `dlquantizer` object**.

Generate CUDA Code

Generate CUDA® code for a quantized deep convolutional neural network.

Create Entry-Point Function

Write an entry-point function in MATLAB® that:

1. Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see Load Pretrained Networks for Code Generation.
2. Calls `predict` (Deep Learning Toolbox) to predict the responses.

```
type('mynet_predict.m');
```

```
function out = mynet_predict(netFile, im)
    persistent net;
    if isempty(net)
        net = coder.loadDeepLearningNetwork(netFile);
    end
    out = net.predict(im);
end
```

A persistent object `mynet` loads the `DAGNetwork` object. The first call to the entry-point function constructs and sets up the persistent object. Subsequent calls to the function reuse the same object to call `predict` on inputs, avoiding reconstructing and reloading the network object.

Code Generation by Using codegen

To configure build settings such as the output file name, location, and type, you create coder configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, when generating CUDA MEX using the codegen command, use `cfg = coder.gpuConfig('mex')`.

To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

Specify the location of the MAT file containing the calibration data.

Specify the precision of the inference computations in supported layers by using the `DataType` property. For 8-bit integers, use `'int8'`. Int8 precision requires a CUDA GPU with compute capability of 6.1, 6.3, or higher. Use the `ComputeCapability` property of the `gpuConfig` object to set the appropriate compute capability value.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.CalibrationResultFile = 'dlquantObj.mat';
netFile = 'mynet.mat';
save(netFile,'net');
```

Run the `codegen` command. The `codegen` command generates CUDA code from the `mynet_predict.m` entry-point function.

```
codegen -config cfg mynet_predict -args {coder.Constant(netFile), ones(inputSize, 'single')} -re
```

When code generation is successful, you can view the resulting code generation report by clicking View Report in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See Code Generation Reports.

```
Code generation successful: View report
```

References

[1] Krizhevsky, Alex. 2009. "Learning Multiple Layers of Features from Tiny Images." https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

See Also

Apps
Deep Network Quantizer

Functions
`dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate`

Related Examples

- “Quantization Workflow Prerequisites” on page 20-193
- “Train Residual Network for Image Classification” on page 3-13
- “Code Generation for Quantized Deep Learning Networks” (GPU Coder)

Quantize Object Detectors and Generate CUDA® Code

This example shows how to generate CUDA® code for an SSD vehicle detector and a YOLO v2 vehicle detector that performs inference computations in 8-bit integers.

Deep learning is a powerful machine learning technique in which you train a network to learn image features and perform detection tasks. There are several techniques for object detection using deep learning, such as Faster R-CNN, You Only Look Once (YOLO v2), and SSD. For more information, see Object Detection Using YOLO v2 Deep Learning and Object Detection Using SSD Deep Learning.

Neural network architectures used for deep learning applications contain many processing layers, including convolutional layers. Deep learning models typically work on large sets of labeled data. Performing inference on these models is computationally intensive, consuming significant amounts of memory. Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Deep neural networks trained in MATLAB use single-precision floating point data types. Even networks that are small in size require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and smaller memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

You can use Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Then, you can use GPU Coder™ to generate CUDA code for the quantized network.

Download Pretrained Network

Download a pretrained object detector to avoid having to wait for training to complete.

```
detectorType = YOLOV2
```

```
detectorType = 2
```

```
switch detectorType
    case 1
        if ~exist('ssdResNet50VehicleExample_20a.mat','file')
            disp('Downloading pretrained detector...');
            pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/ssdResNet50Vehic
            websave('ssdResNet50VehicleExample_20a.mat',pretrainedURL);
        end
    case 2
        if ~exist('yolov2ResNet50VehicleExample_19b.mat','file')
            disp('Downloading pretrained detector...');
            pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50Vel
            websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
        end
end
```

```
Downloading pretrained detector...
```

Load Data

This example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, used with permission and available at the Caltech

Computational Vision website, created by Pietro Perona. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the training procedure, but in practice, more labeled images are needed to train a robust detector. Extract the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

Prepare Data for Training, Calibration, and Validation

The training data is stored in a table. The first column contains the path to the image files. The remaining columns contain the ROI labels for vehicles. Display the first few rows of the data.

```
vehicleDataset(1:4,:)
```

```
ans=4×2 table
              imageFilename                    vehicle
    ____________________________________    _______________

    {'vehicleImages/image_00001.jpg'}    {[220 136 35 28]}
    {'vehicleImages/image_00002.jpg'}    {[175 126 61 45]}
    {'vehicleImages/image_00003.jpg'}    {[108 120 45 33]}
    {'vehicleImages/image_00004.jpg'}    {[124 112 38 36]}
```

Split the data set into training, validation, and test sets. Select 60% of the data for training, 10% for calibration, and the remainder for validating the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

calibrationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
calibrationDataTbl = vehicleDataset(shuffledIndices(calibrationIdx),:);

validationIdx = calibrationIdx(end)+1 : length(shuffledIndices);
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsCalibration = imageDatastore(calibrationDataTbl{:,'imageFilename'});
bldsCalibration = boxLabelDatastore(calibrationDataTbl(:,'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:,'imageFilename'});
bldsValidation = boxLabelDatastore(validationDataTbl(:,'vehicle'));
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
calibrationData = combine(imdsCalibration,bldsCalibration);
validationData = combine(imdsValidation,bldsValidation);
```

Display one of the training images and box labels.

```
data = read(calibrationData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Define Network Parameters

To reduce the computational cost of running the example, specify a network input size that corresponds to the minimum size required to run the network.

```
inputSize = [];
switch detectorType
    case 1
        inputSize = [300 300 3]; % Minimum size for SSD
    case 2
        inputSize = [224 224 3]; % Minimum size for YOLO v2
end
```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use transformations to augment the training data by:

- Randomly flipping the image and associated box labels horizontally.
- Randomly scaling the image and associated box labels.
- Jitter the image color.

Note that data augmentation is not applied to the test data. Ideally, test data is representative of the original data and left unmodified for unbiased evaluation.

```
augmentedCalibrationData = transform(calibrationData,@augmentVehicleData);
```

Visualize augmented training data by reading the same image multiple times.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedCalibrationData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedCalibrationData);
end

figure
montage(augmentedData,'BorderSize',10)
```

Preprocess Calibration Data

Preprocess the augmented calibration data to prepare for calibration of the network.

```
preprocessedCalibrationData = transform(augmentedCalibrationData,@(data)preprocessVehicleData(da
```

Read the preprocessed calibration data.

```
data = read(preprocessedCalibrationData);
```

Display the image and bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Load and Test Pretrained Detector

Load the pretrained detector.

```
switch detectorType
    case 1
        % Load pretrained SSD detector for the example.
        pretrained = load('ssdResNet50VehicleExample_20a.mat');
        detector = pretrained.detector;
    case 2
        % Load pretrained YOLO v2 detector for the example.
        pretrained = load('yolov2ResNet50VehicleExample_19b.mat');
        detector = pretrained.detector;
end
```

As a quick test, run the detector on one test image.

```
data = read(calibrationData);
I = data{1,1};
```

```
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I, 'Threshold', 0.4);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Validate Floating-Point Network

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides functions to measure common object detector metrics, such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (`precision`) and the ability of the detector to find all relevant objects (`recall`).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Ideally, test data is representative of the original data and left unmodified for unbiased evaluation.

```
preprocessedValidationData = transform(validationData,@(data)preprocessVehicleData(data,inputSize
```

Run the detector on all the test images.

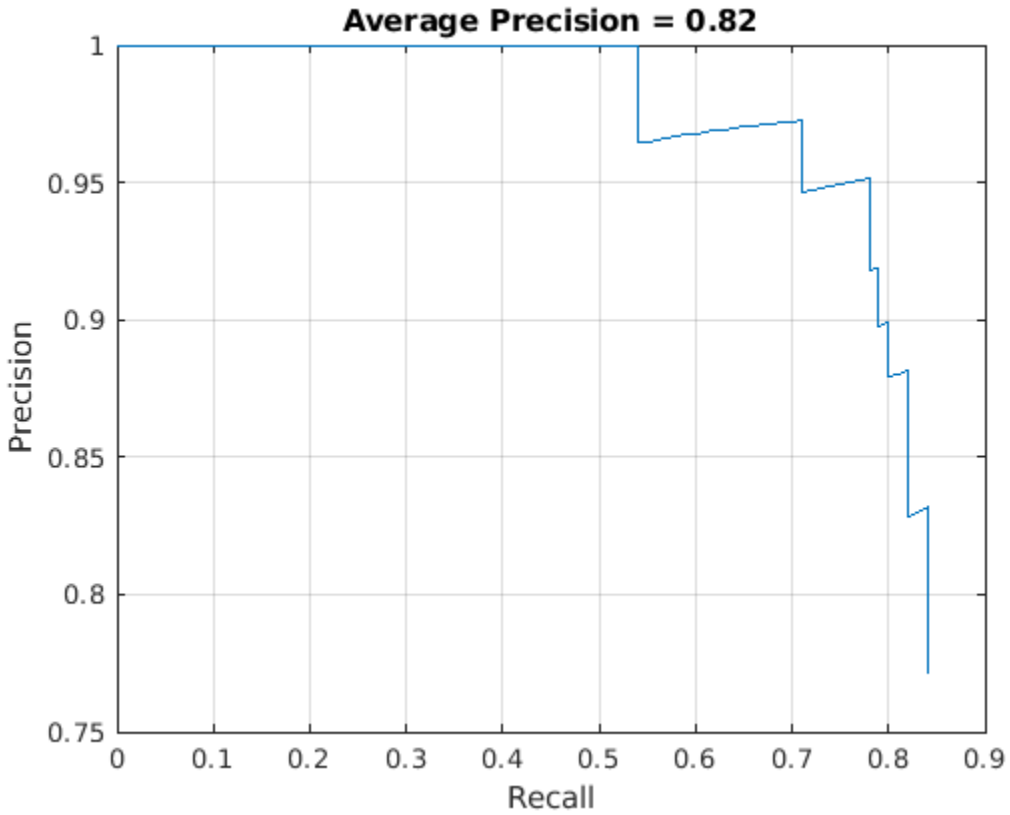
```
detectionResults = detect(detector, preprocessedValidationData,'Threshold',0.4);
```

Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults,preprocessedValidationData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. Ideally, the precision is 1 at all recall levels. Using more data can help improve the average precision, but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Quantize Network

Create a `dlquantizer` object and specify the detector to quantize. By default, the execution environment is set to GPU. To learn about the products required to quantize and deploy the detector to a GPU environment, see Quantization Workflow Prerequisites.

```
quantObj = dlquantizer(detector)
```

```
quantObj =
  dlquantizer with properties:

          NetworkObject: [1×1 yolov2ObjectDetector]
    ExecutionEnvironment: 'GPU'
```

Specify the metric function in a `dlquantizationOptions` object.

```
quantOpts = dlquantizationOptions;
quantOpts = dlquantizationOptions('MetricFcn', ...
    {@(x)hVerifyDetectionResults(x, detector.Network, preprocessedValidationData)});
```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network, as well as the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

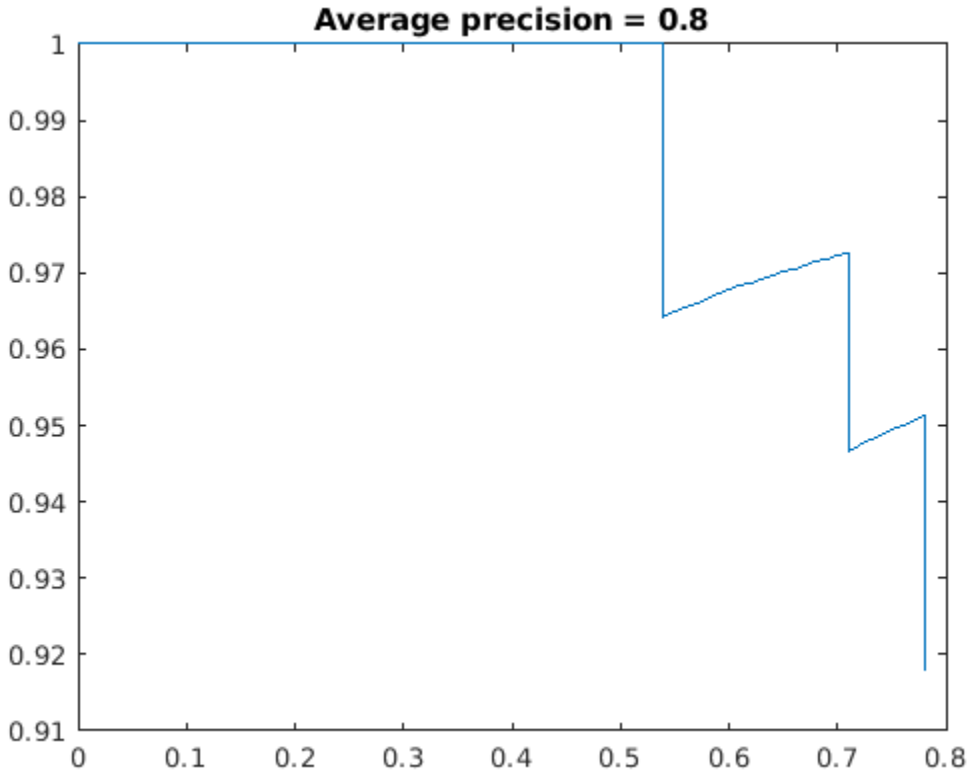
```
calResults = calibrate(quantObj,preprocessedCalibrationData)
```

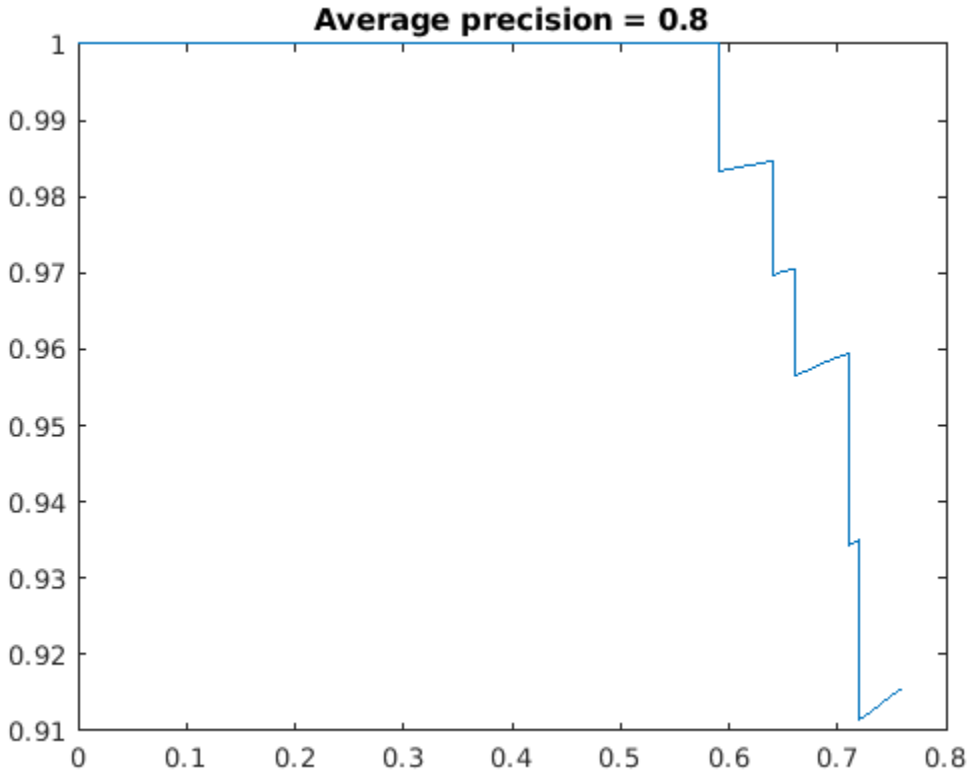
```
calResults=147×5 table
                Optimized Layer Name                 Network Layer Name      Learnables / Activat
    ____________________________________________    ____________________    ____________________

    {'conv1_activation_1_relu_Weights'         }    {'activation_1_relu'}          "Weights"
    {'conv1_activation_1_relu_Bias'            }    {'activation_1_relu'}          "Bias"
    {'res2a_branch2a_activation_2_relu_Weights'}    {'activation_2_relu'}          "Weights"
    {'res2a_branch2a_activation_2_relu_Bias'   }    {'activation_2_relu'}          "Bias"
    {'res2a_branch2b_activation_3_relu_Weights'}    {'activation_3_relu'}          "Weights"
    {'res2a_branch2b_activation_3_relu_Bias'   }    {'activation_3_relu'}          "Bias"
    {'res2a_branch1_Weights'                   }    {'bn2a_branch1'     }          "Weights"
    {'res2a_branch1_Bias'                      }    {'bn2a_branch1'     }          "Bias"
    {'res2a_branch2c_activation_4_relu_Weights'}    {'activation_4_relu'}          "Weights"
    {'res2a_branch2c_activation_4_relu_Bias'   }    {'activation_4_relu'}          "Bias"
    {'res2b_branch2a_activation_5_relu_Weights'}    {'activation_5_relu'}          "Weights"
    {'res2b_branch2a_activation_5_relu_Bias'   }    {'activation_5_relu'}          "Bias"
    {'res2b_branch2b_activation_6_relu_Weights'}    {'activation_6_relu'}          "Weights"
    {'res2b_branch2b_activation_6_relu_Bias'   }    {'activation_6_relu'}          "Bias"
    {'res2b_branch2c_activation_7_relu_Weights'}    {'activation_7_relu'}          "Weights"
    {'res2b_branch2c_activation_7_relu_Bias'   }    {'activation_7_relu'}          "Bias"
      ⋮
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

Examine the `MetricResults.Result` field of the validation output to see the performance of the quantized network. The first row in the results table contains the information for the original, floating-point implementation. The second row contains the information for the quantized implementation. The output of the metric function is displayed in the `MetricOutput` column.

```
valResults = validate(quantObj,preprocessedValidationData,quantOpts)
```


Average precision = 0.8
1
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8




```
valResults = struct with fields:
       NumSamples: 88
    MetricResults: [1×1 struct]
       Statistics: [2×2 table]
```

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}        0.77119
     {'Quantized'     }        0.75244
```

The metrics show that quantization reduces the required memory by approximately 75% and the network accuracy by approximately 3%.

To visualize the calibration statistics, use the Deep Network Quantizer app. First, save the `dlquantizer` object.

```
save('dlquantObj.mat','quantObj')
```

In the MATLAB® Command Window, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Then import the `dlquantizer` object `dq` in the Deep Network Quantizer app by selecting **New > Import `dlquantizer` object**.

Generate CUDA Code

After you train and evaluate the detector, you can generate code for the `ssdObjectDetector` or `yolov2ObjectDetector` using GPU Coder™. For more details, see “Code Generation for Object Detection by Using Single Shot Multibox Detector” (Computer Vision Toolbox) and “Code Generation for Object Detection by Using YOLO v2” (GPU Coder).

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';

% Check compute capability of GPU
gpuInfo = gpuDevice;
cc = gpuInfo.ComputeCapability;

% Create deep learning code generation configuration object
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');

% INT8 precision requires a CUDA GPU with minimum compute capability of
% 6.1, 6.3, or higher
cfg.GpuConfig.ComputeCapability = cc;
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.CalibrationResultFile = 'dlquantObj.mat';
```

Run the `codegen` command to generate CUDA code.

```
codegen -config cfg mynet_detect -args {coder.Constant(detectorType), ones(inputSize, 'single')}
```

When code generation is successful, you can view the resulting code generation report by clicking View Report in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See Code Generation Reports.

References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single Shot Multibox Detector." In Computer Vision - ECCV 2016, edited by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, 9905:21-37. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-46448-0_2

[2] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 6517-25. Honolulu, HI: IEEE, 2017. https://doi.org/10.1109/CVPR.2017.690

Parameter Pruning and Quantization of Image Classification Network

This example shows how to prune the parameters of a trained neural network using two parameter score metrics: The magnitude score [1] and Synaptic Flow score [2].

In many applications where transfer learning is used to retrain an image classification network for a new task or when a new network is trained from scratch, the optimal network architecture is not known and teh network might be overparameterized. An overparameterized network has redundant connections.. Structured pruning, also known as sparsification, is a compression technique that aims to identify redundant, unnecessary connections you can remove without affecting the network accuracy. When you use pruning in combination with network quantization, you can reduce the inference speed and memory footprint of the network making it easier to deploy.

This example shows how to:

- Perform post-training, iterative, unstructured pruning without the need for training data
- Evaluate the performance of two different pruning algorithms
- Investigate the layer-wise sparsity induced after pruning
- Evaluate the impact of pruning on classification accuracy
- Evaluate the impact of quantization on the clasification accuracy of the pruned network

This example uses a simple convolutional neural network to classify handwritten digits from 0 to 9. For more information on setting up the data used for training and validation, see Create Simple Deep Learning Network for Classification.

Load Pretrained Network and Data

Load the training and validation data. Train a convolutional neural network for the classification task.

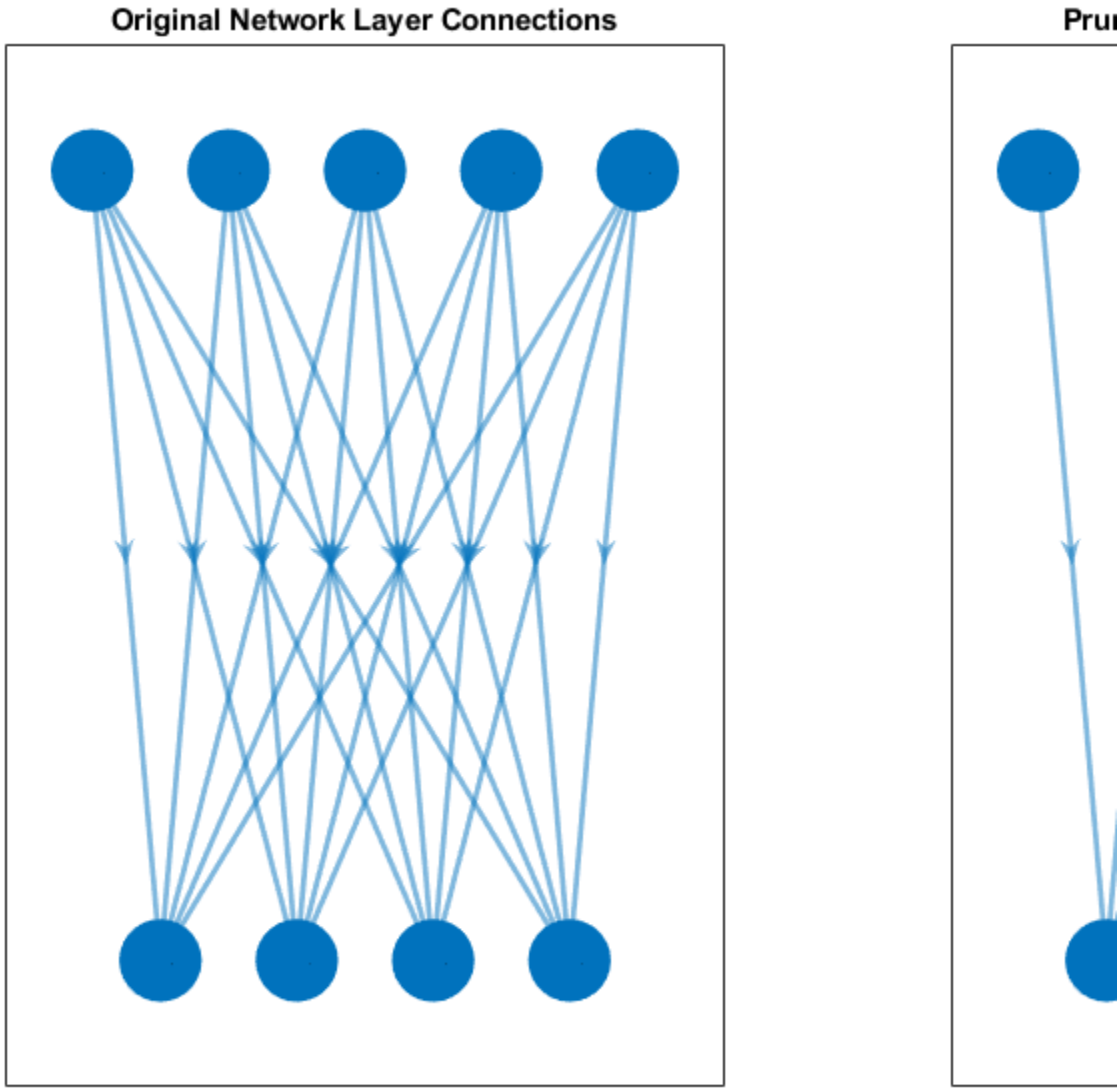
```
[imdsTrain, imdsValidation] = loadDigitDataset;
net = trainDigitDataNetwork(imdsTrain, imdsValidation);
trueLabels = imdsValidation.Labels;
classes = categories(trueLabels);
```

Create a `minibatchqueue` object containing the validation data. SetexecutionEnvironment to auto to evaluate the network on a GPU, if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see GPU Support by Release (Parallel Computing Toolbox).

```
executionEnvironment = auto;
miniBatchSize = 128;
imdsValidation.ReadSize = miniBatchSize;
mbqValidation = minibatchqueue(imdsValidation,1,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFormat','SSCB',...
    'MiniBatchFcn',@preprocessMiniBatch,...
    'OutputEnvironment',executionEnvironment);
```

Neural Network Pruning

The goal of neural network pruning is to identify and remove unimportant connections to reduce the size of the network without affecting network accuracy. In the following figure, on the left, the network has connections that map each neuron to the neuron of the next layer. After pruning, the network has fewer connections than the original network.

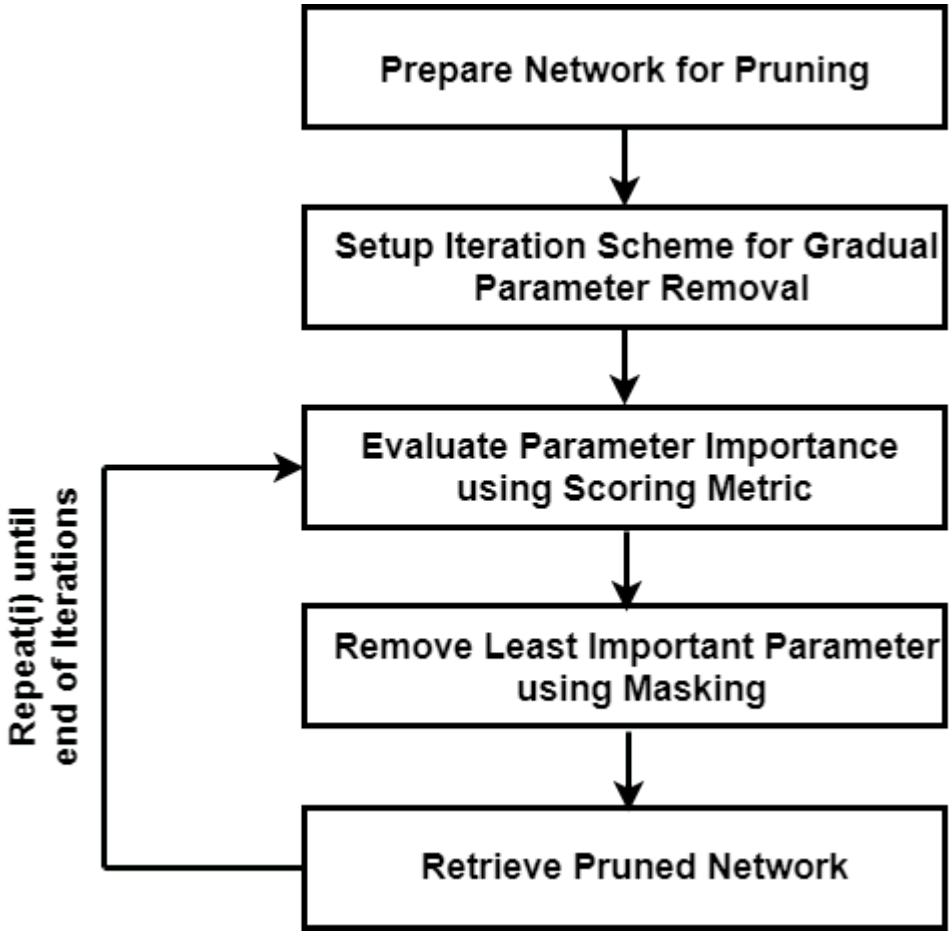


A pruning algorithm assigns a score to each parameter in the network. The score ranks the importance of each connection in the network. You can use one of two pruning approaches to achieve a target sparsity:

- One-shot pruning - Remove a specified percentage of connections based on their score in one step. This method is prone to layer collapse when you specify a high sparsity value.
- Iterative pruning - Achieve the target sparsity in a series of iterative steps. You can use this method when evaluated scores are sensitive to network structure. Scores are reevaluated at every iteration, so using a series of steps allows the network to move toward sparsity incrementally.

This example uses the iterative pruning method to achieve a target sparsity.

Iterative Pruning



Convert to `dlnetwork` Object

In this example, you use the Synaptic Flow algorithm, which requires that you create a custom cost function and evaluate the gradients with respect to the cost function to calculate the parameter score. To create a custom cost function, first convert the pretrained network to a `dlnetwork`.

Convert the network to a layer graph and remove the layers used for classification using `removeLayers`.

```
lgraph = layerGraph(net.Layers);
lgraph = removeLayers(lgraph,["softmax","classoutput"]);
dlnet = dlnetwork(lgraph);
```

Use `analyzeNetwork` to analyze the network architecture and learnable parameters.

```
analyzeNetwork(dlnet)
```

Evaluate the accuracy of the network before pruning.

```
accuracyOriginalNet = evaluateAccuracy(dlnet,mbqValidation,classes,trueLabels)
```

```
accuracyOriginalNet = 0.9900
```

The layers with learnable parameters are the 3 convolution layers and one fully connected layer. The network initially consists of total 21578 learnable parameters.

```
numTotalParams = sum(cellfun(@numel,dlnet.Learnables.Value))
```

```
numTotalParams = 21578
```

```
numNonZeroPerParam = cellfun(@(w)nnz(extractdata(w)),dlnet.Learnables.Value)
```

```
numNonZeroPerParam = 8×1

          72
           8
        1152
          16
        4608
          32
       15680
          10
```

Sparsity is defined as the percentage of parameters in the network with a value of zero. Check the sparsity of the network.

```
initialSparsity = 1-sum(numNonZeroPerParam)/numTotalParams
```

```
initialSparsity = 0
```

Before pruning, the network has a sparsity of zero.

Create Iteration Scheme

To define an iterative pruning scheme, specify the target sparsity and number of iterations. For this example, use linearly spaced iterations to achieve the target sparsity.

```
numIterations = 10;
targetSparsity = 0.90;
iterationScheme = linspace(0,targetSparsity,numIterations);
```

Pruning Loop

For each iteration, the custom pruning loop in this example performs the following steps:

- Calculate the score for each connection.
- Rank the scores for all connections in the network based on the selected pruning algorithm.
- Determine the threshold for removing connections with the lowest scores.
- Create the pruning mask using the threshold.
- Apply the pruning mask to learnable parameters of the network.

Network Mask

Instead of setting entries in the weight arrays directly to zero, the pruning algorithm creates a binary mask for each learnable parameter that specifies whether a connection is pruned. The mask allows you to explore the behavior of the pruned network and try different pruning schemes without changing the underlying network structure.

For example, consider the following weights.

```
testWeight = [10.4 5.6 0.8 9];
```

Create a binary mask for each parameter in testWeight.

```
testMask = [1 0 1 0];
```

Apply the mask to `testWeight` to get the pruned weights.

```
testWeightsPruned = testWeight.*testMask
```

```
testWeightsPruned = 1×4

   10.4000         0    0.8000         0
```

In iterative pruning, you create a binary mask for each iteration that contains pruning information. Applying the mask to the weights array does not change either the size of the array or the structure of the neural network. Therefore, the pruning step does not directly result in any speedup during inference or compression of the network size on disk.

Initialize a plot that compares the accuracy of the pruned network to the original network.

```
figure
plot(100*iterationScheme([1,end]),100*accuracyOriginalNet*[1 1],'*-b','LineWidth',2,"Color","b")
ylim([0 100])
xlim(100*iterationScheme([1,end]))
xlabel("Sparsity (%)")
ylabel("Accuracy (%)")
legend("Original Accuracy","Location","southwest")
title("Pruning Accuracy")
grid on
```

Magnitude Pruning

Magnitude pruning [1] assigns a score to each parameter equal to its absolute value. It is assumed that the absolute value of a parameter corresponds to its relative importance to the accuracy of the trained network.

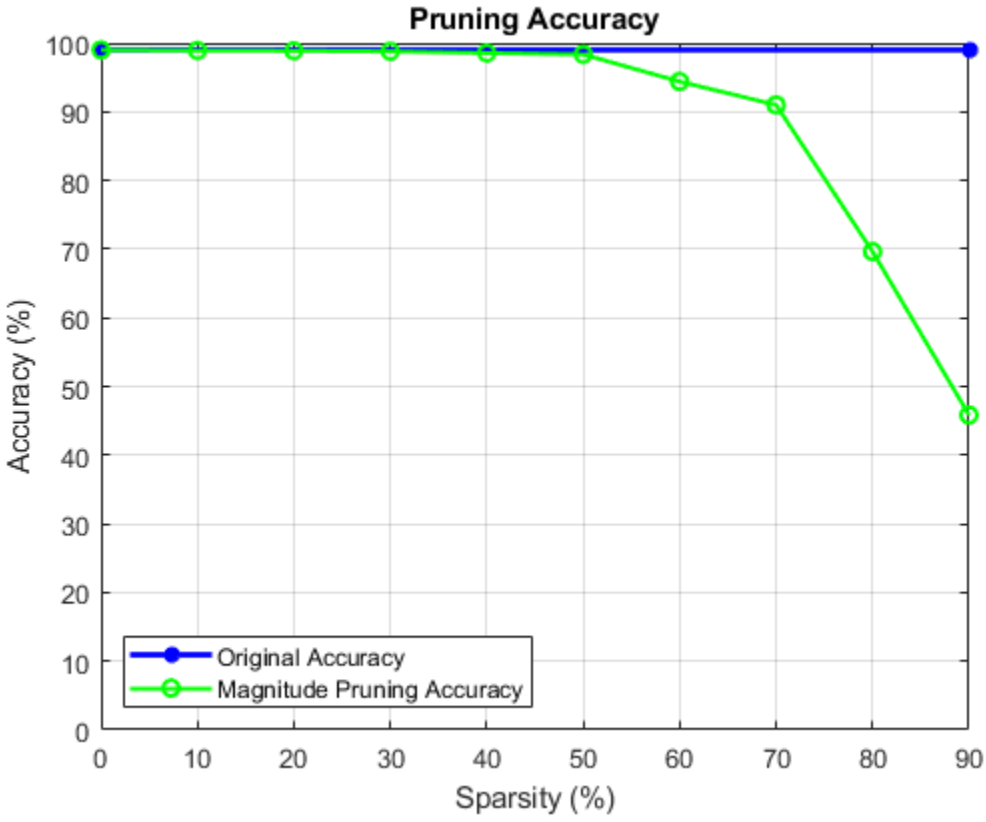
Initialize the mask. For the first iteration, you do not prune any parameters and the sparsity is 0%.

```
pruningMaskMagnitude = cell(1,numIterations);
pruningMaskMagnitude{1} = dlupdate(@(p)true(size(p)), dlnet.Learnables);
```

Below is an implementation of magnitude pruning. The network is pruned to various different target sparsities in a loop to provide the flexibility to choose a pruned network based on its accuracy.

```
lineAccuracyPruningMagnitude = animatedline('Color','g','Marker','o','LineWidth',1.5);
legend("Original Accuracy","Magnitude Pruning Accuracy","Location","southwest")
```

```
% Compute magnitude scores
scoresMagnitude = calculateMagnitudeScore(dlnet);

for idx = 1:numel(iterationScheme)

    prunedNetMagnitude = dlnet;

    % Update the pruning mask
    pruningMaskMagnitude{idx} = calculateMask(scoresMagnitude,iterationScheme(idx));

    % Check the number of zero entries in the pruning mask
    numPrunedParams = sum(cellfun(@(m)nnz(~extractdata(m)),pruningMaskMagnitude{idx}.Value));
    sparsity = numPrunedParams/numTotalParams;

    % Apply pruning mask to network parameters
    prunedNetMagnitude.Learnables = dlupdate(@(W,M)W.*M, prunedNetMagnitude.Learnables, pruningMa

    % Compute validation accuracy on pruned network
    accuracyMagnitude = evaluateAccuracy(prunedNetMagnitude,mbqValidation,classes,trueLabels);

    % Display the pruning progress
    addpoints(lineAccuracyPruningMagnitude,100*sparsity,100*accuracyMagnitude)
    drawnow
end
```



SynFlow Pruning

Synaptic flow conservation (SynFlow) [2] scores are used for pruning. You can use this method to prune networks that use linear activation functions such as ReLU.

Initialize the mask. For the first iteration, no parameters are pruned and the sparsity is 0%.

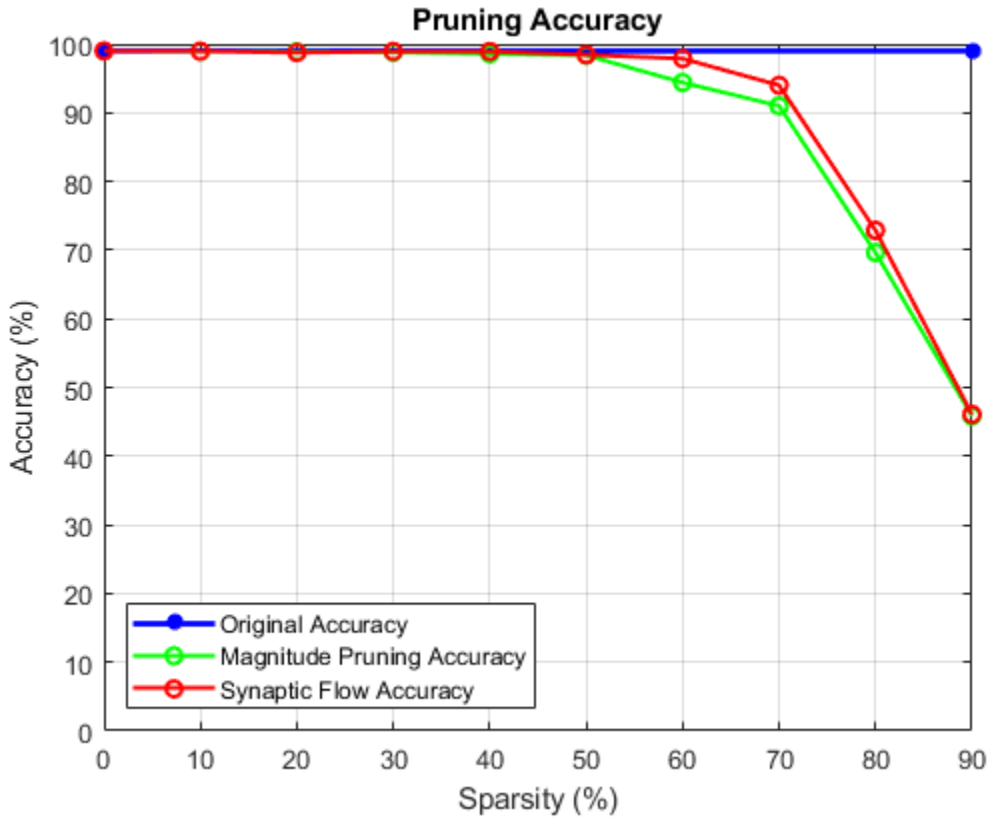
```
pruningMaskSynFlow = cell(1,numIterations);
pruningMaskSynFlow{1} = dlupdate(@(p)true(size(p)),dlnet.Learnables);
```

The input data you use to compute the scores is a single image containing ones. If you are using a GPU, convert the data to a `gpuArray`.

```
dlX = dlarray(ones(net.Layers(1).InputSize),'SSC');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end
```

The below loop implements iterative synaptic flow score for pruning [2] where a custom cost function evaluates the SynFlow score for each parameter used for network pruning.

```
lineAccuracyPruningSynflow = animatedline('Color','r','Marker','o','LineWidth',1.5);
legend("Original Accuracy","Magnitude Pruning Accuracy","Synaptic Flow Accuracy","Location","sou

prunedNetSynFlow = dlnet;

% Iteratively increase sparsity
for idx = 1:numel(iterationScheme)
    % Compute SynFlow scores
    scoresSynFlow = calculateSynFlowScore(prunedNetSynFlow,dlX);

    % Update the pruning mask
    pruningMaskSynFlow{idx} = calculateMask(scoresSynFlow,iterationScheme(idx));

    % Check the number of zero entries in the pruning mask
    numPrunedParams = sum(cellfun(@(m)nnz(~extractdata(m)),pruningMaskSynFlow{idx}.Value));
    sparsity = numPrunedParams/numTotalParams;

    % Apply pruning mask to network parameters
    prunedNetSynFlow.Learnables = dlupdate(@(W,M)W.*M, prunedNetSynFlow.Learnables, pruningMaskSy

    % Compute validation accuracy on pruned network
    accuracySynFlow = evaluateAccuracy(prunedNetSynFlow,mbqValidation,classes,trueLabels);

    % Display the pruning progress
    addpoints(lineAccuracyPruningSynflow,100*sparsity,100*accuracySynFlow)
    drawnow
end
```



Investigate Structure of Pruned Network

Choosing how much to prune a network is a trade-off between accuracy and sparsity. Use the sparsity versus accuracy plot to select the iteration with the desired sparsity level and acceptable accuracy.

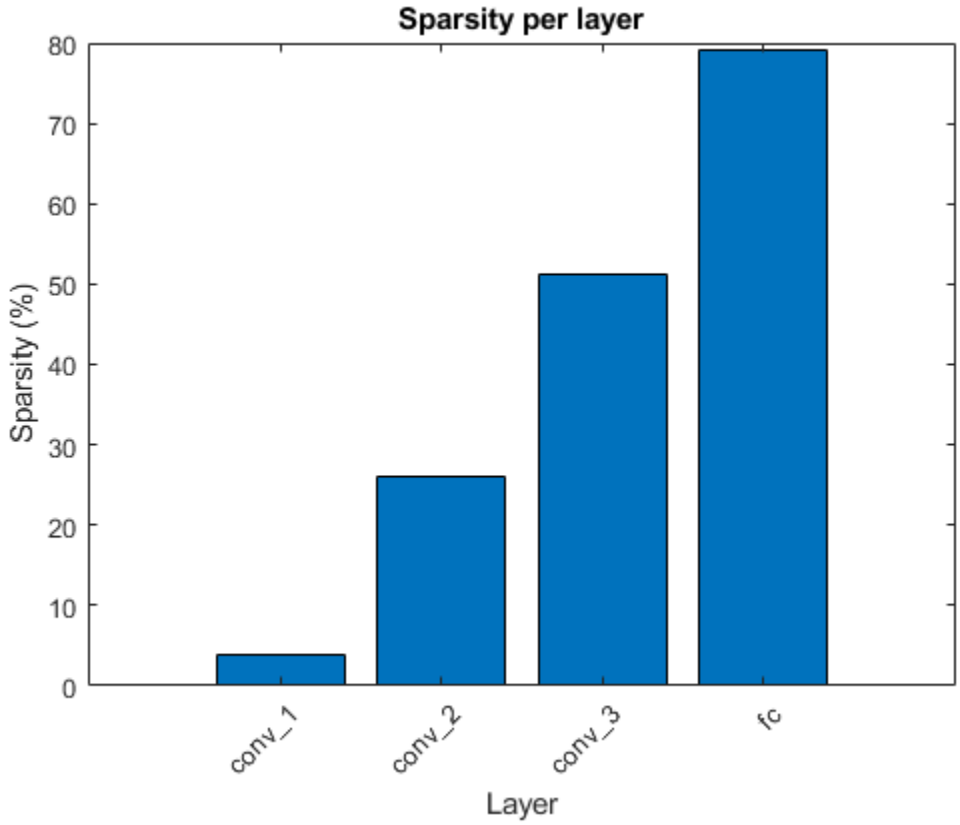
```
pruningMethod = "SynFlow";
selectedIteration = 8;

prunedDLNet = createPrunedNet(dlnet,selectedIteration,pruningMaskSynFlow,pruningMaskMagnitude,pru

[sparsityPerLayer,prunedChannelsPerLayer,numOutChannelsPerLayer,layerNames] = pruningStatistics(p
```

Earlier convolution layers are typically pruned less since they contain more relevant information about the core low-level structure of the image (e.g. edges and corners) which are essential for interpreting the image.

Plot the sparsity per layer for the selected pruning method and iteration.

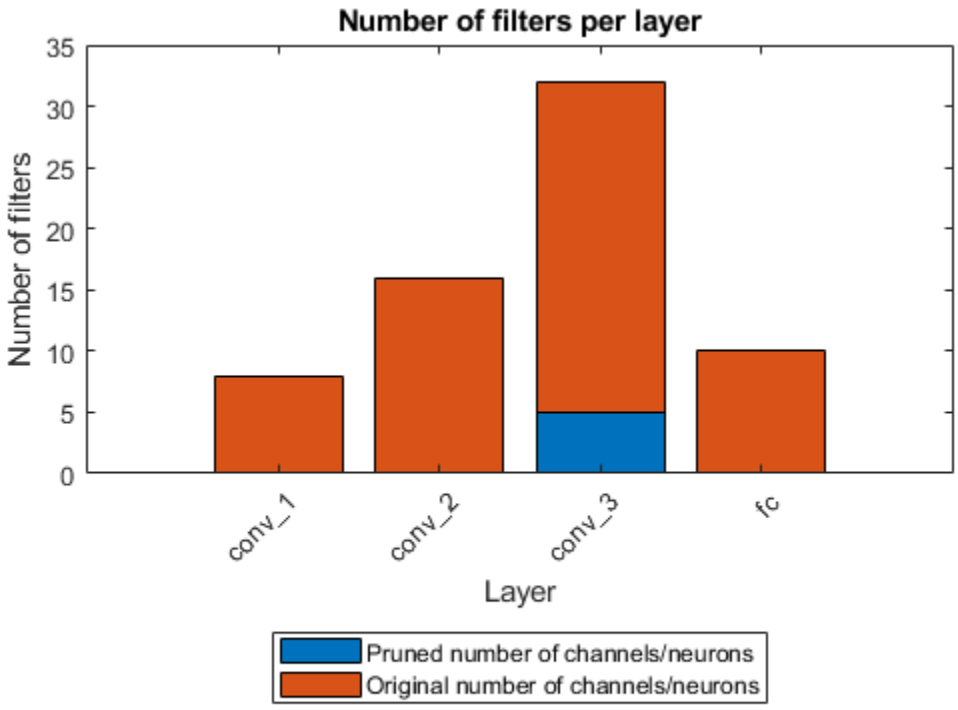
```
figure
bar(sparsityPerLayer*100)
title("Sparsity per layer")
xlabel("Layer")
ylabel("Sparsity (%)")
xticks(1:numel(sparsityPerLayer))
xticklabels(layerNames)
```

```
xtickangle(45)
set(gca,'TickLabelInterpreter','none')
```



The pruning algorithm prunes single connections when you specify a low target sparsity. When you specify a high target sparsity, the pruning algorithm can prune whole filters and neurons in convolutional or fully connected layers, allowing you to significantly reduce the structural size of the network.

```
figure
bar([prunedChannelsPerLayer,numOutChannelsPerLayer-prunedChannelsPerLayer],"stacked")
xlabel("Layer")
ylabel("Number of filters")
title("Number of filters per layer")
xticks(1:(numel(layerNames)))
xticklabels(layerNames)
xtickangle(45)
legend("Pruned number of channels/neurons" , "Original number of channels/neurons","Location","so
set(gca,'TickLabelInterpreter','none')
```



Evaluate Network Accuracy

Compare the accuracy of the network before and after pruning.

```
YPredOriginal = modelPredictions(dlnet,mbqValidation,classes);
accOriginal = mean(YPredOriginal == trueLabels)
```

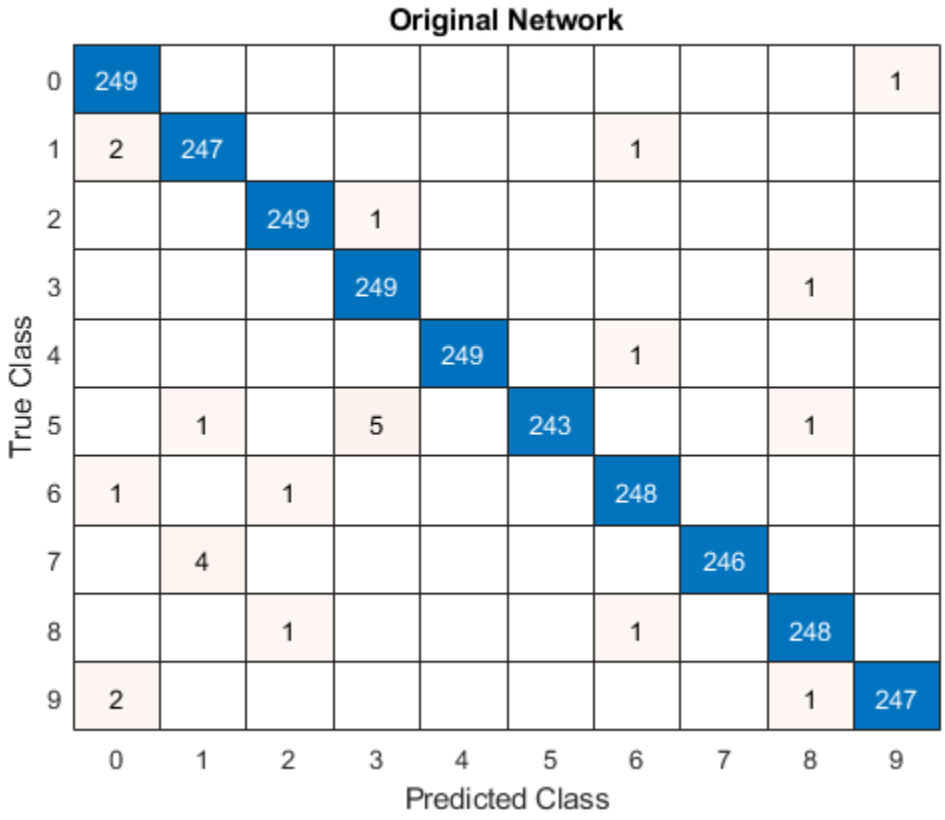
```
accOriginal = 0.9900
```

```
YPredPruned = modelPredictions(prunedDLNet,mbqValidation,classes);
accPruned = mean(YPredPruned == trueLabels)
```

```
accPruned = 0.9400
```

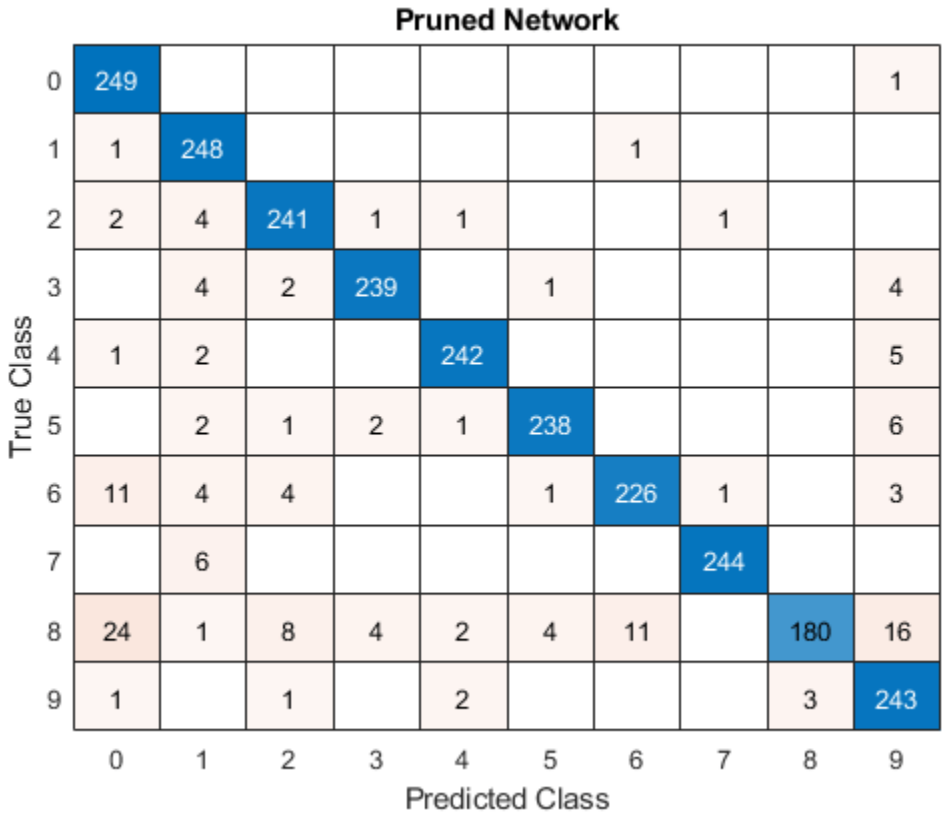
Create a confusion matrix chart to explore the true class labels to the predicted class labels for the original and pruned network.

```
figure
confusionchart(trueLabels,YPredOriginal);
title("Original Network")
```



The validation set of the digits data contains 250 images for each class, so if a network predicts the class of each image perfectly, all scores on the diagonal equal 250 and no values are outside of the diagonal.

```
confusionchart(trueLabels,YPredPruned);
title("Pruned Network")
```



When pruning a network, compare the confusion chart of the original network and the pruned network to check how the accuracy for each class labels changes for the selected sparsity level. If all numbers on the diagonal decrease roughly equally, no bias is present. However, if the decreases are not equal, you might need to choose a pruned network from an earlier iteration by reducing the value of the variable selectedIteration.

Quantize Pruned Network

Deep neural networks trained in MATLAB use single-precision floating point data types. Even networks that are small in size require a considerable amount of memory and hardware to perform floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models that have low computational power and less memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network. You can use Deep Learning Toolbox in tandem with the Deep Learning Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of the convolution layers to 8-bit scaled integer data types.

Pruning a network impacts the range statistics of parameters and activations at each layer, so the accuracy of the quantized network can change. To explore this difference, quantize the pruned network and use the quantized network to perform inference.

Split the data into calibration and validation data sets.

```
calibrationDataStore = splitEachLabel(imdsTrain,0.1,'randomize');
validationDataStore = imdsValidation;
```

Create a `dlquantizer` object and specify the pruned network as the network to quantize.

```
prunedNet  = assembleNetwork([prunedDLNet.Layers ; net.Layers(end-1:end)]);

quantObjPrunedNetwork = dlquantizer(prunedNet,'ExecutionEnvironment','GPU');
```

Use the `calibrate` function to exercise the network with the calibration data and collect range statistics for the weights, biases, and activations at each layer.

```
calResults = calibrate(quantObjPrunedNetwork, calibrationDataStore)
```

```
calResults=18×5 table
        Optimized Layer Name        Network Layer Name    Learnables / Activations    MinValue
    ____________________________    __________________    ________________________    _________

    {'conv_1_relu_1_Weights'   }      {'relu_1'    }          "Weights"                -0.45058
    {'conv_1_relu_1_Bias'      }      {'relu_1'    }          "Bias"                   -0.025525
    {'conv_2_relu_2_Weights'   }      {'relu_2'    }          "Weights"                -0.50476
    {'conv_2_relu_2_Bias'      }      {'relu_2'    }          "Bias"                   -0.080291
    {'conv_3_relu_3_Weights'   }      {'relu_3'    }          "Weights"                -0.42412
    {'conv_3_relu_3_Bias'      }      {'relu_3'    }          "Bias"                   -0.20088
    {'fc_Weights'              }      {'fc'        }          "Weights"                -0.30704
    {'fc_Bias'                 }      {'fc'        }          "Bias"                   -0.23249
    {'imageinput'              }      {'imageinput'}          "Activations"                  0
    {'imageinput_normalization'}      {'imageinput'}          "Activations"                  0
    {'conv_1_relu_1'           }      {'relu_1'    }          "Activations"                  0
    {'maxpool_1'               }      {'maxpool_1' }          "Activations"                  0
    {'conv_2_relu_2'           }      {'relu_2'    }          "Activations"                  0
    {'maxpool_2'               }      {'maxpool_2' }          "Activations"                  0
    {'conv_3_relu_3'           }      {'relu_3'    }          "Activations"                  0
    {'fc'                      }      {'fc'        }          "Activations"              -52.839
      ⋮
```

Use the `validate` function to compare the results of the network before and after quantization using the validation data set.

```
valResults = validate(quantObjPrunedNetwork, validationDataStore);
```

Examine the `MetricResults.Result` field of the validation output to see the accuracy of the quantized network.

```
valResults.MetricResults.Result
```

```
ans=2×2 table
    NetworkImplementation    MetricOutput
    _____________________    ____________

     {'Floating-Point'}           0.94
     {'Quantized'     }         0.9396
```

```
valResults.Statistics
```

```
ans=2×2 table
    NetworkImplementation    LearnableParameterMemory(bytes)
    _____________________    _______________________________

     {'Floating-Point'}                   86320
     {'Quantized'     }                   68824
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating the data over the fourth dimension adds a third dimension to each image to use as a singleton channel dimension.

```
function X = preprocessMiniBatch(XCell)
% Extract image data from cell and concatenate.
X = cat(4,XCell{:});
end
```

Model Accuracy Function

Evaluate the classification accuracy of the `dlnetwork`. Accuracy is the percentage of labels correctly classified by the network.

```
function accuracy = evaluateAccuracy(dlnet,mbqValidation,classes,trueLabels)
YPred = modelPredictions(dlnet,mbqValidation,classes);
accuracy = mean(YPred == trueLabels);
end
```

SynFlow Score Function

The `calculateSynFlowScore` function calculates Synaptic Flow (SynFlow) scores. Synaptic saliency [2] is described as the class of gradient-based scores defined by the product of gradient of loss multiplied by the parameter value:

$$\text{synFlowScore} = \frac{d(\text{loss})}{d\theta} * \theta$$

The SynFlow score is a synaptic saliency score that uses the sum of all network outputs as a loss function:

$$\text{loss} = \sum f(\text{abs}(\theta), X)$$

f is the function represented by the neural network

θ are the parameters of the network

X is the input array to the network

To compute parameter gradients with respect to this loss function, use `dlfeval` and a model gradients function.

```
function score = calculateSynFlowScore(dlnet,dlX)
dlnet.Learnables = dlupdate(@abs, dlnet.Learnables);
gradients = dlfeval(@modelGradients,dlnet,dlX);
score = dlupdate(@(g,w)g.*w, gradients, dlnet.Learnables);
end
```

Model Gradients for SynFlow Score

```
function gradients = modelGradients(dlNet,inputArray)
% Evaluate the gradients on a given input to the dlnetwork
dlYPred = predict(dlNet,inputArray);
pseudoloss = sum(dlYPred,'all');
```

```
gradients = dlgradient(pseudoloss,dlNet.Learnables);
end
```

Magnitude Score Function

The `calculateMagnitudeScore` function returns the magnitude score, defined as the element-wise absolute value of the parameters.

```
function score = calculateMagnitudeScore(dlnet)
score = dlupdate(@abs, dlnet.Learnables);
end
```

Mask Generation Function

The `calculateMask` function returns a binary mask for the network parameters based on the given scores and the target sparsity.

```
function mask = calculateMask(scoresMagnitude,sparsity)
% Compute a binary mask based on the parameter-wise scores such that the mask contains a percenta

% Flatten the cell array of scores into one long score vector
flattenedScores = cell2mat(cellfun(@(S)extractdata(gather(S(:))),scoresMagnitude.Value,'UniformOu
% Rank the scores and determine the threshold for removing connections for the
% given sparsity
flattenedScores = sort(flattenedScores);
k = round(sparsity*numel(flattenedScores));
if k==0
    thresh = 0;
else
    thresh = flattenedScores(k);
end
% Create a binary mask
mask = dlupdate( @(S)S>thresh, scoresMagnitude);
end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object dlnet, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all data in the minibatchqueue object. The function uses the `onehotdecode` function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)
predictions = [];
while hasdata(mbq)
    dlXTest = next(mbq);
    dlYPred = softmax(predict(dlnet,dlXTest));
    YPred = onehotdecode(dlYPred,classes,1)';
    predictions = [predictions; YPred];
end
reset(mbq)
end
```

Apply Pruning Function

The `createPrunedNet` function returns the pruned dlnetwork for the specified pruning algorithm and iteration.

```
function prunedNet = createPrunedNet(dlnet,selectedIteration,pruningMaskSynFlow,pruningMaskMagni
switch pruningMethod
    case "Magnitude"
        prunedNet = dlupdate(@(W,M)W.*M, dlnet, pruningMaskMagnitude{selectedIteration});
    case "SynFlow"
        prunedNet = dlupdate(@(W,M)W.*M, dlnet, pruningMaskSynFlow{selectedIteration});
end
end
```

Pruning Statistics Function

The `pruningStatistics` function extracts detailed layer-level pruning statistics such the layer-level sparsity and the number of filters or neurons being pruned.

sparsityPerLayer - `percentage of parameters pruned in each layer`

prunedChannelsPerLayer - number of channels/neurons in each layer that can be removed as a result of pruning

numOutChannelsPerLayer - number of channels/neurons in each layer

```
function [sparsityPerLayer,prunedChannelsPerLayer,numOutChannelsPerLayer,layerNames] = pruningSta

layerNames = unique(dlnet.Learnables.Layer,'stable');
numLayers = numel(layerNames);
layerIDs = zeros(numLayers,1);
for idx = 1:numel(layerNames)
    layerIDs(idx) = find(layerNames(idx)=={dlnet.Layers.Name});
end

sparsityPerLayer = zeros(numLayers,1);
prunedChannelsPerLayer = zeros(numLayers,1);
numOutChannelsPerLayer = zeros(numLayers,1);

numParams = zeros(numLayers,1);
numPrunedParams = zeros(numLayers,1);
for idx = 1:numLayers
    layer = dlnet.Layers(layerIDs(idx));

    % Calculate the sparsity
    paramIDs = strcmp(dlnet.Learnables.Layer,layerNames(idx));
    paramValue = dlnet.Learnables.Value(paramIDs);
    for p = 1:numel(paramValue)
        numParams(idx) = numParams(idx) + numel(paramValue{p});
        numPrunedParams(idx) = numPrunedParams(idx) + nnz(extractdata(paramValue{p})==0);
    end

    % Calculate channel statistics
    sparsityPerLayer(idx) = numPrunedParams(idx)/numParams(idx);
    switch class(layer)
        case "nnet.cnn.layer.FullyConnectedLayer"
            numOutChannelsPerLayer(idx) = layer.OutputSize;
            prunedChannelsPerLayer(idx) = nnz(all(layer.Weights==0,2)&layer.Bias(:)==0);
        case "nnet.cnn.layer.Convolution2DLayer"
            numOutChannelsPerLayer(idx) = layer.NumFilters;
            prunedChannelsPerLayer(idx) = nnz(reshape(all(layer.Weights==0,[1,2,3]),[],1)&layer.B
        case "nnet.cnn.layer.GroupedConvolution2DLayer"
            numOutChannelsPerLayer(idx) = layer.NumGroups*layer.NumFiltersPerGroup;
```

```
            prunedChannelsPerLayer(idx) = nnz(reshape(all(layer.Weights==0,[1,2,3]),[],1)&layer.B
        otherwise
            error("Unknown layer: "+class(layer))
    end
end
end
```

Load Digits Data set Function

The `loadDigitDataset` function loads the Digits data set and splits the data into training and validation data.

```
function [imdsTrain, imdsValidation] = loadDigitDataset()
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
[imdsTrain, imdsValidation] = splitEachLabel(imds,0.75,"randomized");
end
```

Train Digit Recognition Network Function

The `trainDigitDataNetwork` function trains a convolutional neural network to classify digits in grayscale images.

```
function net = trainDigitDataNetwork(imdsTrain,imdsValidation)
layers = [
    imageInputLayer([28 28 1],"Normalization","rescale-zero-one")
    convolution2dLayer(3,8,'Padding','same')
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

% Specify the training options
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',10, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','none',"ExecutionEnvironment","auto");

% Train network
net = trainNetwork(imdsTrain,layers,options);
end
```

References

[1] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. "Learning Both Weights and Connections for Efficient Neural Networks." Advances in Neural Information Processing Systems 28 (NIPS 2015): 1135–1143.

[2] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli 2020. "Pruning Neural Networks Without Any Data by Iteratively Conserving Synaptic Flow." *34th Conference on Neural Information Processing Systems (NeurlPS 2020)*

Quantization Workflow Prerequisites

This table lists the products required to quantize and deploy deep learning networks.

	Execution Environment		
Development Host Requirements	**FPGA**	**GPU**	**CPU**
Setup Toolkit Environment	`hdlsetuptoolpath` (HDL Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.	"Setting Up the Prerequisite Products" (GPU Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.	"Prerequisites for Deep Learning with MATLAB Coder" (MATLAB Coder) The quantization workflow does not support the `MinGW C/C++` compiler. Use `Microsoft Visual C++ 2017` or `Microsoft Visual C++ 2015`. For a list of supported compilers, see Supported and Compatible Compilers.
Required Products	• Deep Learning Toolbox • Deep Learning HDL Toolbox™	Deep Learning Toolbox	Deep Learning Toolbox
Required Support Packages	• Deep Learning Toolbox Model Quantization Library • Deep Learning HDL Toolbox Support Package for Xilinx® FPGA and SoC Devices • Deep Learning HDL Toolbox Support Package for Intel® FPGA and SoC Devices	Deep Learning Toolbox Model Quantization Library	Deep Learning Toolbox Model Quantization Library
Required Add Ons	MATLAB Coder Interface for Deep Learning Libraries	• GPU Coder Interface for Deep Learning Libraries • CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3 or higher.	MATLAB Coder Interface for Deep Learning Libraries

Supported Networks and Layers	“Supported Networks, Layers, Boards, and Tools” (Deep Learning HDL Toolbox)	“Supported Networks, Layers, and Classes” (GPU Coder)	“Networks and Layers Supported for Code Generation” (MATLAB Coder)
Deployment	Deep Learning HDL Toolbox	GPU Coder	MATLAB Coder

Quantization of Deep Neural Networks

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The data type defines how hardware components or software functions interpret this sequence of 1's and 0's. Numbers are represented as either scaled integer (usually referred to as fixed-point) or floating-point data types.

Most pretrained neural networks and neural networks trained using Deep Learning Toolbox use single-precision floating point data types. Even small trained neural networks require a considerable amount of memory, and require hardware that can perform floating-point arithmetic. These restrictions can inhibit deployment of deep learning capabilities to low-power microcontrollers and FPGAs.

Using the Deep Learning Toolbox Model Quantization Library support package, you can quantize a network to use 8-bit scaled integer data types.

To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites” on page 20-193.

Precision and Range

Scaled 8-bit integer data types have limited precision and range when compared to single-precision floating point data types. There are several numerical considerations when casting a number from a larger floating-point data type to a smaller data type of fixed length.

- Precision loss: Precision loss is a rounding error. When precision loss occurs, the value is rounded to the nearest number that is representable by the data type. In the case of a tie it rounds:
  - Positive numbers to the closest representable value in the direction of positive infinity.
  - Negative numbers to the closest representable value in the direction of negative infinity.

  In MATLAB you can perform this type of rounding using the `round` function.
- Underflow: Underflow is a type of precision loss. Underflows occur when the value is smaller than the smallest value representable by the data type. When this occurs, the value saturates to zero.
- Overflow: When a value is larger than the largest value that a data type can represent, an overflow occurs. When an overflow occurs, the value saturates to the largest value representable by the data type.

Histograms of Dynamic Ranges

Use the **Deep Network Quantizer** app to collect and visualize the dynamic ranges of the weights and biases of the convolution layers and fully connected layers of a network, and the activations of all layers in the network. The app assigns a scaled 8-bit integer data type for the weights, biases, and activations of the convolution layers of the network. The app displays a histogram of the dynamic range for each of these parameters. The following steps describe how these histograms are produced.

1 Consider the following values logged for a parameter while exercising a network.

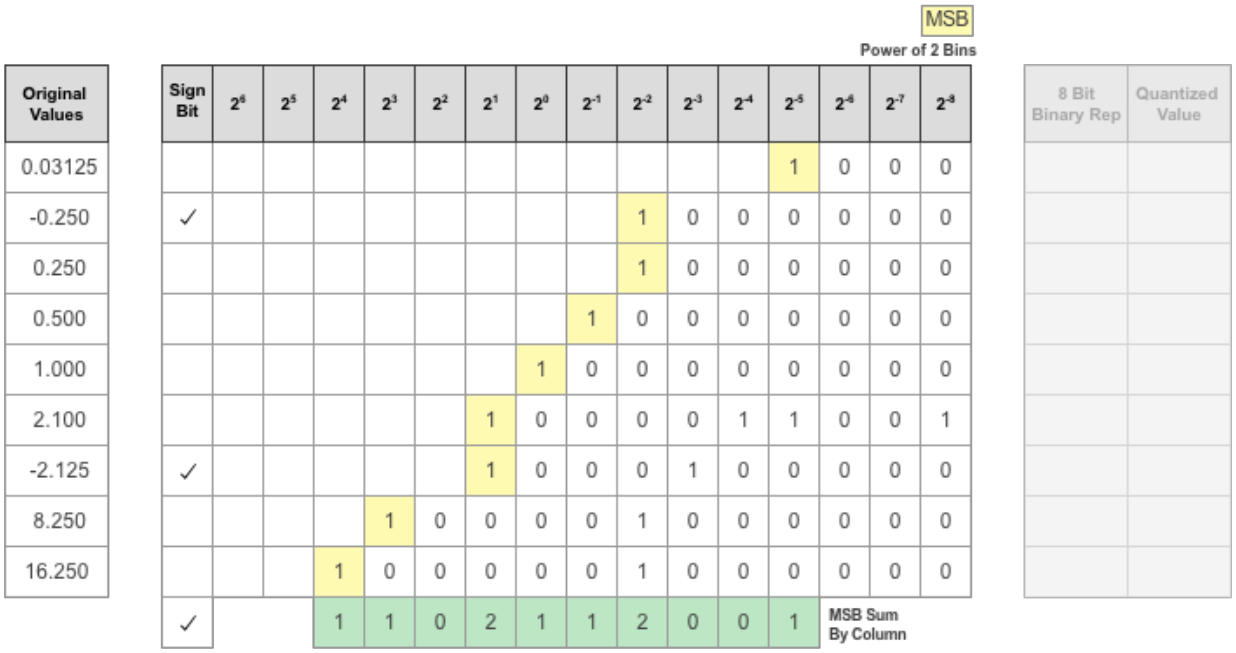
Original Values	Power of 2 Bins																8 Bit Binary Rep	Quantized Value
	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}		
0.03125																		
-0.250																		
0.250																		
0.500																		
1.000																		
2.100																		
-2.125																		
8.250																		
16.250																		

2 Find the ideal binary representation of each logged value of the parameter.

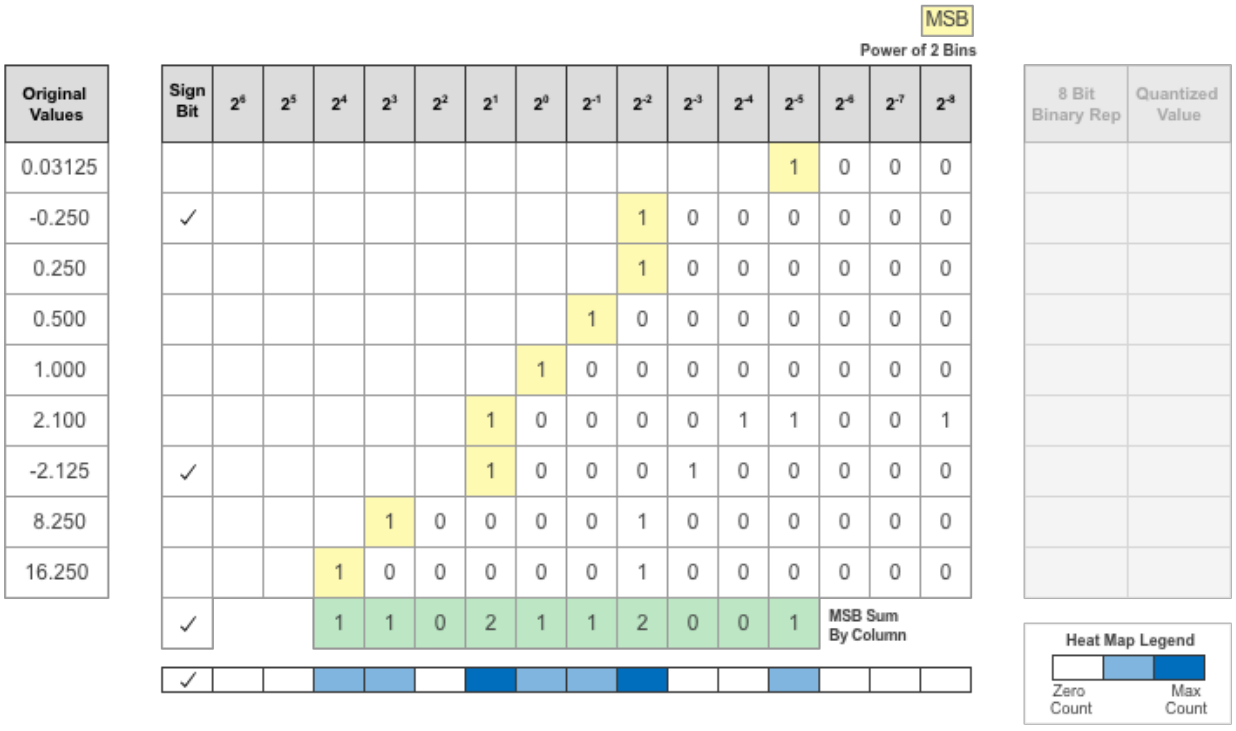
The most significant bit (MSB) is the left-most bit of the binary word. This bit contributes most to the value of the number. The MSB for each value is highlighted in yellow.

													MSB					
													Power of 2 Bins					
Original Values	Sign Bit	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	8 Bit Binary Rep	Quantized Value
0.03125													1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0		
2.100							1	0	0	0	0	1	1	0	0	1		
-2.125	✓						1	0	0	0	1	0	0	0	0	0		
8.250					1	0	0	0	0	1	0	0	0	0	0	0		
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0		

3 By aligning the binary words, you can see the distribution of bits used by the logged values of a parameter. The sum of MSB's in each column, highlighted in green, give an aggregate view of the logged values.

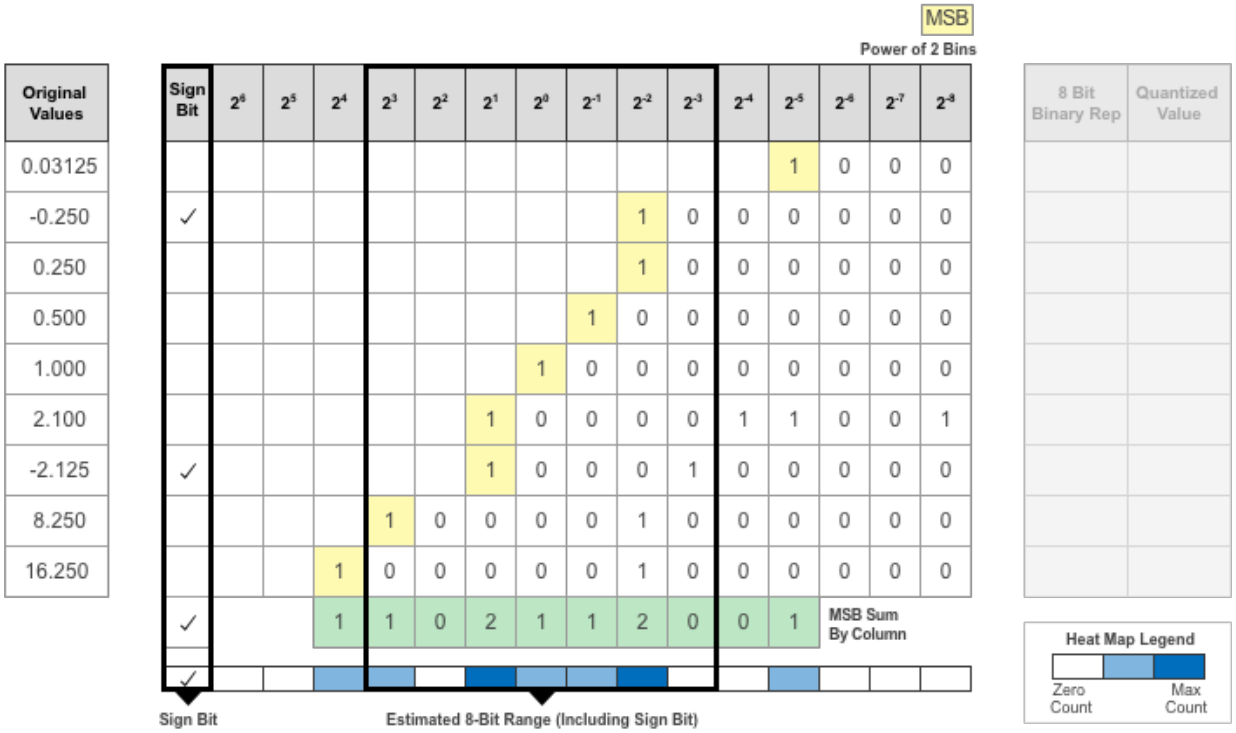


4 The MSB counts of each bit location are displayed as a heat map. In this heat map, darker blue regions correspond to a larger number of MSB's in the bit location.

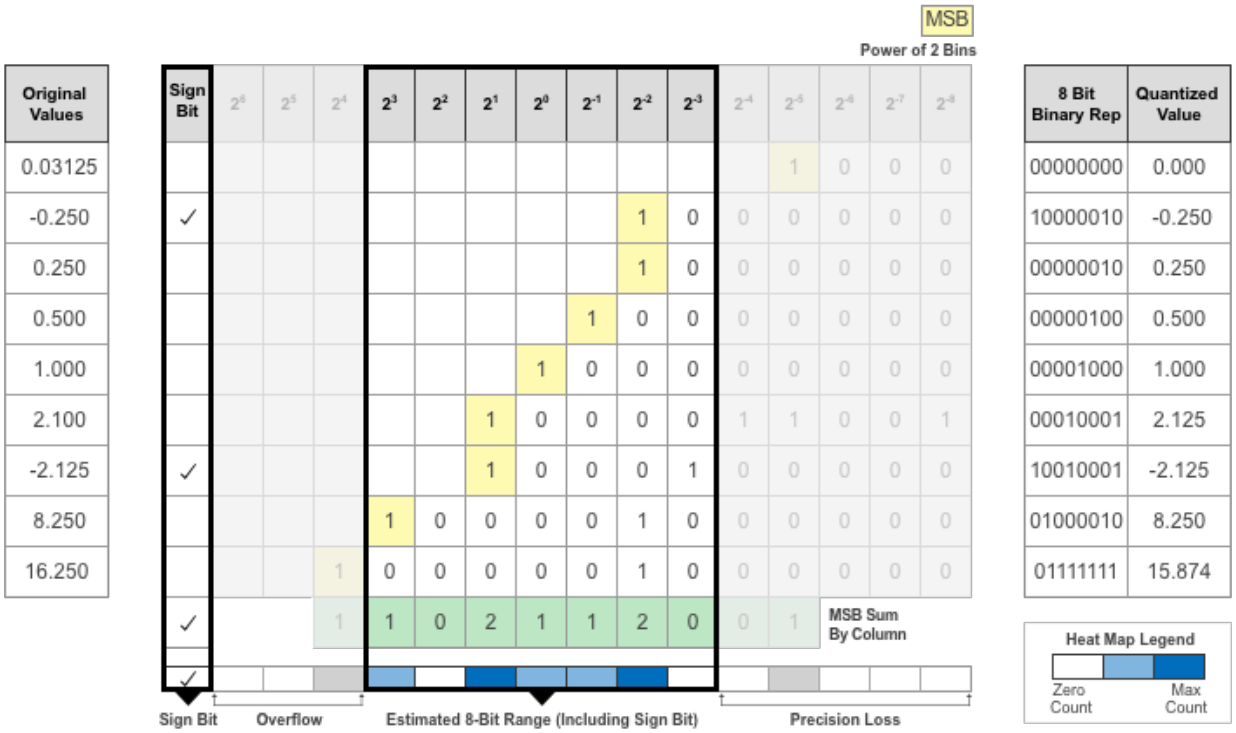


5 The **Deep Network Quantizer** app assigns a data type that can avoid overflow, cover the range, and allow underflow. An additional sign bit is required to represent the signedness of the value.

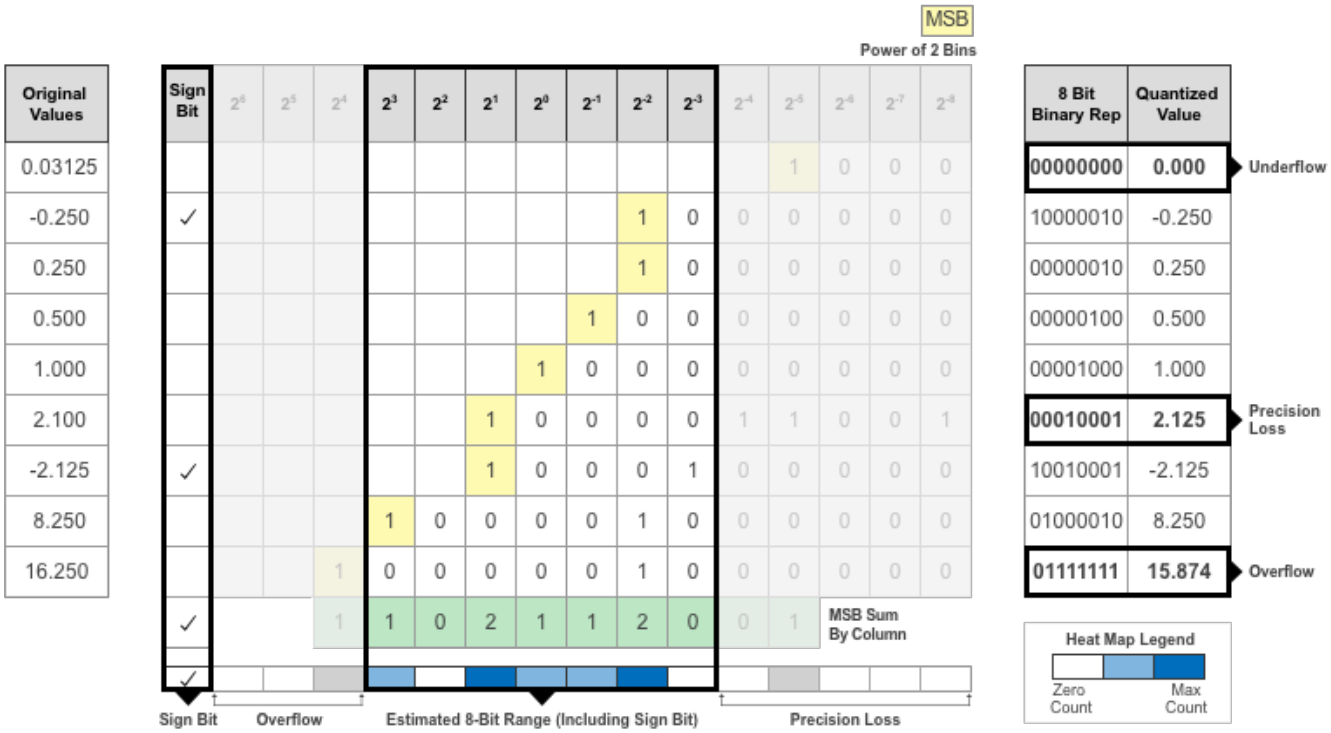
The figure below shows an example of a data type that represents bits from 2^3 to 2^{-3}, including the sign bit.



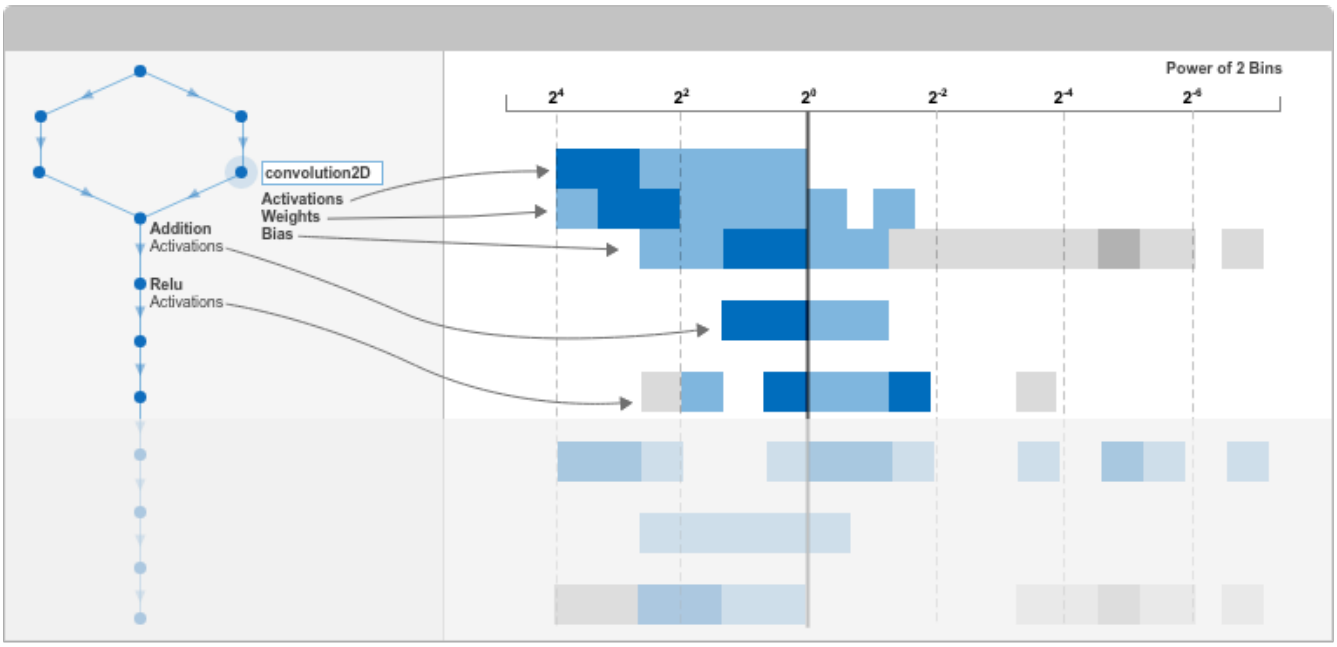
6 After assigning the data type, any bits outside of that data type are removed. Due to the assignment of a smaller data type of fixed length, precision loss, overflow, and underflow can occur for values that are not representable by the data type.



In this example, the value 0.03125, suffers from an underflow, so the quantized value is 0. The value 2.1 suffers some precision loss, so the quantized value is 2.125. The value 16.250 is larger than the largest representable value of the data type, so this value overflows and the quantized value saturates to 15.874.



7 The **Deep Network Quantizer** app displays this heat map histogram for each learnable parameter in the convolution layers and fully connected layers of the network. The gray regions of the histogram show the bits that cannot be represented by the data type.



See Also

Apps
Deep Network Quantizer

Functions
`calibrate` | `validate` | `dlquantizer` | `dlquantizationOptions`

Neural Network Design Book

The developers of the Deep Learning Toolbox software have written a textbook, *Neural Network Design* (Hagan, Demuth, and Beale, ISBN 0-9717321-0-8). The book presents the theory of neural networks, discusses their design and application, and makes considerable use of the MATLAB environment and Deep Learning Toolbox software. Example programs from the book are used in various sections of this documentation. (You can find all the book example programs in the Deep Learning Toolbox software by typing `nnd`.)

Obtain this book from John Stovall at (303) 492-3648, or by email at `John.Stovall@colorado.edu`.

The *Neural Network Design* textbook includes:

- An Instructor's Manual for those who adopt the book for a class
- Transparency Masters for class use

If you are teaching a class and want an Instructor's Manual (with solutions to the book exercises), contact John Stovall at (303) 492-3648, or by email at `John.Stovall@colorado.edu`

To look at sample chapters of the book and to obtain Transparency Masters, go directly to the Neural Network Design page at:

`https://hagan.okstate.edu/nnd.html`

From this link, you can obtain sample book chapters in PDF format and you can download the Transparency Masters by clicking **Transparency Masters (3.6MB)**.

You can get the Transparency Masters in PowerPoint or PDF format.

Neural Network Objects, Data, and Training Styles

Workflow for Neural Network Design

The work flow for the neural network design process has seven primary steps. Referenced topics discuss the basic ideas behind steps 2, 3, and 5.

1. Collect data
2. Create the network — “Create Neural Network Object” on page 21-13
3. Configure the network — “Configure Shallow Neural Network Inputs and Outputs” on page 21-16
4. Initialize the weights and biases
5. Train the network — “Neural Network Training Concepts” on page 21-22
6. Validate the network
7. Use the network

Data collection in step 1 generally occurs outside the framework of Deep Learning Toolbox software, but it is discussed in general terms in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2. Details of the other steps and discussions of steps 4, 6, and 7, are discussed in topics specific to the type of network.

The Deep Learning Toolbox software uses the network object to store all of the information that defines a neural network. This topic describes the basic components of a neural network and shows how they are created and stored in the network object.

After a neural network has been created, it needs to be configured and then trained. Configuration involves arranging the network so that it is compatible with the problem you want to solve, as defined by sample data. After the network has been configured, the adjustable network parameters (called weights and biases) need to be tuned, so that the network performance is optimized. This tuning process is referred to as training the network. Configuration and training require that the network be provided with example data. This topic shows how to format the data for presentation to the network. It also explains network configuration and the two forms of network training: incremental training and batch training.

See Also

More About

- “Four Levels of Neural Network Design” on page 21-3
- “Neuron Model” on page 21-4
- “Neural Network Architectures” on page 21-8
- “Understanding Shallow Network Data Structures” on page 21-18

Four Levels of Neural Network Design

There are four different levels at which the neural network software can be used. The first level is represented by the GUIs that are described in “Get Started with Deep Learning Toolbox”. These provide a quick way to access the power of the toolbox for many problems of function fitting, pattern recognition, clustering and time series analysis.

The second level of toolbox use is through basic command-line operations. The command-line functions use simple argument lists with intelligent default settings for function parameters. (You can override all of the default settings, for increased functionality.) This topic, and the ones that follow, concentrate on command-line operations.

The GUIs described in Getting Started can automatically generate MATLAB code files with the command-line implementation of the GUI operations. This provides a nice introduction to the use of the command-line functionality.

A third level of toolbox use is customization of the toolbox. This advanced capability allows you to create your own custom neural networks, while still having access to the full functionality of the toolbox.

The fourth level of toolbox usage is the ability to modify any of the code files contained in the toolbox. Every computational component is written in MATLAB code and is fully accessible.

The first level of toolbox use (through the GUIs) is described in Getting Started which also introduces command-line operations. The following topics will discuss the command-line operations in more detail. The customization of the toolbox is described in “Define Shallow Neural Network Architectures”.

See Also

More About

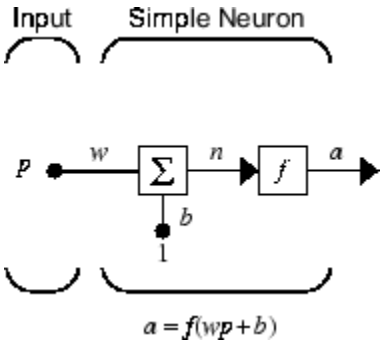
- “Workflow for Neural Network Design” on page 21-2

Neuron Model

In this section...

Simple Neuron

The fundamental building block for neural networks is the single-input neuron, such as this example.



There are three distinct functional operations that take place in this example neuron. First, the scalar input p is multiplied by the scalar weight w to form the product wp, again a scalar. Second, the weighted input wp is added to the scalar bias b to form the net input n. (In this case, you can view the bias as shifting the function f to the left by an amount b. The bias is much like a weight, except that it has a constant input of 1.) Finally, the net input is passed through the transfer function f, which produces the scalar output a. The names given to these three processes are: the weight function, the net input function and the transfer function.

For many types of neural networks, the weight function is a product of a weight times the input, but other weight functions (e.g., the distance between the weight and the input, $|w - p|$) are sometimes used. (For a list of weight functions, type `help nnweight`.) The most common net input function is the summation of the weighted inputs with the bias, but other operations, such as multiplication, can be used. (For a list of net input functions, type `help nnnetinput`.) “Introduction to Radial Basis Neural Networks” on page 25-2 discusses how distance can be used as the weight function and multiplication can be used as the net input function. There are also many types of transfer functions. Examples of various transfer functions are in “Transfer Functions” on page 21-5. (For a list of transfer functions, type `help nntransfer`.)

Note that w and b are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, you can train the network to do a particular job by adjusting the weight or bias parameters.

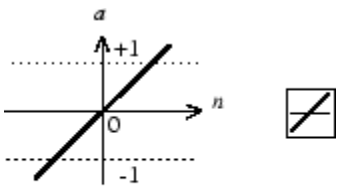
All the neurons in the Deep Learning Toolbox software have provision for a bias, and a bias is used in many of the examples and is assumed in most of this toolbox. However, you can omit a bias in a neuron if you want.

Transfer Functions

Many transfer functions are included in the Deep Learning Toolbox software.

Two of the most commonly used functions are shown below.

The following figure illustrates the linear transfer function.

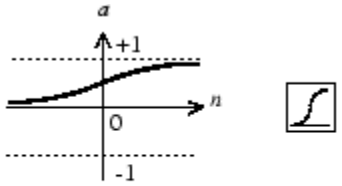


$a = purelin(n)$

Linear Transfer Function

Neurons of this type are used in the final layer of multilayer networks that are used as function approximators. This is shown in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

The sigmoid transfer function shown below takes the input, which can have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



$a = logsig(n)$

Log-Sigmoid Transfer Function

This transfer function is commonly used in the hidden layers of multilayer networks, in part because it is differentiable.

The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons replace the general *f* in the network diagram blocks to show the particular transfer function being used.

For a complete list of transfer functions, type `help nntransfer`. You can also specify your own transfer functions.

You can experiment with a simple neuron and various transfer functions by running the example program `nnd2n1`.

Neuron with Vector Input

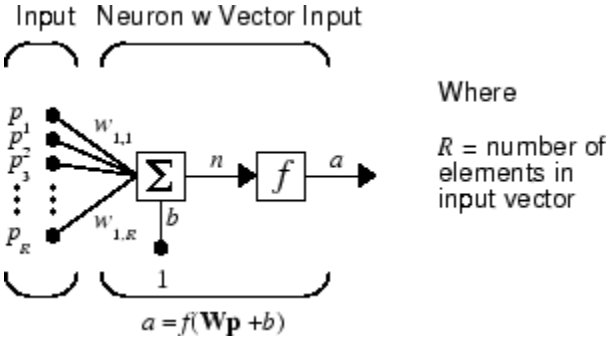
The simple neuron can be extended to handle inputs that are vectors. A neuron with a single *R*-element input vector is shown below. Here the individual input elements

$$p_1, p_2, \ldots p_R$$

are multiplied by weights

$$w_{1,1}, w_{1,2}, \ldots w_{1,R}$$

and the weighted values are fed to the summing junction. Their sum is simply **Wp**, the dot product of the (single row) matrix **W** and the vector **p**. (There are other weight functions, in addition to the dot product, such as the distance between the row of the weight matrix and the input vector, as in “Introduction to Radial Basis Neural Networks” on page 25-2.)



The neuron has a bias b, which is summed with the weighted inputs to form the net input n. (In addition to the summation, other net input functions can be used, such as the multiplication that is used in “Introduction to Radial Basis Neural Networks” on page 25-2.) The net input n is the argument of the transfer function f.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

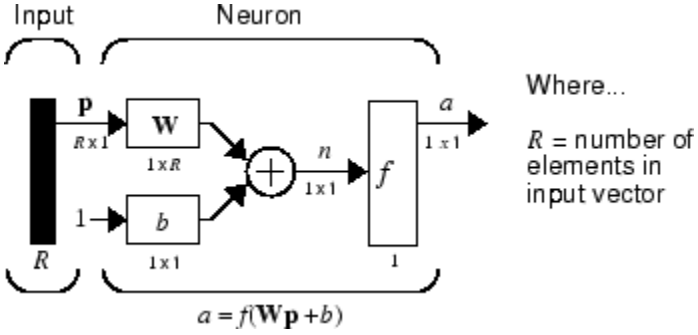
This expression can, of course, be written in MATLAB code as

```
n = W*p + b
```

However, you will seldom be writing code at this level, for such code is already built into functions to define and simulate entire networks.

Abbreviated Notation

The figure of a single neuron shown above contains a lot of detail. When you consider networks with many neurons, and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which is used later in circuits of multiple neurons, is shown here.



Here the input vector **p** is represented by the solid dark vertical bar at the left. The dimensions of **p** are shown below the symbol **p** in the figure as $R \times 1$. (Note that a capital letter, such as R in the

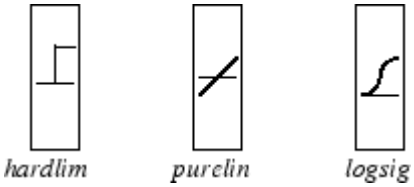
previous sentence, is used when referring to the *size* of a vector.) Thus, **p** is a vector of *R* input elements. These inputs postmultiply the single-row, *R*-column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias *b*. The net input to the transfer function *f* is *n*, the sum of the bias *b* and the product **Wp**. This sum is passed to the transfer function *f* to get the neuron's output *a*, which in this case is a scalar. Note that if there were more than one neuron, the network output would be a vector.

A *layer* of a network is defined in the previous figure. A layer includes the weights, the multiplication and summing operations (here realized as a vector product **Wp**), the bias *b*, and the transfer function *f*. The array of inputs, vector **p**, is not included in or called a layer.

As with the “Simple Neuron” on page 21-4, there are three operations that take place in the layer: the weight function (matrix multiplication, or dot product, in this case), the net input function (summation, in this case), and the transfer function.

Each time this abbreviated network notation is used, the sizes of the matrices are shown just below their matrix variable names. This notation will allow you to understand the architectures and follow the matrix mathematics associated with them.

As discussed in “Transfer Functions” on page 21-5, when a specific transfer function is to be used in a figure, the symbol for that transfer function replaces the *f* shown above. Here are some examples.



You can experiment with a two-element neuron by running the example program nnd2n2.

See Also

More About

- “Neural Network Architectures” on page 21-8
- “Workflow for Neural Network Design” on page 21-2

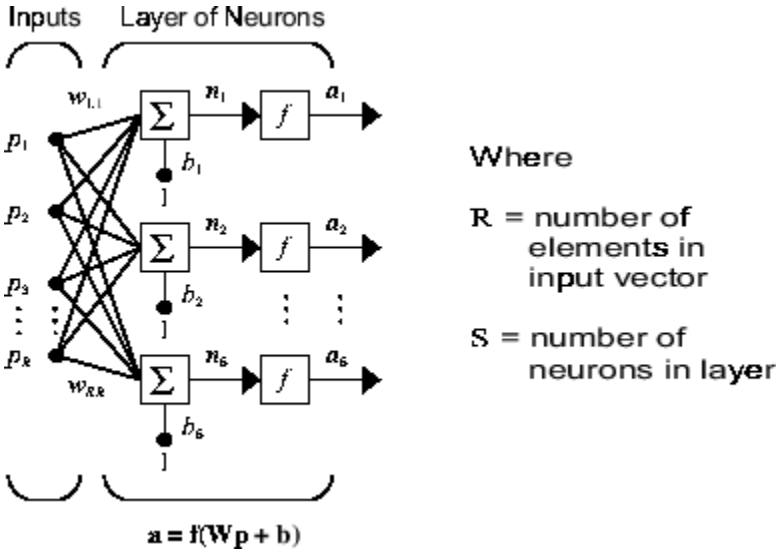
Neural Network Architectures

In this section...

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

One Layer of Neurons

A one-layer network with R input elements and S neurons follows.



In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith neuron has a summer that gathers its weighted inputs and bias to form its own scalar output $n(i)$. The various $n(i)$ taken together form an S-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. The expression for **a** is shown at the bottom of the figure.

Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., R is not necessarily equal to S). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

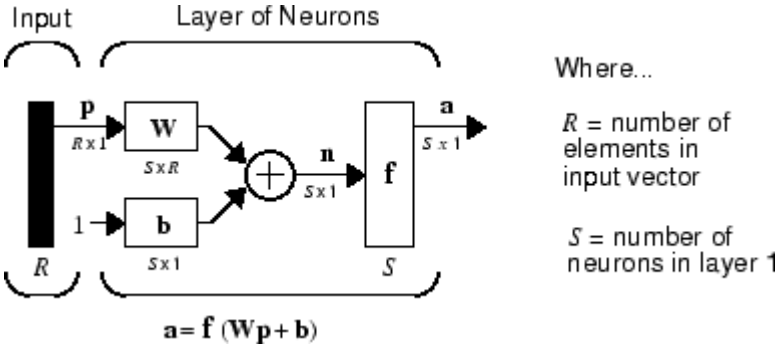
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in $w_{1,2}$ say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R-input one-layer network also can be drawn in abbreviated notation.

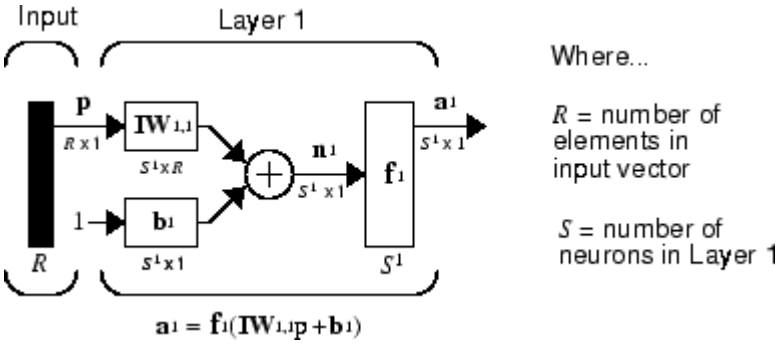


Here **p** is an R-length input vector, **W** is an $S \times R$ matrix, **a** and **b** are S-length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function blocks.

Inputs and Layers

To describe networks having multiple layers, the notation must be extended. Specifically, it needs to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. It also needs to identify the source and destination for the weight matrices.

We will call weight matrices connected to inputs *input weights;* we will call weight matrices connected to layer outputs *layer weights.* Further, superscripts are used to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, the one-layer multiple input network shown earlier is redrawn in abbreviated form here.

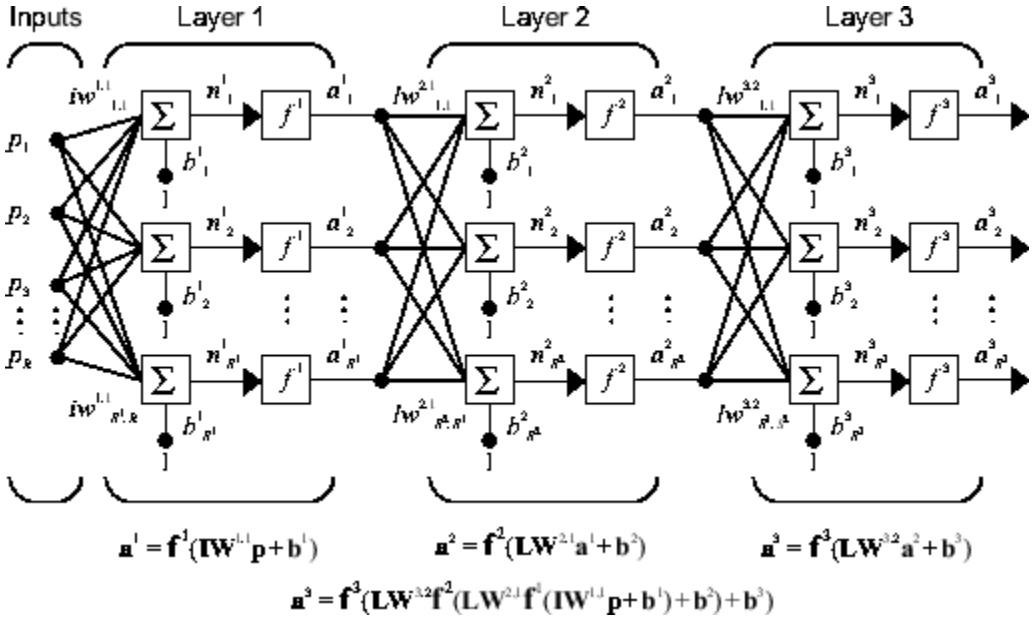


As you can see, the weight matrix connected to the input vector **p** is labeled as an input weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

“Multiple Layers of Neurons” on page 21-10 uses layer weight (**LW**) matrices as well as input weight (**IW**) matrices.

Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in the figures, the number of the layer is appended as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown next, and in the equations at the bottom of the figure.



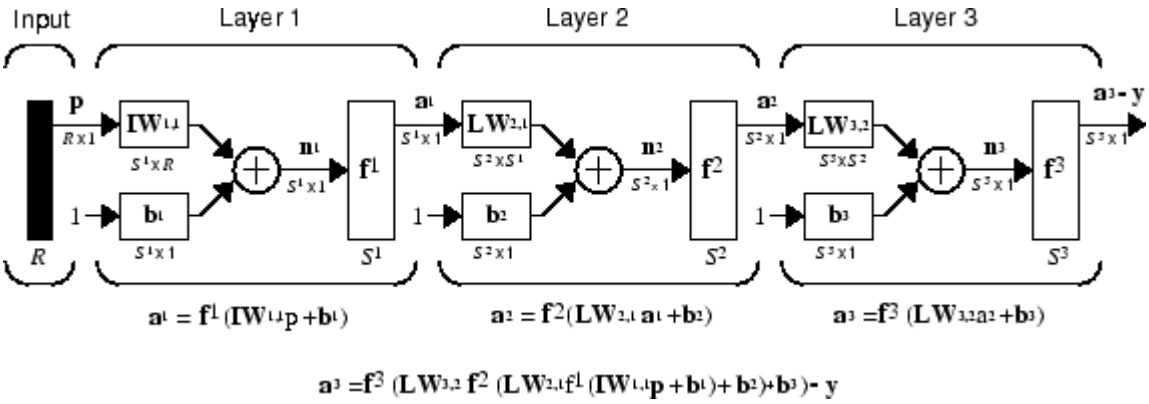
The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S^1 inputs, S^2 neurons, and an $S^2 \times S^1$ weight matrix $\mathbf{W}^2$. The input to layer 2 is $\mathbf{a}^1$; the output is $\mathbf{a}^2$. Now that all the vectors and matrices of layer 2 have been identified, it can be treated as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an *output layer*. All other layers are called *hidden layers*. The three-layer network shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. This toolbox does not use that designation.

The architecture of a multilayer network with a single input vector can be specified with the notation $R - S^1 - S^2 - \ldots - S^M$, where the number of elements of the input vector and the number of neurons in each layer are specified.

The same three-layer network can also be drawn using abbreviated notation.



Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

Here it is assumed that the output of the third layer, $\mathbf{a}^3$, is the network output of interest, and this output is labeled as $\mathbf{y}$. This notation is used to specify the output of multilayer networks.

Input and Output Processing Functions

Network inputs might have associated processing functions. Processing functions transform user input data to a form that is easier or more efficient for a network.

For instance, `mapminmax` transforms input data so that all values fall into the interval [−1, 1]. This can speed up learning for many networks. `removeconstantrows` removes the rows of the input vector that correspond to input elements that always have the same value, because these input elements are not providing any useful information to the network. The third common processing function is `fixunknowns`, which recodes unknown data (represented in the user's data with `NaN` values) into a numerical form for the network. `fixunknowns` preserves information about which values are known and which are unknown.

Similarly, network outputs can also have associated processing functions. Output processing functions are used to transform user-provided target vectors for network use. Then, network outputs are reverse-processed using the same functions to produce output data with the same characteristics as the original user-provided targets.

Both `mapminmax` and `removeconstantrows` are often associated with network outputs. However, `fixunknowns` is not. Unknown values in targets (represented by `NaN` values) do not need to be altered for network use.

Processing functions are described in more detail in “Choose Neural Network Input-Output Processing Functions” on page 22-7.

See Also

More About

- “Neuron Model” on page 21-4

- “Workflow for Neural Network Design” on page 21-2

Create Neural Network Object

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 21-2.

The easiest way to create a neural network is to use one of the network creation functions. To investigate how this is done, you can create a simple, two-layer feedforward network, using the command `feedforwardnet`:

```
net = feedforwardnet

net =

Neural Network

              name: 'Feed-Forward Neural Network'
          userdata: (your custom info)

    dimensions:

         numInputs: 1
         numLayers: 2
        numOutputs: 1
    numInputDelays: 0
    numLayerDelays: 0
 numFeedbackDelays: 0
 numWeightElements: 10
        sampleTime: 1

    connections:

       biasConnect: [1; 1]
      inputConnect: [1; 0]
      layerConnect: [0 0; 1 0]
     outputConnect: [0 1]

    subobjects:

            inputs: {1x1 cell array of 1 input}
            layers: {2x1 cell array of 2 layers}
           outputs: {1x2 cell array of 1 output}
            biases: {2x1 cell array of 2 biases}
      inputWeights: {2x1 cell array of 1 weight}
      layerWeights: {2x2 cell array of 1 weight}

    functions:

          adaptFcn: 'adaptwb'
        adaptParam: (none)
          derivFcn: 'defaultderiv'
         divideFcn: 'dividerand'
       divideParam: .trainRatio, .valRatio, .testRatio
        divideMode: 'sample'
           initFcn: 'initlay'
        performFcn: 'mse'
      performParam: .regularization, .normalization
          plotFcns: {'plotperform', plottrainstate, ploterrhist,
```

```
                  plotregression}
        plotParams: {1x4 cell array of 4 params}
          trainFcn: 'trainlm'
        trainParam: .showWindow, .showCommandLine, .show, .epochs,
                    .time, .goal, .min_grad, .max_fail, .mu, .mu_dec,
                    .mu_inc, .mu_max

    weight and bias values:

                IW: {2x1 cell} containing 1 input weight matrix
                LW: {2x2 cell} containing 1 layer weight matrix
                 b: {2x1 cell} containing 2 bias vectors

    methods:

             adapt: Learn while in continuous use
         configure: Configure inputs & outputs
            gensim: Generate Simulink model
              init: Initialize weights & biases
           perform: Calculate performance
               sim: Evaluate network outputs given inputs
             train: Train network with examples
              view: View diagram
       unconfigure: Unconfigure inputs & outputs

    evaluate:       outputs = net(inputs)
```

This display is an overview of the network object, which is used to store all of the information that defines a neural network. There is a lot of detail here, but there are a few key sections that can help you to see how the network object is organized.

The dimensions section stores the overall structure of the network. Here you can see that there is one input to the network (although the one input can be a vector containing many elements), one network output, and two layers.

The connections section stores the connections between components of the network. For example, there is a bias connected to each layer, the input is connected to layer 1, and the output comes from layer 2. You can also see that layer 1 is connected to layer 2. (The rows of `net.layerConnect` represent the destination layer, and the columns represent the source layer. A one in this matrix indicates a connection, and a zero indicates no connection. For this example, there is a single one in element 2,1 of the matrix.)

The key subobjects of the network object are `inputs`, `layers`, `outputs`, `biases`, `inputWeights`, and `layerWeights`. View the `layers` subobject for the first layer with the command

```
net.layers{1}
```

```
Neural Network Layer

                name: 'Hidden'
          dimensions: 10
         distanceFcn: (none)
       distanceParam: (none)
           distances: []
             initFcn: 'initnw'
         netInputFcn: 'netsum'
       netInputParam: (none)
```

```
          positions: []
              range: [10x2 double]
               size: 10
        topologyFcn: (none)
        transferFcn: 'tansig'
     transferParam: (none)
          userdata: (your custom info)
```

The number of neurons in a layer is given by its `size` property. In this case, the layer has 10 neurons, which is the default size for the `feedforwardnet` command. The net input function is `netsum` (summation) and the transfer function is the `tansig`. If you wanted to change the transfer function to `logsig`, for example, you could execute the command:

```
net.layers{1}.transferFcn = 'logsig';
```

To view the `layerWeights` subobject for the weight between layer 1 and layer 2, use the command:

```
net.layerWeights{2,1}
```

```
Neural Network Weight

            delays: 0
           initFcn: (none)
        initConfig: .inputSize
             learn: true
          learnFcn: 'learngdm'
        learnParam: .lr, .mc
              size: [0 10]
         weightFcn: 'dotprod'
       weightParam: (none)
          userdata: (your custom info)
```

The weight function is `dotprod`, which represents standard matrix multiplication (dot product). Note that the size of this layer weight is 0-by-10. The reason that we have zero rows is because the network has not yet been configured for a particular data set. The number of output neurons is equal to the number of rows in your target vector. During the configuration process, you will provide the network with example inputs and targets, and then the number of output neurons can be assigned.

This gives you some idea of how the network object is organized. For many applications, you will not need to be concerned about making changes directly to the network object, since that is taken care of by the network creation functions. It is usually only when you want to override the system defaults that it is necessary to access the network object directly. Other topics will show how this is done for particular networks and training methods.

To investigate the network object in more detail, you might find that the object listings, such as the one shown above, contain links to help on each subobject. Click the links, and you can selectively investigate those parts of the object that are of interest to you.

Configure Shallow Neural Network Inputs and Outputs

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 21-2.

After a neural network has been created, it must be configured. The configuration step consists of examining input and target data, setting the network's input and output sizes to match the data, and choosing settings for processing inputs and outputs that will enable best network performance. The configuration step is normally done automatically, when the training function is called. However, it can be done manually, by using the configuration function. For example, to configure the network you created previously to approximate a sine function, issue the following commands:

```
p = -2:.1:2;
t = sin(pi*p/2);
net1 = configure(net,p,t);
```

You have provided the network with an example set of inputs and targets (desired network outputs). With this information, the `configure` function can set the network input and output sizes to match the data.

After the configuration, if you look again at the weight between layer 1 and layer 2, you can see that the dimension of the weight is 1 by 20. This is because the target for this network is a scalar.

```
net1.layerWeights{2,1}

    Neural Network Weight

            delays: 0
           initFcn: (none)
        initConfig: .inputSize
             learn: true
          learnFcn: 'learngdm'
        learnParam: .lr, .mc
              size: [1 10]
         weightFcn: 'dotprod'
       weightParam: (none)
          userdata: (your custom info)
```

In addition to setting the appropriate dimensions for the weights, the configuration step also defines the settings for the processing of inputs and outputs. The input processing can be located in the `inputs` subobject:

```
net1.inputs{1}

    Neural Network Input

              name: 'Input'
    feedbackOutput: []
       processFcns: {'removeconstantrows', mapminmax}
     processParams: {1x2 cell array of 2 params}
   processSettings: {1x2 cell array of 2 settings}
     processedRange: [1x2 double]
      processedSize: 1
              range: [1x2 double]
               size: 1
          userdata: (your custom info)
```

Before the input is applied to the network, it will be processed by two functions: `removeconstantrows` and `mapminmax`. These are discussed fully in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2 so we won't address the particulars here. These processing functions may have some processing parameters, which are contained in the subobject `net1.inputs{1}.processParam`. These have default values that you can override. The processing functions can also have configuration settings that are dependent on the sample data. These are contained in `net1.inputs{1}.processSettings` and are set during the configuration process. For example, the `mapminmax` processing function normalizes the data so that all inputs fall in the range [−1, 1]. Its configuration settings include the minimum and maximum values in the sample data, which it needs to perform the correct normalization. This will be discussed in much more depth in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

As a general rule, we use the term “parameter,” as in process parameters, training parameters, etc., to denote constants that have default values that are assigned by the software when the network is created (and which you can override). We use the term “configuration setting,” as in process configuration setting, to denote constants that are assigned by the software from an analysis of sample data. These settings do not have default values, and should not generally be overridden.

For more information, see also “Understanding Shallow Network Data Structures” on page 21-18.

Understanding Shallow Network Data Structures

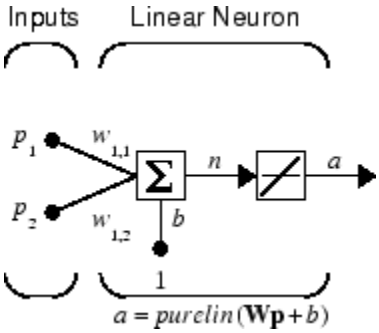
In this section...

This topic discusses how the format of input data structures affects the simulation of networks. It starts with static networks, and then continues with dynamic networks. The following section describes how the format of the data structures affects network training.

There are two basic types of input vectors: those that occur *concurrently* (at the same time, or in no particular time sequence), and those that occur *sequentially* in time. For concurrent vectors, the order is not important, and if there were a number of networks running in parallel, you could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

Simulation with Concurrent Inputs in a Static Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, you need not be concerned about whether or not the input vectors occur in a particular time sequence, so you can treat the inputs as concurrent. In addition, the problem is made even simpler by assuming that the network has only one input vector. Use the following network as an example.



To set up this linear feedforward network, use the following commands:

```
net = linearlayer;
net.inputs{1}.size = 2;
net.layers{1}.dimensions = 1;
```

For simplicity, assign the weight matrix and bias to be **W** = [1 2] and b = [0].

The commands for these assignments are

```
net.IW{1,1} = [1 2];
net.b{1} = 0;
```

Suppose that the network simulation data set consists of Q = 4 concurrent vectors:

$$\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}2\\1\end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix}2\\3\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}3\\1\end{bmatrix}$$

Concurrent vectors are presented to the network as a single matrix:

```
P = [1 2 2 3; 2 1 3 1];
```

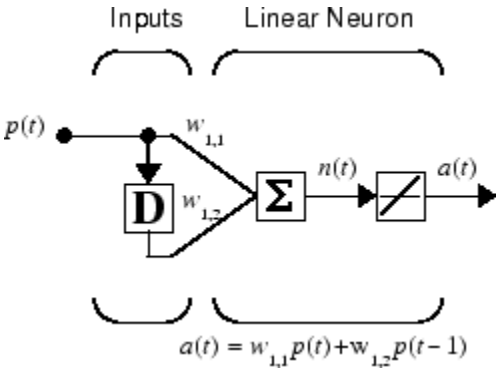
You can now simulate the network:

```
A = net(P)
A =
     5     4     8     5
```

A single matrix of concurrent vectors is presented to the network, and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important, because they do not interact with each other.

Simulation with Sequential Inputs in a Dynamic Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, the next figure shows a simple network that contains one delay.



The following commands create this network:

```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
```

Assign the weight matrix to be **W** = [1 2].

The command is:

```
net.IW{1,1} = [1 2];
```

Suppose that the input sequence is:

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

Sequential inputs are presented to the network as elements of a cell array:

```
P = {1 2 3 4};
```

You can now simulate the network:

```
A = net(P)
A =
    [1]    [4]    [7]    [10]
```

You input a cell array containing a sequence of inputs, and the network produces a cell array containing a sequence of outputs. The order of the inputs is important when they are presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If you were to change the order of the inputs, the numbers obtained in the output would change.

Simulation with Concurrent Inputs in a Dynamic Network

If you were to apply the same inputs as a set of concurrent inputs instead of a sequence of inputs, you would obtain a completely different response. (However, it is not clear why you would want to do this with a dynamic network.) It would be as if each input were applied concurrently to a separate parallel network. For the previous example, “Simulation with Sequential Inputs in a Dynamic Network” on page 21-19, if you use a concurrent set of inputs you have

$$\mathbf{p}_1 = [1], \mathbf{p}_2 = [2], \mathbf{p}_3 = [3], \mathbf{p}_4 = [4]$$

which can be created with the following code:

```
P = [1 2 3 4];
```

When you simulate with concurrent inputs, you obtain

```
A = net(P)
A =
     1     2     3     4
```

The result is the same as if you had concurrently applied each one of the inputs to a separate network and computed one output. Note that because you did not assign any initial conditions to the network delays, they were assumed to be 0. For this case the output is simply 1 times the input, because the weight that multiplies the current input is 1.

In certain special cases, you might want to simulate the network response to several different sequences at the same time. In this case, you would want to present the network with a concurrent set of sequences. For example, suppose you wanted to present the following two sequences to the network:

$$\mathbf{p}_1(1) = [1], \mathbf{p}_1(2) = [2], \mathbf{p}_1(3) = [3], \mathbf{p}_1(4) = [4]$$
$$\mathbf{p}_2(1) = [4], \mathbf{p}_2(2) = [3], \mathbf{p}_2(3) = [2], \mathbf{p}_2(4) = [1]$$

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

```
P = {[1 4] [2 3] [3 2] [4 1]};
```

You can now simulate the network:

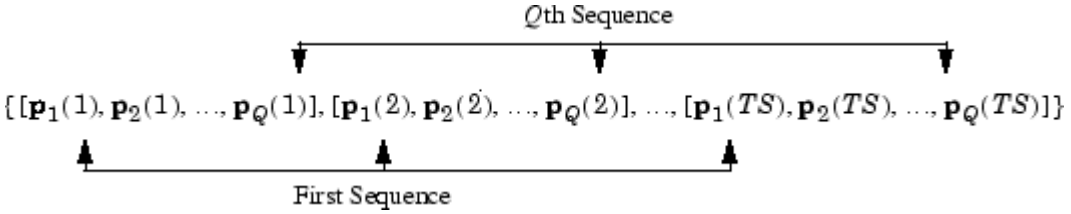
```
A = net(P);
```

The resulting network output would be

```
A = {[1 4] [4 11] [7 8] [10 5]}
```

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the network input P when there are *Q* concurrent sequences of *TS* time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.



In this topic, you apply sequential and concurrent inputs to dynamic networks. In “Simulation with Concurrent Inputs in a Static Network” on page 21-18, you applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It does not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in “Neural Network Training Concepts” on page 21-22.

See also “Configure Shallow Neural Network Inputs and Outputs” on page 21-16.

Neural Network Training Concepts

In this section...

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 21-2.

This topic describes two different styles of training. In *incremental* training the weights and biases of the network are updated each time an input is presented to the network. In *batch* training the weights and biases are only updated after all the inputs are presented. The batch training methods are generally more efficient in the MATLAB environment, and they are emphasized in the Deep Learning Toolbox software, but there some applications where incremental training can be useful, so that paradigm is implemented as well.

Incremental Training with adapt

Incremental training can be applied to both static and dynamic networks, although it is more commonly used with dynamic networks, such as adaptive filters. This section illustrates how incremental training is performed on both static and dynamic networks.

Incremental Training of Static Networks

Consider again the static network used for the first example. You want to train it incrementally, so that the weights and biases are updated after each input is presented. In this case you use the function `adapt`, and the inputs and targets are presented as sequences.

Suppose you want to train the network to create the linear function:

$$t = 2p_1 + p_2$$

Then for the previous inputs,

$$\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix}2\\1\end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix}2\\3\end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix}3\\1\end{bmatrix}$$

the targets would be

$$\mathbf{t}_1 = [4], \mathbf{t}_2 = [5], \mathbf{t}_3 = [7], \mathbf{t}_4 = [7]$$

For incremental training, you present the inputs and targets as sequences:

```
P = {[1;2] [2;1] [2;3] [3;1]};
T = {4 5 7 7};
```

First, set up the network with zero initial weights and biases. Also, set the initial learning rate to zero to show the effect of incremental training.

```
net = linearlayer(0,0);
net = configure(net,P,T);
```

```
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Recall from “Simulation with Concurrent Inputs in a Static Network” on page 21-18 that, for a static network, the simulation of the network produces the same outputs whether the inputs are presented as a matrix of concurrent vectors or as a cell array of sequential vectors. However, this is not true when training the network. When you use the `adapt` function, if the inputs are presented as a cell array of sequential vectors, then the weights are updated as each input is presented (incremental mode). As shown in the next section, if the inputs are presented as a matrix of concurrent vectors, then the weights are updated only after all inputs are presented (batch mode).

You are now ready to train the network incrementally.

```
[net,a,e,pf] = adapt(net,P,T);
```

The network outputs remain zero, because the learning rate is zero, and the weights are not updated. The errors are equal to the targets:

```
a = [0]    [0]    [0]    [0]
e = [4]    [5]    [7]    [7]
```

If you now set the learning rate to 0.1 you can see how the network is adjusted as each input is presented:

```
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1,1}.learnParam.lr = 0.1;
[net,a,e,pf] = adapt(net,P,T);
a = [0]    [2]    [6]    [5.8]
e = [4]    [3]    [1]    [1.2]
```

The first output is the same as it was with zero learning rate, because no update is made until the first input is presented. The second output is different, because the weights have been updated. The weights continue to be modified as each error is computed. If the network is capable and the learning rate is set correctly, the error is eventually driven to zero.

Incremental Training with Dynamic Networks

You can also train dynamic networks incrementally. In fact, this would be the most common situation.

To train the network incrementally, present the inputs and targets as elements of cell arrays. Here are the initial input `Pi` and the inputs `P` and targets `T` as elements of cell arrays.

```
Pi = {1};
P = {2 3 4};
T = {3 5 7};
```

Take the linear network with one delay at the input, as used in a previous example. Initialize the weights to zero and set the learning rate to 0.1.

```
net = linearlayer([0 1],0.1);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
```

You want to train the network to create the current output by summing the current and the previous inputs. This is the same input sequence you used in the previous example with the exception that you

assign the first term in the sequence as the initial condition for the delay. You can now sequentially train the network using `adapt`.

```
[net,a,e,pf] = adapt(net,P,T,Pi);
a = [0] [2.4] [7.98]
e = [3] [2.6] [-0.98]
```

The first output is zero, because the weights have not yet been updated. The weights change at each subsequent time step.

Batch Training

Batch training, in which weights and biases are only updated after all the inputs and targets are presented, can be applied to both static and dynamic networks. Both types of networks are discussed in this section.

Batch Training with Static Networks

Batch training can be done using either `adapt` or `train`, although `train` is generally the best option, because it typically has access to more efficient training algorithms. Incremental training is usually done with `adapt`; batch training is usually done with `train`.

For batch training of a static network with `adapt`, the input vectors must be placed in one matrix of concurrent vectors.

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

Begin with the static network used in previous examples. The learning rate is set to 0.01.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

When you call `adapt`, it invokes `trains` (the default adaption function for the linear network) and `learnwh` (the default learning function for the weights and biases). `trains` uses Widrow-Hoff learning.

```
[net,a,e,pf] = adapt(net,P,T);
a = 0 0 0 0
e = 4 5 7 7
```

Note that the outputs of the network are all zero, because the weights are not updated until all the training set has been presented. If you display the weights, you find

```
net.IW{1,1}
  ans = 0.4900 0.4100
net.b{1}
  ans =
    0.2300
```

This is different from the result after one pass of `adapt` with incremental updating.

Now perform the same batch training using `train`. Because the Widrow-Hoff rule can be used in incremental or batch mode, it can be invoked by `adapt` or `train`. (There are several algorithms that

can only be used in batch mode (e.g., Levenberg-Marquardt), so these algorithms can only be invoked by `train`.)

For this case, the input vectors can be in a matrix of concurrent vectors or in a cell array of sequential vectors. Because the network is static and because `train` always operates in batch mode, `train` converts any cell array of sequential vectors to a matrix of concurrent vectors. Concurrent mode operation is used whenever possible because it has a more efficient implementation in MATLAB code:

```
P = [1 2 2 3; 2 1 3 1];
T = [4 5 7 7];
```

The network is set up in the same way.

```
net = linearlayer(0,0.01);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.b{1} = 0;
```

Now you are ready to train the network. Train it for only one epoch, because you used only one pass of `adapt`. The default training function for the linear network is `trainb`, and the default learning function for the weights and biases is `learnwh`, so you should get the same results obtained using `adapt` in the previous example, where the default adaption function was `trains`.

```
net.trainParam.epochs = 1;
net = train(net,P,T);
```

If you display the weights after one epoch of training, you find

```
net.IW{1,1}
  ans = 0.4900 0.4100
net.b{1}
  ans =
    0.2300
```

This is the same result as the batch mode training in `adapt`. With static networks, the `adapt` function can implement incremental or batch training, depending on the format of the input data. If the data is presented as a matrix of concurrent vectors, batch training occurs. If the data is presented as a sequence, incremental training occurs. This is not true for `train`, which always performs batch training, regardless of the format of the input.

Batch Training with Dynamic Networks

Training static networks is relatively straightforward. If you use `train` the network is trained in batch mode and the inputs are converted to concurrent vectors (columns of a matrix), even if they are originally passed as a sequence (elements of a cell array). If you use `adapt`, the format of the input determines the method of training. If the inputs are passed as a sequence, then the network is trained in incremental mode. If the inputs are passed as concurrent vectors, then batch mode training is used.

With dynamic networks, batch mode training is typically done with `train` only, especially if only one training sequence exists. To illustrate this, consider again the linear network with a delay. Use a learning rate of 0.02 for the training. (When using a gradient descent algorithm, you typically use a smaller learning rate for batch mode training than incremental training, because all the individual gradients are summed before determining the step change to the weights.)

```
net = linearlayer([0 1],0.02);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
Pi = {1};
P = {2 3 4};
T = {3 5 6};
```

You want to train the network with the same sequence used for the incremental training earlier, but this time you want to update the weights only after all the inputs are applied (batch mode). The network is simulated in sequential mode, because the input is a sequence, but the weights are updated in batch mode.

```
net = train(net,P,T,Pi);
```

The weights after one epoch of training are

```
net.IW{1,1}
ans = 0.9000    0.6200
```

These are different weights than you would obtain using incremental training, where the weights would be updated three times during one pass through the training set. For batch training the weights are only updated once in each epoch.

Training Feedback

The `showWindow` parameter allows you to specify whether a training window is visible when you train. The training window appears by default. Two other parameters, `showCommandLine` and `show`, determine whether command-line output is generated and the number of epochs between command-line feedback during training. For instance, this code turns off the training window and gives you training status information every 35 epochs when the network is later trained with `train`:

```
net.trainParam.showWindow = false;
net.trainParam.showCommandLine = true;
net.trainParam.show= 35;
```

Sometimes it is convenient to disable all training displays. To do that, turn off both the training window and command-line feedback:

```
net.trainParam.showWindow = false;
net.trainParam.showCommandLine = false;
```

The training window appears automatically when you train.

Multilayer Shallow Neural Networks and Backpropagation Training

Multilayer Shallow Neural Networks and Backpropagation Training

The shallow multilayer feedforward neural network can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems, as discussed in “Design Time Series Time-Delay Neural Networks” on page 23-10. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network.

Note The training functions described in this topic are not limited to multilayer networks. They can be used to train arbitrary architectures (even custom networks), as long as their components are differentiable.

The work flow for the general neural network design process has seven primary steps:

1. Collect data
2. Create the network
3. Configure the network
4. Initialize the weights and biases
5. Train the network
6. Validate the network (post-training analysis)
7. Use the network

Step 1 might happen outside the framework of Deep Learning Toolbox software, but this step is critical to the success of the design process.

Details of this workflow are discussed in these sections:

- “Multilayer Shallow Neural Network Architecture” on page 22-3
- “Prepare Data for Multilayer Shallow Neural Networks” on page 22-6
- “Create, Configure, and Initialize Multilayer Shallow Neural Networks” on page 22-11
- “Train and Apply Multilayer Shallow Neural Networks” on page 22-13
- “Analyze Shallow Neural Network Performance After Training” on page 22-18
- “Use the Network” on page 22-17
- “Limitations and Cautions” on page 22-22

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 22-7
- “Divide Data for Optimal Neural Network Training” on page 22-9
- “Shallow Neural Networks with Parallel and GPU Computing” on page 28-2

For time series, dynamic modeling, and prediction, see this section:

- “How Dynamic Neural Networks Work” on page 23-3

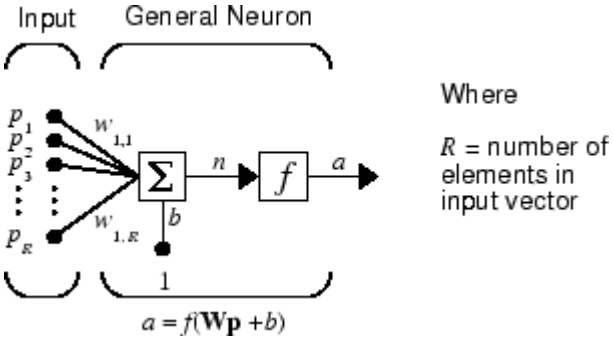
Multilayer Shallow Neural Network Architecture

In this section...

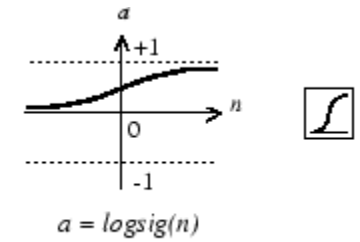
This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

Neuron Model (logsig, tansig, purelin)

An elementary neuron with *R* inputs is shown below. Each input is weighted with an appropriate *w*. The sum of the weighted inputs and the bias forms the input to the transfer function *f*. Neurons can use any differentiable transfer function *f* to generate their output.



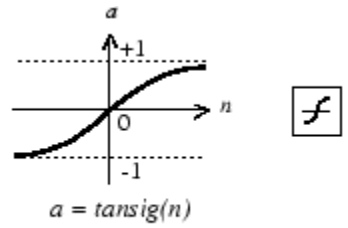
Multilayer networks often use the log-sigmoid transfer function `logsig`.



Log-Sigmoid Transfer Function

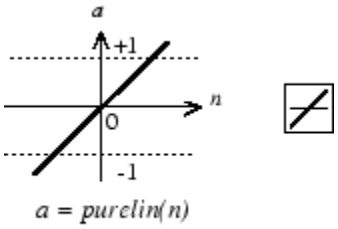
The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

Alternatively, multilayer networks can use the tan-sigmoid transfer function `tansig`.



Tan-Sigmoid Transfer Function

Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function `purelin` is shown below.

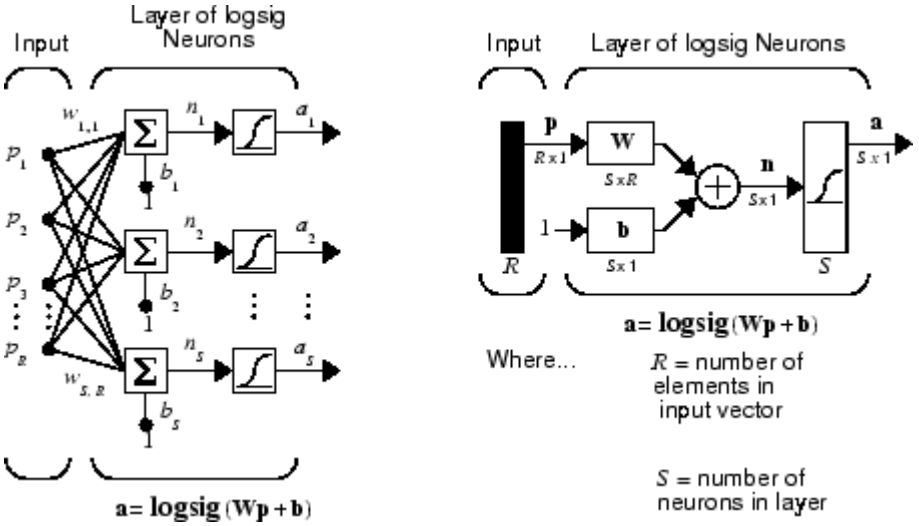


Linear Transfer Function

The three transfer functions described here are the most commonly used transfer functions for multilayer networks, but other differentiable transfer functions can be created and used if desired.

Feedforward Neural Network

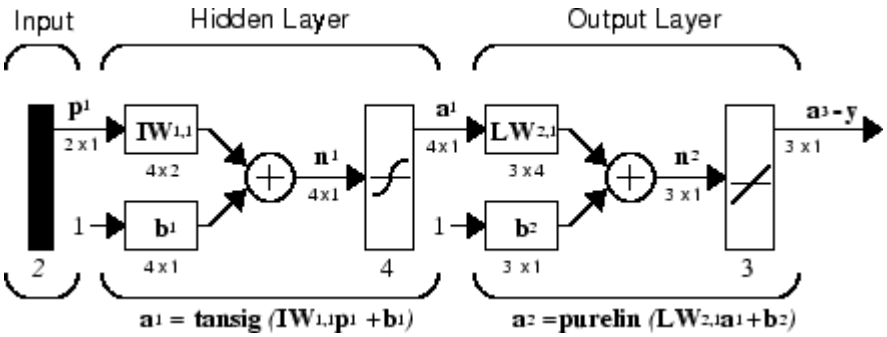
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as `logsig`). This is the case when the network is used for pattern recognition problems (in which a decision is being made by the network).

For multiple-layer networks the layer number determines the superscript on the weight matrix. The appropriate notation is used in the two-layer `tansig`/`purelin` network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer.

Now that the architecture of the multilayer network has been defined, the design process is described in the following sections.

Prepare Data for Multilayer Shallow Neural Networks

Tip To learn how to prepare image data for deep learning networks, see “Preprocess Images for Deep Learning” on page 19-16.

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

Before beginning the network design process, you first collect and prepare sample data. It is generally difficult to incorporate prior knowledge into a neural network, therefore the network can only be as accurate as the data that are used to train the network.

It is important that the data cover the range of inputs for which the network will be used. Multilayer networks can be trained to generalize well within the range of inputs for which they have been trained. However, they do not have the ability to accurately extrapolate beyond this range, so it is important that the training data span the full range of the input space.

After the data have been collected, there are two steps that need to be performed before the data are used to train the network: the data need to be preprocessed, and they need to be divided into subsets.

Choose Neural Network Input-Output Processing Functions

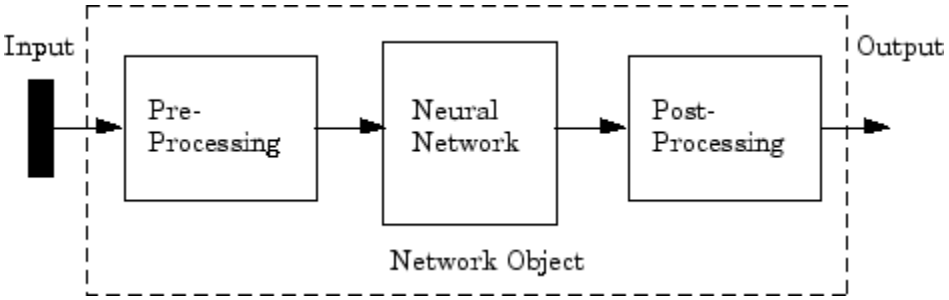
This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

Neural network training can be more efficient if you perform certain preprocessing steps on the network inputs and targets. This section describes several preprocessing routines that you can use. (The most common of these are provided automatically when you create a network, and they become part of the network object, so that whenever the network is used, the data coming into the network is preprocessed in the same way.)

For example, in multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three (exp $(-3) \cong 0.05$). If this happens at the beginning of the training process, the gradients will be very small, and the network training will be very slow. In the first layer of the network, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be very small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network.

Generally, the normalization step is applied to both the input vectors and the target vectors in the data set. In this way, the network output always falls into a normalized range. The network output can then be reverse transformed back into the units of the original target data when the network is put to use in the field.

It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and the output, as shown in the following figure.



Most of the network creation functions in the toolbox, including the multilayer network creation functions, such as `feedforwardnet`, automatically assign processing functions to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for network training.

You can override the default input and output processing functions by adjusting network properties after you create the network.

To see a cell array list of processing functions assigned to the input of a network, access this property:

```
net.inputs{1}.processFcns
```

where the index 1 refers to the first input vector. (There is only one input vector for the feedforward network.) To view the processing functions returned by the output of a two-layer network, access this network property:

```
net.outputs{2}.processFcns
```

where the index 2 refers to the output vector coming from the second layer. (For the feedforward network, there is only one output vector, and it comes from the final layer.) You can use these properties to change the processing functions that you want your network to apply to the inputs and outputs. However, the defaults usually provide excellent performance.

Several processing functions have parameters that customize their operation. You can access or change the parameters of the `i`th input processing function for the network input as follows:

```
net.inputs{1}.processParams{i}
```

You can access or change the parameters of the `i`th output processing function for the network output associated with the second layer, as follows:

```
net.outputs{2}.processParams{i}
```

For multilayer network creation functions, such as `feedforwardnet`, the default input processing functions are `removeconstantrows` and `mapminmax`. For outputs, the default processing functions are also `removeconstantrows` and `mapminmax`.

The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

Function	Algorithm
`mapminmax`	Normalize inputs/targets to fall in the range [−1, 1]
`mapstd`	Normalize inputs/targets to have zero mean and unity variance
`processpca`	Extract principal components from the input vector
`fixunknowns`	Process unknown inputs
`removeconstantrows`	Remove inputs/targets that are constant

Representing Unknown or Don't-Care Targets

Unknown or “don't care” targets can be represented with `NaN` values. We do not want unknown target values to have an impact on training, but if a network has several outputs, some elements of any target vector may be known while others are unknown. One solution would be to remove the partially unknown target vector and its associated input vector from the training set, but that involves the loss of the good target values. A better solution is to represent those unknown targets with `NaN` values. All the performance functions of the toolbox will ignore those targets for purposes of calculating performance and derivatives of performance.

Divide Data for Optimal Neural Network Training

This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

When training multilayer networks, the general practice is to first divide the data into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error. This technique is discussed in more detail in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. The data division is normally performed automatically when you train the network.

Function	Algorithm
`dividerand`	Divide the data randomly (default)
`divideblock`	Divide the data into contiguous blocks
`divideint`	Divide the data using an interleaved selection
`divideind`	Divide the data by index

You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of the division functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

The divide function is accessed automatically whenever the network is trained, and is used to divide the data into training, validation and testing subsets. If `net.divideFcn` is set to `'dividerand'` (the default), then the data is randomly divided into the three subsets using the division parameters `net.divideParam.trainRatio`, `net.divideParam.valRatio`, and `net.divideParam.testRatio`. The fraction of data that is placed in the training set is `trainRatio/(trainRatio+valRatio+testRatio)`, with a similar formula for the other two sets. The default ratios for training, testing and validation are 0.7, 0.15 and 0.15, respectively.

If `net.divideFcn` is set to `'divideblock'`, then the data is divided into three subsets using three contiguous blocks of the original data set (training taking the first block, validation the second and testing the third). The fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

If `net.divideFcn` is set to `'divideint'`, then the data is divided by an interleaved method, as in dealing a deck of cards. It is done so that different percentages of data go into the three subsets. The

fraction of the original data that goes into each subset is determined by the same three division parameters used for `dividerand`.

When `net.divideFcn` is set to `'divideind'`, the data is divided by index. The indices for the three subsets are defined by the division parameters `net.divideParam.trainInd`, `net.divideParam.valInd` and `net.divideParam.testInd`. The default assignment for these indices is the null array, so you must set the indices when using this option.

Create, Configure, and Initialize Multilayer Shallow Neural Networks

In this section...

This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

After the data has been collected, the next step in training a network is to create the network object. The function `feedforwardnet` creates a multilayer feedforward network. If this function is invoked with no input arguments, then a default network object is created that has not been configured. The resulting network can then be configured with the `configure` command.

As an example, the file `bodyfat_dataset.mat` contains a predefined set of input and target vectors. The input vectors define data regarding physical attributes of people and the target values define percentage body fat of the people. Load the data using the following command:

```
load bodyfat_dataset
```

Loading this file creates two variables. The input matrix `bodyfatInputs` consists of 252 column vectors of 13 physical attribute variables for 252 different people. The target matrix `bodyfatTargets` consists of the corresponding 252 body fat percentages.

The next step is to create the network. The following call to `feedforwardnet` creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.)

```
net = feedforwardnet;
net = configure(net, bodyfatInputs, bodyfatTargets);
```

Optional arguments can be provided to `feedforwardnet`. For instance, the first argument is an array containing the number of neurons in each hidden layer. (The default setting is 10, which means one hidden layer with 10 neurons. One hidden layer generally produces excellent results, but you may want to try two hidden layers, if the results with one are not adequate. Increasing the number of neurons in the hidden layer increases the power of the network, but requires more computation and is more likely to produce overfitting.) The second argument contains the name of the training function to be used. If no arguments are supplied, the default number of layers is 2, the default number of neurons in the hidden layer is 10, and the default training function is `trainlm`. The default transfer function for hidden layers is `tansig` and the default for the output layer is `purelin`.

The `configure` command configures the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you might want to reinitialize the weights, or to perform a custom initialization. “Initializing Weights (init)” on page 22-12 explains the details of the initialization process. You can also skip the configuration step and go directly to training the network. The `train` command will automatically configure the network and initialize the weights.

Other Related Architectures

While two-layer feedforward networks can potentially learn virtually any input-output relationship, feedforward networks with more layers might learn complex relationships more quickly. For most

problems, it is best to start with two layers, and then increase to three layers, if the performance with two layers is not satisfactory.

The function `cascadeforwardnet` creates cascade-forward networks. These are similar to feedforward networks, but include a weight connection from the input to each layer, and from each layer to the successive layers. For example, a three-layer network has connections from layer 1 to layer 2, layer 2 to layer 3, and layer 1 to layer 3. The three-layer network also has connections from the input to all three layers. The additional connections might improve the speed at which the network learns the desired relationship.

The function `patternnet` creates a network that is very similar to `feedforwardnet`, except that it uses the `tansig` transfer function in the last layer. This network is generally used for pattern recognition. Other networks can learn dynamic or time-series relationships.

Initializing Weights (init)

Before training a feedforward network, you must initialize the weights and biases. The `configure` command automatically initializes the weights, but you might want to reinitialize them. You do this with the `init` command. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized):

```
net = init(net);
```

Train and Apply Multilayer Shallow Neural Networks

In this section...

Tip To train a deep learning network, use `trainNetwork`.

This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

When the network weights and biases are initialized, the network is ready for training. The multilayer feedforward network can be trained for function approximation (nonlinear regression) or pattern recognition. The training process requires a set of examples of proper network behavior—network inputs `p` and target outputs `t`.

The process of training a neural network involves tuning the values of the weights and biases of the network to optimize network performance, as defined by the network performance function `net.performFcn`. The default performance function for feedforward networks is mean square error `mse`—the average squared error between the network outputs `a` and the target outputs `t`. It is defined as follows:

$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

(Individual squared errors can also be weighted. See “Train Neural Networks with Error Weights” on page 23-32.) There are two different ways in which training can be implemented: incremental mode and batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs in the training set are applied to the network before the weights are updated. This topic describes batch mode training with the `train` command. Incremental training with the `adapt` command is discussed in “Incremental Training with adapt” on page 21-22. For most problems, when using the Deep Learning Toolbox software, batch training is significantly faster and produces smaller errors than incremental training.

For training multilayer feedforward networks, any standard numerical optimization algorithm can be used to optimize the performance function, but there are a few key ones that have shown excellent performance for neural network training. These optimization methods use either the gradient of the network performance with respect to the network weights, or the Jacobian of the network errors with respect to the weights.

The gradient and the Jacobian are calculated using a technique called the *backpropagation* algorithm, which involves performing computations backward through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapters 11 (for the gradient) and 12 (for the Jacobian) of [HDB96 on page 32-2].

Training Algorithms

As an illustration of how the training works, consider the simplest optimization algorithm — gradient descent. It updates the network weights and biases in the direction in which the performance

function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where $\boldsymbol{x}_k$ is a vector of current weights and biases, $\boldsymbol{g}_k$ is the current gradient, and α_k is the learning rate. This equation is iterated until the network converges.

A list of the training algorithms that are available in the Deep Learning Toolbox software and that use gradient- or Jacobian-based methods, is shown in the following table.

For a detailed description of several of these techniques, see also Hagan, M.T., H.B. Demuth, and M.H. Beale, Neural Network Design, Boston, MA: PWS Publishing, 1996, Chapters 11 and 12.

Function	Algorithm
`trainlm`	Levenberg-Marquardt
`trainbr`	Bayesian Regularization
`trainbfg`	BFGS Quasi-Newton
`trainrp`	Resilient Backpropagation
`trainscg`	Scaled Conjugate Gradient
`traincgb`	Conjugate Gradient with Powell/Beale Restarts
`traincgf`	Fletcher-Powell Conjugate Gradient
`traincgp`	Polak-Ribiére Conjugate Gradient
`trainoss`	One Step Secant
`traingdx`	Variable Learning Rate Gradient Descent
`traingdm`	Gradient Descent with Momentum
`traingd`	Gradient Descent

The fastest training function is generally `trainlm`, and it is the default training function for `feedforwardnet`. The quasi-Newton method, `trainbfg`, is also quite fast. Both of these methods tend to be less efficient for large networks (with thousands of weights), since they require more memory and more computation time for these cases. Also, `trainlm` performs better on function fitting (nonlinear regression) problems than on pattern recognition problems.

When training large networks, and when training pattern recognition networks, `trainscg` and `trainrp` are good choices. Their memory requirements are relatively small, and yet they are much faster than standard gradient descent algorithms.

See “Choose a Multilayer Neural Network Training Function” on page 28-14 for a full comparison of the performances of the training algorithms shown in the table above.

As a note on terminology, the term “backpropagation” is sometimes used to refer specifically to the gradient descent algorithm, when applied to neural network training. That terminology is not used here, since the process of computing the gradient and Jacobian by performing calculations backward through the network is applied in all of the training functions listed above. It is clearer to use the name of the specific optimization algorithm that is being used, rather than to use the term backpropagation alone.

Also, the multilayer network is sometimes referred to as a backpropagation network. However, the backpropagation technique that is used to compute gradients and Jacobians in a multilayer network

can also be applied to many different network architectures. In fact, the gradients and Jacobians for any network that has differentiable transfer functions, weight functions and net input functions can be computed using the Deep Learning Toolbox software through a backpropagation process. You can even create your own custom networks and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you.

Training Example

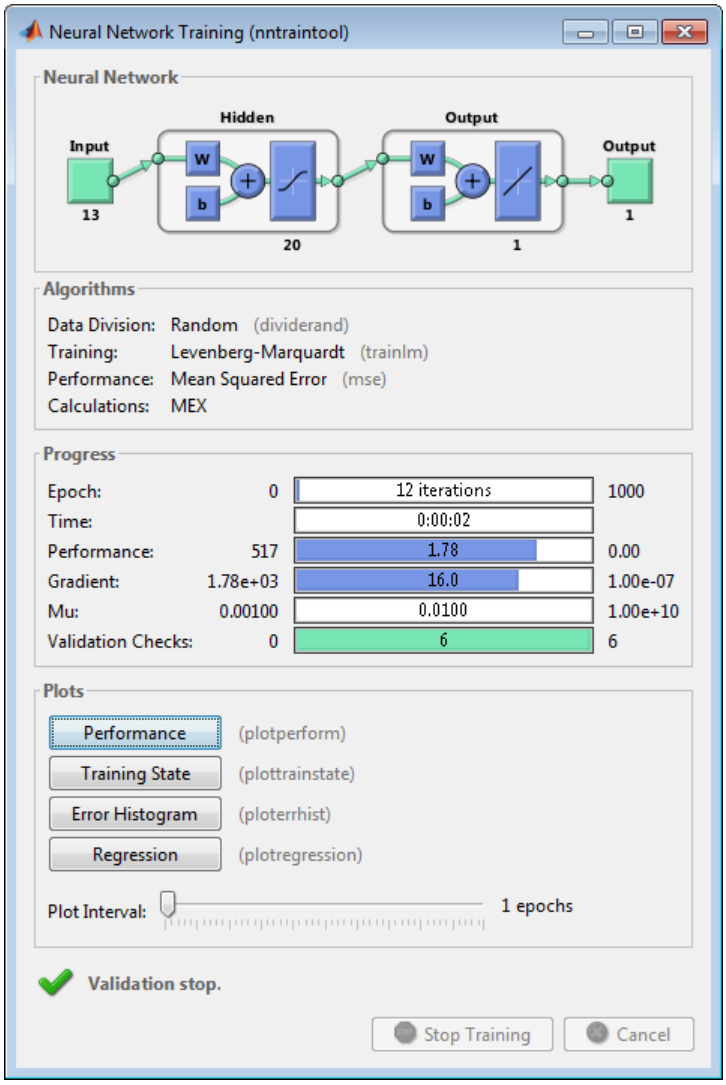
To illustrate the training process, execute the following commands:

```
load bodyfat_dataset
net = feedforwardnet(20);
[net,tr] = train(net,bodyfatInputs,bodyfatTargets);
```

Notice that you did not need to issue the `configure` command, because the configuration is done automatically by the `train` function. The training window will appear during training, as shown in the following figure. (If you do not want to have this window displayed during training, you can set the parameter `net.trainParam.showWindow` to `false`. If you want training information displayed in the command line, you can set the parameter `net.trainParam.showCommandLine` to `true`.)

This window shows that the data has been divided using the `dividerand` function, and the Levenberg-Marquardt (`trainlm`) training method has been used with the mean square error performance function. Recall that these are the default settings for `feedforwardnet`.

During training, the progress is constantly updated in the training window. Of most interest are the performance, the magnitude of the gradient of performance and the number of validation checks. The magnitude of the gradient and the number of validation checks are used to terminate the training. The gradient will become very small as the training reaches a minimum of the performance. If the magnitude of the gradient is less than 1e-5, the training will stop. This limit can be adjusted by setting the parameter `net.trainParam.min_grad`. The number of validation checks represents the number of successive iterations that the validation performance fails to decrease. If this number reaches 6 (the default value), the training will stop. In this run, you can see that the training did stop because of the number of validation checks. You can change this criterion by setting the parameter `net.trainParam.max_fail`. (Note that your results may be different than those shown in the following figure, because of the random setting of the initial weights and biases.)



There are other criteria that can be used to stop network training. They are listed in the following table.

Parameter	Stopping Criteria
`min_grad`	Minimum Gradient Magnitude
`max_fail`	Maximum Number of Validation Increases
`time`	Maximum Training Time
`goal`	Minimum Performance Value
`epochs`	Maximum Number of Training Epochs (Iterations)

The training will also stop if you click the **Stop Training** button in the training window. You might want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by reissuing the `train` command shown above. It will continue to train the network from the completion of the previous run.

From the training window, you can access four plots: performance, training state, error histogram, and regression. The performance plot shows the value of the performance function versus the iteration number. It plots training, validation, and test performances. The training state plot shows the progress of other training variables, such as the gradient magnitude, the number of validation checks, etc. The error histogram plot shows the distribution of the network errors. The regression plot shows a regression between network outputs and network targets. You can use the histogram and regression plots to validate network performance, as is discussed in “Analyze Shallow Neural Network Performance After Training” on page 22-18.

Use the Network

After the network is trained and validated, the network object can be used to calculate the network response to any input. For example, if you want to find the network response to the fifth input vector in the building data set, you can use the following

```
a = net(bodyfatInputs(:,5))
```

```
a =

   27.3740
```

If you try this command, your output might be different, depending on the state of your random number generator when the network was initialized. Below, the network object is called to calculate the outputs for a concurrent set of all the input vectors in the body fat data set. This is the batch mode form of simulation, in which all the input vectors are placed in one matrix. This is much more efficient than presenting the vectors one at a time.

```
a = net(bodyfatInputs);
```

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25.

Analyze Shallow Neural Network Performance After Training

This topic presents part of a typical shallow neural network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2. To learn about how to monitor deep learning training progress, see “Monitor Deep Learning Training Progress” on page 5-115.

When the training in “Train and Apply Multilayer Shallow Neural Networks” on page 22-13 is complete, you can check the network performance and determine if any changes need to be made to the training process, the network architecture, or the data sets. First check the training record, `tr`, which was the second argument returned from the training function.

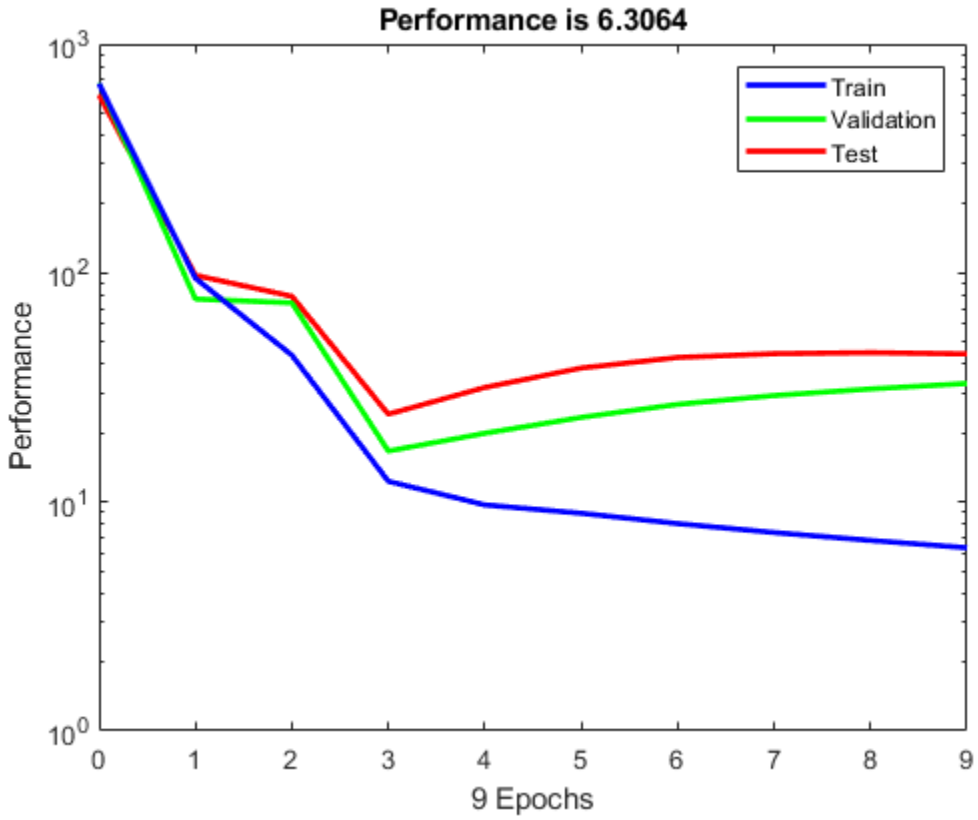
```
tr
```

```
tr = struct with fields:
        trainFcn: 'trainlm'
      trainParam: [1x1 struct]
      performFcn: 'mse'
    performParam: [1x1 struct]
        derivFcn: 'defaultderiv'
       divideFcn: 'dividerand'
      divideMode: 'sample'
     divideParam: [1x1 struct]
        trainInd: [2 3 5 6 9 10 11 13 14 15 18 19 20 22 23 24 25 29 30 ... ]
          valInd: [1 8 17 21 27 28 34 43 63 71 72 74 75 83 106 124 125 ... ]
         testInd: [4 7 12 16 26 32 37 42 53 60 61 67 69 78 82 87 89 104 ... ]
            stop: 'Training finished: Met validation criterion'
      num_epochs: 9
       trainMask: {[NaN 1 1 NaN 1 1 NaN NaN 1 1 1 NaN 1 1 1 NaN NaN 1 1 ... ]}
         valMask: {[1 NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN ... ]}
        testMask: {[NaN NaN NaN 1 NaN NaN 1 NaN NaN NaN NaN 1 NaN NaN ... ]}
      best_epoch: 3
            goal: 0
          states: {1x8 cell}
           epoch: [0 1 2 3 4 5 6 7 8 9]
            time: [0.0600 0.1350 0.1610 0.2010 0.2220 0.2410 0.2580 ... ]
            perf: [672.2031 94.8128 43.7489 12.3078 9.7063 8.9212 8.0412 ... ]
           vperf: [675.3788 76.9621 74.0752 16.6857 19.9424 23.4096 ... ]
           tperf: [599.2224 97.7009 79.1240 24.1796 31.6290 38.4484 ... ]
              mu: [1.0000e-03 0.0100 0.0100 0.1000 0.1000 0.1000 0.1000 ... ]
        gradient: [2.4114e+03 867.8889 301.7333 142.1049 12.4011 85.0504 ... ]
        val_fail: [0 0 0 0 1 2 3 4 5 6]
       best_perf: 12.3078
      best_vperf: 16.6857
      best_tperf: 24.1796
```

This structure contains all of the information concerning the training of the network. For example, `tr.trainInd`, `tr.valInd` and `tr.testInd` contain the indices of the data points that were used in the training, validation and test sets, respectively. If you want to retrain the network using the same division of data, you can set `net.divideFcn` to `'divideInd'`, `net.divideParam.trainInd` to `tr.trainInd`, `net.divideParam.valInd` to `tr.valInd`, `net.divideParam.testInd` to `tr.testInd`.

The `tr` structure also keeps track of several variables during the course of training, such as the value of the performance function, the magnitude of the gradient, etc. You can use the training record to plot the performance progress by using the `plotperf` command:

```
plotperf(tr)
```

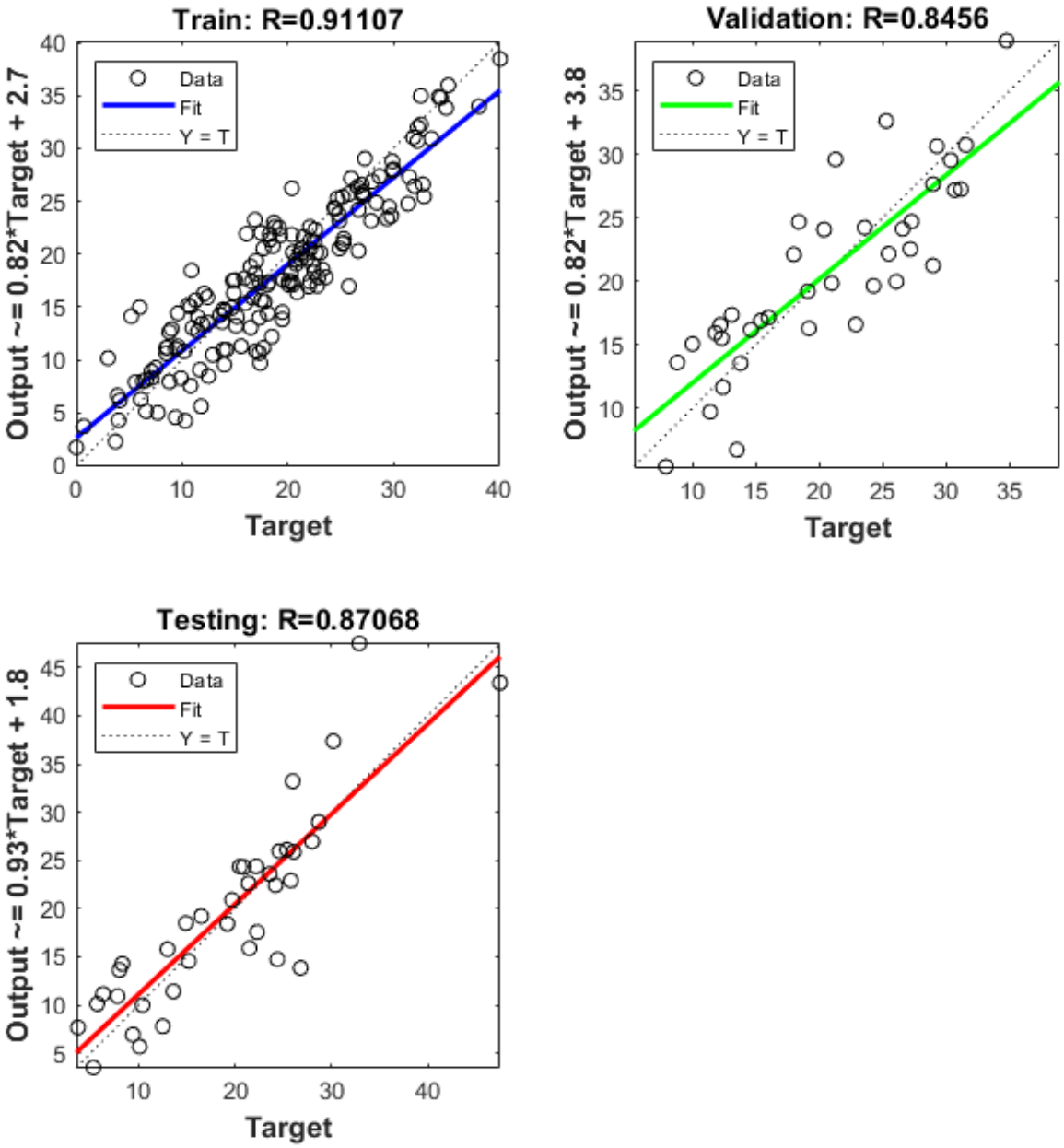


The property `tr.best_epoch` indicates the iteration at which the validation performance reached a minimum. The training continued for 6 more iterations before the training stopped.

This figure does not indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred.

The next step in validating the network is to create a regression plot, which shows the relationship between the outputs of the network and the targets. If the training were perfect, the network outputs and the targets would be exactly equal, but the relationship is rarely perfect in practice. For the body fat example, we can create a regression plot with the following commands. The first command calculates the trained network response to all of the inputs in the data set. The following six commands extract the outputs and targets that belong to the training, validation and test subsets. The final command creates three regression plots for training, testing and validation.

```
bodyfatOutputs = net(bodyfatInputs);
trOut = bodyfatOutputs(tr.trainInd);
vOut = bodyfatOutputs(tr.valInd);
tsOut = bodyfatOutputs(tr.testInd);
trTarg = bodyfatTargets(tr.trainInd);
vTarg = bodyfatTargets(tr.valInd);
tsTarg = bodyfatTargets(tr.testInd);
plotregression(trTarg, trOut, 'Train', vTarg, vOut, 'Validation', tsTarg, tsOut, 'Testing')
```



The three plots represent the training, validation, and testing data. The dashed line in each plot represents the perfect result – outputs = targets. The solid line represents the best fit linear regression line between outputs and targets. The R value is an indication of the relationship between the outputs and targets. If R = 1, this indicates that there is an exact linear relationship between outputs and targets. If R is close to zero, then there is no linear relationship between outputs and targets.

For this example, the training data indicates a good fit. The validation and test results also show large R values. The scatter plot is helpful in showing that certain data points have poor fits. For example, there is a data point in the test set whose network output is close to 35, while the corresponding target value is about 12. The next step would be to investigate this data point to determine if it represents extrapolation (i.e., is it outside of the training data set). If so, then it should be included in the training set, and additional data should be collected to be used in the test set.

Improving Results

If the network is not sufficiently accurate, you can try initializing the network and the training again. Each time your initialize a feedforward network, the network parameters are different and might produce different solutions.

```
net = init(net);
net = train(net, bodyfatInputs, bodyfatTargets);
```

As a second approach, you can increase the number of hidden neurons above 20. Larger numbers of neurons in the hidden layer give the network more flexibility because the network has more parameters it can optimize. (Increase the layer size gradually. If you make the hidden layer too large, you might cause the problem to be under-characterized and the network must optimize more parameters than there are data vectors to constrain these parameters.)

A third option is to try a different training function. Bayesian regularization training with `trainbr`, for example, can sometimes produce better generalization capability than using early stopping.

Finally, try using additional training data. Providing additional data for the network is more likely to produce a network that generalizes well to new data.

Limitations and Cautions

You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use `trainscg` or `trainrp`.

Multilayer networks are capable of performing just about any linear or nonlinear computation, and they can approximate any reasonable function arbitrarily well. However, while the network being trained might theoretically be capable of performing correctly, backpropagation and its variations might not always find a solution. See page 12-8 of [HDB96 on page 32-2] for a discussion of convergence to local minimum points.

The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity, see the figures on pages 12-5 to 12-7 of [HDB96 on page 32-2], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface, depending on the initial starting conditions, it is possible for the network solution to become trapped in one of these local minima. Settling in a local minimum can be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation does not always find the correct weights for the optimum solution. You might want to reinitialize the network and retrain several times to guarantee that you have the best solution.

Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fitted, but the fitting curve oscillates wildly between these points. Ways of dealing with various of these issues are discussed in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25. This topic is also discussed starting on page 11-21 of [HDB96 on page 32-2].

For more information about the workflow with multilayer networks, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.

Dynamic Neural Networks

Introduction to Dynamic Neural Networks

Neural networks can be classified into dynamic and static categories. Static (feedforward) networks have no feedback elements and contain no delays; the output is calculated directly from the input through feedforward connections. In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs, or states of the network.

Details of this workflow are discussed in the following sections:

- “Design Time Series Time-Delay Neural Networks” on page 23-10
- “Prepare Input and Layer Delay States” on page 23-13
- “Design Time Series Distributed Delay Neural Networks” on page 23-14
- “Design Time Series NARX Feedback Neural Networks” on page 23-16
- “Design Layer-Recurrent Neural Networks” on page 23-22

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 22-7
- “Divide Data for Optimal Neural Network Training” on page 22-9
- “Train Neural Networks with Error Weights” on page 23-32

How Dynamic Neural Networks Work

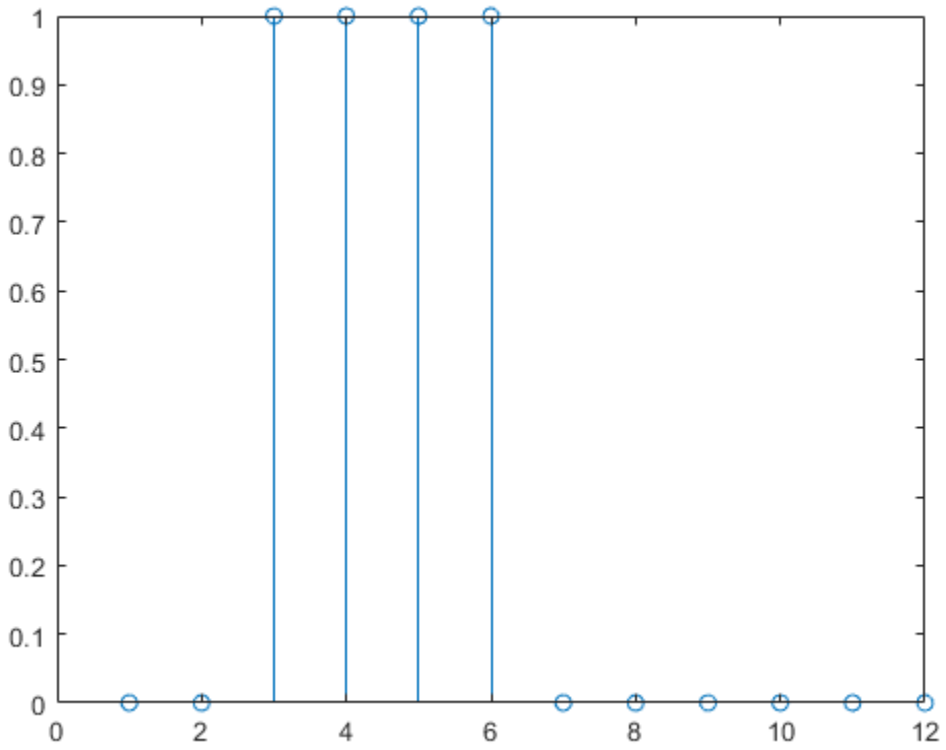
In this section...

Feedforward and Recurrent Neural Networks

Dynamic networks can be divided into two categories: those that have only feedforward connections, and those that have feedback, or recurrent, connections. To understand the differences between static, feedforward-dynamic, and recurrent-dynamic networks, create some networks and see how they respond to an input sequence. (First, you might want to review “Simulation with Sequential Inputs in a Dynamic Network” on page 21-19.)

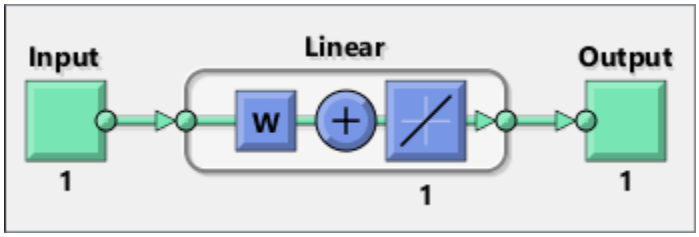
The following commands create a pulse input sequence and plot it:

```
p = {0 0 1 1 1 1 0 0 0 0 0 0};
stem(cell2mat(p))
```



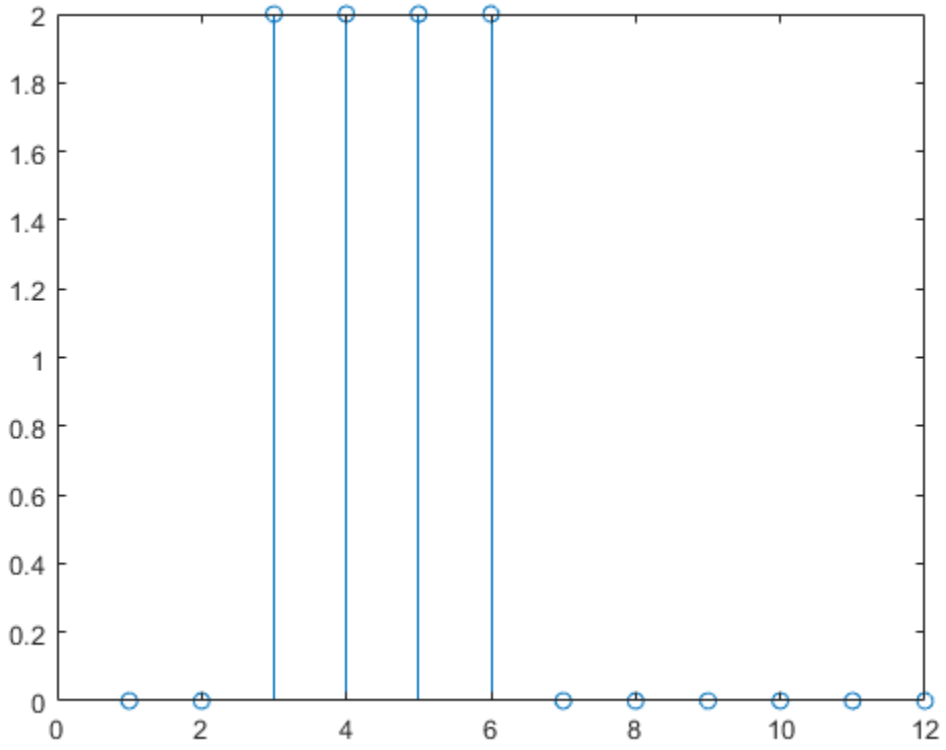
Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

```
net = linearlayer;
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = 2;

view(net)
```



You can now simulate the network response to the pulse input and plot it:

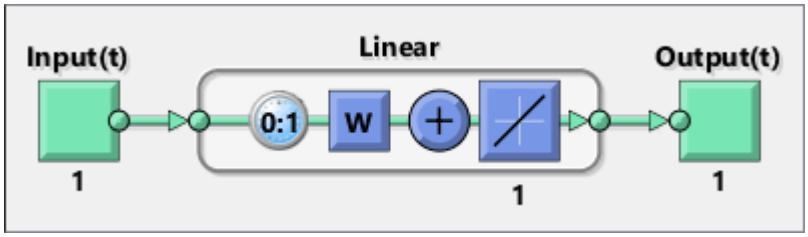
```
a = net(p);
stem(cell2mat(a))
```



Note that the response of the static network lasts just as long as the input pulse. The response of the static network at any time point depends only on the value of the input sequence at that same time point.

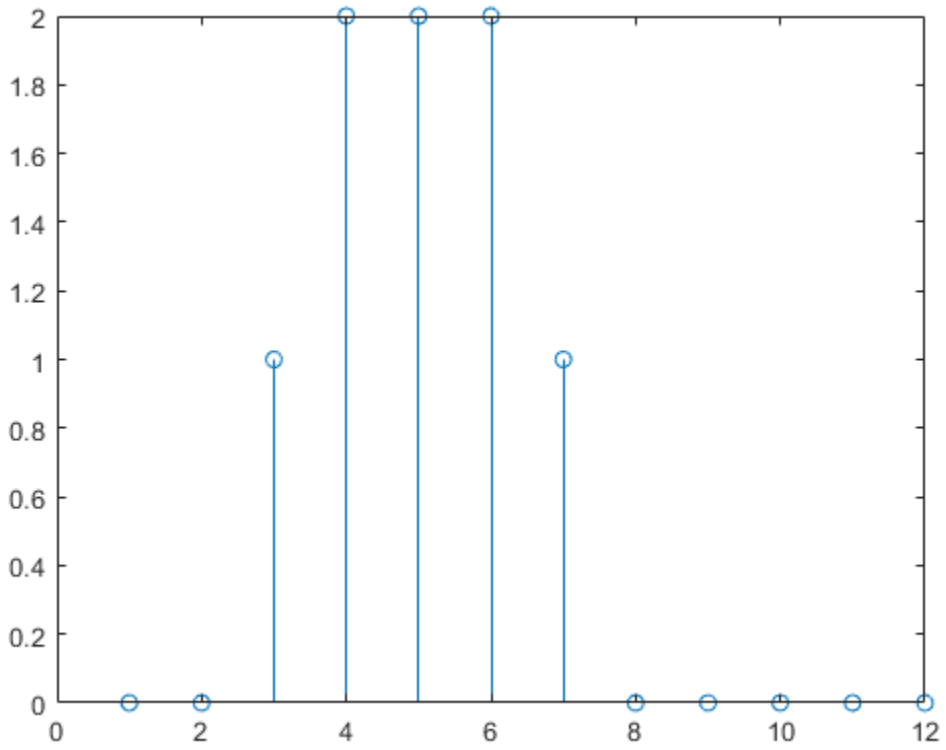
Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used in “Simulation with Concurrent Inputs in a Dynamic Network” on page 21-20, which was a linear network with a tapped delay line on the input:

```
net = linearlayer([0 1]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = [1 1];

view(net)
```



You can again simulate the network response to the pulse input and plot it:

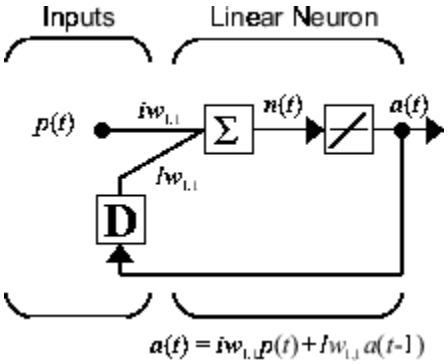
```
a = net(p);
stem(cell2mat(a))
```



The response of the dynamic network lasts longer than the input pulse. The dynamic network has memory. Its response at any given time depends not only on the current input, but also on the history

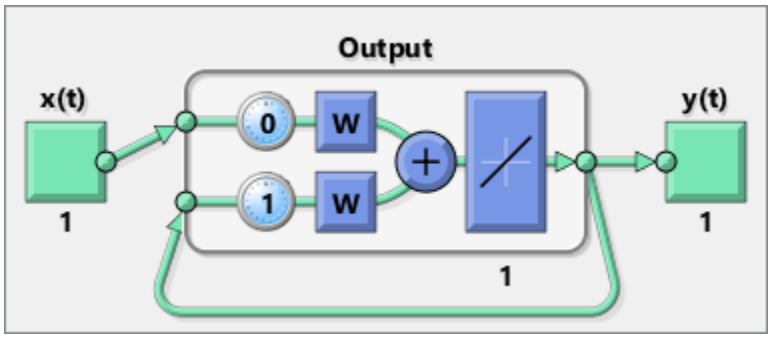
of the input sequence. If the network does not have any feedback connections, then only a finite amount of history will affect the response. In this figure you can see that the response to the pulse lasts one time step beyond the pulse duration. That is because the tapped delay line on the input has a maximum delay of 1.

Now consider a simple recurrent-dynamic network, shown in the following figure.



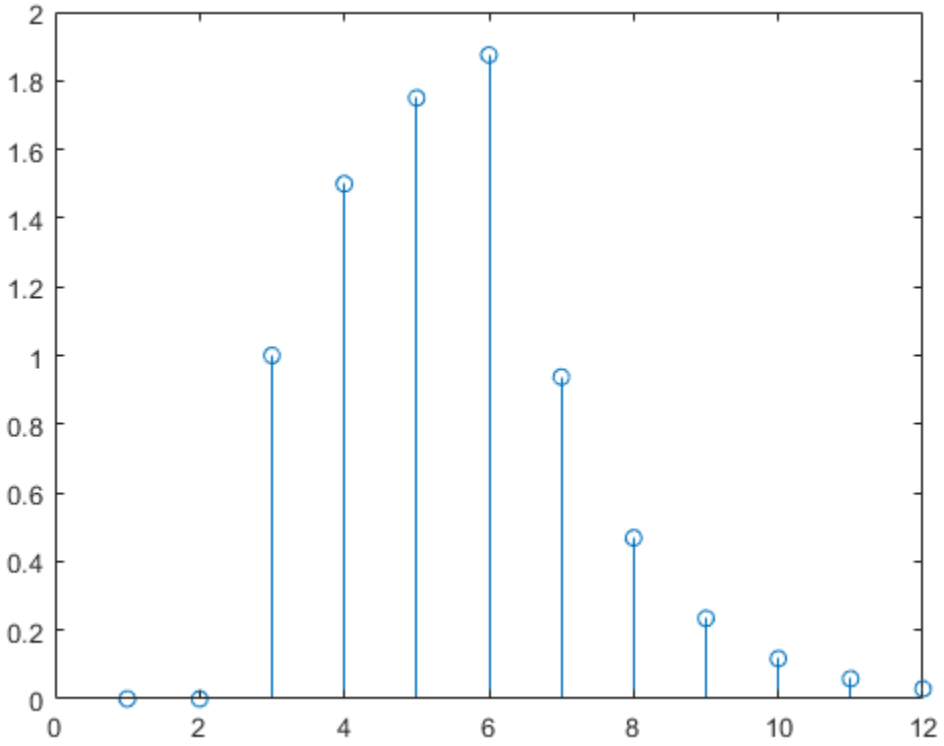
You can create the network, view it and simulate it with the following commands. The `narxnet` command is discussed in “Design Time Series NARX Feedback Neural Networks” on page 23-16.

```
net = narxnet(0,1,[],'closed');
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.LW{1} = .5;
net.IW{1} = 1;
view(net)
```



The following commands plot the network response.

```
a = net(p);
stem(cell2mat(a))
```



Notice that recurrent-dynamic networks typically have a longer response than feedforward-dynamic networks. For linear networks, feedforward-dynamic networks are called finite impulse response (FIR), because the response to an impulse input will become zero after a finite amount of time. Linear recurrent-dynamic networks are called infinite impulse response (IIR), because the response to an impulse can decay to zero (for a stable network), but it will never become exactly equal to zero. An impulse response for a nonlinear network cannot be defined, but the ideas of finite and infinite responses do carry over.

Applications of Dynamic Networks

Dynamic networks are generally more powerful than static networks (although somewhat more difficult to train). Because dynamic networks have memory, they can be trained to learn sequential or time-varying patterns. This has applications in such disparate areas as prediction in financial markets [RoJa96 on page 32-2], channel equalization in communication systems [FeTs03 on page 32-2], phase detection in power systems [KaGr96 on page 32-2], sorting [JaRa04 on page 32-2], fault detection [ChDa99 on page 32-2], speech recognition [Robin94 on page 32-2], and even the prediction of protein structure in genetics [GiPr02 on page 32-2]. You can find a discussion of many more dynamic network applications in [MeJa00 on page 32-2].

One principal application of dynamic neural networks is in control systems. This application is discussed in detail in “Neural Network Control Systems”. Dynamic networks are also well suited for filtering. You will see the use of some linear dynamic networks for filtering in and some of those ideas are extended in this topic, using nonlinear dynamic networks.

Dynamic Network Structures

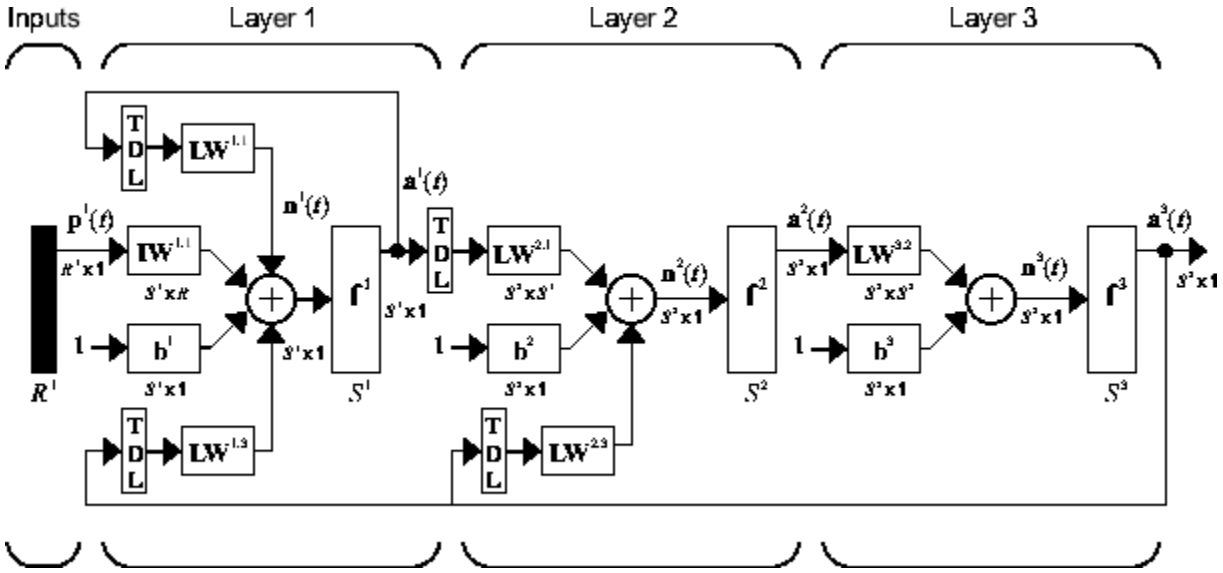
The Deep Learning Toolbox software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

Each layer in the LDDN is made up of the following parts:

- Set of weight matrices that come into that layer (which can connect from other layers or from external inputs), associated weight function rule used to combine the weight matrix with its input (normally standard matrix multiplication, `dotprod`), and associated tapped delay line
- Bias vector
- Net input function rule that is used to combine the outputs of the various weight functions with the bias to produce the net input (normally a summing junction, `netprod`)
- Transfer function

The network has inputs that are connected to special weights, called input weights, and denoted by $\mathbf{IW}^{i,j}$ (`net.IW{i,j}` in the code), where *j* denotes the number of the input vector that enters the weight, and *i* denotes the number of the layer to which the weight is connected. The weights connecting one layer to another are called layer weights and are denoted by $\mathbf{LW}^{i,j}$ (`net.LW{i,j}` in the code), where *j* denotes the number of the layer coming into the weight and *i* denotes the number of the layer at the output of the weight.

The following figure is an example of a three-layer LDDN. The first layer has three weights associated with it: one input weight, a layer weight from layer 1, and a layer weight from layer 3. The two layer weights have tapped delay lines associated with them.

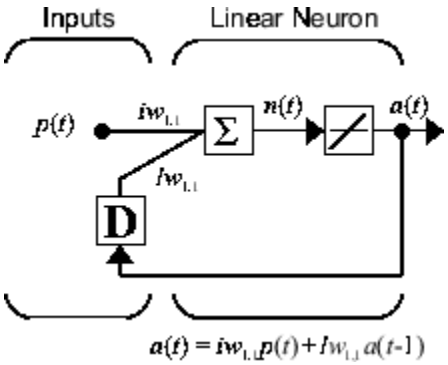


The Deep Learning Toolbox software can be used to train any LDDN, so long as the weight functions, net input functions, and transfer functions have derivatives. Most well-known dynamic network architectures can be represented in LDDN form. In the remainder of this topic you will see how to use some simple commands to create and train several very powerful dynamic networks. Other LDDN networks not covered in this topic can be created using the generic network command, as explained in “Define Shallow Neural Network Architectures”.

Dynamic Network Training

Dynamic networks are trained in the Deep Learning Toolbox software using the same gradient-based algorithms that were described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2. You can select from any of the training functions that were presented in that topic. Examples are provided in the following sections.

Although dynamic networks can be trained using the same gradient-based algorithms that are used for static networks, the performance of the algorithms on dynamic networks can be quite different, and the gradient must be computed in a more complex way. Consider again the simple recurrent network shown in this figure.

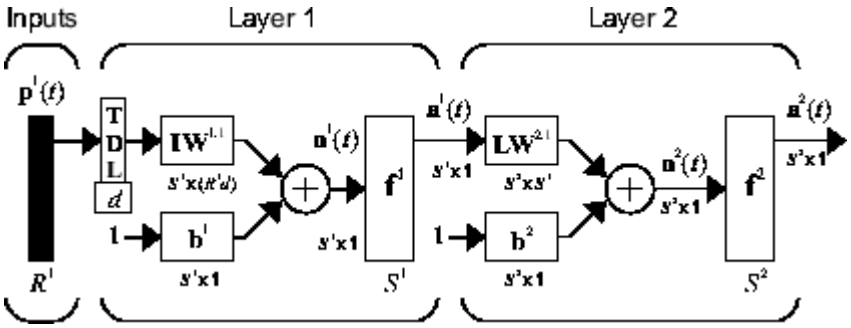


The weights have two different effects on the network output. The first is the direct effect, because a change in the weight causes an immediate change in the output at the current time step. (This first effect can be computed using standard backpropagation.) The second is an indirect effect, because some of the inputs to the layer, such as $a(t - 1)$, are also functions of the weights. To account for this indirect effect, you must use dynamic backpropagation to compute the gradients, which is more computationally intensive. (See [DeHa01a on page 32-2], [DeHa01b on page 32-2] and [DeHa07 on page 32-2].) Expect dynamic backpropagation to take more time to train, in part for this reason. In addition, the error surfaces for dynamic networks can be more complex than those for static networks. Training is more likely to be trapped in local minima. This suggests that you might need to train the network several times to achieve an optimal result. See [DHH01 on page 32-2] and [HDH09 on page 32-2] for some discussion on the training of dynamic networks.

The remaining sections of this topic show how to create, train, and apply certain dynamic networks to modeling, detection, and forecasting problems. Some of the networks require dynamic backpropagation for computing the gradients and others do not. As a user, you do not need to decide whether or not dynamic backpropagation is needed. This is determined automatically by the software, which also decides on the best form of dynamic backpropagation to use. You just need to create the network and then invoke the standard `train` command.

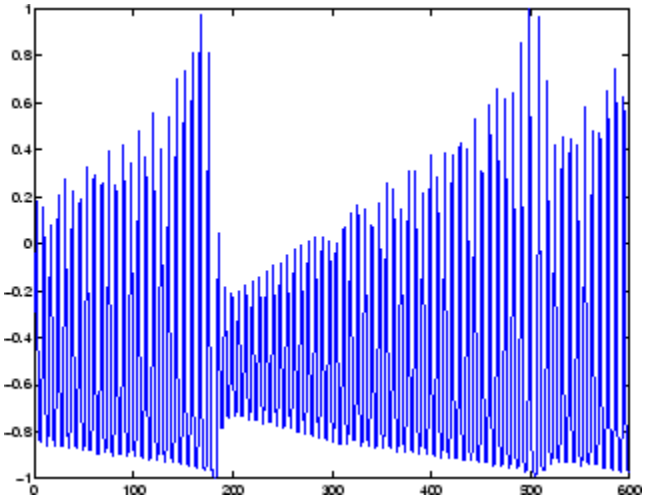
Design Time Series Time-Delay Neural Networks

Begin with the most straightforward dynamic network, which consists of a feedforward network with a tapped delay line at the input. This is called the focused time-delay neural network (FTDNN). This is part of a general class of dynamic networks, called focused networks, in which the dynamics appear only at the input layer of a static multilayer feedforward network. The following figure illustrates a two-layer FTDNN.



This network is well suited to time-series prediction. The following example the use of the FTDNN for predicting a classic time series.

The following figure is a plot of normalized intensity data recorded from a Far-Infrared-Laser in a chaotic state. This is a part of one of several sets of data used for the Santa Fe Time Series Competition [WeGe94 on page 32-2]. In the competition, the objective was to use the first 1000 points of the time series to predict the next 100 points. Because our objective is simply to illustrate how to use the FTDNN for prediction, the network is trained here to perform one-step-ahead predictions. (You can use the resulting network for multistep-ahead predictions by feeding the predictions back to the input of the network and continuing to iterate.)



The first step is to load the data, normalize it, and convert it to a time sequence (represented by a cell array):

```
y = laser_dataset;
y = y(1:600);
```

Now create the FTDNN network, using the `timedelaynet` command. This command is similar to the `feedforwardnet` command, with the additional input of the tapped delay line vector (the first input). For this example, use a tapped delay line with delays from 1 to 8, and use ten neurons in the hidden layer:

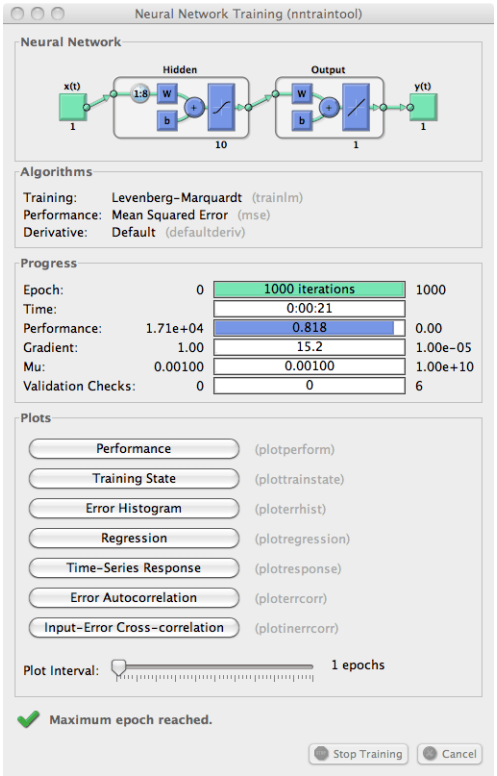
```
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = '';
```

Arrange the network inputs and targets for training. Because the network has a tapped delay line with a maximum delay of 8, begin by predicting the ninth value of the time series. You also need to load the tapped delay line with the eight initial values of the time series (contained in the variable `Pi`):

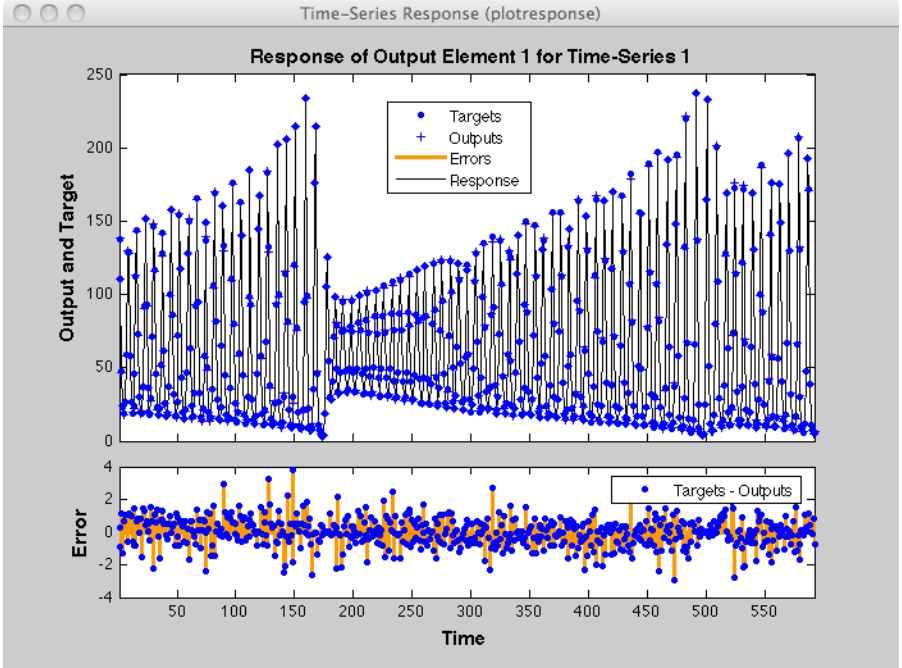
```
p = y(9:end);
t = y(9:end);
Pi=y(1:8);
ftdnn_net = train(ftdnn_net,p,t,Pi);
```

Notice that the input to the network is the same as the target. Because the network has a minimum delay of one time step, this means that you are performing a one-step-ahead prediction.

During training, the following training window appears.



Training stopped because the maximum epoch was reached. From this window, you can display the response of the network by clicking **Time-Series Response**. The following figure appears.



Now simulate the network and determine the prediction error.

```
yp = ftdnn_net(p,Pi);
e = gsubtract(yp,t);
rmse = sqrt(mse(e))

rmse =
    0.9740
```

(Note that `gsubtract` is a general subtraction function that can operate on cell arrays.) This result is much better than you could have obtained using a linear predictor. You can verify this with the following commands, which design a linear filter with the same tapped delay line input as the previous FTDNN.

```
lin_net = linearlayer([1:8]);
lin_net.trainFcn='trainlm';
[lin_net,tr] = train(lin_net,p,t,Pi);
lin_yp = lin_net(p,Pi);
lin_e = gsubtract(lin_yp,t);
lin_rmse = sqrt(mse(lin_e))

lin_rmse =
    21.1386
```

The `rms` error is 21.1386 for the linear predictor, but 0.9740 for the nonlinear FTDNN predictor.

One nice feature of the FTDNN is that it does not require dynamic backpropagation to compute the network gradient. This is because the tapped delay line appears only at the input of the network, and contains no feedback loops or adjustable parameters. For this reason, you will find that this network trains faster than other dynamic networks.

If you have an application for a dynamic network, try the linear network first (`linearlayer`) and then the FTDNN (`timedelaynet`). If neither network is satisfactory, try one of the more complex dynamic networks discussed in the remainder of this topic.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25.

Prepare Input and Layer Delay States

You will notice in the last section that for dynamic networks there is a significant amount of data preparation that is required before training or simulating the network. This is because the tapped delay lines in the network need to be filled with initial conditions, which requires that part of the original data set be removed and shifted. There is a toolbox function that facilitates the data preparation for dynamic (time series) networks - `preparets`. For example, the following lines:

```
p = y(9:end);
t = y(9:end);
Pi = y(1:8);
```

can be replaced with

```
[p,Pi,Ai,t] = preparets(ftdnn_net,y,y);
```

The `preparets` function uses the network object to determine how to fill the tapped delay lines with initial conditions, and how to shift the data to create the correct inputs and targets to use in training or simulating the network. The general form for invoking `preparets` is

```
[X,Xi,Ai,T,EW,shift] = preparets(net,inputs,targets,feedback,EW)
```

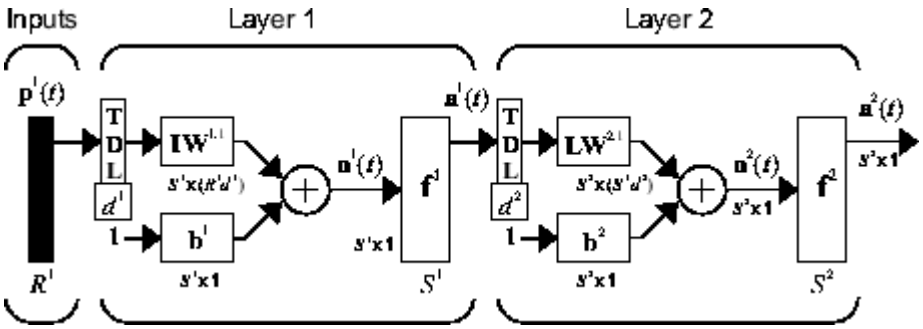
The input arguments for `preparets` are the network object (`net`), the external (non-feedback) input to the network (`inputs`), the non-feedback target (`targets`), the feedback target (`feedback`), and the error weights (`EW`) (see “Train Neural Networks with Error Weights” on page 23-32). The difference between external and feedback signals will become clearer when the NARX network is described in “Design Time Series NARX Feedback Neural Networks” on page 23-16. For the FTDNN network, there is no feedback signal.

The return arguments for `preparets` are the time shift between network inputs and outputs (`shift`), the network input for training and simulation (`X`), the initial inputs (`Xi`) for loading the tapped delay lines for input weights, the initial layer outputs (`Ai`) for loading the tapped delay lines for layer weights, the training targets (`T`), and the error weights (`EW`).

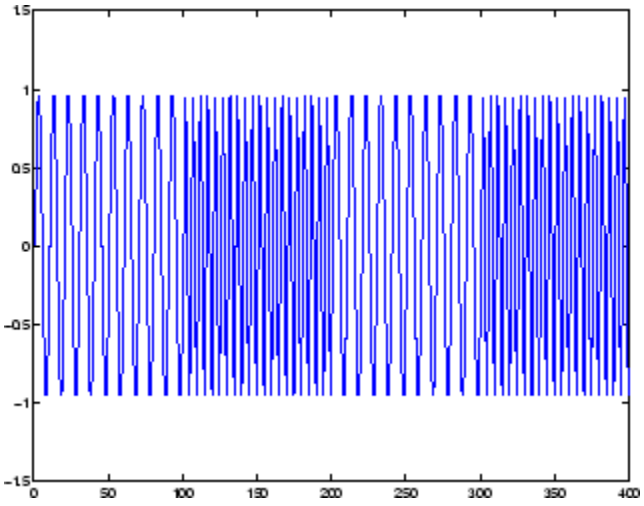
Using preparets eliminates the need to manually shift inputs and targets and load tapped delay lines. This is especially useful for more complex networks.

Design Time Series Distributed Delay Neural Networks

The FTDNN had the tapped delay line memory only at the input to the first layer of the static feedforward network. You can also distribute the tapped delay lines throughout the network. The distributed TDNN was first introduced in [WaHa89 on page 32-2] for phoneme recognition. The original architecture was very specialized for that particular problem. The following figure shows a general two-layer distributed TDNN.



This network can be used for a simplified problem that is similar to phoneme recognition. The network will attempt to recognize the frequency content of an input signal. The following figure shows a signal in which one of two frequencies is present at any given time.



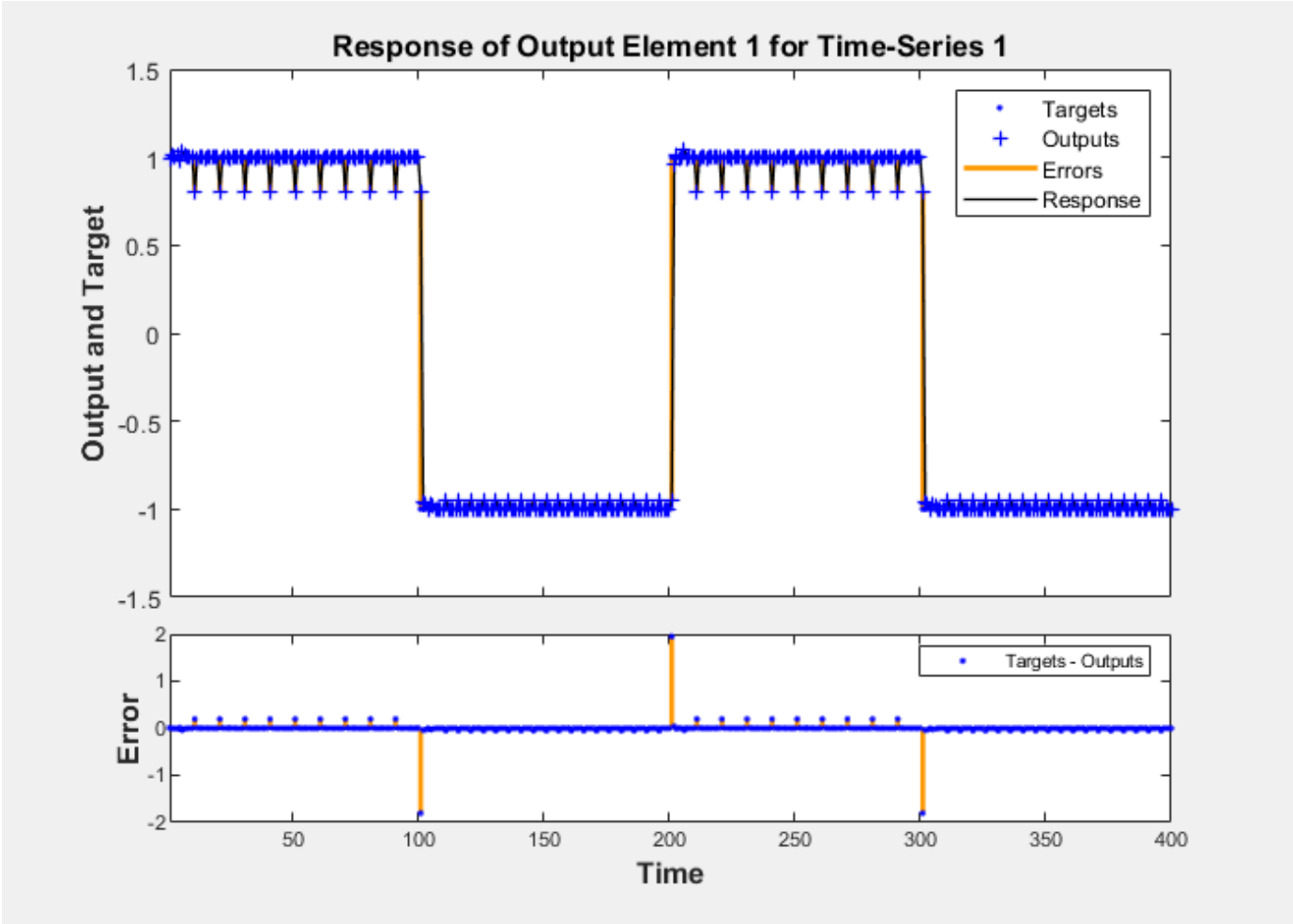
The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
time = 0:99;
y1 = sin(2*pi*time/10);
y2 = sin(2*pi*time/5);
y = [y1 y2 y1 y2];
t1 = ones(1,100);
t2 = -ones(1,100);
t = [t1 t2 t1 t2];
```

Now create the distributed TDNN network with the `distdelaynet` function. The only difference between the `distdelaynet` function and the `timedelaynet` function is that the first input argument is a cell array that contains the tapped delays to be used in each layer. In the next example,

delays of zero to four are used in layer 1 and zero to three are used in layer 2. (To add some variety, the training function `trainbr` is used in this example instead of the default, which is `trainlm`. You can use any training function discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.)

```
d1 = 0:4;
d2 = 0:3;
p = con2seq(y);
t = con2seq(t);
dtdnn_net = distdelaynet({d1,d2},5);
dtdnn_net.trainFcn = 'trainbr';
dtdnn_net.divideFcn = '';
dtdnn_net.trainParam.epochs = 100;
dtdnn_net = train(dtdnn_net,p,t);
yp = sim(dtdnn_net,p);
plotresponse(t,yp)
```



The network is able to accurately distinguish the two “phonemes.”

You will notice that the training is generally slower for the distributed TDNN network than for the FTDNN. This is because the distributed TDNN must use dynamic backpropagation.

Design Time Series NARX Feedback Neural Networks

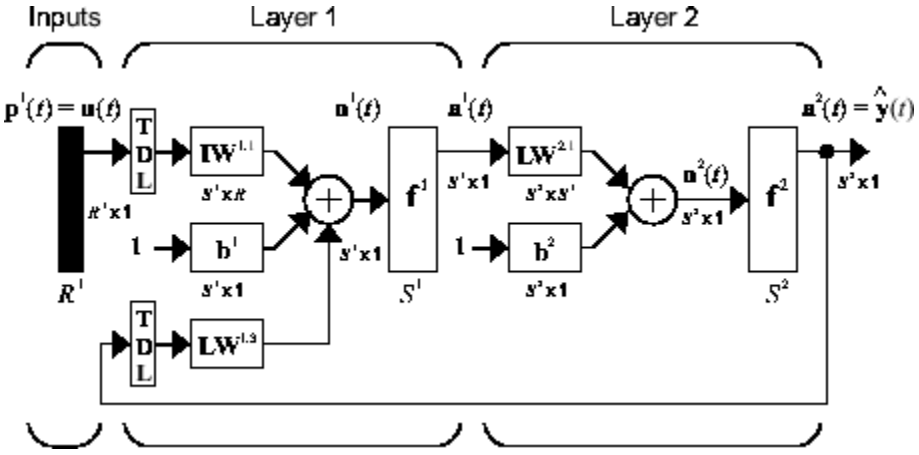
To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 23-39.

All the specific dynamic networks discussed so far have either been focused networks, with the dynamics only at the input layer, or feedforward networks. The nonlinear autoregressive network with exogenous inputs (NARX) is a recurrent dynamic network, with feedback connections enclosing several layers of the network. The NARX model is based on the linear ARX model, which is commonly used in time-series modeling.

The defining equation for the NARX model is

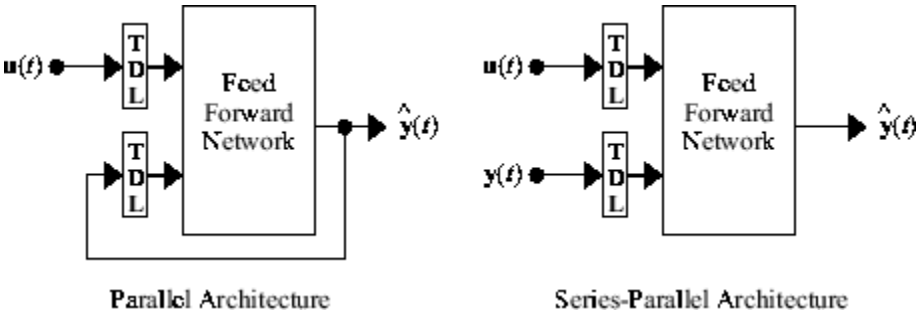
$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))$$

where the next value of the dependent output signal $y(t)$ is regressed on previous values of the output signal and previous values of an independent (exogenous) input signal. You can implement the NARX model by using a feedforward neural network to approximate the function f. A diagram of the resulting network is shown below, where a two-layer feedforward network is used for the approximation. This implementation also allows for a vector ARX model, where the input and output can be multidimensional.



There are many applications for the NARX network. It can be used as a predictor, to predict the next value of the input signal. It can also be used for nonlinear filtering, in which the target output is a noise-free version of the input signal. The use of the NARX network is shown in another important application, the modeling of nonlinear dynamic systems.

Before showing the training of the NARX network, an important configuration that is useful in training needs explanation. You can consider the output of the NARX network to be an estimate of the output of some nonlinear dynamic system that you are trying to model. The output is fed back to the input of the feedforward neural network as part of the standard NARX architecture, as shown in the left figure below. Because the true output is available during the training of the network, you could create a series-parallel architecture (see [NaPa91 on page 32-2]), in which the true output is used instead of feeding back the estimated output, as shown in the right figure below. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and static backpropagation can be used for training.

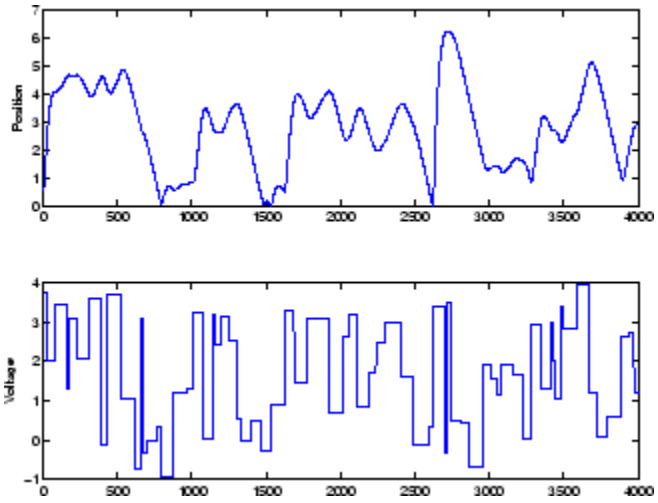


Parallel Architecture Series-Parallel Architecture

The following shows the use of the series-parallel architecture for training a NARX network to model a dynamic system.

The example of the NARX network is the magnetic levitation system described beginning in “Use the NARMA-L2 Controller Block” on page 24-15. The bottom graph in the following figure shows the voltage applied to the electromagnet, and the top graph shows the position of the permanent magnet. The data was collected at a sampling interval of 0.01 seconds to form two time series.

The goal is to develop a NARX model for this magnetic levitation system.



First, load the training data. Use tapped delay lines with two delays for both the input and the output, so training begins with the third data point. There are two inputs to the series-parallel network, the $u(t)$ sequence and the $y(t)$ sequence.

```
load magdata
y = con2seq(y);
u = con2seq(u);
```

Create the series-parallel NARX network using the function `narxnet`. Use 10 neurons in the hidden layer and use `trainlm` for the training function, and then prepare the data with `preparets`:

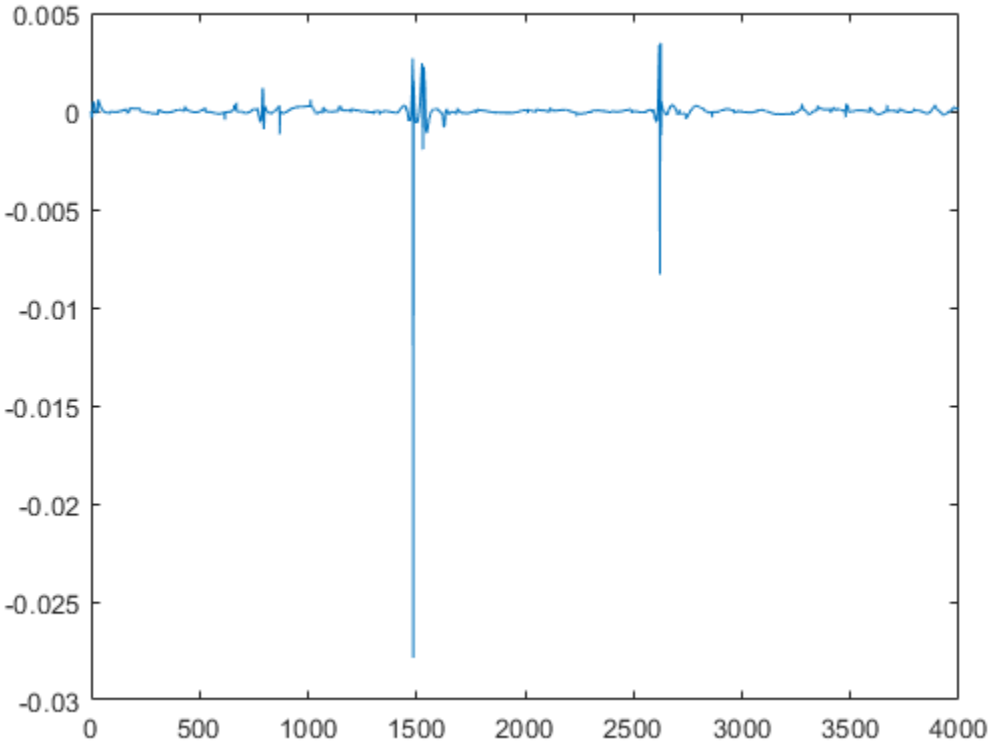
```
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,{},y);
```

(Notice that the y sequence is considered a feedback signal, which is an input that is also an output (target). Later, when you close the loop, the appropriate output will be connected to the appropriate input.) Now you are ready to train the network.

```
narx_net = train(narx_net,p,t,Pi);
```

You can now simulate the network and plot the resulting errors for the series-parallel implementation.

```
yp = sim(narx_net,p,Pi);
e = cell2mat(yp)-cell2mat(t);
plot(e)
```



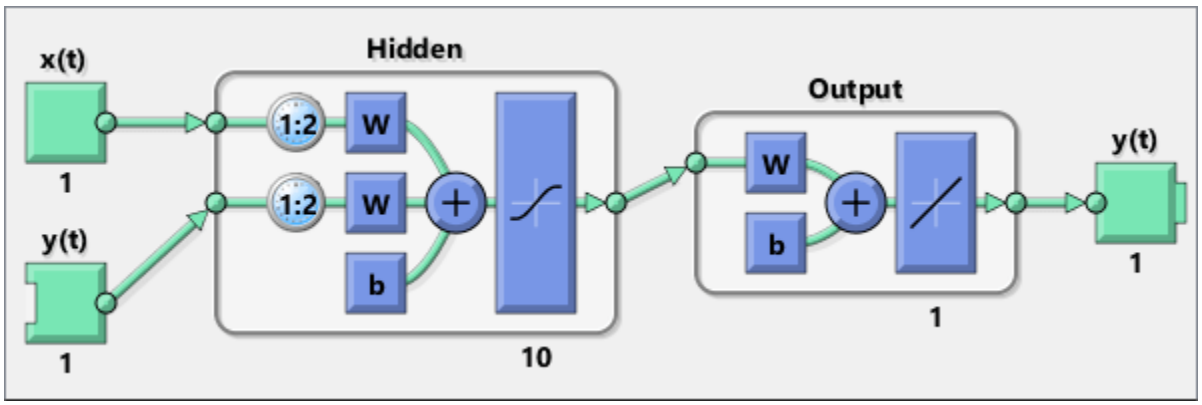
You can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form (closed loop) and then to perform an iterated prediction over many time steps. Now the parallel operation is shown.

There is a toolbox function (`closeloop`) for converting NARX (and other) networks from the series-parallel configuration (open loop), which is useful for training, to the parallel configuration (closed loop), which is useful for multi-step-ahead prediction. The following command illustrates how to convert the network that you just trained to parallel form:

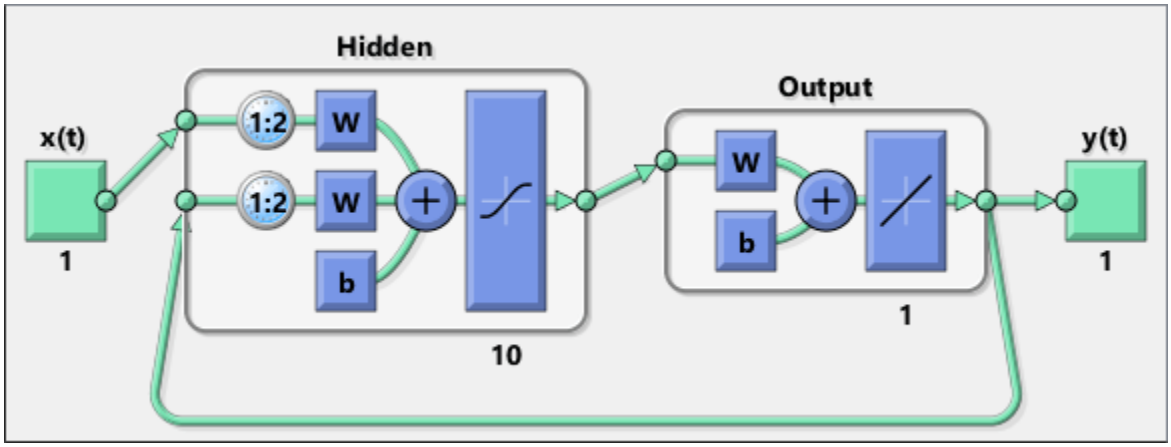
```
narx_net_closed = closeloop(narx_net);
```

To see the differences between the two networks, you can use the view command:

```
view(narx_net)
```



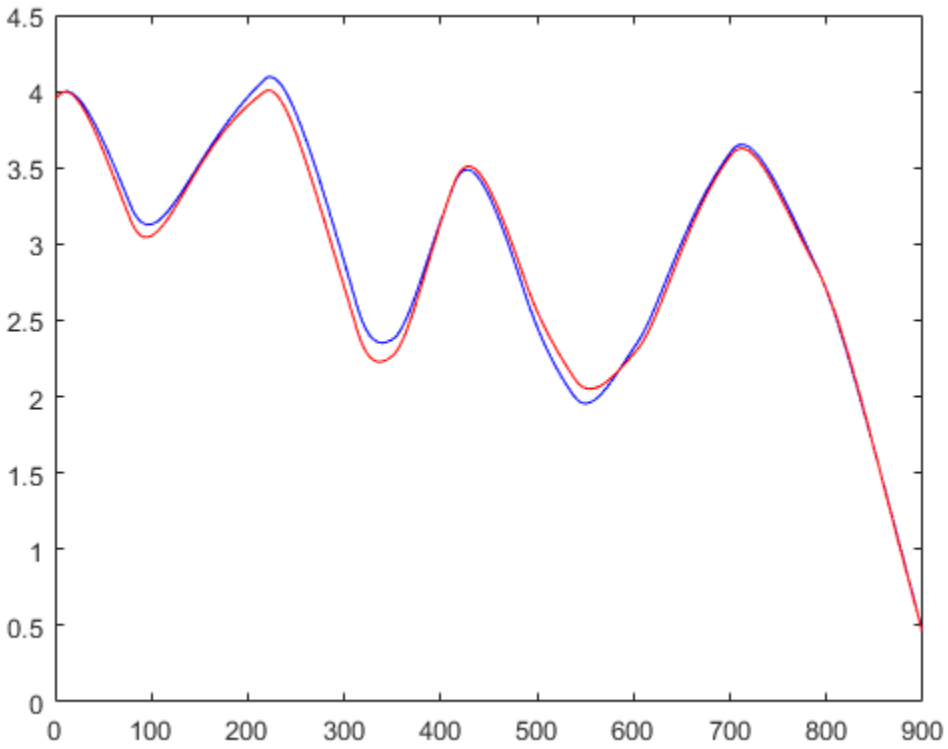

```
view(narx_net_closed)
```



All of the training is done in open loop (also called series-parallel architecture), including the validation and testing steps. The typical workflow is to fully create the network in open loop, and only when it has been trained (which includes validation and testing steps) is it transformed to closed loop for multistep-ahead prediction. Likewise, the R values in the GUI are computed based on the open-loop training results.

You can now use the closed-loop (parallel) configuration to perform an iterated prediction of 900 time steps. In this network you need to load the two initial inputs and the two initial outputs as initial conditions. You can use the `preparets` function to prepare the data. It will use the network structure to determine how to divide and shift the data appropriately.

```
y1 = y(1700:2600);
u1 = u(1700:2600);
[p1,Pi1,Ai1,t1] = preparets(narx_net_closed,u1,{},y1);
yp1 = narx_net_closed(p1,Pi1,Ai1);
TS = size(t1,2);
plot(1:TS,cell2mat(t1),'b',1:TS,cell2mat(yp1),'r')
```



The figure illustrates the iterated prediction. The blue line is the actual position of the magnet, and the red line is the position predicted by the NARX neural network. Even though the network is predicting 900 time steps ahead, the prediction is very accurate.

In order for the parallel response (iterated prediction) to be accurate, it is important that the network be trained so that the errors in the series-parallel configuration (one-step-ahead prediction) are very small.

You can also create a parallel (closed loop) NARX network, using the `narxnet` command with the fourth input argument set to `'closed'`, and train that network directly. Generally, the training takes longer, and the resulting performance is not as good as that obtained with series-parallel training.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25.

Multiple External Variables

The maglev example showed how to model a time series with a single external input value over time. But the NARX network will work for problems with multiple external input elements and predict

series with multiple elements. In these cases, the input and target consist of row cell arrays representing time, but with each cell element being an N-by-1 vector for the N elements of the input or target signal.

For example, here is a dataset which consists of 2-element external variables predicting a 1-element series.

```
[X,T] = ph_dataset;
```

The external inputs X are formatted as a row cell array of 2-element vectors, with each vector representing acid and base solution flow. The targets represent the resulting pH of the solution over time.

You can reformat your own multi-element series data from matrix form to neural network time-series form with the function `con2seq`.

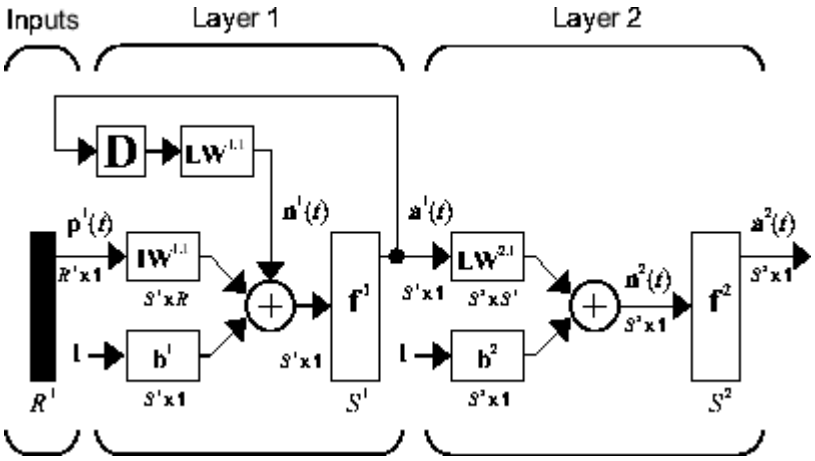
The process for training a network proceeds as it did above for the maglev problem.

```
net = narxnet(10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
e = gsubtract(t,y);
```

To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 23-39.

Design Layer-Recurrent Neural Networks

The next dynamic network to be introduced is the Layer-Recurrent Network (LRN). An earlier simplified version of this network was introduced by Elman [Elma90 on page 32-2]. In the LRN, there is a feedback loop, with a single delay, around each layer of the network except for the last layer. The original Elman network had only two layers, and used a `tansig` transfer function for the hidden layer and a `purelin` transfer function for the output layer. The original Elman network was trained using an approximation to the backpropagation algorithm. The `layrecnet` command generalizes the Elman network to have an arbitrary number of layers and to have arbitrary transfer functions in each layer. The toolbox trains the LRN using exact versions of the gradient-based algorithms discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2. The following figure illustrates a two-layer LRN.

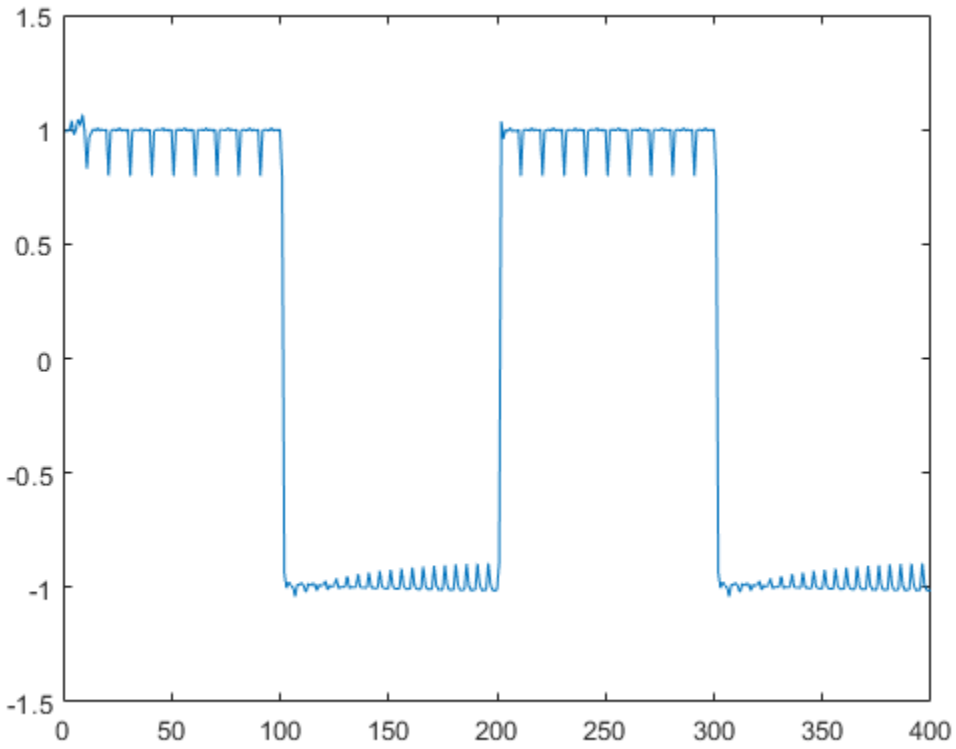


The LRN configurations are used in many filtering and modeling applications discussed already. To show its operation, this example uses the “phoneme” detection problem discussed in “Design Time Series Distributed Delay Neural Networks” on page 23-14. Here is the code to load the data and to create and train the network:

```
load phoneme
p = con2seq(y);
t = con2seq(t);
lrn_net = layrecnet(1,8);
lrn_net.trainFcn = 'trainbr';
lrn_net.trainParam.show = 5;
lrn_net.trainParam.epochs = 50;
lrn_net = train(lrn_net,p,t);
```

After training, you can plot the response using the following code:

```
y = lrn_net(p);
plot(cell2mat(y))
```



The plot shows that the network was able to detect the “phonemes.” The response is very similar to the one obtained using the TDNN.

Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times.

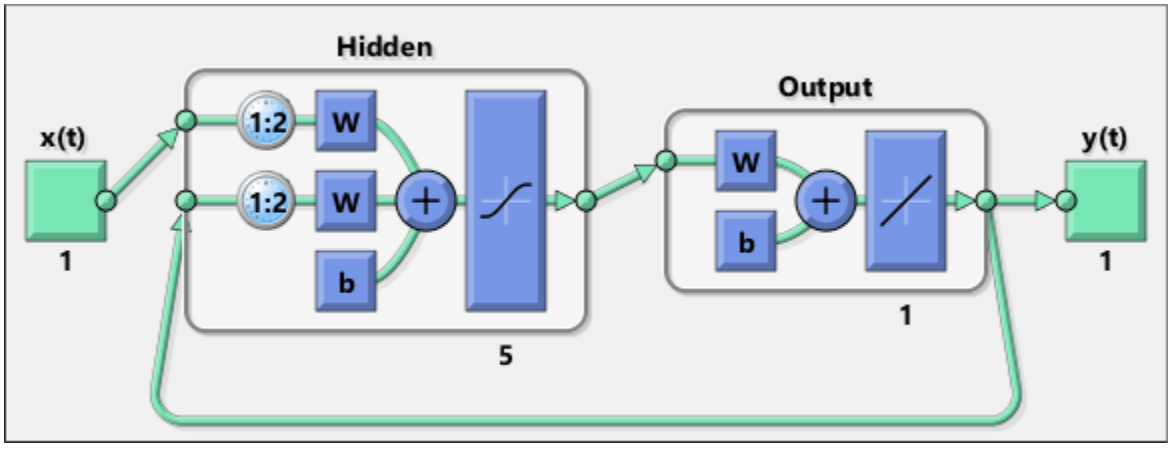
There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25.

Create Reference Model Controller with MATLAB Script

So far, this topic has described the training procedures for several specific dynamic network architectures. However, *any* network that can be created in the toolbox can be trained using the training functions described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2 so long as the components of the network are differentiable. This section gives an example of how to create and train a custom architecture. The custom architecture you will use is the model reference adaptive control (MRAC) system that is described in detail in “Design Model-Reference Neural Controller in Simulink” on page 24-19.

As you can see in “Design Model-Reference Neural Controller in Simulink” on page 24-19, the model reference control architecture has two subnetworks. One subnetwork is the model of the plant that you want to control. The other subnetwork is the controller. You will begin by training a NARX network that will become the plant model subnetwork. For this example, you will use the robot arm to represent the plant, as described in “Design Model-Reference Neural Controller in Simulink” on page 24-19. The following code will load data collected from the robot arm and create and train a NARX network. For this simple problem, you do not need to preprocess the data, and all of the data can be used for training, so no data division is needed.

```
[u,y] = robotarm_dataset;
d1 = [1:2];
d2 = [1:2];
S1 = 5;
narx_net = narxnet(d1,d2,S1);
narx_net.divideFcn = '';
narx_net.inputs{1}.processFcns = {};
narx_net.inputs{2}.processFcns = {};
narx_net.outputs{2}.processFcns = {};
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,{},y);
narx_net = train(narx_net,p,t,Pi);
narx_net_closed = closeloop(narx_net);
view(narx_net_closed)
```



The resulting network is shown in the figure.

Now that the NARX plant model is trained, you can create the total MRAC system and insert the NARX model inside. Begin with a feedforward network, and then add the feedback connections. Also, turn off learning in the plant model subnetwork, since it has already been trained. The next stage of training will train only the controller subnetwork.

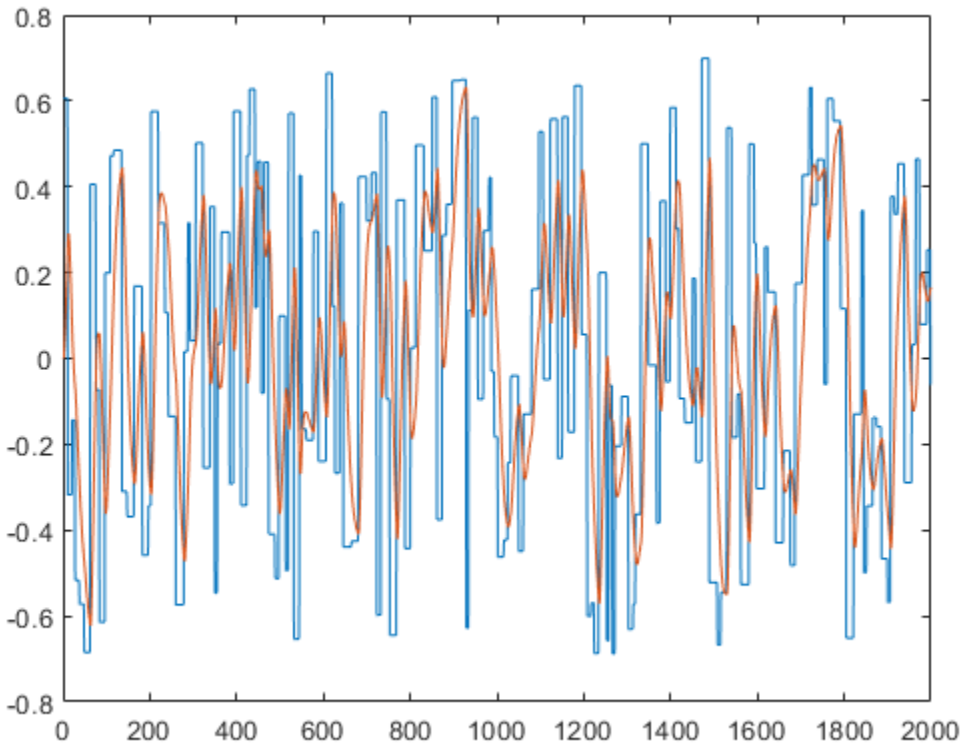
```
mrac_net = feedforwardnet([S1 1 S1]);
mrac_net.layerConnect = [0 1 0 1;1 0 0 0;0 1 0 1;0 0 1 0];
mrac_net.outputs{4}.feedbackMode = 'closed';
mrac_net.layers{2}.transferFcn = 'purelin';
mrac_net.layerWeights{3,4}.delays = 1:2;
mrac_net.layerWeights{3,2}.delays = 1:2;
mrac_net.layerWeights{3,2}.learn = 0;
mrac_net.layerWeights{3,4}.learn = 0;
mrac_net.layerWeights{4,3}.learn = 0;
mrac_net.biases{3}.learn = 0;
mrac_net.biases{4}.learn = 0;
```

The following code turns off data division and preprocessing, which are not needed for this example problem. It also sets the delays needed for certain layers and names the network.

```
mrac_net.divideFcn = '';
mrac_net.inputs{1}.processFcns = {};
mrac_net.outputs{4}.processFcns = {};
mrac_net.name = 'Model Reference Adaptive Control Network';
mrac_net.layerWeights{1,2}.delays = 1:2;
mrac_net.layerWeights{1,4}.delays = 1:2;
mrac_net.inputWeights{1}.delays = 1:2;
```

To configure the network, you need some sample training data. The following code loads and plots the training data, and configures the network:

```
[refin,refout] = refmodel_dataset;
ind = 1:length(refin);
plot(ind,cell2mat(refin),ind,cell2mat(refout))
mrac_net = configure(mrac_net,refin,refout);
```



You want the closed-loop MRAC system to respond in the same way as the reference model that was used to generate this data. (See “Use the Model Reference Controller Block” on page 24-20 for a description of the reference model.)

Now insert the weights from the trained plant model network into the appropriate location of the MRAC system.

```
mrac_net.LW{3,2} = narx_net_closed.IW{1};
mrac_net.LW{3,4} = narx_net_closed.LW{1,2};
mrac_net.b{3} = narx_net_closed.b{1};
mrac_net.LW{4,3} = narx_net_closed.LW{2,1};
mrac_net.b{4} = narx_net_closed.b{2};
```

You can set the output weights of the controller network to zero, which will give the plant an initial input of zero.

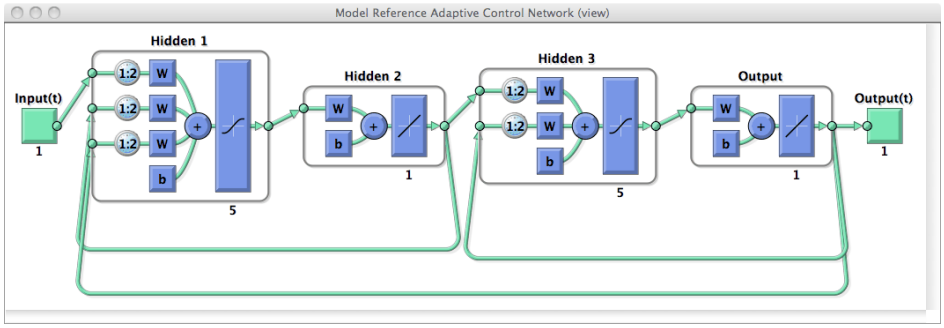
```
mrac_net.LW{2,1} = zeros(size(mrac_net.LW{2,1}));
mrac_net.b{2} = 0;
```

You can also associate any plots and training function that you desire to the network.

```
mrac_net.plotFcns = {'plotperform','plottrainstate',...
    'ploterrhist','plotregression','plotresponse'};
mrac_net.trainFcn = 'trainlm';
```

The final MRAC network can be viewed with the following command:

```
view(mrac_net)
```



Layer 3 and layer 4 (output) make up the plant model subnetwork. Layer 1 and layer 2 make up the controller.

You can now prepare the training data and train the network.

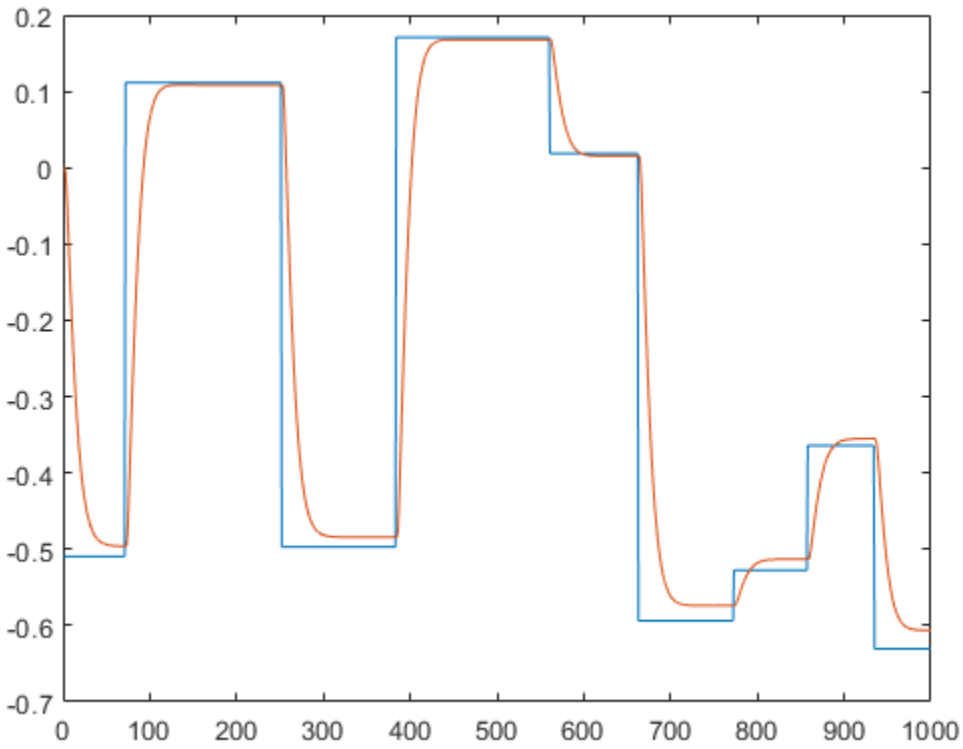
```
[x_tot,xi_tot,ai_tot,t_tot] = ...
            preparets(mrac_net,refin,{},refout);
mrac_net.trainParam.epochs = 50;
mrac_net.trainParam.min_grad = 1e-10;
[mrac_net,tr] = train(mrac_net,x_tot,t_tot,xi_tot,ai_tot);
```

Note Notice that you are using the `trainlm` training function here, but any of the training functions discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2 could be used as well. Any network that you can create in the toolbox can be trained with any of those training functions. The only limitation is that all of the parts of the network must be differentiable.

You will find that the training of the MRAC system takes much longer that the training of the NARX plant model. This is because the network is recurrent and dynamic backpropagation must be used. This is determined automatically by the toolbox software and does not require any user intervention. There are several implementations of dynamic backpropagation (see [DeHa07 on page 32-2]), and the toolbox software automatically determines the most efficient one for the selected network architecture and training algorithm.

After the network has been trained, you can test the operation by applying a test input to the MRAC network. The following code creates a `skyline` input function, which is a series of steps of random height and width, and applies it to the trained MRAC network.

```
testin = skyline(1000,50,200,-.7,.7);
testinseq = con2seq(testin);
testoutseq = mrac_net(testinseq);
testout = cell2mat(testoutseq);
figure
plot([testin' testout'])
```



From the figure, you can see that the plant model output does follow the reference input with the correct critically damped response, even though the input sequence was not the same as the input sequence in the training data. The steady state response is not perfect for each step, but this could be improved with a larger training set and perhaps more hidden neurons.

The purpose of this example was to show that you can create your own custom dynamic network and train it using the standard toolbox training functions without any modifications. Any network that you can create in the toolbox can be trained with the standard training functions, as long as each component of the network has a defined derivative.

It should be noted that recurrent networks are generally more difficult to train than feedforward networks. See [HDH09 on page 32-2] for some discussion of these training difficulties.

Multiple Sequences with Dynamic Neural Networks

There are times when time-series data is not available in one long sequence, but rather as several shorter sequences. When dealing with static networks and concurrent batches of static data, you can simply append data sets together to form one large concurrent batch. However, you would not generally want to append time sequences together, since that would cause a discontinuity in the sequence. For these cases, you can create a concurrent set of sequences, as described in “Understanding Shallow Network Data Structures” on page 21-18.

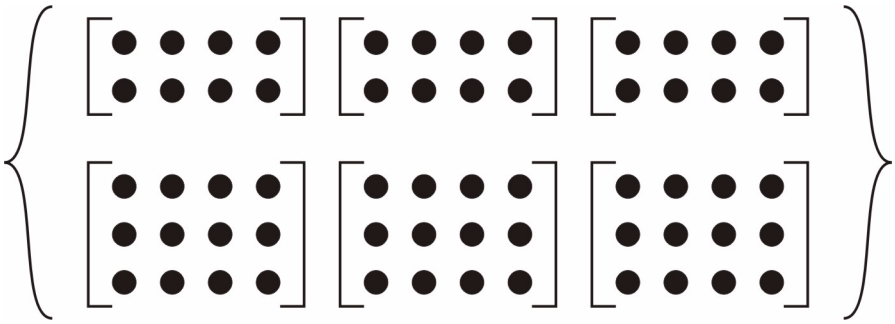
When training a network with a concurrent set of sequences, it is required that each sequence be of the same length. If this is not the case, then the shorter sequence inputs and targets should be padded with NaNs, in order to make all sequences the same length. The targets that are assigned values of NaN will be ignored during the calculation of network performance.

The following code illustrates the use of the function `catsamples` to combine several sequences together to form a concurrent set of sequences, while at the same time padding the shorter sequences.

```
load magmulseq
y_mul = catsamples(y1,y2,y3,'pad');
u_mul = catsamples(u1,u2,u3,'pad');
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u_mul,{},y_mul);
narx_net = train(narx_net,p,t,Pi);
```

Neural Network Time-Series Utilities

There are other utility functions that are useful when manipulating neural network data, which can consist of time sequences, concurrent batches or combinations of both. It can also include multiple signals (as in multiple input, output or target vectors). The following diagram illustrates the structure of a general neural network data object. For this example there are three time steps of a batch of four samples (four sequences) of two signals. One signal has two elements, and the other signal has three elements.



The following table lists some of the more useful toolbox utility functions for neural network data. They allow you to do things like add, subtract, multiply, divide, etc. (Addition and subtraction of cell arrays do not have standard definitions, but for neural network data these operations are well defined and are implemented in the following functions.)

Function	Operation
`gadd`	Add neural network (nn) data.
`gdivide`	Divide nn data.
`getelements`	Select indicated elements from nn data.
`getsamples`	Select indicated samples from nn data.
`getsignals`	Select indicated signals from nn data.
`gettimesteps`	Select indicated time steps from nn data.
`gmultiply`	Multiply nn data.
`gnegate`	Take the negative of nn data.
`gsubtract`	Subtract nn data.
`nndata`	Create an nn data object of specified size, where values are assigned randomly or to a constant.
`nnsize`	Return number of elements, samples, time steps and signals in an nn data object.
`numelements`	Return the number of elements in nn data.
`numsamples`	Return the number of samples in nn data.
`numsignals`	Return the number of signals in nn data.
`numtimesteps`	Return the number of time steps in nn data.
`setelements`	Set specified elements of nn data.
`setsamples`	Set specified samples of nn data.

Function	Operation
`setsignals`	Set specified signals of nn data.
`settimesteps`	Set specified time steps of nn data.

There are also some useful plotting and analysis functions for dynamic networks that are listed in the following table. There are examples of using these functions in the “Get Started with Deep Learning Toolbox”.

Function	Operation
`ploterrcorr`	Plot the autocorrelation function of the error.
`plotinerrcorr`	Plot the crosscorrelation between the error and the input.
`plotresponse`	Plot network output and target versus time.

Train Neural Networks with Error Weights

In the default mean square error performance function (see “Train and Apply Multilayer Shallow Neural Networks” on page 22-13), each squared error contributes the same amount to the performance function as follows:

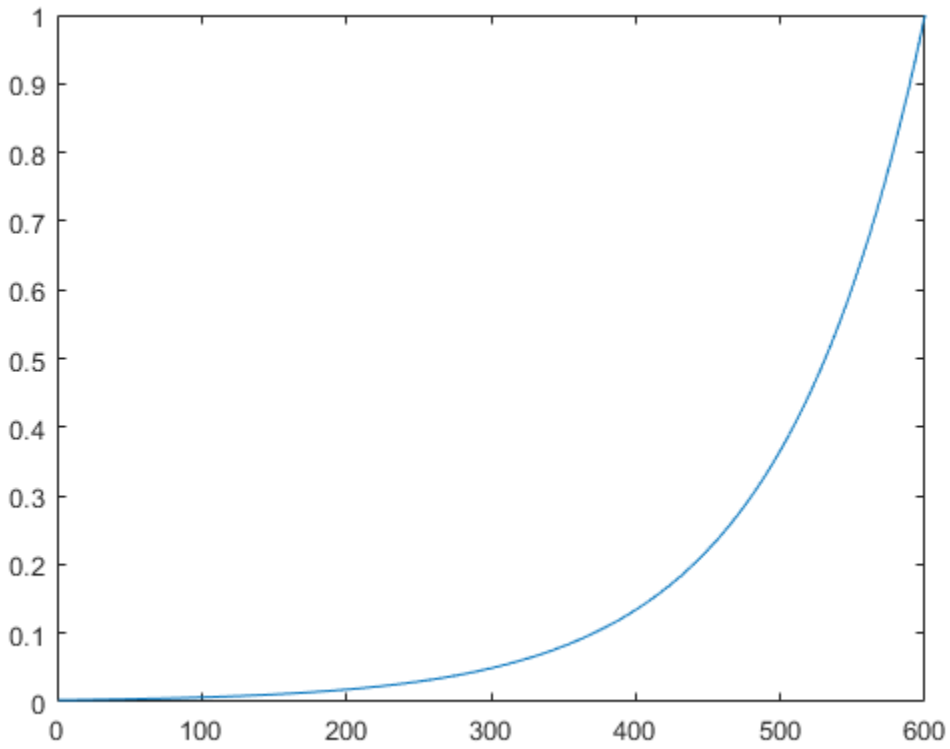
$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

However, the toolbox allows you to weight each squared error individually as follows:

$$F = mse = \frac{1}{N}\sum_{i=1}^{N}w_i^e(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}w_i^e(t_i - a_i)^2$$

The error weighting object needs to have the same dimensions as the target data. In this way, errors can be weighted according to time step, sample number, signal number or element number. The following is an example of weighting the errors at the end of a time sequence more heavily than errors at the beginning of a time sequence. The error weighting object is passed as the last argument in the call to `train`.

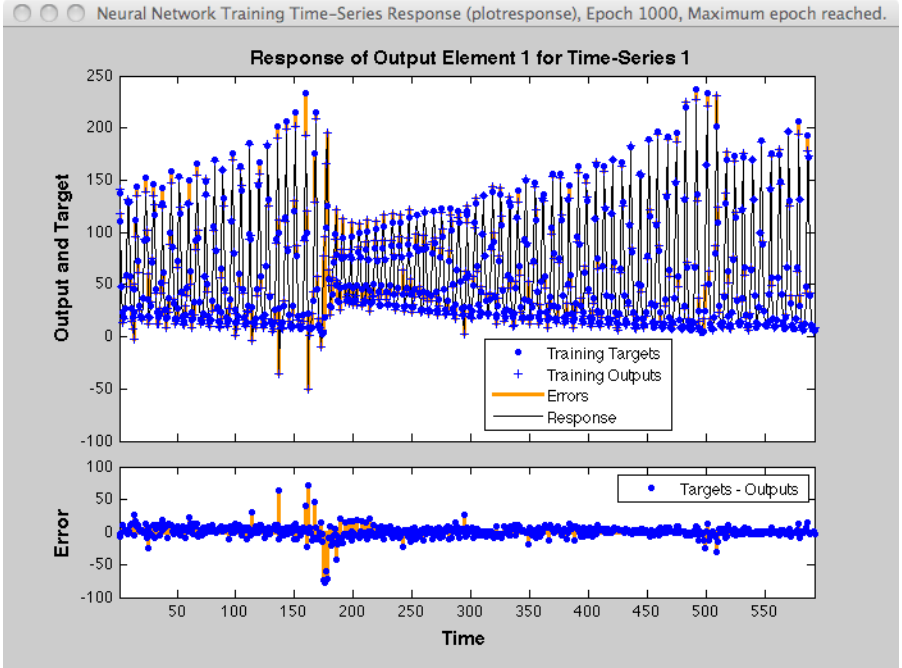
```
y = laser_dataset;
y = y(1:600);
ind = 1:600;
ew = 0.99.^(600-ind);
figure
plot(ew)
```




```
ew = con2seq(ew);
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = '';
[p,Pi,Ai,t,ew1] = preparets(ftdnn_net,y,y,{},ew);
[ftdnn_net1,tr] = train(ftdnn_net,p,t,Pi,Ai,ew1);
```

The figure illustrates the error weighting for this example. There are 600 time steps in the training data, and the errors are weighted exponentially, with the last squared error having a weight of 1, and the squared error at the first time step having a weighting of 0.0024.

The response of the trained network is shown in the following figure. If you compare this response to the response of the network that was trained without exponential weighting on the squared errors, as shown in “Design Time Series Time-Delay Neural Networks” on page 23-10, you can see that the errors late in the sequence are smaller than the errors earlier in the sequence. The errors that occurred later are smaller because they contributed more to the weighted performance index than earlier errors.


Neural Network Training Time-Series Response (plotresponse), Epoch 1000, Maximum epoch reached.
Response of Output Element 1 for Time-Series 1
Output and Target
Training Targets
Training Outputs
Errors
Response
Error
Targets - Outputs
Time

Normalize Errors of Multiple Outputs

The most common performance function used to train neural networks is mean squared error (mse). However, with multiple outputs that have different ranges of values, training with mean squared error tends to optimize accuracy on the output element with the wider range of values relative to the output element with a smaller range.

For instance, here two target elements have very different ranges:

```
x = -1:0.01:1;
t1 = 100*sin(x);
t2 = 0.01*cos(x);
t = [t1; t2];
```

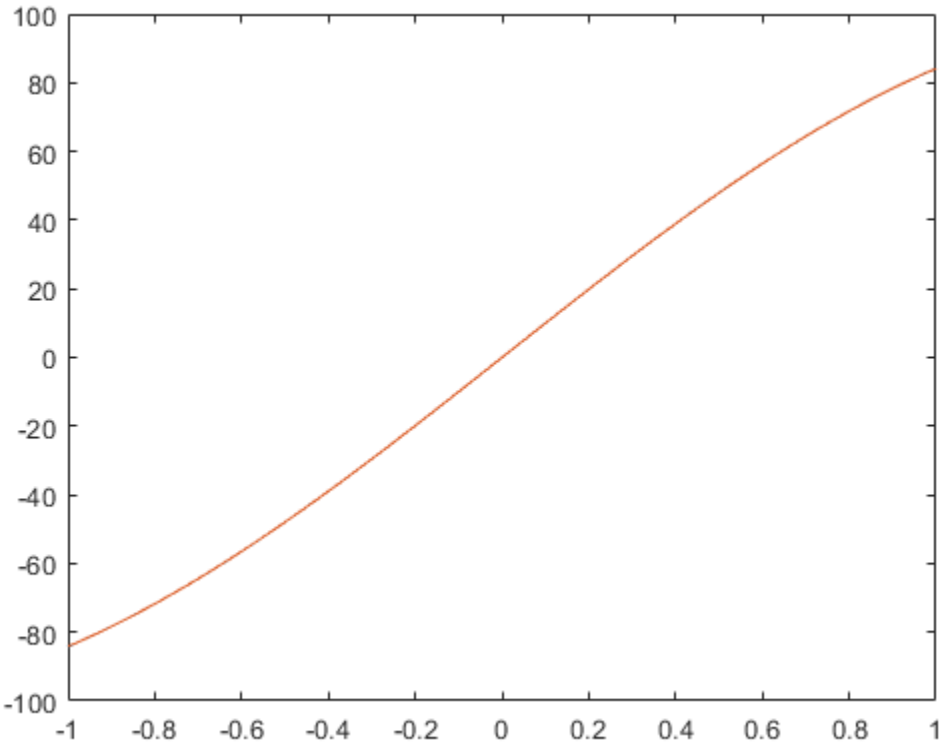
The range of `t1` is 200 (from a minimum of -100 to a maximum of 100), while the range of `t2` is only 0.02 (from -0.01 to 0.01). The range of `t1` is 10,000 times greater than the range of `t2`.

If you create and train a neural network on this to minimize mean squared error, training favors the relative accuracy of the first output element over the second.

```
net = feedforwardnet(5);
net1 = train(net,x,t);
y = net1(x);
```

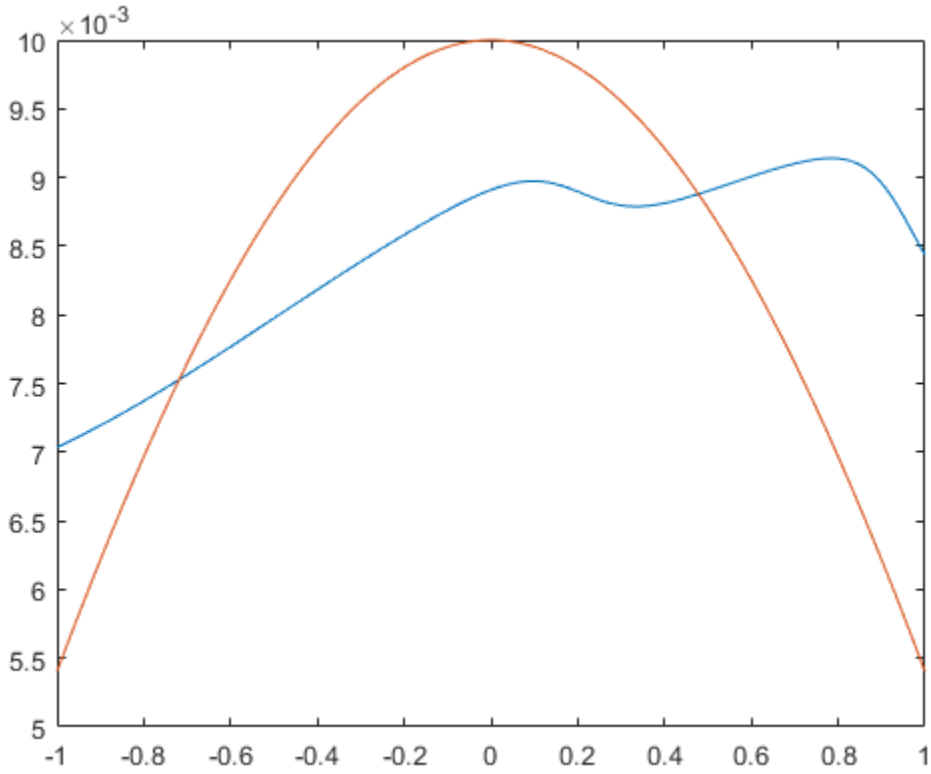
Here you can see that the network has learned to fit the first output element very well.

```
figure(1)
plot(x,y(1,:),x,t(1,:))
```



However, the second element's function is not fit nearly as well.

```
figure(2)
plot(x,y(2,:),x,t(2,:))
```

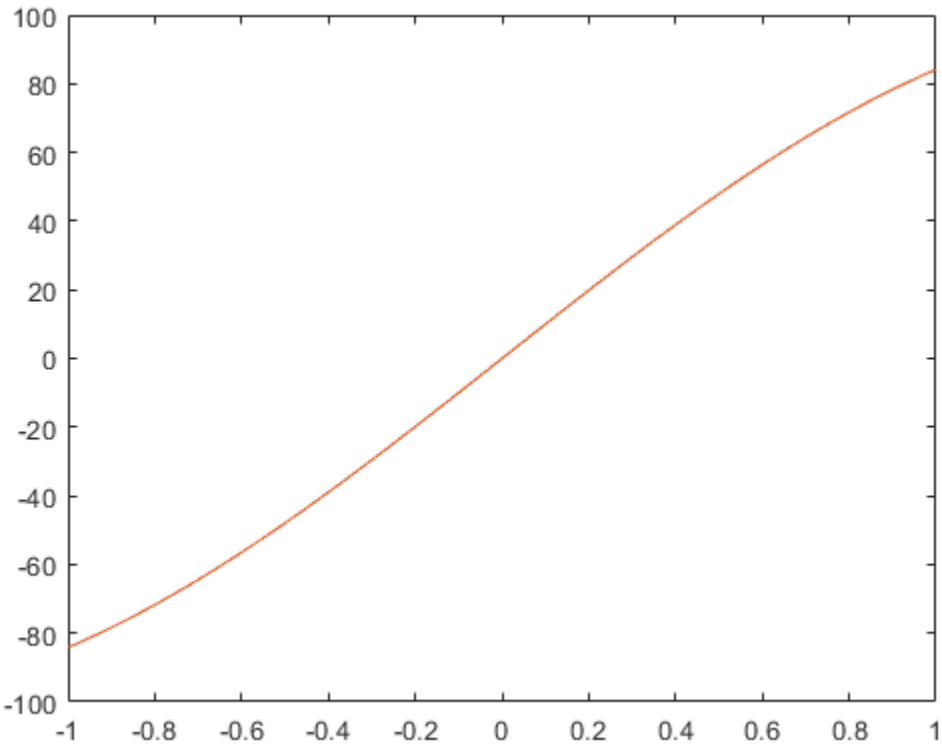


To fit both output elements equally well in a relative sense, set the `normalization` performance parameter to `'standard'`. This then calculates errors for performance measures as if each output element has a range of 2 (i.e., as if each output element's values range from -1 to 1, instead of their differing ranges).

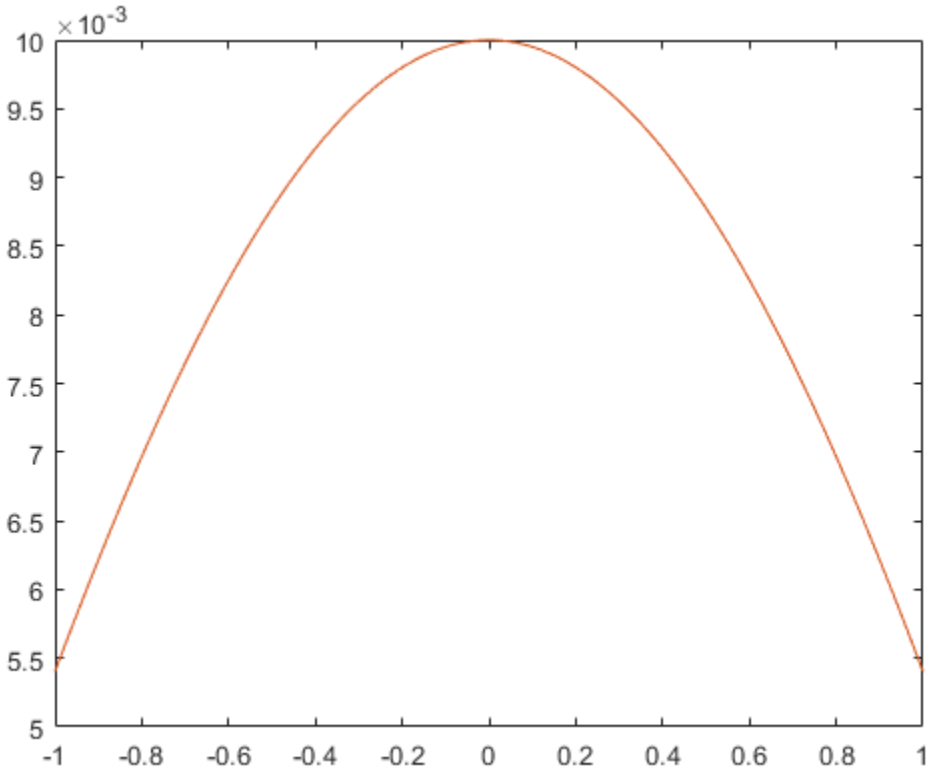
```
net.performParam.normalization = 'standard';
net2 = train(net,x,t);
y = net2(x);
```

Now the two output elements both fit well.

```
figure(3)
plot(x,y(1,:),x,t(1,:))
```




```
figure(4)
plot(x,y(2,:),x,t(2,:))
```


×10^-3
10
9.5
9
8.5
8
7.5
7
6.5
6
5.5
5
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Multistep Neural Network Prediction

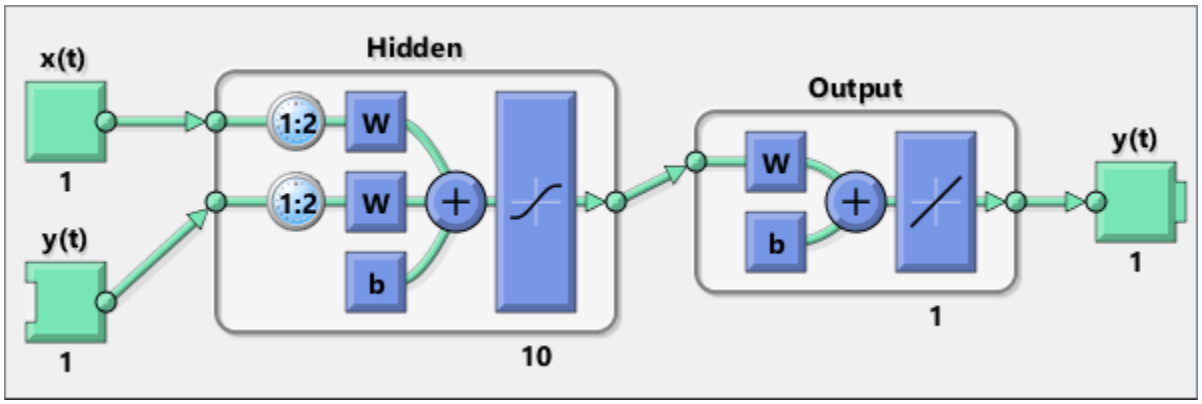
In this section...

Set Up in Open-Loop Mode

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words they continue to predict when external feedback is missing, by using internal feedback.

Here a neural network is trained to model the magnetic levitation system and simulated in the default open-loop mode.

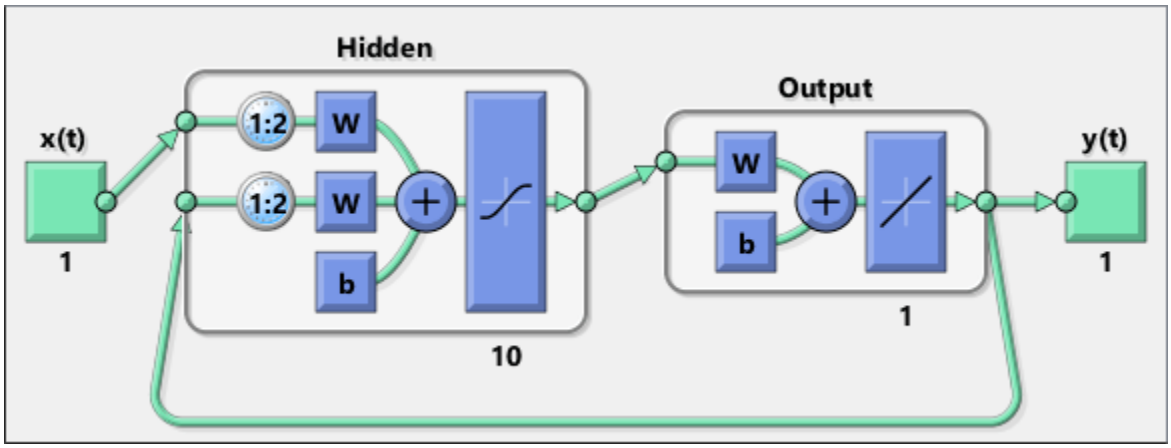
```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
view(net)
```



Multistep Closed-Loop Prediction From Initial Conditions

A neural network can also be simulated only in closed-loop form, so that given an external input series and initial conditions, the neural network performs as many predictions as the input series has time steps.

```
netc = closeloop(net);
view(netc)
```



Here the training data is used to define the inputs `x`, and the initial input and layer delay states, `xi` and `ai`, but they can be defined to make multiple predictions for any input series and initial states.

```
[x,xi,ai,t] = preparets(netc,X,{},T);
yc = netc(x,xi,ai);
```

Multistep Closed-Loop Prediction Following Known Sequence

It can also be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired.

Just as `openloop` and `closeloop` can be used to transform between open- and closed-loop neural networks, they can convert the state of open- and closed-loop networks. Here are the full interfaces for these functions.

```
[open_net,open_xi,open_ai] = openloop(closed_net,closed_xi,closed_ai);
[closed_net,closed_xi,closed_ai] = closeloop(open_net,open_xi,open_ai);
```

Consider the case where you might have a record of the Maglev's behavior for 20 time steps, and you want to predict ahead for 20 more time steps.

First, define the first 20 steps of inputs and targets, representing the 20 time steps where the known output is defined by the targets `t`. With the next 20 time steps of the input are defined, use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);
t1 = t(1:20);
x2 = x(21:40);
```

The open-loop neural network is then simulated on this data.

```
[x,xi,ai,t] = preparets(net,x1,{},t1);
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states `xf` and layer states `af` of the open-loop network become the initial input states `xi` and layer states `ai` of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically use `preparets` to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `preparets` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that you can set `x2` to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs are used:

```
x2 = num2cell(rand(1,10));
[y2,xf,af] = netc(x2,xi,ai);
```

Following Closed-Loop Simulation with Open-Loop Simulation

If after simulating the network in closed-loop form, you can continue the simulation from there in open-loop form. Here the closed-loop state is converted back to open-loop state. (You do not have to convert the network back to open-loop form as you already have the original open-loop network.)

```
[~,xi,ai] = openloop(netc,xf,af);
```

Now you can define continuations of the external input and open-loop feedback, and simulate the open-loop network.

```
x3 = num2cell(rand(2,10));
y3 = net(x3,xi,ai);
```

In this way, you can switch simulation between open-loop and closed-loop manners. One application for this is making time-series predictions of a sensor, where the last sensor value is usually known, allowing open-loop prediction of the next step. But on some occasions the sensor reading is not available, or known to be erroneous, requiring a closed-loop prediction step. The predictions can alternate between open-loop and closed-loop form, depending on the availability of the last step's sensor reading.

Control Systems

Introduction to Neural Network Control Systems

Neural networks have been applied successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron make it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers [HaDe99 on page 32-2]. This topic introduces three popular neural network architectures for prediction and control that have been implemented in the Deep Learning Toolbox software, and presents brief descriptions of each of these architectures and shows how you can use them:

- Model Predictive Control
- NARMA-L2 (or Feedback Linearization) Control
- Model Reference Control

There are typically two steps involved when using neural networks for control:

1. System identification
2. Control design

In the system identification stage, you develop a neural network model of the plant that you want to control. In the control design stage, you use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this topic, the system identification stage is identical. The control design stage, however, is different for each architecture:

- For model predictive control, the plant model is used to predict future behavior of the plant, and an optimization algorithm is used to select the control input that optimizes future performance.
- For NARMA-L2 control, the controller is simply a rearrangement of the plant model.
- For model reference control, the controller is a neural network that is trained to control a plant so that it follows a reference model. The neural network plant model is used to assist in the controller training.

The next three sections discuss model predictive control, NARMA-L2 control, and model reference control. Each section consists of a brief description of the control concept, followed by an example of the use of the appropriate Deep Learning Toolbox function. These three controllers are implemented as Simulink blocks, which are contained in the Deep Learning Toolbox blockset.

To assist you in determining the best controller for your application, the following list summarizes the key controller features. Each controller has its own strengths and weaknesses. No single controller is appropriate for every application.

- **Model Predictive Control** — This controller uses a neural network model to predict future plant responses to potential control signals. An optimization algorithm then computes the control signals that optimize future plant performance. The neural network plant model is trained offline, in batch form. (This is true for all three control architectures.) The controller, however, requires a significant amount of online computation, because an optimization algorithm is performed at each sample time to compute the optimal control input.
- **NARMA-L2 Control** — This controller requires the least computation of these three architectures. The controller is simply a rearrangement of the neural network plant model, which is trained offline, in batch form. The only online computation is a forward pass through the neural network controller. The drawback of this method is that the plant must either be in companion form, or be capable of approximation by a companion form model. (“Identification of the NARMA-L2 Model” on page 24-13 describes the companion form model.)

- **Model Reference Control** — The online computation of this controller, like NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained offline, in addition to the neural network plant model. The controller training is computationally expensive, because it requires the use of dynamic backpropagation [HaJe99 on page 32-2]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control.

Design Neural Network Predictive Controller in Simulink

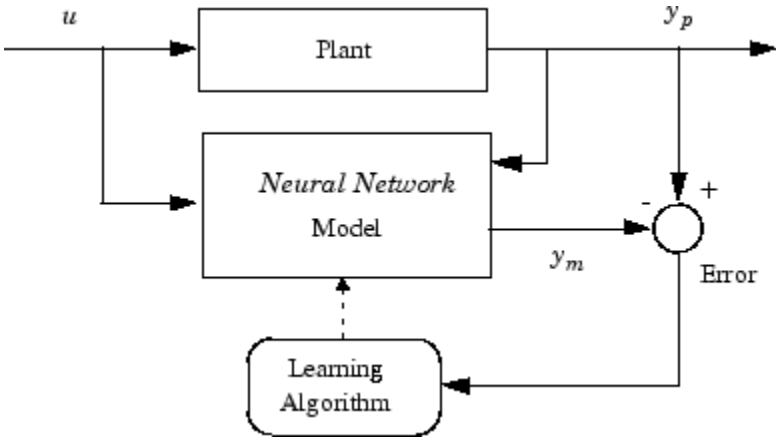
In this section...

The neural network predictive controller that is implemented in the Deep Learning Toolbox software uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance. (See the Model Predictive Control Toolbox™ documentation for complete coverage of the application of various model predictive control strategies to linear systems.)

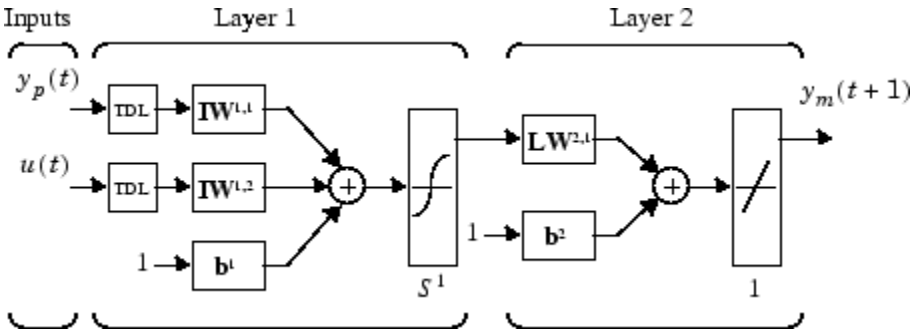
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink environment.

System Identification

The first stage of model predictive control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by the following figure:



The neural network plant model uses previous inputs and previous plant outputs to predict future values of the plant output. The structure of the neural network plant model is given in the following figure.



This network can be trained offline in batch mode, using data collected from the operation of the plant. You can use any of the training algorithms discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2 for network training. This process is discussed in more detail in following sections.

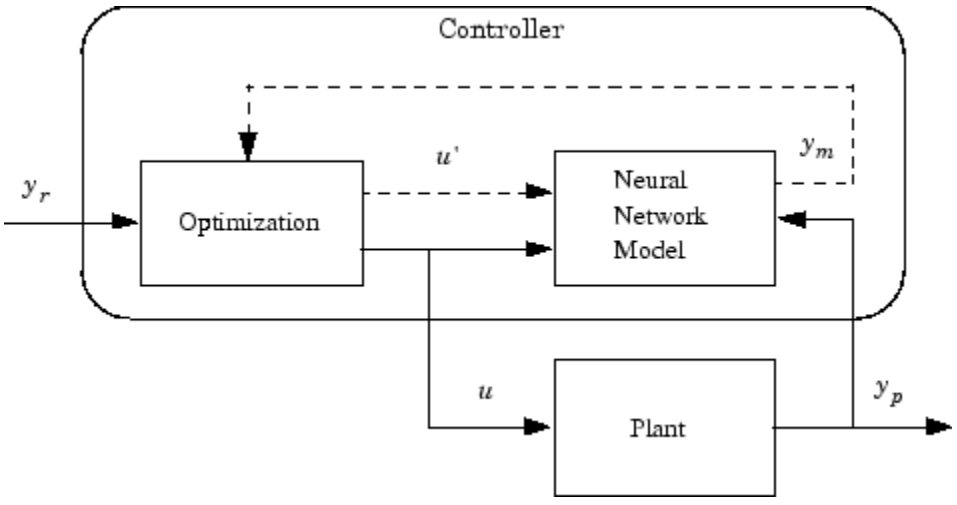
Predictive Control

The model predictive control method is based on the receding horizon technique [SoHa96 on page 32-2]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon

$$J = \sum_{j=N_1}^{N_2} (y_r(t+j) - y_m(t+j))^2 + \rho \sum_{j=1}^{N_u} (u'(t+j-1) - u'(t+j-2))^2$$

where N_1, N_2, and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response, and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index.

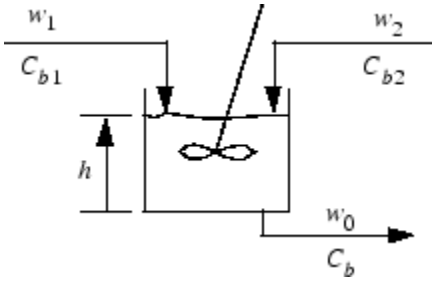
The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and then the optimal u is input to the plant. The controller block is implemented in Simulink, as described in the following section.



Use the Neural Network Predictive Controller Block

This section shows how the NN Predictive Controller block is used. The first step is to copy the NN Predictive Controller block from the Deep Learning Toolbox block library to the Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the predictive controller. This example uses a catalytic Continuous Stirred Tank Reactor (CSTR). A diagram of the process is shown in the following figure.



The dynamic model of the system is

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

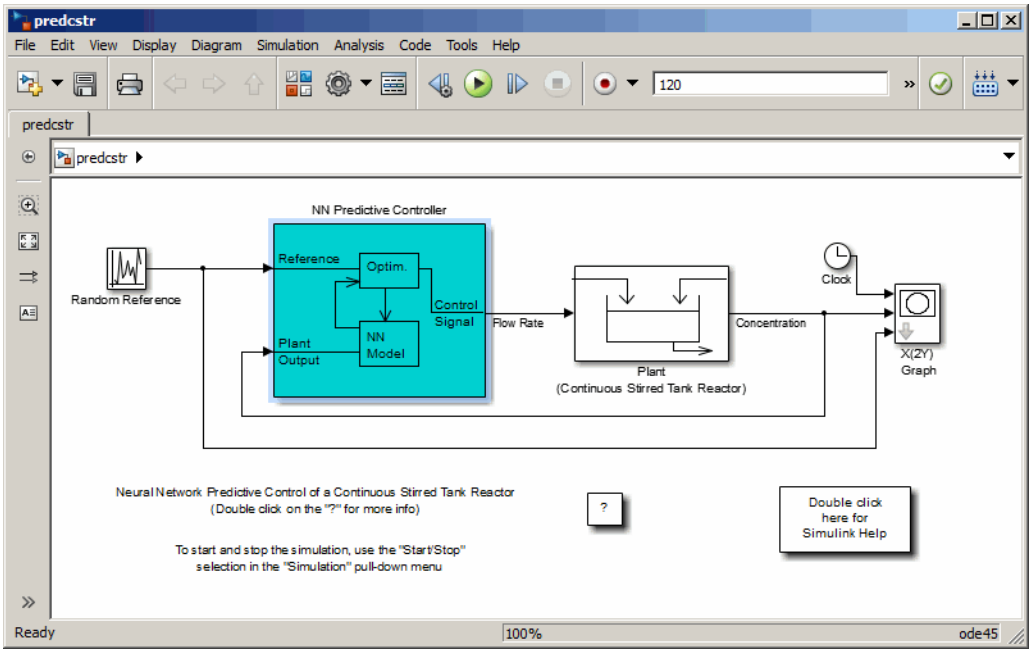
$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1}, and $w_2(t)$ is the flow rate of the diluted feed C_{b2}. The input concentrations are set to $C_{b1} = 24.9$ and $C_{b2} = 0.1$. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$.

The objective of the controller is to maintain the product concentration by adjusting the flow $w_1(t)$. To simplify the example, set $w_2(t) = 0.1$. The level of the tank $h(t)$ is not controlled for this experiment.

To run this example:

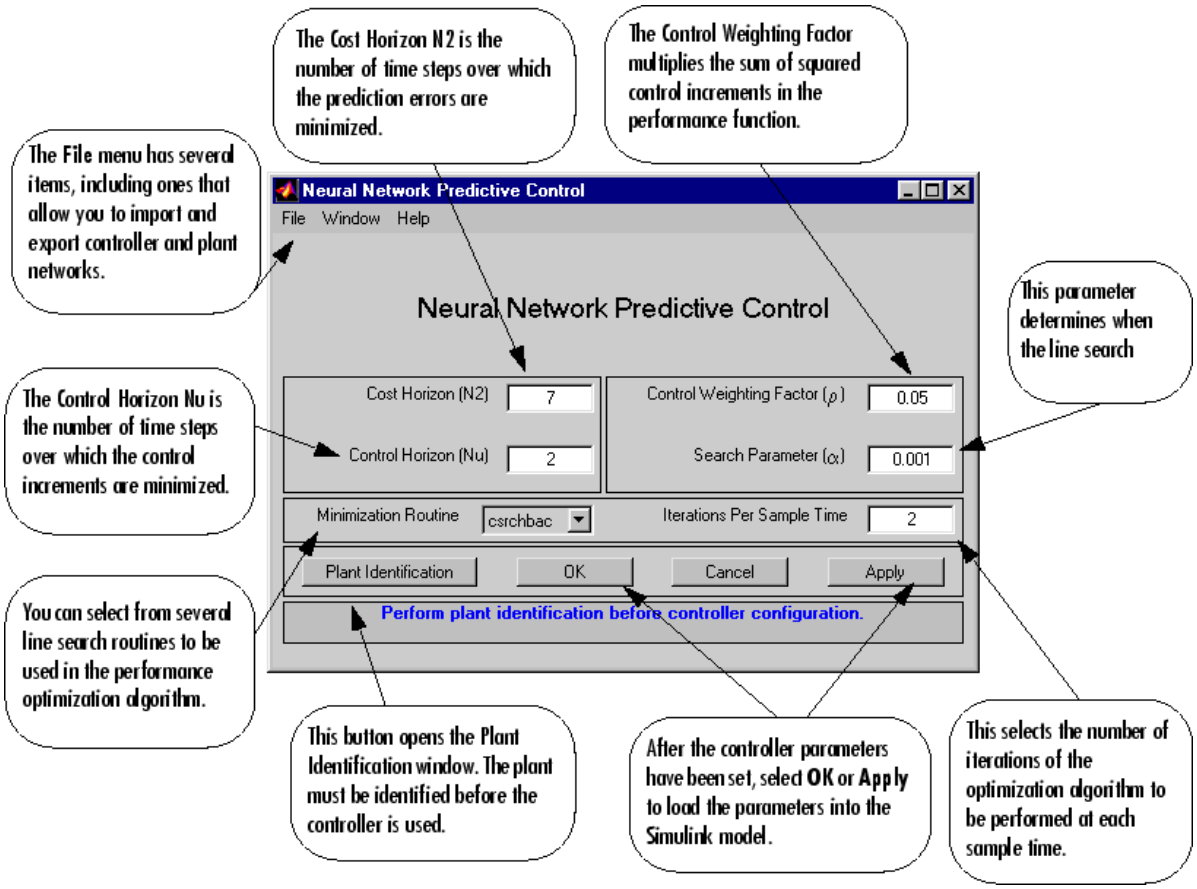
1. Start MATLAB.
2. Type `predcstr` in the MATLAB Command Window. This command opens the Simulink Editor with the following model.



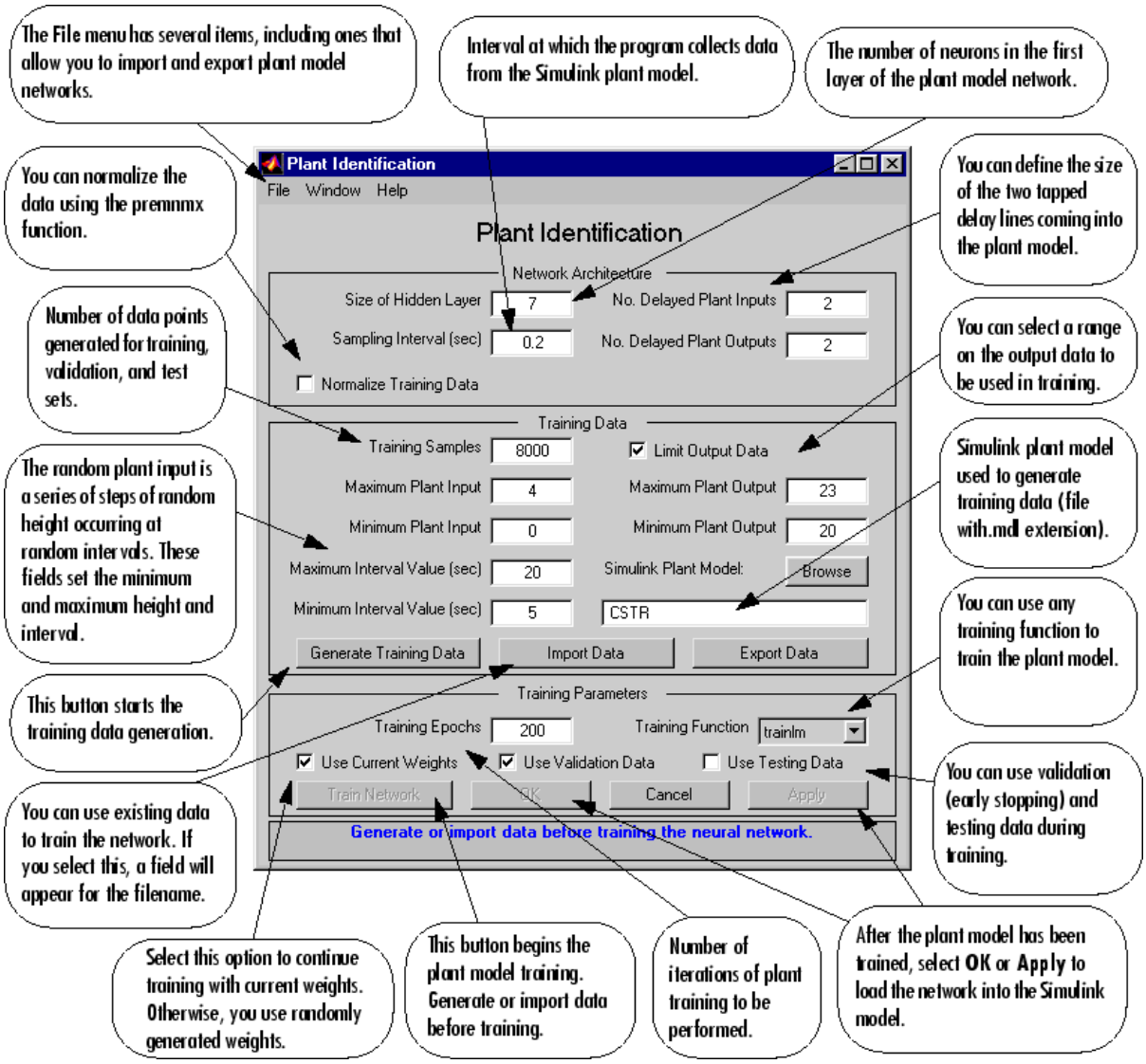
The Plant block contains the Simulink CSTR plant model. The NN Predictive Controller block signals are connected as follows:

- Control Signal is connected to the input of the Plant model.
- The Plant Output signal is connected to the Plant block output.
- The Reference is connected to the Random Reference signal.

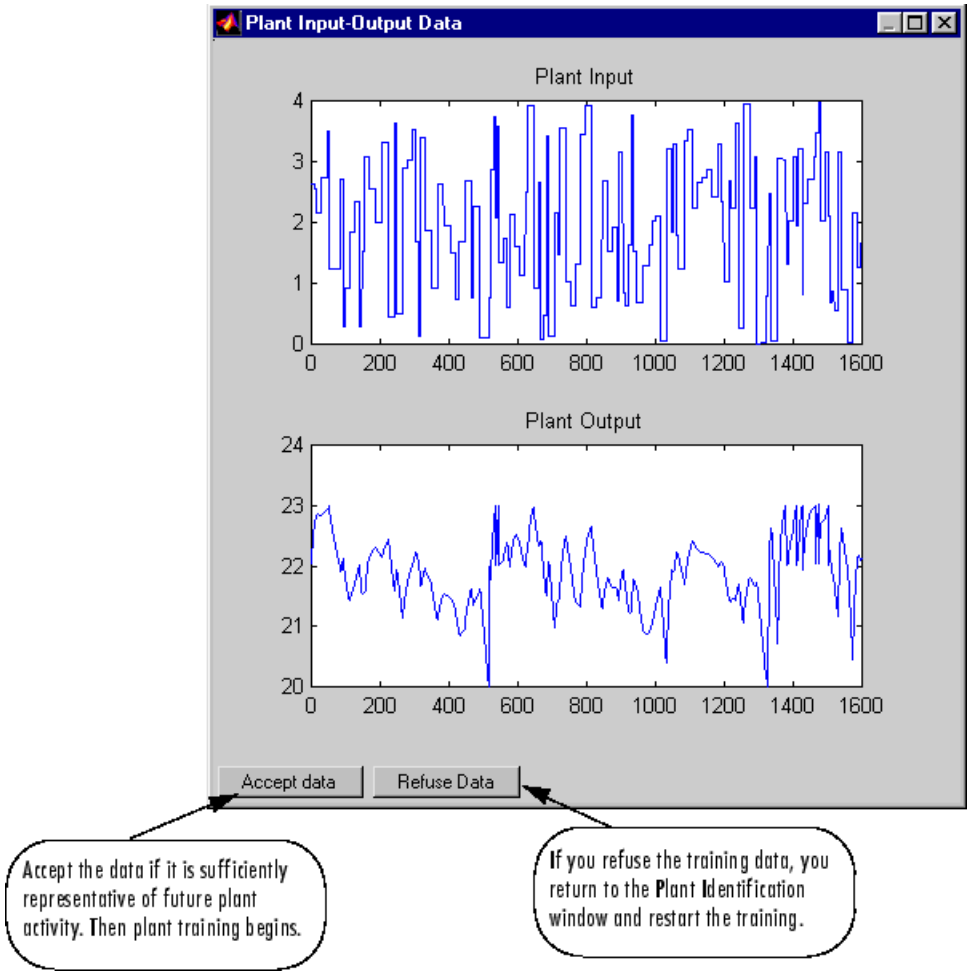
3 Double-click the NN Predictive Controller block. This opens the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u. (N_1 is fixed at 1.) The weighting parameter ρ, described earlier, is also defined in this window. The parameter α is used to control the optimization. It determines how much reduction in performance is required for a successful optimization step. You can select which linear minimization routine is used by the optimization algorithm, and you can decide how many iterations of the optimization algorithm are performed at each sample time. The linear minimization routines are slight modifications of those discussed in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2.



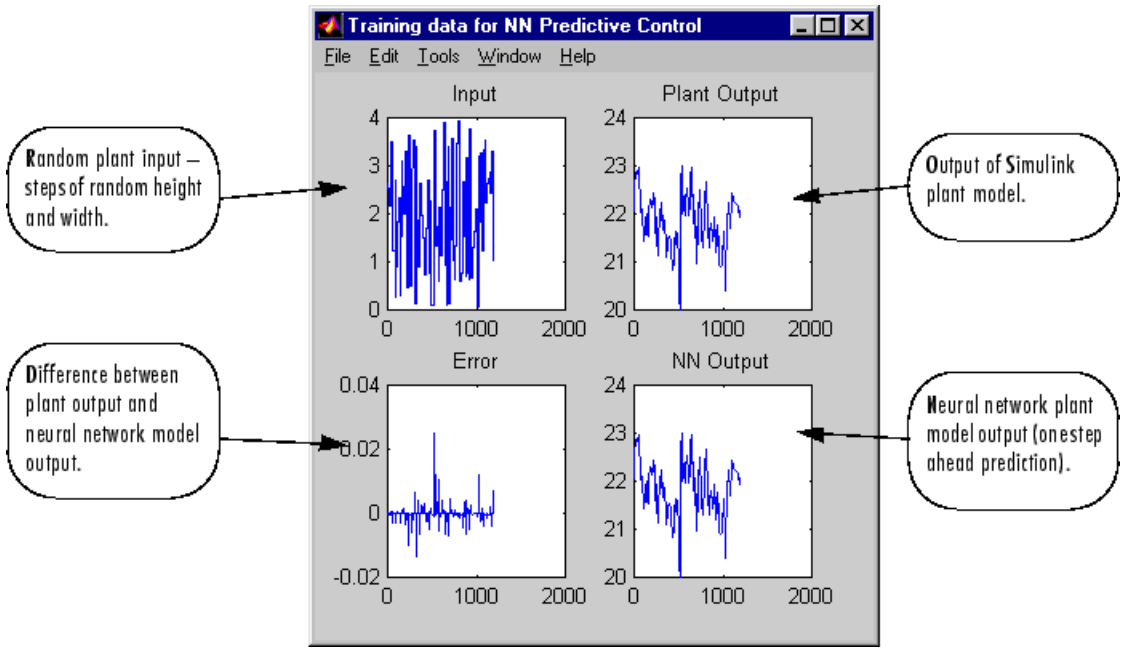
4 Select **Plant Identification**. This opens the following window. You must develop the neural network plant model before you can use the controller. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future performance. The plant model neural network has one hidden layer, as shown earlier. You select the size of that layer, the number of delayed inputs and delayed outputs, and the training function in this window. You can select any of the training functions described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2 to train the neural network plant model.



5 Click **Generate Training Data**. The program generates training data by applying a series of random step inputs to the Simulink plant model. The potential training data is then displayed in a figure similar to the following.

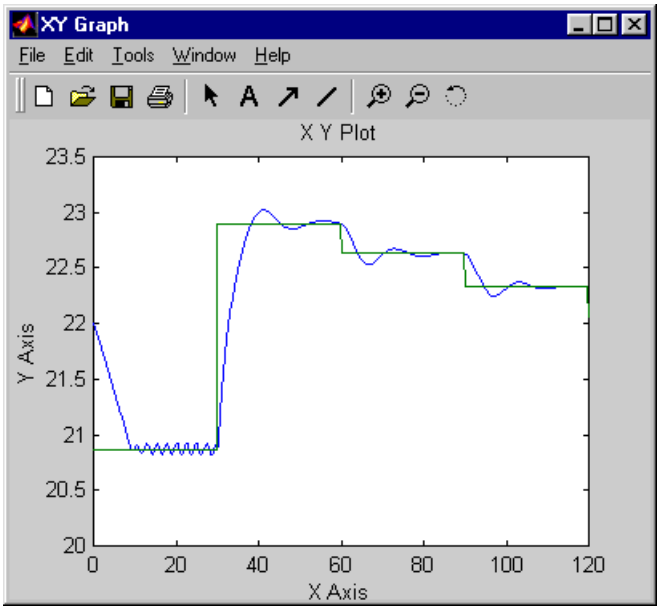


6 Click **Accept Data**, and then click **Train Network** in the Plant Identification window. Plant model training begins. The training proceeds according to the training algorithm (`trainlm` in this case) you selected. This is a straightforward application of batch training, as described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2. After the training is complete, the response of the resulting plant model is displayed, as in the following figure. (There are also separate plots for validation and testing data, if they exist.)



You can then continue training with the same data set by selecting **Train Network** again, you can **Erase Generated Data** and generate a new data set, or you can accept the current plant model and begin simulating the closed loop system. For this example, begin the simulation, as shown in the following steps.

7. Select **OK** in the Plant Identification window. This loads the trained neural network plant model into the NN Predictive Controller block.
8. Select **OK** in the Neural Network Predictive Control window. This loads the controller parameters into the NN Predictive Controller block.
9. Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
23.5
23
22.5
22
21.5
21
20.5
20
Y Axis
0
20
40
60
80
100
120
X Axis

Design NARMA-L2 Neural Controller in Simulink

In this section...

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form). It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by canceling the nonlinearities. This section begins by presenting the companion form system model and showing how you can use a neural network to identify this model. Then it describes how the identified neural network model can be used to develop a controller. This is followed by an example of how to use the NARMA-L2 Control block, which is contained in the Deep Learning Toolbox blockset.

Identification of the NARMA-L2 Model

As with model predictive control, the first step in using feedback linearization (or NARMA-L2) control is to identify the system to be controlled. You train a neural network to represent the forward dynamics of the system. The first step is to choose a model structure to use. One standard model that is used to represent general discrete-time nonlinear systems is the nonlinear autoregressive-moving average (NARMA) model:

$$y(k+d) = N[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)]$$

where $u(k)$ is the system input, and $y(k)$ is the system output. For the identification phase, you could train a neural network to approximate the nonlinear function N. This is the identification procedure used for the NN Predictive Controller.

If you want the system output to follow some reference trajectory $y(k+d) = y_r(k+d)$, the next step is to develop a nonlinear controller of the form:

$$u(k) = G[y(k), y(k-1), \ldots, y(k-n+1), y_r(k+d), u(k-1), \ldots, u(k-m+1)]$$

The problem with using this controller is that if you want to train a neural network to create the function G to minimize mean square error, you need to use dynamic backpropagation ([NaPa91 on page 32-2] or [HaJe99 on page 32-2]). This can be quite slow. One solution, proposed by Narendra and Mukhopadhyay [NaMu97 on page 32-2], is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\begin{aligned}\widehat{y}(k+d) = {} & f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \\ & + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)\end{aligned}$$

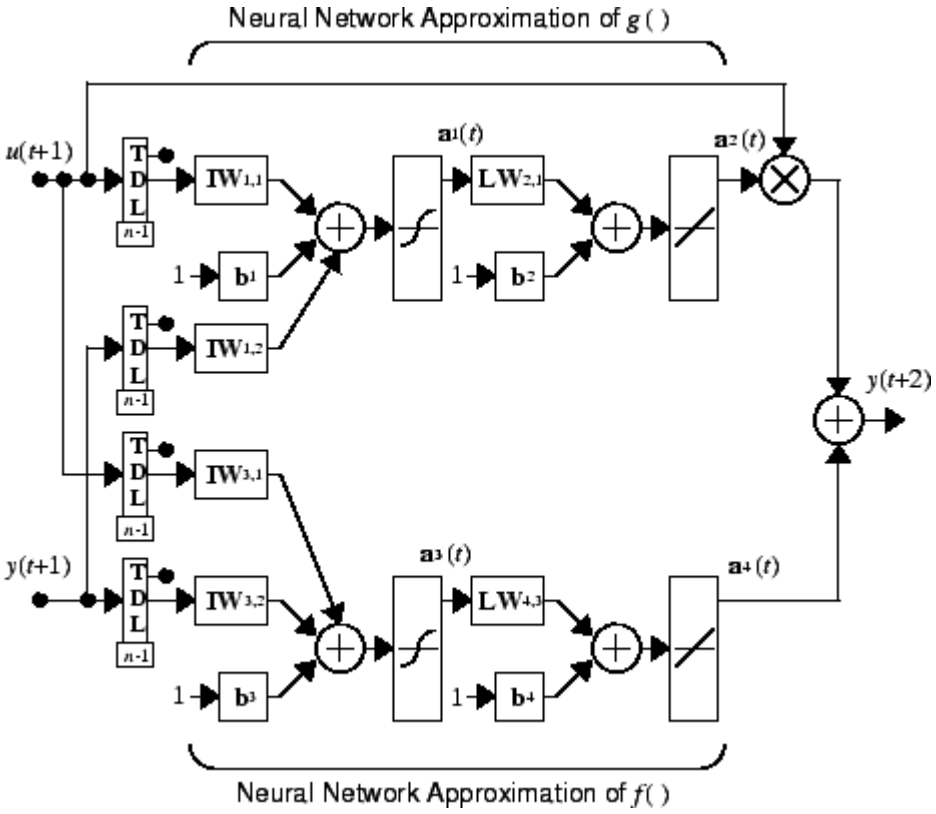
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

Using this equation directly can cause realization problems, because you must determine the control input $u(k)$ based on the output at the same time, $y(k)$. So, instead, use the model

$$y(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-n+1)] + g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)] \cdot u(k+1)$$

where $d \geq 2$. The following figure shows the structure of a neural network representation.

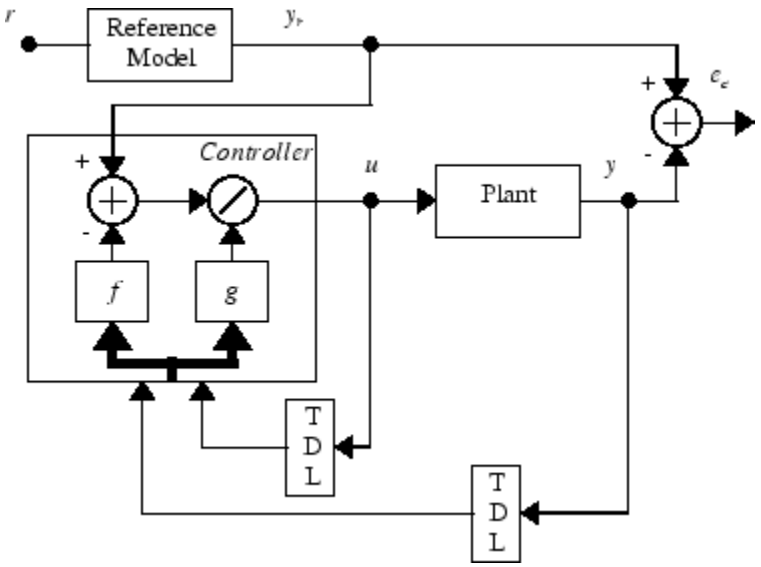


NARMA-L2 Controller

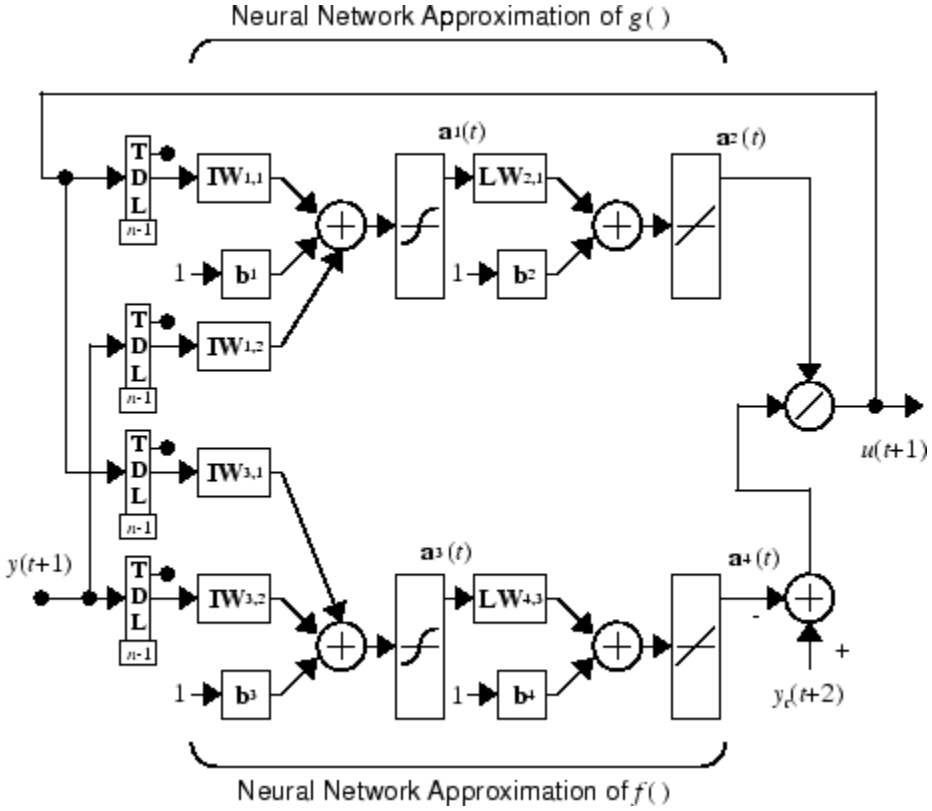
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

which is realizable for $d \geq 2$. The following figure is a block diagram of the NARMA-L2 controller.



This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in the following figure.

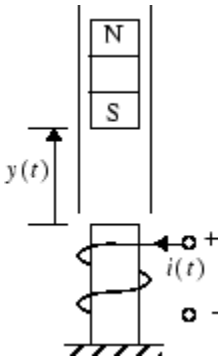


Use the NARMA-L2 Controller Block

This section shows how the NARMA-L2 controller is trained. The first step is to copy the NARMA-L2 Controller block from the Deep Learning Toolbox block library to the Simulink Editor. See the

Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the NARMA-L2 controller. In this example, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as in the following figure.



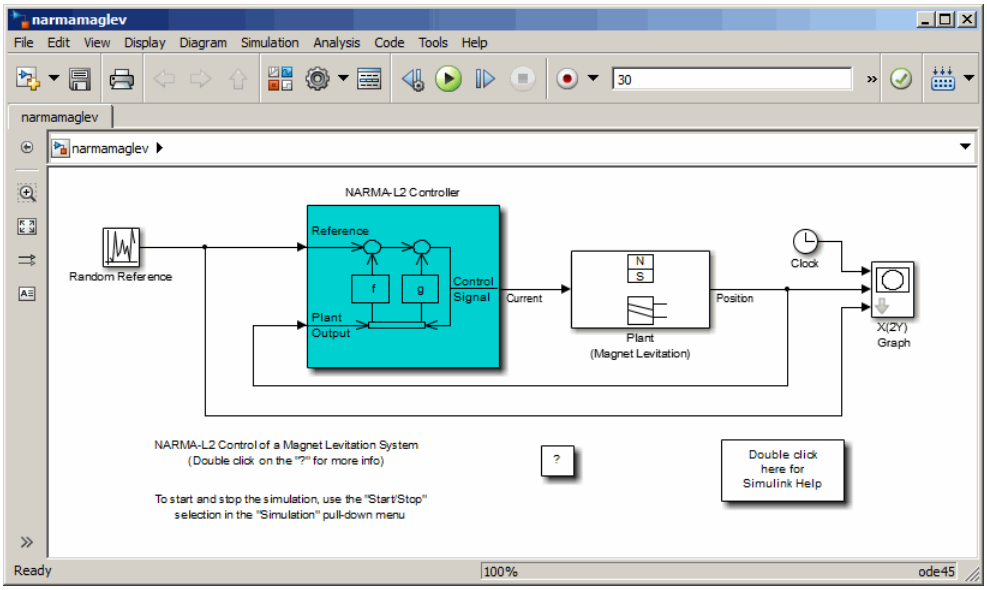
The equation of motion for this system is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

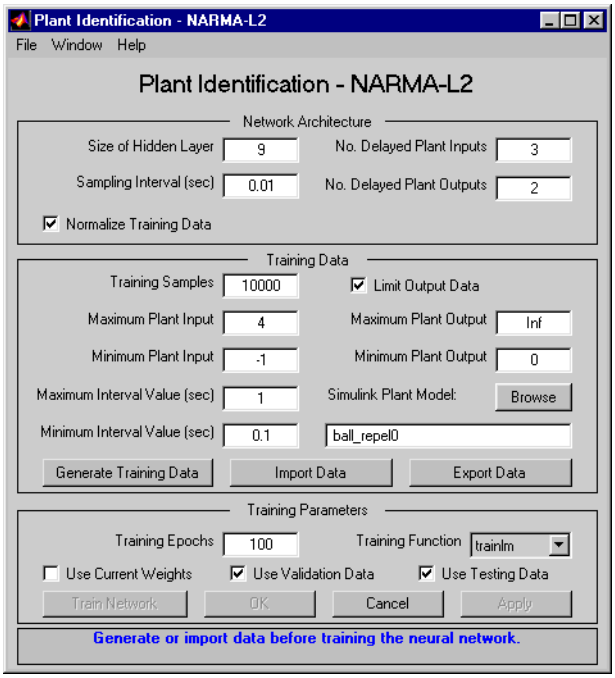
where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet.

To run this example:

1. Start MATLAB.
2. Type `narmamaglev` in the MATLAB Command Window. This command opens the Simulink Editor with the following model. The NARMA-L2 Control block is already in the model.

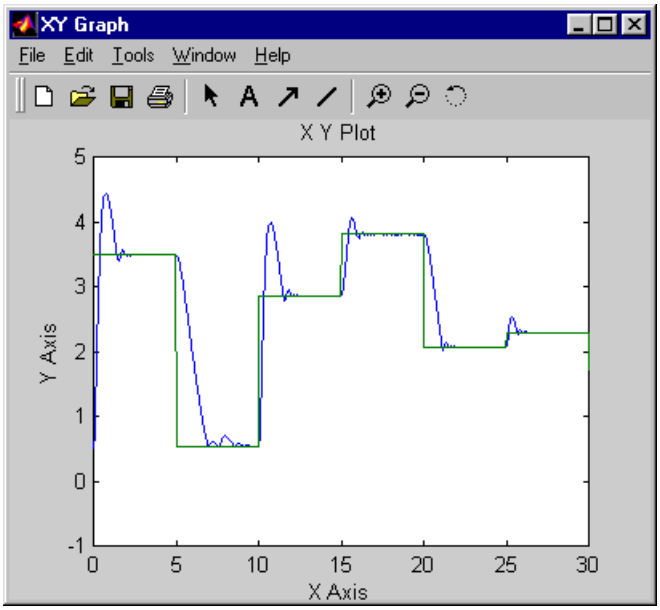


3 Double-click the NARMA-L2 Controller block. This opens the following window. This window enables you to train the NARMA-L2 model. There is no separate window for the controller, because the controller is determined directly from the model, unlike the model predictive controller.



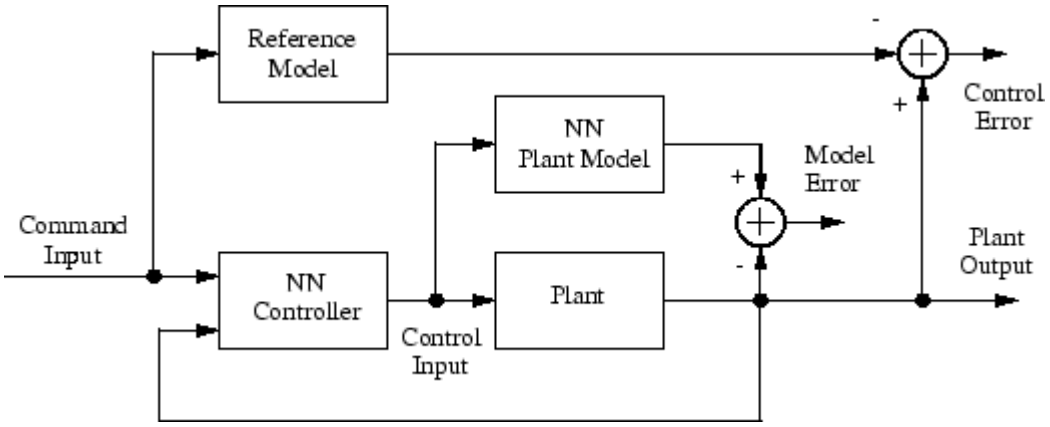
4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.

5 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
Y Axis
X Axis
5
4
3
2
1
0
-1
0
5
10
15
20
25
30

Design Model-Reference Neural Controller in Simulink

The neural model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in the following figure. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.



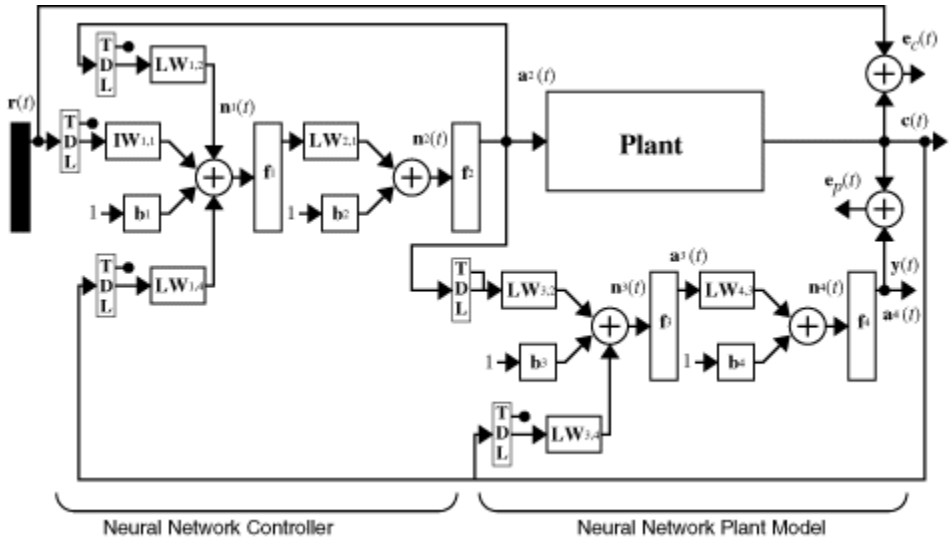
The following figure shows the details of the neural network plant model and the neural network controller as they are implemented in the Deep Learning Toolbox software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller inputs:

- Delayed reference inputs
- Delayed controller outputs
- Delayed plant outputs

For each of these inputs, you can select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model:

- Delayed controller outputs
- Delayed plant outputs

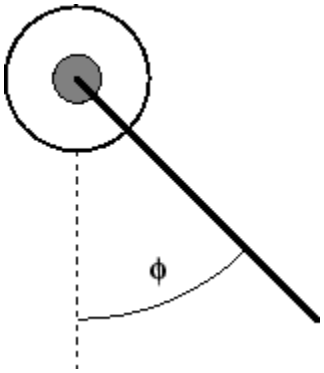
As with the controller, you can set the number of delays. The next section shows how you can set the parameters.



Use the Model Reference Controller Block

This section shows how the neural network controller is trained. The first step is to copy the Model Reference Control block from the Deep Learning Toolbox blockset to Simulink Editor. See the Simulink documentation if you are not sure how to do this. This step is skipped in the following example.

An example model is provided with the Deep Learning Toolbox software to show the use of the model reference controller. In this example, the objective is to control the movement of a simple, single-link robot arm, as shown in the following figure:



The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10\sin\phi - 2\frac{d\phi}{dt} + u$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor.

The objective is to train the controller so that the arm tracks the reference model

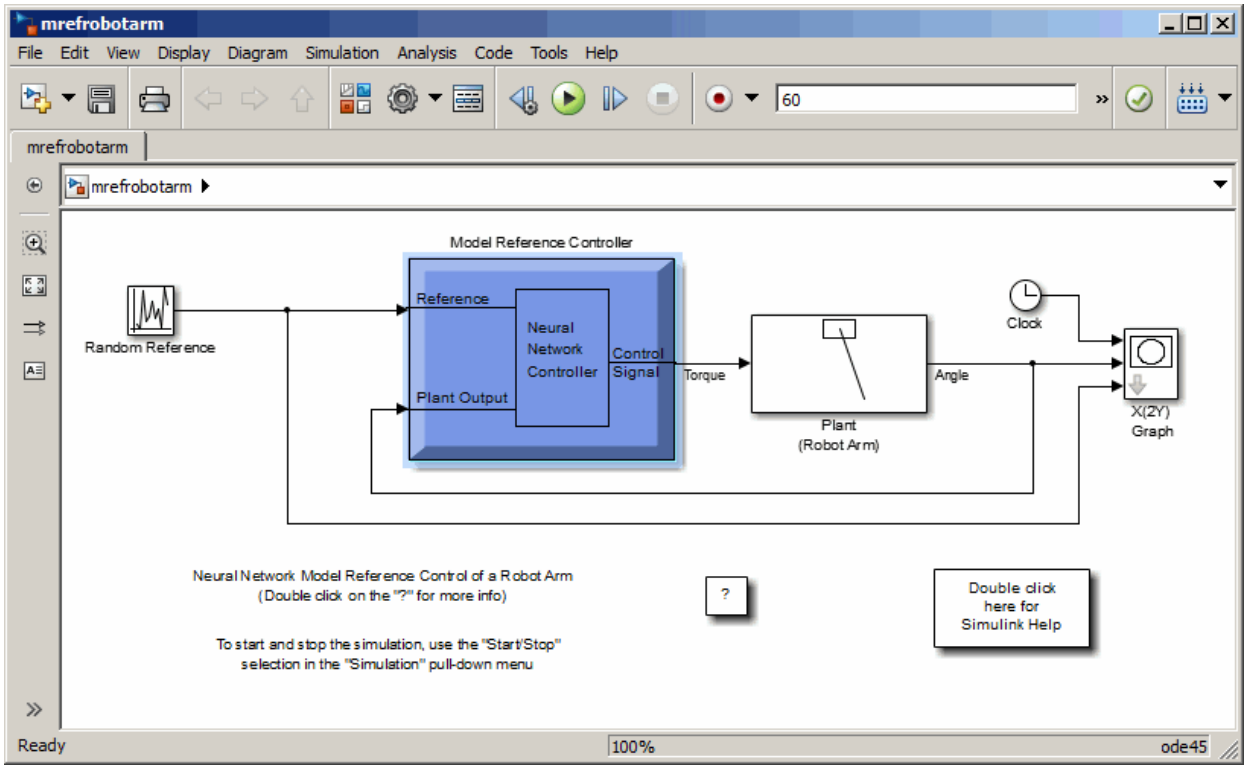
$$\frac{d^2y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

where y_r is the output of the reference model, and r is the input reference signal.

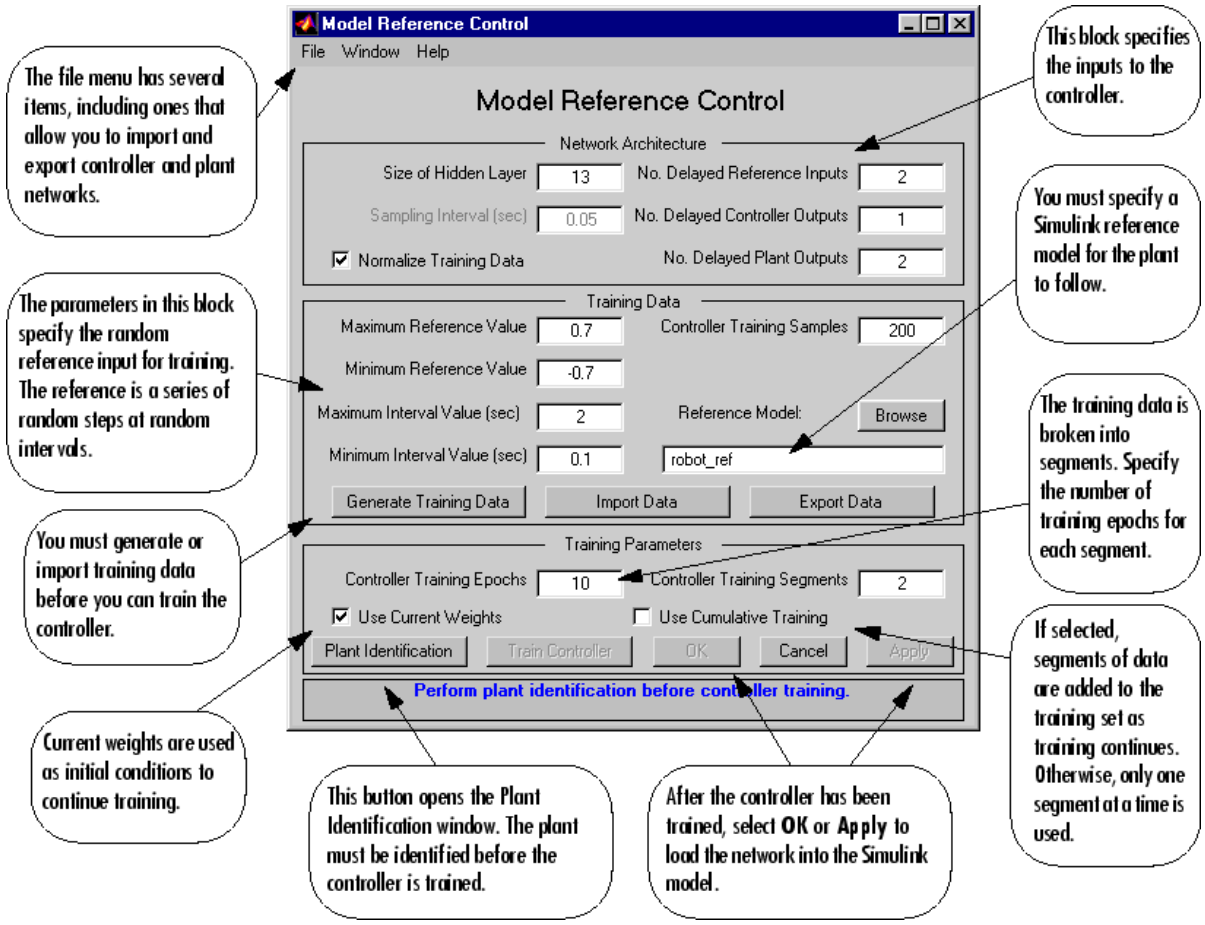
This example uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

To run this example:

1. Start MATLAB.
2. Type `mrefrobotarm` in the MATLAB Command Window. This command opens the Simulink Editor with the Model Reference Control block already in the model.

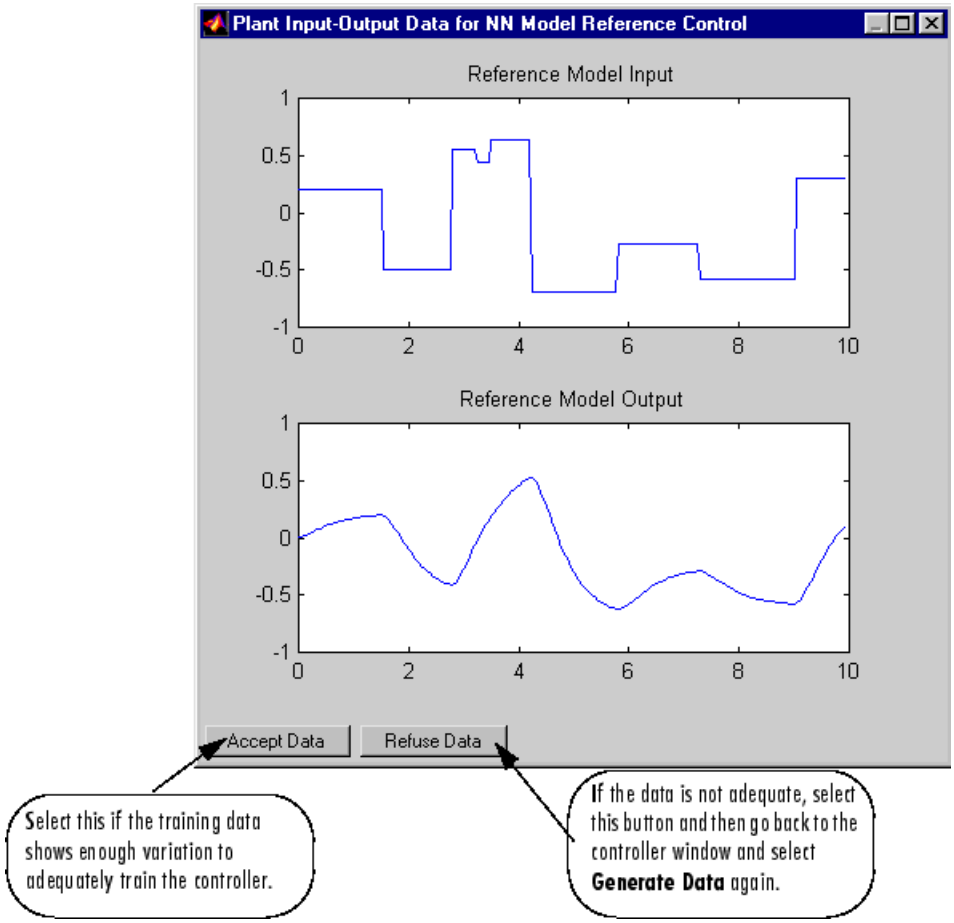


3. Double-click the Model Reference Control block. This opens the following window for training the model reference controller.

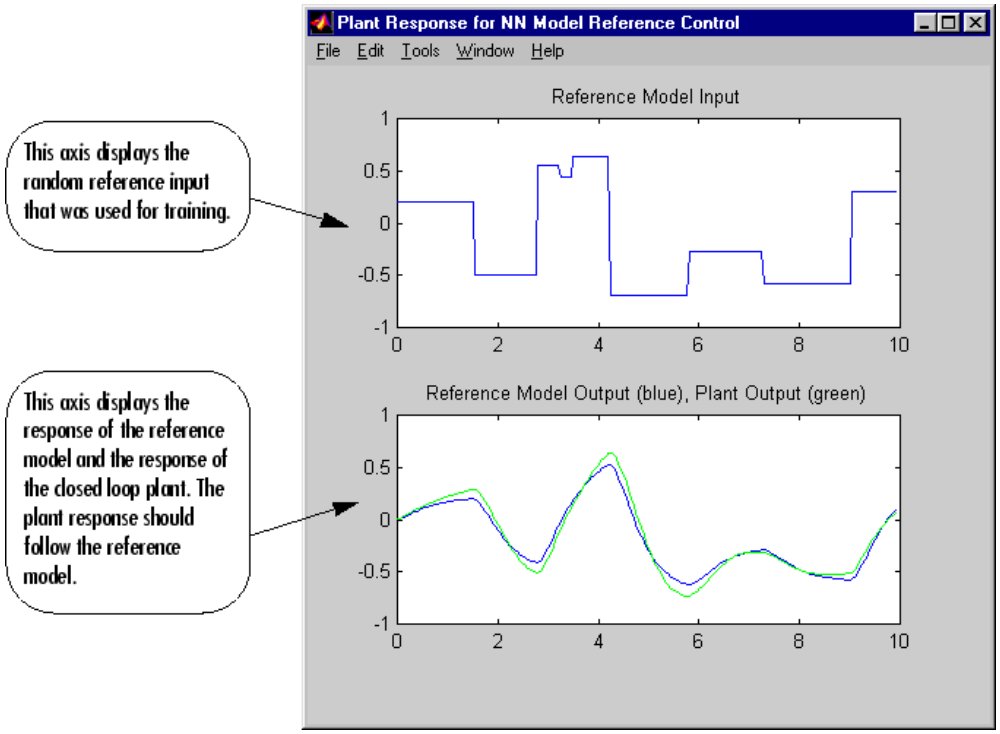


4 The next step would normally be to click **Plant Identification**, which opens the Plant Identification window. You would then train the plant model. Because the Plant Identification window is identical to the one used with the previous controllers, that process is omitted here.

5 Click **Generate Training Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.

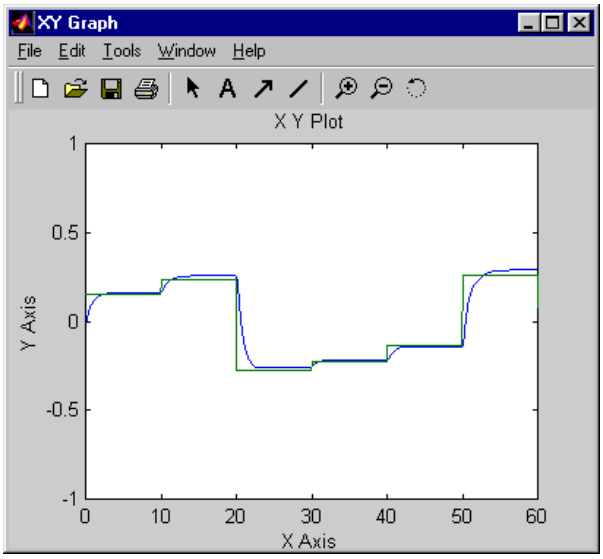


6 Click **Accept Data**. Return to the Model Reference Control window and click **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues, one segment at a time, until the entire training set has been presented to the network. Controller training can be significantly more time consuming than plant model training. This is because the controller must be trained using *dynamic* backpropagation (see [HaJe99 on page 32-2]). After the training is complete, the response of the resulting closed loop system is displayed, as in the following figure.



7 Go back to the Model Reference Control window. If the performance of the controller is not accurate, then you can select **Train Controller** again, which continues the controller training with the same data set. If you would like to use a new data set to continue training, select **Generate Data** or **Import Data** before you select **Train Controller**. (Be sure that **Use Current Weights** is selected if you want to continue training with the same weights.) It might also be necessary to retrain the plant model. If the plant model is not accurate, it can affect the controller training. For this example, the controller should be accurate enough, so select **OK**. This loads the controller weights into the Simulink model.

8 Return to the Simulink Editor and start the simulation by choosing the menu option **Simulation > Run**. As the simulation runs, the plant output and the reference signal are displayed, as in the following figure.


XY Graph
File Edit Tools Window Help
X Y Plot
1
0.5
0
-0.5
-1
Y Axis
0 10 20 30 40 50 60
X Axis

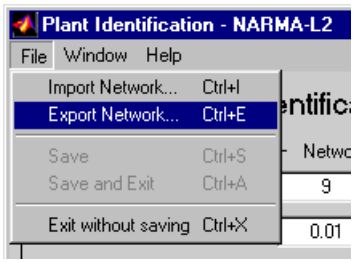
Import-Export Neural Network Simulink Control Systems

In this section...

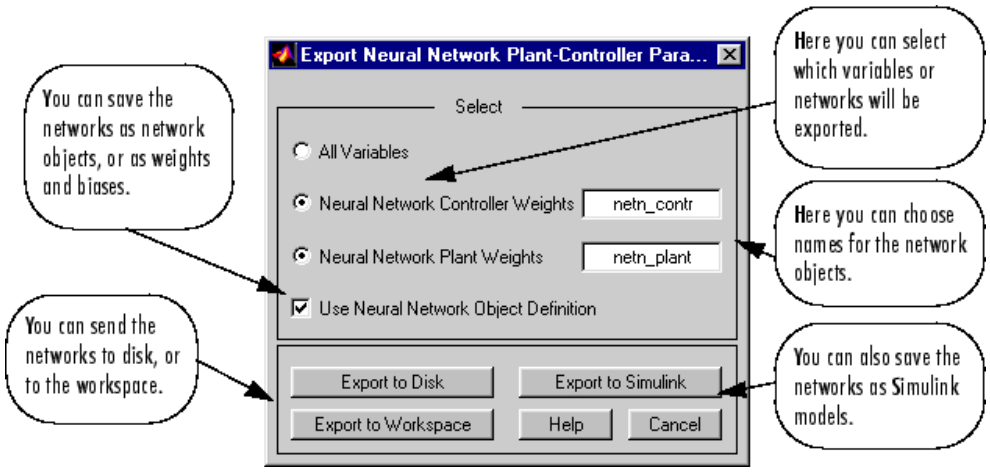
Import and Export Networks

The controller and plant model networks that you develop are stored within Simulink controller blocks. At some point you might want to transfer the networks into other applications, or you might want to transfer a network from one controller block to another. You can do this by using the **Import Network** and **Export Network** menu options. The following example leads you through the export and import processes. (The NARMA-L2 window is used for this example, but the same procedure applies to all the controllers.)

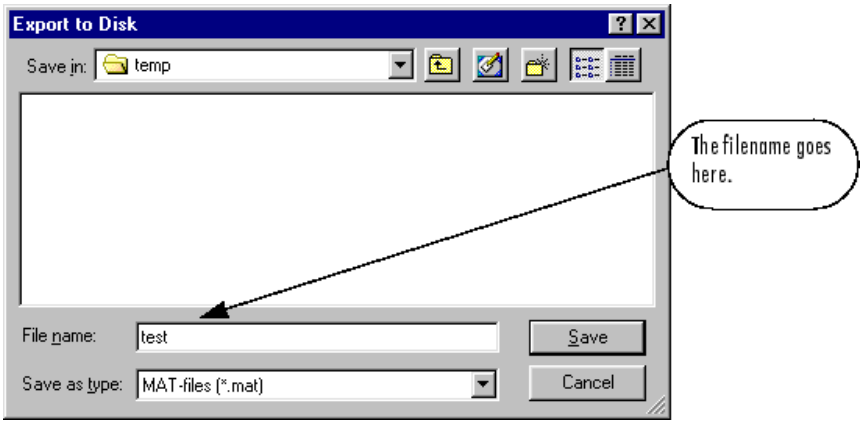
1. Repeat the first three steps of the NARMA-L2 example in “Use the NARMA-L2 Controller Block” on page 24-15. The NARMA-L2 Plant Identification window should now be open.
2. Select **File > Export Network**, as shown below.



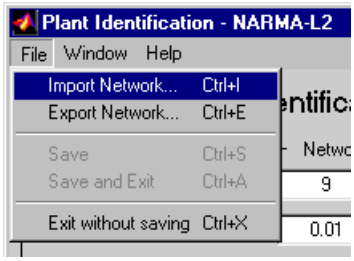
This opens the following window.



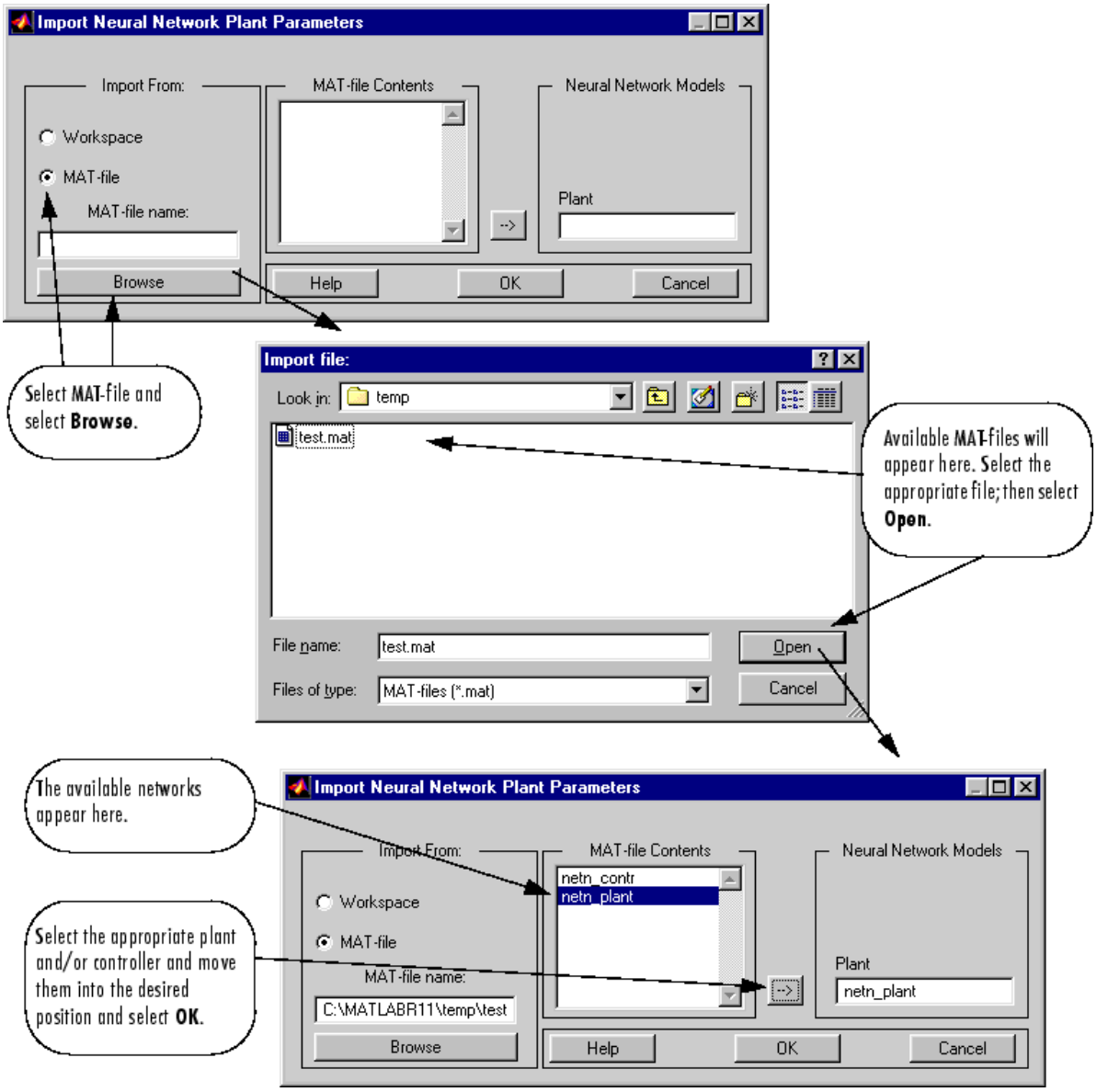
3. Select **Export to Disk**. The following window opens. Enter the file name `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



4 Retrieve that data with the **Import** menu option. Select **File > Import Network**, as in the following figure.



This causes the following window to appear. Follow the steps indicated to retrieve the data that you previously exported. Once the data is retrieved, you can load it into the controller block by clicking **OK** or **Apply**. Notice that the window only has an entry for the plant model, even though you saved both the plant model and the controller. This is because the NARMA-L2 controller is derived directly from the plant model, so you do not need to import both networks.

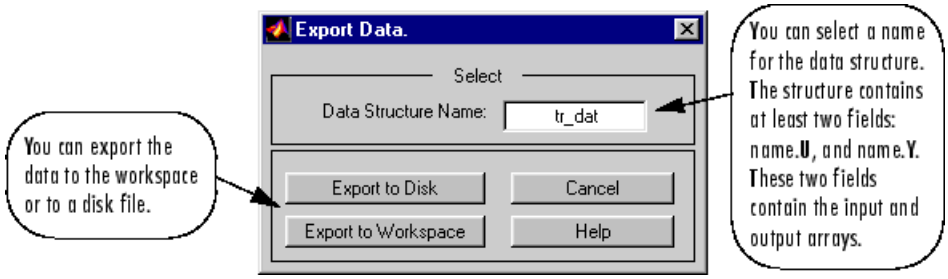


Import and Export Training Data

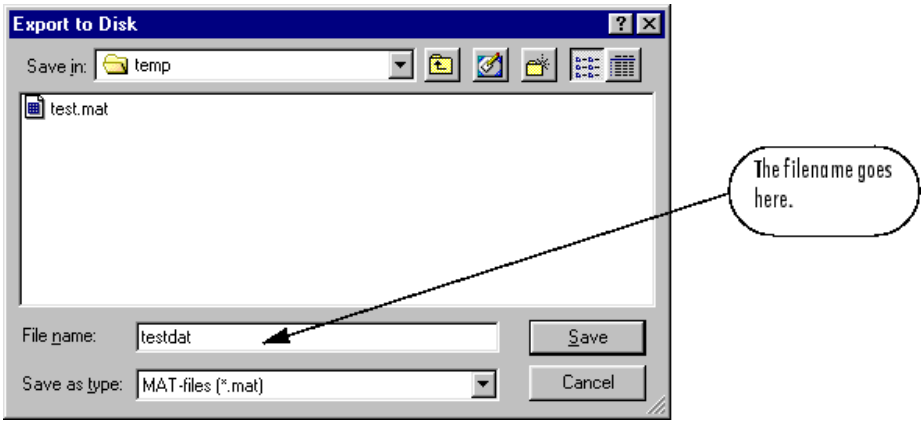
The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You might want to save the training data to the workspace or to a disk file so that you can load it again at a later time. You might also want to combine data sets manually and then load them back into the training window. You can do this by using the **Import** and **Export** buttons. The following example leads you through the import and export processes. (The NN Predictive Control window is used for this example, but the same procedure applies to all the controllers.)

1 Repeat the first five steps of the NN Predictive Control example in “Use the Neural Network Predictive Controller Block” on page 24-6. Then select **Accept Data**. The Plant Identification window should then be open, and the **Import** and **Export** buttons should be active.

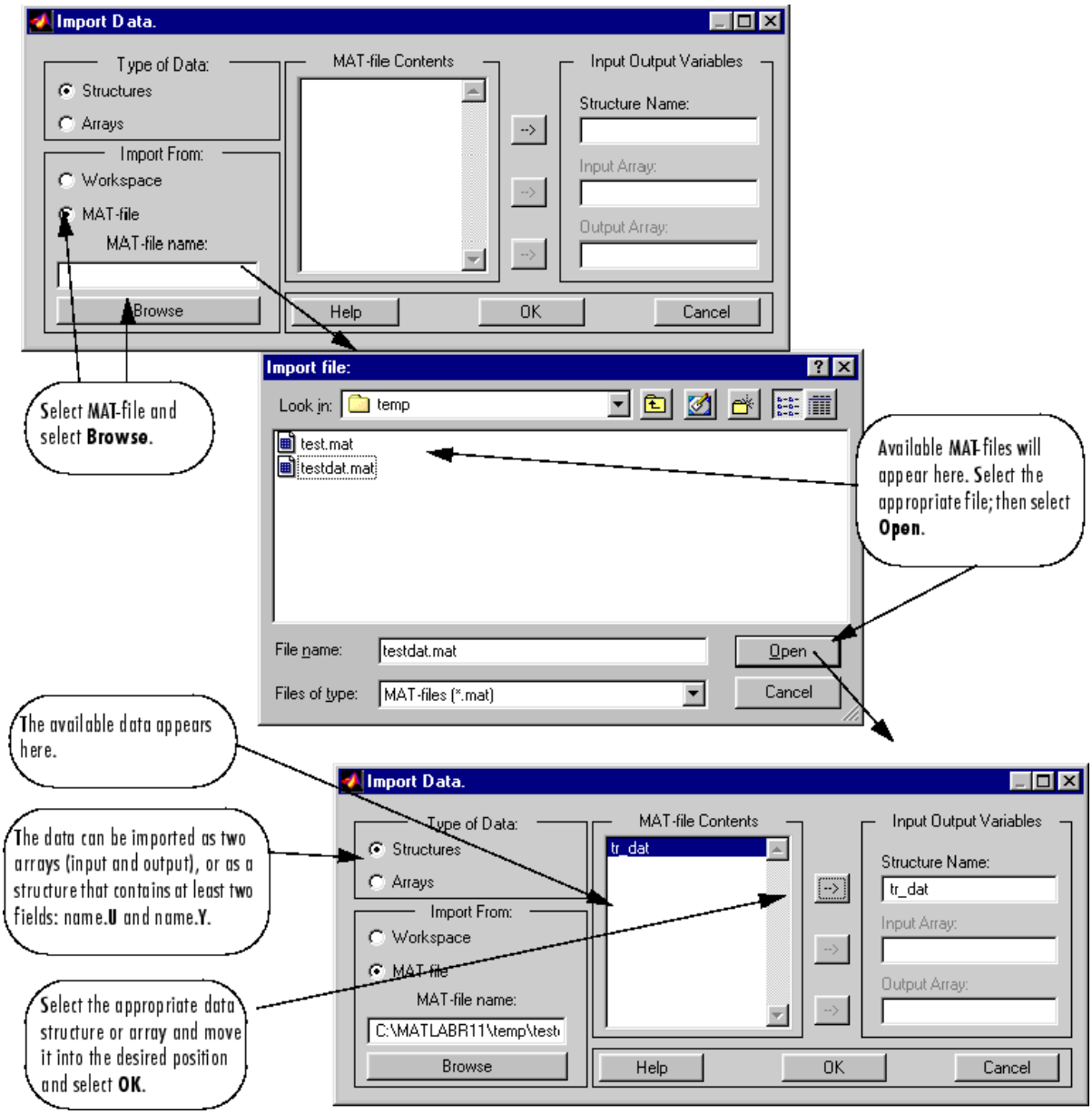
2 Click **Export** to open the following window.



3 Click **Export to Disk**. The following window opens. Enter the filename `testdat` in the box, and select **Save**. This saves the training data structure to disk.



4 Now retrieve the data with the import command. Click **Import** in the Plant Identification window to open the following window. Follow the steps indicated on the following page to retrieve the data that you previously exported. Once the data is imported, you can train the neural network plant model.


Import Data.
Type of Data:
Structures
Arrays
Import From:
Workspace
MAT-file
MAT-file name:
Browse
MAT-file Contents
Input Output Variables
Structure Name:
Input Array:
Output Array:
Help
OK
Cancel
Select MAT-file and select Browse.
Import file:
Look in:
temp
test.mat
testdat.mat
File name:
testdat.mat
Files of type:
MAT-files (*.mat)
Open
Cancel
Available MAT-files will appear here. Select the appropriate file; then select Open.
The available data appears here.
The data can be imported as two arrays (input and output), or as a structure that contains at least two fields: name.U and name.Y.
Select the appropriate data structure or array and move it into the desired position and select OK.
Import Data.
Type of Data:
Structures
Arrays
Import From:
Workspace
MAT-file
MAT-file name:
C:\MATLABR11\temp\test
Browse
MAT-file Contents
tr_dat
Input Output Variables
Structure Name:
tr_dat
Input Array:
Output Array:
Help
OK
Cancel

Radial Basis Neural Networks

Introduction to Radial Basis Neural Networks

Radial basis networks can require more neurons than standard feedforward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feedforward networks. They work best when many training vectors are available.

You might want to consult the following paper on this subject: Chen, S., C.F.N. Cowan, and P.M. Grant, “Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks,” *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, March 1991, pp. 302–309.

This topic discusses two variants of radial basis networks, generalized regression networks (GRNN) and probabilistic neural networks (PNN). You can read about them in P.D. Wasserman, *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993, on pp. 155–61 and pp. 35–55, respectively.

Important Radial Basis Functions

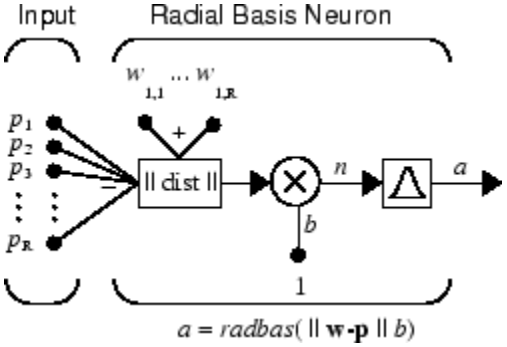
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

Radial Basis Neural Networks

In this section...

Neuron Model

Here is a radial basis network with R inputs.

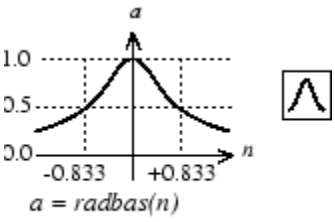


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The `|| dist ||` box in this figure accepts the input vector **p** and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is

$$radbas(n) = e^{-n^2}$$

Here is a plot of the `radbas` transfer function.



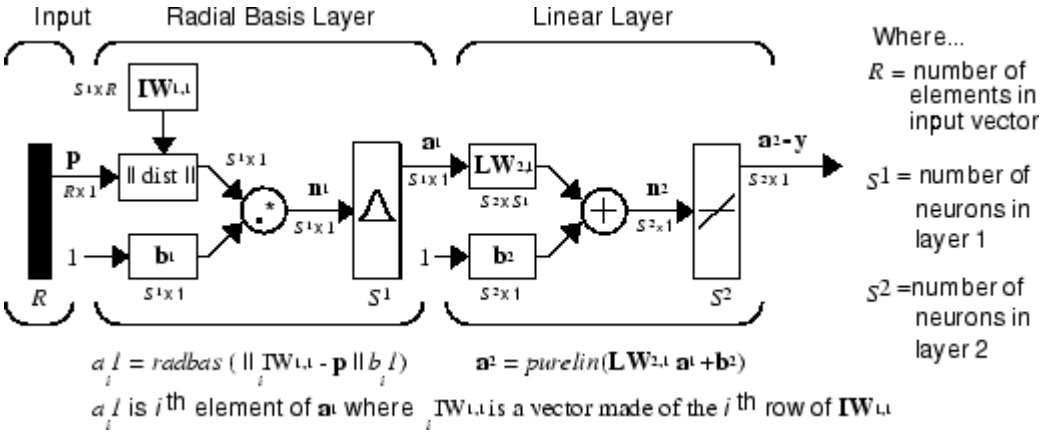
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **w**.

The bias *b* allows the sensitivity of the `radbas` neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector **p** at vector distance of 8.326 (0.8326/*b*) from its weight vector **w**.

Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S^1 neurons, and an output linear layer of S^2 neurons.



The `|| dist ||` box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the distances between the input vector and vectors ${}_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

The bias vector $\mathbf{b}^1$ and the output of `|| dist ||` are combined with the MATLAB operation .* , which does element-by-element multiplication.

The output of the first layer for a feedforward network `net` can be obtained with the following code:

```
a{1} = radbas(netprod(dist(net.IW{1,1},p),net.b{1}))
```

Fortunately, you won't have to write such lines of code. All the details of designing this network are built into design functions `newrbe` and `newrb`, and you can obtain their outputs with `sim`.

You can understand how this network behaves by following an input vector **p** through the network to the output $\mathbf{a}^2$. If you present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1, its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0s (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the

element-by-element product of its weighted input with its bias, calculated with `netprod`. Each neuron's output is its net input passed through `radbas`. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input is `spread`, its net input is sqrt(−log(.5)) (or 0.8326), therefore its output is 0.5.

Exact Design (newrbe)

You can design radial basis networks with the function `newrbe`. This function can produce a network with zero error on training vectors. It is called in the following way:

```
net = newrbe(P,T,SPREAD)
```

The function `newrbe` takes matrices of input vectors `P` and target vectors `T`, and a spread constant `SPREAD` for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly `T` when the inputs are `P`.

This function `newrbe` creates as many `radbas` neurons as there are input vectors in `P`, and sets the first-layer weights to `P'`. Thus, there is a layer of `radbas` neurons in which each neuron acts as a detector for a different input vector. If there are *Q* input vectors, then there will be *Q* neurons.

Each bias in the first layer is set to 0.8326/`SPREAD`. This gives radial basis functions that cross 0.5 at weighted inputs of +/− `SPREAD`. This determines the width of an area in the input space to which each neuron responds. If `SPREAD` is 4, then each `radbas` neuron will respond with 0.5 or more to any input vectors within a vector distance of 4 from their weight vector. `SPREAD` should be large enough that neurons respond strongly to overlapping regions of the input space.

The second-layer weights IW ^{2,1} (or in code, `IW{2,1}`) and biases b^2 (or in code, `b{2}`) are found by simulating the first-layer outputs a^1 (`A{1}`), and then solving the following linear expression:

```
[W{2,1} b{2}] * [A{1}; ones(1,Q)] = T
```

You know the inputs to the second layer (`A{1}`) and the target (`T`), and the layer is linear. You can use the following code to calculate the weights and biases of the second layer to minimize the sum-squared error.

```
Wb = T/[A{1}; ones(1,Q)]
```

Here `Wb` contains both weights and biases, with the biases in the last column. The sum-squared error is always 0, as explained below.

There is a problem with *C* constraints (input/target pairs) and each neuron has *C* +1 variables (the *C* weights from the *C* `radbas` neurons, and a bias). A linear problem with *C* constraints and more than *C* variables has an infinite number of zero error solutions.

Thus, `newrbe` creates a network with zero error on training vectors. The only condition required is to make sure that `SPREAD` is large enough that the active input regions of the `radbas` neurons overlap enough so that several `radbas` neurons always have fairly large outputs at any given moment. This makes the network function smoother and results in better generalization for new input vectors occurring between input vectors used in the design. (However, `SPREAD` should not be so large that each neuron is effectively responding in the same large area of the input space.)

The drawback to `newrbe` is that it produces a network with as many hidden neurons as there are input vectors. For this reason, `newrbe` does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

More Efficient Design (newrb)

The function `newrb` iteratively creates a radial basis network one neuron at a time. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is

```
net = newrb(P,T,GOAL,SPREAD)
```

The function `newrb` takes matrices of input and target vectors `P` and `T`, and design parameters `GOAL` and `SPREAD`, and returns the desired network.

The design method of `newrb` is similar to that of `newrbe`. The difference is that `newrb` creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most is used to create a `radbas` neuron. The error of the new network is checked, and if low enough `newrb` is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met or the maximum number of neurons is reached.

As with `newrbe`, it is important that the spread parameter be large enough that the `radbas` neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feedforward network? Radial basis networks, even when designed efficiently with `newrbe`, tend to have many times more neurons than a comparable feedforward network with `tansig` or `logsig` neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while `radbas` neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more `radbas` neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons' being used, as can be seen in the next example.

Examples

The example “Radial Basis Approximation” on page 31-114 shows how a radial basis network is used to fit a function. Here the problem is solved with only five neurons.

Examples “Radial Basis Underlapping Neurons” on page 31-118 and “Radial Basis Overlapping Neurons” on page 31-120 examine how the spread constant affects the design process for radial basis networks.

In “Radial Basis Underlapping Neurons” on page 31-118, a radial basis network is designed to solve the same problem as in “Radial Basis Approximation” on page 31-114. However, this time the spread constant used is 0.01. Thus, each radial basis neuron returns 0.5 or lower for any input vector with a distance of 0.01 or more from its weight vector.

Because the training inputs occur at intervals of 0.1, no two radial basis neurons have a strong output for any given input.

“Radial Basis Underlapping Neurons” on page 31-118 showed that having too small a spread constant can result in a solution that does not generalize from the input/target vectors used in the design. Example “Radial Basis Overlapping Neurons” on page 31-120 shows the opposite problem. If the

spread constant is large enough, the radial basis neurons will output large values (near 1.0) for all the inputs used to design the network.

If all the radial basis neurons always output 1, any information presented to the network becomes lost. No matter what the input, the second layer outputs 1's. The function `newrb` will attempt to find a network, but cannot because of numerical problems that arise in this situation.

The moral of the story is, choose a spread constant larger than the distance between adjacent input vectors, so as to get good generalization, but smaller than the distance across the whole input space.

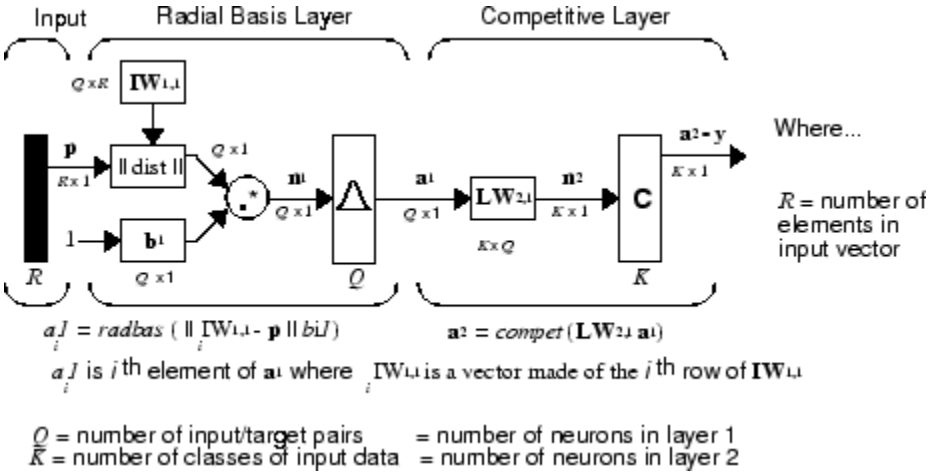
For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the leftmost and rightmost inputs.

Probabilistic Neural Networks

In this section...

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these elements is 1 and the rest are 0. Thus, each input vector is associated with one of K classes.

The first-layer input weights, $IW^{1,1}$ (`net.IW{1,1}`), are set to the transpose of the matrix formed from the Q training pairs, **P'**. When an input is presented, the `|| dist ||` box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent to the `radbas` transfer function. An input vector close to a training vector is represented by a number close to 1 in the output vector $\mathbf{a}^1$. If an input is close to several training vectors of a single class, it is represented by several elements of $\mathbf{a}^1$ that are close to 1.

The second-layer weights, $LW^{1,2}$ (`net.LW{2,1}`), are set to the matrix **T** of target vectors. Each vector has a 1 only in the row associated with that particular class of input, and 0s elsewhere. (Use function `ind2vec` to create the proper vectors.) The multiplication $\mathbf{Ta}^1$ sums the elements of $\mathbf{a}^1$ due to each of the K input classes. Finally, the second-layer transfer function, `compet`, produces a 1 corresponding to the largest element of $\mathbf{n}^2$, and 0s elsewhere. Thus, the network classifies the input vector into a specific K class because that class has the maximum probability of being correct.

Design (newpnn)

You can use the function newpnn to create a PNN. For instance, suppose that seven input vectors and their corresponding targets are

```
P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]'
```

which yields

```
P =
     0     1     0     1     3     4     4
     0     1     3     4     1     1     3
Tc = [1 1 2 2 3 3 3]
```

which yields

```
Tc =
     1     1     2     2     3     3     3
```

You need a target matrix with 1s in the right places. You can get it with the function ind2vec. It gives a matrix with 0s except at the correct spots. So execute

```
T = ind2vec(Tc)
```

which gives

```
T =
   (1,1)        1
   (1,2)        1
   (2,3)        1
   (2,4)        1
   (3,5)        1
   (3,6)        1
   (3,7)        1
```

Now you can create a network and simulate it, using the input P to make sure that it does produce the correct classifications. Use the function vec2ind to convert the output Y into a row Yc to make the classifications clear.

```
net = newpnn(P,T);
Y = sim(net,P);
Yc = vec2ind(Y)
```

This produces

```
Yc =
     1     1     2     2     3     3     3
```

You might try classifying vectors other than those that were used to design the network. Try to classify the vectors shown below in P2.

```
P2 = [1 4;0 1;5 2]'
P2 =
     1     0     5
     4     1     2
```

Can you guess how these vectors will be classified? If you run the simulation and plot the vectors as before, you get

```
Yc =
     2     1     3
```

These results look good, for these test vectors were quite close to members of classes 2, 1, and 3, respectively. The network has managed to generalize its operation to properly classify vectors other than those used to design the network.

You might want to try “PNN Classification” on page 31-126. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

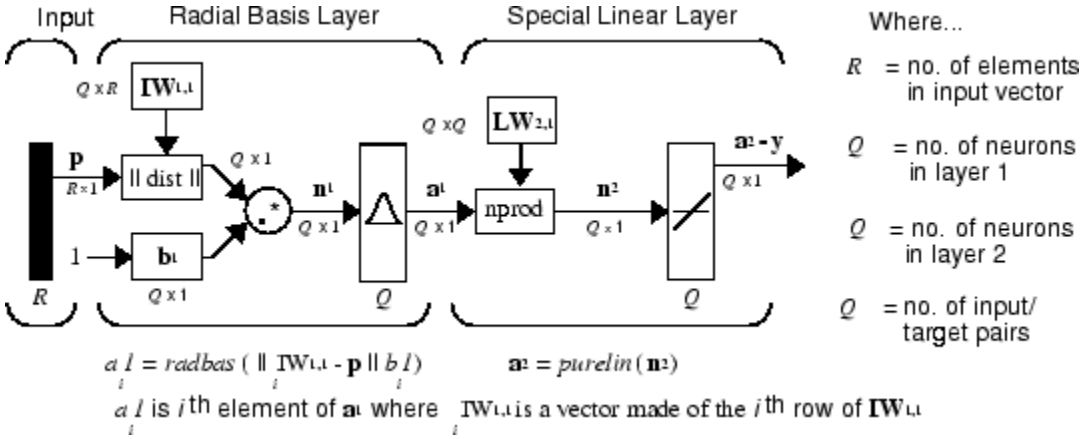
Generalized Regression Neural Networks

In this section...

Network Architecture

A generalized regression neural network (GRNN) is often used for function approximation. It has a radial basis layer and a special linear layer.

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



Here the **nprod** box shown above (code function `normprod`) produces S^2 elements in vector $\mathbf{n}^2$. Each element is the dot product of a row of $LW^{2,1}$ and the input vector $\mathbf{a}^1$, all normalized by the sum of the elements of $\mathbf{a}^1$. For instance, suppose that

```
LW{2,1}= [1 -2;3 4;5 6];
a{1} = [0.7;0.3];
```

Then

```
aout = normprod(LW{2,1},a{1})
aout =
    0.1000
    3.3000
    5.3000
```

The first layer is just like that for `newrbe` networks. It has as many neurons as there are input/ target vectors in **P**. Specifically, the first-layer weights are set to **P'**. The bias $\mathbf{b}^1$ is set to a column vector of 0.8326/SPREAD. The user chooses SPREAD, the distance an input vector must be from a neuron's weight vector to be 0.5.

Again, the first layer operates just like the `newrbe` radial basis layer described previously. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with `dist`. Each neuron's net input is the product of its weighted input with its bias, calculated with `netprod`. Each neuron's output is its net input passed through `radbas`. If a neuron's weight vector is

equal to the input vector (transposed), its weighted input will be 0, its net input will be 0, and its output will be 1. If a neuron's weight vector is a distance of `spread` from the input vector, its weighted input will be `spread`, and its net input will be sqrt(−log(.5)) (or 0.8326). Therefore its output will be 0.5.

The second layer also has as many neurons as input/target vectors, but here `LW{2,1}` is set to `T`.

Suppose you have an input vector **p** close to $\mathbf{p}_i$, one of the input vectors among the input vector/target pairs used in designing layer 1 weights. This input **p** produces a layer 1 $\mathbf{a}^i$ output close to 1. This leads to a layer 2 output close to $\mathbf{t}_i$, one of the targets used to form layer 2 weights.

A larger `spread` leads to a large area around the input vector where layer 1 neurons will respond with significant outputs. Therefore if `spread` is small the radial basis function is very steep, so that the neuron with the weight vector closest to the input will have a much larger output than other neurons. The network tends to respond with the target vector associated with the nearest design input vector.

As `spread` becomes larger the radial basis function's slope becomes smoother and several neurons can respond to an input vector. The network then acts as if it is taking a weighted average between target vectors whose design input vectors are closest to the new input vector. As `spread` becomes larger more and more neurons contribute to the average, with the result that the network function becomes smoother.

Design (newgrnn)

You can use the function `newgrnn` to create a GRNN. For instance, suppose that three input and three target vectors are defined as

```
P = [4 5 6];
T = [1.5 3.6 6.7];
```

You can now obtain a GRNN with

```
net = newgrnn(P,T);
```

and simulate it with

```
P = 4.5;
v = sim(net,P);
```

You might want to try “GRNN Function Approximation” on page 31-122 as well.

Function	Description
`compet`	Competitive transfer function.
`dist`	Euclidean distance weight function.
`dotprod`	Dot product weight function.
`ind2vec`	Convert indices to vectors.
`negdist`	Negative Euclidean distance weight function.
`netprod`	Product net input function.
`newgrnn`	Design a generalized regression neural network.
`newpnn`	Design a probabilistic neural network.

Function	Description
`newrb`	Design a radial basis network.
`newrbe`	Design an exact radial basis network.
`normprod`	Normalized dot product weight function.
`radbas`	Radial basis transfer function.
`vec2ind`	Convert vectors to indices.

Self-Organizing and Learning Vector Quantization Networks

Introduction to Self-Organizing and LVQ

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors. Self-organizing maps do not have target vectors, since their purpose is to divide the input vectors into clusters of similar vectors. There is no desired output for these types of networks.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner (with target outputs). A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

LVQ networks, on the other hand, learn to classify input vectors into target classes chosen by the user.

You might consult the following reference: Kohonen, T., *Self-Organization and Associative Memory, 2nd Edition*, Berlin: Springer-Verlag, 1987.

Important Self-Organizing and LVQ Functions

You can create competitive layers and self-organizing maps with `competlayer` and `selforgmap`, respectively.

You can create an LVQ network with the function `lvqnet`.

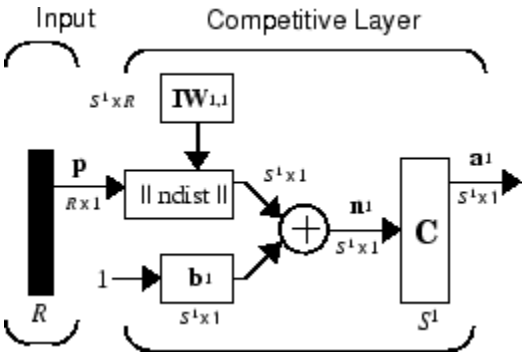
Cluster with a Competitive Neural Network

In this section...

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

Architecture

The architecture for a competitive network is shown below.



The ‖ `dist` ‖ box in this figure accepts the input vector **p** and the input weight matrix $\mathbf{IW}^{1,1}$, and produces a vector having S_1 elements. The elements are the negative of the distances between the input vector and vectors $_i\mathbf{IW}^{1,1}$ formed from the rows of the input weight matrix.

Compute the net input $\mathbf{n}^1$ of a competitive layer by finding the negative distance between input vector **p** and the weight vectors and adding the biases **b**. If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the *winner*, the neuron associated with the most positive element of net input $\mathbf{n}^1$. The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the *least* negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in “Bias Learning Rule (learncon)” on page 26-5.

Create a Competitive Neural Network

You can create a competitive neural network with the function `competlayer`. A simple example shows how this works.

Suppose you want to divide the following four two-element vectors into two classes.

```
p = [.1 .8 .1 .9; .2 .9 .1 .8]

p =
    0.1000    0.8000    0.1000    0.9000
    0.2000    0.9000    0.1000    0.8000
```

There are two vectors near the origin and two vectors near (1,1).

First, create a two-neuron competitive layer.:

```
net = competlayer(2);
```

Now you have a network, but you need to train it to do the classification job.

The first time the network is trained, its weights will initialized to the centers of the input ranges with the function `midpoint`. You can check see these initial values using the number of neurons and the input data:

```
wts = midpoint(2,p)

wts =
    0.5000    0.5000
    0.5000    0.5000
```

These weights are indeed the values at the midpoint of the range (0 to 1) of the inputs.

The initial biases are computed by `initcon`, which gives

```
biases = initcon(2)

biases =
    5.4366
    5.4366
```

Recall that each neuron competes to respond to an input vector **p**. If the biases are all 0, the neuron whose weight vector is closest to **p** gets the highest net input and, therefore, wins the competition, and outputs 1. All other neurons output 0. You want to adjust the winning neuron so as to move it closer to the input. A learning rule to do this is discussed in the next section.

Kohonen Learning Rule (learnk)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the *Kohonen learning* rule. Supposing that the *i*th neuron wins, the elements of the *i*th row of the input weight matrix are adjusted as shown below.

$$ {}_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1)) $$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon

adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

The function `learnk` is used to perform the Kohonen learning rule in this toolbox.

Bias Learning Rule (learncon)

One of the limitations of competitive networks is that some neurons might not always be *allocated*. In other words, some neuron weight vectors might start out far from any input vectors and never win the competition, no matter how long the training is continued. The result is that their weights do not get to learn and they never win. These unfortunate neurons, referred to as *dead neurons*, never perform a useful function.

To stop this, use biases to give neurons that only win the competition rarely (if ever) an advantage over neurons that win often. A positive bias, added to the negative distance, makes a distant neuron more likely to win.

To do this job a running average of neuron outputs is kept. It is equivalent to the percentages of times each output is 1. This average is used to update the biases with the learning function `learncon` so that the biases of frequently active neurons become smaller, and biases of infrequently active neurons become larger.

As the biases of infrequently active neurons increase, the input space to which those neurons respond increases. As that input space increases, the infrequently active neuron responds and moves toward more input vectors. Eventually, the neuron responds to the same number of vectors as other neurons.

This has two good effects. First, if a neuron never wins a competition because its weights are far from any of the input vectors, its bias eventually becomes large enough so that it can win. When this happens, it moves toward some group of input vectors. Once the neuron's weights have moved into a group of input vectors and the neuron is winning consistently, its bias will decrease to 0. Thus, the problem of dead neurons is resolved.

The second advantage of biases is that they force each neuron to classify roughly the same percentage of input vectors. Thus, if a region of the input space is associated with a larger number of input vectors than another region, the more densely filled region will attract more neurons and be classified into smaller subsections.

The learning rates for `learncon` are typically set an order of magnitude or more smaller than for `learnk` to make sure that the running average is accurate.

Training

Now train the network for 500 epochs. You can use either `train` or `adapt`.

```
net.trainParam.epochs = 500;
net = train(net,p);
```

Note that `train` for competitive networks uses the training function `trainru`. You can verify this by executing the following code after creating the network.

```
net.trainFcn
```

```
ans =
trainru
```

For each epoch, all training vectors (or sequences) are each presented once in a different random order with the network and weight and bias values updated after each individual presentation.

Next, supply the original vectors as input to the network, simulate the network, and finally convert its output vectors to class indices.

```
a = sim(net,p);
ac = vec2ind(a)

ac =
     1     2     1     2
```

You see that the network is trained to classify the input vectors into two groups, those near the origin, class 1, and those near (1,1), class 2.

It might be interesting to look at the final weights and biases.

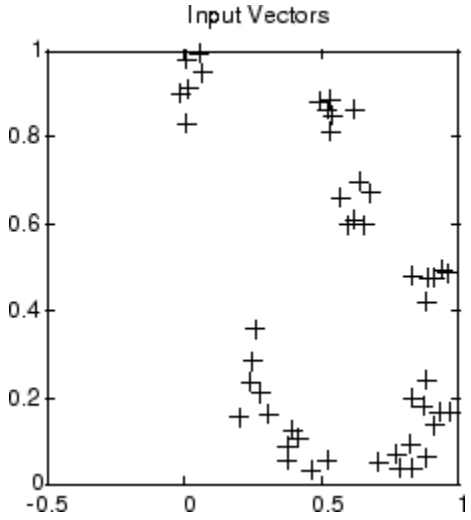
```
net.IW{1,1}

ans =
    0.1000    0.1500
    0.8500    0.8500

net.b{1}

ans =
    5.4367
    5.4365
```

(You might get different answers when you run this problem, because a random seed is used to pick the order of the vectors presented to the network for training.) Note that the first vector (formed from the first row of the weight matrix) is near the input vectors close to the origin, while the vector formed from the second row of the weight matrix is close to the input vectors near (1,1). Thus, the network has been trained—just by exposing it to the inputs—to classify them.

During training each neuron in the layer closest to a group of input vectors adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors has a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input.

Graphical Example

Competitive layers can be understood better when their weight vectors and input vectors are shown graphically. The diagram below shows 48 two-element input vectors represented with + markers.



The input vectors above appear to fall into clusters. You can use a competitive network of eight neurons to classify the vectors into such clusters.

Try “Competitive Learning” on page 31-106 to see a dynamic example of competitive learning.

Cluster with Self-Organizing Map Neural Network

In this section...

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The function `gridtop`, `hextop`, or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. Distances between neurons are calculated from their positions with a distance function. There are four distance functions, `dist`, `boxdist`, `linkdist`, and `mandist`. Link distance is the most common. These topology and distance functions are described in “Topologies (gridtop, hextop, randtop)” on page 26-9 and “Distance Functions (dist, linkdist, mandist, boxdist)” on page 26-12.

Here a self-organizing feature map network identifies a winning neuron i^* using the same procedure as employed by a competitive layer. However, instead of updating only the winning neuron, all neurons within a certain neighborhood $N_{i^*}(d)$ of the winning neuron are updated, using the Kohonen rule. Specifically, all such neurons $i \in N_{i^*}(d)$ are adjusted as follows:

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1))$$

or

$$_i\mathbf{w}(q) = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

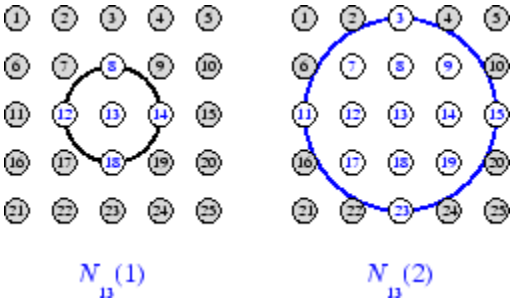
Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*.

$$N_i(d) = \left\{ j, d_{ij} \le d \right\}$$

Thus, when a vector **p** is presented, the weights of the winning neuron *and* its close neighbors move toward **p**. Consequently, after many presentations, neighboring neurons have learned vectors similar to each other.

Another version of SOFM training, called the *batch algorithm*, presents the whole data set to the network before any weights are updated. The algorithm then determines a winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner, or for which it is in the neighborhood of a winner.

To illustrate the concept of neighborhoods, consider the figure below. The left diagram shows a two-dimensional neighborhood of radius $d = 1$ around neuron 13. The right diagram shows a neighborhood of radius $d = 2$.



These neighborhoods could be written as $N_{13}(1) = \{8, 12, 13, 14, 18\}$ and $N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$.

The neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line). You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

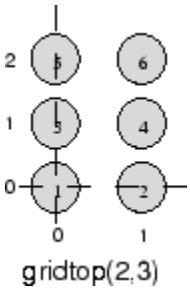
Topologies (gridtop, hextop, randtop)

You can specify different topologies for the original neuron locations with the functions `gridtop`, `hextop`, and `randtop`.

The `gridtop` topology starts with neurons in a rectangular grid similar to that shown in the previous figure. For example, suppose that you want a 2-by-3 array of six neurons. You can get this with

```
pos = gridtop([2, 3])
pos =
     0     1     0     1     0     1
     0     0     1     1     2     2
```

Here neuron 1 has the position (0,0), neuron 2 has the position (1,0), and neuron 3 has the position (0,1), etc.

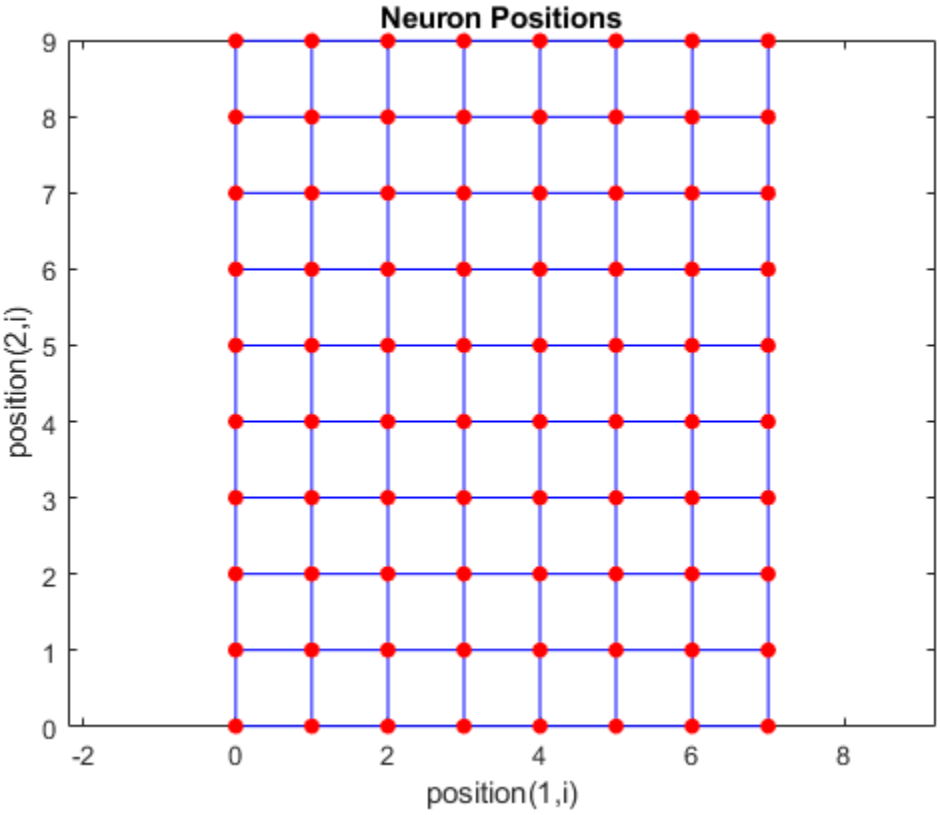


Note that had you asked for a `gridtop` with the dimension sizes reversed, you would have gotten a slightly different arrangement:

```
pos = gridtop([3, 2])
pos =
     0     1     2     0     1     2
     0     0     0     1     1     1
```

You can create an 8-by-10 set of neurons in a `gridtop` topology with the following code:

```
pos = gridtop([8 10]);
plotsom(pos)
```



As shown, the neurons in the `gridtop` topology do indeed lie on a grid.

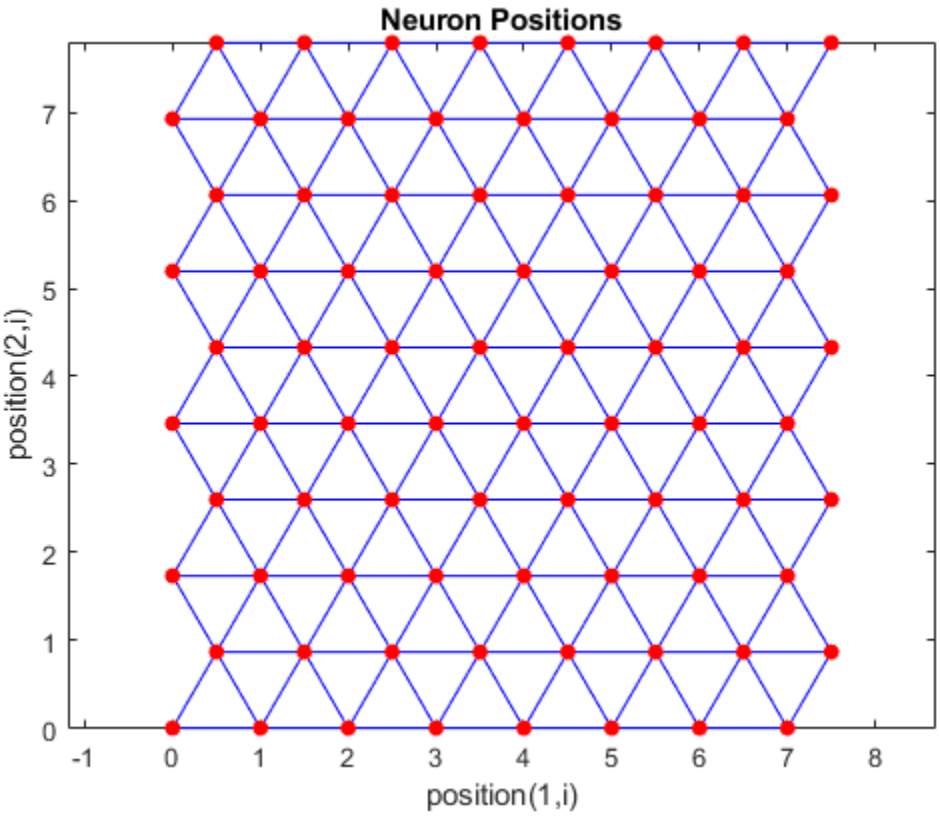
The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
pos = hextop([2, 3])
pos =
         0    1.0000    0.5000    1.5000         0    1.0000
         0         0    0.8660    0.8660    1.7321    1.7321
```

Note that `hextop` is the default pattern for SOM networks generated with `selforgmap`.

You can create and plot an 8-by-10 set of neurons in a `hextop` topology with the following code:

```
pos = hextop([8 10]);
plotsom(pos)
```



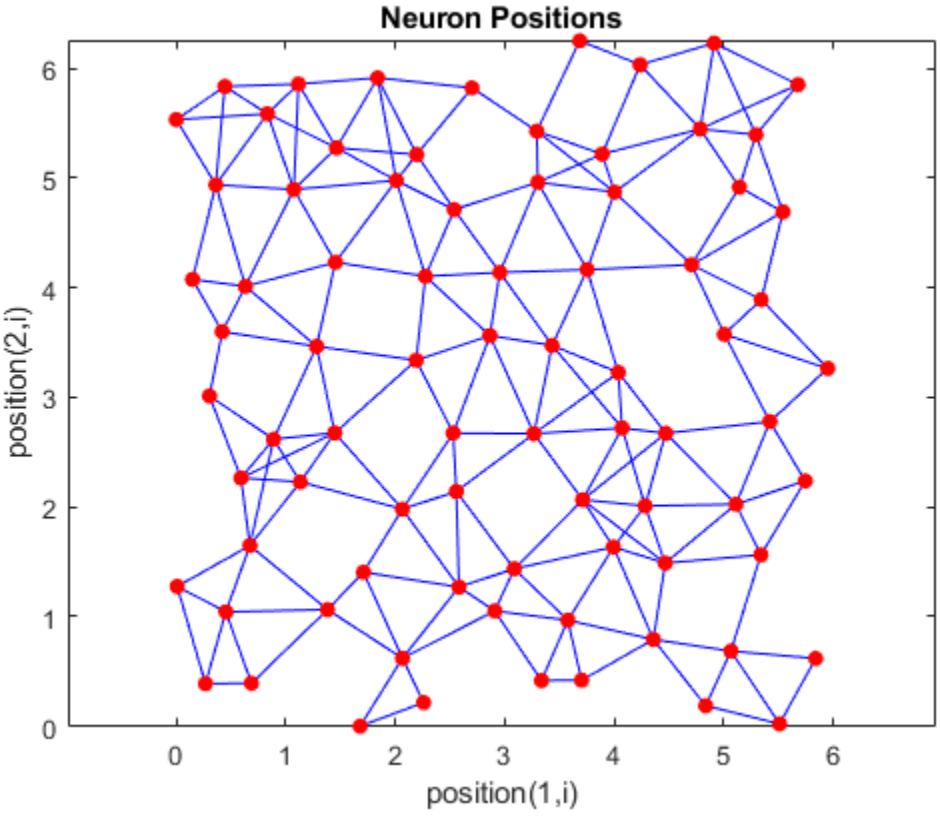
Note the positions of the neurons in a hexagonal arrangement.

Finally, the `randtop` function creates neurons in an N-dimensional random pattern. The following code generates a random pattern of neurons.

```
pos = randtop([2, 3])
pos =
         0    0.7620    0.6268    1.4218    0.0663    0.7862
    0.0925         0    0.4984    0.6007    1.1222    1.4228
```

You can create and plot an 8-by-10 set of neurons in a `randtop` topology with the following code:

```
pos = randtop([8 10]);
plotsom(pos)
```



For examples, see the help for these topology functions.

Distance Functions (dist, linkdist, mandist, boxdist)

In this toolbox, there are four ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

The `dist` function calculates the Euclidean distances from a *home* neuron to other neurons. Suppose you have three neurons:

```
pos2 = [0 1 2; 0 1 2]
pos2 =
     0     1     2
     0     1     2
```

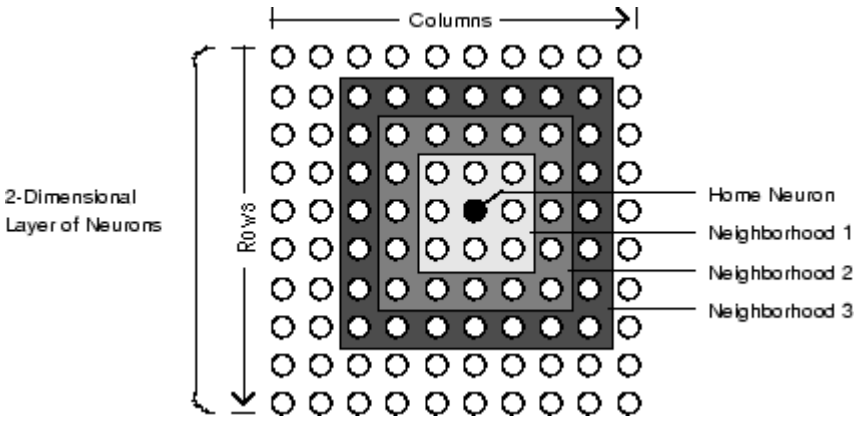
You find the distance from each neuron to the other with

```
D2 = dist(pos2)
D2 =
         0    1.4142    2.8284
    1.4142         0    1.4142
    2.8284    1.4142         0
```

Thus, the distance from neuron 1 to itself is 0, the distance from neuron 1 to neuron 2 is 1.4142, etc.

The graph below shows a home neuron in a two-dimensional (`gridtop`) layer of neurons. The home neuron has neighborhoods of increasing diameter surrounding it. A neighborhood of diameter 1

includes the home neuron and its immediate neighbors. The neighborhood of diameter 2 includes the diameter 1 neurons and their immediate neighbors.



As for the `dist` function, all the neighborhoods for an S-neuron layer map are represented by an *S*-by-*S* matrix of distances. The particular distances shown above (1 in the immediate neighborhood, 2 in neighborhood 2, etc.), are generated by the function `boxdist`. Suppose that you have six neurons in a `gridtop` configuration.

```
pos = gridtop([2, 3])
pos =
     0     1     0     1     0     1
     0     0     1     1     2     2
```

Then the box distances are

```
d = boxdist(pos)
d =
     0     1     1     1     2     2
     1     0     1     1     2     2
     1     1     0     1     1     1
     1     1     1     0     1     1
     2     2     1     1     0     1
     2     2     1     1     1     0
```

The distance from neuron 1 to 2, 3, and 4 is just 1, for they are in the immediate neighborhood. The distance from neuron 1 to both 5 and 6 is 2. The distance from both 3 and 4 to all other neurons is just 1.

The *link distance* from one neuron is just the number of links, or steps, that must be taken to get to the neuron under consideration. Thus, if you calculate the distances from the same set of neurons with `linkdist`, you get

```
dlink =
     0     1     1     2     2     3
     1     0     2     1     3     2
     1     2     0     1     1     2
     2     1     1     0     2     1
     2     3     1     2     0     1
     3     2     2     1     1     0
```

The Manhattan distance between two vectors **x** and **y** is calculated as

```
D = sum(abs(x-y))
```

Thus if you have

```
W1 = [1 2; 3 4; 5 6]
W1 =
     1     2
     3     4
     5     6
```

and

```
P1 = [1;1]
P1 =
     1
     1
```

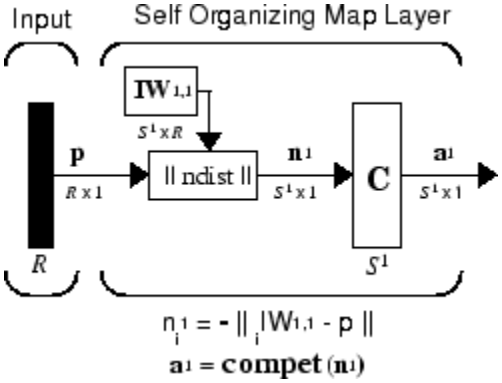
then you get for the distances

```
Z1 = mandist(W1,P1)
Z1 =
     1
     5
     9
```

The distances calculated with `mandist` do indeed follow the mathematical expression given above.

Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element $\mathbf{a}^1_i$ corresponding to i^*, the winning neuron. All other output elements in $\mathbf{a}^1$ are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. You can choose from various topologies of neurons. Similarly, you can choose from various distance expressions to calculate neurons that are close to the winning neuron.

Create a Self-Organizing Map Neural Network (selforgmap)

You can create a new SOM network with the function `selforgmap`. This function defines variables used in two phases of learning:

- Ordering-phase learning rate
- Ordering-phase steps
- Tuning-phase learning rate
- Tuning-phase neighborhood distance

These values are used for training and adapting.

Consider the following example.

Suppose that you want to create a network having input vectors with two elements, and that you want to have six neurons in a hexagonal 2-by-3 network. The code to obtain this network is:

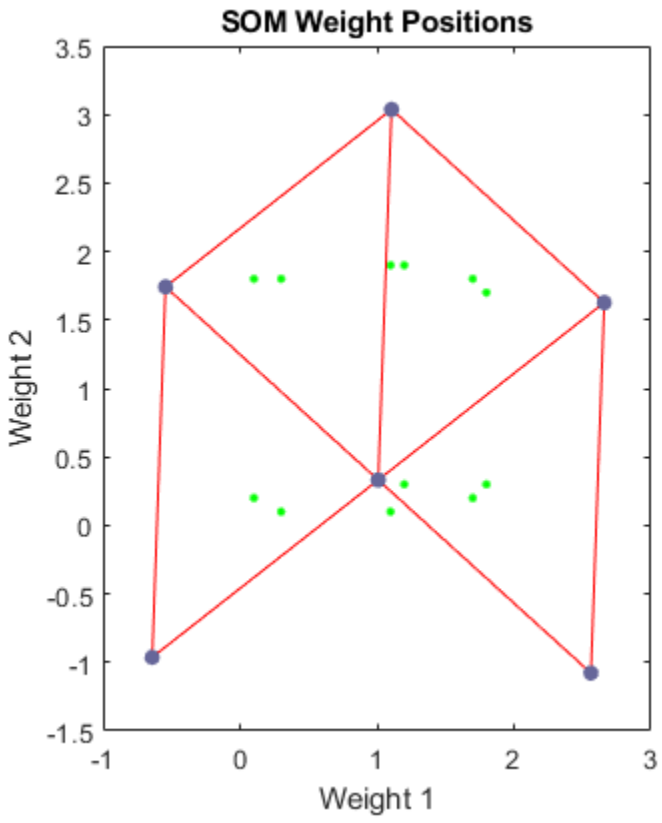
```
net = selforgmap([2 3]);
```

Suppose that the vectors to train on are:

```
P = [.1 .3 1.2 1.1 1.8 1.7 .1 .3 1.2 1.1 1.8 1.7;...
0.2 0.1 0.3 0.1 0.3 0.2 1.8 1.8 1.9 1.9 1.7 1.8];
```

You can configure the network to input the data and plot all of this with:

```
net = configure(net,P);
plotsompos(net,P)
```



The green spots are the training vectors. The initialization for `selforgmap` spreads the initial weights across the input space. Note that they are initially some distance from the training vectors.

When simulating a network, the negative distances between each neuron's weight vector and the input vector are calculated (`negdist`) to get the weighted inputs. The weighted inputs are also the net inputs (`netsum`). The net inputs compete (`compet`) so that only the neuron with the most positive net input will output a 1.

Training (learnsomb)

The default learning in a self-organizing feature map occurs in the batch mode (`trainbu`). The weight learning function for the self-organizing map is `learnsomb`.

First, the network identifies the winning neuron for each input vector. Each weight vector then moves to the average position of all of the input vectors for which it is a winner or for which it is in the neighborhood of a winner. The distance that defines the size of the neighborhood is altered during training through two phases.

Ordering Phase

This phase lasts for the given number of steps. The neighborhood distance starts at a given initial distance, and decreases to the tuning neighborhood distance (1.0). As the neighborhood distance decreases over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered physically.

Tuning Phase

This phase lasts for the rest of training or adaption. The neighborhood size has decreased below 1 so only the winning neuron learns for each sample.

Now take a look at some of the specific values commonly used in these networks.

Learning occurs according to the `learnsomb` learning parameter, shown here with its default value.

Learning Parameter	Default Value	Purpose
`LP.init_neighborhood`	`3`	Initial neighborhood size
`LP.steps`	`100`	Ordering phase steps

The neighborhood size `NS` is altered through two phases: an ordering phase and a tuning phase.

The ordering phase lasts as many steps as `LP.steps`. During this phase, the algorithm adjusts `ND` from the initial neighborhood size `LP.init_neighborhood` down to 1. It is during this phase that neuron weights order themselves in the input space consistent with the associated neuron positions.

During the tuning phase, `ND` is less than 1. During this phase, the weights are expected to spread out relatively evenly over the input space while retaining their topological order found during the ordering phase.

Thus, the neuron's weight vectors initially take large steps all together toward the area of input space where input vectors are occurring. Then as the neighborhood size decreases to 1, the map tends to order itself topologically over the presented input vectors. Once the neighborhood size is 1, the network should be fairly well ordered. The training continues in order to give the neurons time to spread out evenly across the input vectors.

As with competitive layers, the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors appear with even probability throughout

a section of the input space. If input vectors occur with varying frequency throughout the input space, the feature map layer tends to allocate neurons to an area in proportion to the frequency of input vectors there.

Thus, feature maps, while learning to categorize their input, also learn both the topology and distribution of their input.

You can train the network for 1000 epochs with

```
net.trainParam.epochs = 1000;
net = train(net,P);
plotsompos(net,P)
```

You can see that the neurons have started to move toward the various training groups. Additional training is required to get the neurons closer to the various groups.

As noted previously, self-organizing maps differ from conventional competitive learning in terms of which neurons get their weights updated. Instead of updating only the winner, feature maps update the weights of the winner and its neighbors. The result is that neighboring neurons tend to have similar weight vectors and to be responsive to similar input vectors.

Examples

Two examples are described briefly below. You also might try the similar examples “One-Dimensional Self-organizing Map” on page 31-109 and “Two-Dimensional Self-organizing Map” on page 31-111.

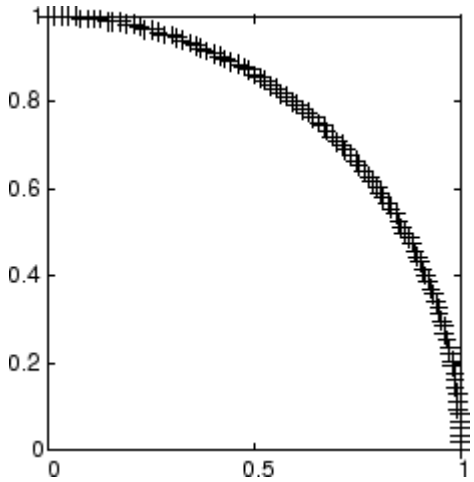
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

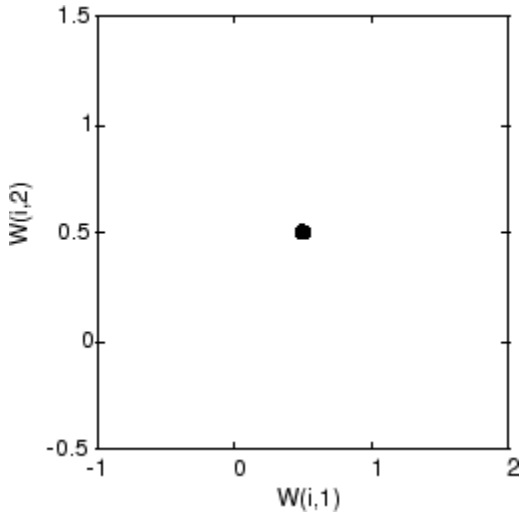
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

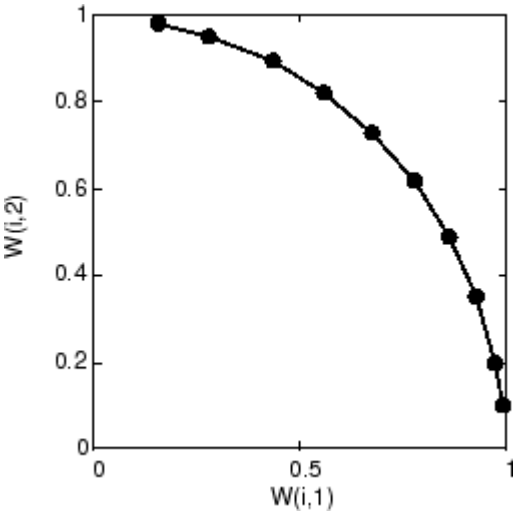


A self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



Of course, because all the weight vectors start in the middle of the input vector space, all you see now is a single circle.

As training starts the weight vectors move together toward the input vectors. They also become ordered as the neighborhood size decreases. Finally the layer adjusts its weights so that each neuron responds strongly to a region of the input space occupied by input vectors. The placement of neighboring neuron weight vectors also reflects the topology of the input vectors.



Note that self-organizing maps are trained with input vectors in a random order, so starting with the same initial vectors does not guarantee identical training results.

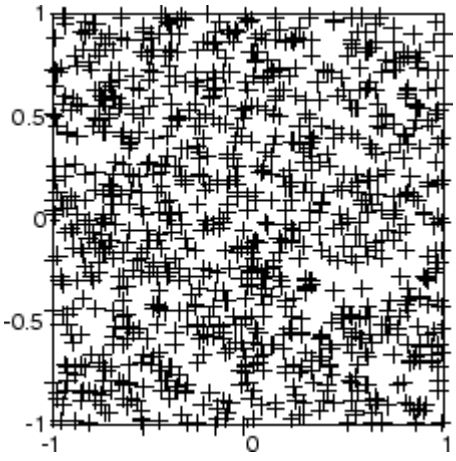
Two-Dimensional Self-Organizing Map

This example shows how a two-dimensional self-organizing map can be trained.

First some random input data is created with the following code:

```
P = rands(2,1000);
```

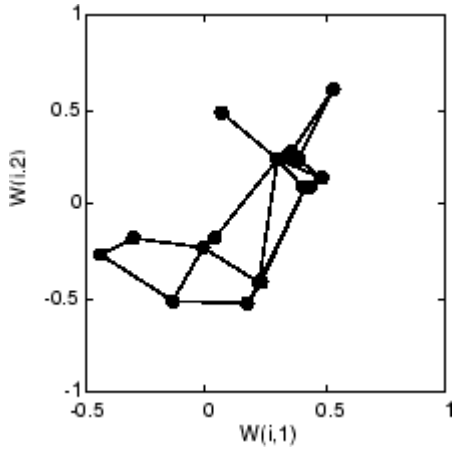
Here is a plot of these 1000 input vectors.



A 5-by-6 two-dimensional map of 30 neurons is used to classify these input vectors. The two-dimensional map is five neurons by six neurons, with distances calculated according to the Manhattan distance neighborhood function `mandist`.

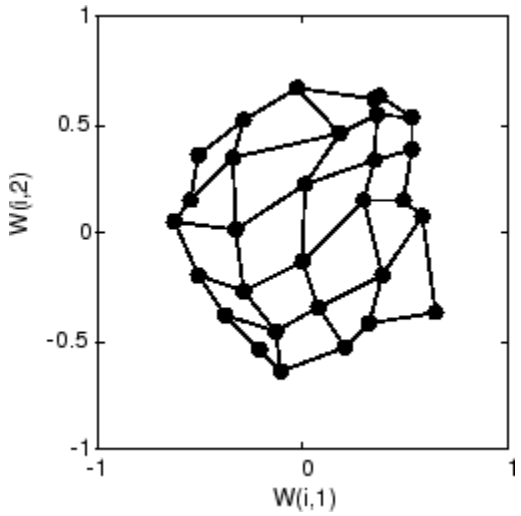
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



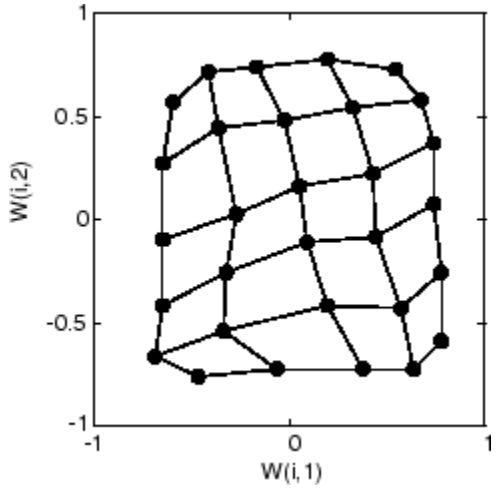
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

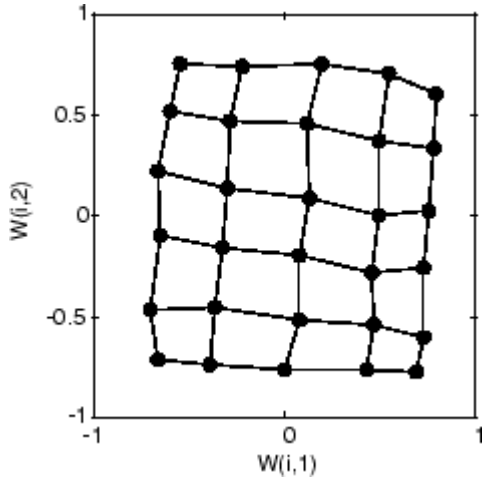


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

The following plot, after 500 cycles, shows the map more evenly distributed across the input space.



Finally, after 5000 cycles, the map is rather evenly spread across the input space. In addition, the neurons are very evenly spaced, reflecting the even distribution of input vectors in this problem.



Thus a two-dimensional self-organizing map has learned the topology of its inputs' space.

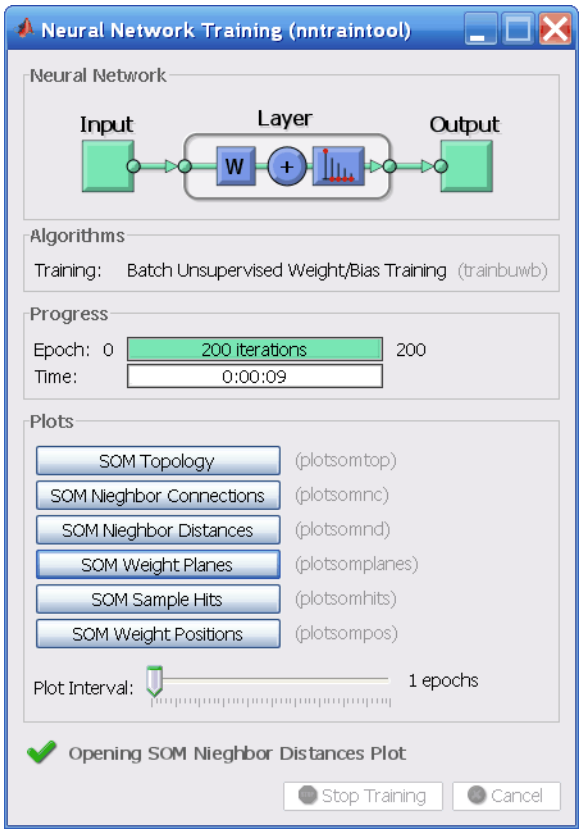
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

Training with the Batch Algorithm

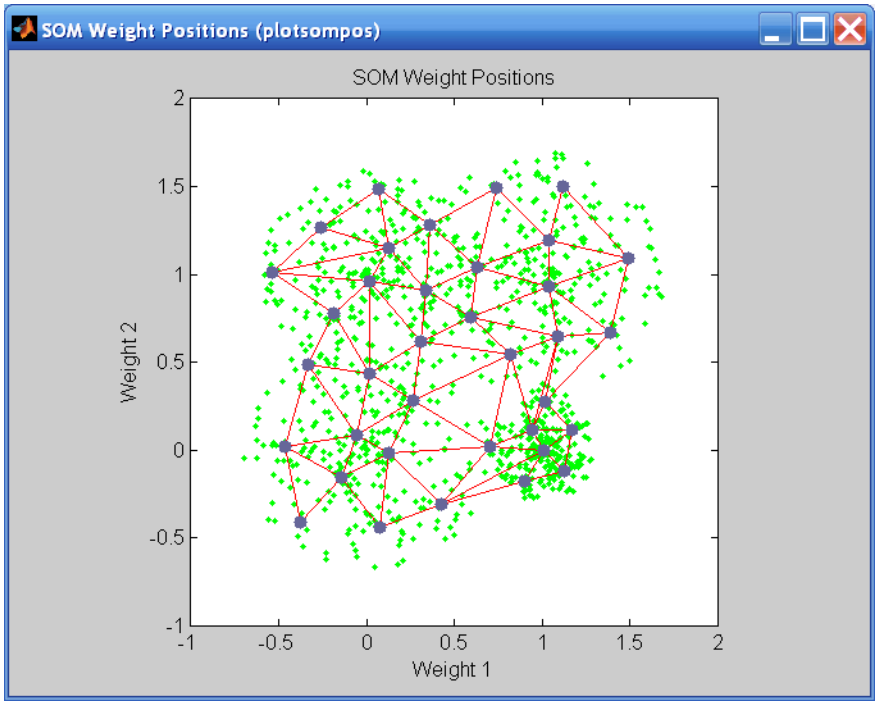
The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
x = simplecluster_dataset
net = selforgmap([6 6]);
net = train(net,x);
```

This command sequence creates and trains a 6-by-6 two-dimensional map of 36 neurons. During training, the following figure appears.



There are several useful visualizations that you can access from this window. If you click **SOM Weight Positions**, the following figure appears, which shows the locations of the data points and the weight vectors. As the figure indicates, after only 200 iterations of the batch algorithm, the map is well distributed through the input space.

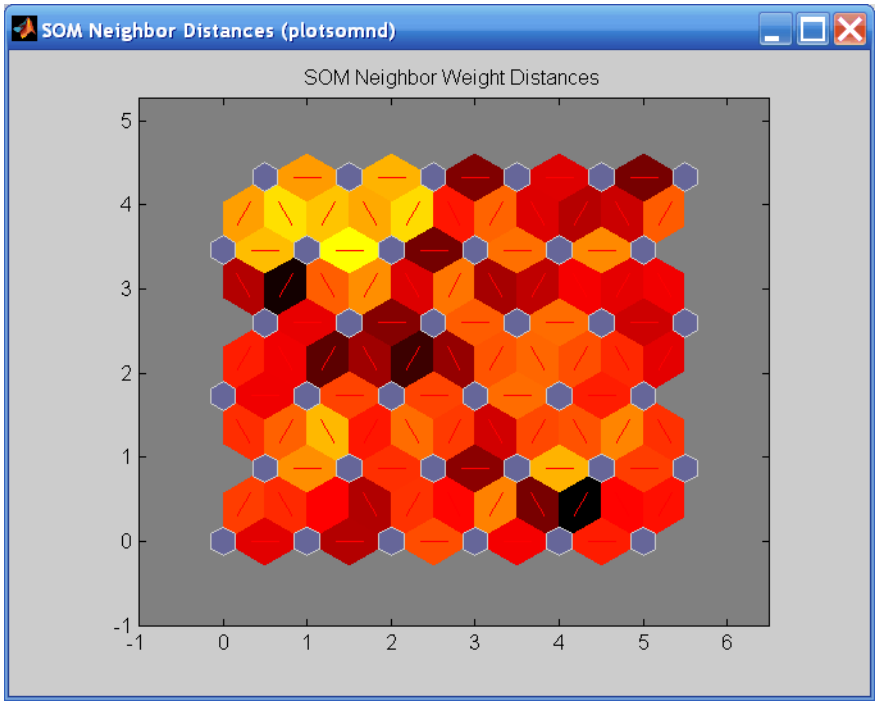


When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

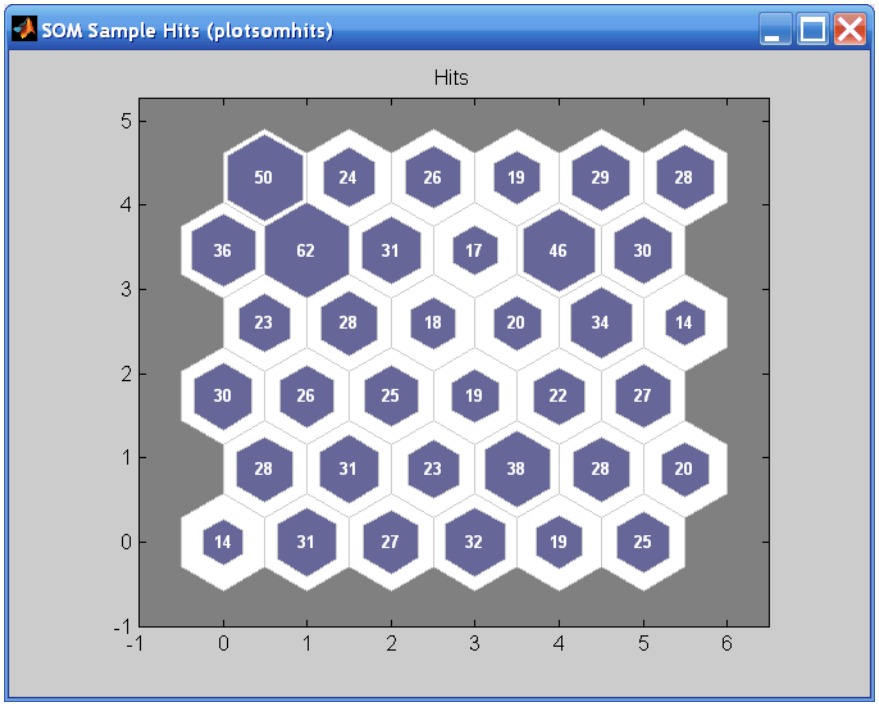
This figure uses the following color coding:

- The blue hexagons represent the neurons.
- The red lines connect neighboring neurons.
- The colors in the regions containing the red lines indicate the distances between neurons.
- The darker colors represent larger distances.
- The lighter colors represent smaller distances.

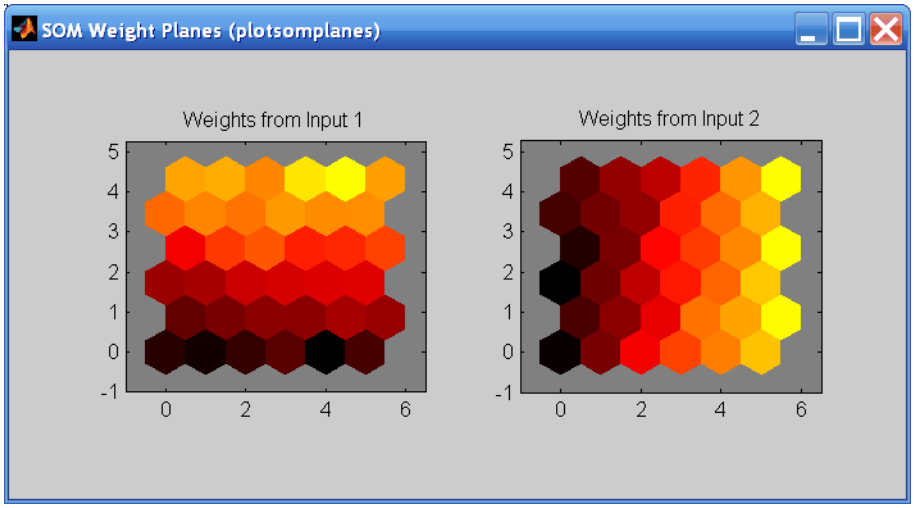
A group of light segments appear in the upper-left region, bounded by some darker segments. This grouping indicates that the network has clustered the data into two groups. These two groups can be seen in the previous weight position figure. The lower-right region of that figure contains a small group of tightly clustered data points. The corresponding weights are closer together in this region, which is indicated by the lighter colors in the neighbor distance figure. Where weights in this small region connect to the larger region, the distances are larger, as indicated by the darker band in the neighbor distance figure. The segments in the lower-right region of the neighbor distance figure are darker than those in the upper left. This color difference indicates that data points in this region are farther apart. This distance is confirmed in the weight positions figure.



Another useful figure can tell you how many data points are associated with each neuron. Click **SOM Sample Hits** to see the following figure. It is best if the data are fairly evenly distributed across the neurons. In this example, the data are concentrated a little more in the upper-left neurons, but overall the distribution is fairly even.



You can also visualize the weights themselves using the weight plane figure. Click **SOM Weight Planes** in the training window to obtain the next figure. There is a weight plane for each element of the input vector (two, in this case). They are visualizations of the weights that connect each input to each of the neurons. (Lighter and darker colors represent larger and smaller weights, respectively.) If the connection patterns of two inputs are very similar, you can assume that the inputs were highly correlated. In this case, input 1 has connections that are very different than those of input 2.



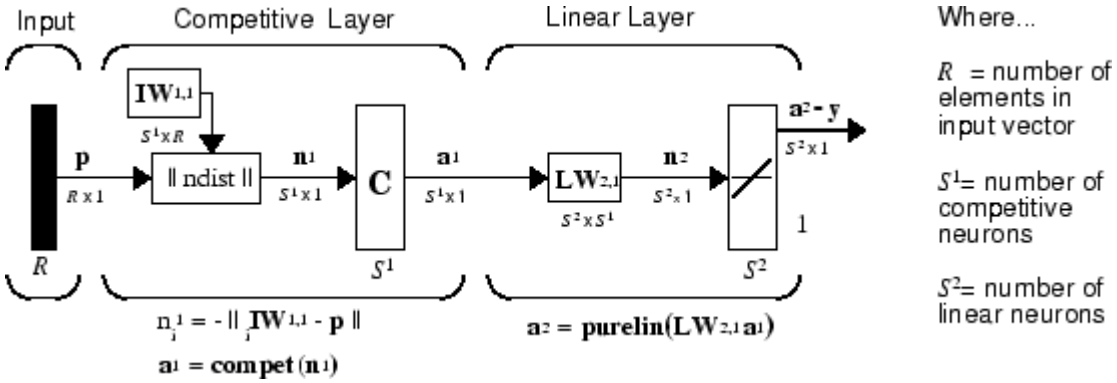
You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`.

Learning Vector Quantization (LVQ) Neural Networks

In this section...

Architecture

The LVQ network architecture is shown below.



An LVQ network has a first competitive layer and a second linear layer. The competitive layer learns to classify input vectors in much the same way as the competitive layers of “Cluster with Self-Organizing Map Neural Network” on page 26-8 described in this topic. The linear layer transforms the competitive layer’s classes into target classifications defined by the user. The classes learned by the competitive layer are referred to as *subclasses* and the classes of the linear layer as *target classes*.

Both the competitive and linear layers have one neuron per (sub or target) class. Thus, the competitive layer can learn up to S^1 subclasses. These, in turn, are combined by the linear layer to form S^2 target classes. (S^1 is always larger than S^2.)

For example, suppose neurons 1, 2, and 3 in the competitive layer all learn subclasses of the input space that belongs to the linear layer target class 2. Then competitive neurons 1, 2, and 3 will have $\mathbf{LW}^{2,1}$ weights of 1.0 to neuron $\mathbf{n}^2$ in the linear layer, and weights of 0 to all other linear neurons. Thus, the linear neuron produces a 1 if any of the three competitive neurons (1, 2, or 3) wins the competition and outputs a 1. This is how the subclasses of the competitive layer are combined into target classes in the linear layer.

In short, a 1 in the ith row of $\mathbf{a}^1$ (the rest to the elements of $\mathbf{a}^1$ will be zero) effectively picks the ith column of $\mathbf{LW}^{2,1}$ as the network output. Each such column contains a single 1, corresponding to a specific class. Thus, subclass 1s from layer 1 are put into various classes by the $\mathbf{LW}^{2,1}\mathbf{a}^1$ multiplication in layer 2.

You know ahead of time what fraction of the layer 1 neurons should be classified into the various class outputs of layer 2, so you can specify the elements of $\mathbf{LW}^{2,1}$ at the start. However, you have to go

through a training procedure to get the first layer to produce the correct subclass output for each vector of the training set. This training is discussed in “Training” on page 26-5. First, consider how to create the original network.

Creating an LVQ Network

You can create an LVQ network with the function `lvqnet`,

```
net = lvqnet(S1,LR,LF)
```

where

- `S1` is the number of first-layer hidden neurons.
- `LR` is the learning rate (default 0.01).
- `LF` is the learning function (default is `learnlv1`).

Suppose you have 10 input vectors. Create a network that assigns each of these input vectors to one of four subclasses. Thus, there are four neurons in the first competitive layer. These subclasses are then assigned to one of two output classes by the two neurons in layer 2. The input vectors and targets are specified by

```
P = [-3 -2 -2 0 0 0 0 2 2 3; 0 1 -1 2 1 -1 -2 1 -1 0];
```

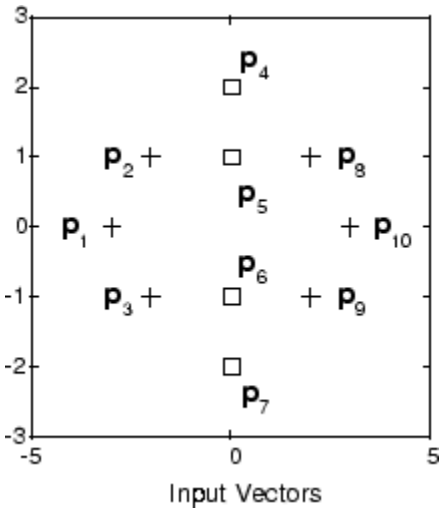
and

```
Tc = [1 1 1 2 2 2 2 1 1 1];
```

It might help to show the details of what you get from these two lines of code.

```
P,Tc
P =
    -3    -2    -2     0     0     0     0     2     2     3
     0     1    -1     2     1    -1    -2     1    -1     0
Tc =
     1     1     1     2     2     2     2     1     1     1
```

A plot of the input vectors follows.



As you can see, there are four subclasses of input vectors. You want a network that classifies $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_8$, $\mathbf{p}_9$, and $\mathbf{p}_{10}$ to produce an output of 1, and that classifies vectors $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$, and $\mathbf{p}_7$ to produce an output of 2. Note that this problem is nonlinearly separable, and so cannot be solved by a perceptron, but an LVQ network has no difficulty.

Next convert the `Tc` matrix to target vectors.

```
T = ind2vec(Tc);
```

This gives a sparse matrix `T` that can be displayed in full with

```
targets = full(T)
```

which gives

```
targets =
     1     1     1     0     0     0     0     1     1     1
     0     0     0     1     1     1     1     0     0     0
```

This looks right. It says, for instance, that if you have the first column of `P` as input, you should get the first column of `targets` as an output; and that output says the input falls in class 1, which is correct. Now you are ready to call `lvqnet`.

Call `lvqnet` to create a network with four neurons.

```
net = lvqnet(4);
```

Configure and confirm the initial values of the first-layer weight matrix are initialized by the function midpoint to values in the center of the input data range.

```
net = configure(net,P,T);
net.IW{1}
ans =
     0     0
     0     0
     0     0
     0     0
```

Confirm that the second-layer weights have 60% (6 of the 10 in `Tc`) of its columns with a 1 in the first row, (corresponding to class 1), and 40% of its columns have a 1 in the second row (corresponding to class 2). With only four columns, the 60% and 40% actually round to 50% and there are two 1's in each row.

```
net.LW{2,1}
ans =
     1     1     0     0
     0     0     1     1
```

This makes sense too. It says that if the competitive layer produces a 1 as the first or second element, the input vector is classified as class 1; otherwise it is a class 2.

You might notice that the first two competitive neurons are connected to the first linear neuron (with weights of 1), while the second two competitive neurons are connected to the second linear neuron. All other weights between the competitive neurons and linear neurons have values of 0. Thus, each of the two target classes (the linear neurons) is, in fact, the union of two subclasses (the competitive neurons).

You can simulate the network with `sim`. Use the original `P` matrix as input just to see what you get.

```
Y = net(P);
Yc = vec2ind(Y)
Yc =
     1     1     1     1     1     1     1     1     1     1
```

The network classifies all inputs into class 1. Because this is not what you want, you have to train the network (adjusting the weights of layer 1 only), before you can expect a good result. The next two sections discuss two LVQ learning rules and the training process.

LVQ1 Learning Rule (learnlv1)

LVQ learning in the competitive layer is based on a set of input/target pairs.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Each target vector has a single 1. The rest of its elements are 0. The 1 tells the proper classification of the associated input. For instance, consider the following training pair.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}\right\}$$

Here there are input vectors of three elements, and each input vector is to be assigned to one of four classes. The network is to be trained so that it classifies the input vector shown above into the third of four classes.

To train the network, an input vector **p** is presented, and the distance from **p** to each row of the input weight matrix $\mathbf{IW}^{1,1}$ is computed with the function `negdist`. The hidden neurons of layer 1 compete. Suppose that the ith element of $\mathbf{n}^1$ is most positive, and neuron i^* wins the competition. Then the competitive transfer function produces a 1 as the i^*th element of $\mathbf{a}^1$. All other elements of $\mathbf{a}^1$ are 0.

When $\mathbf{a}^1$ is multiplied by the layer 2 weights $\mathbf{LW}^{2,1}$, the single 1 in $\mathbf{a}^1$ selects the class k^* associated with the input. Thus, the network has assigned the input vector **p** to class k^* and $\alpha^2_{k^*}$ will be 1. Of course, this assignment can be a good one or a bad one, for t_{k^*} can be 1 or 0, depending on whether the input belonged to class k^* or not.

Adjust the i^*th row of $\mathbf{IW}^{1,1}$ in such a way as to move this row closer to the input vector **p** if the assignment is correct, and to move the row away from **p** if the assignment is incorrect. If **p** is classified correctly,

$$\left(\alpha^2_{k^*} = t_{k^*} = 1\right)$$

compute the new value of the i^*th row of $\mathbf{IW}^{1,1}$ as

$${}_{i^*}\mathbf{IW}^{1,1}(q) = {}_{i^*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{IW}^{1,1}(q-1))$$

On the other hand, if **p** is classified incorrectly,

$$\left(\alpha^2_{k^*} = 1 \neq t_{k^*} = 0\right)$$

compute the new value of the i^*th row of $\mathbf{IW}^{1,1}$ as

$$_{i*}\mathbf{IW}^{1,1}(q) = {}_{i*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i*}\mathbf{IW}^{1,1}(q-1))$$

You can make these corrections to the i*th row of $\mathbf{IW}^{1,1}$ automatically, without affecting other rows of $\mathbf{IW}^{1,1}$, by back-propagating the output errors to layer 1.

Such corrections move the hidden neuron toward vectors that fall into the class for which it forms a subclass, and away from vectors that fall into other classes.

The learning function that implements these changes in the layer 1 weights in LVQ networks is `learnlv1`. It can be applied during training.

Training

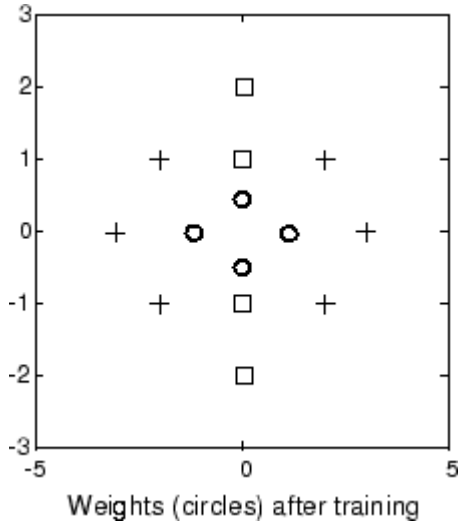
Next you need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. You do this with `train` as with the following commands. First, set the training epochs to 150. Then, use `train`:

```
net.trainParam.epochs = 150;
net = train(net,P,T);
```

Now confirm the first-layer weights.

```
net.IW{1,1}
ans =
    0.3283    0.0051
   -0.1366    0.0001
   -0.0263    0.2234
         0   -0.0685
```

The following plot shows that these weights have moved toward their respective classification groups.



To confirm that these weights do indeed lead to the correct classification, take the matrix `P` as input and simulate the network. Then see what classifications are produced by the network.

```
Y = net(P);
Yc = vec2ind(Y)
```

This gives

```
Yc =
     1     1     1     2     2     2     2     1     1     1
```

which is expected. As a last check, try an input close to a vector that was used in training.

```
pchk1 = [0; 0.5];
Y = net(pchk1);
Yc1 = vec2ind(Y)
```

This gives

```
Yc1 =
     2
```

This looks right, because `pchk1` is close to other vectors classified as 2. Similarly,

```
pchk2 = [1; 0];
Y = net(pchk2);
Yc2 = vec2ind(Y)
```

gives

```
Yc2 =
     1
```

This looks right too, because `pchk2` is close to other vectors classified as 1.

You might want to try the example program “Learning Vector Quantization” on page 31-130. It follows the discussion of training given above.

Supplemental LVQ2.1 Learning Rule (learnlv2)

The following learning rule is one that might be applied *after* first applying LVQ1. It can improve the result of the first learning. This particular version of LVQ2 (referred to as LVQ2.1 in the literature [Koho97 on page 32-2]) is embodied in the function `learnlv2`. Note again that LVQ2.1 is to be used only after LVQ1 has been applied.

Learning here is similar to that in `learnlv2` except now two vectors of layer 1 that are closest to the input vector can be updated, provided that one belongs to the correct class and one belongs to a wrong class, and further provided that the input falls into a “window” near the midplane of the two vectors.

The window is defined by

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s$$

where

$$s \equiv \frac{1-w}{1+w}$$

(where d_i and d_j are the Euclidean distances of $\mathbf{p}$ from ${}_{i*}\mathbf{IW}^{1,1}$ and ${}_{j*}\mathbf{IW}^{1,1}$, respectively). Take a value for w in the range 0.2 to 0.3. If you pick, for instance, 0.25, then $s = 0.6$. This means that if the

minimum of the two distance ratios is greater than 0.6, the two vectors are adjusted. That is, if the input is near the midplane, adjust the two vectors, provided also that the input vector $\mathbf{p}$ and ${}_{j*}\mathbf{IW}^{1,1}$ belong to the same class, and $\mathbf{p}$ and ${}_{i*}\mathbf{IW}^{1,1}$ do not belong in the same class.

The adjustments made are

$${}_{i*}\mathbf{IW}^{1,1}(q) = {}_{i*}\mathbf{IW}^{1,1}(q-1) - \alpha(\mathbf{p}(q) - {}_{i*}\mathbf{IW}^{1,1}(q-1))$$

and

$${}_{j*}\mathbf{IW}^{1,1}(q) = {}_{j*}\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_{j*}\mathbf{IW}^{1,1}(q-1))$$

Thus, given two vectors closest to the input, as long as one belongs to the wrong class and the other to the correct class, and as long as the input falls in a midplane window, the two vectors are adjusted. Such a procedure allows a vector that is just barely classified correctly with LVQ1 to be moved even closer to the input, so the results are more robust.

Function	Description
`competlayer`	Create a competitive layer.
`learnk`	Kohonen learning rule.
`selforgmap`	Create a self-organizing map.
`learncon`	Conscience bias learning function.
`boxdist`	Distance between two position vectors.
`dist`	Euclidean distance weight function.
`linkdist`	Link distance function.
`mandist`	Manhattan distance weight function.
`gridtop`	Gridtop layer topology function.
`hextop`	Hexagonal layer topology function.
`randtop`	Random layer topology function.
`lvqnet`	Create a learning vector quantization network.
`learnlv1`	LVQ1 weight learning function.
`learnlv2`	LVQ2 weight learning function.

Adaptive Filters and Adaptive Training

Adaptive Neural Network Filters

In this section...

The ADALINE (adaptive linear neuron) networks discussed in this topic are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can solve only linearly separable problems. However, here the LMS (least mean squares) learning rule, which is much more powerful than the perceptron learning rule, is used. The LMS, or Widrow-Hoff, learning rule minimizes the mean square error and thus moves the decision boundaries as far as it can from the training patterns.

In this section, you design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancelation, signal processing, and control systems.

The pioneering work in this field was done by Widrow and Hoff, who gave the name ADALINE to adaptive linear elements. The basic reference on this subject is Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York, Prentice-Hall, 1985.

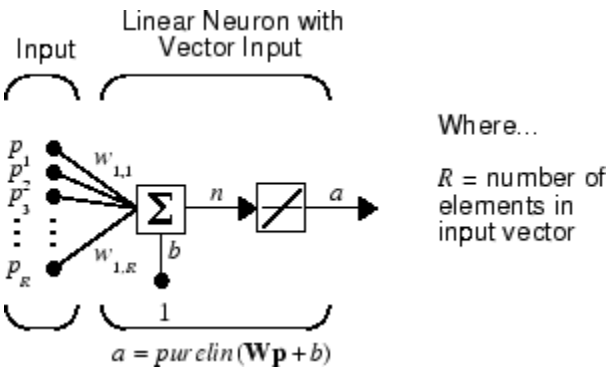
The adaptive training of self-organizing and competitive networks is also considered in this section.

Adaptive Functions

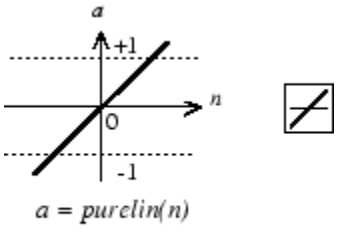
This section introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

Linear Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, named `purelin`.



Linear Transfer Function

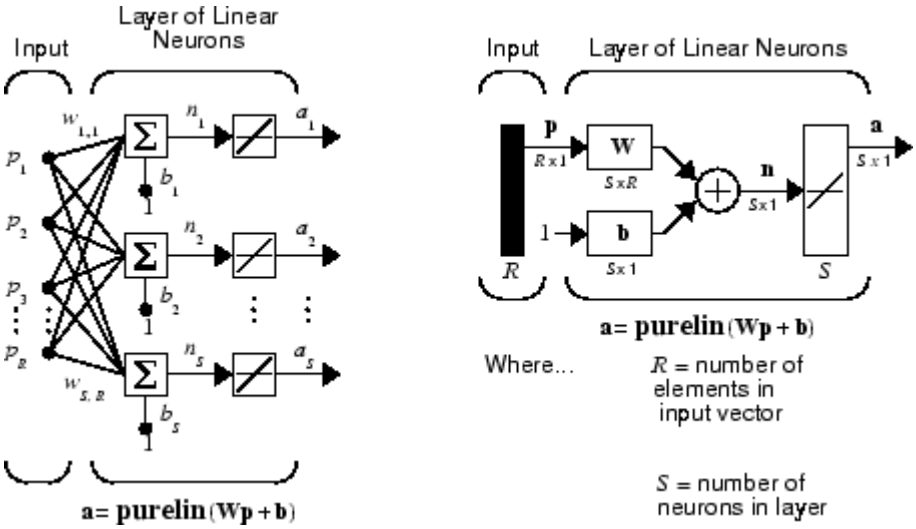
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights $\mathbf{W}$.

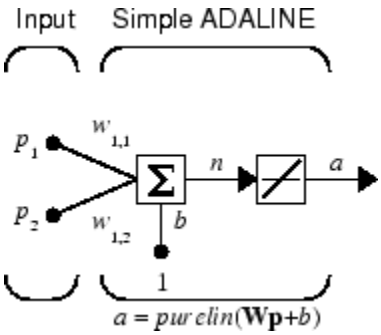


This network is sometimes called a MADALINE for Many ADALINEs. Note that the figure on the right defines an S-length output vector $\mathbf{a}$.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (linearlayer)

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.



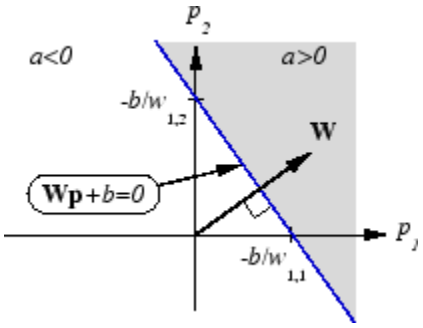
The weight matrix **W** in this case has only one row. The network output is

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

or

$$\alpha = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary, as shown below (adapted with thanks from [HDB96 on page 32-2]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

However, ADALINE can classify objects in this way only when the objects are linearly separable. Thus, ADALINE has the same limitation as the perceptron.

You can create a network similar to the one shown using this command:

```
net = linearlayer;
net = configure(net,[0;0],[0]);
```

The sizes of the two arguments to configure indicate that the layer is to have two inputs and one output. Normally `train` does this configuration for you, but this allows us to inspect the weights before training.

The network weights and biases are set to zero, by default. You can see the current values using the commands:

```
W = net.IW{1,1}
W =
     0     0
```

and

```
b = net.b{1}
b =
     0
```

You can also assign arbitrary values to the weights and bias, such as 2 and 3 for the weights and −4 for the bias:

```
net.IW{1,1} = [2 3];
net.b{1} = -4;
```

You can simulate the ADALINE for a particular input vector.

```
p = [5; 6];
a = sim(net,p)
a =
    24
```

To summarize, you can create an ADALINE network with `linearlayer`, adjust its elements as you want, and simulate it with `sim`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} t(k) - \alpha(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can learn more about this topic in Chapter 10 of [HDB96 on page 32-2].

LMS Algorithm (learnwh)

Adaptive networks will use the LMS algorithm or Widrow-Hoff learning algorithm based on an approximate steepest descent procedure. Here again, adaptive linear networks are trained on examples of correct behavior.

The LMS algorithm, shown here, is discussed in detail in “Linear Neural Networks” on page 29-14.

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

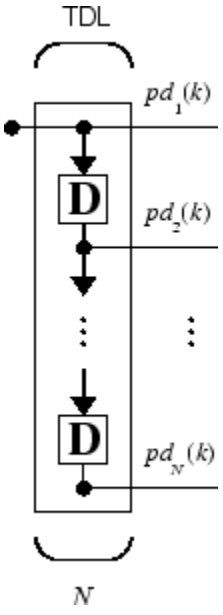
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural networks found in practical applications. Adaptive filtering is one of its major application areas.

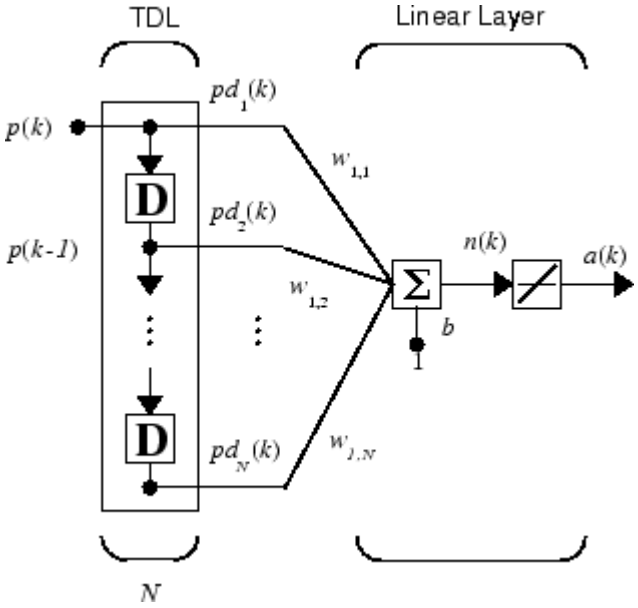
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the ADALINE network. Such a delay line is shown in the next figure. The input signal enters from the left and passes through N-1 delays. The output of the tapped delay line (TDL) is an N-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Adaptive Filter

You can combine a tapped delay line with an ADALINE network to create the *adaptive filter* shown in the next figure.



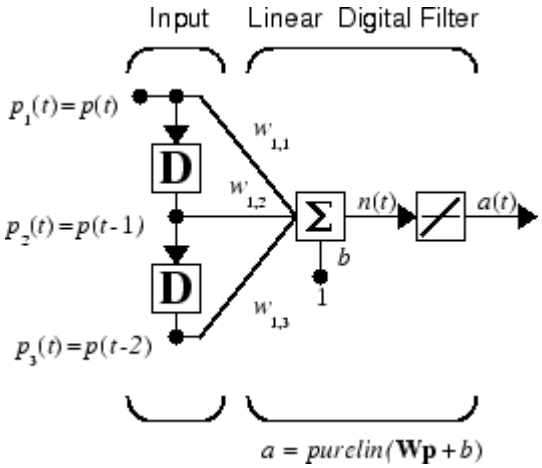
The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i}\alpha(k - i + 1) + b$$

In digital signal processing, this network is referred to as a *finite impulse response (FIR)* filter [WiSt85 on page 32-2]. Take a look at the code used to generate and simulate such an adaptive network.

Adaptive Filter Example

First, define a new linear network using `linearlayer`.



Assume that the linear layer has a single neuron with a single input and a tap delay of 0, 1, and 2 delays.

```
net = linearlayer([0 1 2]);
net = configure(net,0,0);
```

You can specify as many delays as you want, and can omit some values if you like. They must be in ascending order.

You can give the various weights and the bias values with

```
net.IW{1,1} = [7 8 9];
net.b{1} = [0];
```

Finally, define the initial values of the outputs of the delays as

```
pi = {1 2};
```

These are ordered from left to right to correspond to the delays taken from top to bottom in the figure. This concludes the setup of the network.

To set up the input, assume that the input scalars arrive in a sequence: first the value 3, then the value 4, next the value 5, and finally the value 6. You can indicate this sequence by defining the values as elements of a cell array in curly braces.

```
p = {3 4 5 6};
```

Now, you have a network and a sequence of inputs. Simulate the network to see what its output is as a function of time.

```
[a,pf] = sim(net,p,pi)
```

This simulation yields an output sequence

```
a
    [46]    [70]    [94]    [118]
```

and final values for the delay outputs of

```
pf
    [5]    [6]
```

The example is sufficiently simple that you can check it without a calculator to make sure that you understand the inputs, initial values of the delays, etc.

The network just defined can be trained with the function `adapt` to produce a particular output sequence. Suppose, for instance, you want the network to produce the sequence of values 10, 20, 30, 40.

```
t = {10 20 30 40};
```

You can train the defined network to do this, starting from the initial delay conditions used above.

Let the network adapt for 10 passes over the data.

```
for i = 1:10
    [net,y,E,pf,af] = adapt(net,p,t,pi);
end
```

This code returns the final weights, bias, and output sequence shown here.

```
wts = net.IW{1,1}
wts =
    0.5059    3.1053    5.7046
```

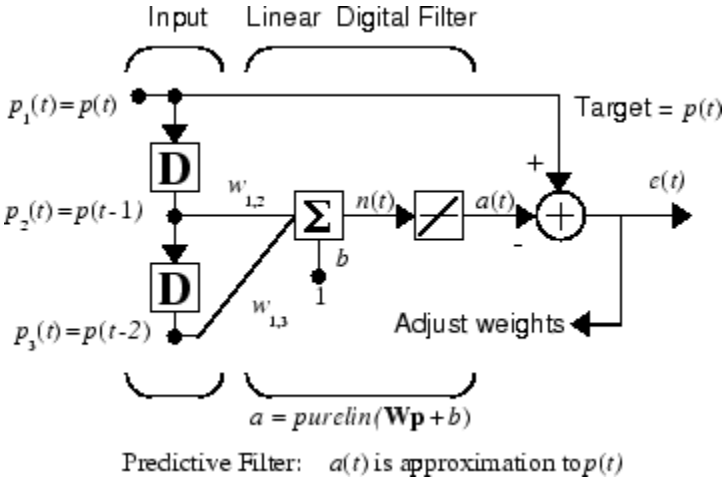
```
bias = net.b{1}
bias =
   -1.5993
y
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

Presumably, if you ran additional passes the output sequence would have been even closer to the desired values of 10, 20, 30, and 40.

Thus, adaptive networks can be specified, simulated, and finally trained with `adapt`. However, the outstanding value of adaptive networks lies in their use to perform a particular function, such as prediction or noise cancelation.

Prediction Example

Suppose that you want to use an adaptive filter to predict the next value of a stationary random process, $p(t)$. You can use the network shown in the following figure to do this prediction.



Predictive Filter: $a(t)$ is approximation to $p(t)$

The signal to be predicted, $p(t)$, enters from the left into a tapped delay line. The previous two values of $p(t)$ are available as outputs from the tapped delay line. The network uses `adapt` to change the weights on each time step so as to minimize the error $e(t)$ on the far right. If this error is 0, the network output $a(t)$ is exactly equal to $p(t)$, and the network has done its prediction properly.

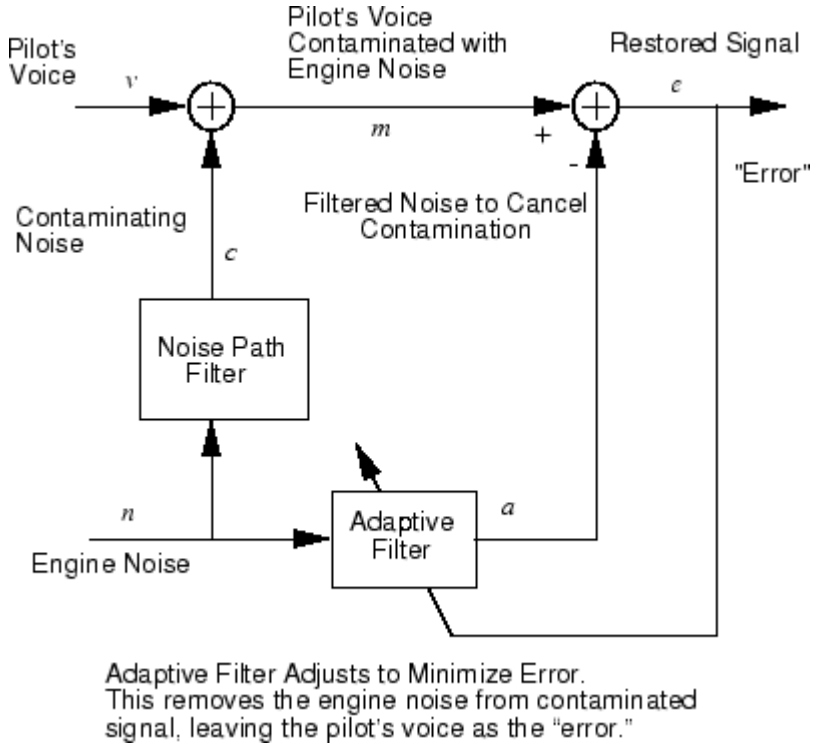
Given the autocorrelation function of the stationary random process $p(t)$, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. This lack does not matter to the network. After it is initialized and operating, the network adapts at each time step to minimize the error and in a relatively short time is able to predict the input $p(t)$.

Chapter 10 of [HDB96 on page 32-2] presents this problem, goes through the analysis, and shows the weight trajectory during training. The network finds the optimum weights on its own without any difficulty whatsoever.

You also can try the example `nnd10nc` to see an adaptive noise cancelation program example in action. This example allows you to pick a learning rate and *momentum* (see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 22-2), and shows the learning trajectory, and the original and cancelation signals versus time.

Noise Cancelation Example

Consider a pilot in an airplane. When the pilot speaks into a microphone, the engine noise in the cockpit combines with the voice signal. This additional noise makes the resultant signal heard by passengers of low quality. The goal is to obtain a signal that contains the pilot's voice, but not the engine noise. You can cancel the noise with an adaptive filter if you obtain a sample of the engine noise and apply it as the input to the adaptive filter.



As the preceding figure shows, you adaptively train the neural linear network to predict the combined pilot/engine signal *m* from an engine signal *n*. The engine signal *n* does not tell the adaptive network anything about the pilot's voice signal contained in *m*. However, the engine signal *n* does give the network information it can use to predict the engine's contribution to the pilot/engine signal *m*.

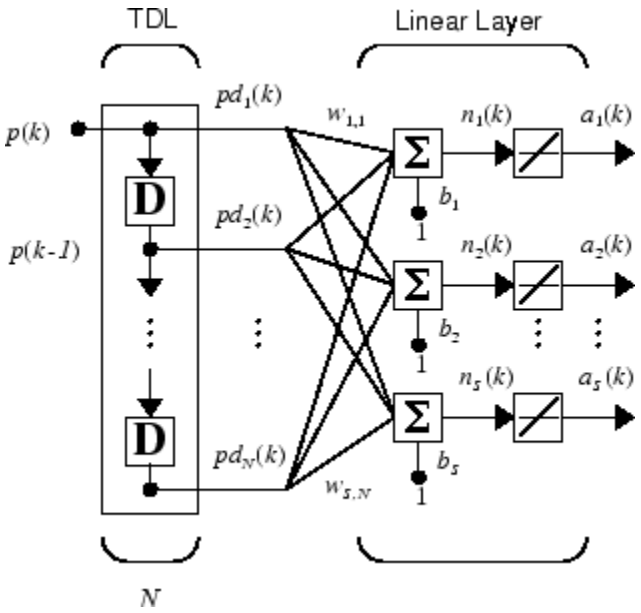
The network does its best to output *m* adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal *m*. The network error *e* is equal to *m*, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, *e* contains only the pilot's voice. The linear adaptive network adaptively learns to cancel the engine noise.

Such adaptive noise canceling generally does a better job than a classical filter, because it subtracts from the signal rather than filtering it out the noise of the signal *m*.

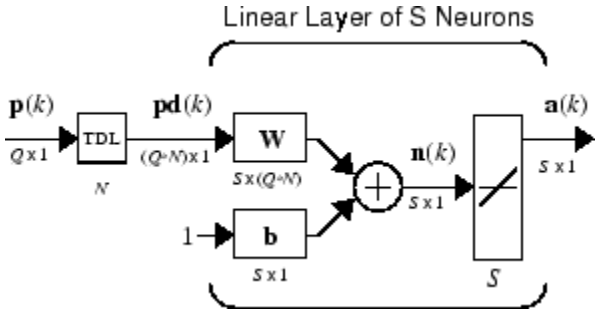
Try “Adaptive Noise Cancellation” on page 31-180 for an example of adaptive noise cancellation.

Multiple Neuron Adaptive Filters

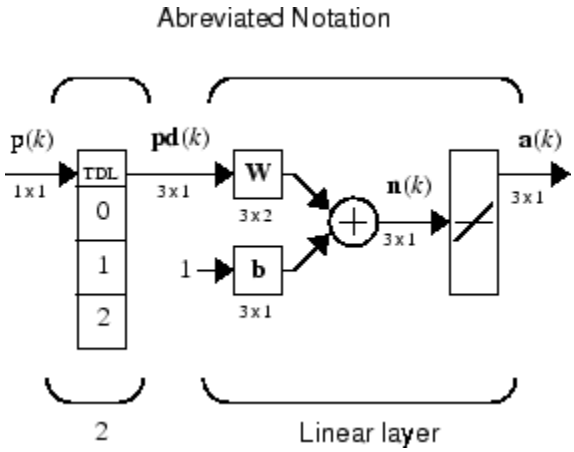
You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with *S* linear neurons, as shown in the next figure.



Alternatively, you can represent this same network in abbreviated form.



If you want to show more of the detail of the tapped delay line—and there are not too many delays—you can use the following notation:



Here, a tapped delay line sends to the weight matrix:

- The current signal
- The previous signal
- The signal delayed before that

You could have a longer list, and some delay values could be omitted if desired. The only requirement is that the delays must appears in increasing order as they go from top to bottom.

Advanced Topics

Shallow Neural Networks with Parallel and GPU Computing

In this section...

Note For deep learning, parallel and GPU support is automatic. You can train a convolutional neural network (CNN, ConvNet) or long short-term memory networks (LSTM or BiLSTM networks) using the `trainNetwork` function and choose the execution environment (CPU, GPU, multi-GPU, and parallel) using `trainingOptions`.

Training in parallel, or on a GPU, requires Parallel Computing Toolbox. For more information on deep learning with GPUs and in parallel, see “Deep Learning with Big Data on CPUs, GPUs, in Parallel, and on the Cloud” on page 1-6.

Modes of Parallelism

Neural networks are inherently parallel algorithms. Multicore CPUs, graphical processing units (GPUs), and clusters of computers with multiple CPUs and GPUs can take advantage of this parallelism.

Parallel Computing Toolbox, when used in conjunction with Deep Learning Toolbox, enables neural network training and simulation to take advantage of each mode of parallelism.

For example, the following shows a standard single-threaded training and simulation session:

```
[x, t] = bodyfat_dataset;
net1 = feedforwardnet(10);
net2 = train(net1, x, t);
y = net2(x);
```

The two steps you can parallelize in this session are the call to `train` and the implicit call to `sim` (where the network `net2` is called as a function).

In Deep Learning Toolbox you can divide any data, such as `x` and `t` in the previous example code, across samples. If `x` and `t` contain only one sample each, there is no parallelism. But if `x` and `t` contain hundreds or thousands of samples, parallelism can provide both speed and problem size benefits.

Distributed Computing

Parallel Computing Toolbox allows neural network training and simulation to run across multiple CPU cores on a single PC, or across multiple CPUs on multiple computers on a network using MATLAB Parallel Server.

Using multiple cores can speed calculations. Using multiple computers can allow you to solve problems using data sets too big to fit in the RAM of a single computer. The only limit to problem size is the total quantity of RAM available across all computers.

To manage cluster configurations, use the Cluster Profile Manager from the MATLAB **Home** tab **Environment** menu **Parallel > Manage Cluster Profiles**.

To open a pool of MATLAB workers using the default cluster profile, which is usually the local CPU cores, use this command:

```
pool = parpool
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

When `parpool` runs, it displays the number of workers available in the pool. Another way to determine the number of workers is to query the pool:

```
pool.NumWorkers
```

```
   4
```

Now you can train and simulate the neural network with data split by sample across all the workers. To do this, set the `train` and `sim` parameter `'useParallel'` to `'yes'`.

```
net2 = train(net1,x,t,'useParallel','yes')
y = net2(x,'useParallel','yes')
```

Use the `'showResources'` argument to verify that the calculations ran across multiple workers.

```
net2 = train(net1,x,t,'useParallel','yes','showResources','yes');
y = net2(x,'useParallel','yes','showResources','yes');
```

MATLAB indicates which resources were used. For example:

```
Computing Resources:
Parallel Workers
  Worker 1 on MyComputer, MEX on PCWIN64
  Worker 2 on MyComputer, MEX on PCWIN64
  Worker 3 on MyComputer, MEX on PCWIN64
  Worker 4 on MyComputer, MEX on PCWIN64
```

When `train` and `sim` are called, they divide the input matrix or cell array data into distributed Composite values before training and simulation. When `sim` has calculated a Composite, this output is converted back to the same matrix or cell array form before it is returned.

However, you might want to perform this data division manually if:

- The problem size is too large for the host computer. Manually defining the elements of Composite values sequentially allows much bigger problems to be defined.
- It is known that some workers are on computers that are faster or have more memory than others. You can distribute the data with differing numbers of samples per worker. This is called load balancing.

The following code sequentially creates a series of random datasets and saves them to separate files:

```
pool = gcp;
for i=1:pool.NumWorkers
```

```
  x = rand(2,1000);
  save(['inputs' num2str(i)],'x');
  t = x(1,:) .* x(2,:) + 2 * (x(1,:) + x(2,:));
  save(['targets' num2str(i)],'t');
  clear x t
end
```

Because the data was defined sequentially, you can define a total dataset larger than can fit in the host PC memory. PC memory must accommodate only a sub-dataset at a time.

Now you can load the datasets sequentially across parallel workers, and train and simulate a network on the Composite data. When `train` or `sim` is called with Composite data, the `'useParallel'` argument is automatically set to `'yes'`. When using Composite data, configure the network's input and outputs to match one of the datasets manually using the `configure` function before training.

```
xc = Composite;
tc = Composite;
for i=1:pool.NumWorkers
  data = load(['inputs' num2str(i)],'x');
  xc{i} = data.x;
  data = load(['targets' num2str(i)],'t');
  tc{i} = data.t;
  clear data
end
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc);
yc = net2(xc);
```

To convert the Composite output returned by `sim`, you can access each of its elements, separately if concerned about memory limitations.

```
for i=1:pool.NumWorkers
  yi = yc{i}
end
```

Combined the Composite value into one local value if you are not concerned about memory limitations.

```
y = {yc{:}};
```

When load balancing, the same process happens, but, instead of each dataset having the same number of samples (1000 in the previous example), the numbers of samples can be adjusted to best take advantage of the memory and speed differences of the worker host computers.

It is not required that each worker have data. If element `i` of a Composite value is undefined, worker `i` will not be used in the computation.

Single GPU Computing

The number of cores, size of memory, and speed efficiencies of GPU cards are growing rapidly with each new generation. Where video games have long benefited from improved GPU performance, these cards are now flexible enough to perform general numerical computing tasks like training neural networks.

For the latest GPU requirements, see the web page for Parallel Computing Toolbox; or query MATLAB to determine whether your PC has a supported GPU. This function returns the number of GPUs in your system:

```
count = gpuDeviceCount
```

```
count =

    1
```

If the result is one or more, you can query each GPU by index for its characteristics. This includes its name, number of multiprocessors, `SIMDWidth` of each multiprocessor, and total memory.

```
gpu1 = gpuDevice(1)
```

```
gpu1 =

  CUDADevice with properties:

                      Name: 'GeForce GTX 470'
                     Index: 1
         ComputeCapability: '2.0'
            SupportsDouble: 1
             DriverVersion: 4.1000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [65535 65535 1]
                 SIMDWidth: 32
               TotalMemory: 1.3422e+09
           AvailableMemory: 1.1056e+09
       MultiprocessorCount: 14
              ClockRateKHz: 1215000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1
```

The simplest way to take advantage of the GPU is to specify call `train` and `sim` with the parameter argument `'useGPU'` set to `'yes'` (`'no'` is the default).

```
net2 = train(net1,x,t,'useGPU','yes')
y = net2(x,'useGPU','yes')
```

If `net1` has the default training function `trainlm`, you see a warning that GPU calculations do not support Jacobian training, only gradient training. So the training function is automatically changed to the gradient training function `trainscg`. To avoid the notice, you can specify the function before training:

```
net1.trainFcn = 'trainscg';
```

To verify that the training and simulation occur on the GPU device, request that the computer resources be shown:

```
net2 = train(net1,x,t,'useGPU','yes','showResources','yes')
y = net2(x,'useGPU','yes','showResources','yes')
```

Each of the above lines of code outputs the following resources summary:

```
Computing Resources:
GPU device #1, GeForce GTX 470
```

Many MATLAB functions automatically execute on a GPU when any of the input arguments is a gpuArray. Normally you move arrays to and from the GPU with the functions `gpuArray` and `gather`. However, for neural network calculations on a GPU to be efficient, matrices need to be transposed and the columns padded so that the first element in each column aligns properly in the GPU memory. Deep Learning Toolbox provides a special function called `nndata2gpu` to move an array to a GPU and properly organize it:

```
xg = nndata2gpu(x);
tg = nndata2gpu(t);
```

Now you can train and simulate the network using the converted data already on the GPU, without having to specify the `'useGPU'` argument. Then convert and return the resulting GPU array back to MATLAB with the complementary function `gpu2nndata`.

Before training with gpuArray data, the network's input and outputs must be manually configured with regular MATLAB matrices using the `configure` function:

```
net2 = configure(net1,x,t);  % Configure with MATLAB arrays
net2 = train(net2,xg,tg);    % Execute on GPU with NNET formatted gpuArrays
yg = net2(xg);               % Execute on GPU
y = gpu2nndata(yg);          % Transfer array to local workspace
```

On GPUs and other hardware where you might want to deploy your neural networks, it is often the case that the exponential function `exp` is not implemented with hardware, but with a software library. This can slow down neural networks that use the `tansig` sigmoid transfer function. An alternative function is the Elliot sigmoid function whose expression does not include a call to any higher order functions:

```
(equation)    a = n / (1 + abs(n))
```

Before training, the network's `tansig` layers can be converted to `elliotsig` layers as follows:

```
for i=1:net.numLayers
  if strcmp(net.layers{i}.transferFcn,'tansig')
    net.layers{i}.transferFcn = 'elliotsig';
  end
end
```

Now training and simulation might be faster on the GPU and simpler deployment hardware.

Distributed GPU Computing

Distributed and GPU computing can be combined to run calculations across multiple CPUs and/or GPUs on a single computer, or on a cluster with MATLAB Parallel Server.

The simplest way to do this is to specify `train` and `sim` to do so, using the parallel pool determined by the cluster profile you use. The `'showResources'` option is especially recommended in this case, to verify that the expected hardware is being employed:

```
net2 = train(net1,x,t,'useParallel','yes','useGPU','yes','showResources','yes')
y = net2(x,'useParallel','yes','useGPU','yes','showResources','yes')
```

These lines of code use all available workers in the parallel pool. One worker for each unique GPU employs that GPU, while other workers operate as CPUs. In some cases, it might be faster to use only

GPUs. For instance, if a single computer has three GPUs and four workers each, the three workers that are accelerated by the three GPUs might be speed limited by the fourth CPU worker. In these cases, you can specify that `train` and `sim` use only workers with unique GPUs.

```
net2 = train(net1,x,t,'useParallel','yes','useGPU','only','showResources','yes')
y = net2(x,'useParallel','yes','useGPU','only','showResources','yes')
```

As with simple distributed computing, distributed GPU computing can benefit from manually created Composite values. Defining the Composite values yourself lets you indicate which workers to use, how many samples to assign to each worker, and which workers use GPUs.

For instance, if you have four workers and only three GPUs, you can define larger datasets for the GPU workers. Here, a random dataset is created with different sample loads per Composite element:

```
numSamples = [1000 1000 1000 300];
xc = Composite;
tc = Composite;
for i=1:4
  xi = rand(2,numSamples(i));
  ti = xi(1,:).^2 + 3*xi(2,:);
  xc{i} = xi;
  tc{i} = ti;
end
```

You can now specify that `train` and `sim` use the three GPUs available:

```
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc,'useGPU','yes','showResources','yes');
yc = net2(xc,'showResources','yes');
```

To ensure that the GPUs get used by the first three workers, manually converting each worker's Composite elements to gpuArrays. Each worker performs this transformation within a parallel executing `spmd` block.

```
spmd
  if labindex <= 3
    xc = nndata2gpu(xc);
    tc = nndata2gpu(tc);
  end
end
```

Now the data specifies when to use GPUs, so you do not need to tell `train` and `sim` to do so.

```
net2 = configure(net1,xc{1},tc{1});
net2 = train(net2,xc,tc,'showResources','yes');
yc = net2(xc,'showResources','yes');
```

Ensure that each GPU is used by only one worker, so that the computations are most efficient. If multiple workers assign gpuArray data on the same GPU, the computation will still work but will be slower, because the GPU will operate on the multiple workers' data sequentially.

Parallel Time Series

For time series networks, simply use cell array values for `x` and `t`, and optionally include initial input delay states `xi` and initial layer delay states `ai`, as required.

```
net2 = train(net1,x,t,xi,ai,'useGPU','yes')
y = net2(x,xi,ai,'useParallel','yes','useGPU','yes')
```

```
net2 = train(net1,x,t,xi,ai,'useParallel','yes')
y = net2(x,xi,ai,'useParallel','yes','useGPU','only')

net2 = train(net1,x,t,xi,ai,'useParallel','yes','useGPU','only')
y = net2(x,xi,ai,'useParallel','yes','useGPU','only')
```

Note that parallelism happens across samples, or in the case of time series across different series. However, if the network has only input delays, with no layer delays, the delayed inputs can be precalculated so that for the purposes of computation, the time steps become different samples and can be parallelized. This is the case for networks such as `timedelaynet` and open-loop versions of `narxnet` and `narnet`. If a network has layer delays, then time cannot be “flattened” for purposes of computation, and so single series data cannot be parallelized. This is the case for networks such as `layrecnet` and closed-loop versions of `narxnet` and `narnet`. However, if the data consists of multiple sequences, it can be parallelized across the separate sequences.

Parallel Availability, Fallbacks, and Feedback

As mentioned previously, you can query MATLAB to discover the current parallel resources that are available.

To see what GPUs are available on the host computer:

```
gpuCount = gpuDeviceCount
for i=1:gpuCount
  gpuDevice(i)
end
```

To see how many workers are running in the current parallel pool:

```
poolSize = pool.NumWorkers
```

To see the GPUs available across a parallel pool running on a PC cluster using MATLAB Parallel Server:

```
spmd
  worker.index = labindex;
  worker.name = system('hostname');
  worker.gpuCount = gpuDeviceCount;
  try
    worker.gpuInfo = gpuDevice;
  catch
    worker.gpuInfo = [];
  end
  worker
end
```

When `'useParallel'` or `'useGPU'` are set to `'yes'`, but parallel or GPU workers are unavailable, the convention is that when resources are requested, they are used if available. The computation is performed without error even if they are not. This process of falling back from requested resources to actual resources happens as follows:

- If `'useParallel'` is `'yes'` but Parallel Computing Toolbox is unavailable, or a parallel pool is not open, then computation reverts to single-threaded MATLAB.
- If `'useGPU'` is `'yes'` but the gpuDevice for the current MATLAB session is unassigned or not supported, then computation reverts to the CPU.

- If `'useParallel'` and `'useGPU'` are `'yes'`, then each worker with a unique GPU uses that GPU, and other workers revert to CPU.
- If `'useParallel'` is `'yes'` and `'useGPU'` is `'only'`, then workers with unique GPUs are used. Other workers are not used, unless no workers have GPUs. In the case with no GPUs, all workers use CPUs.

When unsure about what hardware is actually being employed, check `gpuDeviceCount`, `gpuDevice`, and `pool.NumWorkers` to ensure the desired hardware is available, and call `train` and `sim` with `'showResources'` set to `'yes'` to verify what resources were actually used.

Optimize Neural Network Training Speed and Memory

In this section...

Memory Reduction

Depending on the particular neural network, simulation and gradient calculations can occur in MATLAB or MEX. MEX is more memory efficient, but MATLAB can be made more memory efficient in exchange for time.

To determine whether MATLAB or MEX is being used, use the `'showResources'` option, as shown in this general form of the syntax:

```
net2 = train(net1,x,t,'showResources','yes')
```

If MATLAB is being used and memory limitations are a problem, the amount of temporary storage needed can be reduced by a factor of `N`, in exchange for performing the computations `N` times sequentially on each of `N` subsets of the data.

```
net2 = train(net1,x,t,'reduction',N);
```

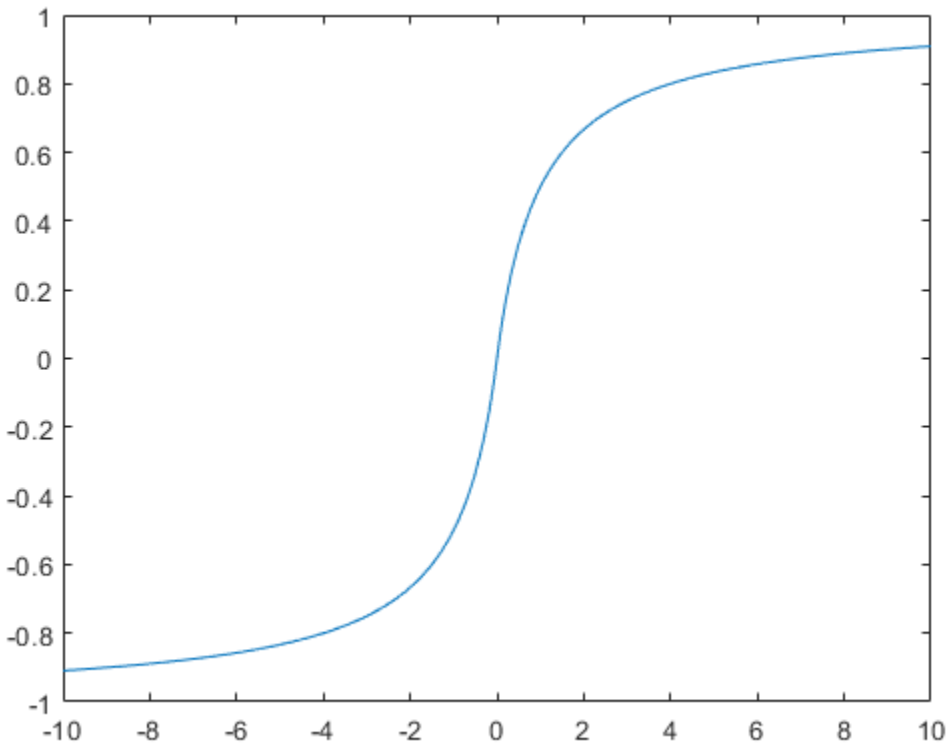
This is called memory reduction.

Fast Elliot Sigmoid

Some simple computing hardware might not support the exponential function directly, and software implementations can be slow. The Elliot sigmoid `elliotsig` function performs the same role as the symmetric sigmoid `tansig` function, but avoids the exponential function.

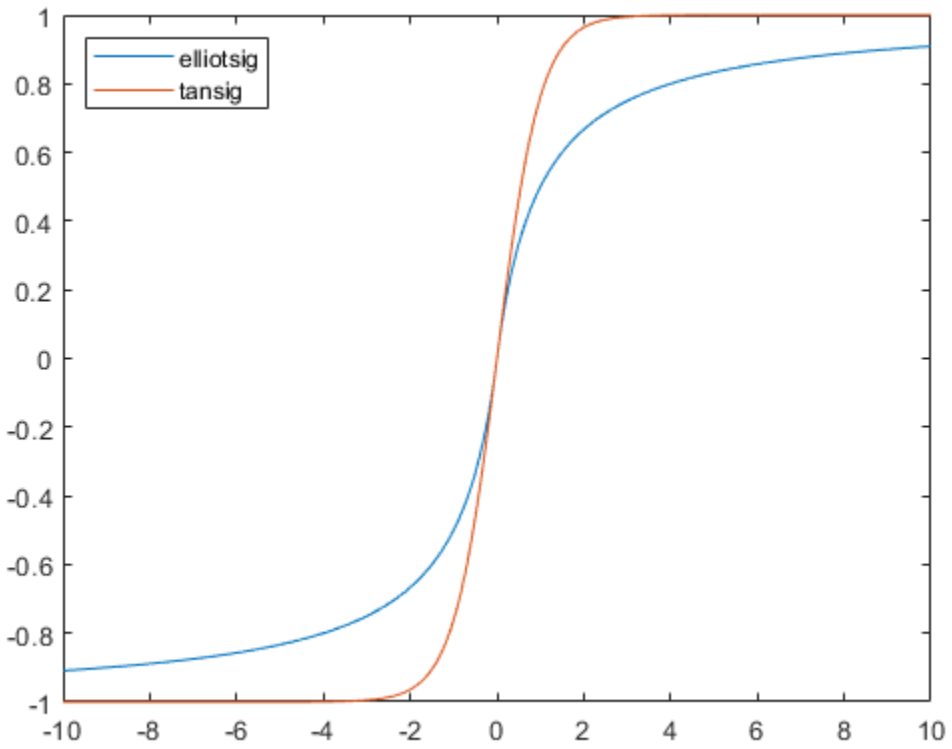
Here is a plot of the Elliot sigmoid:

```
n = -10:0.01:10;
a = elliotsig(n);
plot(n,a)
```



Next, `elliotsig` is compared with `tansig`.

```
a2 = tansig(n);
h = plot(n,a,n,a2);
legend(h,'elliotsig','tansig','Location','NorthWest')
```



To train a neural network using `elliotsig` instead of `tansig`, transform the network's transfer functions:

```
[x,t] = bodyfat_dataset;
net = feedforwardnet;
view(net)
net.layers{1}.transferFcn = 'elliotsig';
view(net)
net = train(net,x,t);
y = net(x)
```

Here, the times to execute `elliotsig` and `tansig` are compared. `elliotsig` is approximately four times faster on the test system.

```
n = rand(5000,5000);
tic,for i=1:100,a=tansig(n); end, tansigTime = toc;
tic,for i=1:100,a=elliotsig(n); end, elliotTime = toc;
speedup = tansigTime / elliotTime

speedup =

    4.1406
```

However, while simulation is faster with `elliotsig`, training is not guaranteed to be faster, due to the different shapes of the two transfer functions. Here, 10 networks are each trained for `tansig` and `elliotsig`, but training times vary significantly even on the same problem with the same network.

```
[x,t] = bodyfat_dataset;
tansigNet = feedforwardnet;
tansigNet.trainParam.showWindow = false;
elliotNet = tansigNet;
elliotNet.layers{1}.transferFcn = 'elliotsig';
for i=1:10, tic, net = train(tansigNet,x,t); tansigTime = toc, end
for i=1:10, tic, net = train(elliotNet,x,t), elliotTime = toc, end
```

Choose a Multilayer Neural Network Training Function

In this section...

It is very difficult to know which training algorithm will be the fastest for a given problem. It depends on many factors, including the complexity of the problem, the number of data points in the training set, the number of weights and biases in the network, the error goal, and whether the network is being used for pattern recognition (discriminant analysis) or function approximation (regression). This section compares the various training algorithms. Feedforward networks are trained on six different problems. Three of the problems fall in the pattern recognition category and the three others fall in the function approximation category. Two of the problems are simple “toy” problems, while the other four are “real world” problems. Networks with a variety of different architectures and complexities are used, and the networks are trained to a variety of different accuracy levels.

The following table lists the algorithms that are tested and the acronyms used to identify them.

Acronym	Algorithm	Description
LM	`trainlm`	Levenberg-Marquardt
BFG	`trainbfg`	BFGS Quasi-Newton
RP	`trainrp`	Resilient Backpropagation
SCG	`trainscg`	Scaled Conjugate Gradient
CGB	`traincgb`	Conjugate Gradient with Powell/Beale Restarts
CGF	`traincgf`	Fletcher-Powell Conjugate Gradient
CGP	`traincgp`	Polak-Ribiére Conjugate Gradient
OSS	`trainoss`	One Step Secant
GDX	`traingdx`	Variable Learning Rate Backpropagation

The following table lists the six benchmark problems and some characteristics of the networks, training processes, and computers used.

Problem Title	Problem Type	Network Structure	Error Goal	Computer
SIN	Function approximation	1-5-1	0.002	Sun Sparc 2
PARITY	Pattern recognition	3-10-10-1	0.001	Sun Sparc 2
ENGINE	Function approximation	2-30-2	0.005	Sun Enterprise 4000

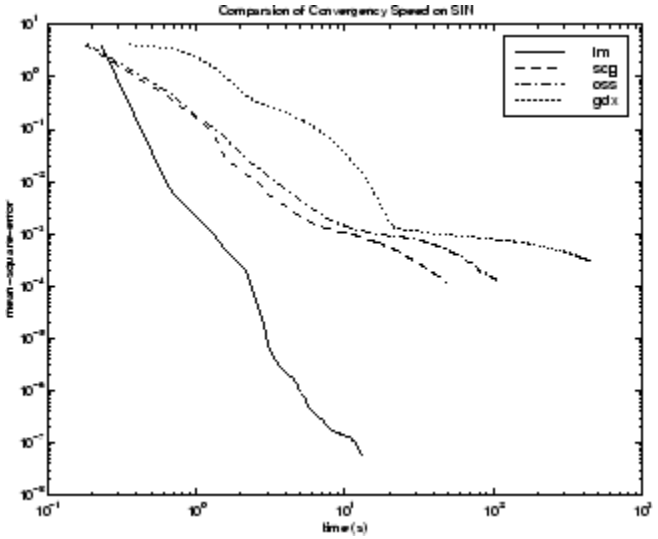
Problem Title	Problem Type	Network Structure	Error Goal	Computer
CANCER	Pattern recognition	9-5-5-2	0.012	Sun Sparc 2
CHOLESTEROL	Function approximation	21-15-3	0.027	Sun Sparc 20
DIABETES	Pattern recognition	8-15-15-2	0.05	Sun Sparc 20

SIN Data Set

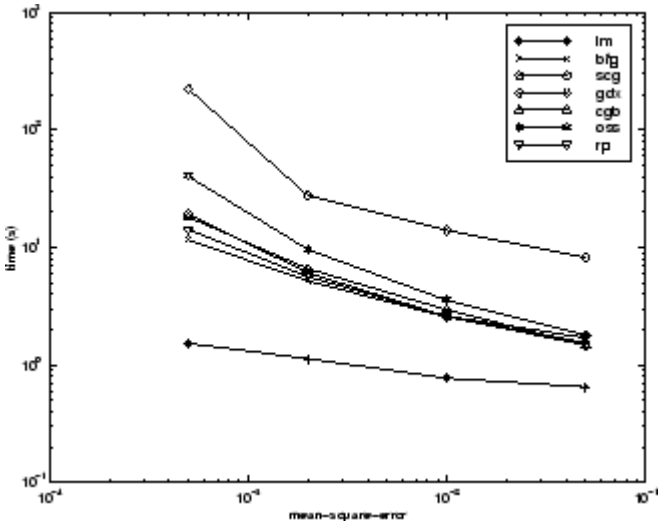
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with `tansig` transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training the network using nine different training algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.002. The fastest algorithm for this problem is the Levenberg-Marquardt algorithm. On the average, it is over four times faster than the next fastest algorithm. This is the type of problem for which the LM algorithm is best suited—a function approximation problem where the network has fewer than one hundred weights and the approximation must be very accurate.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	1.14	1.00	0.65	1.83	0.38
BFG	5.22	4.58	3.17	14.38	2.08
RP	5.67	4.97	2.66	17.24	3.72
SCG	6.09	5.34	3.18	23.64	3.81
CGB	6.61	5.80	2.99	23.65	3.67
CGF	7.86	6.89	3.57	31.23	4.76
CGP	8.24	7.23	4.07	32.32	5.03
OSS	9.64	8.46	3.97	59.63	9.79
GDX	27.69	24.29	17.21	258.15	43.65

The performance of the various algorithms can be affected by the accuracy required of the approximation. This is shown in the following figure, which plots the mean square error versus execution time (averaged over the 30 trials) for several representative algorithms. Here you can see that the error in the LM algorithm decreases much more rapidly with time than the other algorithms shown.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Here you can see that as the error goal is reduced, the improvement provided by the LM algorithm becomes more pronounced. Some algorithms perform better as the error goal is reduced (LM and BFG), and other algorithms degrade as the error goal is reduced (OSS and GDX).



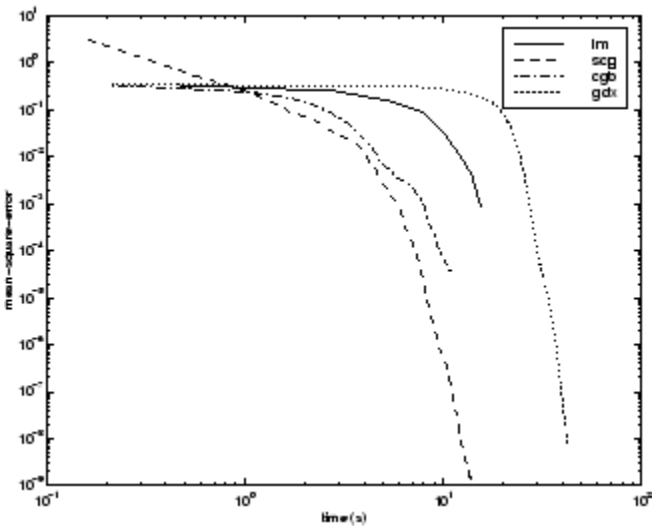
PARITY Data Set

The second benchmark problem is a simple pattern recognition problem—detect the parity of a 3-bit number. If the number of ones in the input pattern is odd, then the network should output a 1; otherwise, it should output a -1. The network used for this problem is a 3-10-10-1 network with tansig neurons in each layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.001. The fastest algorithm for this problem is the resilient backpropagation algorithm, although the conjugate gradient algorithms (in particular, the scaled conjugate gradient algorithm) are almost as fast. Notice that the LM algorithm does not perform well on this problem. In

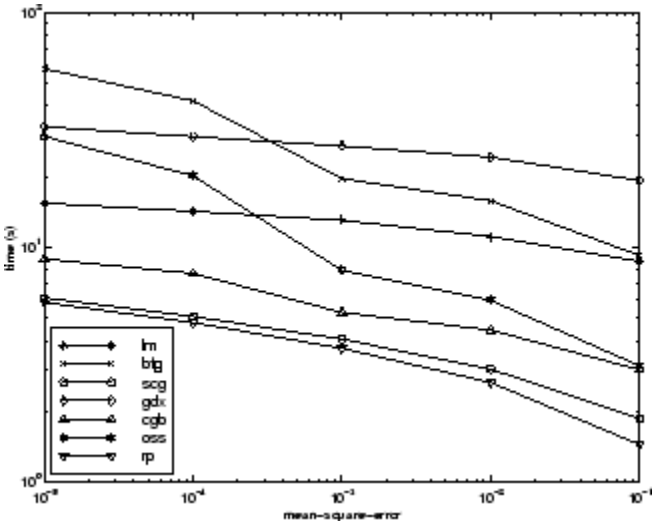
general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Because the output neurons in pattern recognition problems are generally saturated, you will not be operating in the linear region.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	3.73	1.00	2.35	6.89	1.26
SCG	4.09	1.10	2.36	7.48	1.56
CGP	5.13	1.38	3.50	8.73	1.05
CGB	5.30	1.42	3.91	11.59	1.35
CGF	6.62	1.77	3.96	28.05	4.32
OSS	8.00	2.14	5.06	14.41	1.92
LM	13.07	3.50	6.48	23.78	4.96
BFG	19.68	5.28	14.19	26.64	2.85
GDX	27.07	7.26	25.21	28.52	0.86

As with function approximation problems, the performance of the various algorithms can be affected by the accuracy required of the network. This is shown in the following figure, which plots the mean square error versus execution time for some typical algorithms. The LM algorithm converges rapidly after some point, but only after the other algorithms have already converged.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG).

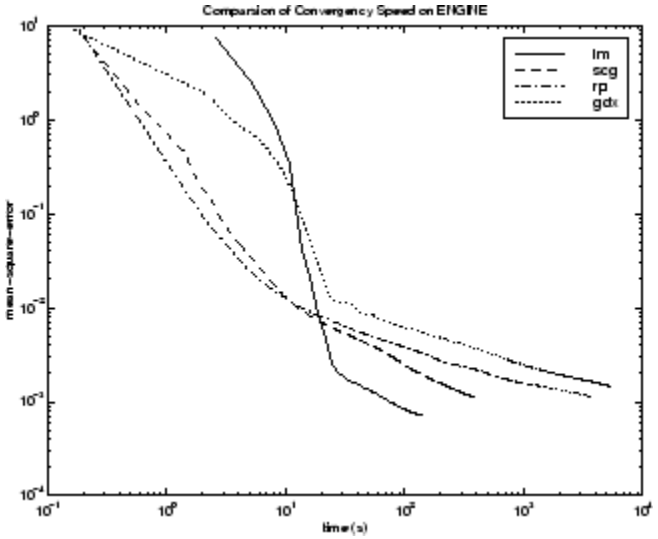


ENGINE Data Set

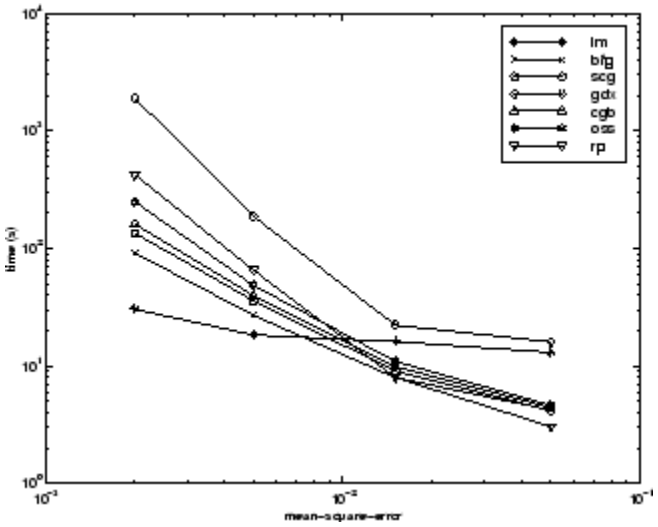
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem is a 2-30-2 network with tansig neurons in the hidden layer and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials (10 trials for RP and GDX because of time constraints), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.005. The fastest algorithm for this problem is the LM algorithm, followed by the BFGS quasi-Newton algorithm and the conjugate gradient algorithms. Although this is a function approximation problem, the LM algorithm is not as clearly superior as it was on the SIN data set. In this case, the number of weights and biases in the network is much larger than the one used on the SIN problem (152 versus 16), and the advantages of the LM algorithm decrease as the number of network parameters increases.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
LM	18.45	1.00	12.01	30.03	4.27
BFG	27.12	1.47	16.42	47.36	5.95
SCG	36.02	1.95	19.39	52.45	7.78
CGF	37.93	2.06	18.89	50.34	6.12
CGB	39.93	2.16	23.33	55.42	7.50
CGP	44.30	2.40	24.99	71.55	9.89
OSS	48.71	2.64	23.51	80.90	12.33
RP	65.91	3.57	31.83	134.31	34.24
GDX	188.50	10.22	81.59	279.90	66.67

The following figure plots the mean square error versus execution time for some typical algorithms. The performance of the LM algorithm improves over time relative to the other algorithms.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (GDX and RP), while the LM algorithm improves.



CANCER Data Set

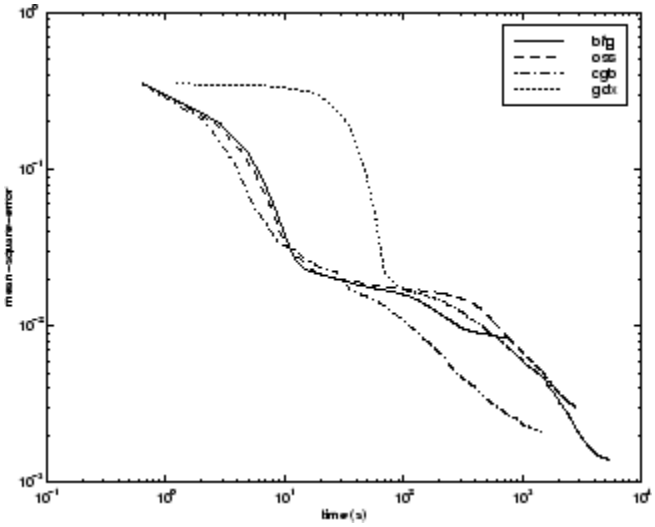
The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity of cell size and cell shape, the amount of marginal adhesion, and the frequency of bare nuclei. The data was obtained from the University of Wisconsin Hospitals, Madison, from Dr. William H. Wolberg. The network used for this problem is a 9-5-5-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 30 different trials, where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than

0.012. A few runs failed to converge for some of the algorithms, so only the top 75% of the runs from each algorithm were used to obtain the statistics.

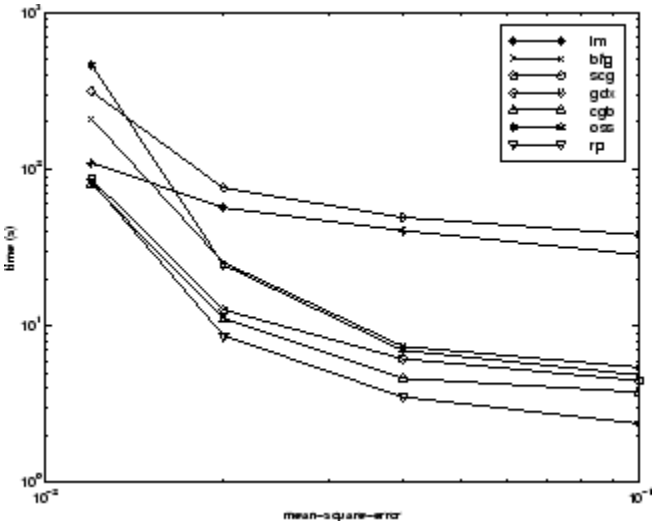
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
CGB	80.27	1.00	55.07	102.31	13.17
RP	83.41	1.04	59.51	109.39	13.44
SCG	86.58	1.08	41.21	112.19	18.25
CGP	87.70	1.09	56.35	116.37	18.03
CGF	110.05	1.37	63.33	171.53	30.13
LM	110.33	1.37	58.94	201.07	38.20
BFG	209.60	2.61	118.92	318.18	58.44
GDX	313.22	3.90	166.48	446.43	75.44
OSS	463.87	5.78	250.62	599.99	97.35

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem there is not as much variation in performance as in previous problems.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. Again you can see that some algorithms degrade as the error goal is reduced (OSS and BFG) while the LM algorithm improves. It is typical of the LM algorithm on any problem that its performance improves relative to other algorithms as the error goal is reduced.



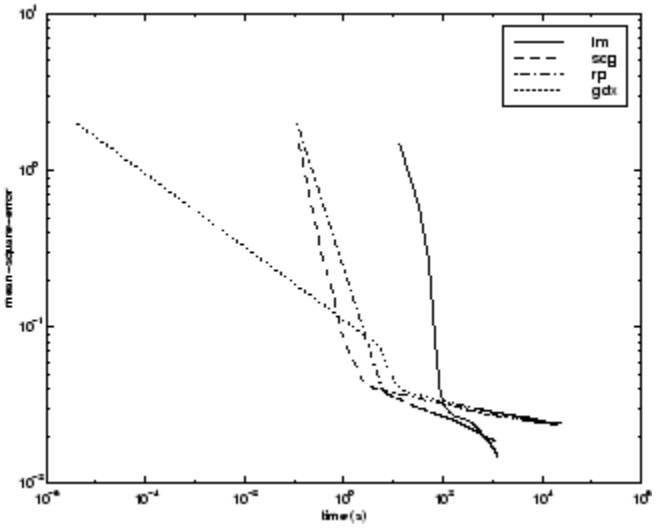
CHOLESTEROL Data Set

The fifth benchmark problem is a realistic function approximation (or nonlinear regression) problem. The objective of the network is to predict cholesterol levels (ldl, hdl, and vldl) based on measurements of 21 spectral components. The data was obtained from Dr. Neil Purdie, Department of Chemistry, Oklahoma State University [PuLu92 on page 32-2]. The network used for this problem is a 21-15-3 network with tansig neurons in the hidden layers and linear neurons in the output layer. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 20 different trials (10 trials for RP and GDX), where different random initial weights are used in each trial. In each case, the network is trained until the squared error is less than 0.027.

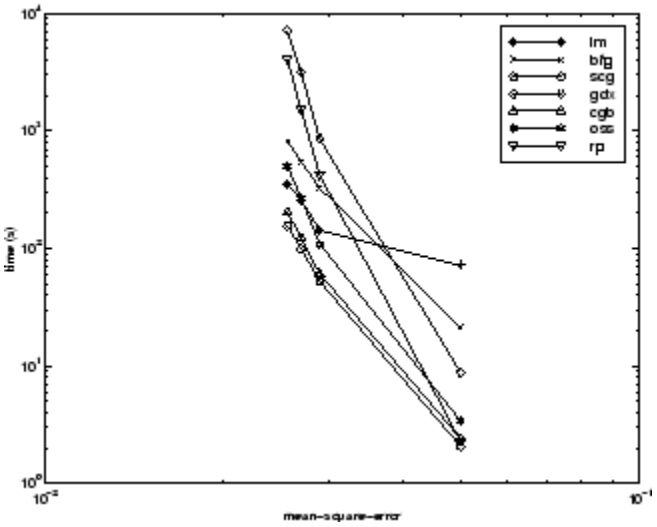
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases in the network has increased again (378 versus 152 versus 16). As the number of parameters increases, the computation required in the LM algorithm increases geometrically.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
SCG	99.73	1.00	83.10	113.40	9.93
CGP	121.54	1.22	101.76	162.49	16.34
CGB	124.06	1.2	107.64	146.90	14.62
CGF	136.04	1.36	106.46	167.28	17.67
LM	261.50	2.62	103.52	398.45	102.06
OSS	268.55	2.69	197.84	372.99	56.79
BFG	550.92	5.52	471.61	676.39	46.59
RP	1519.00	15.23	581.17	2256.10	557.34
GDX	3169.50	31.78	2514.90	4168.20	610.52

The following figure plots the mean square error versus execution time for some typical algorithms. For this problem, you can see that the LM algorithm is able to drive the mean square error to a lower level than the other algorithms. The SCG and RP algorithms provide the fastest initial convergence.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. You can see that the LM and BFG algorithms improve relative to the other algorithms as the error goal is reduced.



DIABETES Data Set

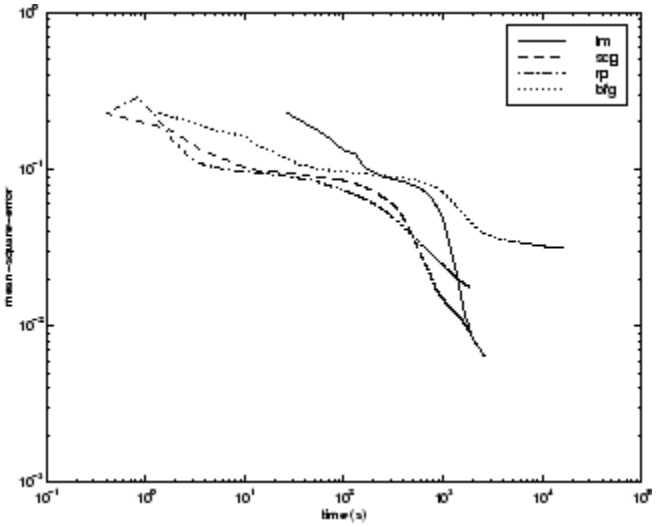
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance test, etc.). The data was obtained from the University of California, Irvine, machine learning data base. The network used for this problem is an 8-15-15-2 network with tansig neurons in all layers. The following table summarizes the results of training this network with the nine different algorithms. Each entry in the table represents 10 different trials, where different random initial

weights are used in each trial. In each case, the network is trained until the squared error is less than 0.05.

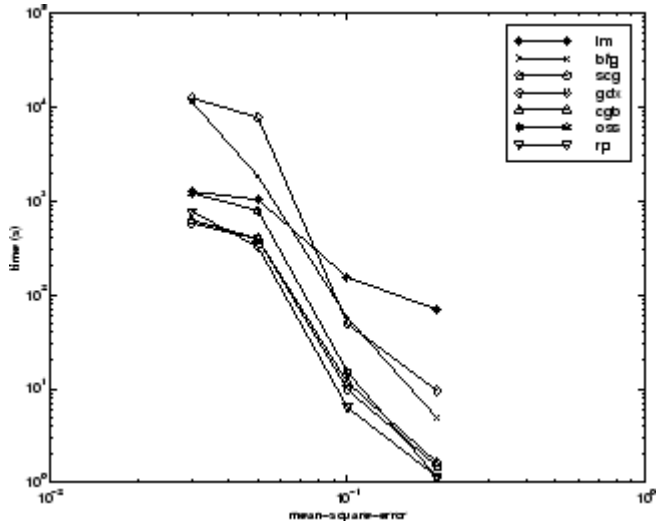
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems considered. The RP algorithm works well on all the pattern recognition problems. This is reasonable, because that algorithm was designed to overcome the difficulties caused by training with sigmoid functions, which have very small slopes when operating far from the center point. For pattern recognition problems, you use sigmoid transfer functions in the output layer, and you want the network to operate at the tails of the sigmoid function.

Algorithm	Mean Time (s)	Ratio	Min. Time (s)	Max. Time (s)	Std. (s)
RP	323.90	1.00	187.43	576.90	111.37
SCG	390.53	1.21	267.99	487.17	75.07
CGB	394.67	1.22	312.25	558.21	85.38
CGP	415.90	1.28	320.62	614.62	94.77
OSS	784.00	2.42	706.89	936.52	76.37
CGF	784.50	2.42	629.42	1082.20	144.63
LM	1028.10	3.17	802.01	1269.50	166.31
BFG	1821.00	5.62	1415.80	3254.50	546.36
GDX	7687.00	23.73	5169.20	10350.00	2015.00

The following figure plots the mean square error versus execution time for some typical algorithms. As with other problems, you see that the SCG and RP have fast initial convergence, while the LM algorithm is able to provide smaller final error.



The relationship between the algorithms is further illustrated in the following figure, which plots the time required to converge versus the mean square error convergence goal. In this case, you can see that the BFG algorithm degrades as the error goal is reduced, while the LM algorithm improves. The RP algorithm is best, except at the smallest error goal, where SCG is better.



Summary

There are several algorithm characteristics that can be deduced from the experiments described. In general, on function approximation problems, for networks that contain up to a few hundred weights, the Levenberg-Marquardt algorithm will have the fastest convergence. This advantage is especially noticeable if very accurate training is required. In many cases, `trainlm` is able to obtain lower mean square errors than any of the other algorithms tested. However, as the number of weights in the network increases, the advantage of `trainlm` decreases. In addition, `trainlm` performance is relatively poor on pattern recognition problems. The storage requirements of `trainlm` are larger than the other algorithms tested.

The `trainrp` function is the fastest algorithm on pattern recognition problems. However, it does not perform well on function approximation problems. Its performance also degrades as the error goal is reduced. The memory requirements for this algorithm are relatively small in comparison to the other algorithms considered.

The conjugate gradient algorithms, in particular `trainscg`, seem to perform well over a wide variety of problems, particularly for networks with a large number of weights. The SCG algorithm is almost as fast as the LM algorithm on function approximation problems (faster for large networks) and is almost as fast as `trainrp` on pattern recognition problems. Its performance does not degrade as quickly as `trainrp` performance does when the error is reduced. The conjugate gradient algorithms have relatively modest memory requirements.

The performance of `trainbfg` is similar to that of `trainlm`. It does not require as much storage as `trainlm`, but the computation required does increase geometrically with the size of the network, because the equivalent of a matrix inverse must be computed at each iteration.

The variable learning rate algorithm `traingdx` is usually much slower than the other methods, and has about the same storage requirements as `trainrp`, but it can still be useful for some problems. There are certain situations in which it is better to converge more slowly. For example, when using early stopping you can have inconsistent results if you use an algorithm that converges too quickly. You might overshoot the point at which the error on the validation set is minimized.

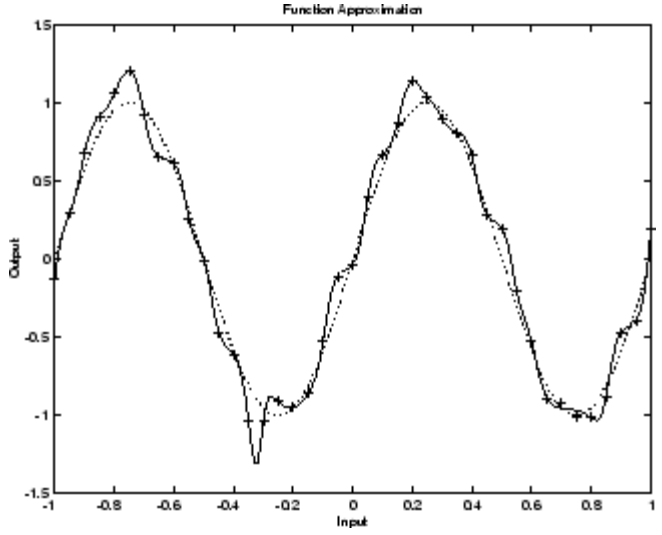
Improve Shallow Neural Network Generalization and Avoid Overfitting

In this section...

Tip To learn how to set up parameters for a deep learning network, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-42.

One of the problems that occur during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations.

The following figure shows the response of a 1-20-1 neural network that has been trained to approximate a noisy sine function. The underlying sine function is shown by the dotted line, the noisy measurements are given by the + symbols, and the neural network response is given by the solid line. Clearly this network has overfitted the data and will not generalize well.



One method for improving network generalization is to use a network that is just large enough to provide an adequate fit. The larger network you use, the more complex the functions the network can create. If you use a small enough network, it will not have enough power to overfit the data. Run the *Neural Network Design* example nnd11gn [HDB96 on page 32-2] to investigate how reducing the size of a network can prevent overfitting.

Unfortunately, it is difficult to know beforehand how large a network should be for a specific application. There are two other methods for improving generalization that are implemented in Deep Learning Toolbox software: regularization and early stopping. The next sections describe these two techniques and the routines to implement them.

Note that if the number of parameters in the network is much smaller than the total number of points in the training set, then there is little or no chance of overfitting. If you can easily collect more data and increase the size of the training set, then there is no need to worry about the following techniques to prevent overfitting. The rest of this section only applies to those situations in which you want to make the most of a limited supply of data.

Retraining Neural Networks

Typically each backpropagation training session starts with different initial weights and biases, and different divisions of data into training, validation, and test sets. These different conditions can lead to very different solutions for the same problem.

It is a good idea to train several networks to ensure that a network with good generalization is found.

Here a dataset is loaded and divided into two parts: 90% for designing networks and 10% for testing them all.

```
[x, t] = bodyfat_dataset;
Q = size(x, 2);
Q1 = floor(Q * 0.90);
Q2 = Q - Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1 + (1:Q2));
x1 = x(:, ind1);
t1 = t(:, ind1);
x2 = x(:, ind2);
t2 = t(:, ind2);
```

Next a network architecture is chosen and trained ten times on the first part of the dataset, with each network's mean square error on the second part of the dataset.

```
net = feedforwardnet(10);
numNN = 10;
NN = cell(1, numNN);
perfs = zeros(1, numNN);
for i = 1:numNN
  fprintf('Training %d/%d\n', i, numNN);
  NN{i} = train(net, x1, t1);
  y2 = NN{i}(x2);
  perfs(i) = mse(net, t2, y2);
end
```

Each network will be trained starting from different initial weights and biases, and with a different division of the first dataset into training, validation, and test sets. Note that the test sets are a good

measure of generalization for each respective network, but not for all the networks, because data that is a test set for one network will likely be used for training or validation by other neural networks. This is why the original dataset was divided into two parts, to ensure that a completely independent test set is preserved.

The neural network with the lowest performance is the one that generalized best to the second part of the dataset.

Multiple Neural Networks

Another simple way to improve generalization, especially when caused by noisy data or a small dataset, is to train multiple neural networks and average their outputs.

For instance, here 10 neural networks are trained on a small problem and their mean squared errors compared to the means squared error of their average.

First, the dataset is loaded and divided into a design and test set.

```
[x, t] = bodyfat_dataset;
Q = size(x, 2);
Q1 = floor(Q * 0.90);
Q2 = Q - Q1;
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1 + (1:Q2));
x1 = x(:, ind1);
t1 = t(:, ind1);
x2 = x(:, ind2);
t2 = t(:, ind2);
```

Then, ten neural networks are trained.

```
net = feedforwardnet(10);
numNN = 10;
nets = cell(1, numNN);
for i = 1:numNN
  fprintf('Training %d/%d\n', i, numNN)
  nets{i} = train(net, x1, t1);
end
```

Next, each network is tested on the second dataset with both individual performances and the performance for the average output calculated.

```
perfs = zeros(1, numNN);
y2Total = 0;
for i = 1:numNN
  neti = nets{i};
  y2 = neti(x2);
  perfs(i) = mse(neti, t2, y2);
  y2Total = y2Total + y2;
end
perfs
y2AverageOutput = y2Total / numNN;
perfAveragedOutputs = mse(nets{1}, t2, y2AverageOutput)
```

The mean squared error for the average output is likely to be lower than most of the individual performances, perhaps not all. It is likely to generalize better to additional new data.

For some very difficult problems, a hundred networks can be trained and the average of their outputs taken for any input. This is especially helpful for a small, noisy dataset in conjunction with the Bayesian Regularization training function `trainbr`, described below.

Early Stopping

The default method for improving generalization is called *early stopping*. This technique is automatically provided for all of the supervised network creation functions, including the backpropagation network creation functions such as `feedforwardnet`.

In this technique the available data is divided into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. When the validation error increases for a specified number of iterations (`net.trainParam.max_fail`), the training is stopped, and the weights and biases at the minimum of the validation error are returned.

The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error in the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

There are four functions provided for dividing data into training, validation and test sets. They are `dividerand` (the default), `divideblock`, `divideint`, and `divideind`. You can access or change the division function for your network with this property:

```
net.divideFcn
```

Each of these functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property:

```
net.divideParam
```

Index Data Division (divideind)

Create a simple test problem. For the full data set, generate a noisy sine wave with 201 input points ranging from −1 to 1 at steps of 0.01:

```
p = [-1:0.01:1];
t = sin(2*pi*p)+0.1*randn(size(p));
```

Divide the data by index so that successive samples are assigned to the training set, validation set, and test set successively:

```
trainInd = 1:3:201
valInd = 2:3:201;
testInd = 3:3:201;
[trainP,valP,testP] = divideind(p,trainInd,valInd,testInd);
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Random Data Division (dividerand)

You can divide the input data randomly so that 60% of the samples are assigned to the training set, 20% to the validation set, and 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = dividerand(p);
```

This function not only divides the input data, but also returns indices so that you can divide the target data accordingly using `divideind`:

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Block Data Division (divideblock)

You can also divide the input data randomly such that the first 60% of the samples are assigned to the training set, the next 20% to the validation set, and the last 20% to the test set, as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = divideblock(p);
```

Divide the target data accordingly using `divideind`:

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Interleaved Data Division (divideint)

Another way to divide the input data is to cycle samples between the training set, validation set, and test set according to percentages. You can interleave 60% of the samples to the training set, 20% to the validation set and 20% to the test set as follows:

```
[trainP,valP,testP,trainInd,valInd,testInd] = divideint(p);
```

Divide the target data accordingly using `divideind`.

```
[trainT,valT,testT] = divideind(t,trainInd,valInd,testInd);
```

Regularization

Another method for improving generalization is called regularization. This involves modifying the performance function, which is normally chosen to be the sum of squares of the network errors on the training set. The next section explains how the performance function can be modified, and the following section describes a routine that automatically sets the optimal performance function to achieve the best generalization.

Modified Performance Function

The typical performance function used for training feedforward neural networks is the mean sum of squares of the network errors.

$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

It is possible to improve generalization if you modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights and biases
$msereg = \gamma * msw + (1 - \gamma) * mse$, where γ is the performance ratio, and

$$msw = \frac{1}{n}\sum_{j=1}^{n} w_j^2$$

Using this performance function causes the network to have smaller weights and biases, and this forces the network response to be smoother and less likely to overfit.

The following code reinitializes the previous network and retrains it using the BFGS algorithm with the regularized performance function. Here the performance ratio is set to 0.5, which gives equal weight to the mean square errors and the mean square weights.

```
[x,t] = simplefit_dataset;
net = feedforwardnet(10,'trainbfg');
net.divideFcn = '';
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
net.performParam.regularization = 0.5;
net = train(net,x,t);
```

The problem with regularization is that it is difficult to determine the optimum value for the performance ratio parameter. If you make this parameter too large, you might get overfitting. If the ratio is too small, the network does not adequately fit the training data. The next section describes a routine that automatically sets the regularization parameters.

Automated Regularization (trainbr)

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach to this process is the Bayesian framework of David MacKay [MacK92 on page 32-2]. In this framework, the weights and biases of the network are assumed to be random variables with specified distributions. The regularization parameters are related to the unknown variances associated with these distributions. You can then estimate these parameters using statistical techniques.

A detailed discussion of Bayesian regularization is beyond the scope of this user guide. A detailed discussion of the use of Bayesian regularization, in combination with Levenberg-Marquardt training, can be found in [FoHa97 on page 32-2].

Bayesian regularization has been implemented in the function `trainbr`. The following code shows how you can train a 1-20-1 network using this function to approximate the noisy sine wave shown in the figure in “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 28-25. (Data division is cancelled by setting `net.divideFcn` so that the effects of `trainbr` are isolated from early stopping.)

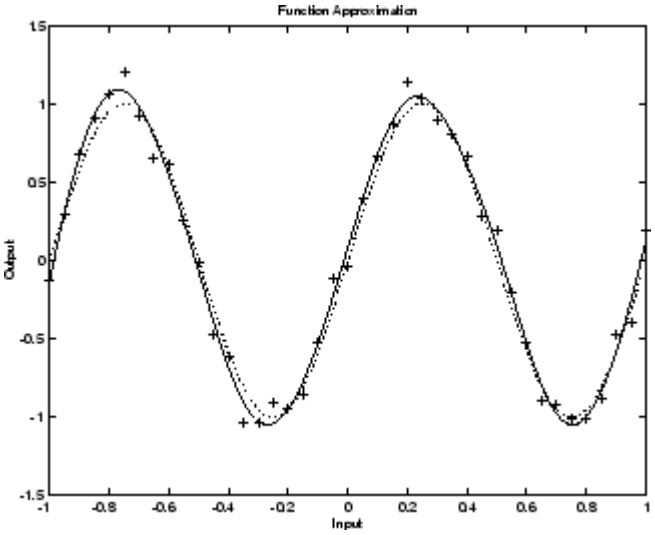
```
x = -1:0.05:1;
t = sin(2*pi*x) + 0.1*randn(size(x));
net = feedforwardnet(20,'trainbr');
net = train(net,x,t);
```

One feature of this algorithm is that it provides a measure of how many network parameters (weights and biases) are being effectively used by the network. In this case, the final trained network uses approximately 12 parameters (indicated by `#Par` in the printout) out of the 61 total weights and biases in the 1-20-1 network. This effective number of parameters should remain approximately the same, no matter how large the number of parameters in the network becomes. (This assumes that the network has been trained for a sufficient number of iterations to ensure convergence.)

The `trainbr` algorithm generally works best when the network inputs and targets are scaled so that they fall approximately in the range [−1,1]. That is the case for the test problem here. If your inputs and targets do not fall in this range, you can use the function `mapminmax` or `mapstd` to perform the scaling, as described in “Choose Neural Network Input-Output Processing Functions” on page 22-7. Networks created with `feedforwardnet` include `mapminmax` as an input and output processing function by default.

The following figure shows the response of the trained network. In contrast to the previous figure, in which a 1-20-1 network overfits the data, here you see that the network response is very close to the underlying sine function (dotted line), and, therefore, the network will generalize well to new inputs. You could have tried an even larger network, but the network response would never overfit the data. This eliminates the guesswork required in determining the optimum network size.

When using `trainbr`, it is important to let the algorithm run until the effective number of parameters has converged. The training might stop with the message "Maximum MU reached." This is typical, and is a good indication that the algorithm has truly converged. You can also tell that the algorithm has converged if the sum squared error (SSE) and sum squared weights (SSW) are relatively constant over several iterations. When this occurs you might want to click the **Stop Training** button in the training window.



Summary and Discussion of Early Stopping and Regularization

Early stopping and regularization can ensure network generalization when you apply them properly.

For early stopping, you must be careful not to use an algorithm that converges too rapidly. If you are using a fast algorithm (like `trainlm`), set the training parameters so that the convergence is relatively slow. For example, set `mu` to a relatively large value, such as 1, and set `mu_dec` and `mu_inc` to values close to 1, such as 0.8 and 1.5, respectively. The training functions `trainscg` and `trainbr` usually work well with early stopping.

With early stopping, the choice of the validation set is also important. The validation set should be representative of all points in the training set.

When you use Bayesian regularization, it is important to train the network until it reaches convergence. The sum-squared error, the sum-squared weights, and the effective number of parameters should reach constant values when the network has converged.

With both early stopping and regularization, it is a good idea to train the network starting from several different initial conditions. It is possible for either method to fail in certain circumstances. By testing several different initial conditions, you can verify robust network performance.

When the data set is small and you are training function approximation networks, Bayesian regularization provides better generalization performance than early stopping. This is because Bayesian regularization does not require that a validation data set be separate from the training data set; it uses all the data.

To provide some insight into the performance of the algorithms, both early stopping and Bayesian regularization were tested on several benchmark data sets, which are listed in the following table.

Data Set Title	Number of Points	Network	Description
BALL	67	2-10-1	Dual-sensor calibration for a ball position measurement
SINE (5% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 5% level
SINE (2% N)	41	1-15-1	Single-cycle sine wave with Gaussian noise at 2% level
ENGINE (ALL)	1199	2-30-2	Engine sensor—full data set
ENGINE (1/4)	300	2-30-2	Engine sensor—1/4 of data set
CHOLEST (ALL)	264	5-15-3	Cholesterol measurement—full data set
CHOLEST (1/2)	132	5-15-3	Cholesterol measurement—1/2 data set

These data sets are of various sizes, with different numbers of inputs and targets. With two of the data sets the networks were trained once using all the data and then retrained using only a fraction of the data. This illustrates how the advantage of Bayesian regularization becomes more noticeable when the data sets are smaller. All the data sets are obtained from physical systems except for the SINE data sets. These two were artificially created by adding various levels of noise to a single cycle of a sine wave. The performance of the algorithms on these two data sets illustrates the effect of noise.

The following table summarizes the performance of early stopping (ES) and Bayesian regularization (BR) on the seven test sets. (The `trainscg` algorithm was used for the early stopping tests. Other algorithms provide similar performance.)

Mean Squared Test Set Error

Method	Ball	Engine (All)	Engine (1/4)	Choles (All)	Choles (1/2)	Sine (5% N)	Sine (2% N)
ES	1.2e-1	1.3e-2	1.9e-2	1.2e-1	1.4e-1	1.7e-1	1.3e-1
BR	1.3e-3	2.6e-3	4.7e-3	1.2e-1	9.3e-2	3.0e-2	6.3e-3
ES/BR	92	5	4	1	1.5	5.7	21

You can see that Bayesian regularization performs better than early stopping in most cases. The performance improvement is most noticeable when the data set is small, or if there is little noise in the data set. The BALL data set, for example, was obtained from sensors that had very little noise.

Although the generalization performance of Bayesian regularization is often better than early stopping, this is not always the case. In addition, the form of Bayesian regularization implemented in the toolbox does not perform as well on pattern recognition problems as it does on function approximation problems. This is because the approximation to the Hessian that is used in the

Levenberg-Marquardt algorithm is not as accurate when the network output is saturated, as would be the case in pattern recognition problems. Another disadvantage of the Bayesian regularization method is that it generally takes longer to converge than early stopping.

Posttraining Analysis (regression)

The performance of a trained network can be measured to some extent by the errors on the training, validation, and test sets, but it is often useful to investigate the network response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets. The routine `regression` is designed to perform this analysis.

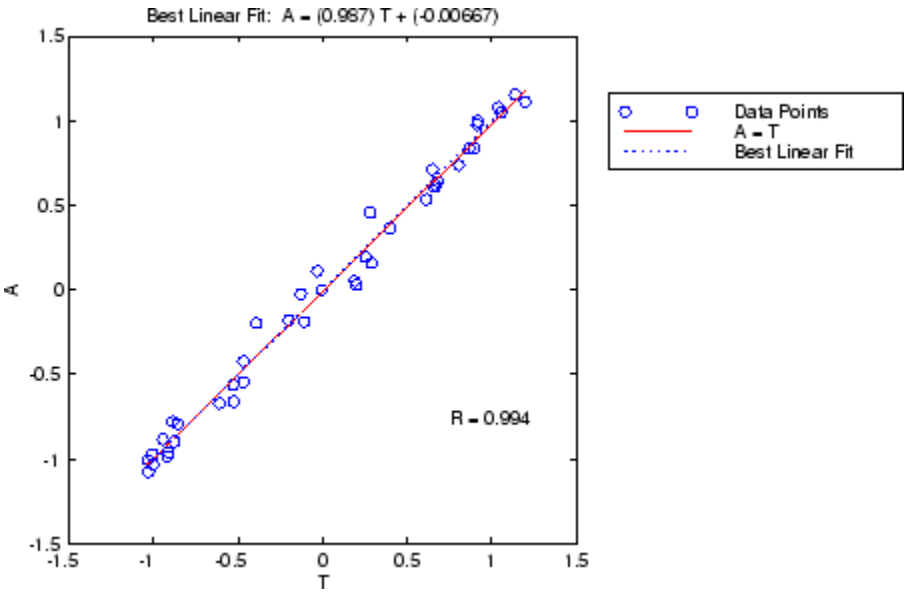
The following commands illustrate how to perform a regression analysis on a network trained.

```
x = [-1:.05:1];
t = sin(2*pi*x)+0.1*randn(size(x));
net = feedforwardnet(10);
net = train(net,x,t);
y = net(x);
[r,m,b] = regression(t,y)
```

```
r =
    0.9935
m =
    0.9874
b =
   -0.0067
```

The network output and the corresponding targets are passed to `regression`. It returns three parameters. The first two, `m` and `b`, correspond to the slope and the *y*-intercept of the best linear regression relating targets to network outputs. If there were a perfect fit (outputs exactly equal to targets), the slope would be 1, and the *y*-intercept would be 0. In this example, you can see that the numbers are very close. The third variable returned by `regression` is the correlation coefficient (R-value) between the outputs and targets. It is a measure of how well the variation in the output is explained by the targets. If this number is equal to 1, then there is perfect correlation between targets and outputs. In the example, the number is very close to 1, which indicates a good fit.

The following figure illustrates the graphical output provided by `regression`. The network outputs are plotted versus the targets as open circles. The best linear fit is indicated by a dashed line. The perfect fit (output equal to targets) is indicated by the solid line. In this example, it is difficult to distinguish the best linear fit line from the perfect fit line because the fit is so good.


Best Linear Fit: A = (0.987) T + (-0.00667)
R = 0.994
A
T
Data Points
A = T
Best Linear Fit

Edit Shallow Neural Network Properties

In this section...

Tip To learn how to define your own layers for deep learning networks, see “Define Custom Deep Learning Layers” on page 18-9.

Deep Learning Toolbox software provides a flexible network object type that allows many kinds of networks to be created and then used with functions such as `init`, `sim`, and `train`.

Type the following to see all the network creation functions in the toolbox.

```
help nnnetwork
```

This flexibility is possible because networks have an object-oriented representation. The representation allows you to define various architectures and assign various algorithms to those architectures.

To create custom networks, start with an empty network (obtained with the `network` function) and set its properties as desired.

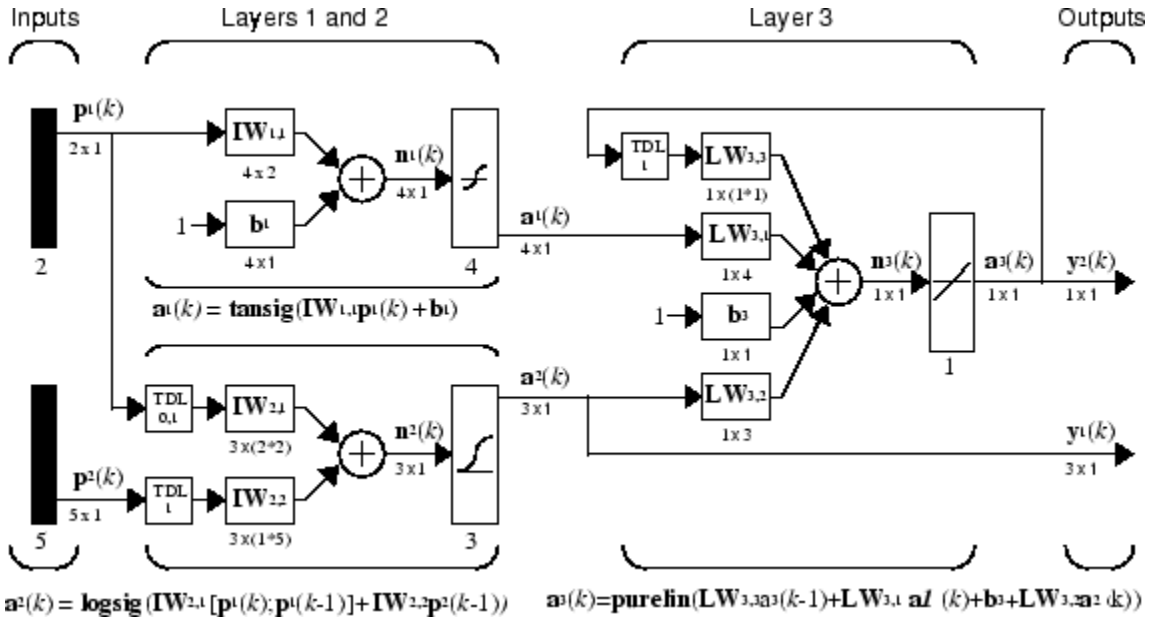
```
net = network
```

The network object consists of many properties that you can set to specify the structure and behavior of your network.

The following sections show how to create a custom network by using these properties.

Custom Network

Before you can build a network you need to know what it looks like. For dramatic purposes (and to give the toolbox a workout) this section leads you through the creation of the wild and complicated network shown below.



Each of the two elements of the first network input is to accept values ranging between 0 and 10. Each of the five elements of the second network input ranges from −2 to 2.

Before you can complete your design of this network, the algorithms it employs for initialization and training must be specified.

Each layer's weights and biases are initialized with the Nguyen-Widrow layer initialization method (`initnw`). The network is trained with Levenberg-Marquardt backpropagation (`trainlm`), so that, given example input vectors, the outputs of the third layer learn to match the associated target vectors with minimal mean squared error (`mse`).

Network Definition

The first step is to create a new network. Type the following code to create a network and view its many properties:

```
net = network
```

Architecture Properties

The first group of properties displayed is labeled `architecture` properties. These properties allow you to select the number of inputs and layers and their connections.

Number of Inputs and Layers

The first two properties displayed in the dimensions group are `numInputs` and `numLayers`. These properties allow you to select how many inputs and layers you want the network to have.

```
net =

    dimensions:
         numInputs: 0
         numLayers: 0
         ...
```

Note that the network has no inputs or layers at this time.

Change that by setting these properties to the number of inputs and number of layers in the custom network diagram.

```
net.numInputs = 2;
net.numLayers = 3;
```

`net.numInputs` is the number of input sources, not the number of elements in an input vector (`net.inputs{i}.size`).

Bias Connections

Type `net` and press **Enter** to view its properties again. The network now has two inputs and three layers.

```
net =
    Neural Network:
    dimensions:
         numInputs: 2
         numLayers: 3
```

Examine the next four properties in the connections group:

```
       biasConnect: [0; 0; 0]
      inputConnect: [0 0; 0 0; 0 0]
      layerConnect: [0 0 0; 0 0 0; 0 0 0]
     outputConnect: [0 0 0]
```

These matrices of 1s and 0s represent the presence and absence of bias, input weight, layer weight, and output connections. They are currently all zeros, indicating that the network does not have any such connections.

The bias connection matrix is a 3-by-1 vector. To create a bias connection to the *i*th layer you can set `net.biasConnect(i)` to `1`. Specify that the first and third layers are to have bias connections, as the diagram indicates, by typing the following code:

```
net.biasConnect(1) = 1;
net.biasConnect(3) = 1;
```

You could also define those connections with a single line of code.

```
net.biasConnect = [1; 0; 1];
```

Input and Layer Weight Connections

The input connection matrix is 3-by-2, representing the presence of connections from two sources (the two inputs) to three destinations (the three layers). Thus, `net.inputConnect(i,j)` represents the presence of an input weight connection going to the *i*th layer from the *j*th input.

To connect the first input to the first and second layers, and the second input to the second layer (as indicated by the custom network diagram), type

```
net.inputConnect(1,1) = 1;
net.inputConnect(2,1) = 1;
net.inputConnect(2,2) = 1;
```

or this single line of code:

```
net.inputConnect = [1 0; 1 1; 0 0];
```

Similarly, `net.layerConnect(i.j)` represents the presence of a layer-weight connection going to the *i*th layer from the *j*th layer. Connect layers 1, 2, and 3 to layer 3 as follows:

```
net.layerConnect = [0 0 0; 0 0 0; 1 1 1];
```

Output Connections

The output connections are a 1-by-3 matrix, indicating that they connect to one destination (the external world) from three sources (the three layers).

To connect layers 2 and 3 to the network output, type

```
net.outputConnect = [0 1 1];
```

Number of Outputs

Type `net` and press **Enter** to view the updated properties. The final three architecture properties are read-only values, which means their values are determined by the choices made for other properties. The first read-only property in the dimension group is the number of outputs:

```
numOutputs: 2
```

By defining output connection from layers 2 and 3, you specified that the network has two outputs.

Subobject Properties

The next group of properties in the output display is subobjects:

```
subobjects:
            inputs: {2x1 cell array of 2 inputs}
            layers: {3x1 cell array of 3 layers}
           outputs: {1x3 cell array of 2 outputs}
            biases: {3x1 cell array of 2 biases}
      inputWeights: {3x2 cell array of 3 weights}
      layerWeights: {3x3 cell array of 3 weights}
```

Inputs

When you set the number of inputs (`net.numInputs`) to 2, the `inputs` property becomes a cell array of two input structures. Each *i*th input structure (`net.inputs{i}`) contains additional properties associated with the *i*th input.

To see how the input structures are arranged, type

```
net.inputs
ans =
    [1x1 nnetInput]
    [1x1 nnetInput]
```

To see the properties associated with the first input, type

```
net.inputs{1}
```

The properties appear as follows:

```
ans =
              name: 'Input'
```

```
    feedbackOutput: []
       processFcns: {}
     processParams: {1x0 cell array of 0 params}
   processSettings: {0x0 cell array of 0 settings}
    processedRange: []
     processedSize: 0
             range: []
              size: 0
          userdata: (your custom info)
```

If you set the `exampleInput` property, the `range`, `size`, `processedSize`, and `processedRange` properties will automatically be updated to match the properties of the value of `exampleInput`.

Set the `exampleInput` property as follows:

```
net.inputs{1}.exampleInput = [0 10 5; 0 3 10];
```

If you examine the structure of the first input again, you see that it now has new values.

The property `processFcns` can be set to one or more processing functions. Type `help nnprocess` to see a list of these functions.

Set the second input vector ranges to be from −2 to 2 for five elements as follows:

```
net.inputs{1}.processFcns = {'removeconstantrows','mapminmax'};
```

View the new input properties. You will see that `processParams`, `processSettings`, `processedRange` and `processedSize` have all been updated to reflect that inputs will be processed using `removeconstantrows` and `mapminmax` before being given to the network when the network is simulated or trained. The property `processParams` contains the default parameters for each processing function. You can alter these values, if you like. See the reference page for each processing function to learn more about their parameters.

You can set the size of an input directly when no processing functions are used:

```
net.inputs{2}.size = 5;
```

Layers

When you set the number of layers (`net.numLayers`) to 3, the `layers` property becomes a cell array of three-layer structures. Type the following line of code to see the properties associated with the first layer.

```
net.layers{1}
ans =
    Neural Network Layer

              name: 'Layer'
        dimensions: 0
       distanceFcn: (none)
     distanceParam: (none)
         distances: []
            initFcn: 'initwb'
       netInputFcn: 'netsum'
     netInputParam: (none)
         positions: []
             range: []
              size: 0
```

```
       topologyFcn: (none)
       transferFcn: 'purelin'
     transferParam: (none)
          userdata: (your custom info)
```

Type the following three lines of code to change the first layer's size to 4 neurons, its transfer function to `tansig`, and its initialization function to the Nguyen-Widrow function, as required for the custom network diagram.

```
net.layers{1}.size = 4;
net.layers{1}.transferFcn = 'tansig';
net.layers{1}.initFcn = 'initnw';
```

The second layer is to have three neurons, the `logsig` transfer function, and be initialized with `initnw`. Set the second layer's properties to the desired values as follows:

```
net.layers{2}.size = 3;
net.layers{2}.transferFcn = 'logsig';
net.layers{2}.initFcn = 'initnw';
```

The third layer's size and transfer function properties don't need to be changed, because the defaults match those shown in the network diagram. You need to set only its initialization function, as follows:

```
net.layers{3}.initFcn = 'initnw';
```

Outputs

Use this line of code to see how the `outputs` property is arranged:

```
net.outputs
ans =
    []    [1x1 nnetOutput]    [1x1 nnetOutput]
```

Note that `outputs` contains two output structures, one for layer 2 and one for layer 3. This arrangement occurs automatically when `net.outputConnect` is set to `[0 1 1]`.

View the second layer's output structure with the following expression:

```
net.outputs{2}
ans =
    Neural Network Output

              name: 'Output'
     feedbackInput: []
     feedbackDelay: 0
      feedbackMode: 'none'
       processFcns: {}
     processParams: {1x0 cell array of 0 params}
   processSettings: {0x0 cell array of 0 settings}
    processedRange: [3x2 double]
     processedSize: 3
             range: [3x2 double]
              size: 3
          userdata: (your custom info)
```

The `size` is automatically set to 3 when the second layer's size (`net.layers{2}.size`) is set to that value. Look at the third layer's output structure if you want to verify that it also has the correct `size`.

Outputs have processing properties that are automatically applied to target values before they are used by the network during training. The same processing settings are applied in reverse on layer output values before they are returned as network output values during network simulation or training.

Similar to input-processing properties, setting the `exampleOutput` property automatically causes `size`, `range`, `processedSize`, and `processedRange` to be updated. Setting `processFcns` to a cell array list of processing function names causes `processParams`, `processSettings`, `processedRange` to be updated. You can then alter the `processParam` values, if you want to.

Biases, Input Weights, and Layer Weights

Enter the following commands to see how bias and weight structures are arranged:

```
net.biases
net.inputWeights
net.layerWeights
```

Here are the results of typing `net.biases`:

```
ans =
    [1x1 nnetBias]
    []
    [1x1 nnetBias]
```

Each contains a structure where the corresponding connections (`net.biasConnect`, `net.inputConnect`, and `net.layerConnect`) contain a 1.

Look at their structures with these lines of code:

```
net.biases{1}
net.biases{3}
net.inputWeights{1,1}
net.inputWeights{2,1}
net.inputWeights{2,2}
net.layerWeights{3,1}
net.layerWeights{3,2}
net.layerWeights{3,3}
```

For example, typing `net.biases{1}` results in the following output:

```
    initFcn: (none)
      learn: true
   learnFcn: (none)
 learnParam: (none)
       size: 4
   userdata: (your custom info)
```

Specify the weights' tap delay lines in accordance with the network diagram by setting each weight's `delays` property:

```
net.inputWeights{2,1}.delays = [0 1];
net.inputWeights{2,2}.delays = 1;
net.layerWeights{3,3}.delays = 1;
```

Network Functions

Type `net` and press **Return** again to see the next set of properties.

```
functions:
      adaptFcn: (none)
    adaptParam: (none)
      derivFcn: 'defaultderiv'
     divideFcn: (none)
   divideParam: (none)
    divideMode: 'sample'
       initFcn: 'initlay'
    performFcn: 'mse'
  performParam: .regularization, .normalization
      plotFcns: {}
    plotParams: {1x0 cell array of 0 params}
      trainFcn: (none)
    trainParam: (none)
```

Each of these properties defines a function for a basic network operation.

Set the initialization function to `initlay` so the network initializes itself according to the layer initialization functions already set to `initnw`, the Nguyen-Widrow initialization function.

```
net.initFcn = 'initlay';
```

This meets the initialization requirement of the network.

Set the performance function to `mse` (mean squared error) and the training function to `trainlm` (Levenberg-Marquardt backpropagation) to meet the final requirement of the custom network.

```
net.performFcn = 'mse';
net.trainFcn = 'trainlm';
```

Set the divide function to `dividerand` (divide training data randomly).

```
net.divideFcn = 'dividerand';
```

During supervised training, the input and target data are randomly divided into training, test, and validation data sets. The network is trained on the training data until its performance begins to decrease on the validation data, which signals that generalization has peaked. The test data provides a completely independent test of network generalization.

Set the plot functions to `plotperform` (plot training, validation and test performance) and `plottrainstate` (plot the state of the training algorithm with respect to epochs).

```
net.plotFcns = {'plotperform','plottrainstate'};
```

Weight and Bias Values

Before initializing and training the network, type `net` and press **Return**, then look at the weight and bias group of network properties.

```
weight and bias values:
           IW: {3x2 cell} containing 3 input weight matrices
           LW: {3x3 cell} containing 3 layer weight matrices
            b: {3x1 cell} containing 2 bias vectors
```

These cell arrays contain weight matrices and bias vectors in the same positions that the connection properties (`net.inputConnect`, `net.layerConnect`, `net.biasConnect`) contain 1s and the subobject properties (`net.inputWeights`, `net.layerWeights`, `net.biases`) contain structures.

Evaluating each of the following lines of code reveals that all the bias vectors and weight matrices are set to zeros.

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

Each input weight `net.IW{i,j}`, layer weight `net.LW{i,j}`, and bias vector `net.b{i}` has as many rows as the size of the *i*th layer (`net.layers{i}.size`).

Each input weight `net.IW{i,j}` has as many columns as the size of the *j*th input (`net.inputs{j}.size`) multiplied by the number of its delay values (`length(net.inputWeights{i,j}.delays)`).

Likewise, each layer weight has as many columns as the size of the *j*th layer (`net.layers{j}.size`) multiplied by the number of its delay values (`length(net.layerWeights{i,j}.delays)`).

Network Behavior

Initialization

Initialize your network with the following line of code:

```
net = init(net);
```

Check the network's biases and weights again to see how they have changed:

```
net.IW{1,1}, net.IW{2,1}, net.IW{2,2}
net.LW{3,1}, net.LW{3,2}, net.LW{3,3}
net.b{1}, net.b{3}
```

For example,

```
net.IW{1,1}
ans =
   -0.3040    0.4703
   -0.5423   -0.1395
    0.5567    0.0604
    0.2667    0.4924
```

Training

Define the following cell array of two input vectors (one with two elements, one with five) for two time steps (i.e., two columns).

```
X = {[0; 0] [2; 0.5]; [2; -2; 1; 0; 1] [-1; -1; 1; 0; 1]};
```

You want the network to respond with the following target sequences for the second layer, which has three neurons, and the third layer with one neuron:

```
T = {[1; 1; 1] [0; 0; 0]; 1 -1};
```

Before training, you can simulate the network to see whether the initial network's response `Y` is close to the target `T`.

```
Y = sim(net,X)
Y =
```

```
    [3x1 double]    [3x1 double]
    [      1.7148]    [      2.2726]
```

The cell array Y is the output sequence of the network, which is also the output sequence of the second and third layers. The values you got for the second row can differ from those shown because of different initial weights and biases. However, they will almost certainly not be equal to targets T, which is also true of the values shown.

The next task is optional. On some occasions you may wish to alter the training parameters before training. The following line of code displays the default Levenberg-Marquardt training parameters (defined when you set `net.trainFcn` to `trainlm`).

```
net.trainParam
```

The following properties should be displayed.

```
ans =
    Show Training Window Feedback   showWindow: true
    Show Command Line Feedback showCommandLine: false
    Command Line Frequency                show: 25
    Maximum Epochs                      epochs: 1000
    Maximum Training Time                 time: Inf
    Performance Goal                      goal: 0
    Minimum Gradient                   min_grad: 1e-07
    Maximum Validation Checks          max_fail: 6
    Mu                                       mu: 0.001
    Mu Decrease Ratio                   mu_dec: 0.1
    Mu Increase Ratio                   mu_inc: 10
    Maximum mu                          mu_max: 10000000000
```

You will not often need to modify these values. See the documentation for the training function for information about what each of these means. They have been initialized with default values that work well for a large range of problems, so there is no need to change them here.

Next, train the network with the following call:

```
net = train(net,X,T);
```

Training launches the neural network training window. To open the performance and training state plots, click the plot buttons.

After training, you can simulate the network to see if it has learned to respond correctly:

```
Y = sim(net,X)

    [3x1 double]    [3x1 double]
    [      1.0000]    [     -1.0000]
```

The second network output (i.e., the second row of the cell array Y), which is also the third layer's output, matches the target sequence T.

Custom Neural Network Helper Functions

The toolbox allows you to create and use your own custom functions. This gives you a great deal of control over the algorithms used to initialize, simulate, and train your networks.

Be aware, however, that custom functions may need updating to remain compatible with future versions of the software. Backward compatibility of custom functions cannot be guaranteed.

Template functions are available for you to copy, rename and customize, to create your own versions of these kinds of functions. You can see the list of all template functions by typing the following:

```
help nncustom
```

Each template is a simple version of a different type of function that you can use with your own custom networks.

For instance, make a copy of the file `tansig.m` with the new name `mytransfer.m`. Start editing the new file by changing the function name a the top from `tansig` to `mytransfer`.

You can now edit each of the sections of code that make up a transfer function, using the help comments in each of those sections to guide you.

Once you are done, store the new function in your working folder, and assign the name of your transfer function to the `transferFcn` property of any layer of any network object to put it to use.

Automatically Save Checkpoints During Neural Network Training

During neural network training, intermediate results can be periodically saved to a MAT file for recovery if the computer fails or you kill the training process. This helps protect the value of long training runs, which if interrupted would need to be completely restarted otherwise. This feature is especially useful for long parallel training sessions, which are more likely to be interrupted by computing resource failures.

Checkpoint saves are enabled with the optional `'CheckpointFile'` training argument followed by the checkpoint file name or path. If you specify only a file name, the file is placed in the working directory by default. The file must have the `.mat` file extension, but if this is not specified it is automatically appended. In this example, checkpoint saves are made to the file called `MyCheckpoint.mat` in the current working directory.

```
[x,t] = bodyfat_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t,'CheckpointFile','MyCheckpoint.mat');
```

```
22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the previous short training example, this results in only two checkpoint saves: one at the beginning and one at the end of training.

The optional training argument `'CheckpointDelay'` can change the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);
```

```
22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, you can reload the checkpoint structure containing the best neural network obtained before the interruption, and the training record. In this case, the `stage` field value is `'Final'`, indicating the last save was at the final epoch because training completed successfully. The first epoch checkpoint is indicated by `'First'`, and intermediate checkpoints by `'Write'`.

```
load('MyCheckpoint.mat')
```

```
checkpoint =

      file: '/WorkdingDir/MyCheckpoint.mat'
      time: [2013 3 22 5 0 9.0712]
    number: 6
     stage: 'Final'
       net: [1x1 network]
        tr: [1x1 struct]
```

You can resume training from the last checkpoint by reloading the dataset (if necessary), then calling train with the recovered network.

```
net = checkpoint.net;
[x,t] = maglev_dataset;
load('MyCheckpoint.mat');
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);
```

Deploy Shallow Neural Network Functions

In this section...

Deployment Functions and Tools for Trained Networks

The function `genFunction` allows stand-alone MATLAB functions for a trained shallow neural network. The generated code contains all the information needed to simulate a neural network, including settings, weight and bias values, module functions, and calculations.

The generated MATLAB function can be used to inspect the exact simulation calculations that a particular shallow neural network performs, and makes it easier to deploy neural networks for many purposes with a wide variety of MATLAB deployment products and tools.

The function `genFunction` is used by the **Neural Net Fitting**, **Neural Net Pattern Recognition**, **Neural Net Clustering** and **Neural Net Time Series** apps. For information on these apps, see “Fit Data with a Shallow Neural Network”, “Classify Patterns with a Shallow Neural Network”, “Cluster Data with a Self-Organizing Map”, and “Shallow Neural Network Time-Series Prediction and Modeling”.

The comprehensive scripts generated by these apps includes an example of deploying networks with `genFunction`.

Generate Neural Network Functions for Application Deployment

The function `genFunction` generates a stand-alone MATLAB function for simulating any trained shallow neural network and preparing it for deployment. This might be useful for several tasks:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Use the MATLAB Function block to create a Simulink block
- Use MATLAB Compiler to:
  - Generate stand-alone executables
  - Generate Excel® add-ins
- Use MATLAB Compiler SDK™ to:
  - Generate C/C++ libraries
  - Generate .COM components
  - Generate Java® components
  - Generate .NET components
- Use MATLAB Coder to:
  - Generate C/C++ code

- Generate efficient MEX-functions

`genFunction(net,'pathname')` takes a neural network and file path, and produces a standalone MATLAB function file `filename.m`.

`genFunction(...,'MatrixOnly','yes')` overrides the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks, the matrix columns are interpreted as independent samples. For dynamic networks, the matrix columns are interpreted as a series of time steps. The default value is `'no'`.

`genFunction(___,'ShowLinks','no')` disables the default behavior of displaying links to generated help and source code. The default is `'yes'`.

Here a static network is trained and its outputs calculated.

```
[x, t] = bodyfat_dataset;
bodyfatNet = feedforwardnet(10);
bodyfatNet = train(bodyfatNet, x, t);
y = bodyfatNet(x);
```

The following code generates, tests, and displays a MATLAB function with the same interface as the neural network object.

```
genFunction(bodyfatNet, 'bodyfatFcn');
y2 = bodyfatFcn(x);
accuracy2 = max(abs(y - y2))
edit bodyfatFcn
```

You can compile the new function with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libBodyfat -T link:lib bodyfatFcn
```

The next code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen` (license required), which is also tested.

```
genFunction(bodyfatNet, 'bodyfatFcn', 'MatrixOnly', 'yes');
y3 = bodyfatFcn(x);
accuracy3 = max(abs(y - y3))

x1Type = coder.typeof(double(0), [13, Inf]); % Coder type of input 1
codegen bodyfatFcn.m -config:mex -o bodyfatCodeGen -args {x1Type}
y4 = bodyfatCodeGen(x);
accuracy4 = max(abs(y - y4))
```

Here a dynamic network is trained and its outputs calculated.

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,{},t);
maglevNet = train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next a MATLAB function is generated and tested. The function is then used to create a shared/dynamically linked library with `mcc`.

```
genFunction(maglevNet,'maglevFcn');
[y2,xf,af] = maglevFcn(X,Xi,Ai);
```

```
accuracy2 = max(abs(cell2mat(y)-cell2mat(y2)))
mcc -W lib:libMaglev -T link:lib maglevFcn
```

The following code generates another version of the MATLAB function that supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool `codegen`, which is also tested.

```
genFunction(maglevNet,'maglevFcn','MatrixOnly','yes');
x1 = cell2mat(X(1,:)); % Convert each input to matrix
x2 = cell2mat(X(2,:));
xi1 = cell2mat(Xi(1,:)); % Convert each input state to matrix
xi2 = cell2mat(Xi(2,:));
[y3,xf1,xf2] = maglevFcn(x1,x2,xi1,xi2);
accuracy3 = max(abs(cell2mat(y)-y3))

x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen ...
                    -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

Generate Simulink Diagrams

For information on simulating shallow neural networks and deploying trained neural networks with Simulink tools, see “Deploy Shallow Neural Network Simulink Diagrams” on page B-5.

See Also

More About

- “Deploy Training of Shallow Neural Networks” on page 28-51

Deploy Training of Shallow Neural Networks

Tip To learn about code generation for deep learning, see “Deep Learning Code Generation”.

Use MATLAB Runtime to deploy functions that can train a model. You can deploy MATLAB code that trains neural networks as described in “Create Standalone Application from MATLAB” (MATLAB Compiler).

The following methods and functions are NOT supported in deployed mode:

- Training progress dialog, `nntraintool`.
- `genFunction` and `gensim` to generate MATLAB code or Simulink blocks
- `view` method
- `nctool`, `nftool`, `nnstart`, `nprtool`, `ntstool`
- Plot functions (such as `plotperform`, `plottrainstate`, `ploterrhist`, `plotregression`, `plotfit`, and so on)
- `perceptron`, `newlind`, and `elmannet` functions.

Here is an example of how you can deploy training of a network. Create a script to train a neural network, for example, `mynntraining.m`:

```
% Create the predictor and response (target)
x = [0.054 0.78 0.13 0.47 0.34 0.79 0.53 0.6 0.65 0.75 0.084 0.91 0.83
     0.53 0.93 0.57 0.012 0.16 0.31 0.17 0.26 0.69 0.45 0.23 0.15 0.54];
t = [0.46 0.079 0.42 0.48 0.95 0.63 0.48 0.51 0.16 0.51 1 0.28 0.3];
% Create and display the network
net = fitnet();
disp('Training fitnet')
% Train the network using the data in x and t
net = train(net,x,t);
% Predict the responses using the trained network
y = net(x);
% Measure the performance
perf = perform(net,y,t)
```

Compile the script `mynntraining.m` by using the command line:

```
mcc -m 'mynntraining.m'
```

`mcc` invokes the MATLAB Compiler to compile code at the prompt. The flag `–m` compiles a MATLAB function and generates a standalone executable. The EXE file is now in your local computer in the working directory.

To run the compiled EXE application on computers that do not have MATLAB installed, you need to download and install MATLAB Runtime. The `readme.txt` created in your working folder has more information about the deployment requirements.

See Also

More About

- “Deploy Shallow Neural Network Functions” on page 28-48

Historical Neural Networks

Historical Neural Networks Overview

This section covers networks that are of historical interest, but that are not as actively used today as networks presented in other sections. Two of the networks are single-layer networks that were the first neural networks for which practical training algorithms were developed: perceptron networks and ADALINE networks.

The perceptron network is a single-layer network whose weights and biases can be trained to produce a correct target vector when presented with the corresponding input vector. This perceptron rule was the first training algorithm developed for neural networks. The original book on the perceptron is Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C., Spartan Press, 1961 [Rose61 on page 32-2].

At about the same time that Rosenblatt developed the perceptron network, Widrow and Hoff developed a single-layer linear network and associated learning rule, which they called the ADALINE (Adaptive Linear Neuron). This network was used to implement adaptive filters, which are still actively used today. The original paper describing this network is Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.

Perceptron Neural Networks

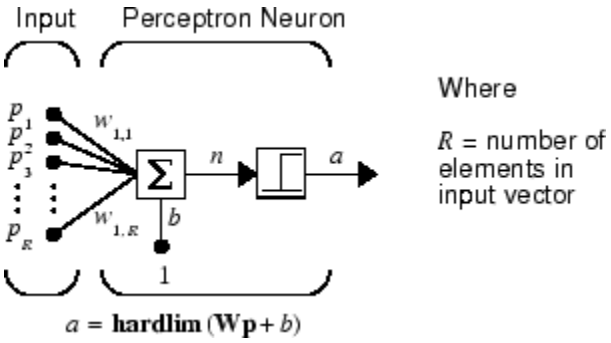
In this section...

Rosenblatt [Rose61 on page 32-2] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

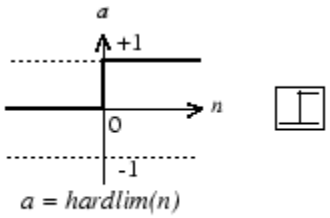
The discussion of perceptrons in this section is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996 on page 32-2], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



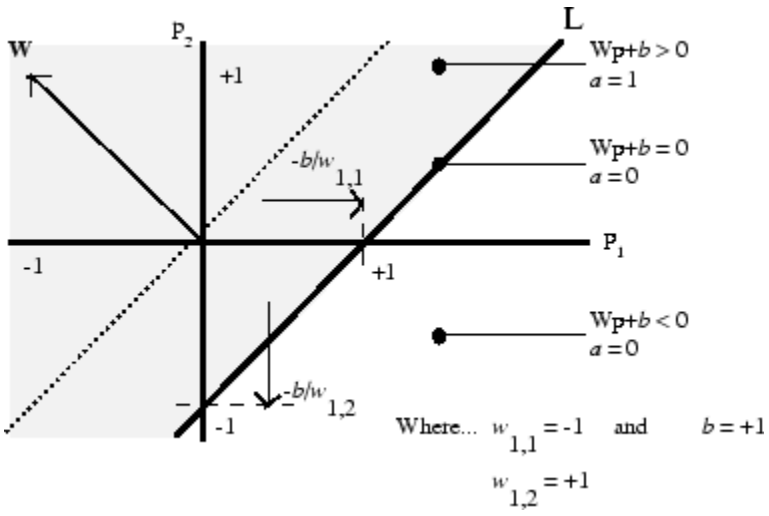
Each external input is weighted with an appropriate weight w_{1j}, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The following figure show the input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$.



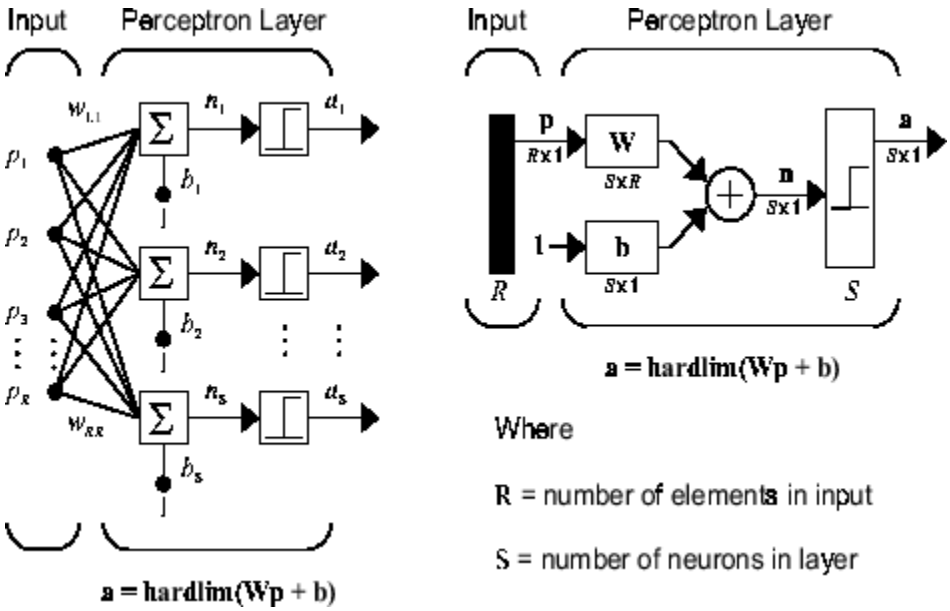
Two classification regions are formed by the *decision boundary* line L at **Wp** + b = 0. This line is perpendicular to the weight matrix **W** and shifted according to the bias b. Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin, as shown in the plot above.

You might want to run the example program nnd4db. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights $w_{i,j}$, as shown below in two forms. As before, the network indices i and j indicate that $w_{i,j}$ is the strength of the connection from the jth input to the ith neuron.



The perceptron learning rule described shortly is capable of training only a single layer. Thus only one-layer networks are considered here. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed in “Limitations and Cautions” on page 29-12.

Create a Perceptron

You can create a perceptron with the following:

```
net = perceptron;
net = configure(net,P,T);
```

where input arguments are as follows:

- `P` is an R-by-Q matrix of Q input vectors of R elements each.
- `T` is an S-by-Q matrix of Q target vectors of S elements each.

Commonly, the `hardlim` function is used in perceptrons, so it is the default.

The following commands create a perceptron network with a single one-element input vector with the values 0 and 2, and one neuron with outputs that can be either 0 or 1:

```
P = [0 2];
T = [0 1];
net = perceptron;
net = configure(net,P,T);
```

You can see what network has been created by executing the following command:

```
inputweights = net.inputweights{1,1}
```

which yields

```
inputweights =
        delays: 0
```

```
       initFcn: 'initzero'
         learn: true
      learnFcn: 'learnp'
    learnParam: (none)
          size: [1 1]
     weightFcn: 'dotprod'
   weightParam: (none)
      userdata: (your custom info)
```

The default learning function is `learnp`, which is discussed in “Perceptron Learning Rule (learnp)” on page 29-6. The net input to the `hardlim` transfer function is `dotprod`, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

The default initialization function `initzero` is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: 1
      userdata: [1x1 struct]
```

You can see that the default initialization for the bias is also 0.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1\mathbf{t}_1, \mathbf{p}_2\mathbf{t}_1, \ldots, \mathbf{p}_Q\mathbf{t}_Q$$

where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response **a** and the target vector **t**. The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector **p** and the associated error **e**. The target vector **t** must contain values of either 0 or 1, because perceptrons (with `hardlim` transfer functions) can only output these values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector **w** to point toward input vectors to be classified as 1 and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to **w** and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct (**a** = **t** and **e** = **t** – **a** = 0), then the weight vector **w** is not altered.

CASE 2. If the neuron output is 0 and should have been 1 (**a** = 0 and **t** = 1, and **e** = **t** – **a** = 1), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 (**a** = 1 and **t** = 0, and **e** = **t** – **a** = –1), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error **e** = **t** – **a** and the change to be made to the weight vector Δ**w**:

CASE 1. If **e** = 0, then make a change Δ**w** equal to 0.

CASE 2. If **e** = 1, then make a change Δ**w** equal to $\mathbf{p}^T$.

CASE 3. If **e** = –1, then make a change Δ**w** equal to $-\mathbf{p}^T$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t - a)\mathbf{p}^T = e\mathbf{p}^T$$

You can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - a)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^T = \mathbf{e}(\mathbf{p})^T$$

and

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = e$$

The perceptron learning rule can be summarized as follows:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

and

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where **e** = **t** – **a**.

Now try a simple example. Start with a single neuron having an input vector with just two elements.

```
net = perceptron;
net = configure(net,[0;0],0);
```

To simplify matters, set the bias equal to 0 and the weights to 1 and -0.8:

```
net.b{1} =  [0];
w = [1 -0.8];
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];
t = [1];
```

You can compute the output and error with

```
a = net(p)
a =
     0
e = t-a
e =
     1
```

and use the function `learnp` to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[],[],[])
dw =
     1     2
```

The new weights, then, are obtained as

```
w = w + dw
w =
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Recall that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are linearly separable. The objects to be classified in such cases can be separated by a single line.

You might want to try the example `nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

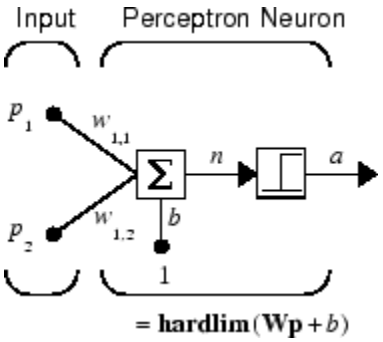
Training (train)

If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traversal through all the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. You must check the new values of **W** and **b** by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully you can train it further by calling `train` again with the new weights and biases for more training passes, or you can analyze the problem to see if it is a suitable problem for the perceptron. Problems that cannot be solved by the perceptron network are discussed in “Limitations and Cautions” on page 29-12.

To illustrate the training procedure, work through a simple problem. Consider a one-neuron perceptron with a single vector input having two elements:



This network, and the problem you are about to consider, are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996 on page 32-2].

Suppose you have the following classification problem and would like to solve it with a single vector input, two-element perceptron network.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias. Denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, the initial values are $\mathbf{W}(0)$ and $b(0)$.

$$\mathbf{W}(0) = [0\ \ 0]\ \ b(0) = 0$$

Start by calculating the perceptron's output a for the first input vector $\mathbf{p}_1$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$
$$= hardlim\left([0\ \ 0]\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output a does not equal the target value t_1, so use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)[2\ \ 2] = [-2\ \ -2]$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the perceptron update rules.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = [0\ \ 0] + [-2\ \ -2] = [-2\ \ -2] = \mathbf{W}(1)$$
$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, $\mathbf{p}_2$. The output is calculated below.

$$\alpha = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$
$$= hardlim\left([-2\ -2]\begin{bmatrix}1\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = [-2\ -2]$ and $b(2) = b(1) = -1$.

You can continue in this fashion, presenting $\mathbf{p}_3$ next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values $\mathbf{W}(4) = [-3\ -1]$ and $b(4) = 0$. To determine whether a satisfactory solution is obtained, make one pass through all input vectors to see if they all produce the desired target values. This is not true for the fourth input, but the algorithm does converge on the sixth presentation of an input. The final values are

$\mathbf{W}(6) = [-2\ -3]$ and $b(6) = 1$.

This concludes the hand calculation. Now, how can you do this using the `train` function?

The following code defines a perceptron.

```
net = perceptron;
```

Consider the application of a single input

```
p = [2; 2];
```

having the target

```
t = [0];
```

Set `epochs` to 1, so that `train` goes through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2    -2
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values $[-2\ -2]$ and -1, just as you hand calculated.

Now apply the second input vector $\mathbf{p}_2$. The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0, and the change will be zero. You could proceed in this way, starting from the previous result and applying a new input vector time after time. But you can do this job automatically with `train`.

Apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = perceptron;
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]
t = [0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -3    -1
b =
     0
```

This is the same result as you got previously by hand.

Finally, simulate the trained network for each of the inputs.

```
a = net(p)
a =
      0     0     1     1
```

The outputs do not yet equal the targets, so you need to train the network for more than one pass. Try more epochs. This run gives a mean absolute error performance of 0 after two epochs:

```
net.trainParam.epochs = 1000;
net = train(net,p,t);
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As you know from hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w = net.iw{1,1}, b = net.b{1}
w =
    -2    -3
b =
     1
```

The simulated output and errors for the various inputs are

```
a = net(p)
a =
             0             1             0             1
error = a-t
error =
             0             0             0             0
```

You confirm that the training procedure is successful. The network converges and produces the correct target outputs for the four input vectors.

The default training function for networks created with `perceptron` is `trainc`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with

`train` will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You might want to try various example programs. For instance, “Classification with a Two-Input Perceptron” on page 31-141 illustrates classification and training of a simple perceptron.

Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of `adapt` in this way guarantees that any linearly separable problem is solved in a finite number of training presentations.

As noted in the previous pages, perceptrons can also be trained with the function `train`. Commonly when `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) because of the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. However, it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try “Linearly Non-separable Vectors” on page 31-158. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two-neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996 on page 32-2].

Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try “Outlier Input Vectors” on page 31-146 to see how an outlier affects the training.

By changing the perceptron learning rule slightly, you can make training times insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t - a)\mathbf{p}^T = e\mathbf{p}^T$$

As shown above, the larger an input vector **p**, the larger its effect on the weight vector **w**. Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that the effect of each input vector on the weights is of the same magnitude:

$$\Delta\mathbf{w} = (t - \alpha)\frac{\mathbf{p}^T}{\|\mathbf{p}\|} = e\frac{\mathbf{p}^T}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnp`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces the number of epochs considerably if there are outlier input vectors. You might try “Normalized Perceptron Rule” on page 31-152 to see how this normalized training rule works.

Linear Neural Networks

In this section...

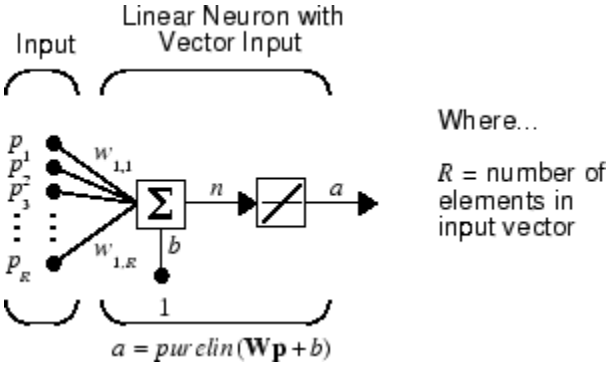
The linear networks discussed in this section are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve linearly separable problems.

Here you design a linear network that, when presented with a set of given input vectors, produces outputs of corresponding target vectors. For each input vector, you can calculate the network's output vector. The difference between an output vector and its target vector is the error. You would like to find values for the network weights and biases such that the sum of the squares of the errors is minimized or below a specific value. This problem is manageable because linear systems have a single error minimum. In most cases, you can calculate a linear network directly, such that its error is a minimum for the given input vectors and target vectors. In other cases, numerical problems prohibit direct calculation. Fortunately, you can always train the network to have a minimum error by using the least mean squares (Widrow-Hoff) algorithm.

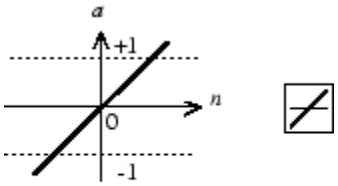
This section introduces `linearlayer`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function `purelin`.



Linear Transfer Function

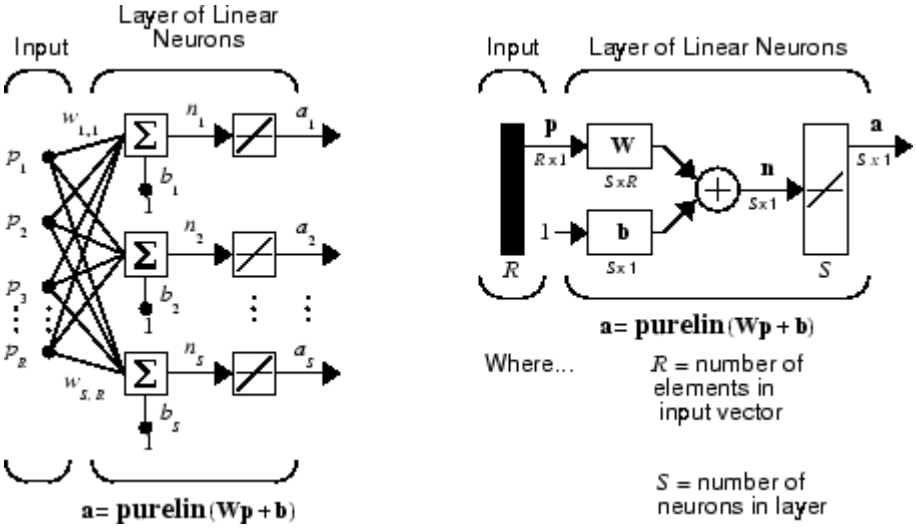
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Network Architecture

The linear network shown below has one layer of S neurons connected to R inputs through a matrix of weights **W**.

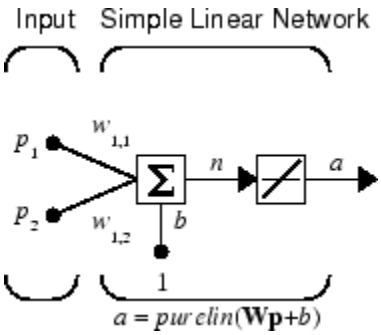


Note that the figure on the right defines an S-length output vector **a**.

A single-layer linear network is shown. However, this network is just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Create a Linear Neuron (linearlayer)

Consider a single linear neuron with two inputs. The following figure shows the diagram for this network.



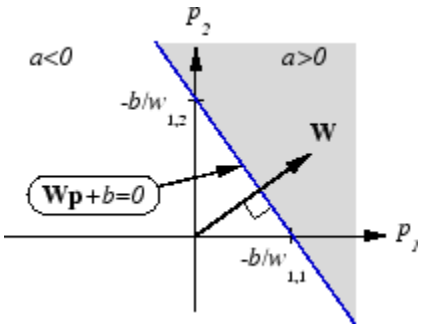
The weight matrix **W** in this case has only one row. The network output is

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

or

$$\alpha = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the linear network has a *decision boundary* that is determined by the input vectors for which the net input n is zero. For $n = 0$ the equation $\mathbf{Wp} + b = 0$ specifies such a decision boundary, as shown below (adapted with thanks from [HDB96 on page 32-2]).



Input vectors in the upper right gray area lead to an output greater than 0. Input vectors in the lower left white area lead to an output less than 0. Thus, the linear network can be used to classify objects into two categories. However, it can classify in this way only if the objects are linearly separable. Thus, the linear network has the same limitation as the perceptron.

You can create this network using `linearlayer`, and configure its dimensions with two values so the input has two elements and the output has one.

```
net = linearlayer;
net = configure(net,[0;0],0);
```

The network weights and biases are set to zero by default. You can see the current values with the commands

```
W = net.IW{1,1}
W =
     0     0
```

and

```
b= net.b{1}
b =
     0
```

However, you can give the weights any values that you want, such as 2 and 3, respectively, with

```
net.IW{1,1} = [2 3];
W = net.IW{1,1}
W =
     2     3
```

You can set and check the bias in the same way.

```
net.b{1} = [-4];
b = net.b{1}
b =
     -4
```

You can simulate the linear network for a particular input vector. Try

```
p = [5;6];
```

You can find the network output with the function `sim`.

```
a = net(p)
a =
    24
```

To summarize, you can create a linear network with `linearlayer`, adjust its elements as you want, and simulate it with `sim`.

Least Mean Square Error

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The error is calculated as the difference between the target output and the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the linear network so as to minimize this mean square error.

Fortunately, the mean square error performance index for the linear network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

You can find more about this topic in Chapter 10 of [HDB96 on page 32-2].

Linear System Design (newlind)

Unlike most other network architectures, linear networks can be designed directly if input/target vector pairs are known. You can obtain specific network values for weights and biases to minimize the mean square error by using the function `newlind`.

Suppose that the inputs and targets are

```
P = [1 2 3];
T= [2.0 4.1 5.9];
```

Now you can design a network.

```
net = newlind(P,T);
```

You can simulate the network behavior to check that the design was done properly.

```
Y = net(P)
Y =
    2.0500    4.0000    5.9500
```

Note that the network outputs are quite close to the desired targets.

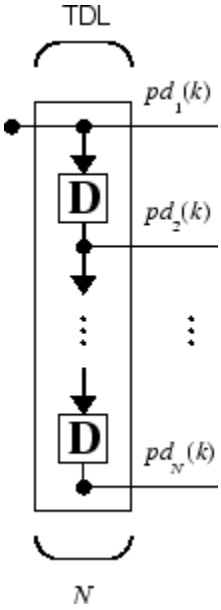
You might try “Pattern Association Showing Error Surface” on page 31-161. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

You can also use the function `newlind` to design linear networks having delays in the input. Such networks are discussed in “Linear Networks with Delays” on page 29-18. First, however, delays must be discussed.

Linear Networks with Delays

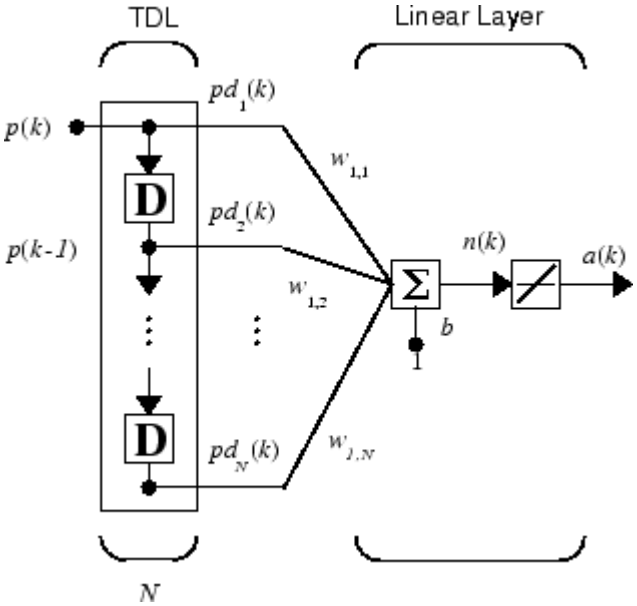
Tapped Delay Line

You need a new component, the tapped delay line, to make full use of the linear network. Such a delay line is shown below. There the input signal enters from the left and passes through *N*-1 delays. The output of the tapped delay line (TDL) is an *N*-dimensional vector, made up of the input signal at the current time, the previous input signal, etc.



Linear Filter

You can combine a tapped delay line with a linear network to create the linear *filter shown*.



The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i} p(k - i + 1) + b$$

The network shown is referred to in the digital signal processing field as a finite impulse response (FIR) filter [WiSt85 on page 32-2]. Look at the code used to generate and simulate such a network.

Suppose that you want a linear layer that outputs the sequence T, given the sequence P and two initial input delay states Pi.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5 6 4 20 7 8};
```

You can use newlind to design a network with delays to give the appropriate outputs for the inputs. The delay initial outputs are supplied as a third argument, as shown below.

```
net = newlind(P,T,Pi);
```

You can obtain the output of the designed network with

```
Y = net(P,Pi)
```

to give

```
Y = [2.7297] [10.5405] [5.0090] [14.9550] [10.7838] [5.9820]
```

As you can see, the network outputs are not exactly equal to the targets, but they are close and the mean square error is minimized.

LMS Algorithm (learnwh)

The LMS algorithm, or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If you take the partial derivative of the squared error with respect to the weights and biases at the kth iteration, you have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1,2,\ldots,R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - \alpha(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - (\mathbf{W}\mathbf{p}(k) + b)]$$

or

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[t(k) - \left(\sum_{i=1}^{R} w_{1,i}p_i(k) + b\right)\right]$$

Here $p_i(k)$ is the ith element of the input vector at the kth iteration.

This can be simplified to

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

and

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, change the weight matrix, and the bias will be

$2\alpha e(k)\mathbf{p}(k)$

and

$2\alpha e(k)$

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors, and α is a *learning rate*. If α is large, learning occurs quickly, but if it is too large it can lead to instability and errors might even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix $\mathbf{p}^T\mathbf{p}$ of the input vectors.

You might want to read some of Chapter 10 of [HDB96 on page 32-2] for more information about the LMS algorithm and its convergence.

Fortunately, there is a toolbox function, `learnwh`, that does all the calculation for you. It calculates the change in weights as

```
dw = lr*e*p'
```

and the bias change as

```
db = lr*e
```

The constant 2, shown a few lines above, has been absorbed into the code learning rate `lr`. The function `maxlinlr` calculates this maximum stable learning rate `lr` as 0.999 * `P'*P`.

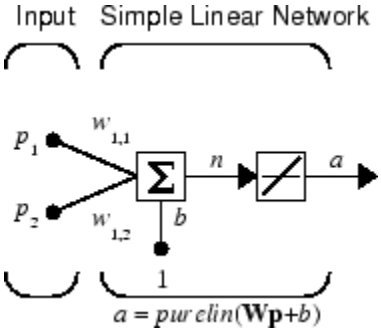
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

Linear Classification (train)

Linear networks can be trained to perform linear classification with the function `train`. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to `learnp`. Then the network is adjusted with the sum of all these corrections. Each pass through the input vectors is called an *epoch*. This contrasts with `adapt` which adjusts weights for each input vector as it is presented.

Finally, `train` applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped, and `train` returns the new network and a training record. Otherwise `train` goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.

A simple problem illustrates this procedure. Consider the linear network introduced earlier.



Suppose you have the following classification problem.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Here there are four input vectors, and you want a network that produces the output corresponding to each input vector when that vector is presented.

Use `train` to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network are 0 by default. Set the error goal to 0.1 rather than accept its default of 0.

```
P = [2 1 -2 -1;2 -2 2 1];
T = [0 1 0 1];
net = linearlayer;
net.trainParam.goal= 0.1;
net = train(net,P,T);
```

The problem runs for 64 epochs, achieving a mean square error of 0.0999. The new weights and bias are

```
weights = net.iw{1,1}
weights =
   -0.0615   -0.2194
bias = net.b(1)
bias =
    [0.5899]
```

You can simulate the new network as shown below.

```
A = net(P)
A =
    0.0282    0.9672    0.2741    0.4320
```

You can also calculate the error.

```
err = T - sim(net,P)
err =
   -0.0282    0.0328   -0.2741    0.5680
```

Note that the targets are not realized exactly. The problem would have run longer in an attempt to get perfect results had a smaller error goal been chosen, but in this problem it is not possible to

obtain a goal of 0. The network is limited in its capability. See “Limitations and Cautions” on page 29-23 for examples of various limitations.

This example program, “Training a Linear Neuron” on page 31-164, shows the training of a linear neuron and plots the weight trajectory and error during training.

You might also try running the example program `nnd10lc`. It addresses a classic and historically interesting problem, shows how a network can be trained to classify various patterns, and shows how the trained network responds when noisy patterns are presented.

Limitations and Cautions

Linear networks can only learn linear relationships between input and output vectors. Thus, they cannot find solutions to some problems. However, even if a perfect solution does not exist, the linear network will minimize the sum of squared errors if the learning rate `lr` is sufficiently small. The network will find as close a solution as is possible given the linear nature of the network's architecture. This property holds because the error surface of a linear network is a multidimensional parabola. Because parabolas have only one minimum, a gradient descent algorithm (such as the LMS rule) must produce a solution at that minimum.

Linear networks have various other limitations. Some of them are discussed below.

Overdetermined Systems

Consider an overdetermined system. Suppose that you have a network to be trained with four one-element input vectors and four targets. A perfect solution to $wp + b = t$ for each of the inputs might not exist, for there are four constraining equations, and only one weight and one bias to adjust. However, the LMS rule still minimizes the error. You might try “Linear Fit of Nonlinear Problem” on page 31-167 to see how this is done.

Underdetermined Systems

Consider a single linear neuron with one input. This time, in “Underdetermined Problem” on page 31-171, train it on only one one-element input vector and its one-element target vector:

```
P = [1.0];
T = [0.5];
```

Note that while there is only one constraint arising from the single input/target pair, there are two variables, the weight and the bias. Having more variables than constraints results in an underdetermined problem with an infinite number of solutions. You can try “Underdetermined Problem” on page 31-171 to explore this topic.

Linearly Dependent Vectors

Normally it is a straightforward job to determine whether or not a linear network can solve a problem. Commonly, if a linear network has at least as many degrees of freedom ($S*R + S$ = number of weights and biases) as constraints (Q = pairs of input/target vectors), then the network can solve the problem. This is true except when the input vectors are linearly dependent and they are applied to a network without biases. In this case, as shown with the example “Linearly Dependent Problem” on page 31-175, the network cannot solve the problem with zero error. You might want to try “Linearly Dependent Problem” on page 31-175.

Too Large a Learning Rate

You can always train a linear network with the Widrow-Hoff rule to find the minimum error solution for its weights and biases, as long as the learning rate is small enough. Example “Too Large a Learning Rate” on page 31-176 shows what happens when a neuron with one input and a bias is trained with a learning rate larger than that recommended by `maxlinlr`. The network is trained with two different learning rates to show the results of using too large a learning rate.

Neural Network Object Reference

Neural Network Object Properties

In this section...

These properties define the basic features of a network. “Neural Network Subobject Properties” on page 30-11 describes properties that define network details.

General

Here are the general properties of neural networks.

net.name

This property consists of a string defining the network name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.userdata

This property provides a place for users to add custom information to a network object. Only one field is predefined. It contains a *secret* message to all Deep Learning Toolbox users:

```
net.userdata.note
```

Architecture

These properties determine the number of network subobjects (which include inputs, layers, outputs, targets, biases, and weights), and how they are connected.

net.numInputs

This property defines the number of inputs a network receives. It can be set to 0 or a positive integer.

Clarification

The number of network inputs and the size of a network input are *not* the same thing. The number of inputs defines how many sets of vectors the network receives as input. The size of each input (i.e., the number of elements in each input vector) is determined by the input size (`net.inputs{i}.size`).

Most networks have only one input, whose size is determined by the problem.

Side Effects

Any change to this property results in a change in the size of the matrix defining connections to layers from inputs, (`net.inputConnect`) and the size of the cell array of input subobjects (`net.inputs`).

net.numLayers

This property defines the number of layers a network has. It can be set to 0 or a positive integer.

Side Effects

Any change to this property changes the size of each of these Boolean matrices that define connections to and from layers:

```
net.biasConnect
net.inputConnect
net.layerConnect
net.outputConnect
```

and changes the size of each cell array of subobject structures whose size depends on the number of layers:

```
net.biases
net.inputWeights
net.layerWeights
net.outputs
```

and also changes the size of each of the network's adjustable parameter's properties:

```
net.IW
net.LW
net.b
```

net.biasConnect

This property defines which layers have biases. It can be set to any N-by-1 matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a bias to the ith layer is indicated by a 1 (or 0) at

```
net.biasConnect(i)
```

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of biases (`net.biases`) and, in the presence or absence of vectors in the cell array, of bias vectors (`net.b`).

net.inputConnect

This property defines which layers have weights coming from inputs.

It can be set to any $N_l \times N_i$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`). The presence (or absence) of a weight going to the ith layer from the jth input is indicated by a 1 (or 0) at `net.inputConnect(i,j)`.

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of input weight subobjects (`net.inputWeights`) and the presence or absence of matrices in the cell array of input weight matrices (`net.IW`).

net.layerConnect

This property defines which layers have weights coming from other layers. It can be set to any $N_l \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a weight going to the *i*th layer from the *j*th layer is indicated by a 1 (or 0) at

```
net.layerConnect(i,j)
```

Side Effects

Any change to this property alters the presence or absence of structures in the cell array of layer weight subobjects (`net.layerWeights`) and the presence or absence of matrices in the cell array of layer weight matrices (`net.LW`).

net.outputConnect

This property defines which layers generate network outputs. It can be set to any $1 \times N_l$ matrix of Boolean values, where N_l is the number of network layers (`net.numLayers`). The presence (or absence) of a network output from the *i*th layer is indicated by a 1 (or 0) at `net.outputConnect(i)`.

Side Effects

Any change to this property alters the number of network outputs (`net.numOutputs`) and the presence or absence of structures in the cell array of output subobjects (`net.outputs`).

net.numOutputs (read only)

This property indicates how many outputs the network has. It is always equal to the number of 1s in `net.outputConnect`.

net.numInputDelays (read only)

This property indicates the number of time steps of past inputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's input weights:

```
numInputDelays = 0;
for i=1:net.numLayers
  for j=1:net.numInputs
    if net.inputConnect(i,j)
      numInputDelays = max( ...
        [numInputDelays net.inputWeights{i,j}.delays]);
    end
  end
end
```

net.numLayerDelays (read only)

This property indicates the number of time steps of past layer outputs that must be supplied to simulate the network. It is always set to the maximum delay value associated with any of the network's layer weights:

```
numLayerDelays = 0;
for i=1:net.numLayers
  for j=1:net.numLayers
    if net.layerConnect(i,j)
```

```
      numLayerDelays = max( ...
       [numLayerDelays net.layerWeights{i,j}.delays]);
    end
  end
end
```

net.numWeightElements (read only)

This property indicates the number of weight and bias values in the network. It is the sum of the number of elements in the matrices stored in the two cell arrays:

```
net.IW
new.b
```

Subobject Structures

These properties consist of cell arrays of structures that define each of the network's inputs, layers, outputs, targets, biases, and weights.

The properties for each kind of subobject are described in “Neural Network Subobject Properties” on page 30-11.

net.inputs

This property holds structures of properties for each of the network's inputs. It is always an $N_i \times 1$ cell array of input structures, where N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the ith network input is located at

```
net.inputs{i}
```

If a neural network has only one input, then you can access `net.inputs{1}` without the cell array notation as follows:

```
net.input
```

Input Properties

See “Inputs” on page 30-11 for descriptions of input properties.

net.layers

This property holds structures of properties for each of the network's layers. It is always an $N_l \times 1$ cell array of layer structures, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the ith layer is located at `net.layers{i}`.

Layer Properties

See “Layers” on page 30-12 for descriptions of layer properties.

net.outputs

This property holds structures of properties for each of the network's outputs. It is always a $1 \times N_l$ cell array, where N_l is the number of network outputs (`net.numOutputs`).

The structure defining the properties of the output from the *i*th layer (or a null matrix `[]`) is located at `net.outputs{i}` if `net.outputConnect(i)` is 1 (or 0).

If a neural network has only one output at layer `i`, then you can access `net.outputs{i}` without the cell array notation as follows:

```
net.output
```

Output Properties

See “Outputs” on page 30-16 for descriptions of output properties.

net.biases

This property holds structures of properties for each of the network's biases. It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the bias associated with the *i*th layer (or a null matrix `[]`) is located at `net.biases{i}` if `net.biasConnect(i)` is 1 (or 0).

Bias Properties

See “Biases” on page 30-18 for descriptions of bias properties.

net.inputWeights

This property holds structures of properties for each of the network's input weights. It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The structure defining the properties of the weight going to the *i*th layer from the *j*th input (or a null matrix `[]`) is located at `net.inputWeights{i,j}` if `net.inputConnect(i,j)` is 1 (or 0).

Input Weight Properties

See “Input Weights” on page 30-19 for descriptions of input weight properties.

net.layerWeights

This property holds structures of properties for each of the network's layer weights. It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The structure defining the properties of the weight going to the *i*th layer from the *j*th layer (or a null matrix `[]`) is located at `net.layerWeights{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

Layer Weight Properties

See “Layer Weights” on page 30-20 for descriptions of layer weight properties.

Functions

These properties define the algorithms to use when a network is to adapt, is to be initialized, is to have its performance measured, or is to be trained.

net.adaptFcn

This property defines the function to be used when the network adapts. It can be set to the name of any network adapt function. The network adapt function is used to perform adaption whenever `adapt` is called.

```
[net,Y,E,Pf,Af] = adapt(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

Side Effects

Whenever this property is altered, the network's adaption parameters (`net.adaptParam`) are set to contain the parameters and default values of the new function.

net.adaptParam

This property defines the parameters and values of the current adapt function. Call `help` on the current adapt function to get a description of what each field means:

```
help(net.adaptFcn)
```

net.derivFcn

This property defines the derivative function to be used to calculate error gradients and Jacobians when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of any derivative function.

For a list of functions, type `help nnderivative`.

net.divideFcn

This property defines the data division function to be used when the network is trained using a supervised algorithm, such as backpropagation. You can set this property to the name of a division function.

For a list of functions, type `help nndivision`.

Side Effects

Whenever this property is altered, the network's adaption parameters (`net.divideParam`) are set to contain the parameters and default values of the new function.

net.divideParam

This property defines the parameters and values of the current data-division function. To get a description of what each field means, type the following command:

```
help(net.divideFcn)
```

net.divideMode

This property defines the target data dimensions which to divide up when the data division function is called. Its default value is `'sample'` for static networks and `'time'` for dynamic networks. It may also be set to `'sampletime'` to divide targets by both sample and timestep, `'all'` to divide up targets by every scalar value, or `'none'` to not divide up data at all (in which case all data is used for training, none for validation or testing).

net.initFcn

This property defines the function used to initialize the network's weight matrices and bias vectors. . The initialization function is used to initialize the network whenever `init` is called:

```
net = init(net)
```

Side Effects

Whenever this property is altered, the network's initialization parameters (`net.initParam`) are set to contain the parameters and default values of the new function.

net.initParam

This property defines the parameters and values of the current initialization function. Call `help` on the current initialization function to get a description of what each field means:

```
help(net.initFcn)
```

net.performFcn

This property defines the function used to measure the network’s performance. The performance function is used to calculate network performance during training whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nnperformance`.

Side Effects

Whenever this property is altered, the network's performance parameters (`net.performParam`) are set to contain the parameters and default values of the new function.

net.performParam

This property defines the parameters and values of the current performance function. Call `help` on the current performance function to get a description of what each field means:

```
help(net.performFcn)
```

net.plotFcns

This property consists of a row cell array of strings, defining the plot functions associated with a network. The neural network training window, which is opened by the `train` function, shows a button for each plotting function. Click the button during or after training to open the desired plot.

net.plotParams

This property consists of a row cell array of structures, defining the parameters and values of each plot function in `net.plotFcns`. Call `help` on the each plot function to get a description of what each field means:

```
help(net.plotFcns{i})
```

net.trainFcn

This property defines the function used to train the network. It can be set to the name of any of the training functions, which is used to train the network whenever `train` is called.

```
[net,tr] = train(NET,P,T,Pi,Ai)
```

For a list of functions, type `help nntrain`.

Side Effects

Whenever this property is altered, the network's training parameters (`net.trainParam`) are set to contain the parameters and default values of the new function.

net.trainParam

This property defines the parameters and values of the current training function. Call `help` on the current training function to get a description of what each field means:

```
help(net.trainFcn)
```

Weight and Bias Values

These properties define the network's adjustable parameters: its weight matrices and bias vectors.

net.IW

This property defines the weight matrices of weights going to layers from network inputs. It is always an $N_l \times N_i$ cell array, where N_l is the number of network layers (`net.numLayers`), and N_i is the number of network inputs (`net.numInputs`).

The weight matrix for the weight going to the *i*th layer from the *j*th input (or a null matrix `[]`) is located at `net.IW{i,j}` if `net.inputConnect(i,j)` is `1` (or `0`).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the input size with the number of delays associated with the weight:

```
net.inputs{j}.size * length(net.inputWeights{i,j}.delays)
```

The preprocessing function `net.inputs{i}.processFcns` is specified as `'removeconstantrows'` by default in some networks. In this case, if the network input `X` contains `m` rows where all row elements have the same value, the weight matrix has `m` less columns than the above product. For more details about the network input `X`, see `train`.

These dimensions can also be obtained from the input weight properties:

```
net.inputWeights{i,j}.size
```

net.LW

This property defines the weight matrices of weights going to layers from other layers. It is always an $N_l \times N_l$ cell array, where N_l is the number of network layers (`net.numLayers`).

The weight matrix for the weight going to the *i*th layer from the *j*th layer (or a null matrix `[]`) is located at `net.LW{i,j}` if `net.layerConnect(i,j)` is 1 (or 0).

The weight matrix has as many rows as the size of the layer it goes to (`net.layers{i}.size`). It has as many columns as the product of the size of the layer it comes from with the number of delays associated with the weight:

```
net.layers{j}.size * length(net.layerWeights{i,j}.delays)
```

These dimensions can also be obtained from the layer weight properties:

```
net.layerWeights{i,j}.size
```

net.b

This property defines the bias vectors for each layer with a bias. It is always an $N_l \times 1$ cell array, where N_l is the number of network layers (`net.numLayers`).

The bias vector for the *i*th layer (or a null matrix `[]`) is located at `net.b{i}` if `net.biasConnect(i)` is 1 (or 0).

The number of elements in the bias vector is always equal to the size of the layer it is associated with (`net.layers{i}.size`).

This dimension can also be obtained from the bias properties:

```
net.biases{i}.size
```

Neural Network Subobject Properties

These properties define the details of a network's inputs, layers, outputs, targets, biases, and weights.

In this section...

Inputs

These properties define the details of each *i*th network input.

net.inputs{1}.name

This property consists of a string defining the input name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.inputs{i}.feedbackInput (read only)

If this network is associated with an open-loop feedback output, then this property will indicate the index of that output. Otherwise it will be an empty matrix.

net.inputs{i}.processFcns

This property defines a row cell array of processing function names to be used by *i*th network input. The processing functions are applied to input values before the network uses them.

Side Effects

Whenever this property is altered, the input `processParams` are set to default values for the given processing functions, `processSettings`, `processedSize`, and `processedRange` are defined by applying the process functions and parameters to `exampleInput`.

For a list of processing functions, type `help nnprocess`.

net.inputs{i}.processParams

This property holds a row cell array of processing function parameters to be used by *i*th network input. The processing parameters are applied by the processing functions to input values before the network uses them.

Side Effects

Whenever this property is altered, the input `processSettings`, `processedSize`, and `processedRange` are defined by applying the process functions and parameters to `exampleInput`.

net.inputs{i}.processSettings (read only)

This property holds a row cell array of processing function settings to be used by *i*th network input. The processing settings are found by applying the processing functions and parameters to

exampleInput and then used to provide consistent results to new input values before the network uses them.

net.inputs{i}.processedRange (read only)

This property defines the range of exampleInput values after they have been processed with processingFcns and processingParams.

net.inputs{i}.processedSize (read only)

This property defines the number of rows in the exampleInput values after they have been processed with processingFcns and processingParams.

net.inputs{i}.range

This property defines the range of each element of the *i*th network input.

It can be set to any $R_i \times 2$ matrix, where R_i is the number of elements in the input (net.inputs{i}.size), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum values of the *j*th input element, in that order:

```
net.inputs{i}(j,:)
```

Uses

Some initialization functions use input ranges to find appropriate initial values for input weight matrices.

Side Effects

Whenever the number of rows in this property is altered, the input size, processedSize, and processedRange change to remain consistent. The sizes of any weights coming from this input and the dimensions of the weight matrices also change.

net.inputs{i}.size

This property defines the number of elements in the *i*th network input. It can be set to 0 or a positive integer.

Side Effects

Whenever this property is altered, the input range, processedRange, and processedSize are updated. Any associated input weights change size accordingly.

net.inputs{i}.userdata

This property provides a place for users to add custom information to the *i*th network input.

Layers

These properties define the details of each *i*th network layer.

net.layers{i}.name

This property consists of a string defining the layer name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.layers{i}.dimensions

This property defines the *physical* dimensions of the *i*th layer's neurons. Being able to arrange a layer's neurons in a multidimensional manner is important for self-organizing maps.

It can be set to any row vector of 0 or positive integer elements, where the product of all the elements becomes the number of neurons in the layer (`net.layers{i}.size`).

Uses

Layer dimensions are used to calculate the neuron positions within the layer (`net.layers{i}.positions`) using the layer's topology function (`net.layers{i}.topologyFcn`).

Side Effects

Whenever this property is altered, the layer's size (`net.layers{i}.size`) changes to remain consistent. The layer's neuron positions (`net.layers{i}.positions`) and the distances between the neurons (`net.layers{i}.distances`) are also updated.

net.layers{i}.distanceFcn

This property defines which of the distance functions is used to calculate `distances` between neurons in the *i*th layer from the neuron `positions`. Neuron distances are used by self-organizing maps. It can be set to the name of any distance function.

For a list of functions, type `help nndistance`.

Side Effects

Whenever this property is altered, the distances between the layer's neurons (`net.layers{i}.distances`) are updated.

net.layers{i}.distances (read only)

This property defines the distances between neurons in the *i*th layer. These distances are used by self-organizing maps:

```
net.layers{i}.distances
```

It is always set to the result of applying the layer's distance function (`net.layers{i}.distanceFcn`) to the positions of the layer's neurons (`net.layers{i}.positions`).

net.layers{i}.initFcn

This property defines which of the layer initialization functions are used to initialize the *i*th layer, if the network initialization function (`net.initFcn`) is `initlay`. If the network initialization is set to `initlay`, then the function indicated by this property is used to initialize the layer's weights and biases.

net.layers{i}.netInputFcn

This property defines which of the net input functions is used to calculate the *i*th layer's net input, given the layer's weighted inputs and bias during simulating and training.

For a list of functions, type `help nnnetinput`.

net.layers{i}.netInputParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field:

```
help(net.layers{i}.netInputFcn)
```

net.layers{i}.positions (read only)

This property defines the positions of neurons in the *i*th layer. These positions are used by self-organizing maps.

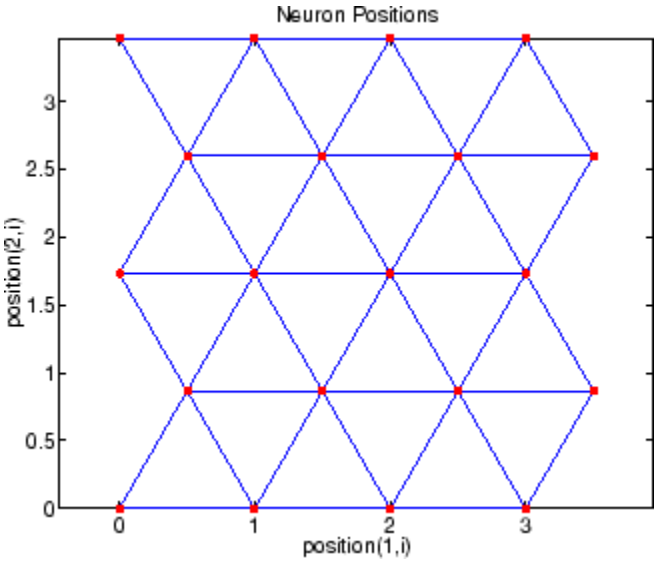
It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

Plotting

Use `plotsom` to plot the positions of a layer's neurons.

For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [4 5], and the topology function (`net.layers{1}.topologyFcn`) is `hextop`, the neurons' positions can be plotted as follows:

```
plotsom(net.layers{1}.positions)
```



net.layers{i}.range (read only)

This property defines the output range of each neuron of the *i*th layer.

It is set to an $S_i \times 2$ matrix, where S_i is the number of neurons in the layer (`net.layers{i}.size`), and each element in column 1 is less than the element next to it in column 2.

Each *j*th row defines the minimum and maximum output values of the layer's transfer function `net.layers{i}.transferFcn`.

net.layers{i}.size

This property defines the number of neurons in the *i*th layer. It can be set to 0 or a positive integer.

Side Effects

Whenever this property is altered, the sizes of any input weights going to the layer (`net.inputWeights{i,:}.size`), any layer weights going to the layer (`net.layerWeights{i,:}.size`) or coming from the layer (`net.layerWeights{i,:}.size`), and the layer's bias (`net.biases{i}.size`), change.

The dimensions of the corresponding weight matrices (`net.IW{i,:}`, `net.LW{i,:}`, `net.LW{:,i}`), and biases (`net.b{i}`) also change.

Changing this property also changes the size of the layer's output (`net.outputs{i}.size`) and target (`net.targets{i}.size`) if they exist.

Finally, when this property is altered, the dimensions of the layer's neurons (`net.layers{i}.dimension`) are set to the same value. (This results in a one-dimensional arrangement of neurons. If another arrangement is required, set the `dimensions` property directly instead of using `size`.)

net.layers{i}.topologyFcn

This property defines which of the topology functions are used to calculate the *i*th layer's neuron positions (`net.layers{i}.positions`) from the layer's dimensions (`net.layers{i}.dimensions`).

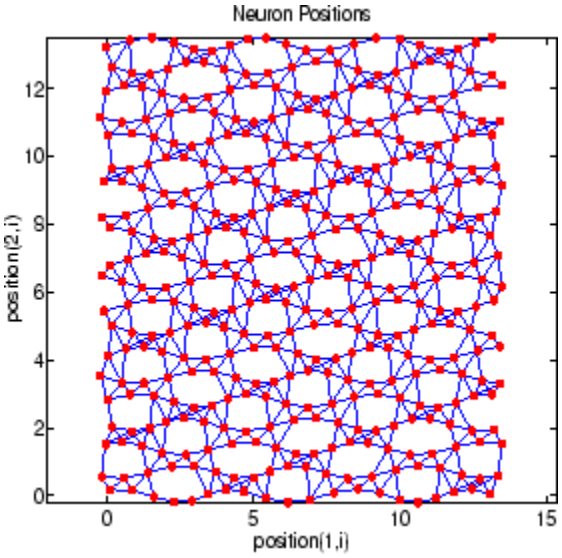
For a list of functions, type `help nntopology`.

Side Effects

Whenever this property is altered, the positions of the layer's neurons (`net.layers{i}.positions`) are updated.

Use `plotsom` to plot the positions of the layer neurons. For instance, if the first-layer neurons of a network are arranged with dimensions (`net.layers{1}.dimensions`) of [8 10] and the topology function (`net.layers{1}.topologyFcn`) is `randtop`, the neuron positions are arranged to resemble the following plot:

```
plotsom(net.layers{1}.positions)
```



net.layers{i}.transferFcn

This function defines which of the transfer functions is used to calculate the *i*th layer's output, given the layer's net input, during simulation and training.

For a list of functions, type `help nntransfer`.

net.layers{i}.transferParam

This property defines the parameters of the layer's transfer function. Call `help` on the current transfer function to get a description of what each field means:

```
help(net.layers{i}.transferFcn)
```

net.layers{i}.userdata

This property provides a place for users to add custom information to the *i*th network layer.

Outputs

net.outputs{i}.name

This property consists of a string defining the output name. Network creation functions, such as `feedforwardnet`, define this appropriately. But it can be set to any string as desired.

net.outputs{i}.feedbackInput

If the output implements open-loop feedback (`net.outputs{i}.feedbackMode = 'open'`), then this property indicates the index of the associated feedback input, otherwise it will be an empty matrix.

net.outputs{i}.feedbackDelay

This property defines the timestep difference between this output and network inputs. Input-to-output network delays can be removed and added with `removedelay` and `adddelay` functions resulting in this property being incremented or decremented respectively. The difference in timing between

inputs and outputs is used by `preparets` to properly format simulation and training data, and used by `closeloop` to add the correct number of delays when closing an open-loop output, and `openloop` to remove delays when opening a closed loop.

net.outputs{i}.feedbackMode

This property is set to the string `'none'` for non-feedback outputs. For feedback outputs it can either be set to `'open'` or `'closed'`. If it is set to `'open'`, then the output will be associated with a feedback input, with the property `feedbackInput` indicating the input's index.

net.outputs{i}.processFcns

This property defines a row cell array of processing function names to be used by the *i*th network output. The processing functions are applied to target values before the network uses them, and applied in reverse to layer output values before being returned as network output values.

Side Effects

When you change this property, you also affect the following settings: the output parameters `processParams` are modified to the default values of the specified processing functions; `processSettings`, `processedSize`, and `processedRange` are defined using the results of applying the process functions and parameters to `exampleOutput`; the *i*th layer size is updated to match the `processedSize`.

For a list of functions, type `help nnprocess`.

net.outputs{i}.processParams

This property holds a row cell array of processing function parameters to be used by *i*th network output on target values. The processing parameters are applied by the processing functions to input values before the network uses them.

Side Effects

Whenever this property is altered, the output `processSettings`, `processedSize` and `processedRange` are defined by applying the process functions and parameters to `exampleOutput`. The *i*th layer's size is also updated to match `processedSize`.

net.outputs{i}.processSettings (read only)

This property holds a row cell array of processing function settings to be used by *i*th network output. The processing settings are found by applying the processing functions and parameters to `exampleOutput` and then used to provide consistent results to new target values before the network uses them. The processing settings are also applied in reverse to layer output values before being returned by the network.

net.outputs{i}.processedRange (read only)

This property defines the range of `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

net.outputs{i}.processedSize (read only)

This property defines the number of rows in the `exampleOutput` values after they have been processed with `processingFcns` and `processingParams`.

net.outputs{i}.size (read only)

This property defines the number of elements in the *i*th layer's output. It is always set to the size of the *i*th layer (`net.layers{i}.size`).

net.outputs{i}.userdata

This property provides a place for users to add custom information to the *i*th layer's output.

Biases

net.biases{i}.initFcn

This property defines the weight and bias initialization functions used to set the *i*th layer's bias vector (`net.b{i}`) if the network initialization function is `initlay` and the *i*th layer's initialization function is `initwb`.

net.biases{i}.learn

This property defines whether the *i*th bias vector is to be altered during training and adaption. It can be set to 0 or 1.

It enables or disables the bias's learning during calls to `adapt` and `train`.

net.biases{i}.learnFcn

This property defines which of the learning functions is used to update the *i*th layer's bias vector (`net.b{i}`) during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`.

For a list of functions, type `help nnlearn`.

Side Effects

Whenever this property is altered, the biases learning parameters (`net.biases{i}.learnParam`) are set to contain the fields and default values of the new function.

net.biases{i}.learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's bias. The fields of this property depend on the current learning function. Call `help` on the current learning function to get a description of what each field means.

net.biases{i}.size (read only)

This property defines the size of the *i*th layer's bias vector. It is always set to the size of the *i*th layer (`net.layers{i}.size`).

net.biases{i}.userdata

This property provides a place for users to add custom information to the *i*th layer's bias.

Input Weights

net.inputWeights{i,j}.delays

This property defines a tapped delay line between the *j*th input and its weight to the *i*th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

Side Effects

Whenever this property is altered, the weight's size (`net.inputWeights{i,j}.size`) and the dimensions of its weight matrix (`net.IW{i,j}`) are updated.

net.inputWeights{i,j}.initFcn

This property defines which of the Weight and Bias Initialization Functions is used to initialize the weight matrix (`net.IW{i,j}`) going to the *i*th layer from the *j*th input, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

net.inputWeights{i,j}.initSettings (read only)

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

net.inputWeights{i,j}.learn

This property defines whether the weight matrix to the *i*th layer from the *j*th input is to be altered during training and adaption. It can be set to 0 or 1.

net.inputWeights{i,j}.learnFcn

This property defines which of the learning functions is used to update the weight matrix (`net.IW{i,j}`) going to the *i*th layer from the *j*th input during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

net.inputWeights{i,j}.learnParam

This property defines the learning parameters and values for the current learning function of the *i*th layer's weight coming from the *j*th input.

The fields of this property depend on the current learning function (`net.inputWeights{i,j}.learnFcn`). Evaluate the above reference to see the fields of the current learning function.

Call `help` on the current learning function to get a description of what each field means.

net.inputWeights{i,j}.size (read only)

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th network input. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.IW{i,j}`). The first element is equal to the size of the *i*th layer

(`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the *j*th input:

```
length(net.inputWeights{i,j}.delays) * net.inputs{j}.size
```

net.inputWeights{i,j}.userdata

This property provides a place for users to add custom information to the (*i*,*j*)th input weight.

net.inputWeights{i,j}.weightFcn

This property defines which of the weight functions is used to apply the *i*th layer's weight from the *j*th input to that input. It can be set to the name of any weight function. The weight function is used to transform layer inputs during simulation and training.

For a list of functions, type `help nnweight`.

net.inputWeights{i,j}.weightParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

Layer Weights

net.layerWeights{i,j}.delays

This property defines a tapped delay line between the *j*th layer and its weight to the *i*th layer. It must be set to a row vector of increasing values. The elements must be either 0 or positive integers.

net.layerWeights{i,j}.initFcn

This property defines which of the weight and bias initialization functions is used to initialize the weight matrix (`net.LW{i,j}`) going to the *i*th layer from the *j*th layer, if the network initialization function is `initlay`, and the *i*th layer's initialization function is `initwb`. This function can be set to the name of any weight initialization function.

net.layerWeights{i,j}.initSettings (read only)

This property is set to values useful for initializing the weight as part of the configuration process that occurs automatically the first time a network is trained, or when the function `configure` is called on a network directly.

net.layerWeights{i,j}.learn

This property defines whether the weight matrix to the *i*th layer from the *j*th layer is to be altered during training and adaption. It can be set to 0 or 1.

net.layerWeights{i,j}.learnFcn

This property defines which of the learning functions is used to update the weight matrix (`net.LW{i,j}`) going to the *i*th layer from the *j*th layer during training, if the network training function is `trainb`, `trainc`, or `trainr`, or during adaption, if the network adapt function is `trains`. It can be set to the name of any weight learning function.

For a list of functions, type `help nnlearn`.

net.layerWeights{i,j}.learnParam

This property defines the learning parameters fields and values for the current learning function of the *i*th layer's weight coming from the *j*th layer. The fields of this property depend on the current learning function. Call `help` on the current net input function to get a description of each field.

net.layerWeights{i,j}.size (read only)

This property defines the dimensions of the *i*th layer's weight matrix from the *j*th layer. It is always set to a two-element row vector indicating the number of rows and columns of the associated weight matrix (`net.LW{i,j}`). The first element is equal to the size of the *i*th layer (`net.layers{i}.size`). The second element is equal to the product of the length of the weight's delay vectors and the size of the *j*th layer.

net.layerWeights{i,j}.userdata

This property provides a place for users to add custom information to the (*i*,*j*)th layer weight.

net.layerWeights{i,j}.weightFcn

This property defines which of the weight functions is used to apply the *i*th layer's weight from the *j*th layer to that layer's output. It can be set to the name of any weight function. The weight function is used to transform layer inputs when the network is simulated.

For a list of functions, type `help nnweight`.

net.layerWeights{i,j}.weightParam

This property defines the parameters of the layer's net input function. Call `help` on the current net input function to get a description of each field.

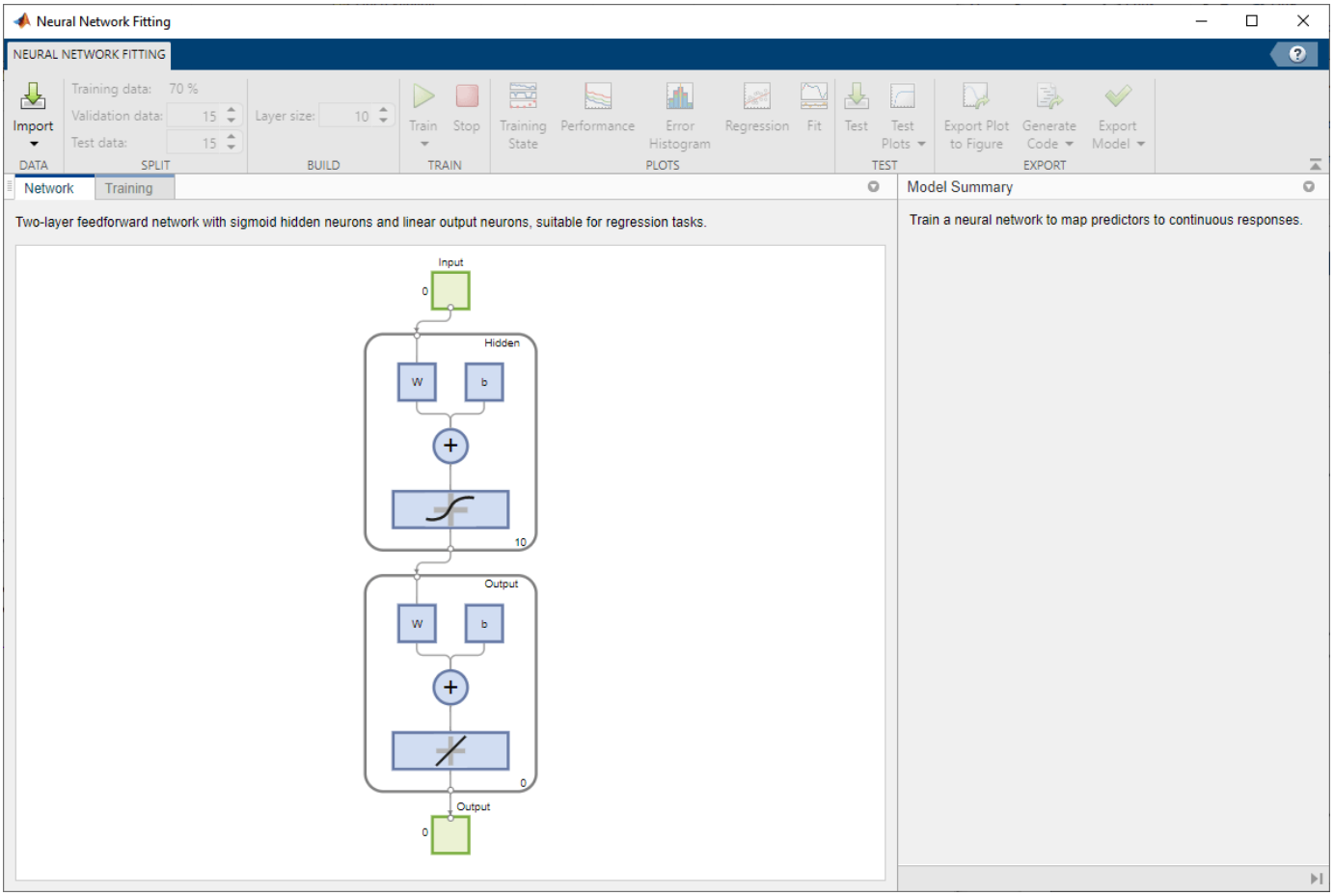
Function Approximation, Clustering, and Control Examples

Fit Data Using the Neural Net Fitting App

This example shows how to train a shallow neural network to fit data using the **Neural Net Fitting** app.

Open the **Neural Net Fitting** app using `nftool`.

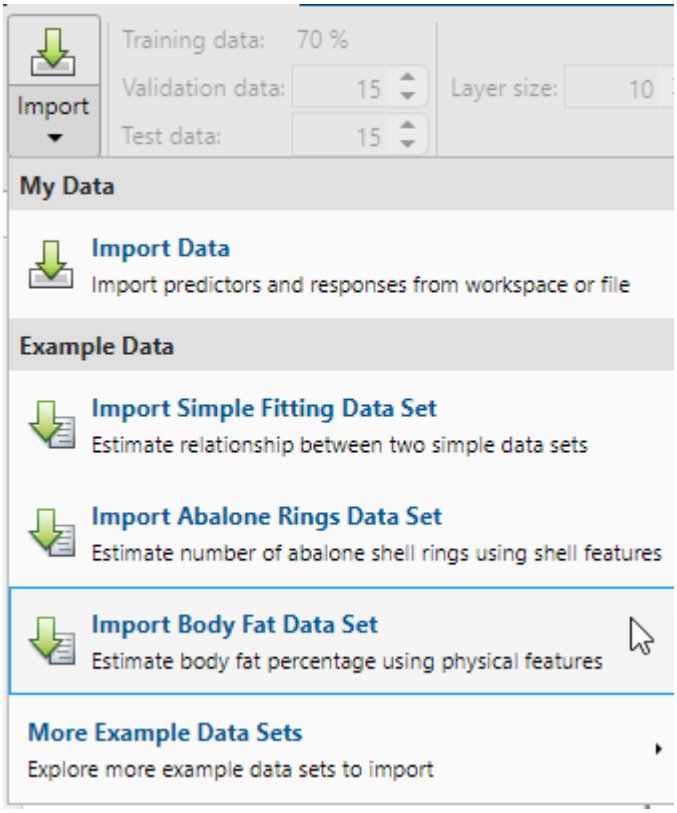
```
nftool
```



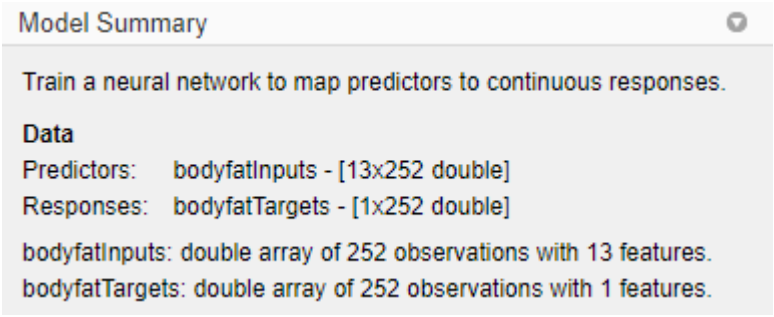
Select Data

The **Neural Net Fitting** app has example data to help you get started training a neural network.

To import example body fat data, select **Import > Import Body Fat Data Set**. You can use this data set to train a neural network to estimate the body fat of someone from various measurements. If you import your own data from file or the workspace, you must specify the predictors and responses, and if the observations are in rows or columns.



Information about the imported data appears in the **Model Summary**. This data set contains 252 observations, each with 13 features. The responses contain the body fat percentage for each observation.



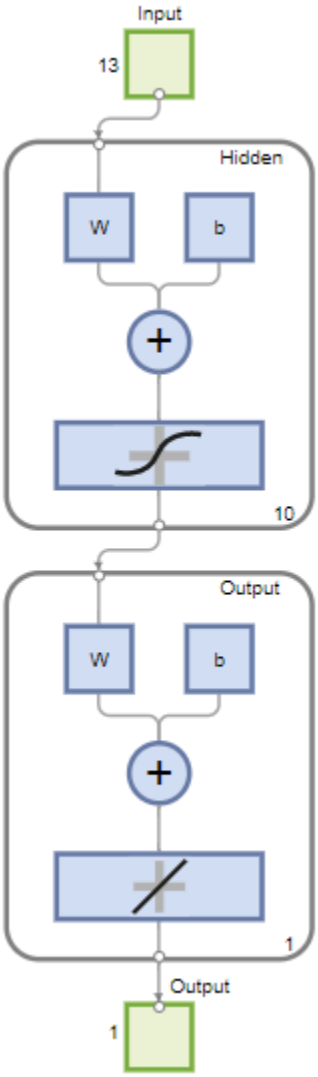
Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training” on page 22-9.

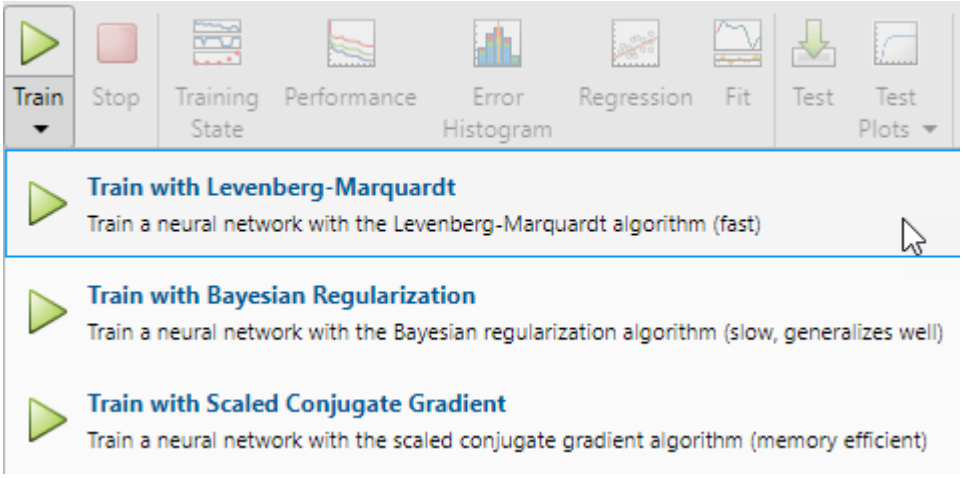
Create Network

The network is a two-layer feedforward network with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. The **Layer size** value defines the number of hidden neurons. Keep the default layer size, 10. You can see the network architecture in the **Network** pane. The network plot updates to reflect the input data. In this example, the data has 13 inputs (features) and one output.



Train Network

To train the network, select **Train** > **Train with Levenberg-Marquardt**. This is the default training algorithm and the same as clicking **Train**.



Training with Levenberg-Marquardt (`trainlm`) is recommended for most problems. For noisy or small problems, Bayesian Regularization (`trainbr`) can obtain a better solution, at the cost of taking longer. For large problems, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").

Training Results

Training finished: Met validation criterion

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	9	1000
Elapsed time	-	00:00:00	-
Performance	466	8.7	0
Gradient	2.67e+03	20.5	1e-07
Mu	0.001	0.1	1e+10
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.

Algorithm

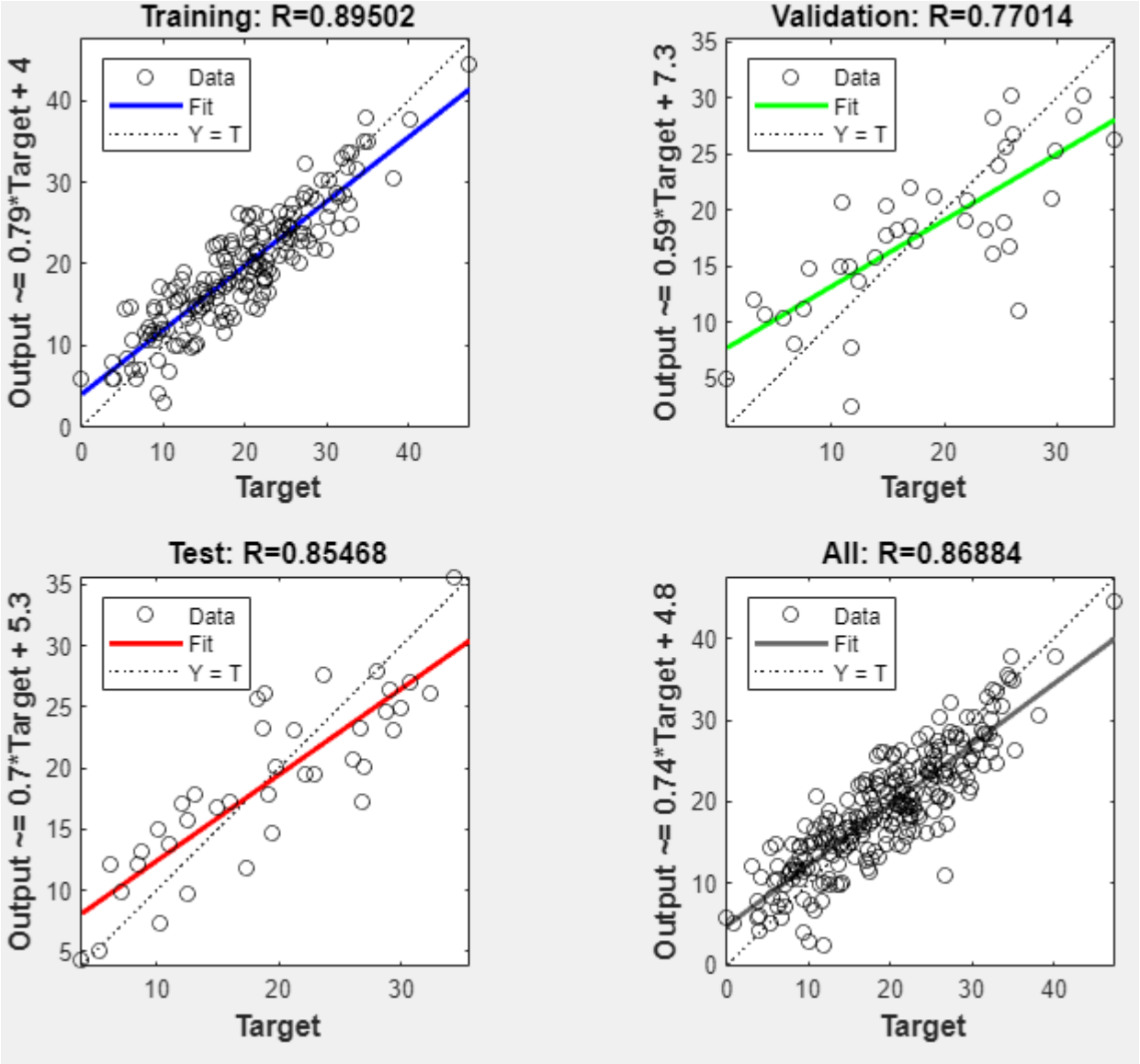
Data division: Random
Training algorithm: Levenberg-Marquardt
Performance: Mean squared error

Training Results

Training start time: 02-Jul-2021 12:10:54
Layer size: 10

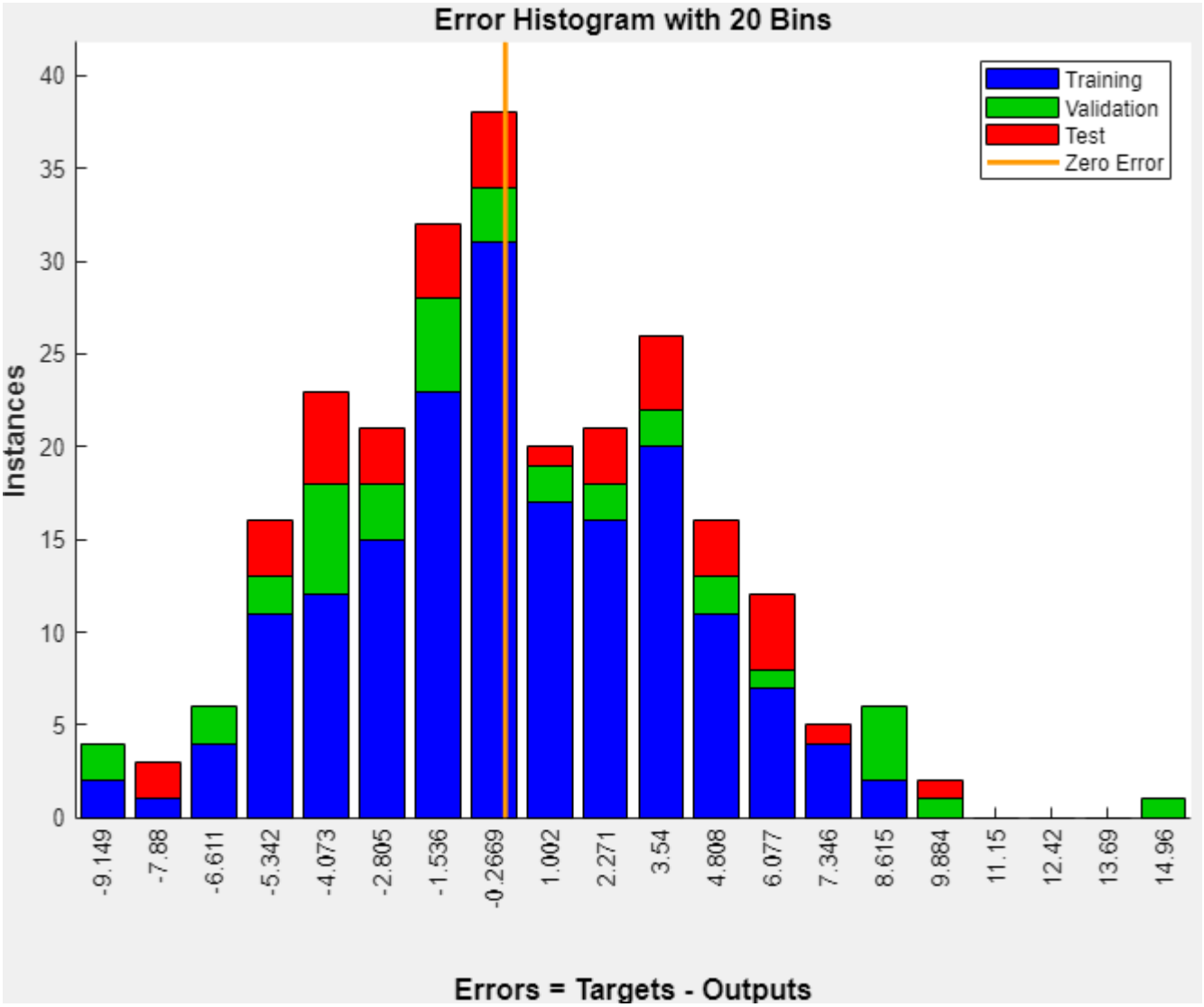
	Observations	MSE	R
Training	176	13.3810	0.8950
Validation	38	32.4817	0.7701
Test	38	19.1691	0.8547

You can further analyze the results by generating plots. To plot the linear regression, in the **Plots** section, click **Regression**. The regression plot displays the network predictions (output) with respect to responses (target) for the training, validation, and test sets.



For a perfect fit, the data should fall along a 45 degree line, where the network outputs are equal to the responses. For this problem, the fit is reasonably good for all of the data sets. If you require more accurate results, you can retrain the network by clicking **Train** again. Each training will have different initial weights and biases of the network, and can produce an improved network after retraining.

View the error histogram to obtain additional verification of network performance. In the **Plots** section, click **Error Histogram**.



The blue bars represent training data, the green bars represent validation data, and the red bars represent testing data. The histogram provides an indication of outliers, which are data points where the fit is significantly worse than most of the data. It is a good idea to check the outliers to determine if the data is poor, or if those data points are different than the rest of the data set. If the outliers are valid data points, but are unlike the rest of the data, then the network is extrapolating for these points. You should collect more data that looks like the outlier points and retrain the network.

If you are unhappy with the network performance, you can do one of the following:

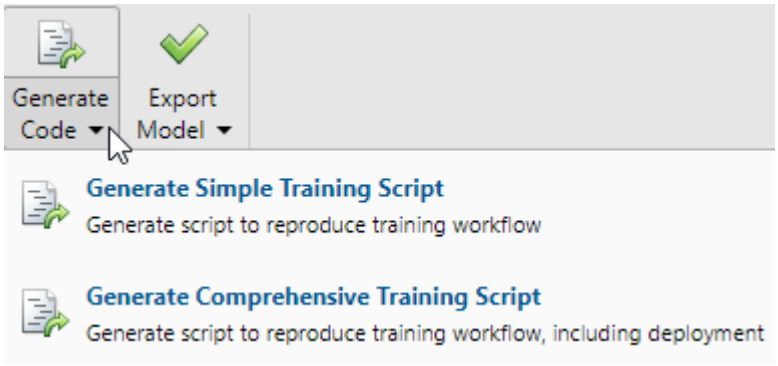
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Reducing the number of neurons can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test results. You can also generate plots to analyze the additional test data results.

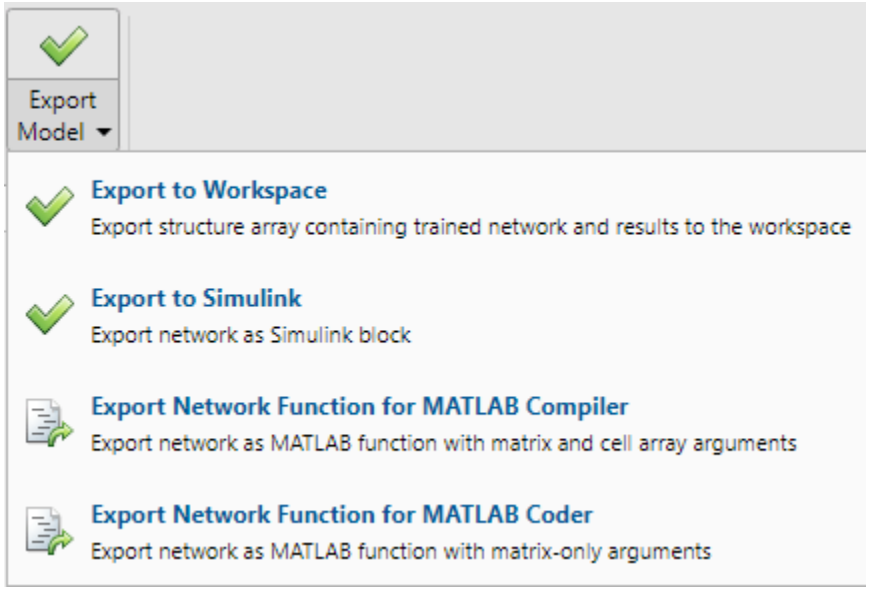
Generate Code

Select **Generate Code > Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command line functionality of the toolbox to customize the training process. In “Fit Data Using Command-Line Functions”, you will investigate the generated scripts in more detail.



Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model > Export to Workspace**.



See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `trainlm` | `fitnet`

Related Examples

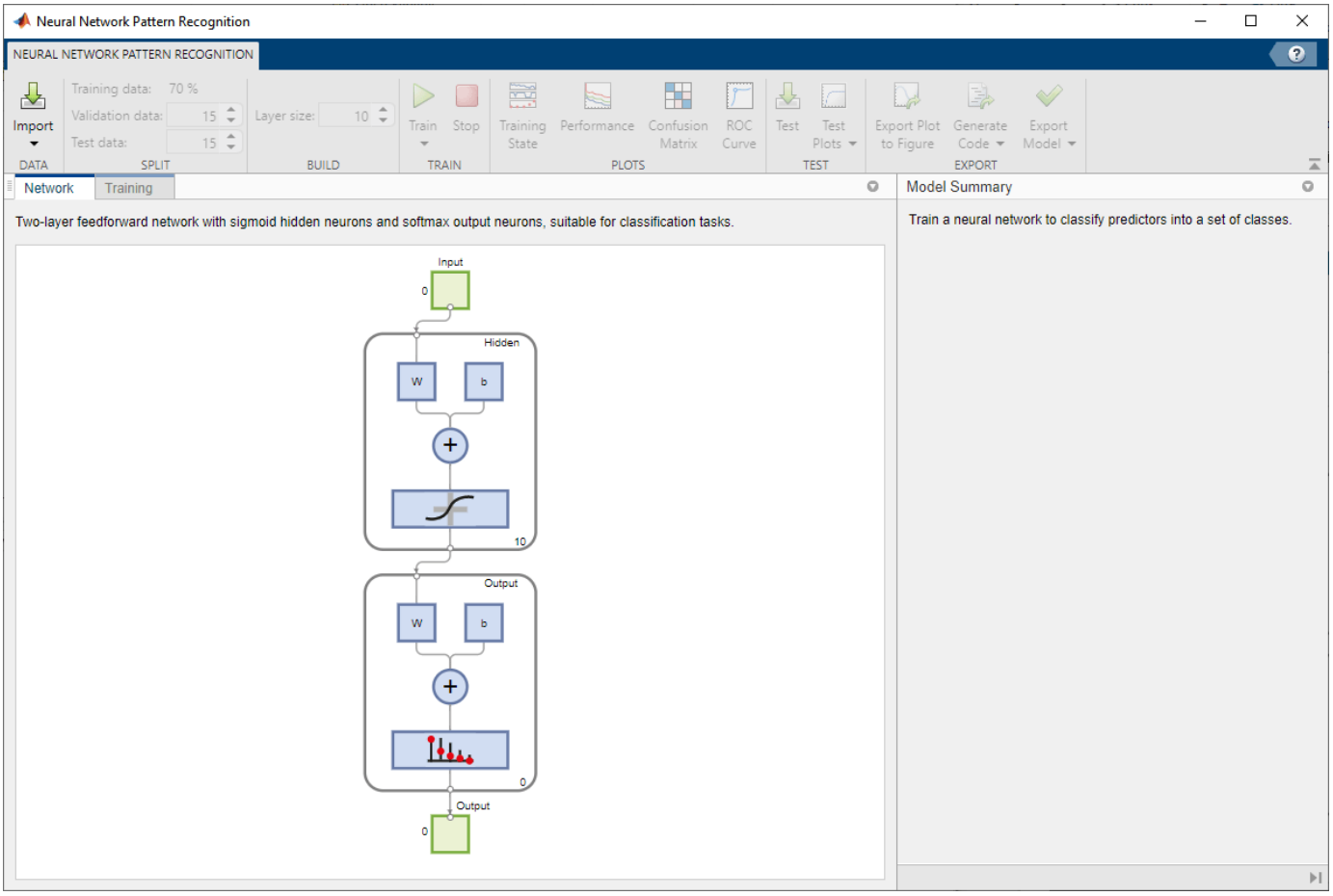
- “Fit Data with a Shallow Neural Network”
- “Classify Patterns with a Shallow Neural Network”
- “Cluster Data with a Self-Organizing Map”
- “Shallow Neural Network Time-Series Prediction and Modeling”

Classify Patterns Using the Neural Net Pattern Recognition App

This example shows how to train a shallow neural network to classify patterns using the **Neural Net Pattern Recognition** app.

Open the **Neural Net Pattern Recognition** app using `nprtool`.

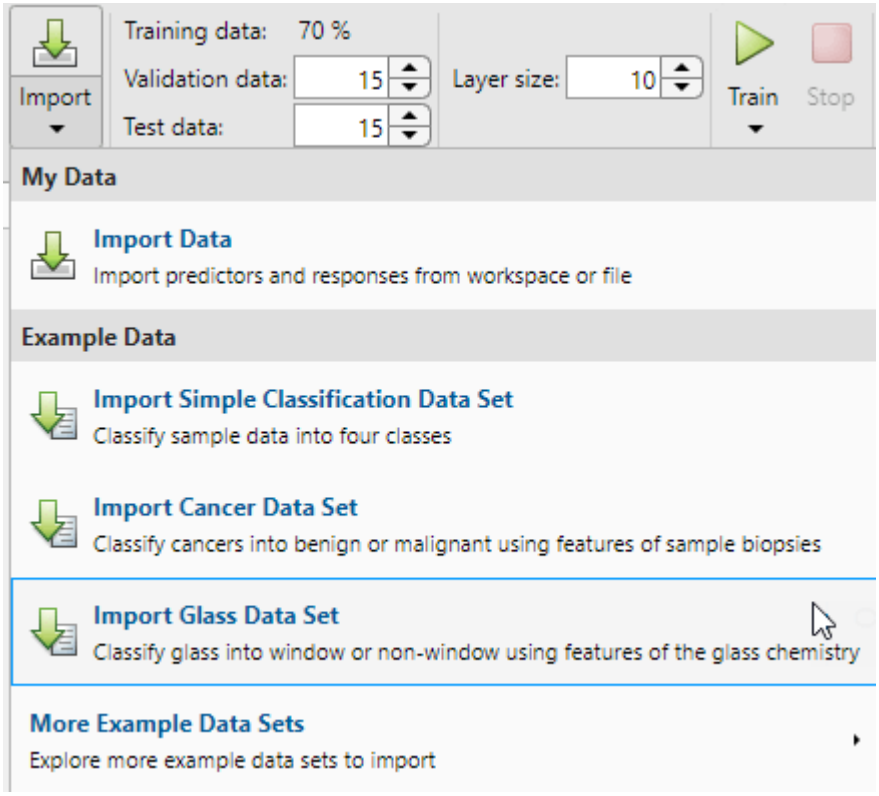
```
nprtool
```



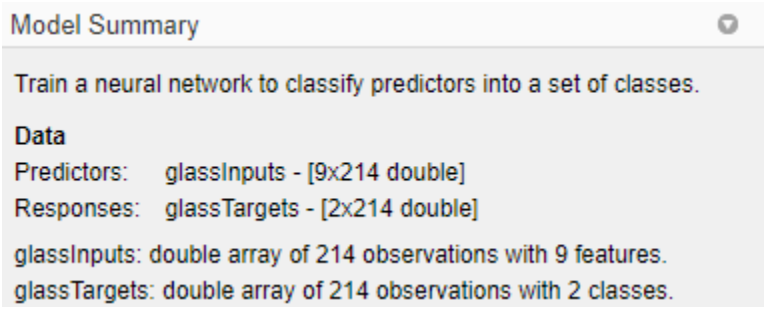
Select Data

The **Neural Net Pattern Recognition** app has example data to help you get started training a neural network.

To import example glass classification data, select **Import** > **Import Glass Data Set**. You can use this data set to train a neural network to classify glass as window or non-window, using properties of the glass chemistry. If you import your own data from file or the workspace, you must specify the predictors and responses, and if the observations are in rows or columns.



Information about the imported data appears in the **Model Summary**. This data set contains 214 observations, each with 9 features. Each observation is classified into one of two classes: window or non-window.



Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

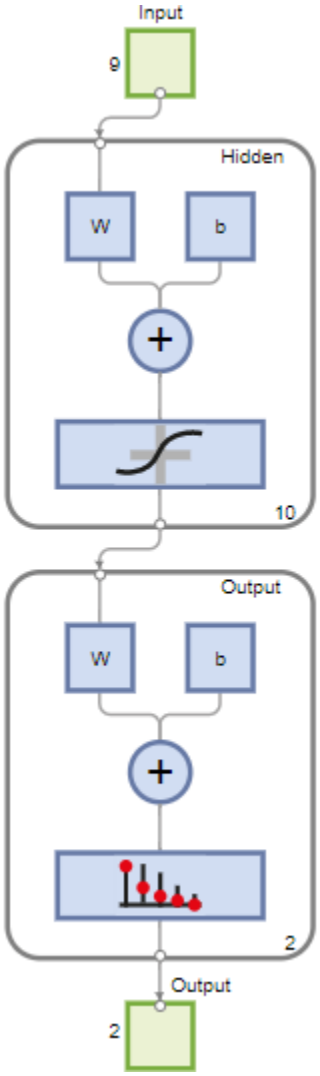
- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training” on page 22-9.

Create Network

The network is a two-layer feedforward network with a sigmoid transfer function in the hidden layer and a softmax transfer function in the output layer. The size of the hidden layer corresponds to the

number of hidden neurons. The default layer size is `10`. You can see the network architecture in the **Network** pane. The number of output neurons is set to 2, which is equal to the number of classes specified by the response data.



Train Network

To train the network, click **Train**.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").

Training Results

Training finished: Met validation criterion

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	14	1000
Elapsed time	-	00:00:00	-
Performance	0.418	0.0523	0
Gradient	0.808	0.017	1e-06
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.

Algorithm

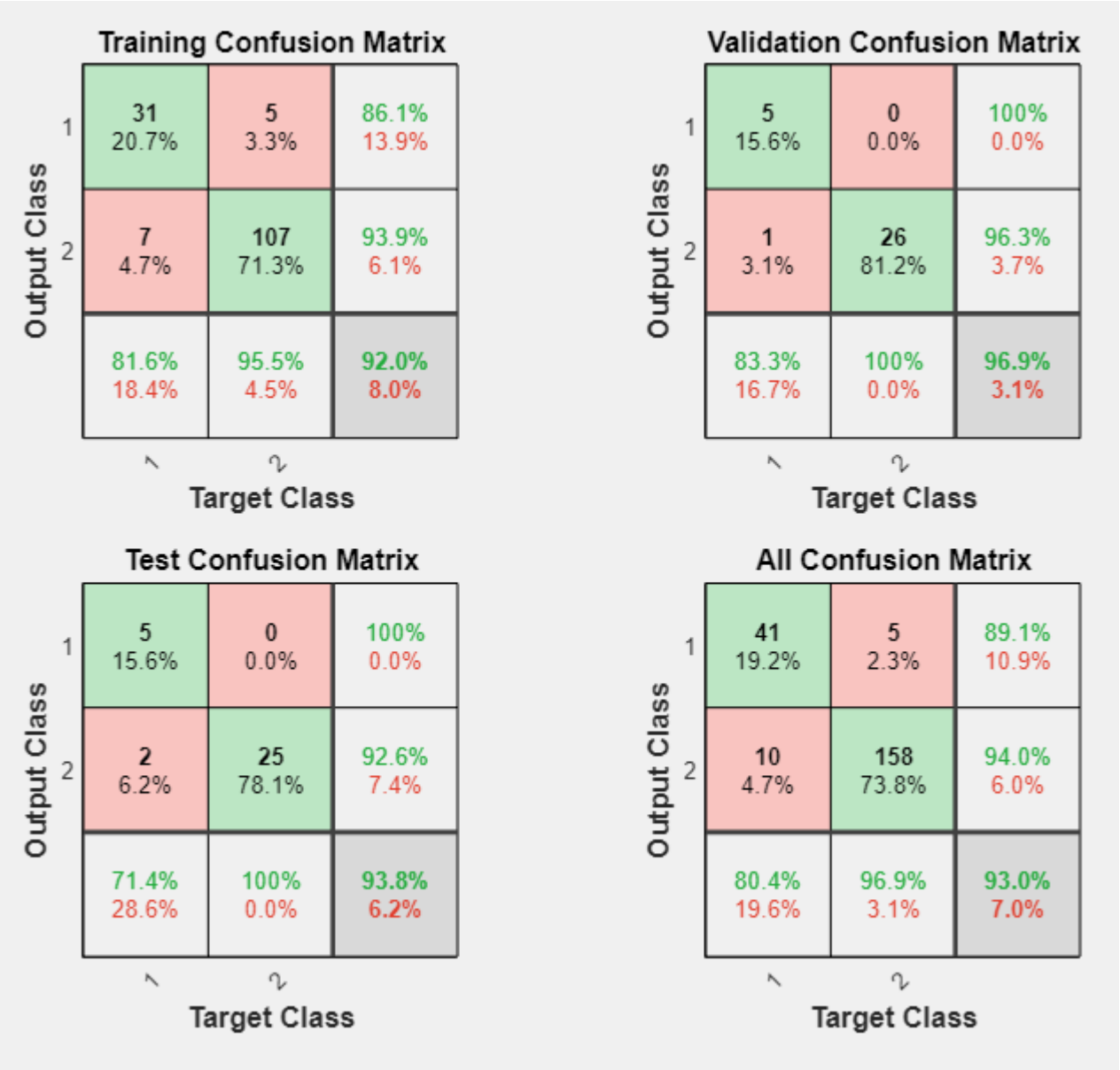
Data division: Random
Training algorithm: Scaled conjugate gradient
Performance: Cross-entropy error

Training Results

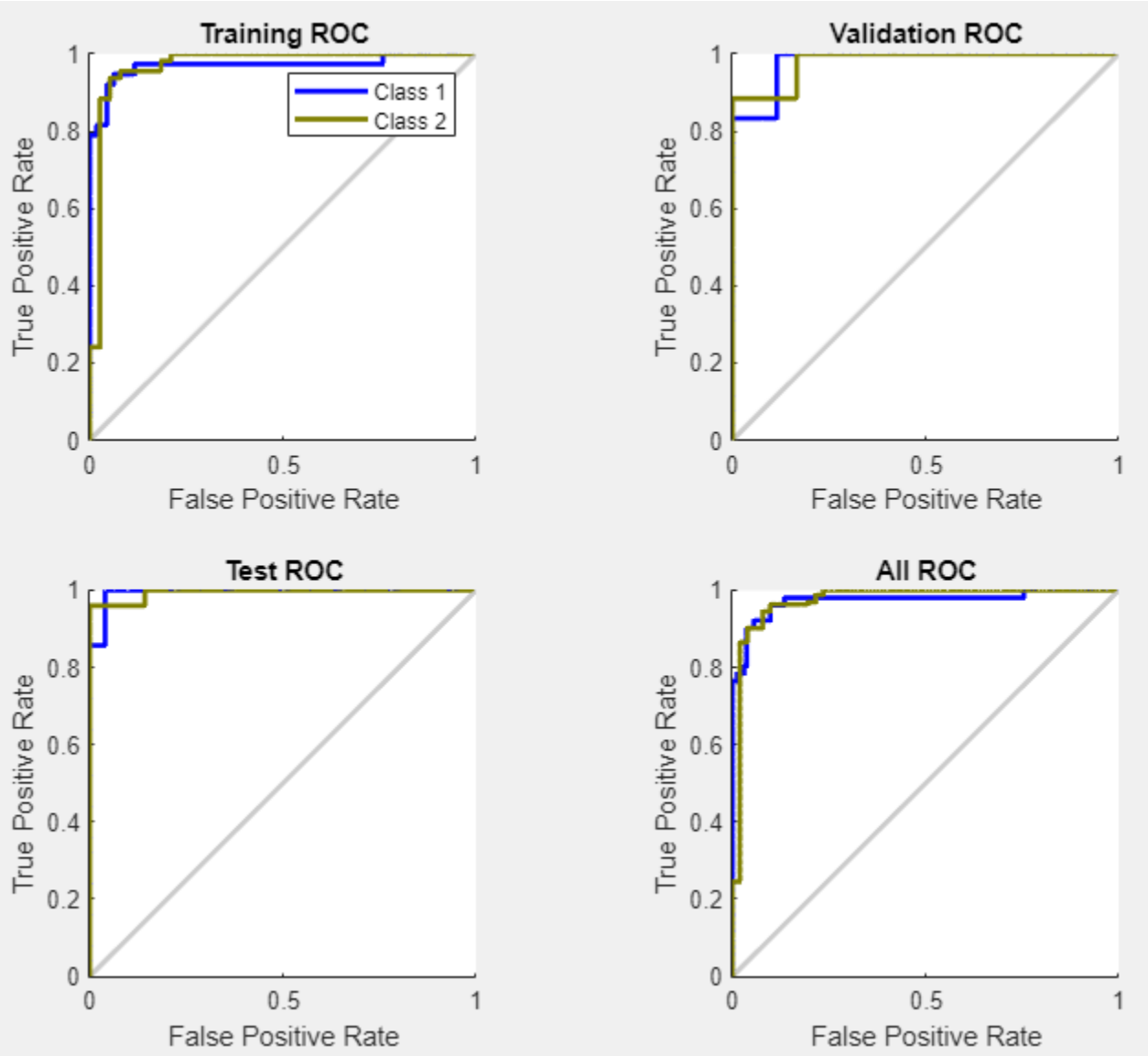
Training start time: 02-Jul-2021 11:53:16
Layer size: 10

	Observations	Cross-entropy	Error
Training	150	0.0749	0.0800
Validation	32	0.0798	0.0312
Test	32	0.0720	0.0625

You can further analyze the results by generating plots. To plot the confusion matrices, in the **Plots** section, click **Confusion Matrix**. The network outputs are very accurate, as you can see by the high numbers of correct classifications in the green squares (diagonal) and the low numbers of incorrect classifications in the red squares (off-diagonal).



View the ROC curve to obtain additional verification of network performance. In the **Plots** section, click **ROC Curve**.



The colored lines in each axis represent the ROC curves. The ROC curve is a plot of the true positive rate (sensitivity) versus the false positive rate (1 - specificity) as the threshold is varied. A perfect test would show points in the upper-left corner, with 100% sensitivity and 100% specificity. For this problem, the network performs very well.

If you are unhappy with the network performance, you can do one of the following:

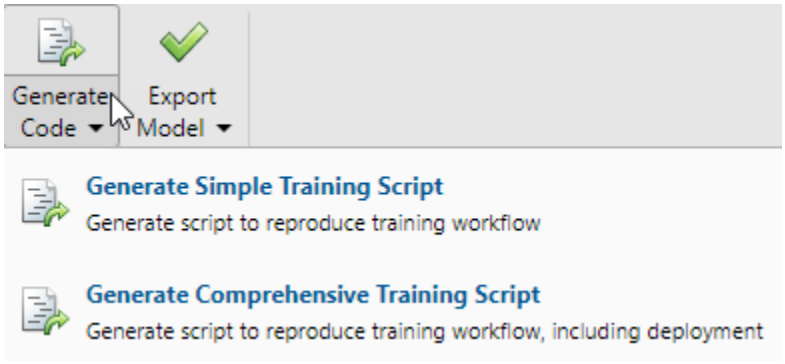
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Reducing the number of neurons can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test results. You can also generate plots to analyze the additional test results.

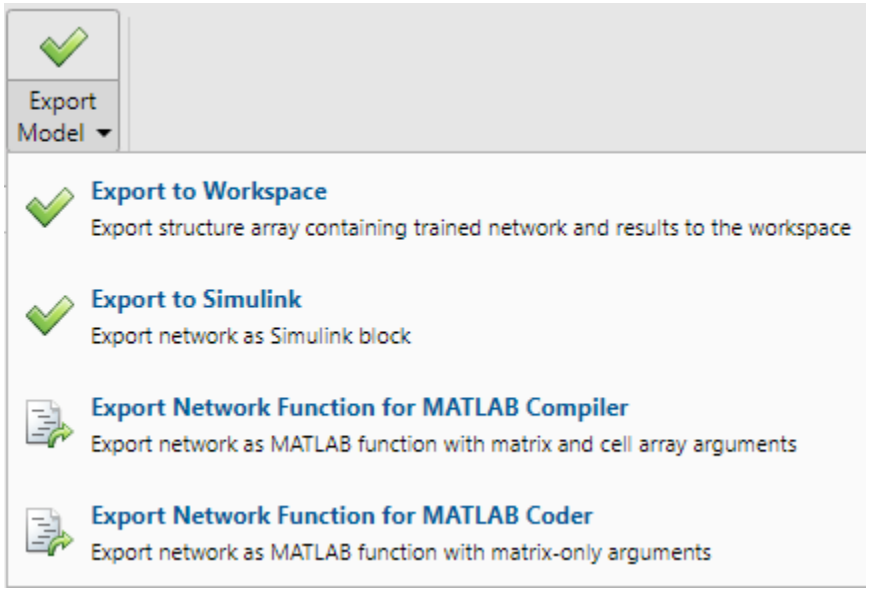
Generate Code

Select **Generate Code > Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command line functionality of the toolbox to customize the training process. In “Classify Patterns Using Command-Line Functions”, you will investigate the generated scripts in more detail.



Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ and other MATLAB code generation tools. To export your trained network and results, select **Export Model > Export to Workspace**.



See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `trainscg`

Related Examples

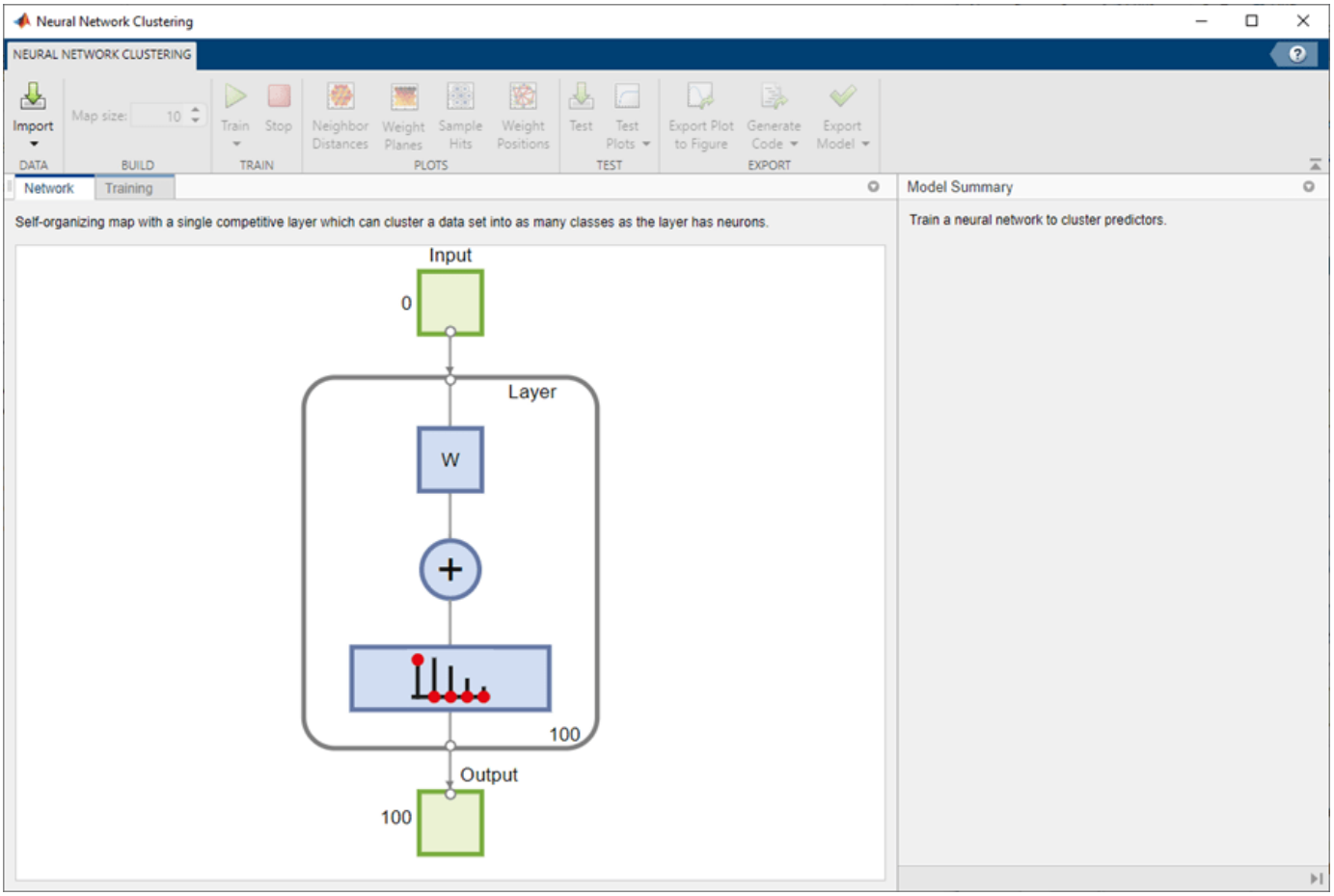
- “Classify Patterns with a Shallow Neural Network”
- “Fit Data with a Shallow Neural Network”
- “Cluster Data with a Self-Organizing Map”
- “Shallow Neural Network Time-Series Prediction and Modeling”

Cluster Data Using the Neural Net Clustering App

This example shows how to train a shallow neural network to cluster data using the **Neural Net Clustering** app.

Open the **Neural Net Clustering** app using `nctool`.

```
nctool
```

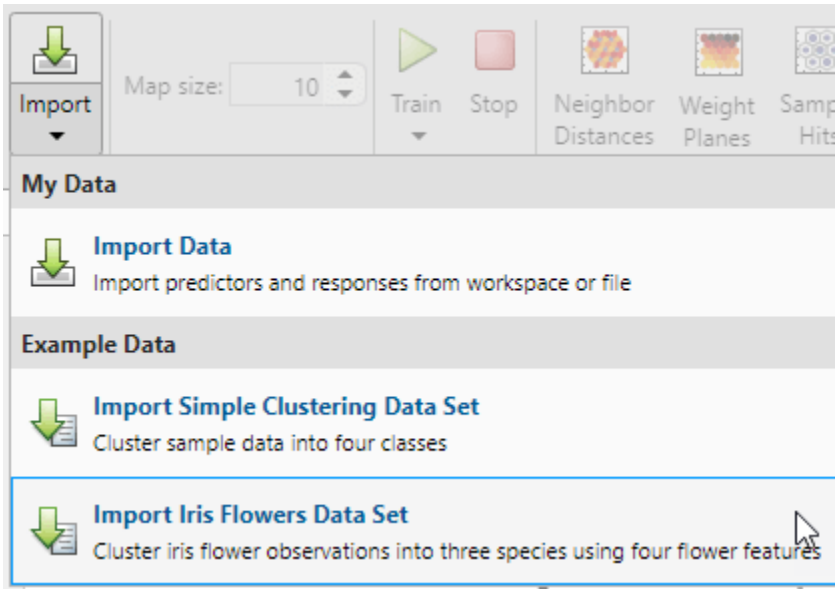


Select Data

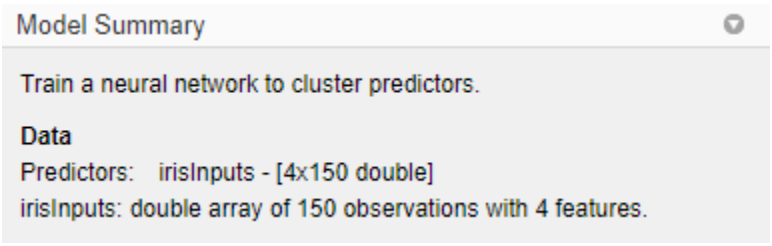
The **Neural Net Clustering** app has example data to help you get started training a neural network.

To import the example iris flower clustering data, select **Import** > **Import Iris Flowers Data Set**. If you import your own data from file or the workspace, you must specify the predictors and if the

observations are in rows or columns.



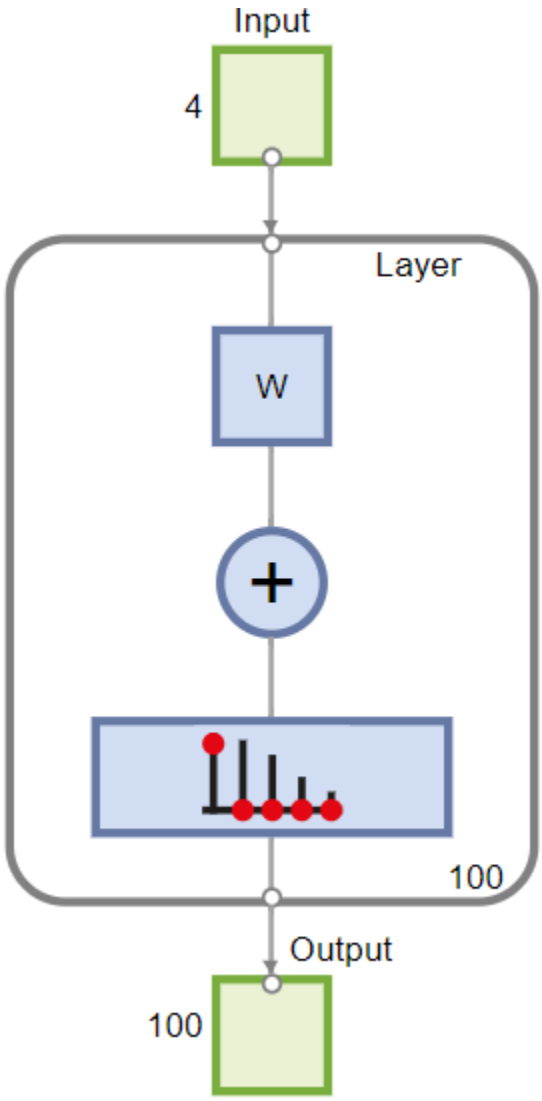
Information about the imported data appears in the **Model Summary**. This data set contains 150 observations, each with four features.



Create Network

For clustering problems, the self-organizing feature map (SOM) is the most commonly used network. This network has one layer, with neurons organized in a grid. Self-organizing maps learn to cluster data based on similarity. For more information on the SOM, see “Cluster with Self-Organizing Map Neural Network” on page 26-8.

To create the network, specify the map size, this corresponds to the number of rows and columns in the grid. For this example, set the **Map size** value to 10, this corresponds to a grid with 10 rows and 10 columns. The total number of neurons is equal to the number of points in the grid, in this example, the map has 100 neurons. You can see the network architecture in the **Network** pane.



Train Network

To train the network, click **Train**. In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the maximum number of epochs is reached.

Training Results

Training finished: Reached maximum number of epochs

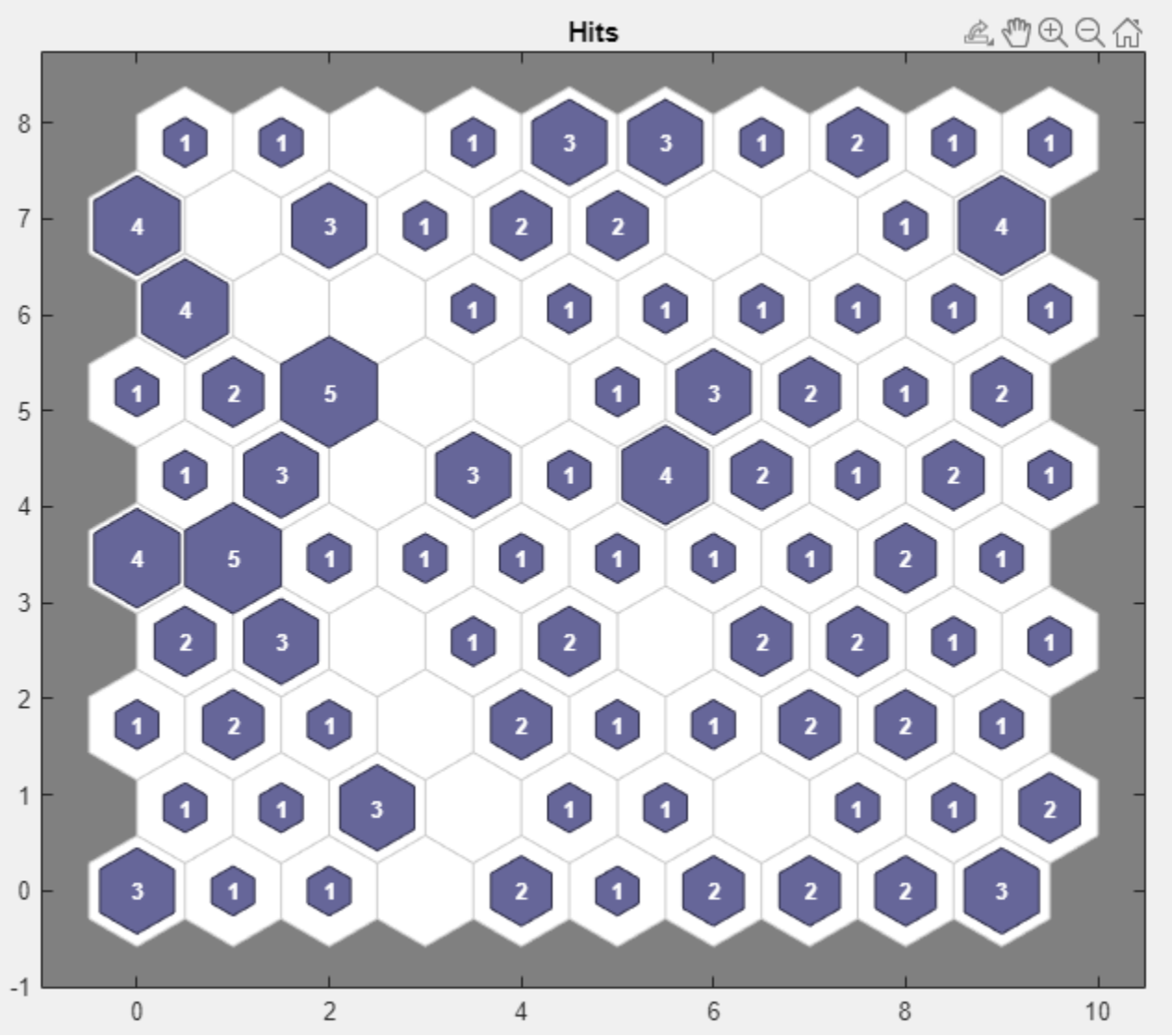
Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	200	200
Elapsed time	-	00:00:00	-

Analyze Results

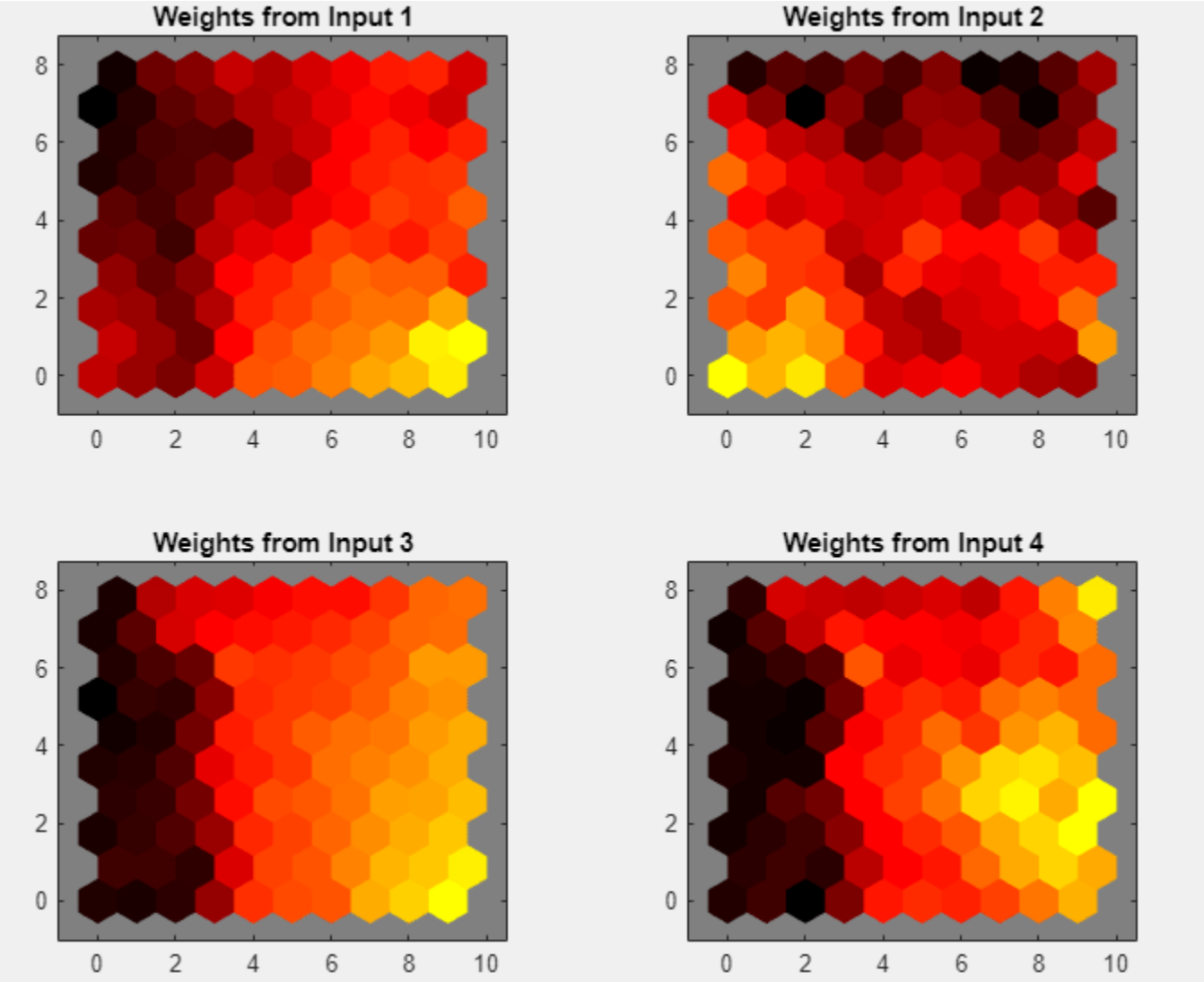
To analyze the training results, generate plots. For SOM training, the weight vector associated with each neuron moves to become the center of a cluster of input vectors. In addition, neurons that are adjacent to each other in the topology should also move close to each other in the input space, therefore it is possible to visualize a high-dimensional inputs space in the two dimensions of the network topology. The default topology of the SOM is hexagonal.

To plot the SOM Sample Hits, in the **Plots** section, click **Sample Hits**. This figure shows the neuron locations in the topology, and indicates how many of the observations are associated with each of the neurons (cluster centers). The topology is a 10-by-10 grid, so there are 100 neurons. The maximum number of hits associated with any neuron is 5. Thus, there are 5 input vectors in that cluster.



Plot the weight planes (also referred to as *component planes*). In the **Plots** section, click **Weight Planes**. This figure shows a weight plane for each element of the input features (four, in this example). The plot shows the weights that connect each input to each of the neurons, with darker

colors representing larger weights. If the connection patterns of two features are very similar, you can assume that the features are highly correlated.



If you are unhappy with the network performance, you can do one of the following:

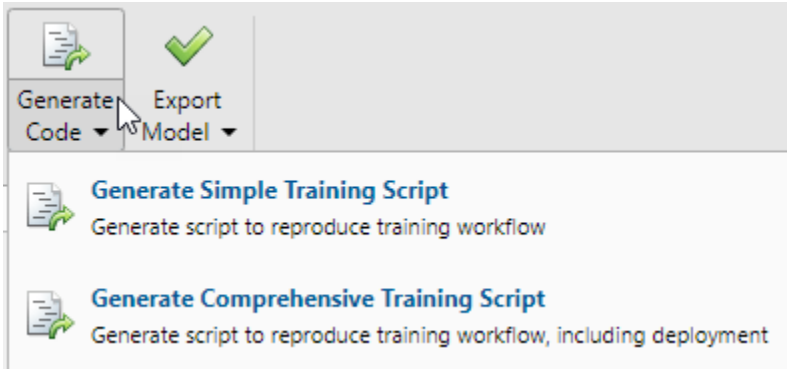
- Train the network again. Each training will have different initial weights and biases of the network, and can produce an improved network after retraining.
- Increase the number of neurons by increasing the map size.
- Use a larger training data set.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. Generate plots to analyze the additional test results.

Generate Code

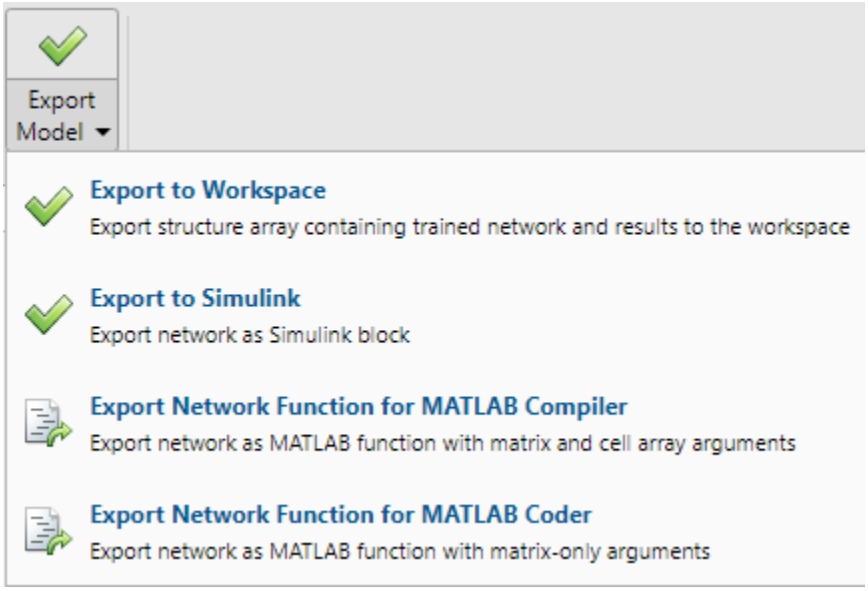
Select **Generate Code > Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn

how to use the command-line functionality of the toolbox to customize the training process. In "Cluster Data Using Command-Line Functions", you will investigate the generated scripts in more detail.



Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model > Export to Workspace**.



See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `train`

Related Examples

- "Cluster Data with a Self-Organizing Map"
- "Fit Data with a Shallow Neural Network"
- "Classify Patterns with a Shallow Neural Network"

- “Shallow Neural Network Time-Series Prediction and Modeling”

Fit Time Series Data Using the Neural Net Time Series App

This example shows how to train a shallow neural network to fit time series data using the **Neural Net Time Series** app.

Open the **Neural Net Time Series** app using `ntstool`.

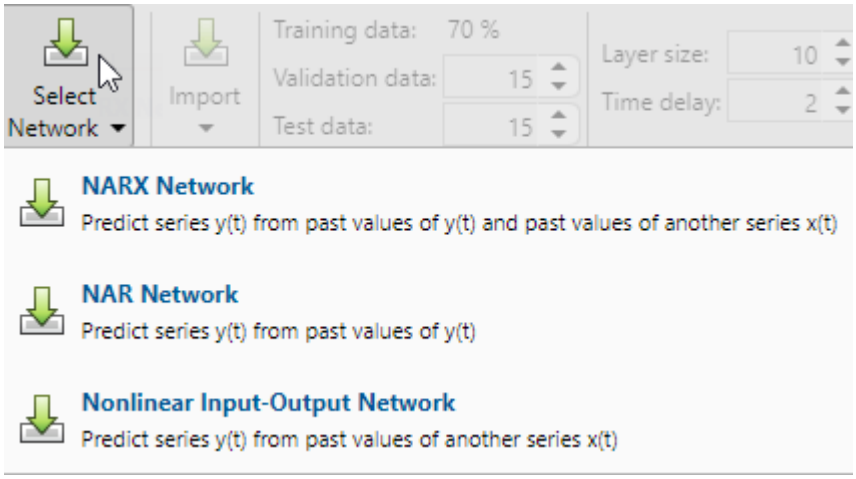
```
ntstool
```

Select Network

You can use the **Neural Net Time Series** app to solve three different kinds of time series problems.

- In the first type of time series problem, you would like to predict future values of a time series $y(t)$ from past values of that time series and past values of a second time series $x(t)$. This form of prediction is called nonlinear autoregressive network with exogenous (external) input, or NARX.
- In the second type of time series problem, there is only one series involved. The future values of a time series $y(t)$ are predicted only from past values of that series. This form of prediction is called nonlinear autoregressive, or NAR.
- The third time series problem is similar to the first type, in that two series are involved, an input series (predictors) $x(t)$ and an output series (responses) $y(t)$. Here you want to predict values of $y(t)$ from previous values of $x(t)$, but without knowledge of previous values of $y(t)$.

For this example, use a NARX network. Click **Select Network > NARX Network**.

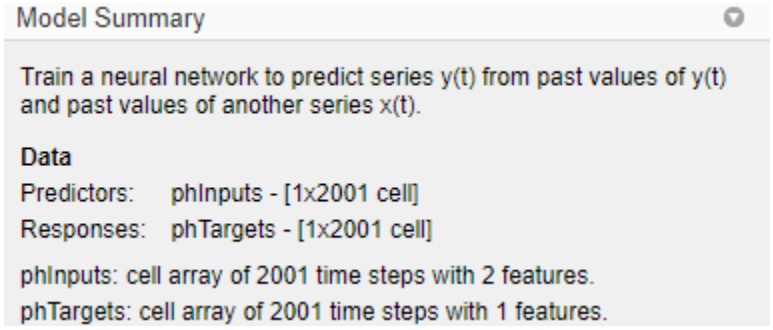


Select Data

The **Neural Net Time Series** app has example data to help you get started training a neural network.

To import example pH neutralization process data, select **Import > More Example Data Sets > Import pH Neutralization Data Set**. You can use this data set to train a neural network to predict the pH of a solution using acid and base solution flow. If you import your own data from file or the workspace, you must specify the predictors and responses.

Information about the imported data appears in the **Model Summary**. This data set contains 2001 time steps. The predictors have two features (acid and base solution flow) and the responses have a single feature (solution pH).



Split the data into training, validation, and test sets. Keep the default settings. The data is split into:

- 70% for training.
- 15% to validate that the network is generalizing and to stop training before overfitting.
- 15% to independently test network generalization.

For more information on data division, see “Divide Data for Optimal Neural Network Training” on page 22-9.

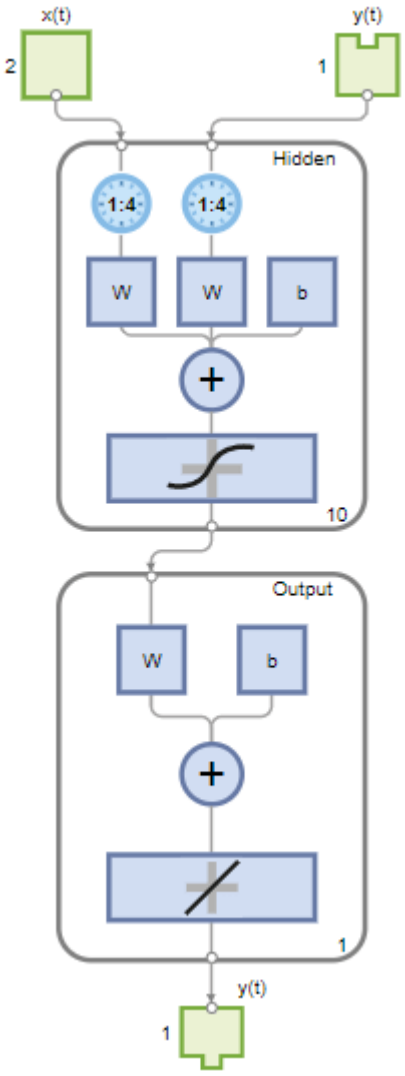
Create Network

The standard NARX network is a two-layer feedforward network, with a sigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This network also uses tapped delay lines to store previous values of the $x(t)$ and $y(t)$ sequences. Note that the output of the NARX network, $y(t)$, is fed back to the input of the network (through delays), since $y(t)$ is a function of $y(t–1), y(t–2), \ldots, y(t–d)$. However, for efficient training this feedback loop can be opened.

Because the true output is available during the training of the network, you can use the open-loop architecture shown below, in which the true output is used instead of feeding back the estimated output. This has two advantages. The first is that the input to the feedforward network is more accurate. The second is that the resulting network has a purely feedforward architecture, and therefore a more efficient algorithm can be used for training. This network is discussed in more detail in “Design Time Series NARX Feedback Neural Networks” on page 23-16.

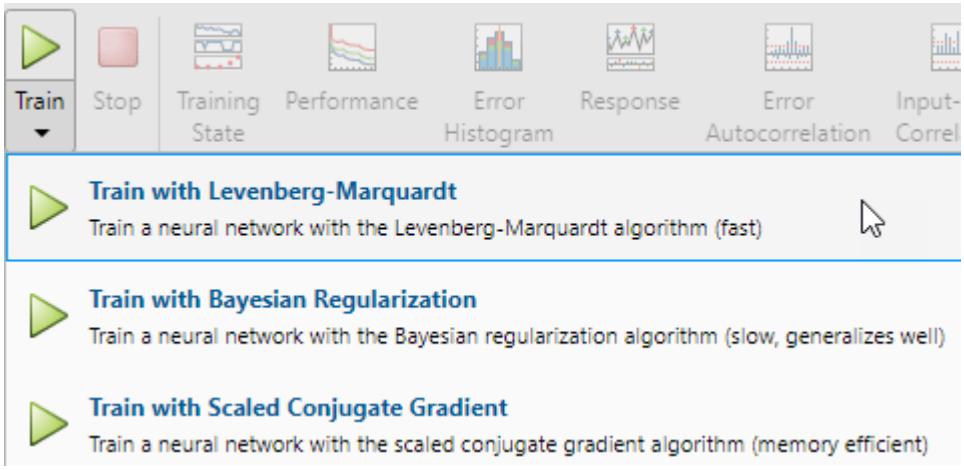
The **Layer size** value defines the number of hidden neurons. Keep the default layer size, `10`. Change the **Time delay** value to `4`. You might want to adjust these numbers if the network training performance is poor.

You can see the network architecture in the **Network** pane.



Train Network

To train the network, select **Train > Train with Levenberg-Marquardt**. This is the default training algorithm and the same as clicking **Train**.



Training with Levenberg-Marquardt (`trainlm`) is recommended for most problems. For noisy or small problems, Bayesian Regularization (`trainbr`) can obtain a better solution, at the cost of taking longer. For large problems, Scaled Conjugate Gradient (`trainscg`) is recommended as it uses gradient calculations which are more memory efficient than the Jacobian calculations the other two algorithms use.

In the **Training** pane, you can see the training progress. Training continues until one of the stopping criteria is met. In this example, training continues until the validation error increases consecutively for six iterations ("Met validation criterion").

Training Results

Training finished: Met validation criterion

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	43	1000
Elapsed time	-	00:00:00	-
Performance	69	0.00212	0
Gradient	125	0.0068	1e-07
Mu	0.001	1e-06	1e+10
Validation Checks	0	6	6

Analyze Results

The **Model Summary** contains information about the training algorithm and the training results for each data set.

Algorithm

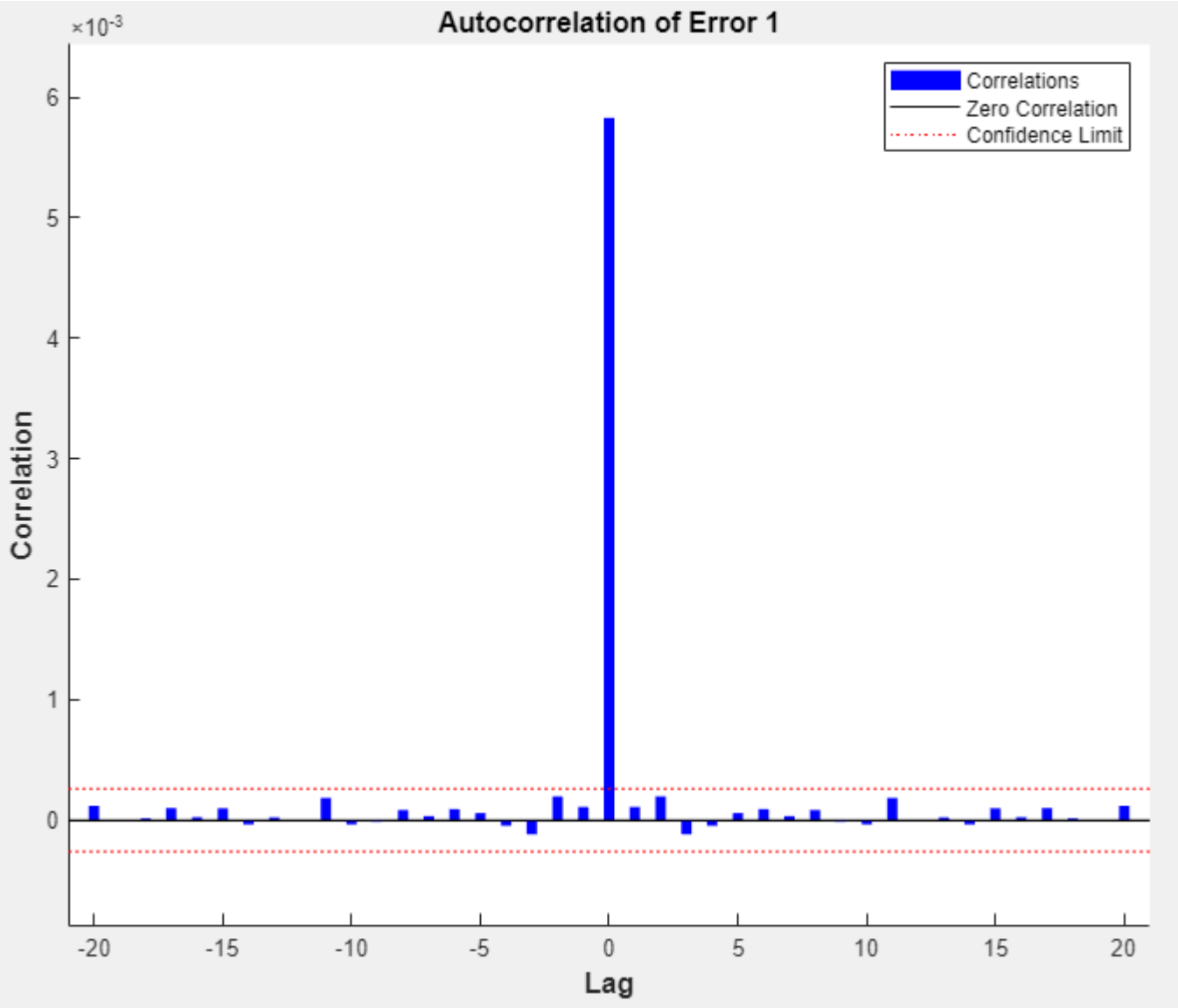
Data division:	Random
Training algorithm:	Levenberg-Marquardt
Performance:	Mean squared error

Training Results

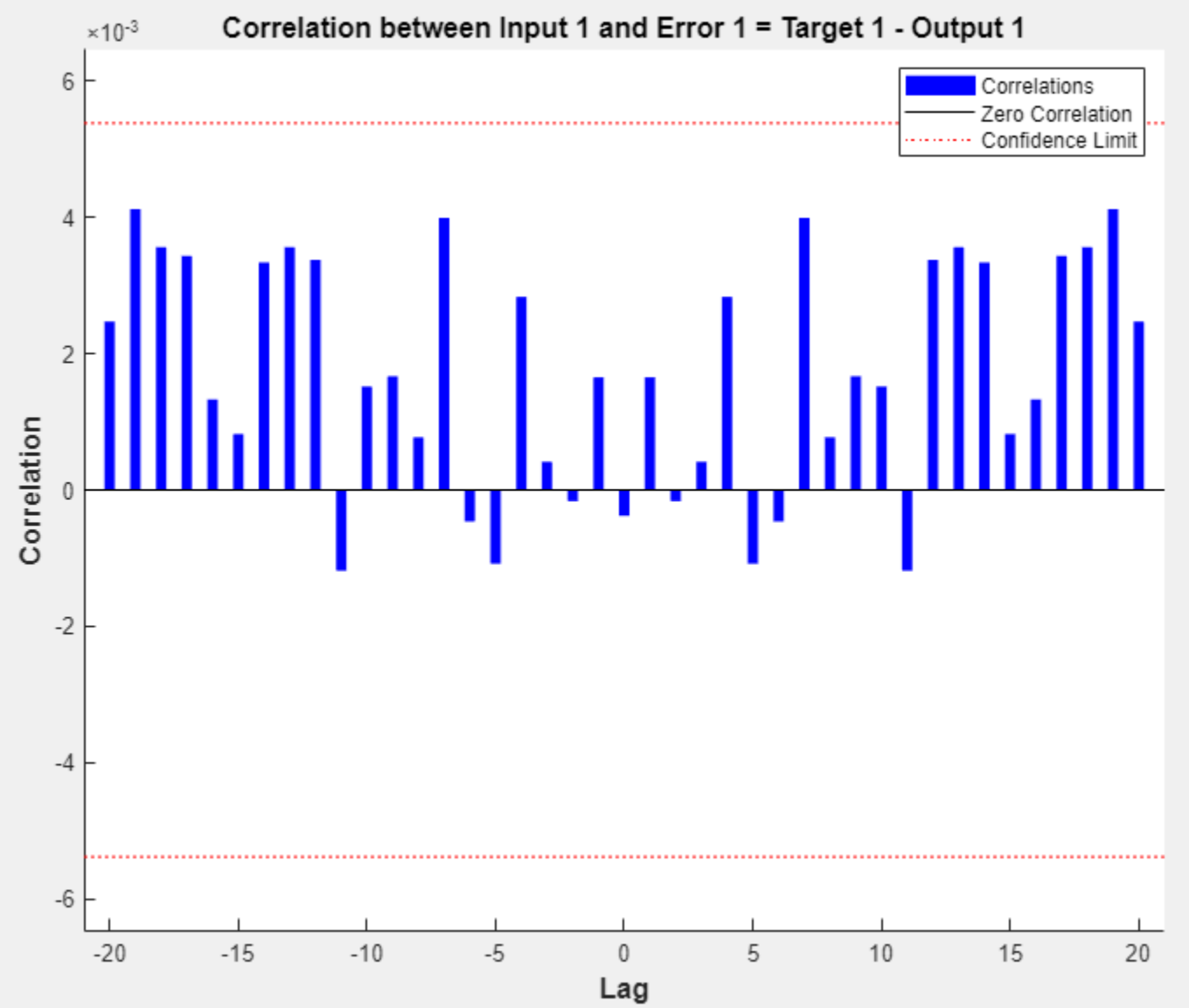
Training start time:	02-Jul-2021 11:27:58
Layer size:	10
Time delay:	4

	Observations	MSE	R
Training	1397	0.0021	0.9998
Validation	300	0.0069	0.9994
Test	300	0.0220	0.9985

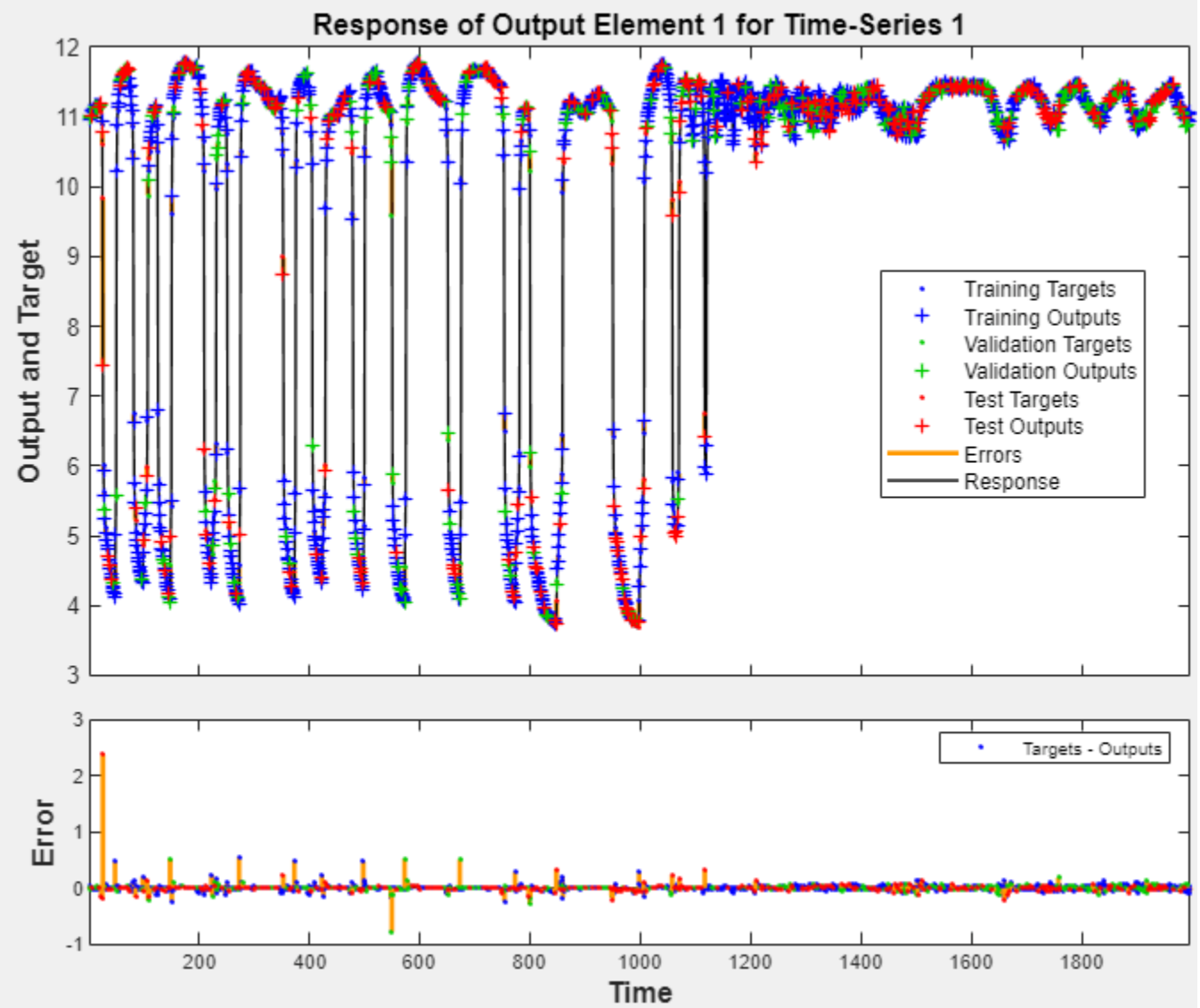
You can further analyze the results by generating plots. To plot the error autocorrelation, in the **Plots** section, click **Error Autocorrelation**. The autocorrelation plot describes how the prediction errors are related in time. For a perfect prediction model, there should only be one nonzero value of the autocorrelation function, and it should occur at zero lag (this is the mean square error). This would mean that the prediction errors were completely uncorrelated with each other (white noise). If there was significant correlation in the prediction errors, then it should be possible to improve the prediction - perhaps by increasing the number of delays in the tapped delay lines. In this case, the correlations, except for the one at zero lag, fall approximately within the 95% confidence limits around zero, so the model seems to be adequate. If even more accurate results were required, you could retrain the network. This will change the initial weights and biases of the network, and may produce an improved network after retraining.



View the input-error cross-correlation plot to obtain additional verification of network performance. In the **Plots** section, click **Input-Error Correlation**. The input-error cross-correlation plot illustrates how the errors are correlated with the input sequence $x(t)$. For a perfect prediction model, all of the correlations should be zero. If the input is correlated with the error, then it should be possible to improve the prediction, perhaps by increasing the number of delays in the tapped delay lines. In this case, most of the correlations fall within the confidence bounds around zero.



In the **Plots** section, click **Response**. This displays the outputs, responses (targets), and errors versus time. It also indicates which time points were selected for training, testing, and validation.



If you are unhappy with the network performance, you can do one of the following:

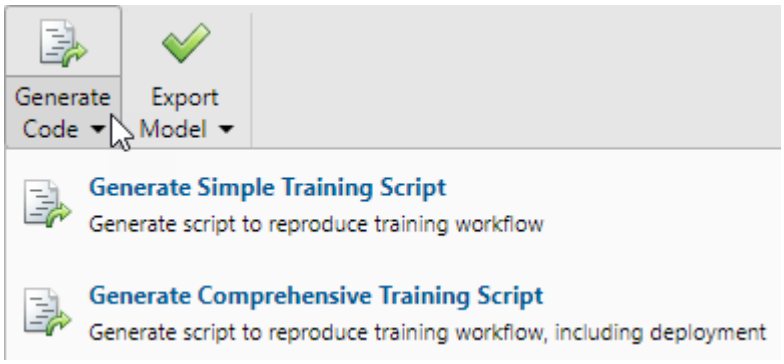
- Train the network again.
- Increase the number of hidden neurons.
- Use a larger training data set.

If performance on the training set is good but the test set performance is poor, this could indicate the model is overfitting. Decreasing the layer size, and therefore decreasing the number of neurons, can reduce the overfitting.

You can also evaluate the network performance on an additional test set. To load additional test data to evaluate the network with, in the **Test** section, click **Test**. The **Model Summary** displays the additional test data results. You can also generate plots to analyze the additional test data results.

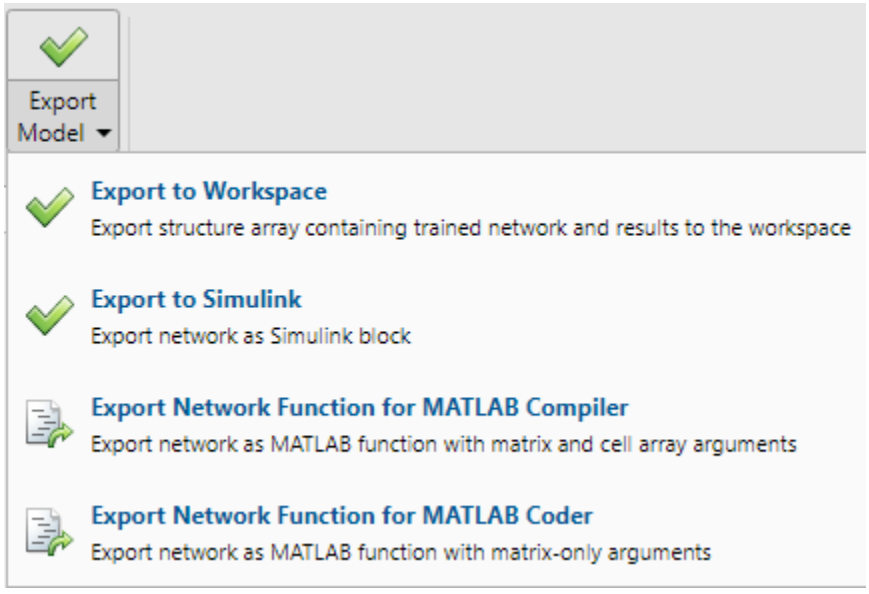
Generate Code

Select **Generate Code** > **Generate Simple Training Script** to create MATLAB code to reproduce the previous steps from the command line. Creating MATLAB code can be helpful if you want to learn how to use the command-line functionality of the toolbox to customize the training process. In “Fit Time Series Data Using Command-Line Functions”, you will investigate the generated scripts in more detail.



Export Network

You can export your trained network to the workspace or Simulink®. You can also deploy the network with MATLAB Compiler™ tools and other MATLAB code generation tools. To export your trained network and results, select **Export Model** > **Export to Workspace**.



See Also

Neural Net Fitting | **Neural Net Time Series** | **Neural Net Pattern Recognition** | **Neural Net Clustering** | `train` | `preparets` | `narxnet` | `closeloop` | `perform` | `removedelay`

Related Examples

- “Shallow Neural Network Time-Series Prediction and Modeling”
- “Fit Data with a Shallow Neural Network”
- “Classify Patterns with a Shallow Neural Network”
- “Cluster Data with a Self-Organizing Map”

Body Fat Estimation

This example illustrates how a function fitting neural network can estimate body fat percentage based on anatomical measurements.

The Problem: Estimate Body Fat Percentage

In this example we attempt to build a neural network that can estimate the body fat percentage of a person described by thirteen physical attributes:

- Age (years)
- Weight (lbs)
- Height (inches)
- Neck circumference (cm)
- Chest circumference (cm)
- Abdomen circumference (cm)
- Hip circumference (cm)
- Thigh circumference (cm)
- Knee circumference (cm)
- Ankle circumference (cm)
- Biceps (extended) circumference (cm)
- Forearm circumference (cm)
- Wrist circumference (cm)

This is an example of a fitting problem, where inputs are matched up to associated target outputs, and we would like to create a neural network which not only estimates the known targets given known inputs, but can also generalize to accurately estimate outputs for inputs that were not used to design the solution.

Why Neural Networks?

Neural networks are very good at function fit problems. A neural network with enough elements (called neurons) can fit any data with arbitrary accuracy. They are particularly well suited for addressing nonlinear problems. Given the nonlinear nature of real world phenomena, like body fat accretion, neural networks are a good candidate for solving the problem.

The thirteen physical attributes will act as inputs to a neural network, and the body fat percentage will be the target.

The network will be designed by using the anatomical quantities of bodies whose body fat percentage is already known to train it to produce the target valuations.

Preparing the Data

Data for function fitting problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have thirteen elements representing a body with known body fat percentage.

Each corresponding column of the target matrix will have one element, representing the body fat percentage.

Here such a dataset is loaded.

```
[X,T] = bodyfat_dataset;
```

We can view the sizes of inputs X and targets T.

Note that both X and T have 252 columns. These represent 252 physiques (inputs) and associated body fat percentages (targets).

The input matrix X has thirteen rows, for the thirteen attributes. The target matrix T has only one row, as for each example we only have one desired output, the body fat percentage.

```
size(X)
```

```
ans = 1×2

    13   252
```

```
size(T)
```

```
ans = 1×2

     1   252
```

Fitting a Function with a Neural Network

The next step is to create a neural network that will learn to estimate body fat percentages.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

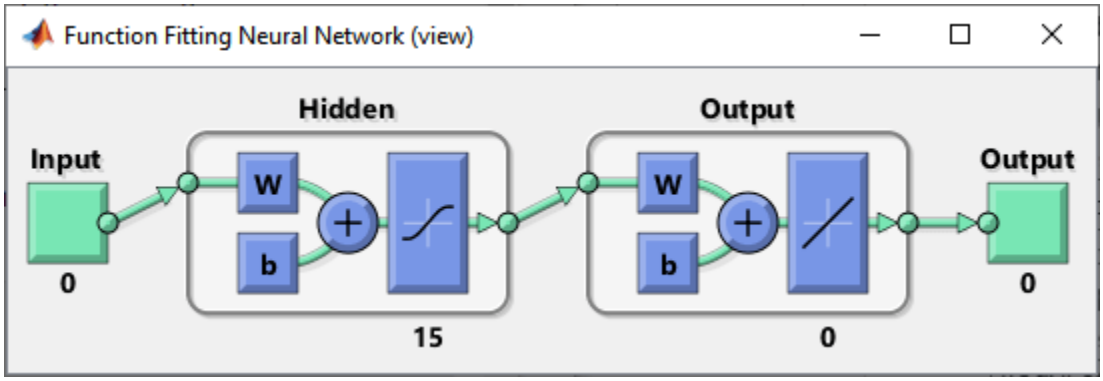
```
setdemorandstream(491218382)
```

Two-layer (i.e. one-hidden-layer) feed forward neural networks can fit any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 15 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

```
net = fitnet(15);
view(net)
```

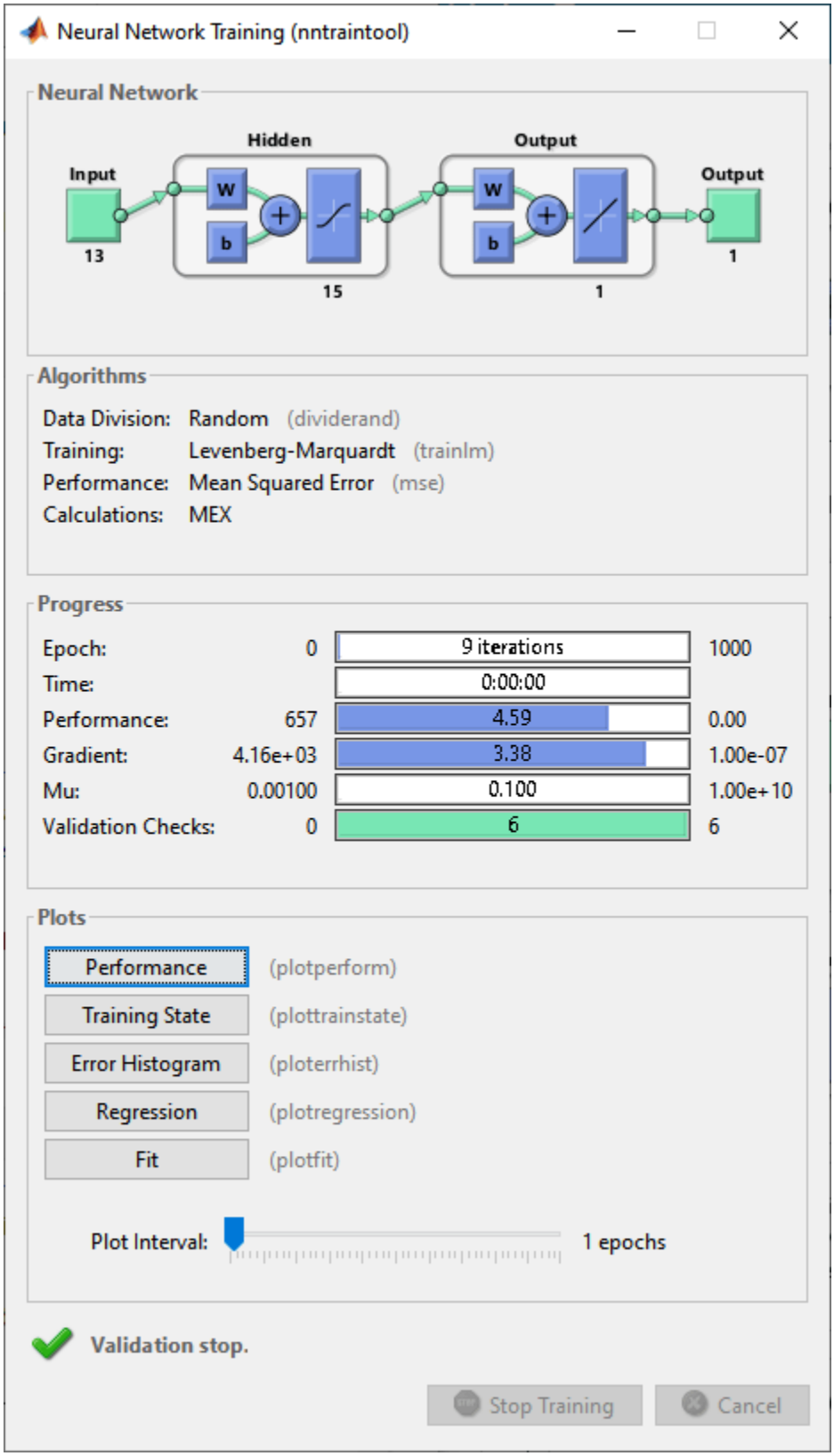


Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,X,T);
```

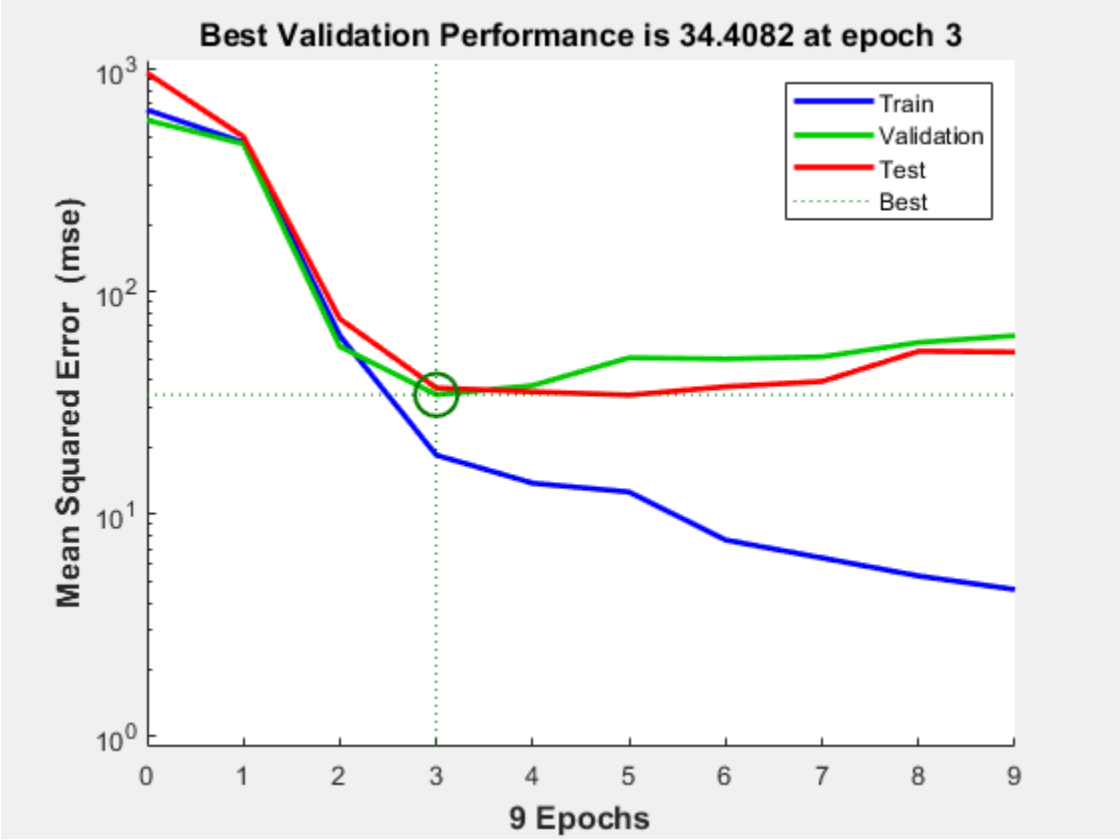

Neural Network Training (nntraintool)
Neural Network
Input
13
Hidden
W
b
15
Output
W
b
1
Output
1
Algorithms
Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX
Progress
Epoch: 0 9 iterations 1000
Time: 0:00:00
Performance: 657 4.59 0.00
Gradient: 4.16e+03 3.38 1.00e-07
Mu: 0.00100 0.100 1.00e+10
Validation Checks: 0 6 6
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Fit (plotfit)
Plot Interval: 1 epochs
Validation stop.
Stop Training
Cancel

To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and shown in log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation, and test sets. The final network is the network that performed best on the validation set.

```
plotperform(tr)
```



Testing the Neural Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

```
testX = X(:,tr.testInd);
testT = T(:,tr.testInd);

testY = net(testX);

perf = mse(net,testT,testY)
```

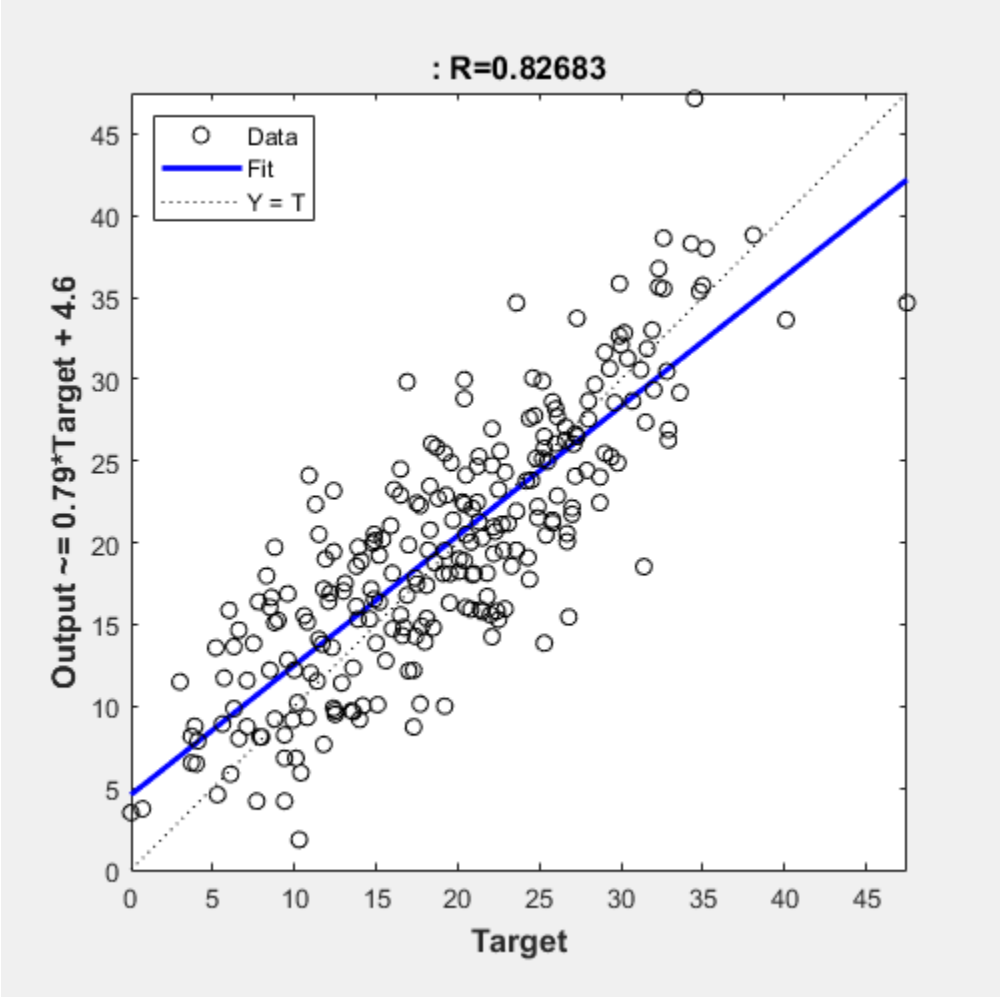
```
perf = 36.9404
```

Another measure of how well the neural network has fit the data is the regression plot. Here the regression is plotted across all samples.

The regression plot shows the actual network outputs plotted in terms of the associated target values. If the network has learned to fit the data well, the linear fit to this output-target relationship should closely intersect the bottom-left and top-right corners of the plot.

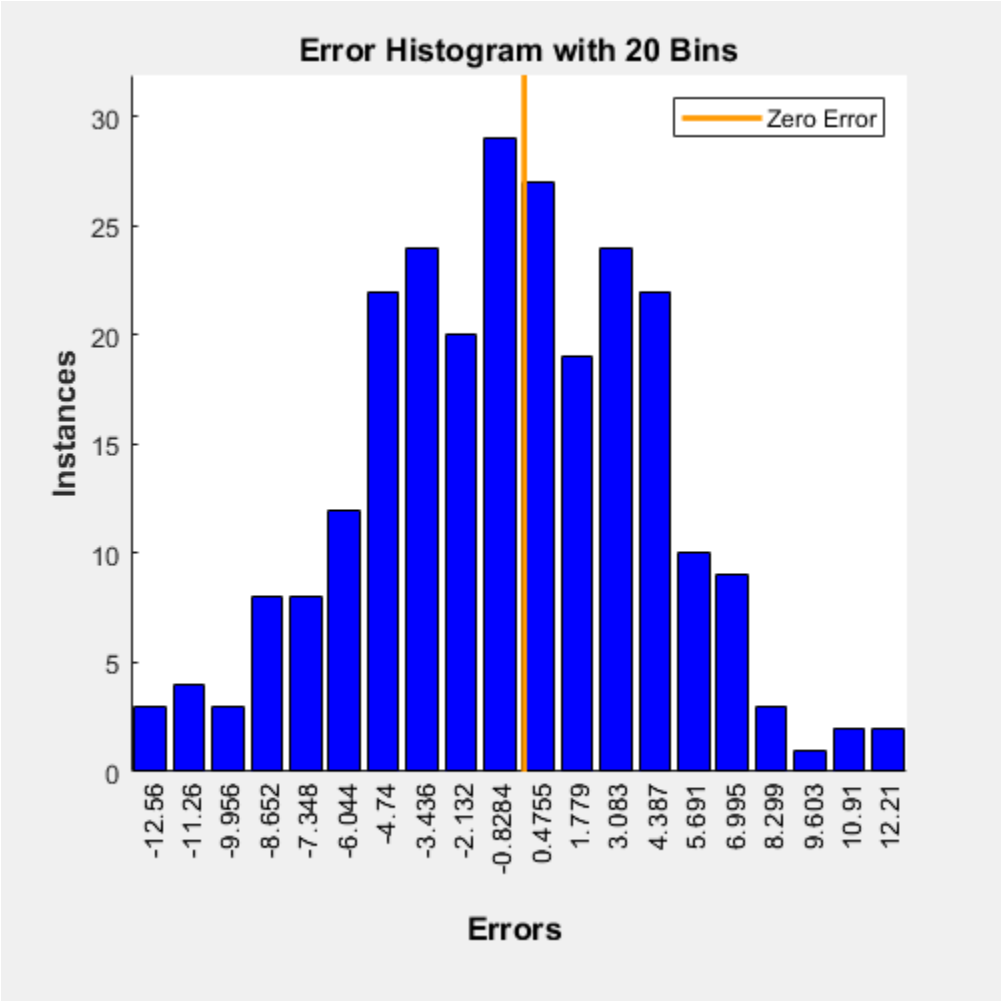
If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
Y = net(X);

plotregression(T,Y)
```



Another third measure of how well the neural network has fit data is the error histogram. This shows how the error sizes are distributed. Typically most errors are near zero, with very few errors far from that.

```
e = T - Y;

ploterrhist(e)
```



This example illustrated how to design a neural network that estimates the body fat percentage from physical characteristics.

Explore other examples and the documentation for more insight into neural networks and their applications.

Crab Classification

This example illustrates using a neural network as a classifier to identify the sex of crabs from physical dimensions of the crab.

The Problem: Classification of Crabs

In this example we attempt to build a classifier that can identify the sex of a crab from its physical measurements. Six physical characteristics of a crab are considered: species, frontallip, rearwidth, length, width and depth. The problem on hand is to identify the sex of a crab given the observed values for each of these 6 physical characteristics.

Why Neural Networks?

Neural networks have proven themselves as proficient classifiers and are particularly well suited for addressing non-linear problems. Given the non-linear nature of real world phenomena, like crab classification, neural networks are certainly a good candidate for solving the problem.

The six physical characteristics will act as inputs to a neural network and the sex of the crab will be the target. Given an input, which constitutes the six observed values for the physical characteristics of a crab, the neural network is expected to identify if the crab is male or female.

This is achieved by presenting previously recorded inputs to a neural network and then tuning it to produce the desired target outputs. This process is called neural network training.

Preparing the Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have six elements representing a crab's species, frontallip, rearwidth, length, width, and depth.

Each corresponding column of the target matrix will have two elements. Female crabs are represented with a one in the first element, male crabs with a one in the second element. (All other elements are zero).

Here the dataset is loaded.

```
[x,t] = crab_dataset;
size(x)
```

```
ans = 1×2

     6   200
```

```
size(t)
```

```
ans = 1×2

     2   200
```

Building the Neural Network Classifier

The next step is to create a neural network that will learn to identify the sex of the crabs.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

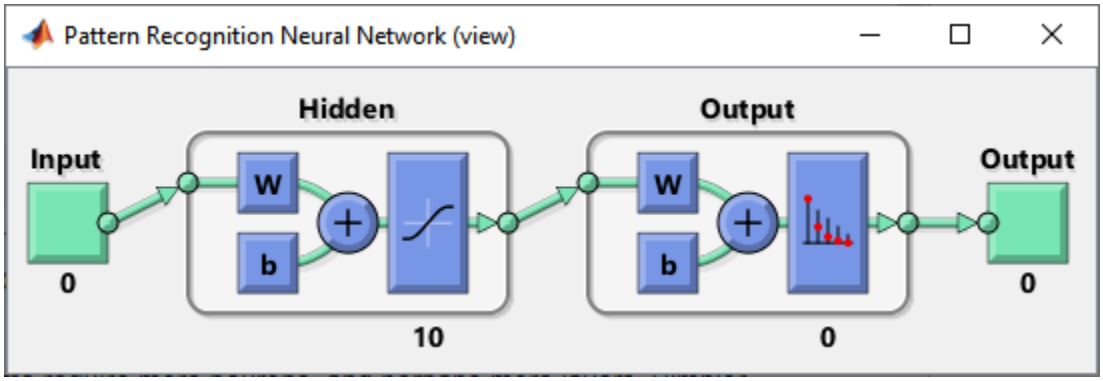
```
setdemorandstream(491218382)
```

Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

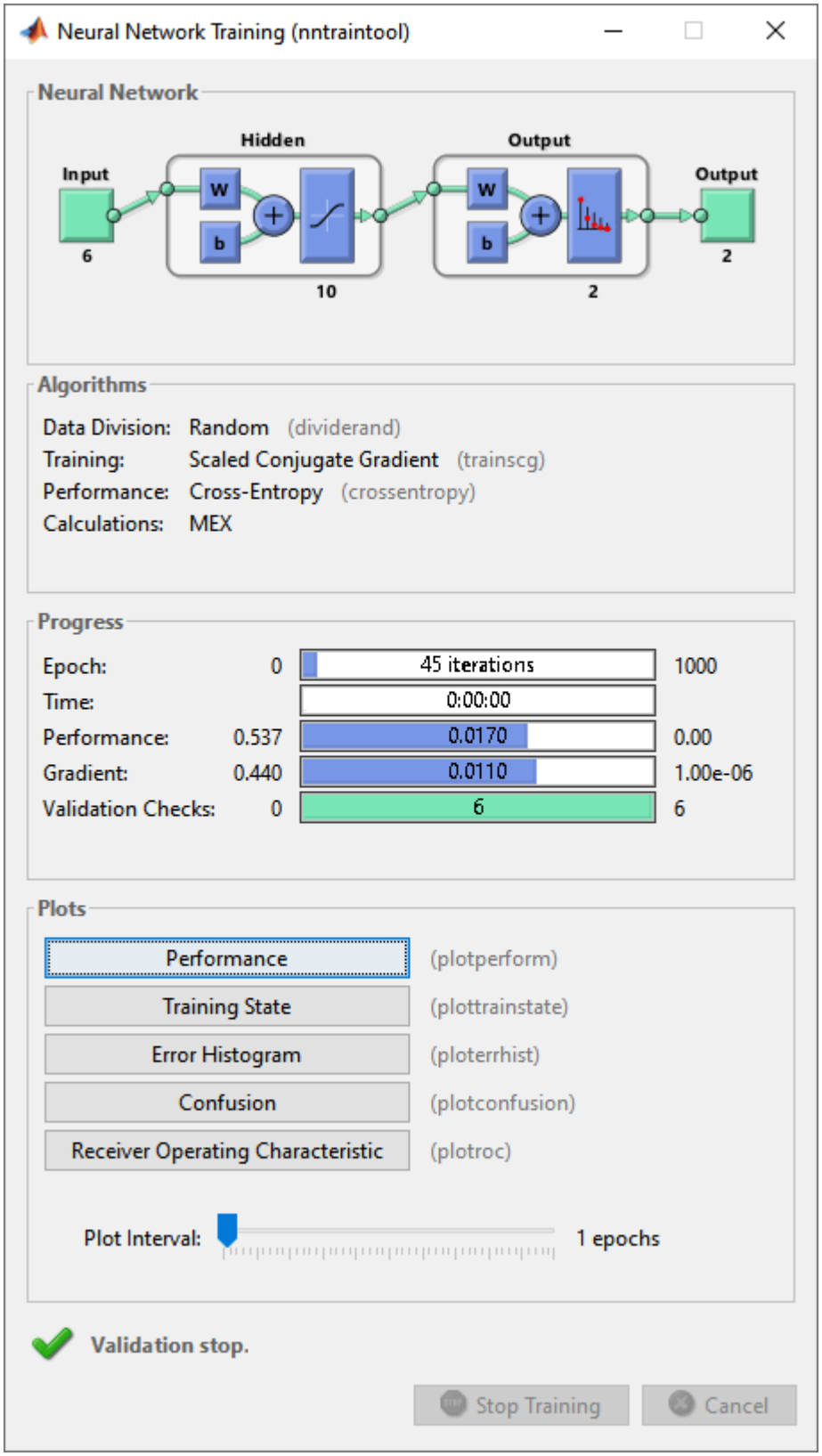
The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

```
net = patternnet(10);
view(net)
```



Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

```
[net,tr] = train(net,x,t);
```

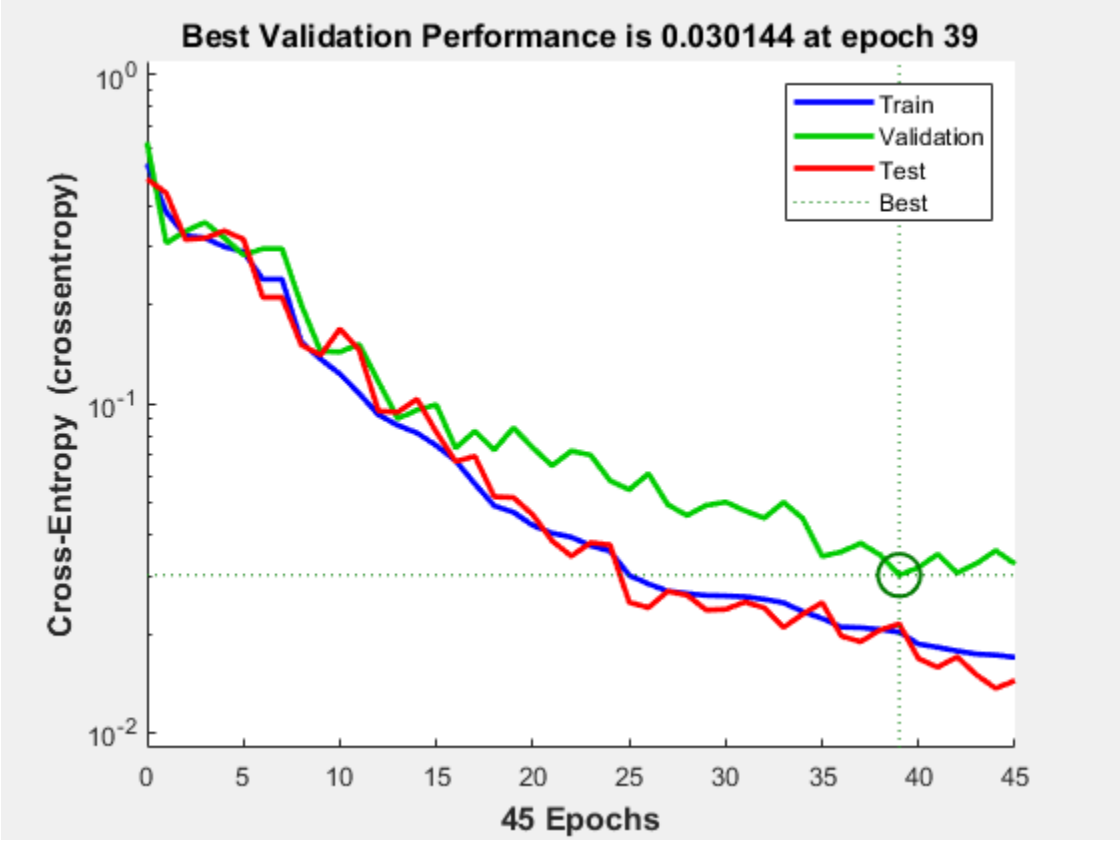

Neural Network Training (nntraintool)
Neural Network
Input
6
Hidden
W
b
10
Output
W
b
2
Output
2
Algorithms
Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Calculations: MEX
Progress
Epoch: 0 45 iterations 1000
Time: 0:00:00
Performance: 0.537 0.0170 0.00
Gradient: 0.440 0.0110 1.00e-06
Validation Checks: 0 6 6
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Confusion (plotconfusion)
Receiver Operating Characteristic (plotroc)
Plot Interval: 1 epochs
Validation stop.
Stop Training
Cancel

To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and is shown in a log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets.

```
plotperform(tr)
```



Testing the Classifier

The trained neural network can now be tested with the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX = x(:,tr.testInd);
testT = t(:,tr.testInd);

testY = net(testX);
testIndices = vec2ind(testY)
```

```
testIndices = 1×30

     2     2     1     1     2     2     1     2     2     2     1     2     2     2     2     2
```

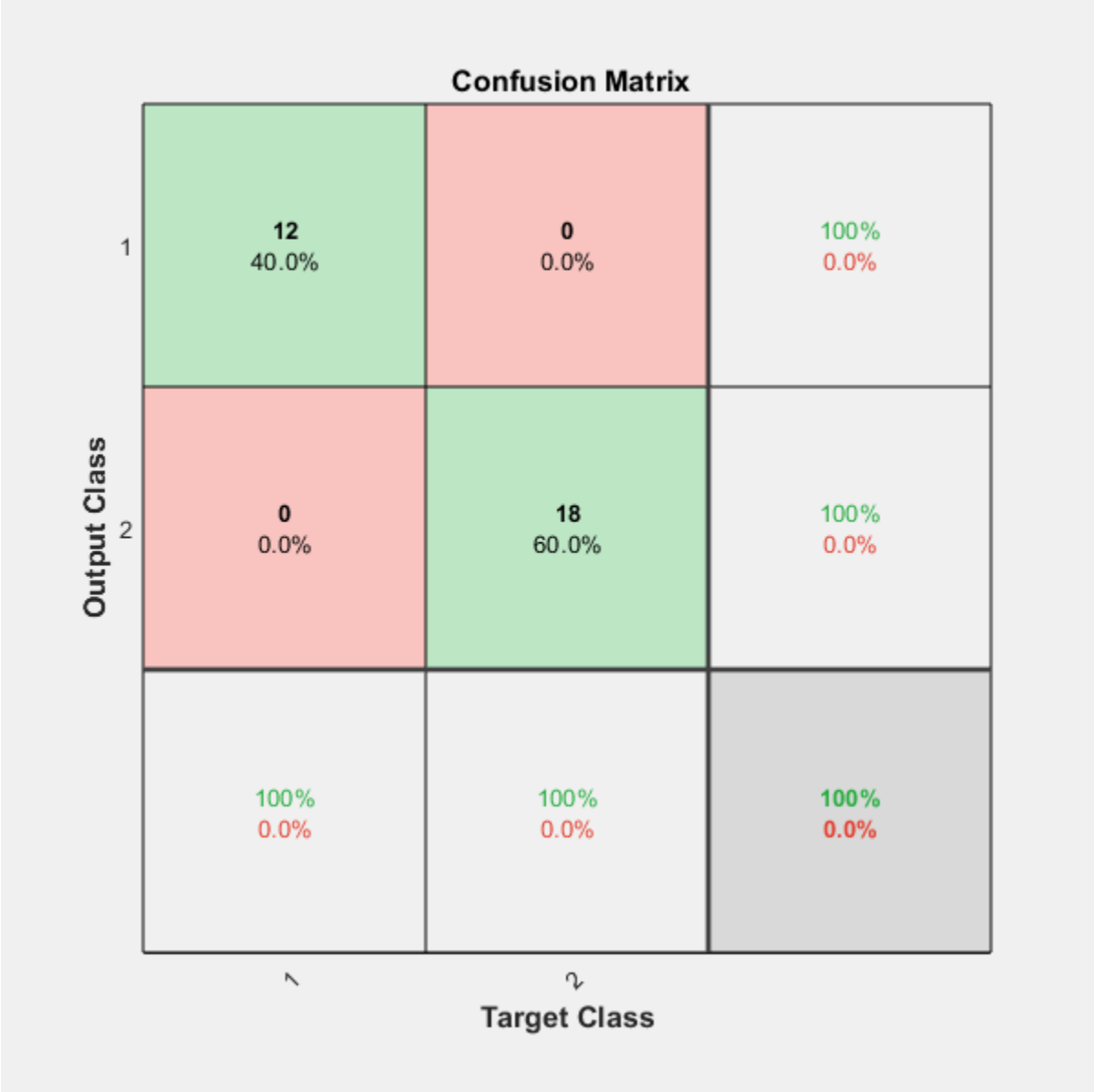
One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
plotconfusion(testT,testY)
```



Here are the overall percentages of correct and incorrect classification.

```
[c,cm] = confusion(testT,testY)
```

```
c = 0
```

```
cm = 2×2

    12     0
     0    18
```

```
fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c));
```

```
Percentage Correct Classification   : 100.000000%
```

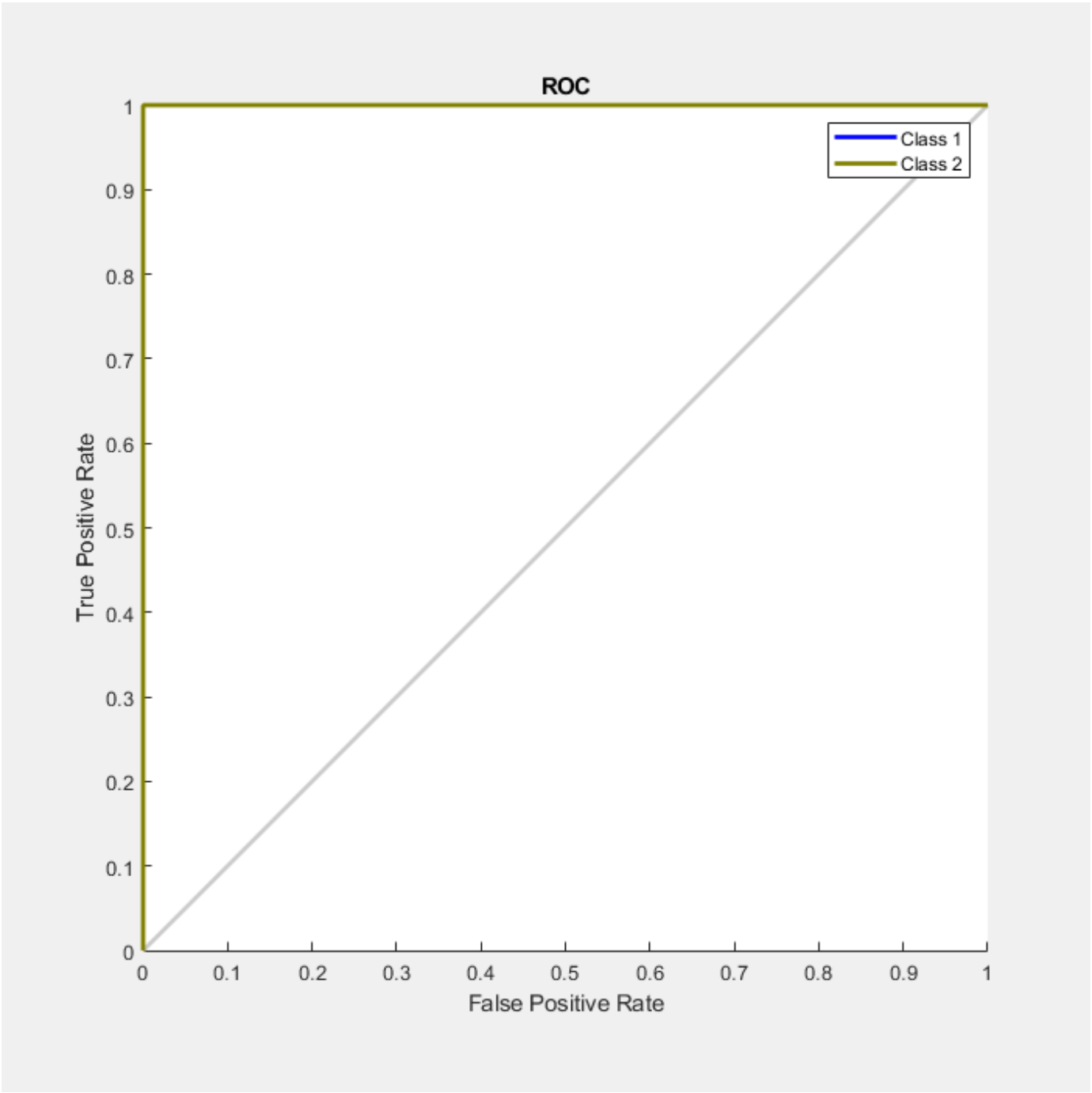
```
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

```
Percentage Incorrect Classification : 0.000000%
```

Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

```
plotroc(testT,testY)
```



This example illustrated using a neural network to classify crabs.

Explore other examples and the documentation for more insight into neural networks and their applications.

Wine Classification

This example illustrates how a pattern recognition neural network can classify wines by winery based on its chemical characteristics.

The Problem: Classify Wines

In this example we attempt to build a neural network that can classify wines from three wineries by thirteen attributes:

- Alcohol
- Malic acid
- Ash
- Alkalinity of ash
- Magnesium
- Total phenols
- Flavonoids
- Nonflavonoid phenols
- Proanthocyanidins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

This is an example of a pattern recognition problem, where inputs are associated with different classes, and we would like to create a neural network that not only classifies the known wines properly, but can also generalize to accurately classify wines that were not used to design the solution.

Why Neural Networks?

Neural networks are very good at pattern recognition problems. A neural network with enough elements (called neurons) can classify any data with arbitrary accuracy. They are particularly well suited for complex decision boundary problems over many variables. Therefore, neural networks are a good candidate for solving the wine classification problem.

The thirteen neighborhood attributes will act as inputs to a neural network, and the respective target for each will be a 3-element class vector with a 1 in the position of the associated winery, #1, #2 or #3.

The network will be designed by using the attributes of neighborhoods to train the network to produce the correct target classes.

Prepare Data

Data for classification problems are set up for a neural network by organizing the data into two matrices, the input matrix X and the target matrix T.

Each ith column of the input matrix will have thirteen elements representing a wine whose winery is already known.

Each corresponding column of the target matrix will have three elements, consisting of two zeros and a 1 in the location of the associated winery.

Here such a dataset is loaded.

```
[x,t] = wine_dataset;
```

We can view the sizes of inputs X and targets T.

Note that both X and T have 178 columns. These represent 178 wine sample attributes (inputs) and associated winery class vectors (targets).

Input matrix X has thirteen rows, for the thirteen attributes. Target matrix T has three rows, as for each example we have three possible wineries.

```
size(x)
```

```
ans = 1×2

    13   178
```

```
size(t)
```

```
ans = 1×2

     3   178
```

Pattern Recognition with a Neural Network

The next step is to create a neural network that will learn to classify the wines.

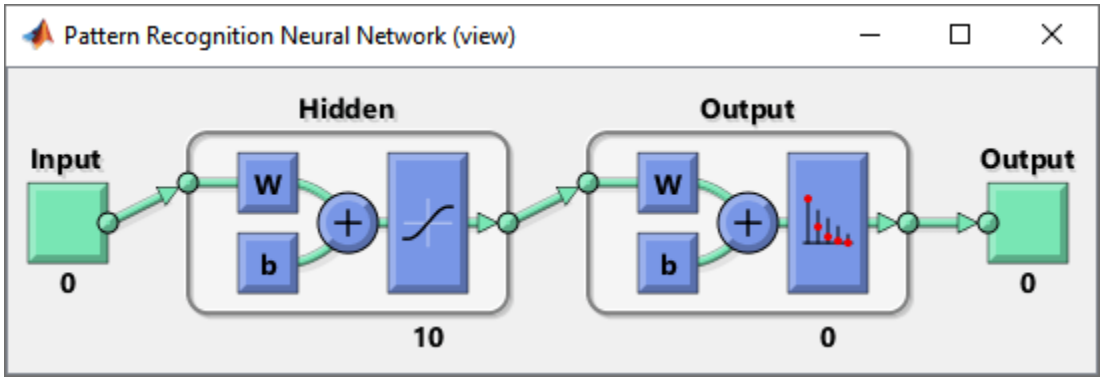
Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run.

Two-layer (i.e. one-hidden-layer) feed forward neural networks can learn any input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

```
net = patternnet(10);
view(net)
```

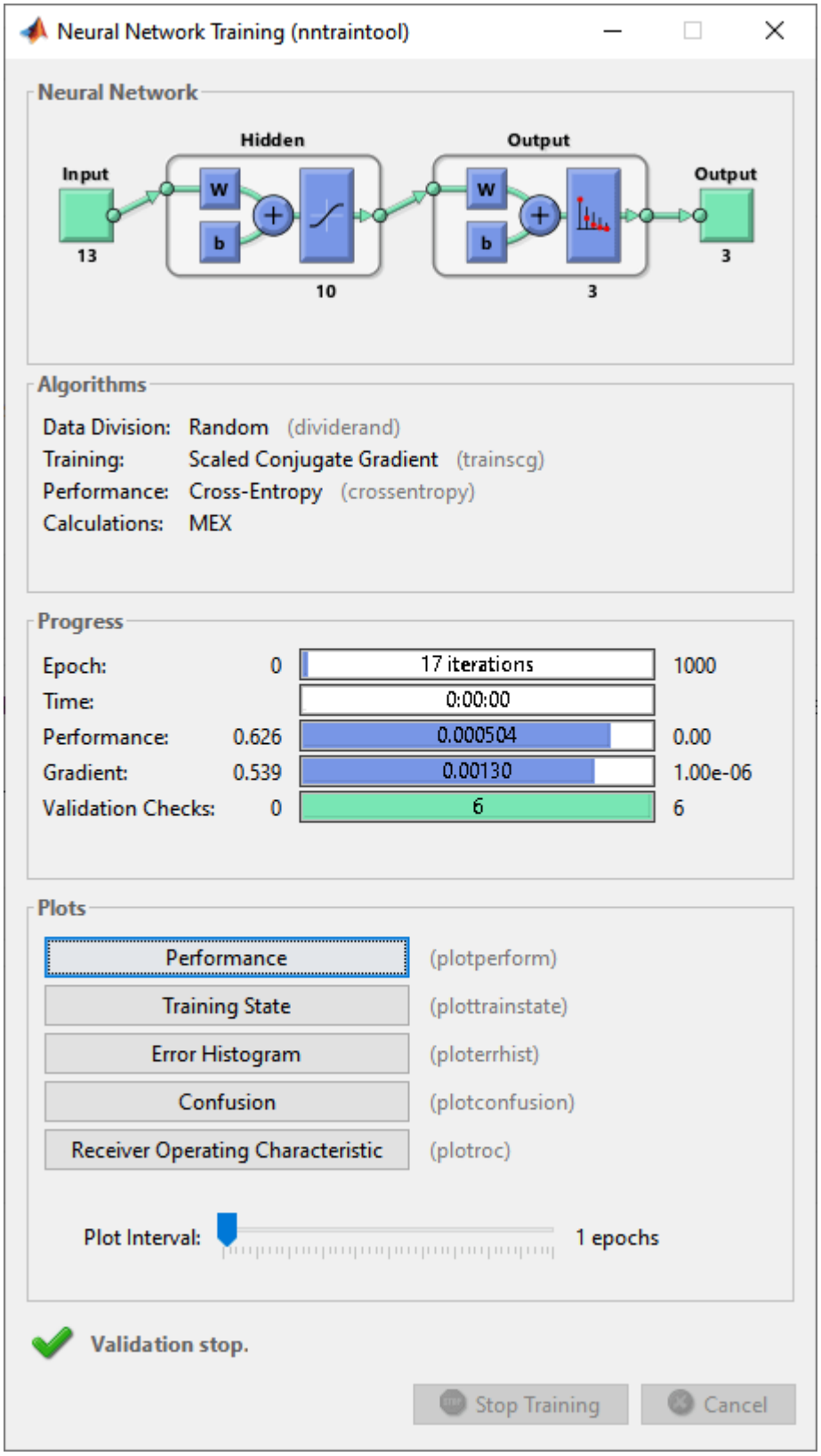


Now the network is ready to be trained. The samples are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,x,t);
```

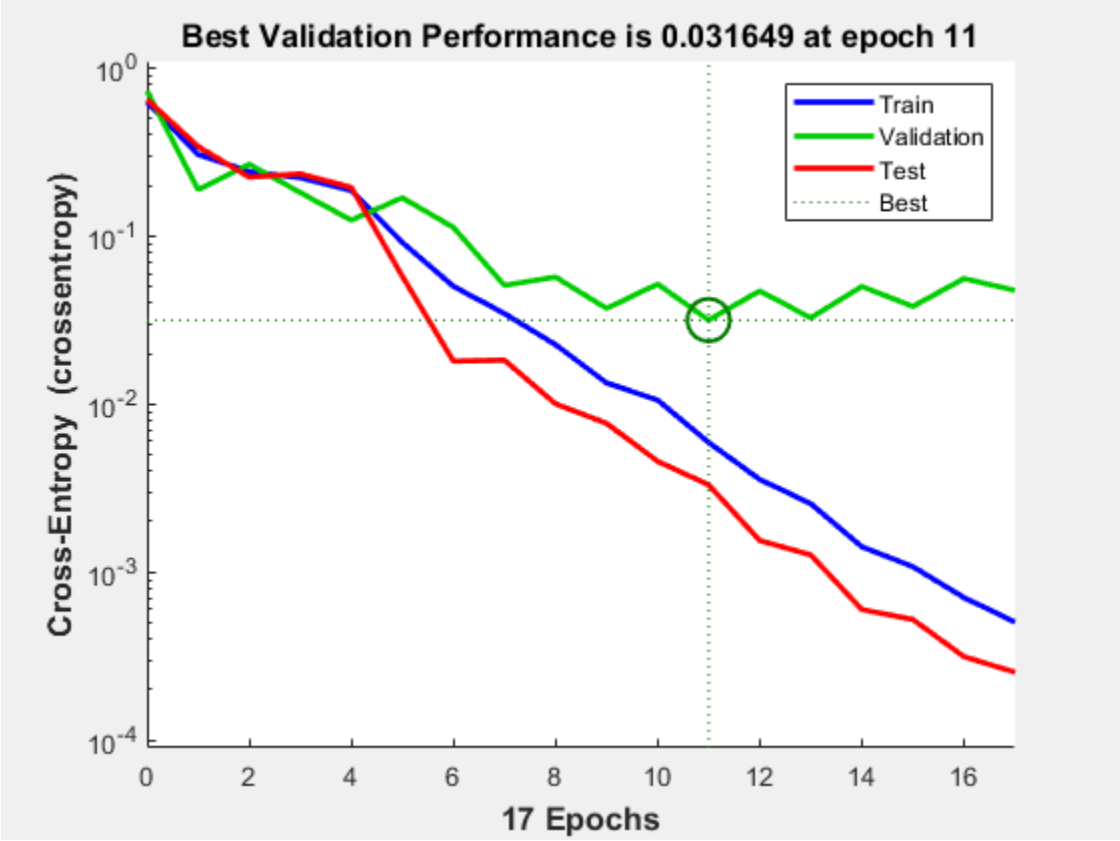

Neural Network Training (nntraintool)
Neural Network
Input
13
Hidden
W
b
10
Output
W
b
3
Output
3
Algorithms
Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Calculations: MEX
Progress
Epoch: 0 17 iterations 1000
Time: 0:00:00
Performance: 0.626 0.000504 0.00
Gradient: 0.539 0.00130 1.00e-06
Validation Checks: 0 6 6
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Confusion (plotconfusion)
Receiver Operating Characteristic (plotroc)
Plot Interval: 1 epochs
Validation stop.
Stop Training
Cancel

To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and is shown in a log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets.

```
plotperform(tr)
```



Test the Network

The mean squared error of the trained neural network can now be measured with respect to the testing samples. This will give us a sense of how well the network will do when applied to data from the real world.

The network outputs will be in the range 0 to 1, so we can use **vec2ind** function to get the class indices as the position of the highest element in each output vector.

```
testX = x(:,tr.testInd);
testT = t(:,tr.testInd);

testY = net(testX);
testIndices = vec2ind(testY)
```

```
testIndices = 1×27
```

```
     1     1     1     1     1     1     1     2     2     2     2     2     2     2     2     2
```

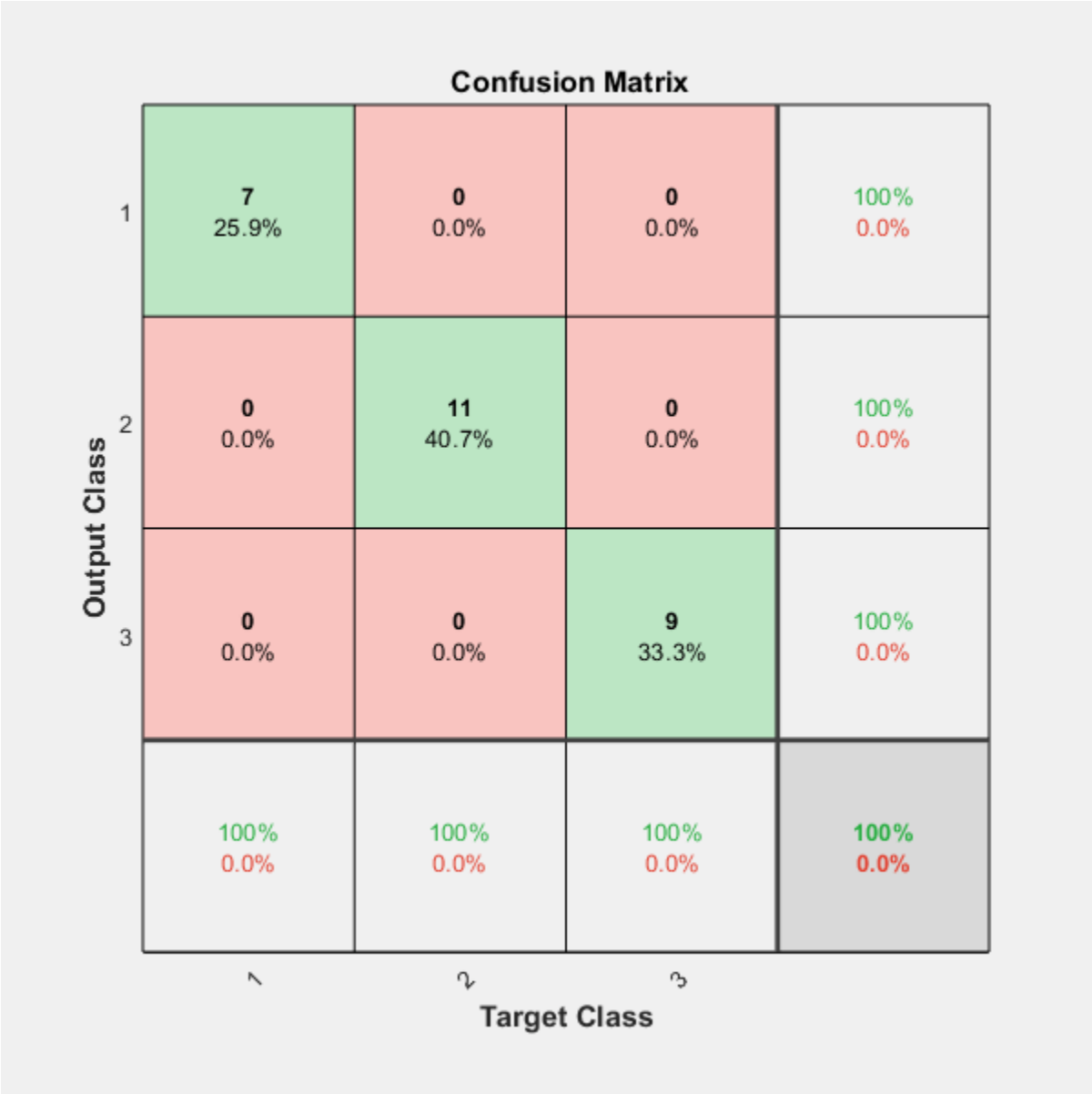
Another measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices diagonal. Incorrect classifications form the red squares.

If the network has learned to classify properly, the percentages in the red squares should be very small, indicating few misclassifications.

If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
plotconfusion(testT,testY)
```



Here are the overall percentages of correct and incorrect classification.

```
[c,cm] = confusion(testT,testY)
```

```
c = 0
```

```
cm = 3×3

     7     0     0
     0    11     0
     0     0     9
```

```
fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c));
```

```
Percentage Correct Classification   : 100.000000%
```

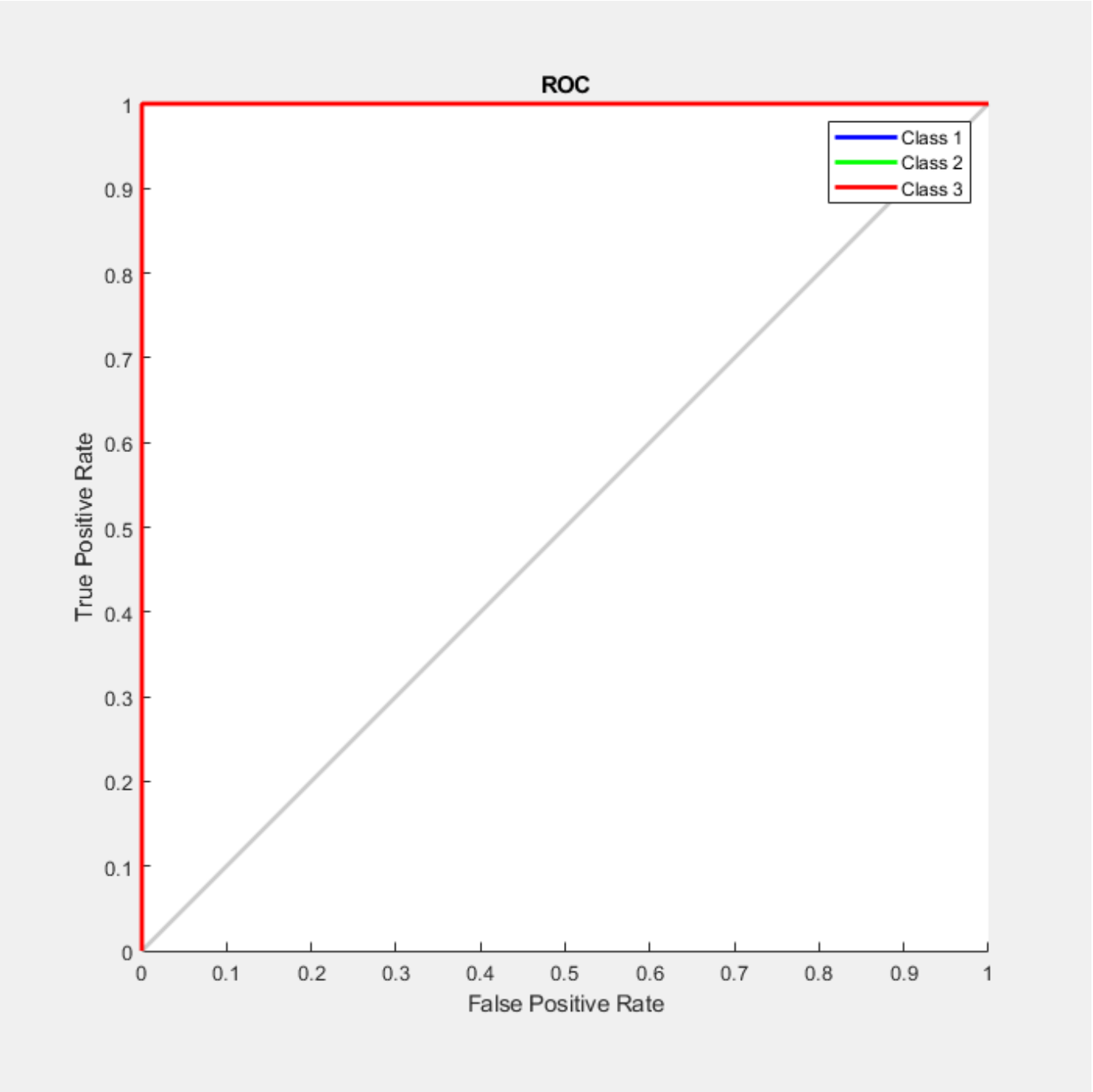
```
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

```
Percentage Incorrect Classification : 0.000000%
```

A third measure of how well the neural network has fit data is the receiver operating characteristic plot. This shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers will have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

```
plotroc(testT,testY)
```



This example illustrated how to design a neural network that classifies wines into three wineries from each wine's characteristics.

Explore other examples and the documentation for more insight into neural networks and their applications.

Cancer Detection

This example shows how to train a neural network to detect cancer using mass spectrometry data on protein profiles.

Introduction

Serum proteomic pattern diagnostics can be used to differentiate samples from patients with and without disease. Profile patterns are generated using surface-enhanced laser desorption and ionization (SELDI) protein mass spectrometry. This technology has the potential to improve clinical diagnostics tests for cancer pathologies.

The Problem: Cancer Detection

The goal is to build a classifier that can distinguish between cancer and control patients from the mass spectrometry data.

The methodology followed in this example is to select a reduced set of measurements or "features" that can be used to distinguish between cancer and control patients using a classifier. These features are ion intensity levels at specific mass/charge values.

Formatting the Data

The data used in this example, provided in the file `ovarian_dataset.mat`, is from the FDA-NCI Clinical Proteomics Program Databank. For a detailed description of this data set, see [1] and [2].

Create the data file `OvarianCancerQAQCdataset.mat` by following the steps in “Batch Processing of Spectra Using Sequential and Parallel Computing” (Bioinformatics Toolbox). The new file contains the variables `Y`, `MZ`, and `grp`.

Each column in `Y` represents measurements taken from a patient. There are `216` columns in `Y` representing `216` patients, out of which `121` are ovarian cancer patients and `95` are normal patients.

Each row in `Y` represents the ion intensity level at a specific mass-charge value indicated in `MZ`. There are `15000` mass-charge values in `MZ` and each row in `Y` represents the ion-intensity levels of the patients at that particular mass-charge value.

The variable `grp` holds the index information as to which of these samples represent cancer patients and which ones represent normal patients.

Ranking Key Features

This task is a typical classification problem where the number of features is much larger than the number of observations but single feature achieves a correct classification. Therefore, the goal is to find a classifier which appropriately learns how to weight multiple features and at the same time produces a generalized mapping which is not over-fitted.

A simple approach for finding significant features is to assume that each M/Z value is independent and compute a two-way t-test. `rankfeatures` returns an index to the most significant M/Z values, for instance 100 indices ranked by the absolute value of the test statistic.

Load the `OvarianCancerQAQCdataset.mat` and rank features using `rankfeatures` (Bioinformatics Toolbox) to choose 100 highest ranked measurements as inputs `x`.

```
ind = rankfeatures(Y,grp,'Criterion','ttest','NumberOfIndices',100);
x = Y(ind,:);
```

Define the targets `t` for the two classes as follows:

```
t = double(strcmp('Cancer',grp));
t = [t; 1-t];
```

The preprocessing steps from the script and example listed above are intended to demonstrate a representative set of possible preprocessing and feature selection procedures. Using different steps or parameters can lead to different and possibly better results.

```
[x,t] = ovarian_dataset;
whos x t
```

```
  Name        Size              Bytes  Class     Attributes

  t           2x216              3456  double
  x         100x216            172800  double
```

Each column in `x` represents one of 216 different patients.

Each row in `x` represents the ion intensity level at one of the 100 specific mass-charge values for each patient.

The variable `t` has two rows with 216 values each of which are either [1;0], indicating a cancer patient, or [0;1] for a normal patient.

Classification Using a Feed Forward Neural Network

Now that you have identified some significant features, you can use this information to classify the cancer and normal samples.

Since the neural network is initialized with random initial weights, the results after training the network vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. However, setting the random seed is not necessary for your own applications.

```
setdemorandstream(672880951)
```

A 1-hidden layer feed forward neural network with 5 hidden layer neurons is created and trained. The input and target samples are automatically divided into training, validation, and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides an independent measure of the network accuracy.

The input and output have sizes of 0 because the network has not yet been configured to match the input and target data. This configuration happens when you train the network.

```
net = patternnet(5);
view(net)
```

Now the network is ready to be trained. The samples are automatically divided into training, validation, and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides an independent measure of network accuracy.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training are highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

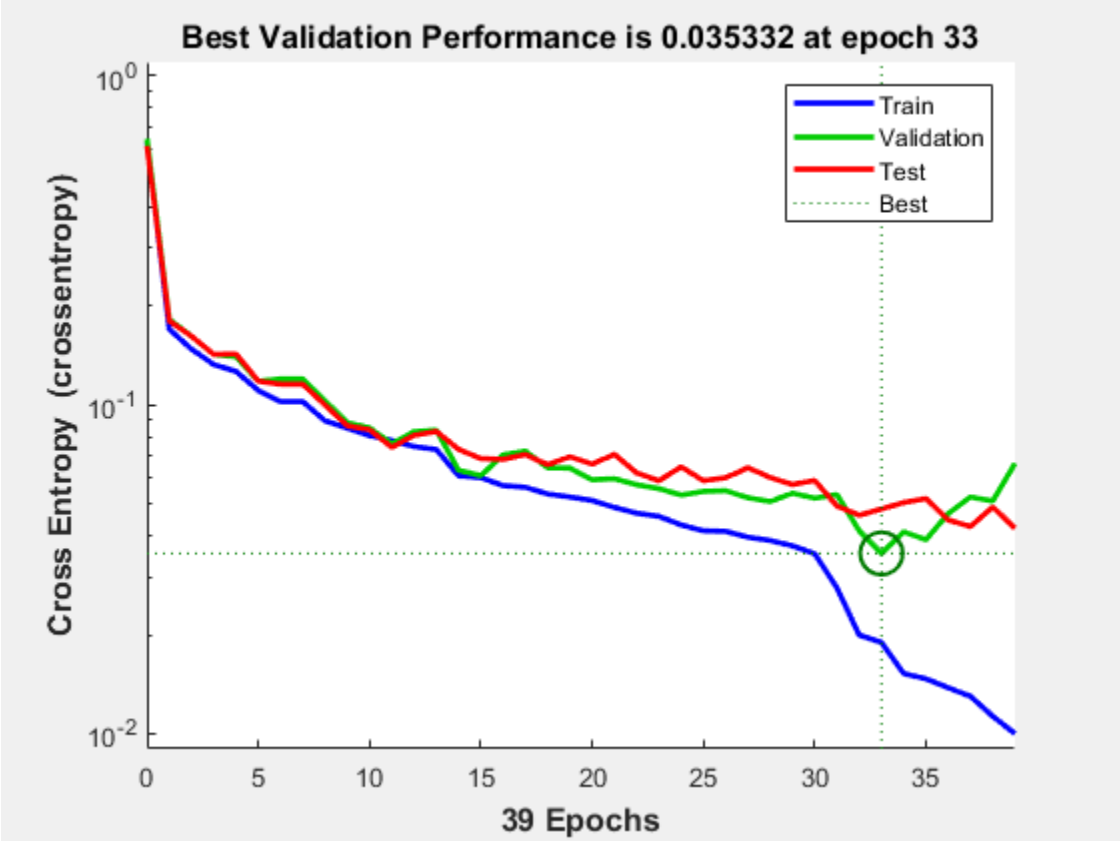
```
[net,tr] = train(net,x,t);
```

To see how the network's performance improved during training, either click the "Performance" button in the training tool, or use the `plotperform` function.

Performance is measured in terms of mean squared error, and shown on a logarithmic scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation, and test sets.

```
plotperform(tr)
```



The trained neural network can now be tested with the testing samples we partitioned from the main dataset. The testing data was not used in training in any way and hence provides an "out-of-sample" dataset to test the network on. This gives an estimate of how well the network will perform when tested with data from the real world.

The network outputs are in the range 0-1. Threshold the outputs to obtain 1's and 0's indicating cancer or normal patients, respectively.

```
testX = x(:,tr.testInd);
testT = t(:,tr.testInd);

testY = net(testX);
testClasses = testY > 0.5
```

```
testClasses = 2x32 logical array

   0   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   0   0   1   0
   1   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   1   1   0   1
```

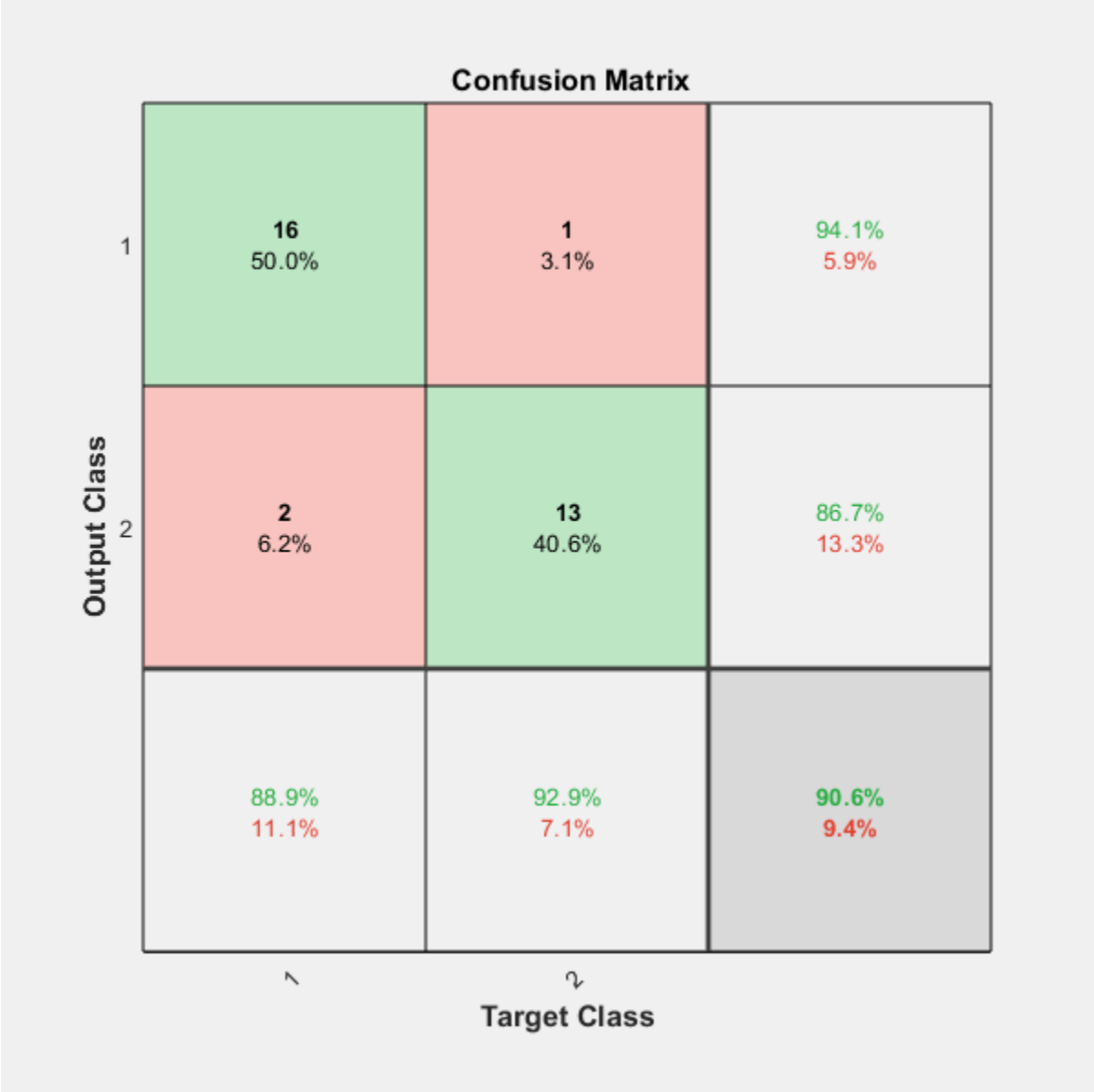
One measure of how well the neural network has fit the data is the confusion plot.

The confusion matrix shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrix diagonal. The red squares represent incorrect classifications.

If the network is accurate, then the percentages in the red squares are small, indicating few misclassifications.

If the network is not accurate, then you can try training for a longer time, or training a network with more hidden neurons.

```
plotconfusion(testT,testY)
```



Here are the overall percentages of correct and incorrect classification.

```
[c,cm] = confusion(testT,testY);
```

```
fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c));
```

```
Percentage Correct Classification   : 90.625000%
```

```
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

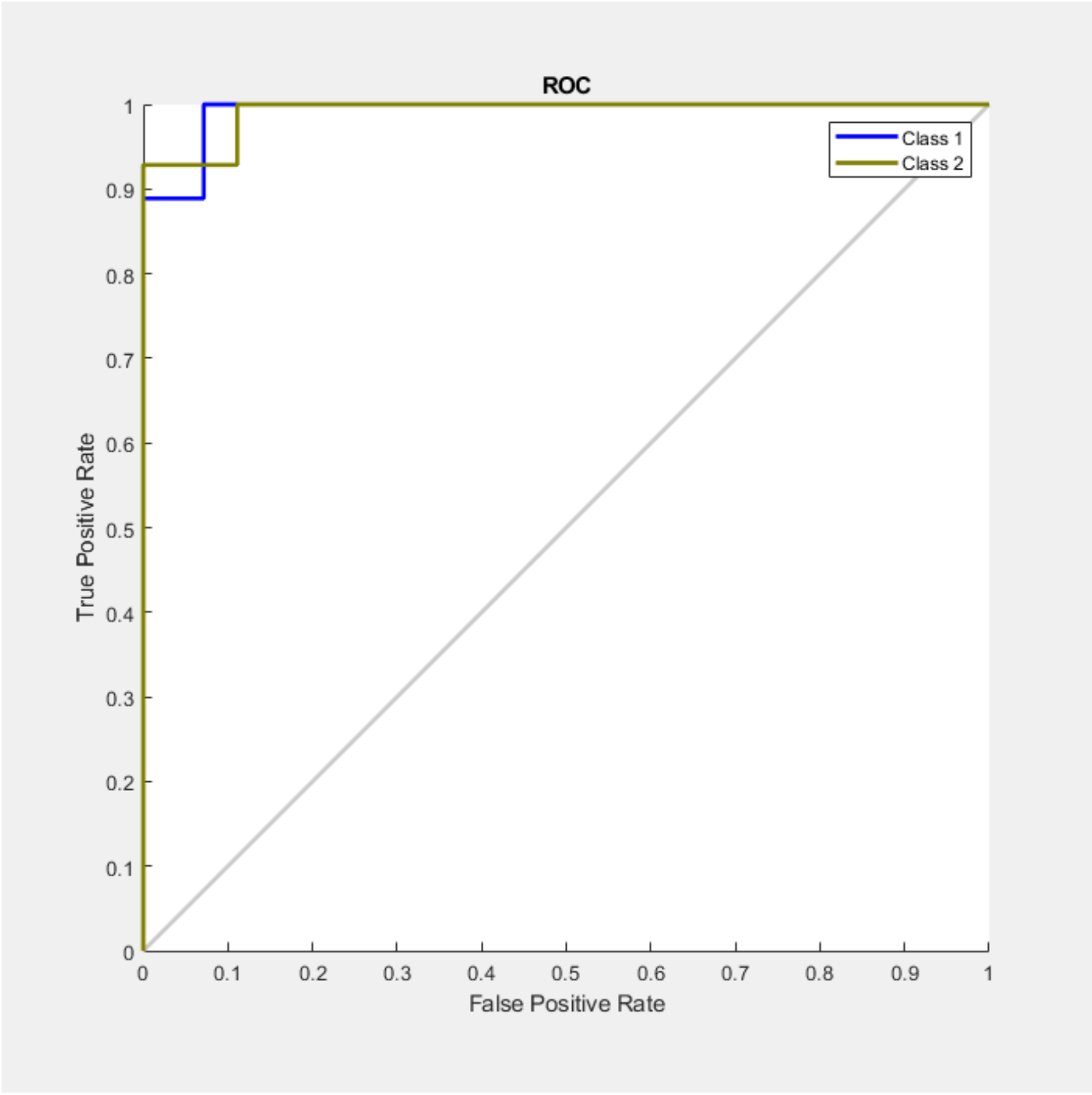
```
Percentage Incorrect Classification : 9.375000%
```

Another measure of how well the neural network has fit data is the receiver operating characteristic plot. This plot shows how the false positive and true positive rates relate as the thresholding of outputs is varied from 0 to 1.

The farther left and up the line is, the fewer false positives need to be accepted in order to get a high true positive rate. The best classifiers have a line going from the bottom left corner, to the top left corner, to the top right corner, or close to that.

Class 1 indicates cancer patients and class 2 indicates normal patients.

```
plotroc(testT,testY)
```



This example demonstrates how neural networks can be used as classifiers for cancer detection. To improve classifier performance, you can also try using techniques like principal component analysis for reducing the dimensionality of the data used for neural network training.

References

[1] T.P. Conrads, et al., "High-resolution serum proteomic features for ovarian detection", Endocrine-Related Cancer, 11, 2004, pp. 163-178.

[2] E.F. Petricoin, et al., "Use of proteomic patterns in serum to identify ovarian cancer", Lancet, 359(9306), 2002, pp. 572-577.

Character Recognition

This example illustrates how to train a neural network to perform simple character recognition.

Defining the Problem

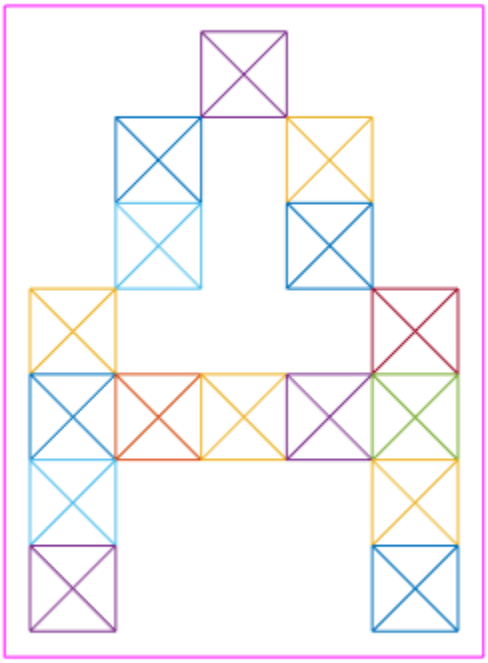
The script **prprob** defines a matrix X with 26 columns, one for each letter of the alphabet. Each column has 35 values which can either be 1 or 0. Each column of 35 values defines a 5x7 bitmap of a letter.

The matrix T is a 26x26 identity matrix which maps the 26 input vectors to the 26 classes.

```
[X,T] = prprob;
```

Here A, the first letter, is plotted as a bit map.

```
plotchar(X(:,1))
```

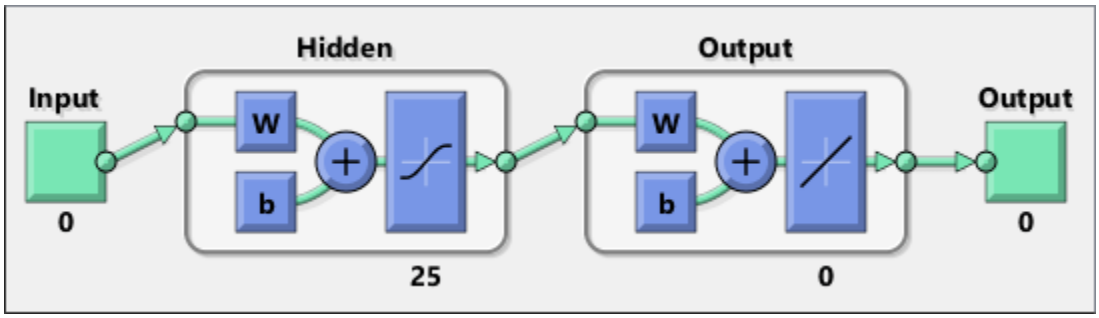


Creating the First Neural Network

To solve this problem we will use a feedforward neural network set up for pattern recognition with 25 hidden neurons.

Since the neural network is initialized with random initial weights, the results after training vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. This is not necessary for your own applications.

```
setdemorandstream(pi);

net1 = feedforwardnet(25);
view(net1)
```



Training the first Neural Network

The function **train** divides up the data into training, validation and test sets. The training set is used to update the network, the validation set is used to stop the network before it overfits the training data, thus preserving good generalization. The test set acts as a completely independent measure of how well the network can be expected to do on new samples.

Training stops when the network is no longer likely to improve on the training or validation sets.

```
net1.divideFcn = '';
net1 = train(net1,X,T,nnMATLAB);


Computing Resources:
MATLAB on PCWIN64
```

Training the Second Neural Network

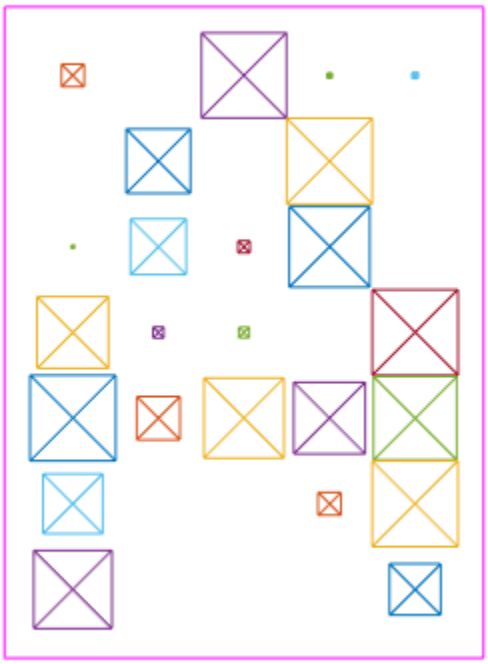
We would like the network to not only recognize perfectly formed letters, but also noisy versions of the letters. So we will try training a second network on noisy data and compare its ability to generalize with the first network.

Here 30 noisy copies of each letter Xn are created. Values are limited by **min** and **max** to fall between 0 and 1. The corresponding targets Tn are also defined.

```
numNoise = 30;
Xn = min(max(repmat(X,1,numNoise)+randn(35,26*numNoise)*0.2,0),1);
Tn = repmat(T,1,numNoise);
```

Here is a noise version of A.

```
figure
plotchar(Xn(:,1))
```



Here the second network is created and trained.

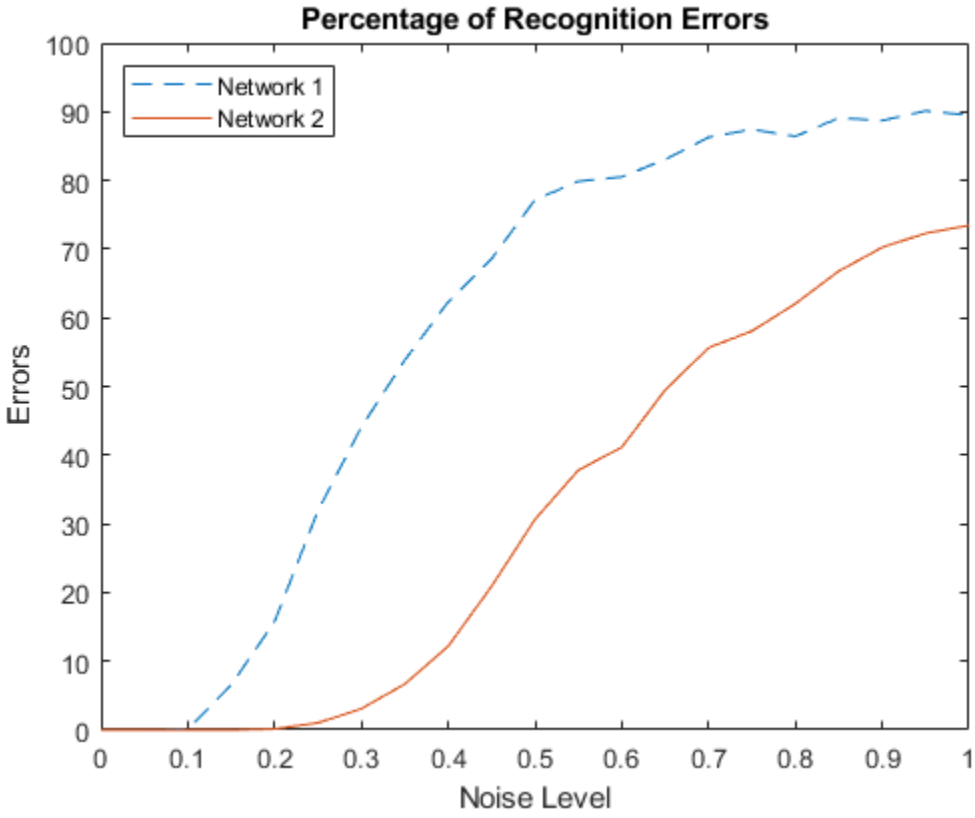
```
net2 = feedforwardnet(25);
net2 = train(net2,Xn,Tn,nnMATLAB);
```

```
Computing Resources:
MATLAB on PCWIN64
```

Testing Both Neural Networks

```
noiseLevels = 0:.05:1;
numLevels = length(noiseLevels);
percError1 = zeros(1,numLevels);
percError2 = zeros(1,numLevels);
for i = 1:numLevels
  Xtest = min(max(repmat(X,1,numNoise)+randn(35,26*numNoise)*noiseLevels(i),0),1);
  Y1 = net1(Xtest);
  percError1(i) = sum(sum(abs(Tn-compet(Y1))))/(26*numNoise*2);
  Y2 = net2(Xtest);
  percError2(i) = sum(sum(abs(Tn-compet(Y2))))/(26*numNoise*2);
end

figure
plot(noiseLevels,percError1*100,'--',noiseLevels,percError2*100);
title('Percentage of Recognition Errors');
xlabel('Noise Level');
```

```
ylabel('Errors');
legend('Network 1','Network 2','Location','NorthWest')
```



Network 1, trained without noise, has more errors due to noise than does Network 2, which was trained with noise.

Train Stacked Autoencoders for Image Classification

This example shows how to train stacked autoencoders to classify images of digits.

Neural networks with multiple hidden layers can be useful for solving classification problems with complex data, such as images. Each layer can learn features at a different level of abstraction. However, training neural networks with multiple hidden layers can be difficult in practice.

One way to effectively train a neural network with multiple layers is by training one layer at a time. You can achieve this by training a special type of network known as an autoencoder for each desired hidden layer.

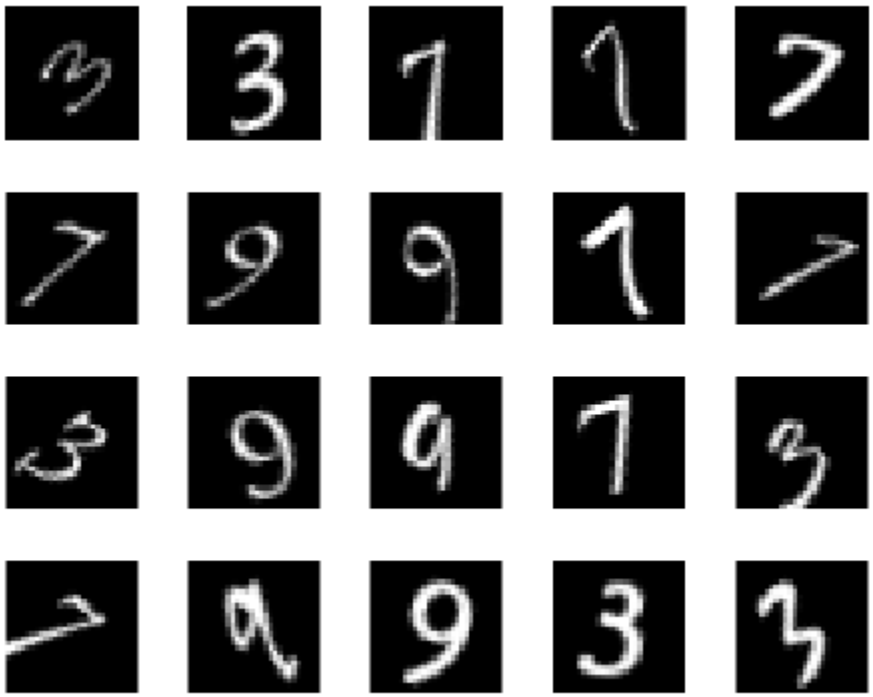
This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final softmax layer, and join the layers together to form a stacked network, which you train one final time in a supervised fashion.

Data set

This example uses synthetic data throughout, for training and testing. The synthetic images have been generated by applying random affine transformations to digit images created using different fonts.

Each digit image is 28-by-28 pixels, and there are 5,000 training examples. You can load the training data, and view some of the images.

```
% Load the training data into memory
[xTrainImages,tTrain] = digitTrainCellArrayData;

% Display some of the training images
clf
for i = 1:20
    subplot(4,5,i);
    imshow(xTrainImages{i});
end
```



The labels for the images are stored in a 10-by-5000 matrix, where in every column a single element will be 1 to indicate the class that the digit belongs to, and all other elements in the column will be 0. It should be noted that if the tenth element is 1, then the digit image is a zero.

Training the first autoencoder

Begin by training a sparse autoencoder on the training data without using the labels.

An autoencoder is a neural network which attempts to replicate its input at its output. Thus, the size of its input will be the same as the size of its output. When the number of neurons in the hidden layer is less than the size of the input, the autoencoder learns a compressed representation of the input.

Neural networks have weights randomly initialized before training. Therefore the results from training are different each time. To avoid this behavior, explicitly set the random number generator seed.

```
rng('default')
```

Set the size of the hidden layer for the autoencoder. For the autoencoder that you are going to train, it is a good idea to make this smaller than the input size.

```
hiddenSize1 = 100;
```

The type of autoencoder that you will train is a sparse autoencoder. This autoencoder uses regularizers to learn a sparse representation in the first layer. You can control the influence of these regularizers by setting various parameters:

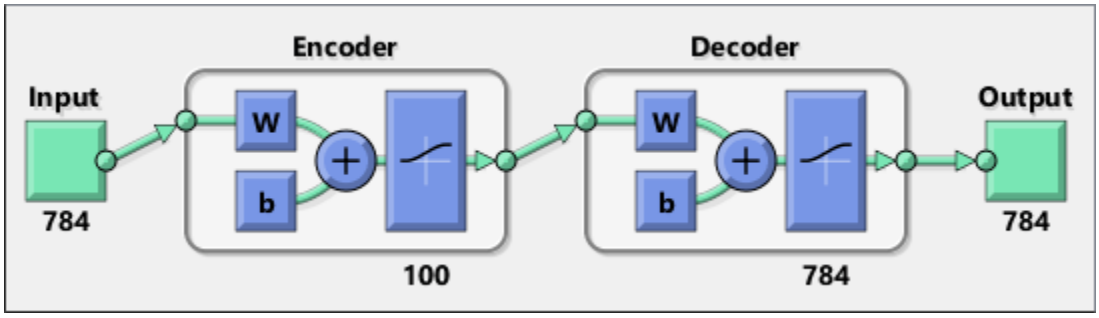
- `L2WeightRegularization` controls the impact of an L2 regularizer for the weights of the network (and not the biases). This should typically be quite small.
- `SparsityRegularization` controls the impact of a sparsity regularizer, which attempts to enforce a constraint on the sparsity of the output from the hidden layer. Note that this is different from applying a sparsity regularizer to the weights.
- `SparsityProportion` is a parameter of the sparsity regularizer. It controls the sparsity of the output from the hidden layer. A low value for `SparsityProportion` usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples. For example, if `SparsityProportion` is set to 0.1, this is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples. This value must be between 0 and 1. The ideal value varies depending on the nature of the problem.

Now train the autoencoder, specifying the values for the regularizers that are described above.

```
autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.004, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
```

You can view a diagram of the autoencoder. The autoencoder is comprised of an encoder followed by a decoder. The encoder maps an input to a hidden representation, and the decoder attempts to reverse this mapping to reconstruct the original input.

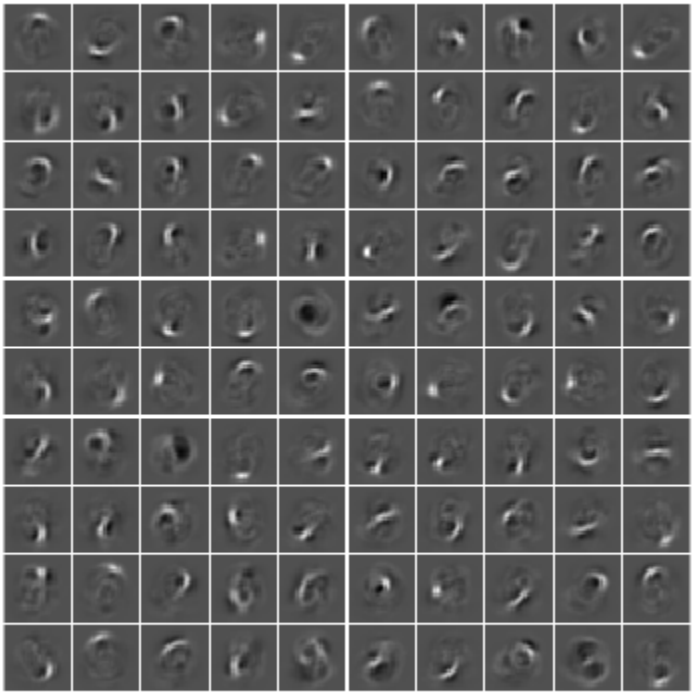
```
view(autoenc1)
```



Visualizing the weights of the first autoencoder

The mapping learned by the encoder part of an autoencoder can be useful for extracting features from data. Each neuron in the encoder has a vector of weights associated with it which will be tuned to respond to a particular visual feature. You can view a representation of these features.

```
figure()
plotWeights(autoenc1);
```



You can see that the features learned by the autoencoder represent curls and stroke patterns from the digit images.

The 100-dimensional output from the hidden layer of the autoencoder is a compressed version of the input, which summarizes its response to the features visualized above. Train the next autoencoder on a set of these vectors extracted from the training data. First, you must use the encoder from the trained autoencoder to generate the features.

```
feat1 = encode(autoenc1,xTrainImages);
```

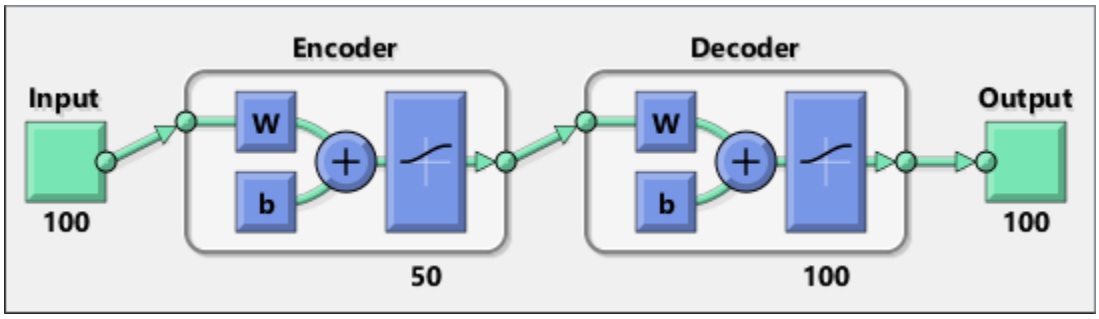
Training the second autoencoder

After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that you use the features that were generated from the first autoencoder as the training data in the second autoencoder. Also, you decrease the size of the hidden representation to 50, so that the encoder in the second autoencoder learns an even smaller representation of the input data.

```
hiddenSize2 = 50;
autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
    'MaxEpochs',100, ...
    'L2WeightRegularization',0.002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);
```

Once again, you can view a diagram of the autoencoder with the `view` function.

```
view(autoenc2)
```



You can extract a second set of features by passing the previous set through the encoder from the second autoencoder.

```
feat2 = encode(autoenc2,feat1);
```

The original vectors in the training data had 784 dimensions. After passing them through the first encoder, this was reduced to 100 dimensions. After using the second encoder, this was reduced again to 50 dimensions. You can now train a final layer to classify these 50-dimensional vectors into different digit classes.

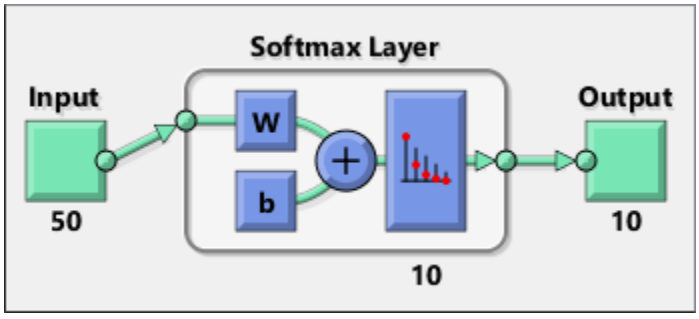
Training the final softmax layer

Train a softmax layer to classify the 50-dimensional feature vectors. Unlike the autoencoders, you train the softmax layer in a supervised fashion using labels for the training data.

```
softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',400);
```

You can view a diagram of the softmax layer with the `view` function.

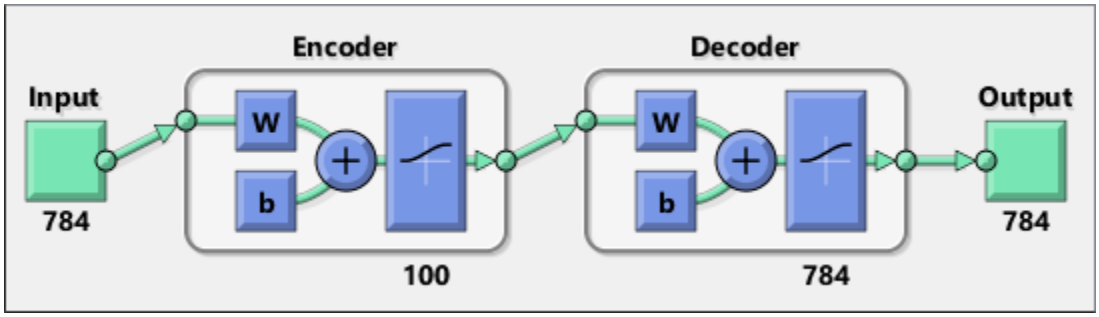
```
view(softnet)
```

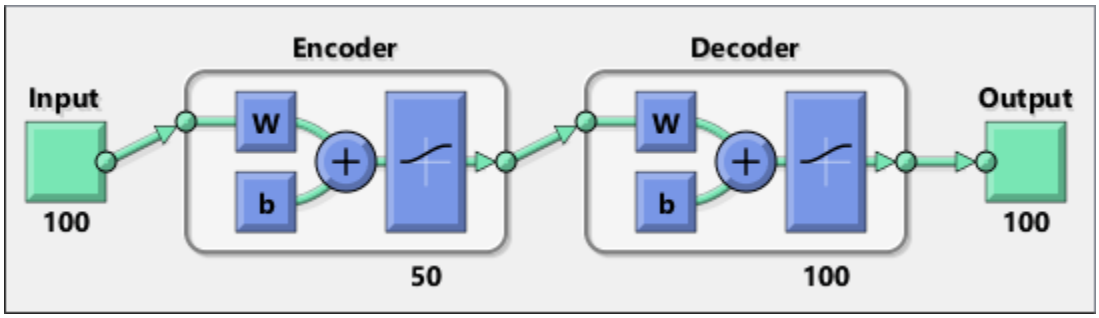


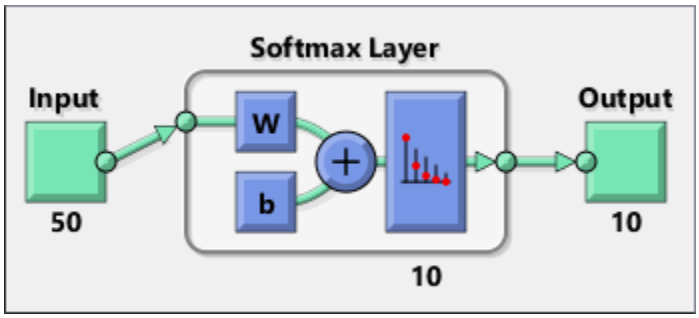
Forming a stacked neural network

You have trained three separate components of a stacked neural network in isolation. At this point, it might be useful to view the three neural networks that you have trained. They are `autoenc1`, `autoenc2`, and `softnet`.

```
view(autoenc1)
view(autoenc2)
view(softnet)
```





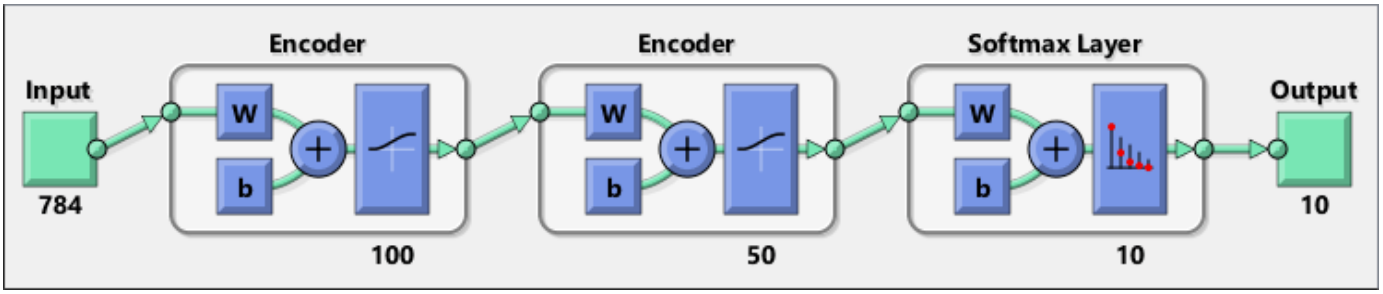


As was explained, the encoders from the autoencoders have been used to extract features. You can stack the encoders from the autoencoders together with the softmax layer to form a stacked network for classification.

```
stackednet = stack(autoenc1,autoenc2,softnet);
```

You can view a diagram of the stacked network with the `view` function. The network is formed by the encoders from the autoencoders and the softmax layer.

```
view(stackednet)
```

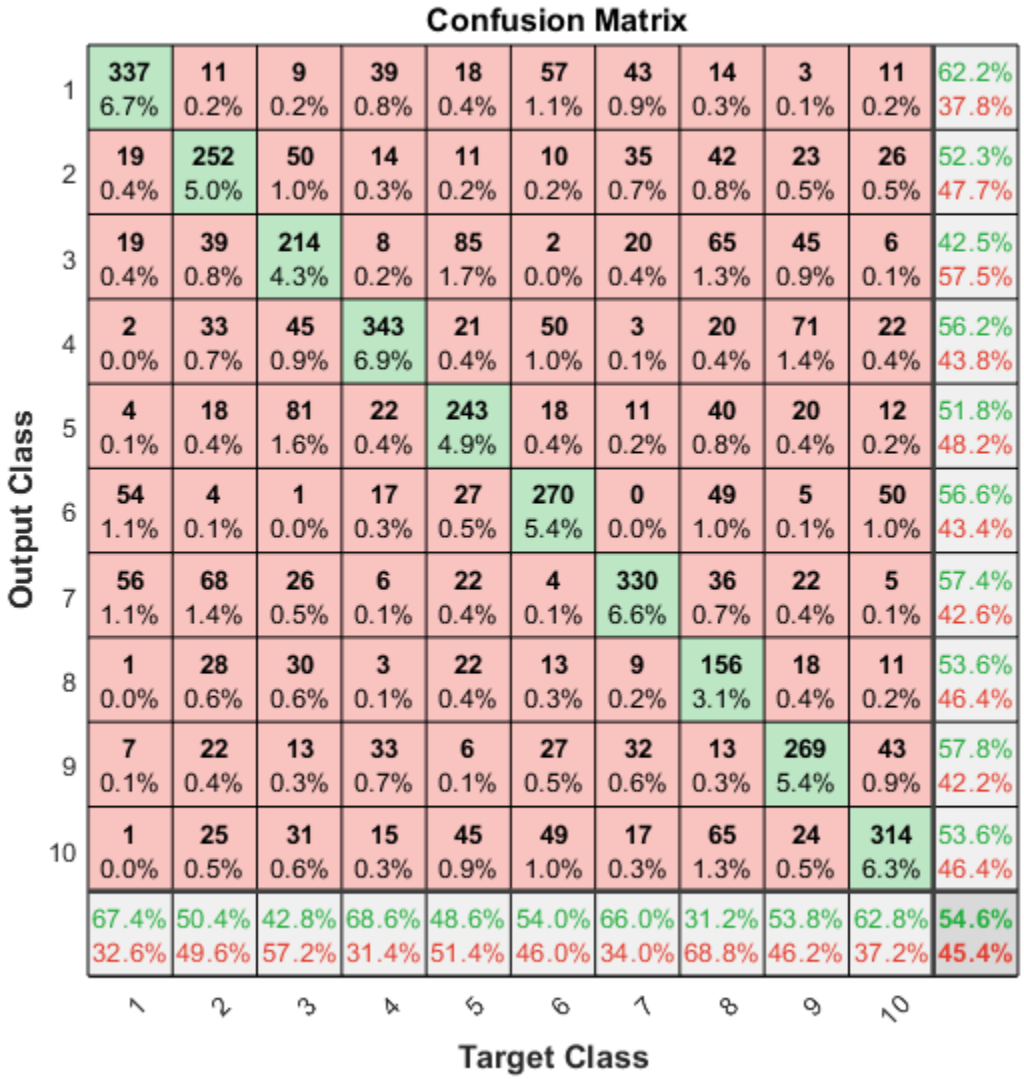


With the full network formed, you can compute the results on the test set. To use images with the stacked network, you have to reshape the test images into a matrix. You can do this by stacking the columns of an image to form a vector, and then forming a matrix from these vectors.

```
% Get the number of pixels in each image
imageWidth = 28;
imageHeight = 28;
inputSize = imageWidth*imageHeight;

% Load the test images
[xTestImages,tTest] = digitTestCellArrayData;

% Turn the test images into vectors and put them in a matrix
xTest = zeros(inputSize,numel(xTestImages));
for i = 1:numel(xTestImages)
    xTest(:,i) = xTestImages{i}(:);
end
```

You can visualize the results with a confusion matrix. The numbers in the bottom right-hand square of the matrix give the overall accuracy.

```
y = stackednet(xTest);
plotconfusion(tTest,y);
```



Fine tuning the stacked neural network

The results for the stacked neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

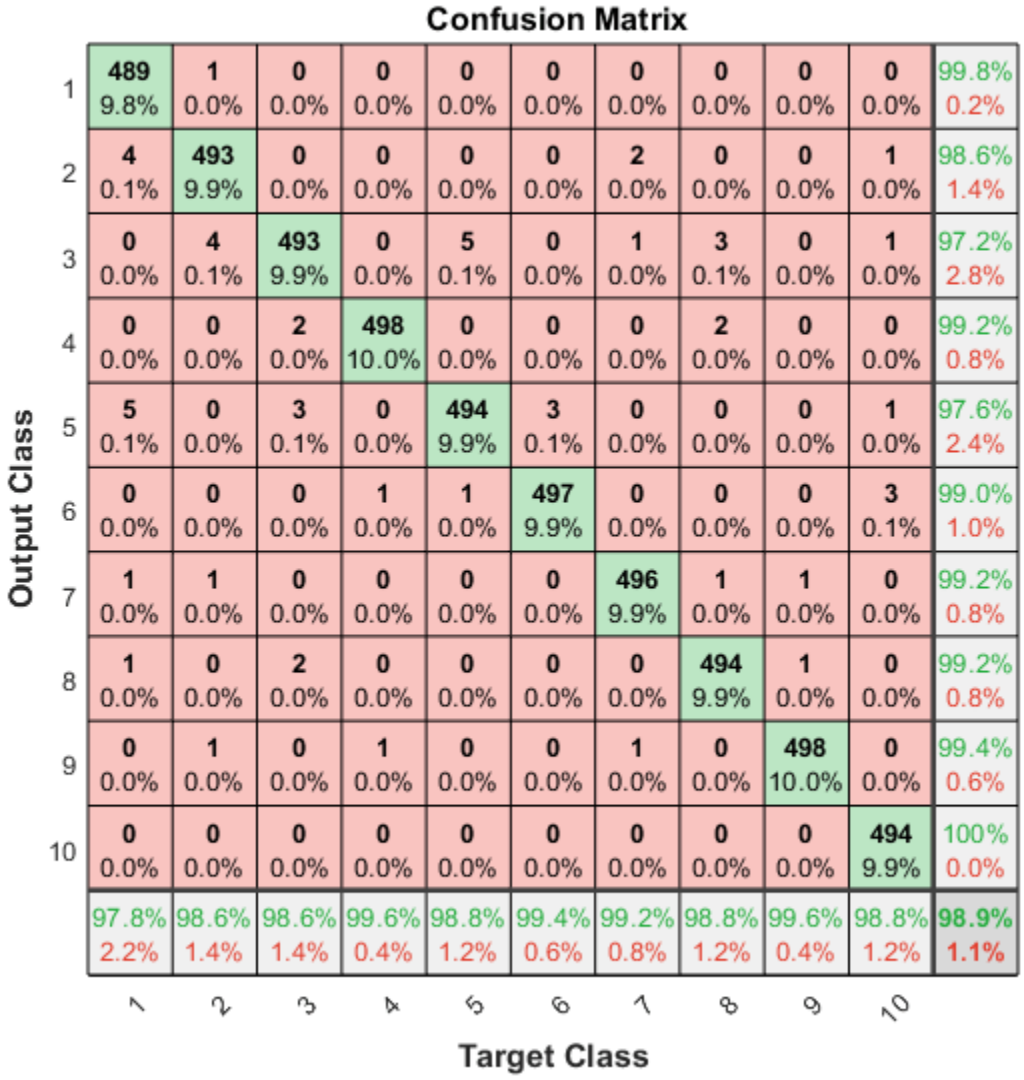
You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

```
% Turn the training images into vectors and put them in a matrix
xTrain = zeros(inputSize,numel(xTrainImages));
for i = 1:numel(xTrainImages)
    xTrain(:,i) = xTrainImages{i}(:);
end
```

```
% Perform fine tuning
stackednet = train(stackednet,xTrain,tTrain);
```

You then view the results again using a confusion matrix.

```
y = stackednet(xTest);
plotconfusion(tTest,y);
```



Summary

This example showed how to train a stacked neural network to classify digits in images using autoencoders. The steps that have been outlined can be applied to other similar problems, such as classifying images of letters, or even small images of objects of a specific category.

Iris Clustering

This example illustrates how a self-organizing map neural network can cluster iris flowers into classes topologically, providing insight into the types of flowers and a useful tool for further analysis.

The Problem: Cluster Iris Flowers

In this example we attempt to build a neural network that clusters iris flowers into natural classes, such that similar classes are grouped together. Each iris is described by four features:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm

This is an example of a clustering problem, where we would like to group samples into classes based on the similarity between samples. We would like to create a neural network which not only creates class definitions for the known inputs, but will also let us classify unknown inputs accordingly.

Why Self-Organizing Map Neural Networks?

Self-organizing maps (SOMs) are very good at creating classifications. Further, the classifications retain topological information about which classes are most similar to others. Self-organizing maps can be created with any desired level of detail. They are particularly well suited for clustering data in many dimensions and with complexly shaped and connected feature spaces. They are well suited to cluster iris flowers.

The four flower attributes will act as inputs to the SOM, which will map them onto a 2-dimensional layer of neurons.

Preparing the Data

Data for clustering problems are set up for a SOM by organizing the data into an input matrix X.

Each ith column of the input matrix will have four elements representing the four measurements taken on a single flower.

Here such a dataset is loaded.

```
x = iris_dataset;
```

We can view the size of inputs X.

Note that X has 150 columns. These represent 150 sets of iris flower attributes. It has four rows, for the four measurements.

```
size(x)
```

```
ans = 1×2

     4   150
```

Clustering with a Neural Network

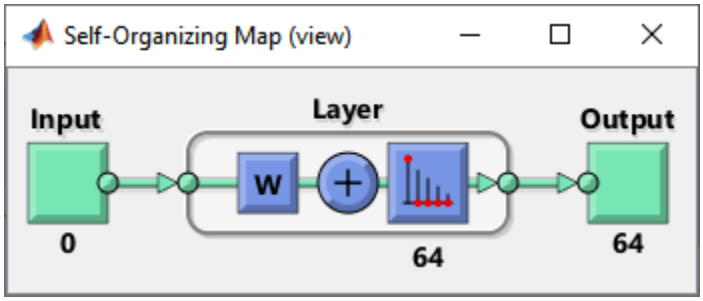
The next step is to create a neural network that will learn to cluster.

selforgmap creates self-organizing maps for classifying samples with as much detail as desired by selecting the number of neurons in each dimension of the layer.

We will try a 2-dimension layer of 64 neurons arranged in an 8x8 hexagonal grid for this example. In general, greater detail is achieved with more neurons, and more dimensions allows for modelling the topology of more complex feature spaces.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([8 8]);
view(net)
```

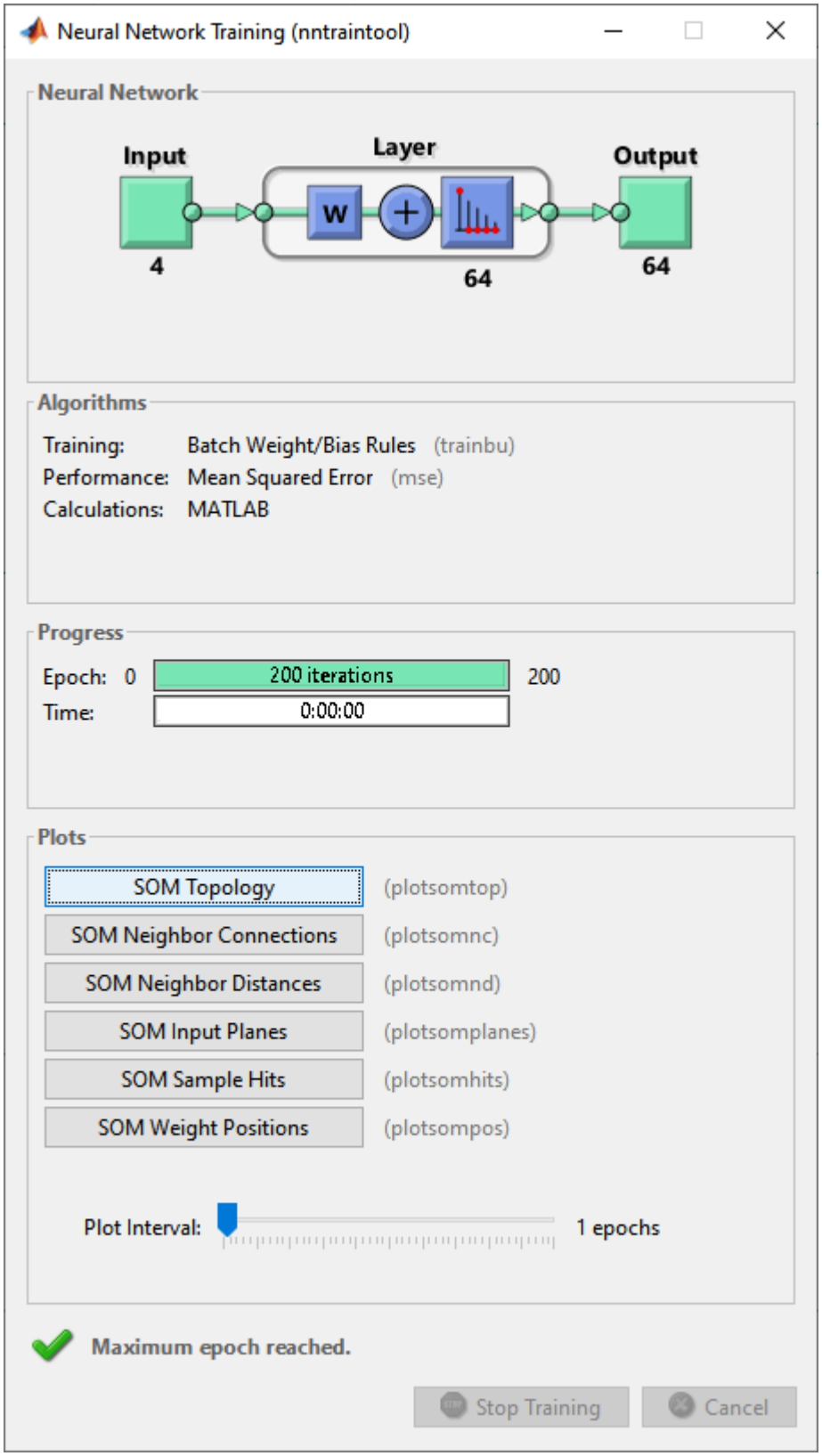


Now the network is ready to be optimized with **train**.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,x);
```


Neural Network Training (nntraintool)
Neural Network
Input
Layer
Output
4
64
64
Algorithms
Training: Batch Weight/Bias Rules (trainbu)
Performance: Mean Squared Error (mse)
Calculations: MATLAB
Progress
Epoch: 0
200 iterations
200
Time:
0:00:00
Plots
SOM Topology (plotsomtop)
SOM Neighbor Connections (plotsomnc)
SOM Neighbor Distances (plotsomnd)
SOM Input Planes (plotsomplanes)
SOM Sample Hits (plotsomhits)
SOM Weight Positions (plotsompos)
Plot Interval:
1 epochs
Maximum epoch reached.
Stop Training
Cancel

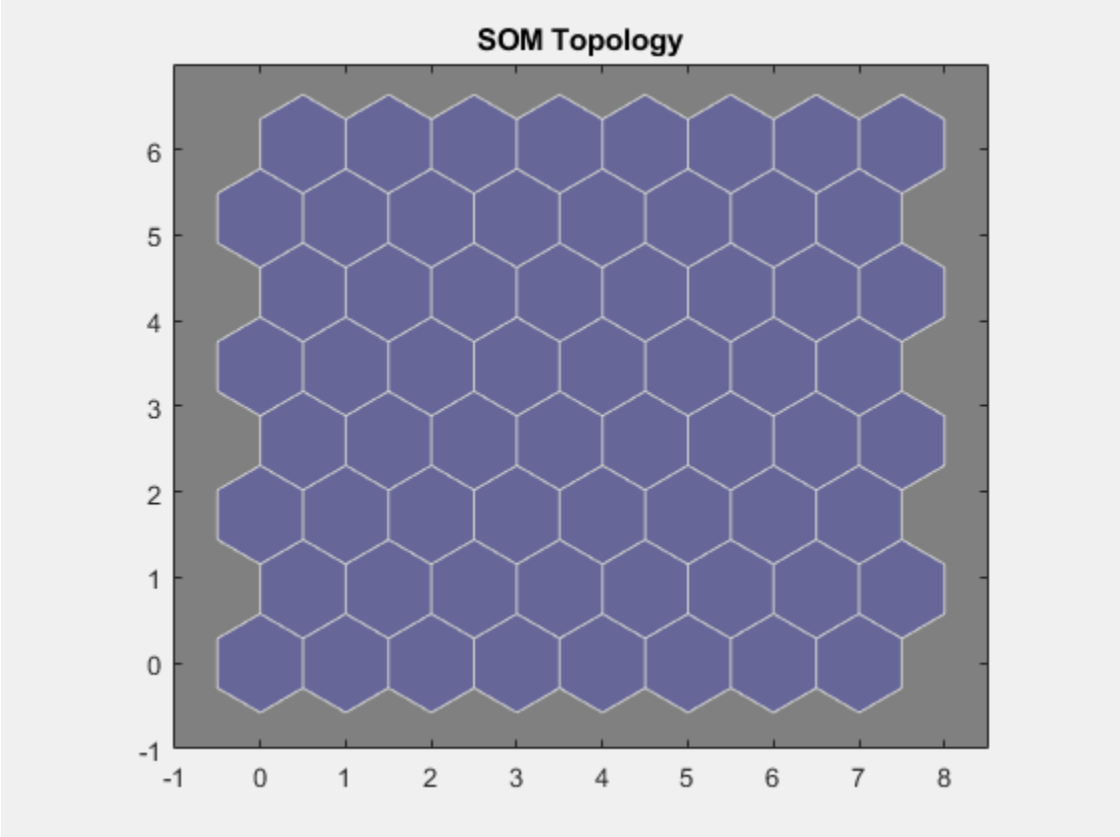
Here the self-organizing map is used to compute the class vectors of each of the training inputs. These classifications cover the feature space populated by the known flowers, and can now be used to classify new flowers accordingly. The network output will be a 64x150 matrix, where each ith column represents the jth cluster for each ith input vector with a 1 in its jth element.

The function **vec2ind** returns the index of the neuron with an output of 1, for each vector. The indices will range between 1 and 64 for the 64 clusters represented by the 64 neurons.

```
y = net(x);
cluster_index = vec2ind(y);
```

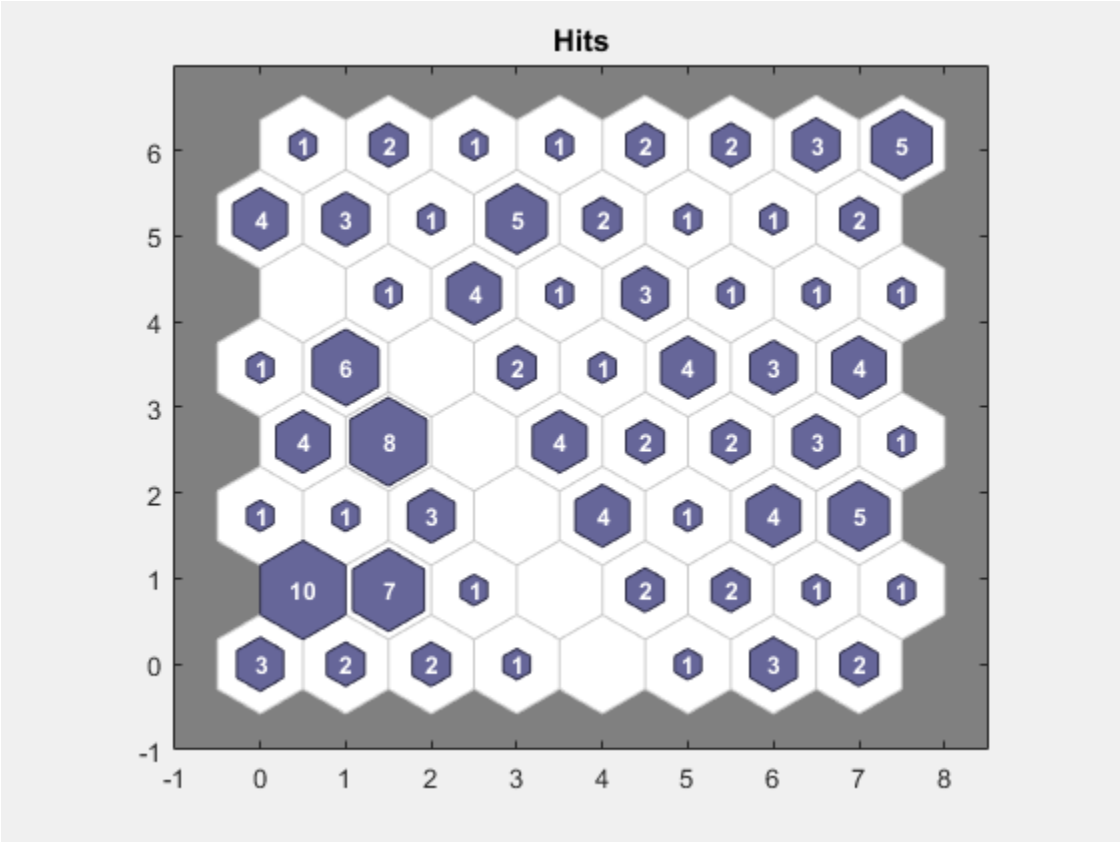
plotsomtop plots the self-organizing maps topology of 64 neurons positioned in an 8x8 hexagonal grid. Each neuron has learned to represent a different class of flower, with adjacent neurons typically representing similar classes.

```
plotsomtop(net)
```



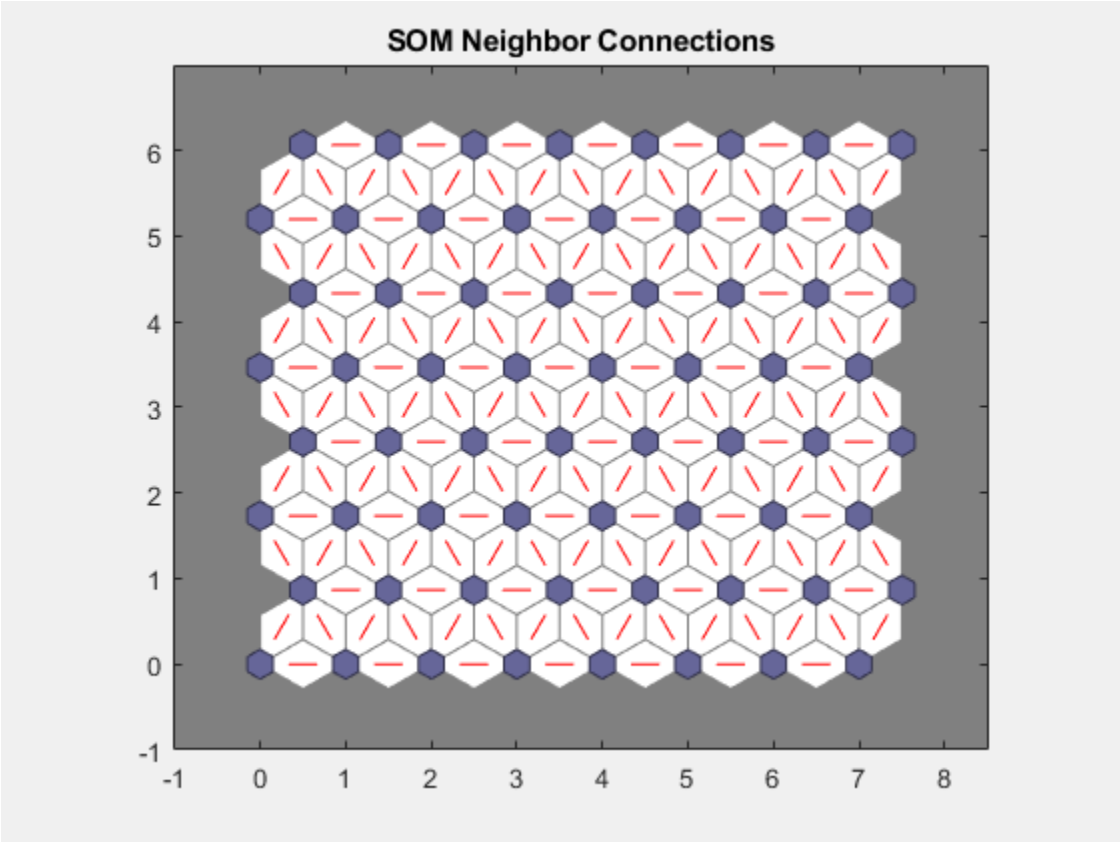
plotsomhits calculates the classes for each flower and shows the number of flowers in each class. Areas of neurons with large numbers of hits indicate classes representing similar highly populated regions of the feature space. Whereas areas with few hits indicate sparsely populated regions of the feature space.

```
plotsomhits(net,x)
```



plotsomnc shows the neuron neighbor connections. Neighbors typically classify similar samples.

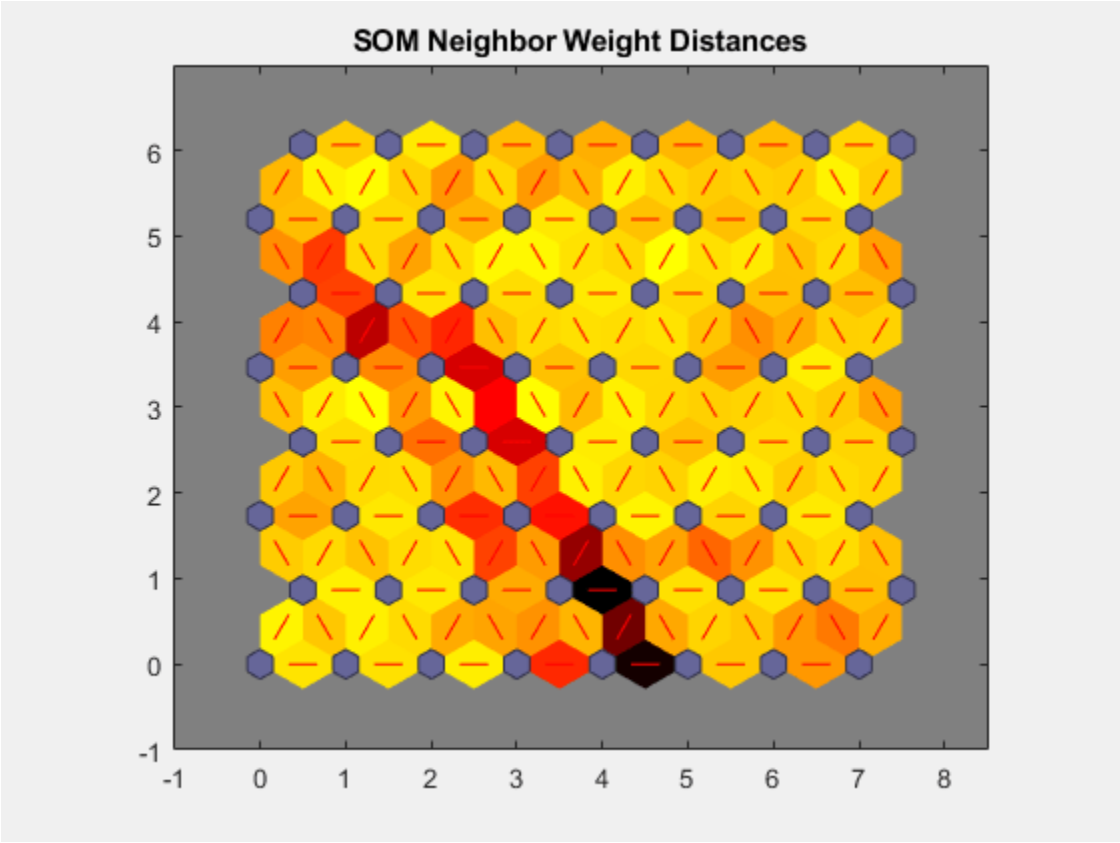
```
plotsomnc(net)
```



plotsomnd shows how distant (in terms of Euclidian distance) each neuron's class is from its neighbors. Connections which are bright indicate highly connected areas of the input space. While dark connections indicate classes representing regions of the feature space which are far apart, with few or no flowers between them.

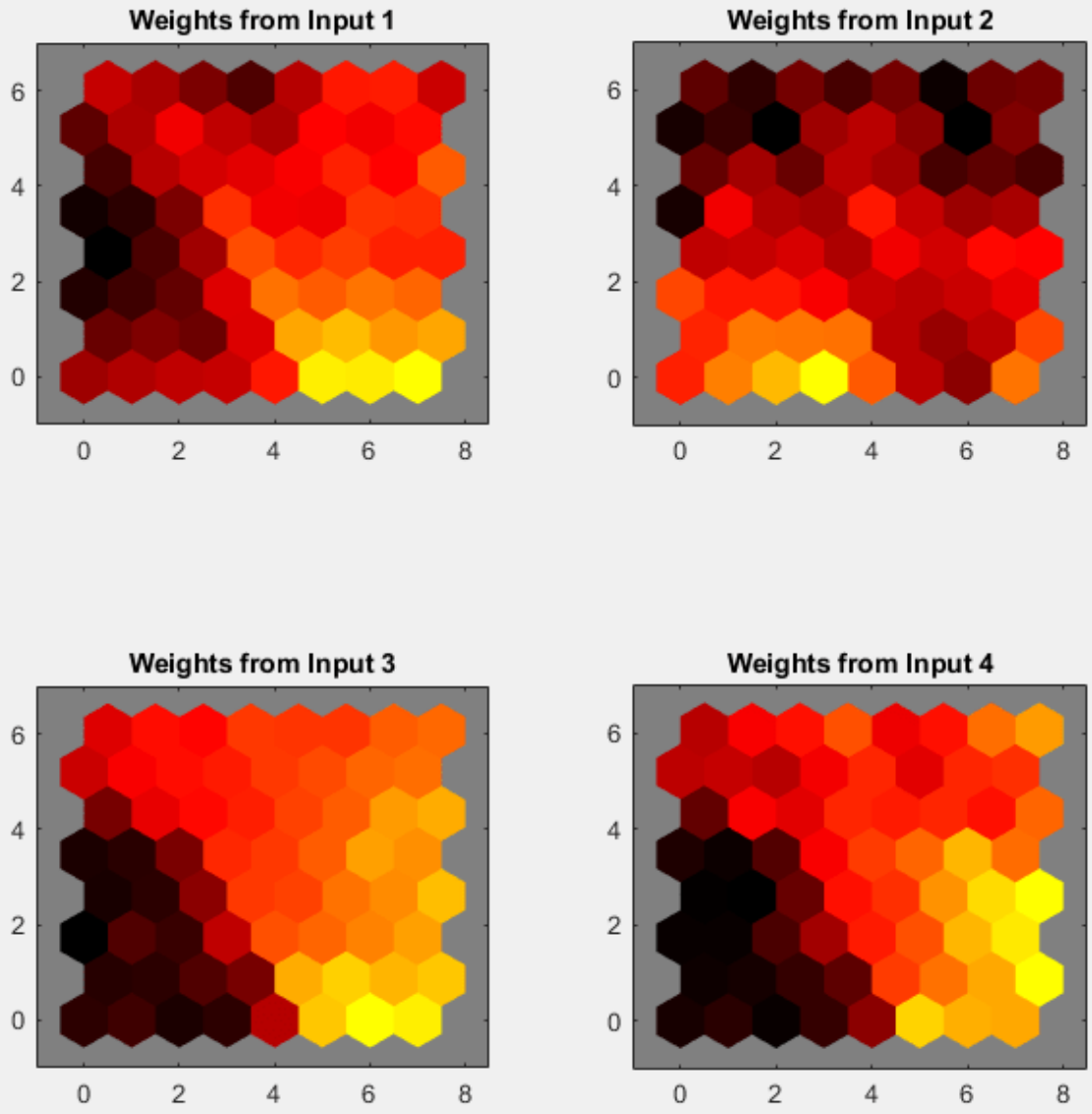
Long borders of dark connections separating large regions of the input space indicate that the classes on either side of the border represent flowers with very different features.

```
plotsomnd(net)
```



plotsomplanes shows a weight plane for each of the four input features. They are visualizations of the weights that connect each input to each of the 64 neurons in the 8x8 hexagonal grid. Darker colors represent larger weights. If two inputs have similar weight planes (their color gradients may be the same or in reverse) it indicates they are highly correlated.

```
plotsomplanes(net)
```



This example illustrated how to design a neural network that clusters iris flowers based on four of their characteristics.

Explore other examples and the documentation for more insight into neural networks and their applications.

Gene Expression Analysis

This example demonstrates looking for patterns in gene expression profiles in baker's yeast using neural networks.

The Problem: Analyzing Gene Expressions in Baker's Yeast (Saccharomyces Cerevisiae)

The goal is to gain some understanding of gene expressions in Saccharomyces cerevisiae, which is commonly known as baker's yeast or brewer's yeast. It is the fungus that is used to bake bread and ferment wine from grapes.

Saccharomyces cerevisiae, when introduced in a medium rich in glucose, can convert glucose to ethanol. Initially, yeast converts glucose to ethanol by a metabolic process called "fermentation". However, once the supply of glucose is exhausted yeast shifts from anaerobic fermentation of glucose to aerobic respiration of ethanol. This process is called diauxic shift. This process is of considerable interest since it is accompanied by major changes in gene expression.

The example uses DNA microarray data to study temporal gene expression of almost all genes in Saccharomyces cerevisiae during the diauxic shift.

You need Bioinformatics Toolbox™ to run this example.

```
if ~nnDependency.bioInfoAvailable
    errordlg('This example requires Bioinformatics Toolbox.');
    return;
end
```

The Data

This example uses data from DeRisi, JL, Iyer, VR, Brown, PO. "Exploring the metabolic and genetic control of gene expression on a genomic scale." Science. 1997 Oct 24;278(5338):680-6. PMID: 9381177

The full data set can be downloaded from the Gene Expression Omnibus website: https://www.yeastgenome.org

Start by loading the data into MATLAB®.

```
load yeastdata.mat
```

Gene expression levels were measured at seven time points during the diauxic shift. The variable `times` contains the times at which the expression levels were measured in the experiment. The variable `genes` contains the names of the genes whose expression levels were measured. The variable `yeastvalues` contains the "VALUE" data or LOG_RAT2N_MEAN, or log2 of ratio of CH2DN_MEAN and CH1DN_MEAN from the seven time steps in the experiment.

To get an idea of the size of the data you can use **numel(genes)** to show how many genes there are in the data set.

```
numel(genes)
```

```
ans = 6400
```

genes is a cell array of the gene names. You can access the entries using MATLAB cell array indexing:

```
genes{15}
```

```
ans =
'YAL054C'
```

This indicates that the 15th row of the variable **yeastvalues** contains expression levels for the ORF YAL054C.

Filtering the Genes

The data set is quite large and a lot of the information corresponds to genes that do not show any interesting changes during the experiment. To make it easier to find the interesting genes, the first thing to do is to reduce the size of the data set by removing genes with expression profiles that do not show anything of interest. There are 6400 expression profiles. You can use a number of techniques to reduce this to some subset that contains the most significant genes.

If you look through the gene list you will see several spots marked as 'EMPTY'. These are empty spots on the array, and while they might have data associated with them, for the purposes of this example, you can consider these points to be noise. These points can be found using the **strcmp** function and removed from the data set with indexing commands.

```
emptySpots = strcmp('EMPTY',genes);
yeastvalues(emptySpots,:) = [];
genes(emptySpots) = [];
numel(genes)
```

```
ans = 6314
```

In the yeastvalues data you will also see several places where the expression level is marked as NaN. This indicates that no data was collected for this spot at the particular time step. One approach to dealing with these missing values would be to impute them using the mean or median of data for the particular gene over time. This example uses a less rigorous approach of simply throwing away the data for any genes where one or more expression level was not measured.

The function **isnan** is used to identify the genes with missing data and indexing commands are used to remove the genes with missing data.

```
nanIndices = any(isnan(yeastvalues),2);
yeastvalues(nanIndices,:) = [];
genes(nanIndices) = [];
numel(genes)
```

```
ans = 6276
```

If you were to plot the expression profiles of all the remaining profiles, you would see that most profiles are flat and not significantly different from the others. This flat data is obviously of use as it indicates that the genes associated with these profiles are not significantly affected by the diauxic shift; however, in this example, you are interested in the genes with large changes in expression accompanying the diauxic shift. You can use filtering functions in the Bioinformatics Toolbox™ to remove genes with various types of profiles that do not provide useful information about genes affected by the metabolic change.

You can use the **genevarfilter** function to filter out genes with small variance over time. The function returns a logical array of the same size as the variable genes with ones corresponding to rows of yeastvalues with variance greater than the 10th percentile and zeros corresponding to those below the threshold.

```
mask = genevarfilter(yeastvalues);
% Use the mask as an index into the values to remove the filtered genes.
```

```
yeastvalues = yeastvalues(mask,:);
genes = genes(mask);
numel(genes)

ans = 5648
```

The function **genelowvalfilter** removes genes that have very low absolute expression values. Note that the gene filter functions can also automatically calculate the filtered data and names.

```
[mask, yeastvalues, genes] = ...
    genelowvalfilter(yeastvalues,genes,'absval',log2(3));
numel(genes)

ans = 822
```

Use **geneentropyfilter** to remove genes whose profiles have low entropy:

```
[mask, yeastvalues, genes] = ...
    geneentropyfilter(yeastvalues,genes,'prctile',15);
numel(genes)

ans = 614
```

Principal Component Analysis

Now that you have a manageable list of genes, you can look for relationships between the profiles.

Normalizing the standard deviation and mean of data allows the network to treat each input as equally important over its range of values.

Principal-component analysis (PCA) is a useful technique that can be used to reduce the dimensionality of large data sets, such as those from microarray analysis. This technique isolates the principal components of the dataset eliminating those components that contribute the least to the variation in the data set.

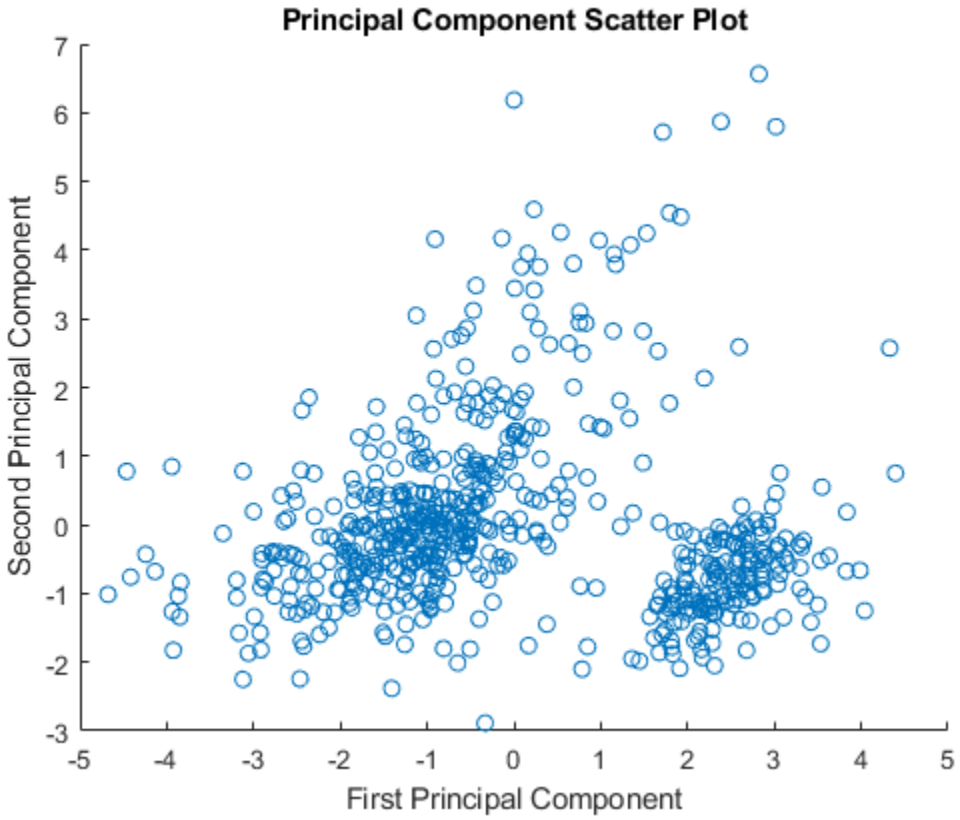
The two settings variables can be used to apply **mapstd** and **processpca** to new data to be consistent.

```
[x,std_settings] = mapstd(yeastvalues');  % Normalize data
[x,pca_settings] = processpca(x,0.15);    % PCA
```

The input vectors are first normalized, using `mapstd`, so that they have zero mean and unity variance. `processpca` is the function that implements the PCA algorithm. The second argument passed to `processpca` is 0.15. This means that `processpca` eliminates those principal components that contribute less than 15% to the total variation in the data set. The variable `pc` now contains the principal components of the yeastvalues data.

The principal components can be visualized using the **scatter** function.

```
figure
scatter(x(1,:),x(2,:));
xlabel('First Principal Component');
ylabel('Second Principal Component');
title('Principal Component Scatter Plot');
```



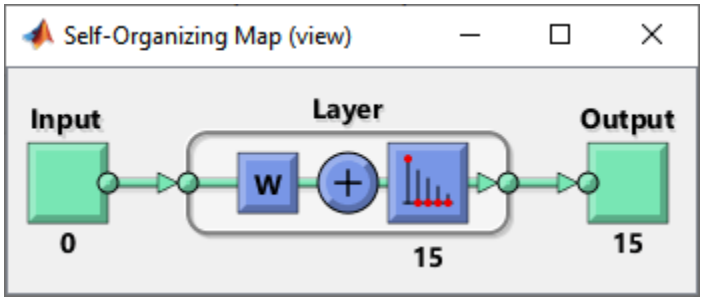
Cluster Analysis: Self-Organizing Maps

The principal components can now be clustered using the Self-Organizing map (SOM) clustering algorithm.

The **selforgmap** function creates a Self-Organizing map network which can then be trained with the **train** function.

The input size is 0 because the network has not yet been configured to match our input data. This will happen when the network is trained.

```
net = selforgmap([5 3]);
view(net)
```

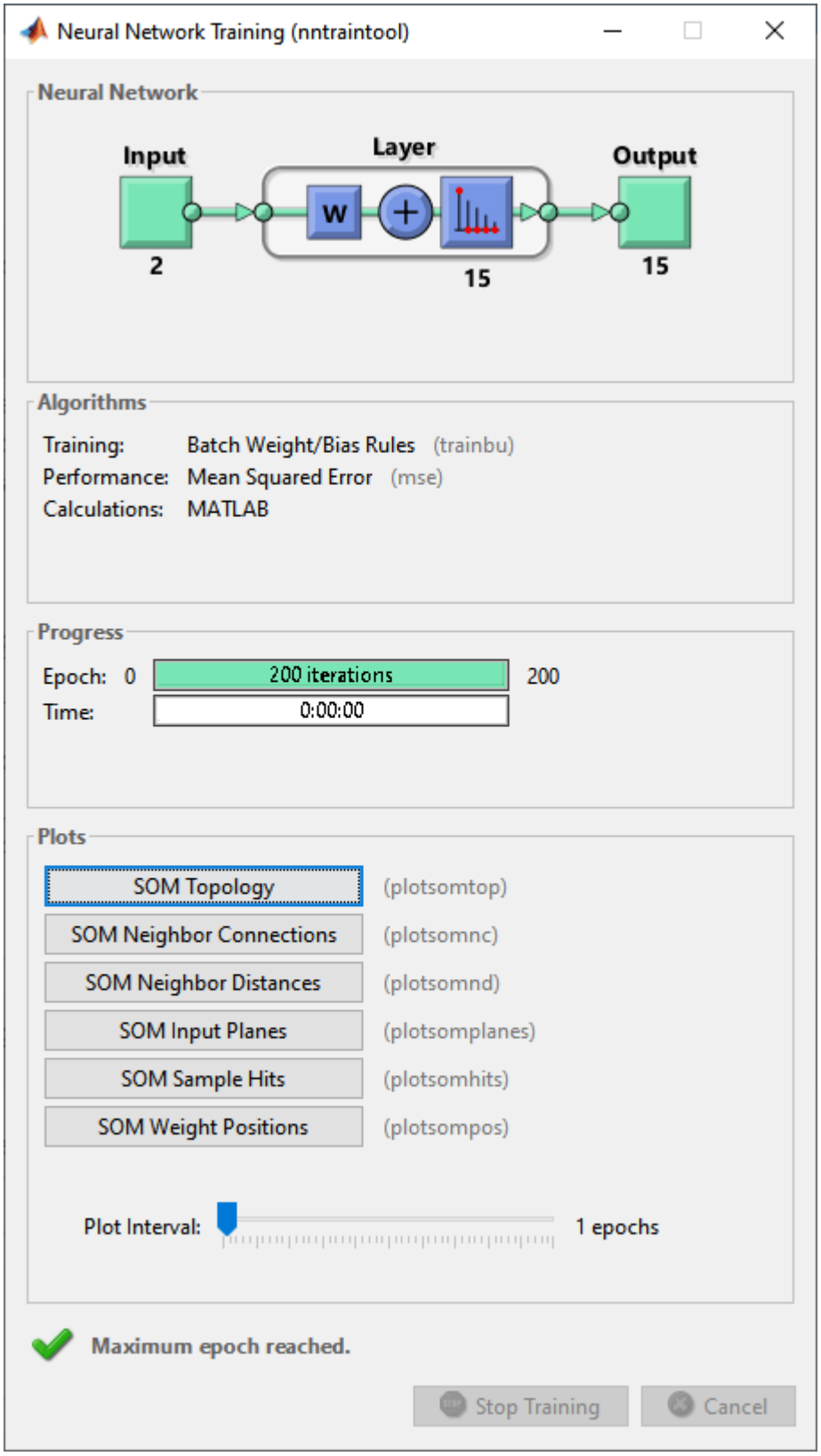


Now the network is ready to be trained.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

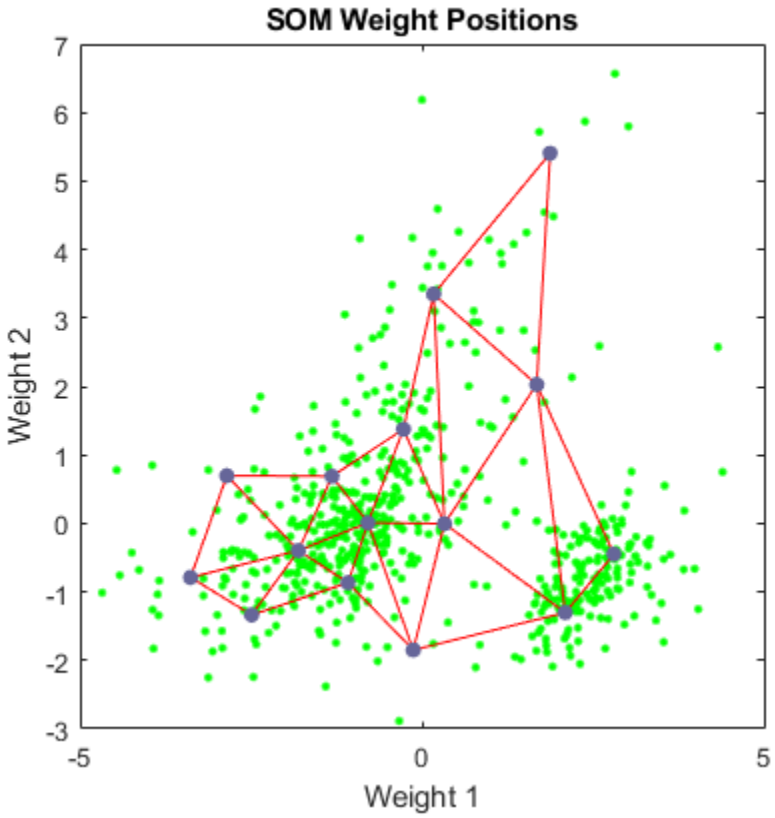
The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
net = train(net,x);
```


Neural Network Training (nntraintool)
Neural Network
Input
Layer
Output
W
2
15
15
Algorithms
Training: Batch Weight/Bias Rules (trainbu)
Performance: Mean Squared Error (mse)
Calculations: MATLAB
Progress
Epoch: 0
200 iterations
200
Time:
0:00:00
Plots
SOM Topology (plotsomtop)
SOM Neighbor Connections (plotsomnc)
SOM Neighbor Distances (plotsomnd)
SOM Input Planes (plotsomplanes)
SOM Sample Hits (plotsomhits)
SOM Weight Positions (plotsompos)
Plot Interval:
1 epochs
Maximum epoch reached.
Stop Training
Cancel

Use **plotsompos** to display the network over a scatter plot of the first two dimensions of the data.

```
figure
plotsompos(net,x);
```

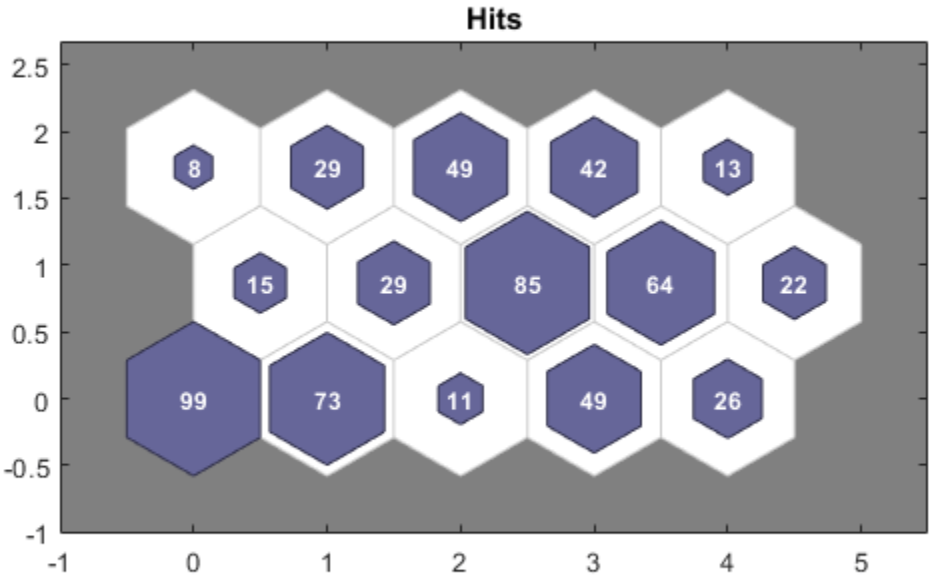


You can assign clusters using the SOM by finding the nearest node to each point in the data set.

```
y = net(x);
cluster_indices = vec2ind(y);
```

Use **plotsomhits** to see how many vectors are assigned to each of the neurons in the map.

```
figure
plotsomhits(net,x);
```



You can also use other clustering algorithms like Hierarchical clustering and K-means, available in the Statistics and Machine Learning Toolbox™ for cluster analysis.

Glossary

ORF - An open reading frame (ORF) is a portion of a gene's sequence that contains a sequence of bases, uninterrupted by stop sequences, that could potentially encode a protein.

Maglev Modeling

This example illustrates how a NARX (Nonlinear AutoRegressive with eXternal input) neural network can model a magnet levitation dynamical system.

The Problem: Model a Magnetic Levitation System

In this example we attempt to build a neural network that can predict the dynamic behavior of a magnet levitated using a control current.

The system is characterized by the magnet's position and a control current, both of which determine where the magnet will be an instant later.

This is an example of a time series problem, where past values of a feedback time series (the magnet position) and an external input series (the control current) are used to predict future values of the feedback series.

Why Neural Networks?

Neural networks are very good at time series problems. A neural network with enough elements (called neurons) can model dynamic systems with arbitrary accuracy. They are particularly well suited for addressing non-linear dynamic problems. Neural networks are a good candidate for solving this problem.

The network will be designed by using recordings of an actual levitated magnet's position responding to a control current.

Preparing the Data

Data for function fitting problems are set up for a neural network by organizing the data into two matrices, the input time series X and the target time series T.

The input series X is a row cell array, where each element is the associated timestep of the control current.

The target series T is a row cell array, where each element is the associated timestep of the levitated magnet position.

Here such a dataset is loaded.

```
[x,t] = maglev_dataset;
```

We can view the sizes of inputs X and targets T.

Note that both X and T have 4001 columns. These represent 4001 timesteps of the control current and magnet position.

```
size(x)
```

```
ans = 1×2

           1        4001
```

```
size(t)
```

```
ans = 1×2

     1        4001
```

Time Series Modelling with a Neural Network

The next step is to create a neural network that will learn to model how the magnet changes position.

Since the neural network starts with random initial weights, the results of this example will differ slightly every time it is run. The random seed is set to avoid this randomness. However this is not necessary for your own applications.

```
setdemorandstream(491218381)
```

Two-layer (i.e. one-hidden-layer) NARX neural networks can fit any dynamical input-output relationship given enough neurons in the hidden layer. Layers which are not output layers are called hidden layers.

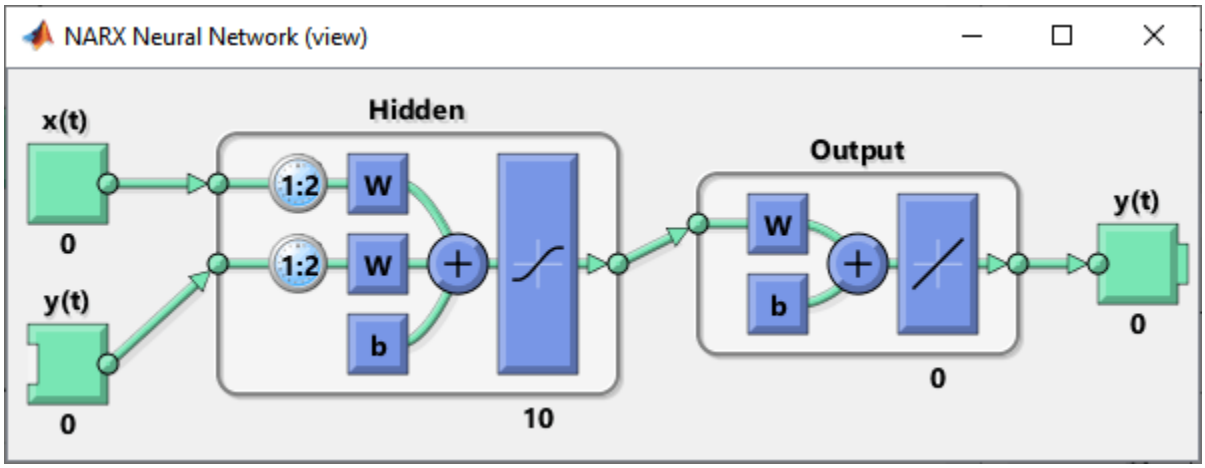
We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

We will also try using tap delays with two delays for the external input (control current) and feedback (magnet position). More delays allow the network to model more complex dynamic systems.

The input and output have sizes of 0 because the network has not yet been configured to match our input and target data. This will happen when the network is trained.

The output y(t) is also an input, whose delayed version is fed back into the network.

```
net = narxnet(1:2,1:2,10);
view(net)
```



Before we can train the network, we must use the first two timesteps of the external input and feedback time series to fill the two tap delay states of the network.

Furthermore, we need to use the feedback series both as an input series and target series.

The function PREPARETS prepares time series data for simulation and training for us. Xs will consist of shifted input and target series to be presented to the network. Xi is the initial input delay states. Ai

is the layer delay states (empty in this case as there are no layer-to-layer delays), and Ts is the shifted feedback series.

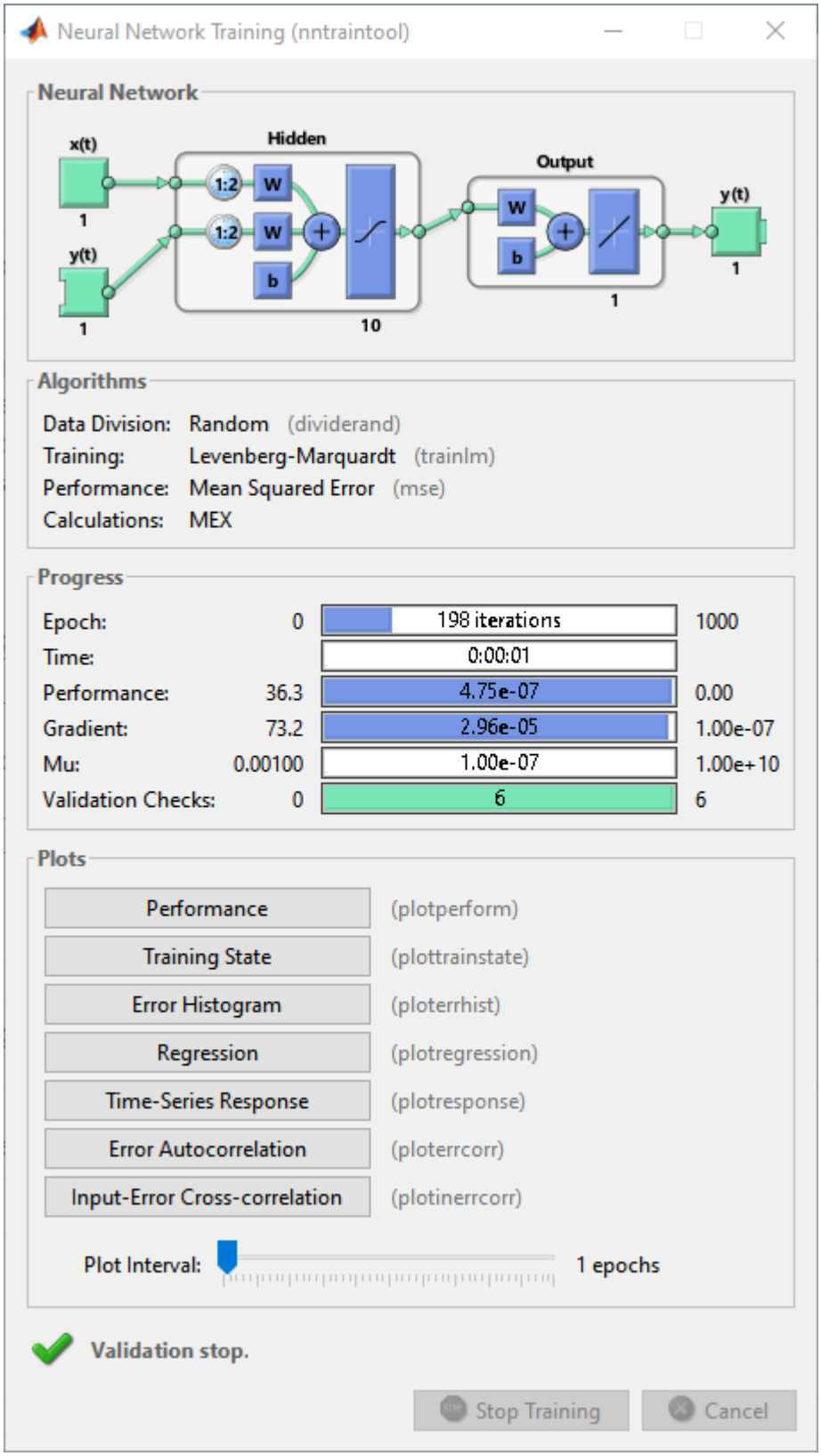
```
[Xs,Xi,Ai,Ts] = preparets(net,x,{},t);
```

Now the network is ready to be trained. The timesteps are automatically divided into training, validation and test sets. The training set is used to teach the network. Training continues as long as the network continues improving on the validation set. The test set provides a completely independent measure of network accuracy.

The Neural Network Training Tool shows the network being trained and the algorithms used to train it. It also displays the training state during training and the criteria which stopped training will be highlighted in green.

The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
[net,tr] = train(net,Xs,Ts,Xi,Ai);
```

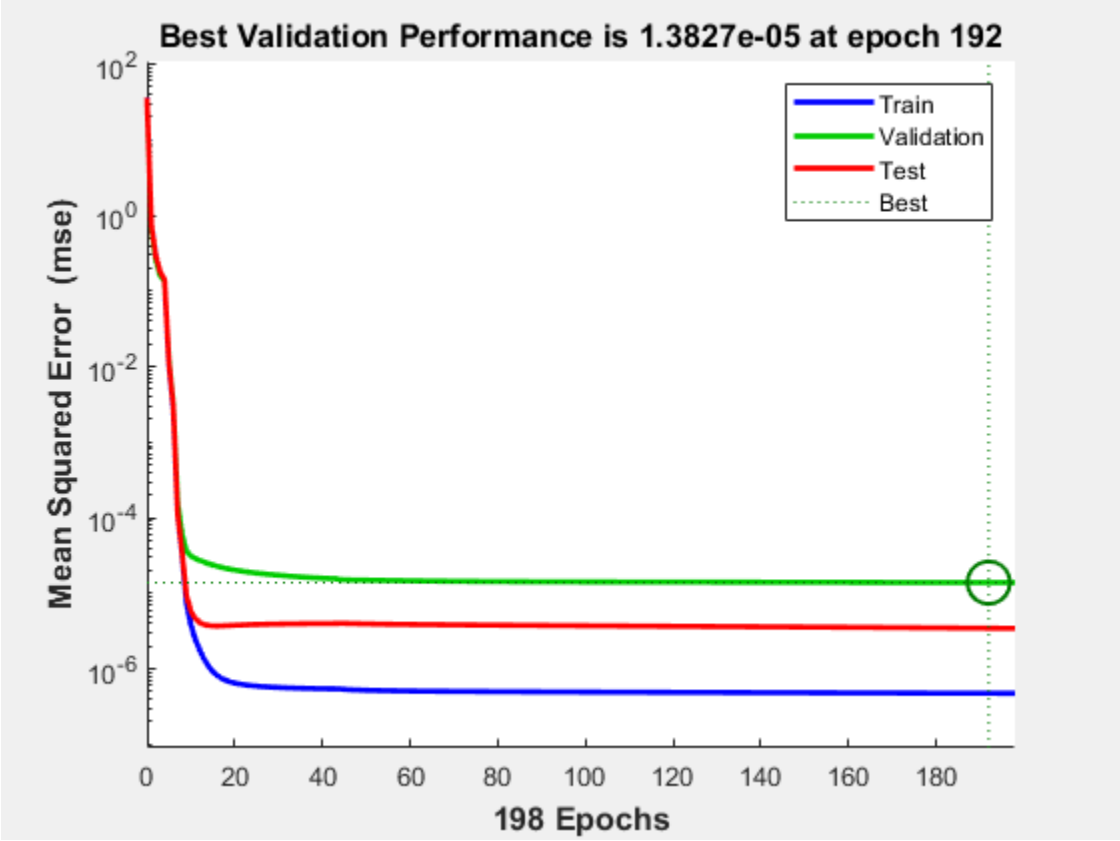

Neural Network Training (nntraintool)
Neural Network
x(t)
1
y(t)
1
Hidden
1:2
W
1:2
W
b
10
Output
W
b
1
y(t)
1
Algorithms
Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX
Progress
Epoch: 0 198 iterations 1000
Time: 0:00:01
Performance: 36.3 4.75e-07 0.00
Gradient: 73.2 2.96e-05 1.00e-07
Mu: 0.00100 1.00e-07 1.00e+10
Validation Checks: 0 6 6
Plots
Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)
Time-Series Response (plotresponse)
Error Autocorrelation (ploterrcorr)
Input-Error Cross-correlation (plotinerrcorr)
Plot Interval: 1 epochs
Validation stop.
Stop Training
Cancel

To see how the network's performance improved during training, either click the "Performance" button in the training tool, or call PLOTPERFORM.

Performance is measured in terms of mean squared error, and is shown in a log scale. It rapidly decreased as the network was trained.

Performance is shown for each of the training, validation and test sets.

```
plotperform(tr)
```



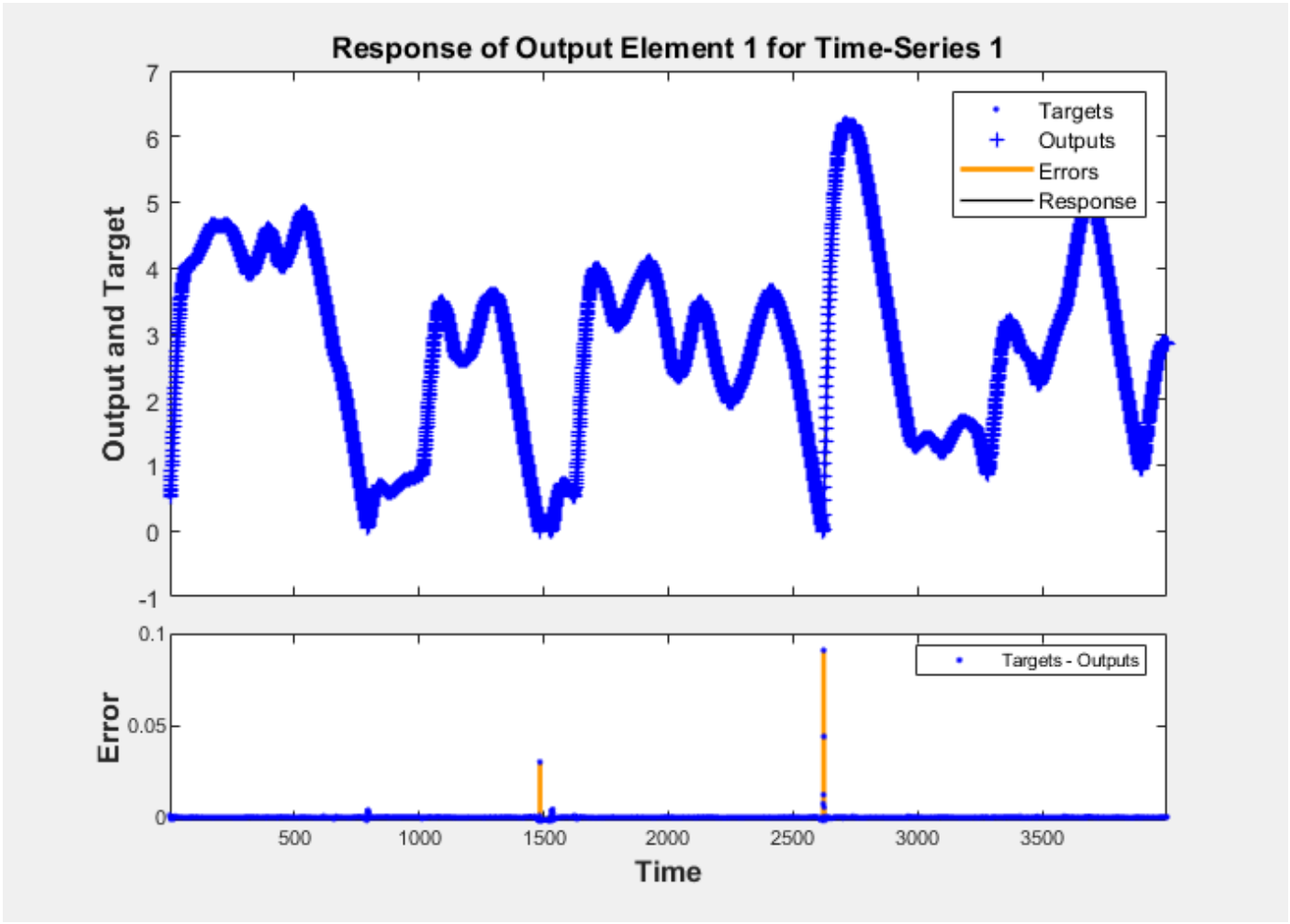
Testing the Neural Network

The mean squared error of the trained neural network for all timesteps can now be measured.

```
Y = net(Xs,Xi,Ai);

perf = mse(net,Ts,Y)

perf = 2.9245e-06
```

PLOTRESPONSE will show us the network's response in comparison to the actual magnet position. If the model is accurate the '+' points will track the diamond points, and the errors in the bottom axis will be very small.

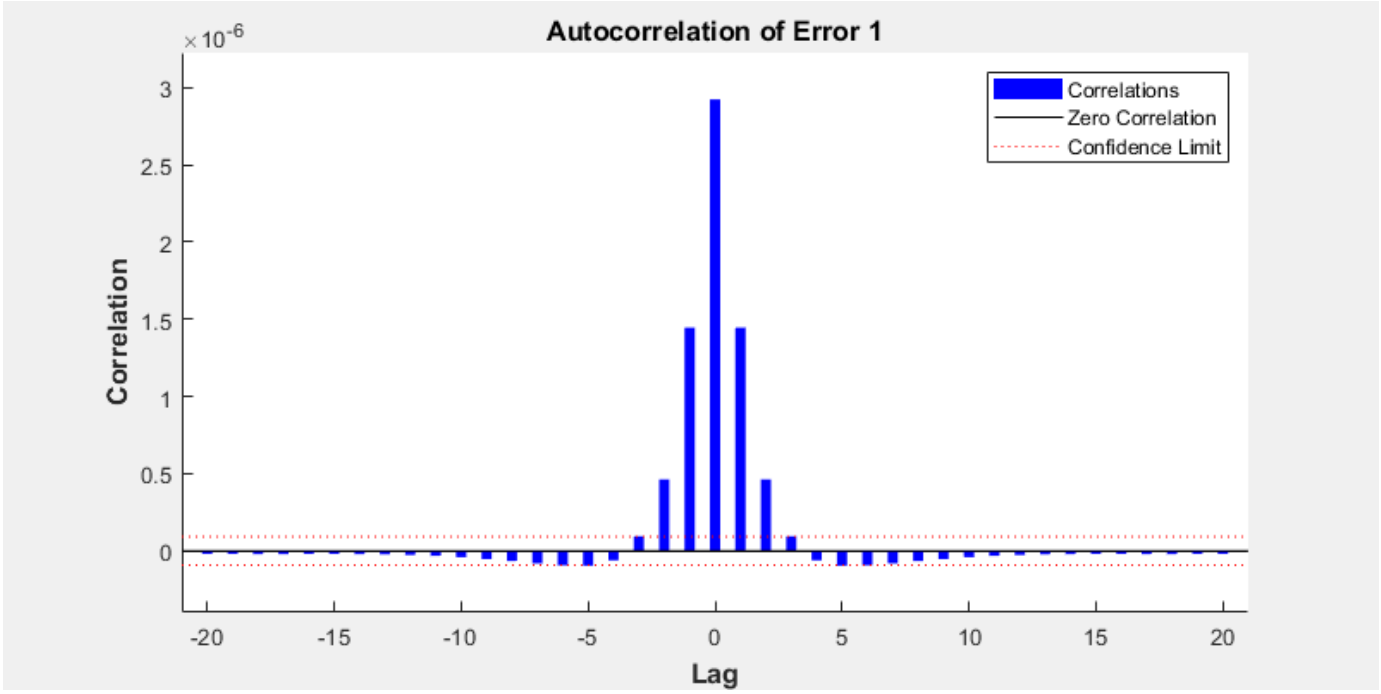
```
plotresponse(Ts,Y)
```



PLOTERRCORR shows the correlation of error at time t, e(t) with errors over varying lags, e(t+lag). The center line shows the mean squared error. If the network has been trained well all the other lines will be much shorter, and most if not all will fall within the red confidence limits.

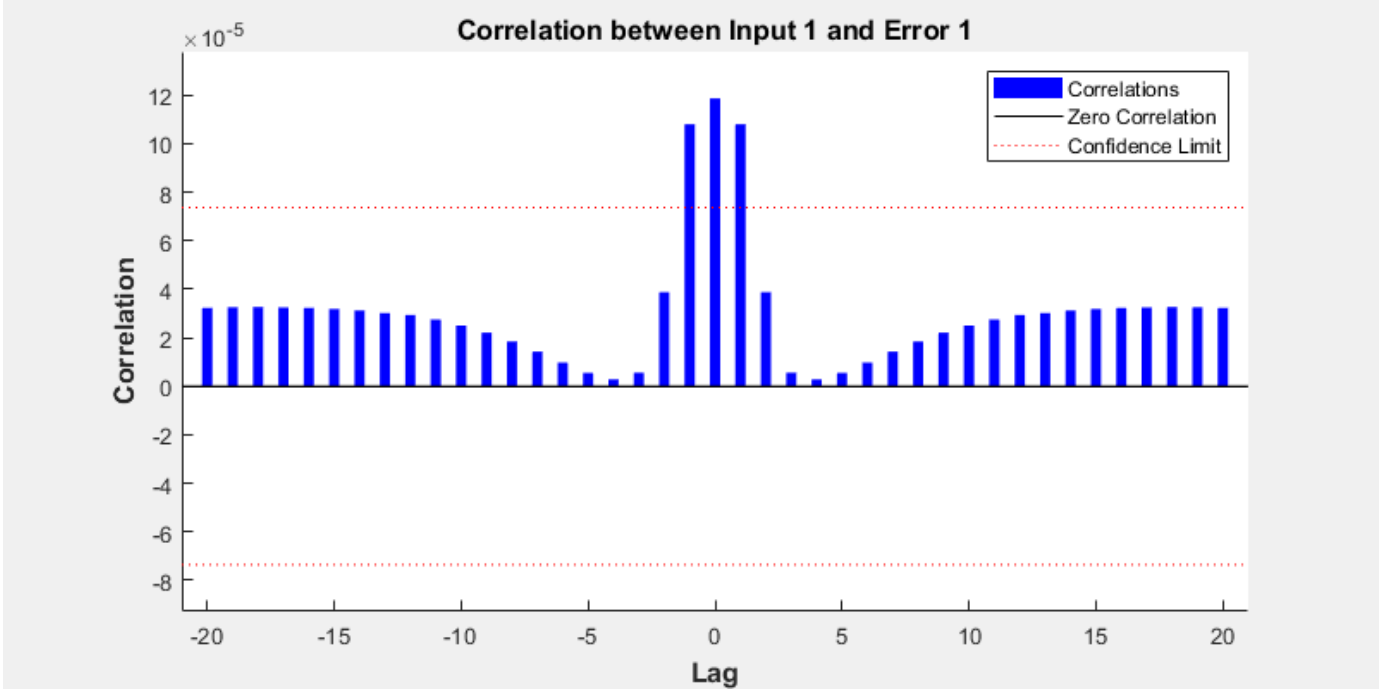
The function GSUBTRACT is used to calculate the error. This function generalizes subtraction to support differences between cell array data.

```
E = gsubtract(Ts,Y);

ploterrcorr(E)
```



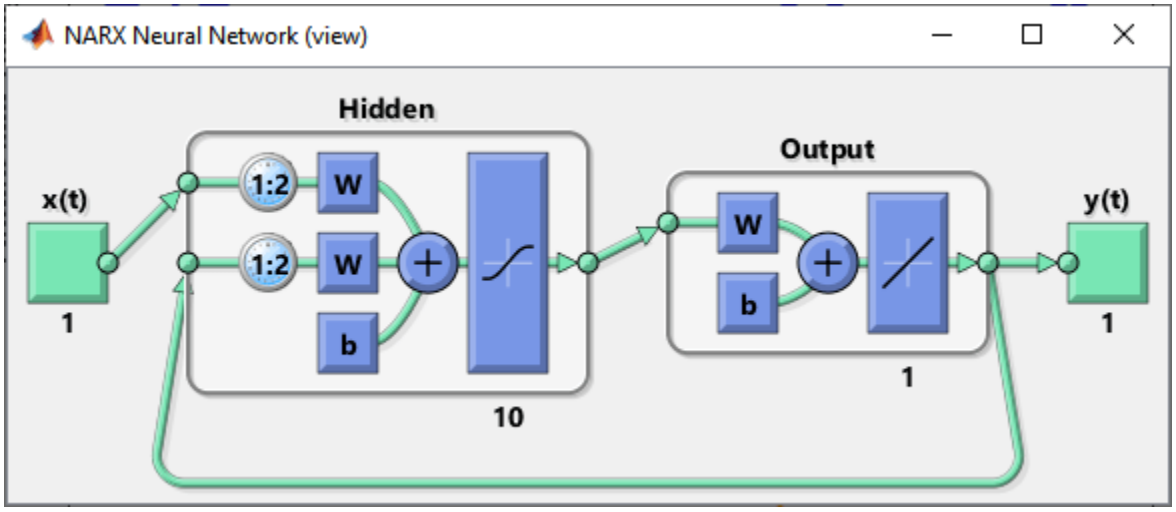
Similarly, PLOTINERRCORR shows the correlation of error with respect to the inputs, with varying degrees of lag. In this case, most or all the lines should fall within the confidence limits, including the center line.

```
plotinerrcorr(Xs,E)
```



The network was trained in open loop form, where targets were used as feedback inputs. The network can also be converted to closed loop form, where its own predictions become the feedback inputs.

```
net2 = closeloop(net);
view(net2)
```

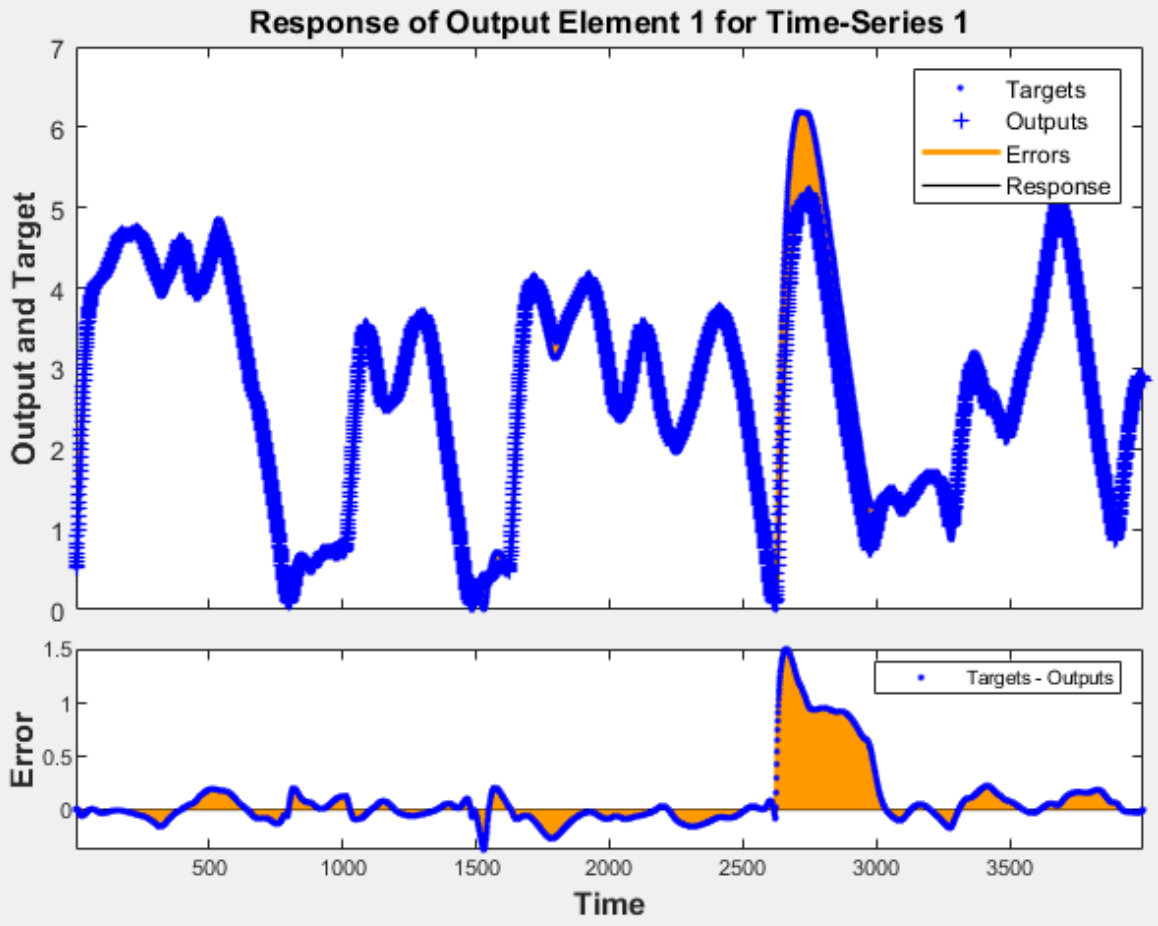


We can simulate the network in closed loop form. In this case the network is only given initial magnet positions, and then must use its own predicted positions recursively to predict new positions.

This quickly results in a poor fit between the predicted and actual response. This will occur even if the model is very good. But it is interesting to see how many steps they match before separating.

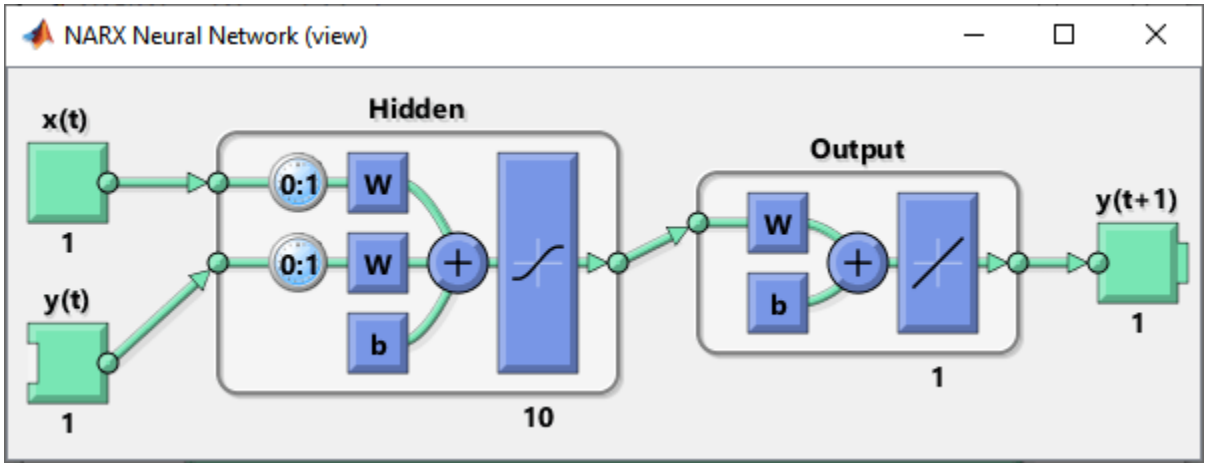
Again, PREPARETS does the work of preparing the time series data for us taking into account the altered network.

```
[Xs,Xi,Ai,Ts] = preparets(net2,x,{},t);
Y = net2(Xs,Xi,Ai);
plotresponse(Ts,Y)
```



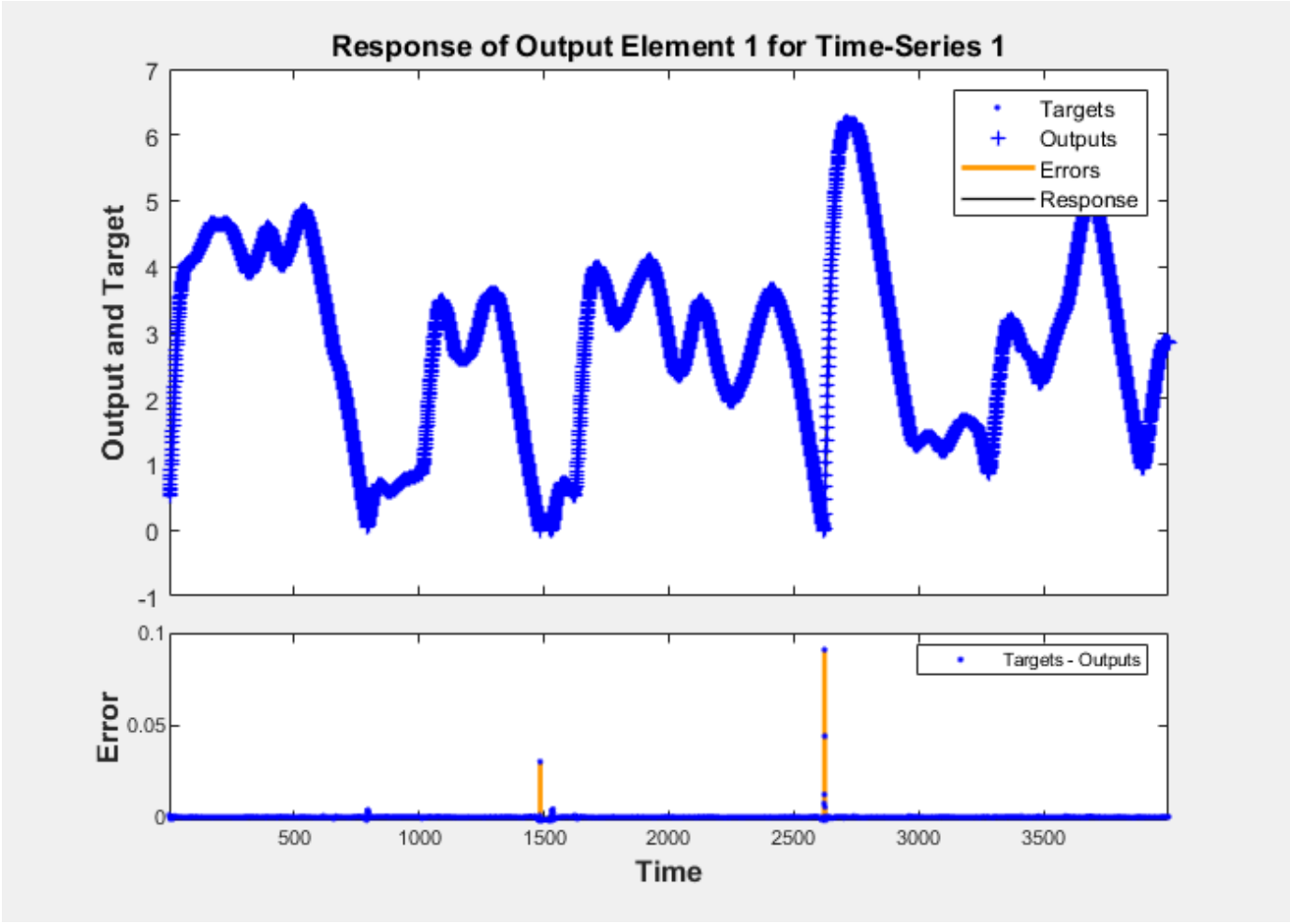
If the application required us to access the predicted magnet position a timestep ahead of when it actually occurs, we can remove a delay from the network so at any given time t, the output is an estimate of the position at time t+1.

```
net3 = removedelay(net);
view(net3)
```



Again we use PREPARETS to prepare the time series for simulation. This time the network is again very accurate as it is doing open loop prediction, but the output is shifted one timestep.

```
[Xs,Xi,Ai,Ts] = preparets(net3,x,{},t);
Y = net3(Xs,Xi,Ai);
plotresponse(Ts,Y)
```



This example illustrated how to design a neural network that models the behavior of a dynamical magnet levitation system.

Explore other examples and the documentation for more insight into neural networks and their applications.

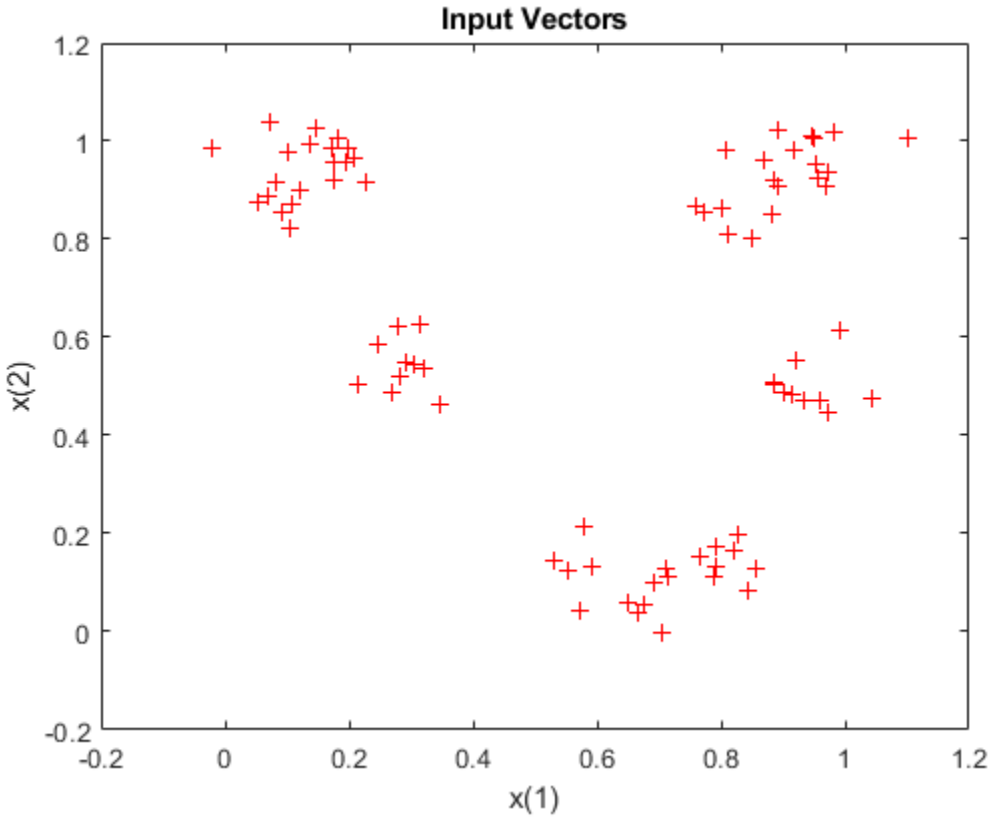
Competitive Learning

Neurons in a competitive layer learn to represent different regions of the input space where input vectors occur.

P is a set of randomly generated but clustered test data points. Here the data points are plotted.

A competitive network will be used to classify these points into natural classes.

```
% Create inputs X.
bounds = [0 1; 0 1];   % Cluster centers to be in these bounds.
clusters = 8;          % This many clusters.
points = 10;           % Number of points in each cluster.
std_dev = 0.05;        % Standard deviation of each cluster.
x = nngenc(bounds,clusters,points,std_dev);

% Plot inputs X.
plot(x(1,:),x(2,:),'+r');
title('Input Vectors');
xlabel('x(1)');
ylabel('x(2)');
```

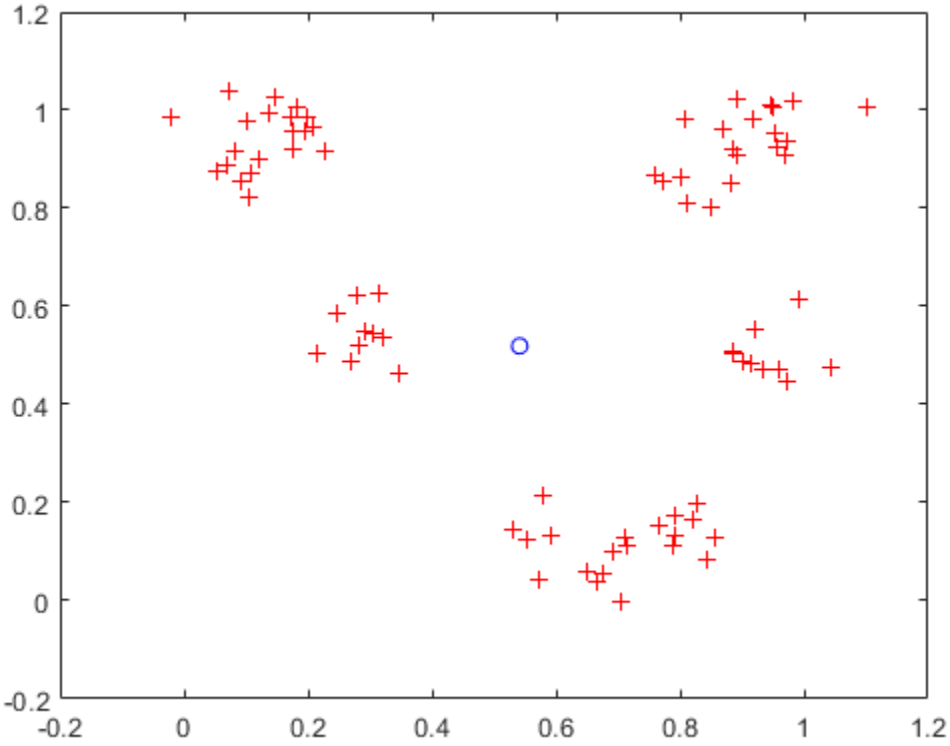


Here COMPETLAYER takes two arguments, the number of neurons and the learning rate.

We can configure the network inputs (normally done automatically by TRAIN) and plot the initial weight vectors to see their attempt at classification.

The weight vectors (o's) will be trained so that they occur centered in clusters of input vectors (+'s).

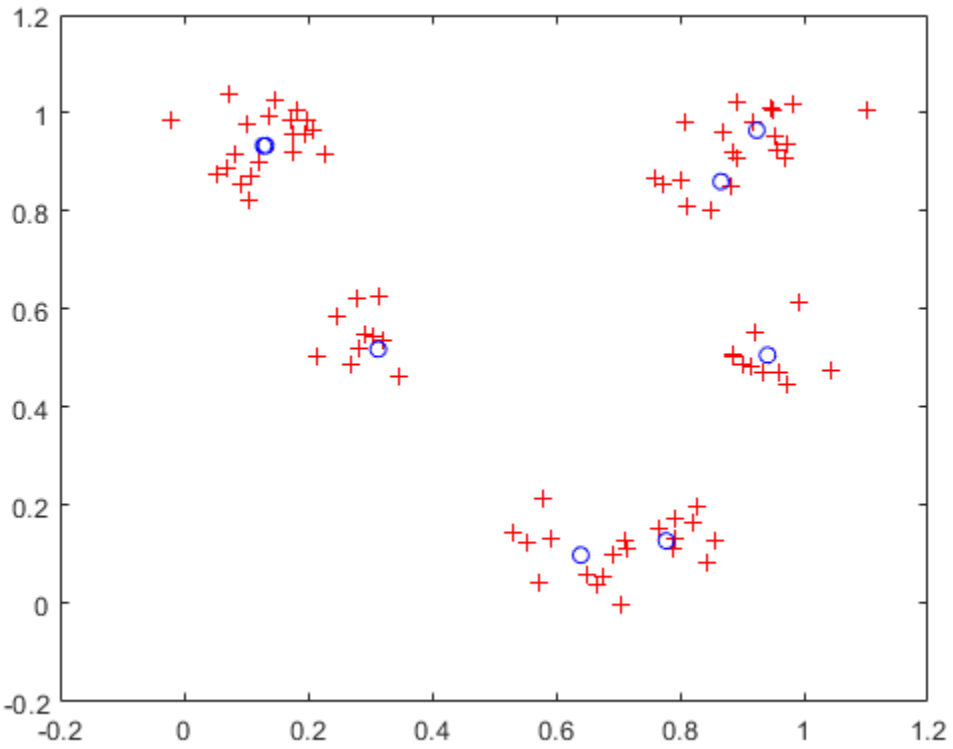
```
net = competlayer(8,.1);
net = configure(net,x);
w = net.IW{1};
plot(x(1,:),x(2,:),'+r');
hold on;
circles = plot(w(:,1),w(:,2),'ob');
```



Set the number of epochs to train before stopping and train this competitive layer (may take several seconds).

Plot the updated layer weights on the same graph.

```
net.trainParam.epochs = 7;
net = train(net,x);
w = net.IW{1};
delete(circles);
plot(w(:,1),w(:,2),'ob');
```



Now we can use the competitive layer as a classifier, where each neuron corresponds to a different category. Here we define am input vector X1 as [0; 0.2].

The output Y, indicates which neuron is responding, and thereby which class the input belongs.

```
x1 = [0; 0.2];
y = net(x1)
```

```
y = 8×1

     0
     1
     0
     0
     0
     0
     0
     0
```

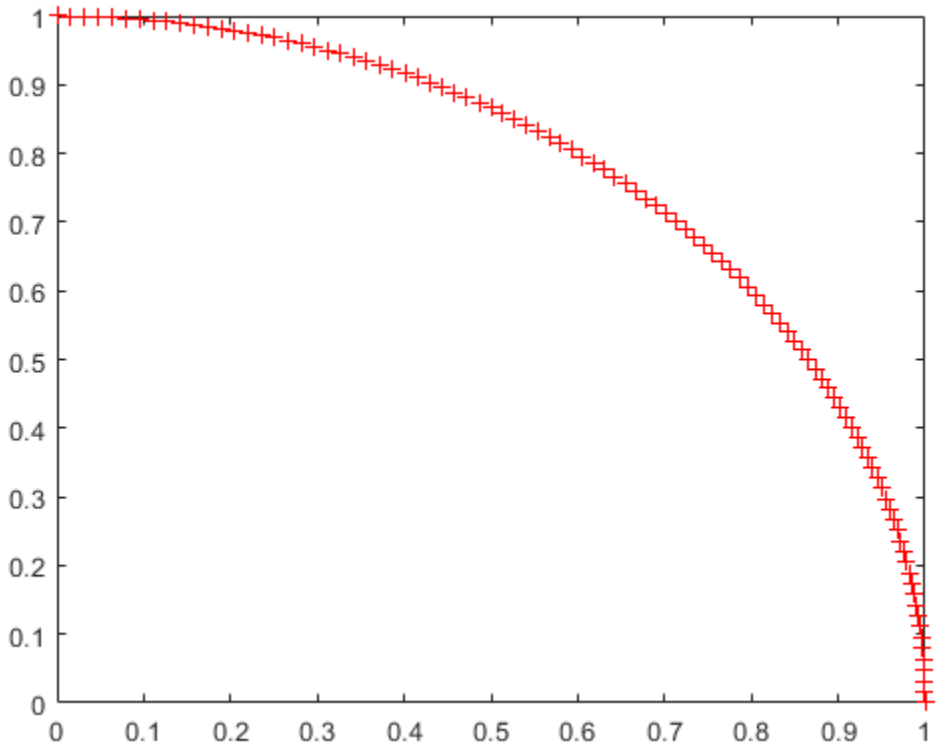
One-Dimensional Self-organizing Map

Neurons in a 2-D layer learn to represent different regions of the input space where input vectors occur. In addition, neighboring neurons learn to respond to similar inputs, thus the layer learns the topology of the presented input space.

Here 100 data points are created on the unit circle.

A competitive network will be used to classify these points into natural classes.

```
angles = 0:0.5*pi/99:0.5*pi;
X = [sin(angles); cos(angles)];
plot(X(1,:),X(2,:),'+r')
```



The map will be a 1-dimensional layer of 10 neurons.

```
net = selforgmap(10);
```

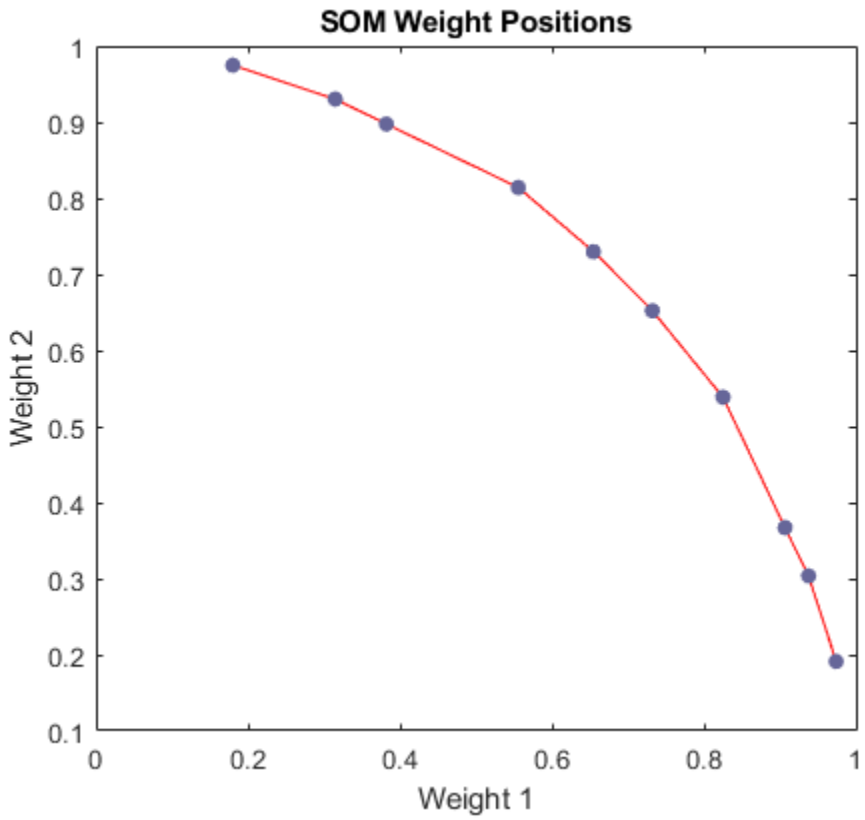
Specify that the network is to be trained for 10 epochs and use `train` to train the network on the input data.

```
net.trainParam.epochs = 10;
net = train(net,X);
```

Now plot the trained network's weight positions by using `plotsompos`.

The red dots are the neuron's weight vectors, and the blue lines connect each pair within a distance of 1.

```
plotsompos(net)
```



The map can now be used to classify inputs, such as [1; 0]. Either neuron 1 or 10 should have an output of 1, as the above input vector was at one end of the presented input space. The first pair of numbers indicate the neuron, and the single number indicates its output.

```
x = [1;0];
a = net(x)
```

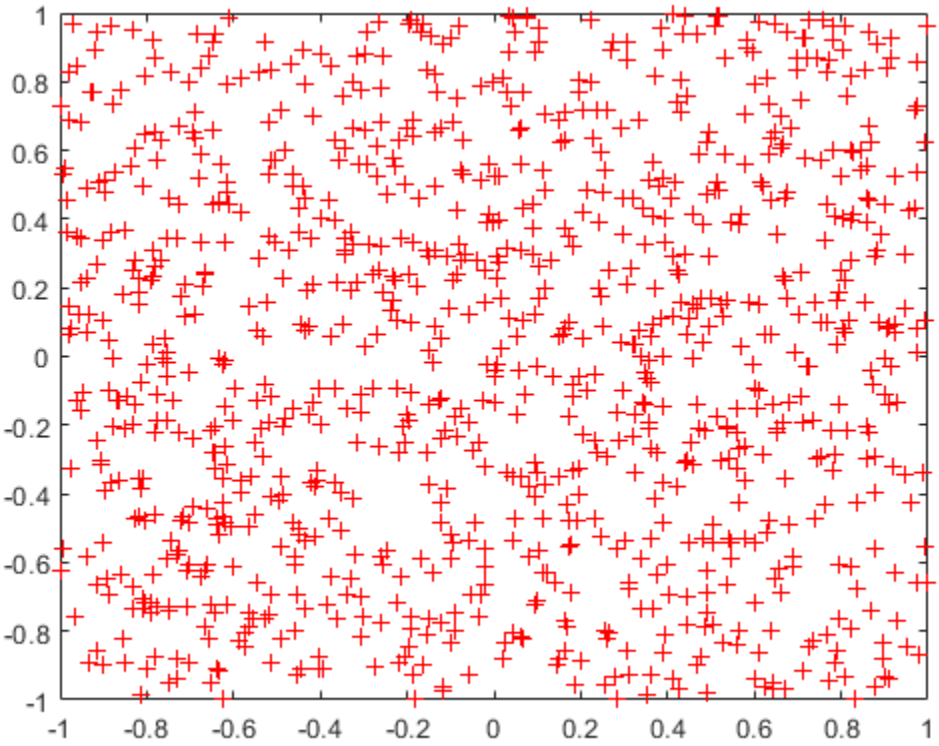
```
a = 10×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     1
```

Two-Dimensional Self-organizing Map

As in one-dimensional problems, this self-organizing map will learn to represent different regions of the input space where input vectors occur. In this example, however, the neurons will arrange themselves in a two-dimensional grid, rather than a line.

We would like to classify 1000 two-element vectors in a rectangle.

```
X = rands(2,1000);
plot(X(1,:),X(2,:),'+r')
```



We will use a 5-by-6 layer of neurons to classify the vectors above. We would like each neuron to respond to a different region of the rectangle, and neighboring neurons to respond to adjacent regions.

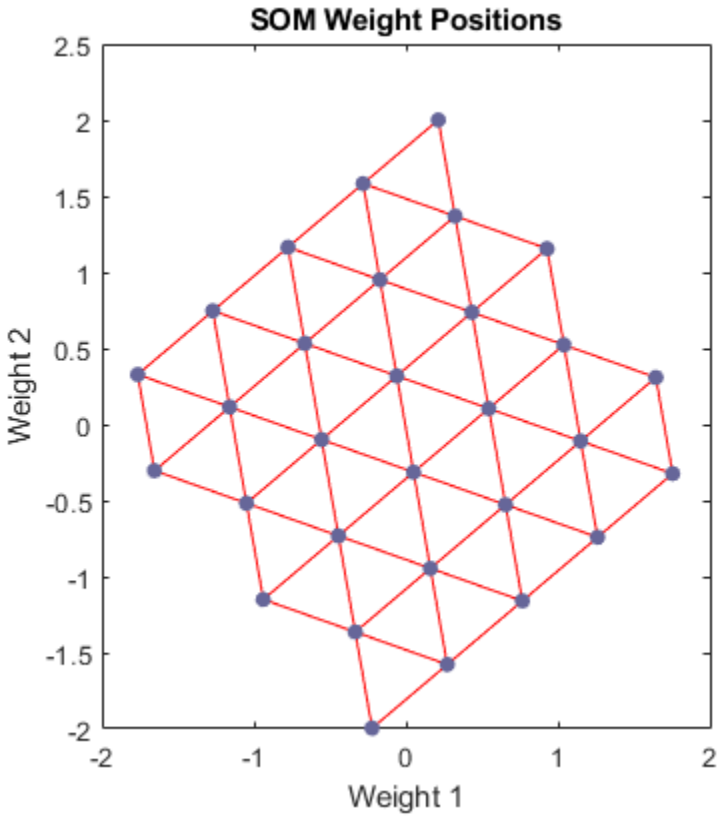
The network is configured to match the dimensions of the inputs. This step is required here because we will plot the initial weights. Normally configuration is performed automatically when training.

```
net = selforgmap([5 6]);
net = configure(net,X);
```

We can visualize the network we have just created by using `plotsompos`.

Each neuron is represented by a red dot at the location of its two weights. Initially, all the neurons have the same weights in the middle of the vectors, so only one dot appears.

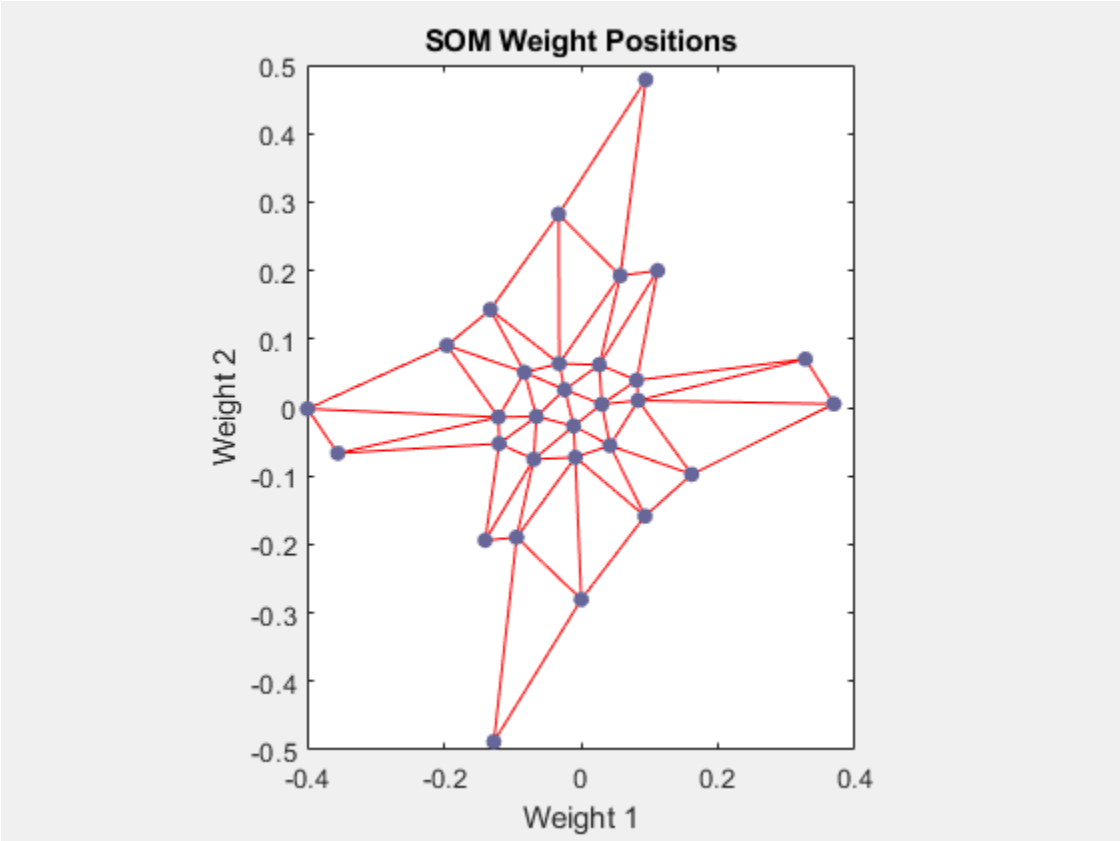
```
plotsompos(net)
```



Now we train the map on the 1000 vectors for 1 epoch and replot the network weights.

After training, note that the layer of neurons has begun to self-organize so that each neuron now classifies a different region of the input space, and adjacent (connected) neurons respond to adjacent regions.

```
net.trainParam.epochs = 1;
net = train(net,X);
plotsompos(net)
```



We can now classify vectors by giving them to the network and seeing which neuron responds.

The neuron indicated by a "1" responded, so x belongs to that class.

```
x = [0.5;0.3];
y = net(x)
```

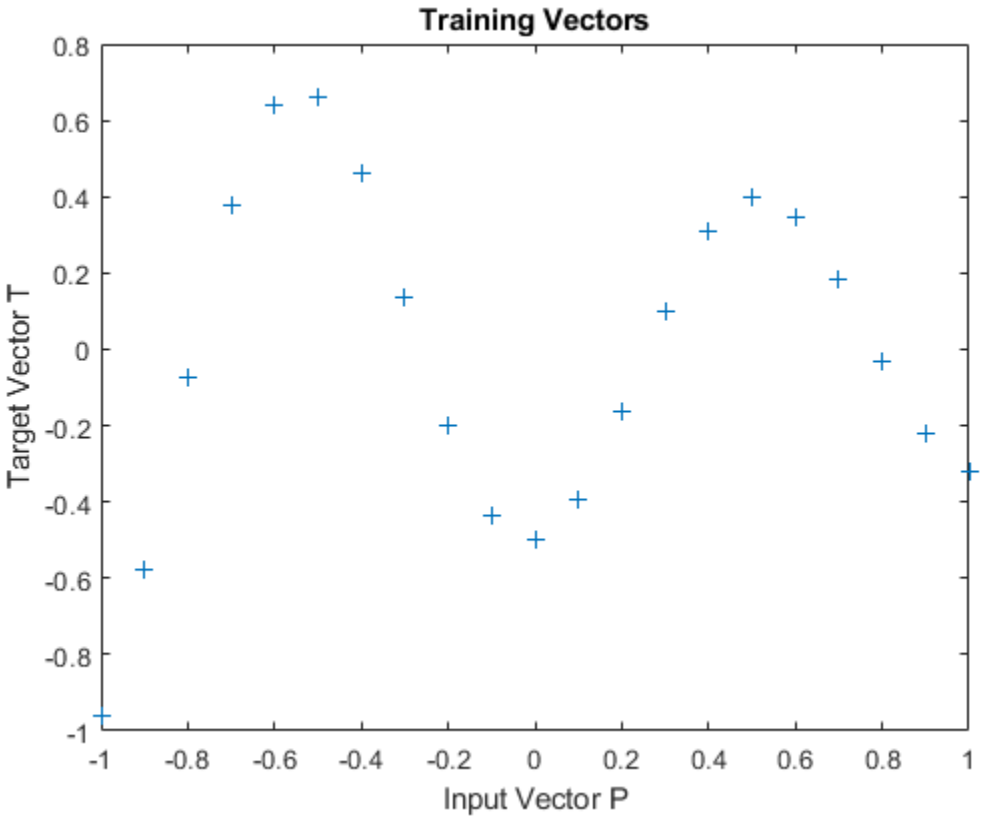
```
y = 30×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
  ⋮
```

Radial Basis Approximation

This example uses the NEWRB function to create a radial basis network that approximates a function defined by a set of data points.

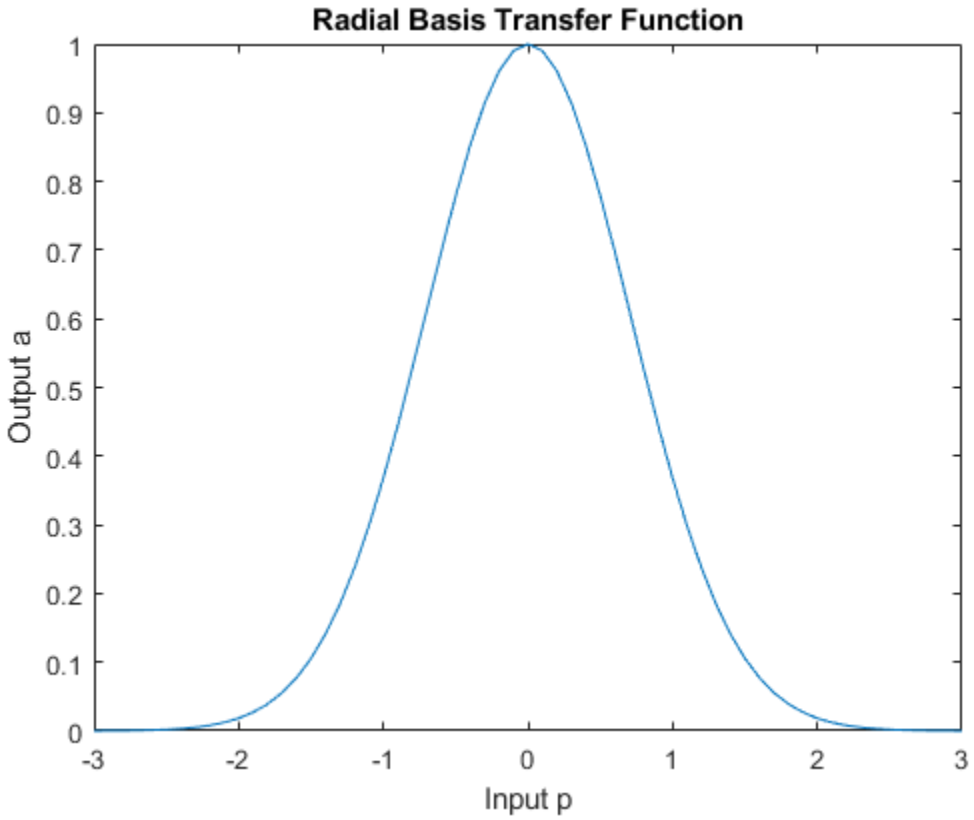
Define 21 inputs P and associated targets T.

```
X = -1:.1:1;
T = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
      .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
      .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];
plot(X,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
```



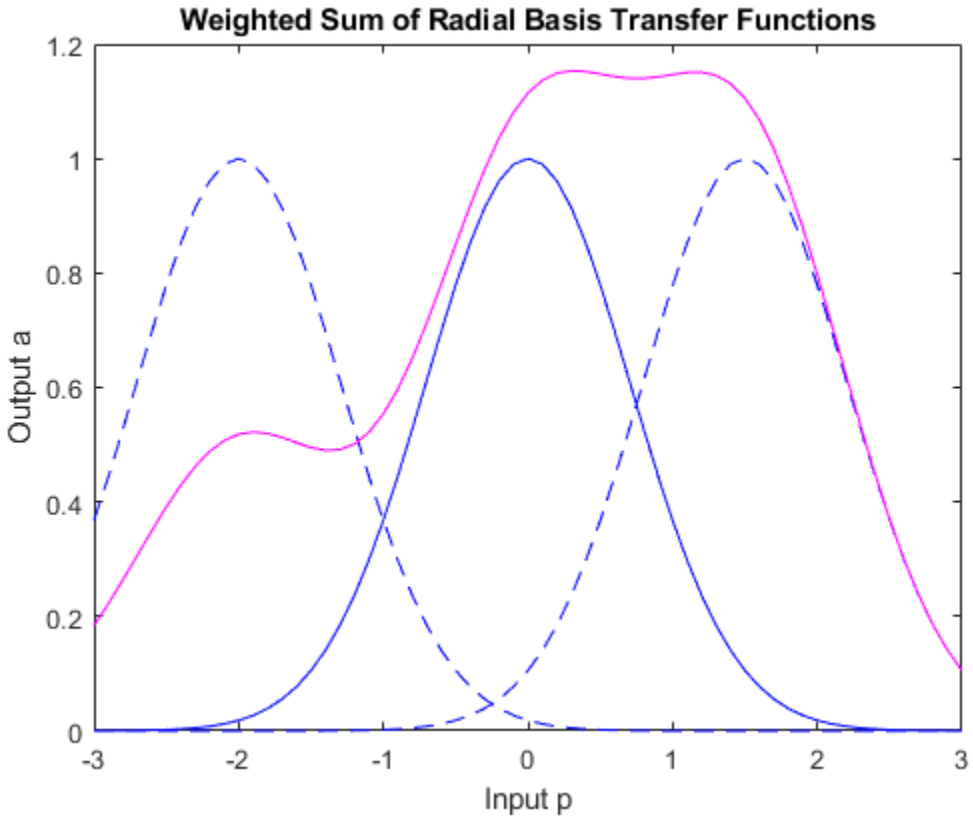
We would like to find a function which fits the 21 data points. One way to do this is with a radial basis network. A radial basis network is a network with two layers. A hidden layer of radial basis neurons and an output layer of linear neurons. Here is the radial basis transfer function used by the hidden layer.

```
x = -3:.1:3;
a = radbas(x);
plot(x,a)
title('Radial Basis Transfer Function');
xlabel('Input p');
ylabel('Output a');
```



The weights and biases of each neuron in the hidden layer define the position and width of a radial basis function. Each linear output neuron forms a weighted sum of these radial basis functions. With the correct weight and bias values for each layer, and enough hidden neurons, a radial basis network can fit any function with any desired accuracy. This is an example of three radial basis functions (in blue) are scaled and summed to produce a function (in magenta).

```
a2 = radbas(x-1.5);
a3 = radbas(x+2);
a4 = a + a2*1 + a3*0.5;
plot(x,a,'b-',x,a2,'b--',x,a3,'b--',x,a4,'m-')
title('Weighted Sum of Radial Basis Transfer Functions');
xlabel('Input p');
ylabel('Output a');
```



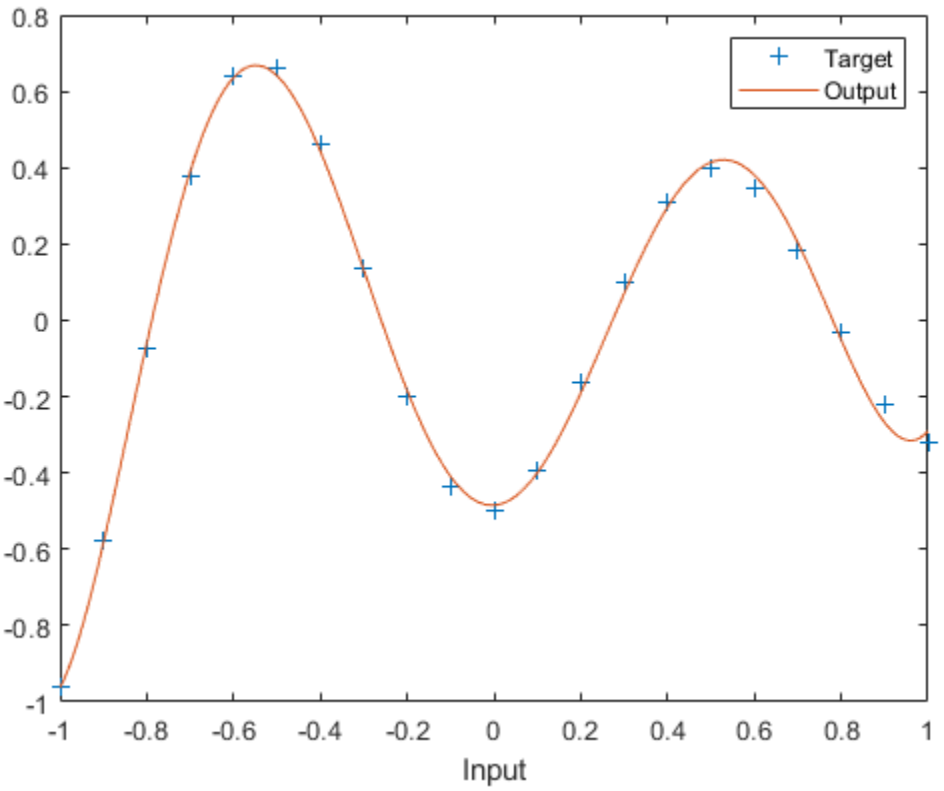
The function NEWRB quickly creates a radial basis network which approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant.

```
eg = 0.02; % sum-squared error goal
sc = 1;    % spread constant
net = newrb(X,T,eg,sc);
```

```
NEWRB, neurons = 0, MSE = 0.176192
```

To see how the network performs, replot the training set. Then simulate the network response for inputs over the same range. Finally, plot the results on the same graph.

```
plot(X,T,'+');
xlabel('Input');

X = -1:.01:1;
Y = net(X);

hold on;
plot(X,Y);
hold off;
legend({'Target','Output'})
```

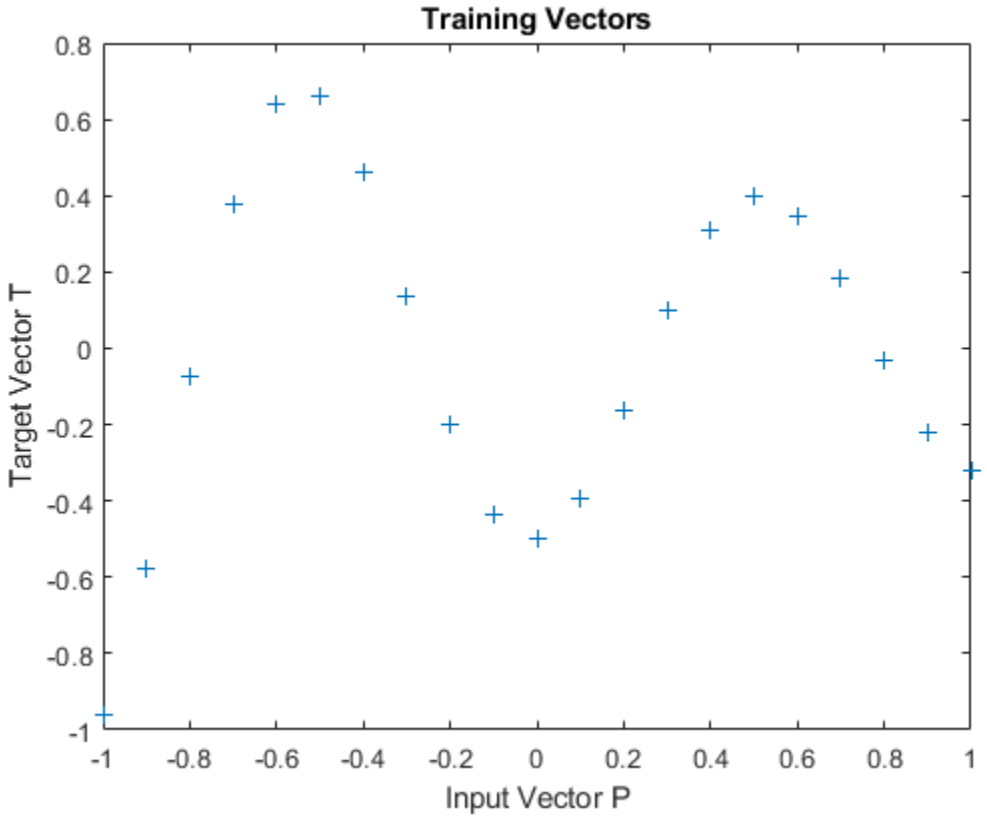

Target
Output
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
Input

Radial Basis Underlapping Neurons

A radial basis network is trained to respond to specific inputs with target outputs. However, because the spread of the radial basis neurons is too low, the network requires many neurons.

Define 21 inputs P and associated targets T.

```
P = -1:.1:1;
T = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
      .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
      .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];
plot(P,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
```



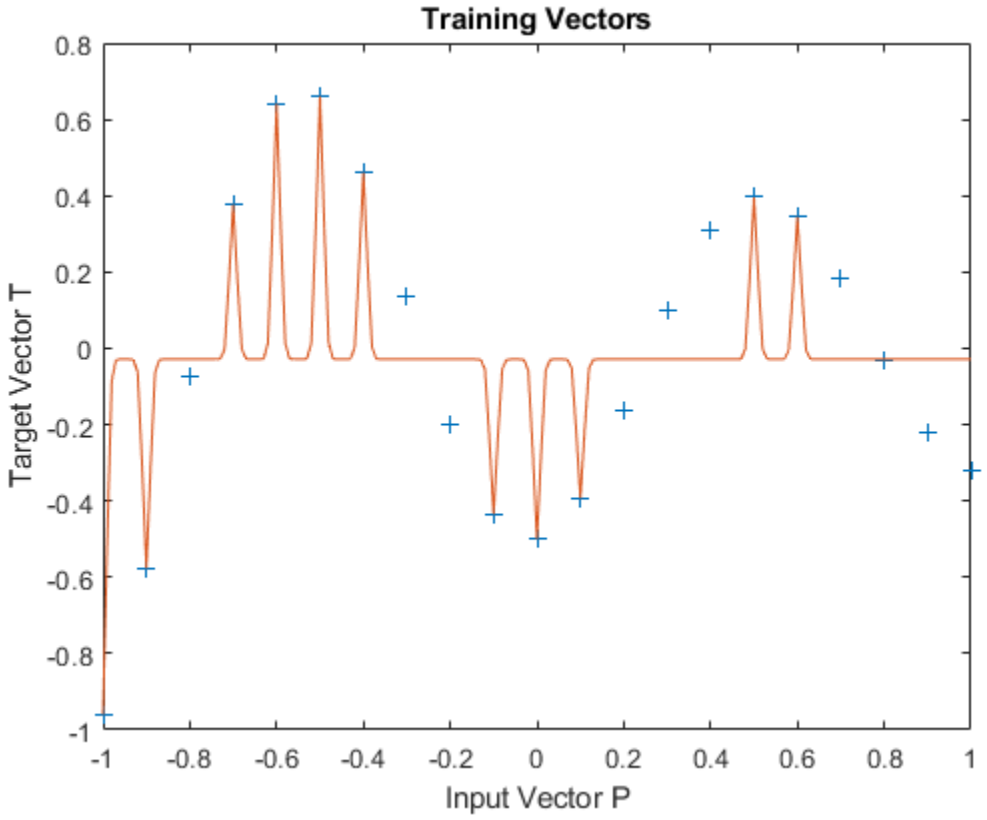
The function NEWRB quickly creates a radial basis network which approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant. The spread of the radial basis neurons B is set to a very small number.

```
eg = 0.02; % sum-squared error goal
sc = .01;  % spread constant
net = newrb(P,T,eg,sc);
```

```
NEWRB, neurons = 0, MSE = 0.176192
```

To check that the network fits the function in a smooth way, define another set of test input vectors and simulate the network with these new inputs. Plot the results on the same graph as the training set. The test vectors reveal that the function has been overfit! The network could have done better with a higher spread constant.

```
X = -1:.01:1;
Y = net(X);
hold on;
plot(X,Y);
hold off;
```

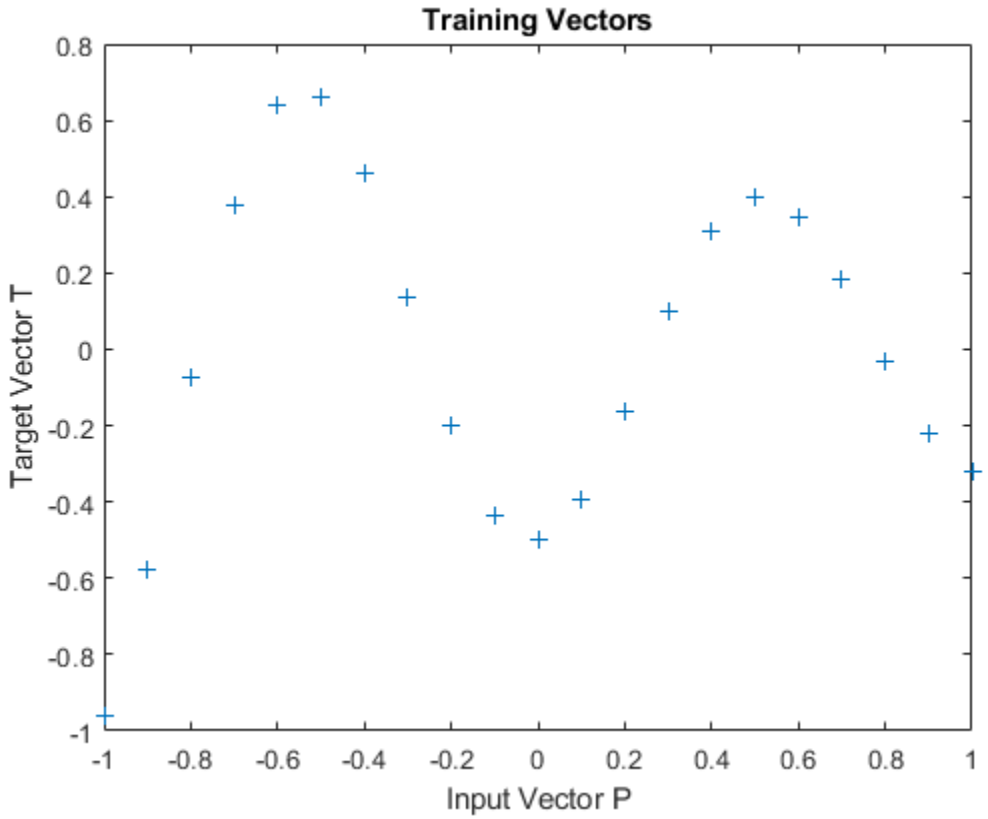


Radial Basis Overlapping Neurons

A radial basis network is trained to respond to specific inputs with target outputs. However, because the spread of the radial basis neurons is too high, each neuron responds essentially the same, and the network cannot be designed.

Define 21 inputs P and associated targets T.

```
P = -1:.1:1;
T = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
      .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
      .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];
plot(P,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
```



The function NEWRB quickly creates a radial basis network which approximates the function defined by P and T.

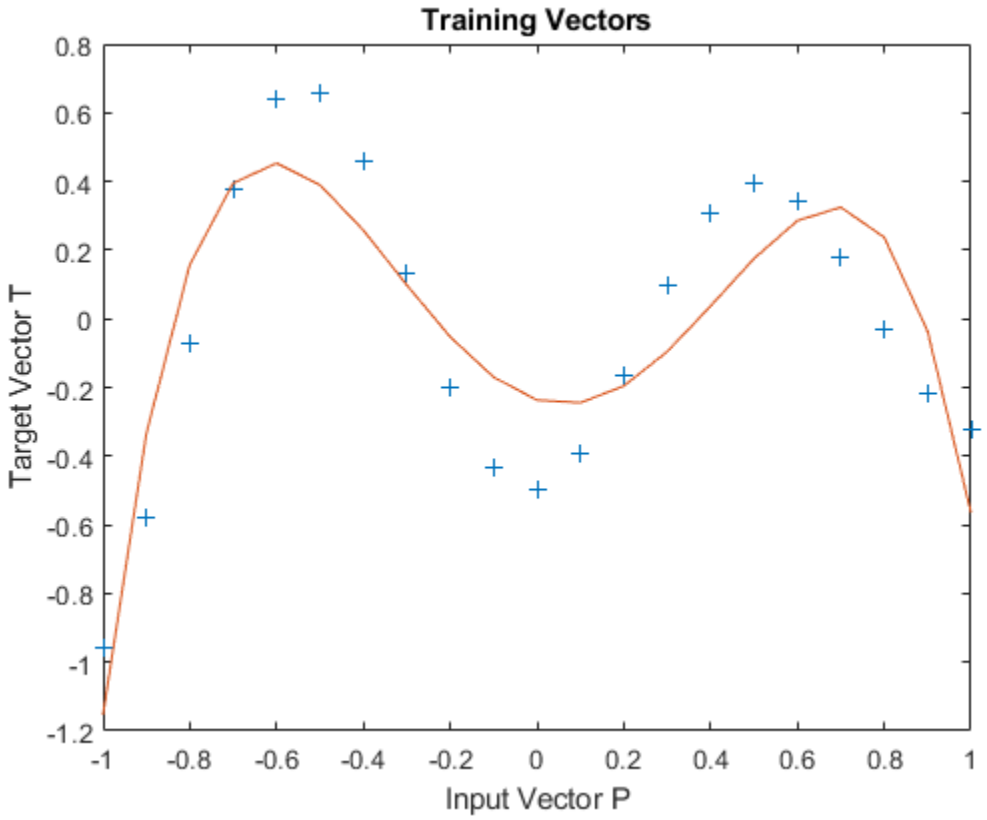
In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant. The spread of the radial basis neurons B is set to a very large number.

```
eg = 0.02; % sum-squared error goal
sc = 100;  % spread constant
net = newrb(P,T,eg,sc);
```

```
NEWRB, neurons = 0, MSE = 0.176192
```

NEWRB cannot properly design a radial basis network due to the large overlap of the input regions of the radial basis neurons. All the neurons always output 1, and so cannot be used to generate different responses. To see how the network performs with the training set, simulate the network with the original inputs. Plot the results on the same graph as the training set.

```
Y = net(P);
hold on;
plot(P,Y);
hold off;
```

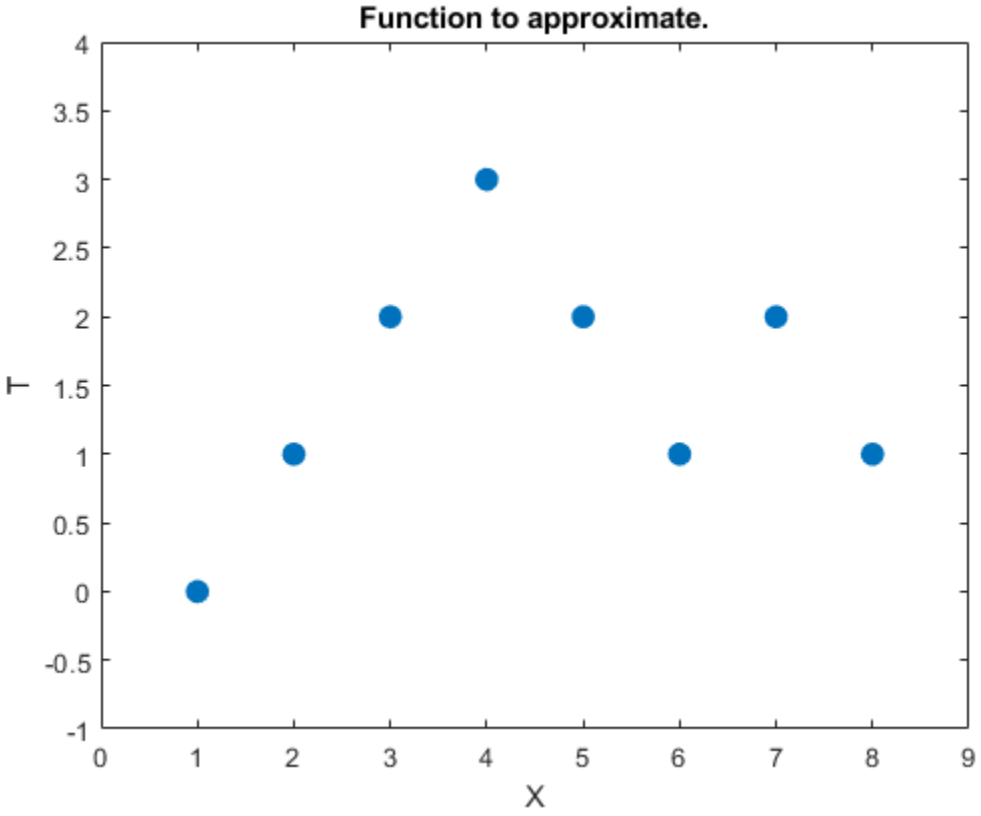


GRNN Function Approximation

This example uses functions NEWGRNN and SIM.

Here are eight data points of y function we would like to fit. The functions inputs X should result in target outputs T.

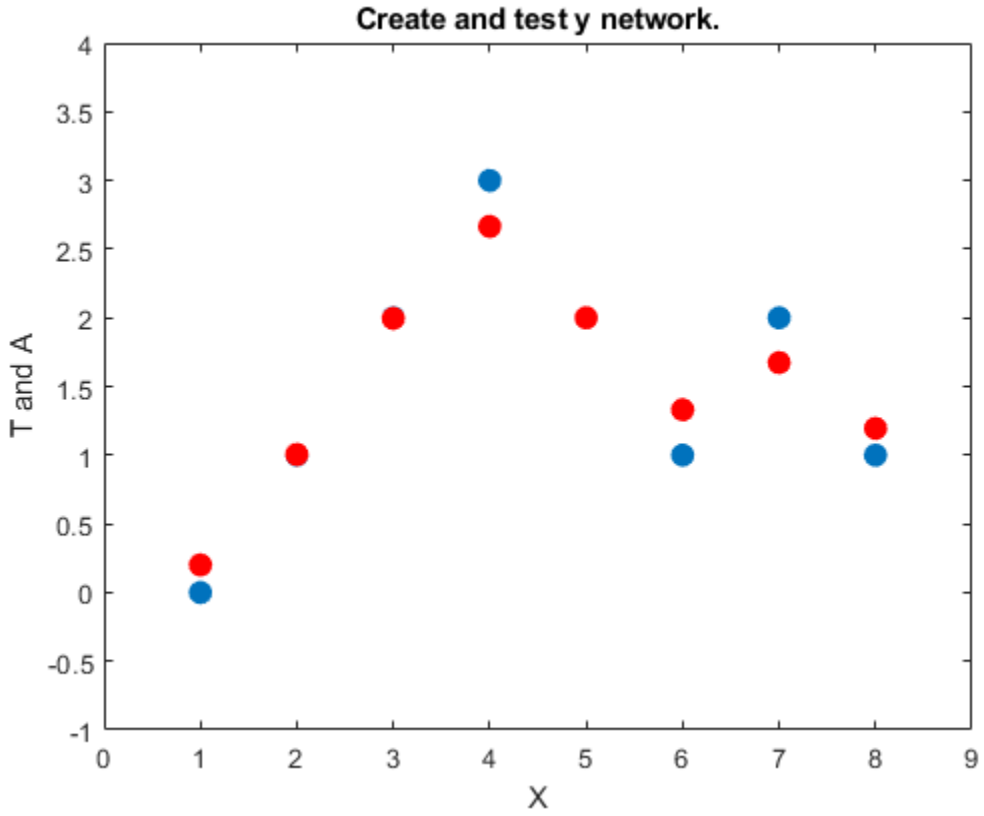
```
X = [1 2 3 4 5 6 7 8];
T = [0 1 2 3 2 1 2 1];

plot(X,T,'.','markersize',30)
axis([0 9 -1 4])
title('Function to approximate.')
xlabel('X')
ylabel('T')
```



We use NEWGRNN to create y generalized regression network. We use y SPREAD slightly lower than 1, the distance between input values, in order, to get y function that fits individual data points fairly closely. A smaller spread would fit data better but be less smooth.

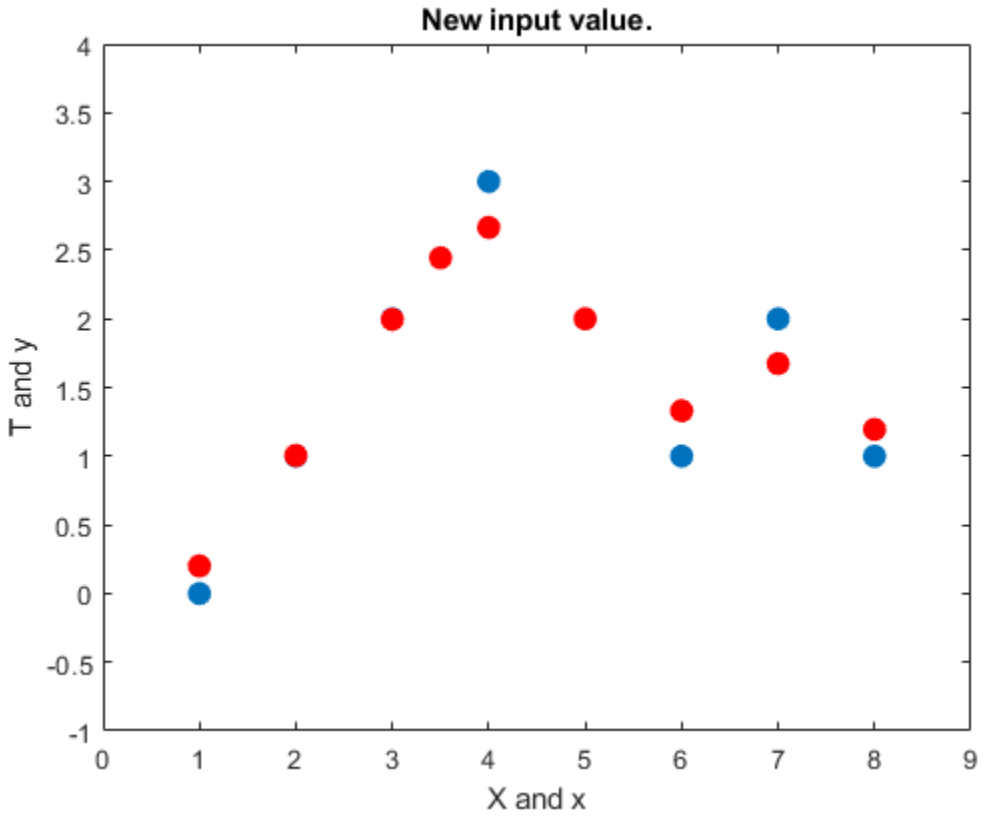
```
spread = 0.7;
net = newgrnn(X,T,spread);
A = net(X);

hold on
outputline = plot(X,A,'.','markersize',30,'color',[1 0 0]);
title('Create and test y network.')
```

```
xlabel('X')
ylabel('T and A')
```



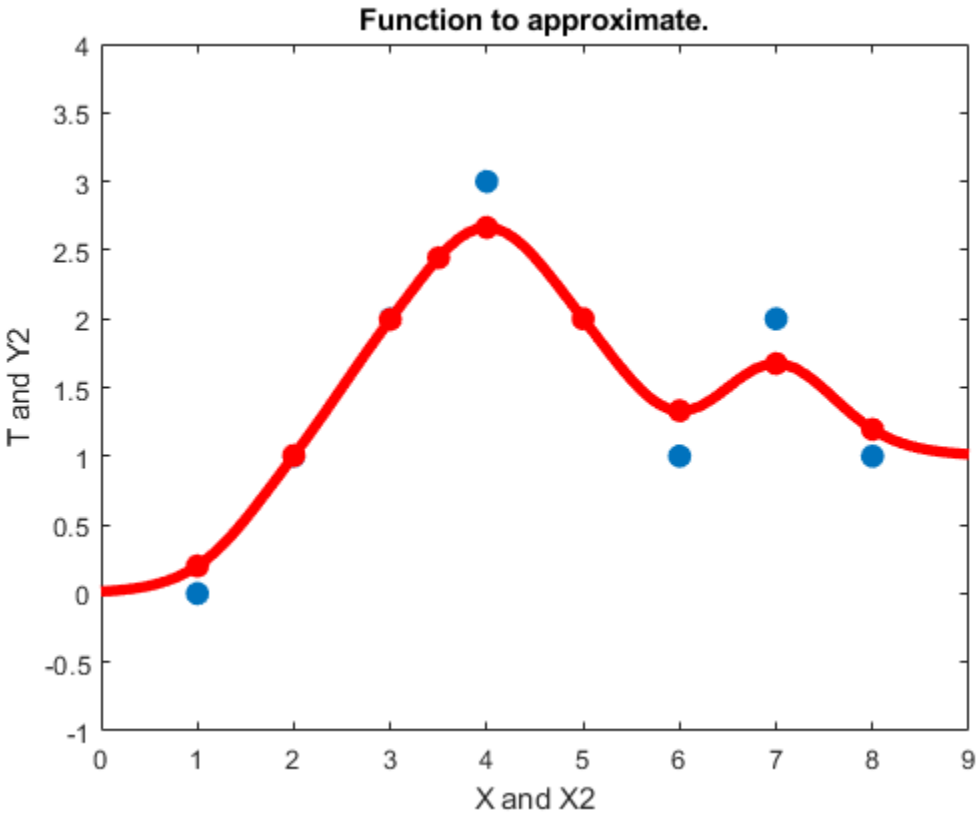
We can use the network to approximate the function at y new input value.

```
x = 3.5;
y = net(x);
plot(x,y,'.','markersize',30,'color',[1 0 0]);
title('New input value.')
xlabel('X and x')
ylabel('T and y')
```



Here the network's response is simulated for many values, allowing us to see the function it represents.

```
X2 = 0:.1:9;
Y2 = net(X2);
plot(X2,Y2,'linewidth',4,'color',[1 0 0])
title('Function to approximate.')
xlabel('X and X2')
ylabel('T and Y2')
```

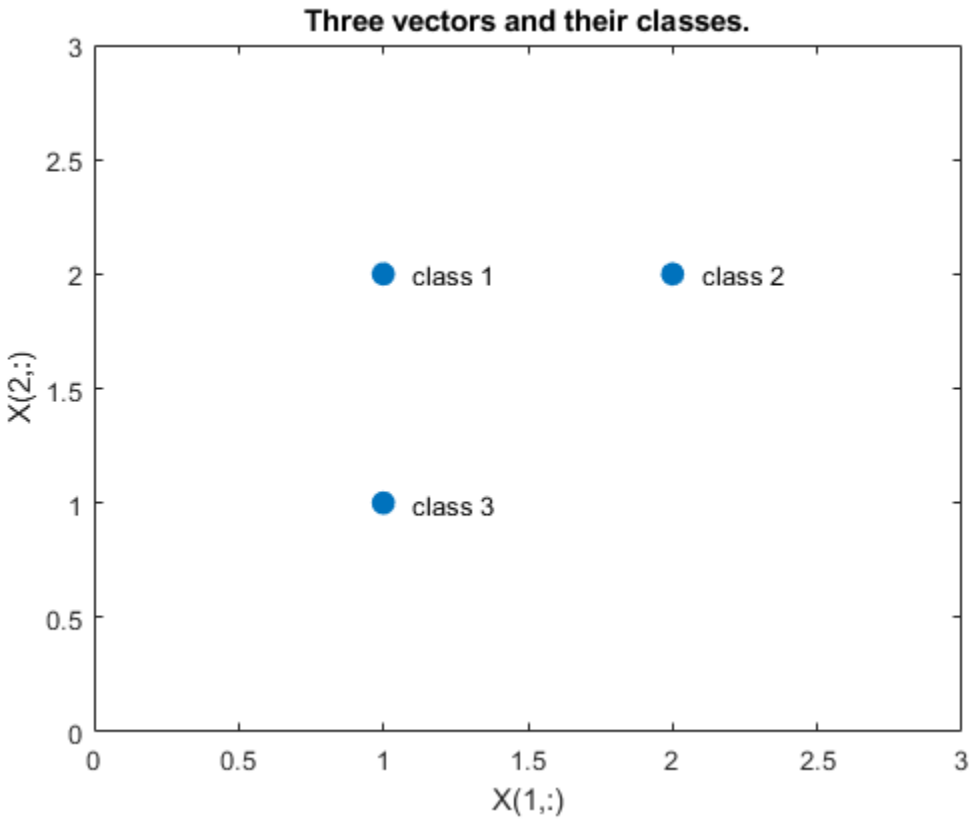

Function to approximate.
T and Y2
X and X2
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
7
8
9

PNN Classification

This example uses functions NEWPNN and SIM.

Here are three two-element input vectors X and their associated classes Tc. We would like to create y probabilistic neural network that classifies these vectors properly.

```
X = [1 2; 2 2; 1 1]';
Tc = [1 2 3];
plot(X(1,:),X(2,:),'.','markersize',30)
for i = 1:3, text(X(1,i)+0.1,X(2,i),sprintf('class %g',Tc(i))), end
axis([0 3 0 3])
title('Three vectors and their classes.')
xlabel('X(1,:)')
ylabel('X(2,:)')
```

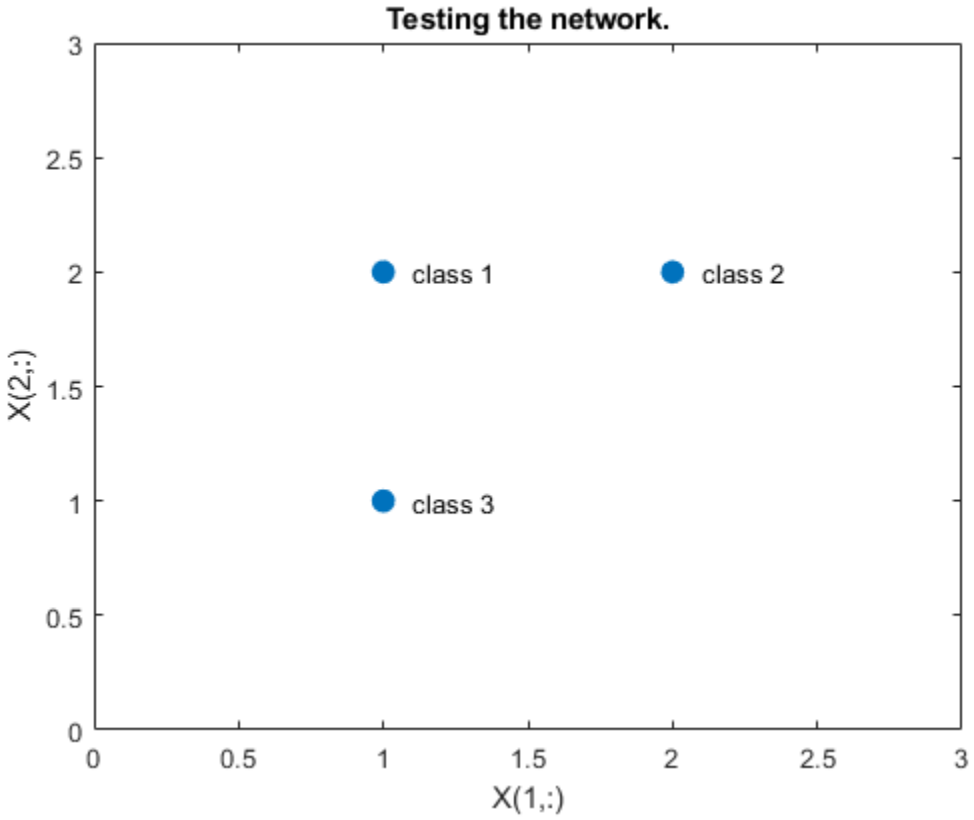


First we convert the target class indices Tc to vectors T. Then we design y probabilistic neural network with NEWPNN. We use y SPREAD value of 1 because that is y typical distance between the input vectors.

```
T = ind2vec(Tc);
spread = 1;
net = newpnn(X,T,spread);
```

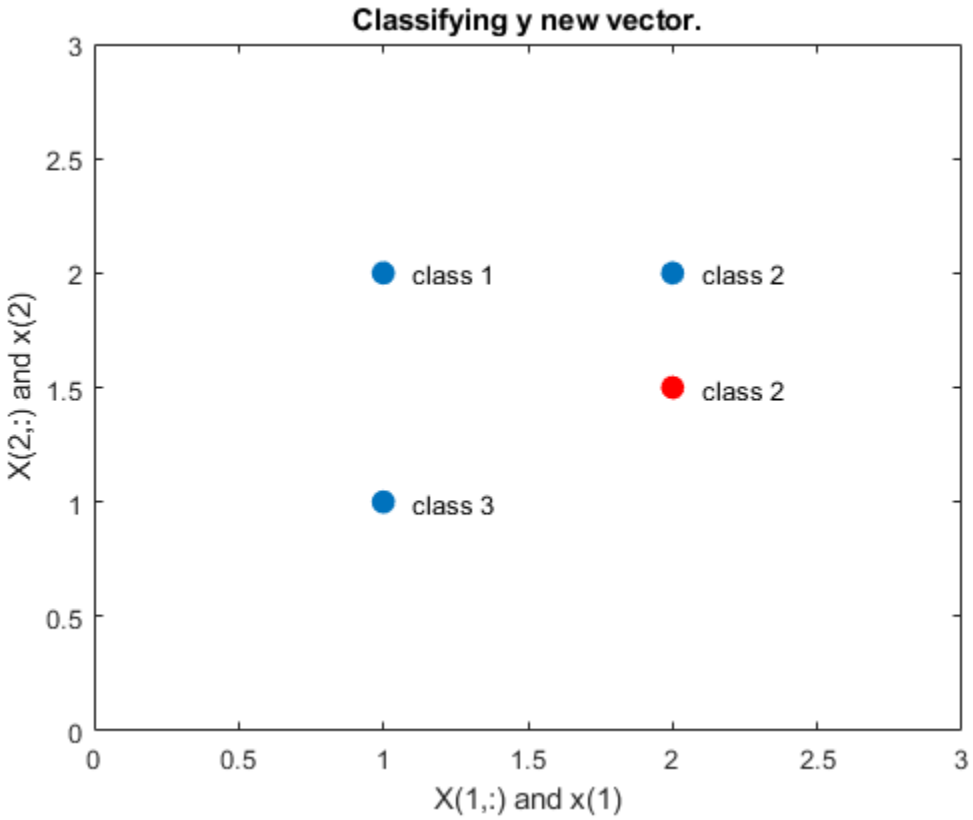
Now we test the network on the design input vectors. We do this by simulating the network and converting its vector outputs to indices.

```
Y = net(X);
Yc = vec2ind(Y);
plot(X(1,:),X(2,:),'.','markersize',30)
axis([0 3 0 3])
for i = 1:3,text(X(1,i)+0.1,X(2,i),sprintf('class %g',Yc(i))),end
title('Testing the network.')
xlabel('X(1,:)')
ylabel('X(2,:)')
```



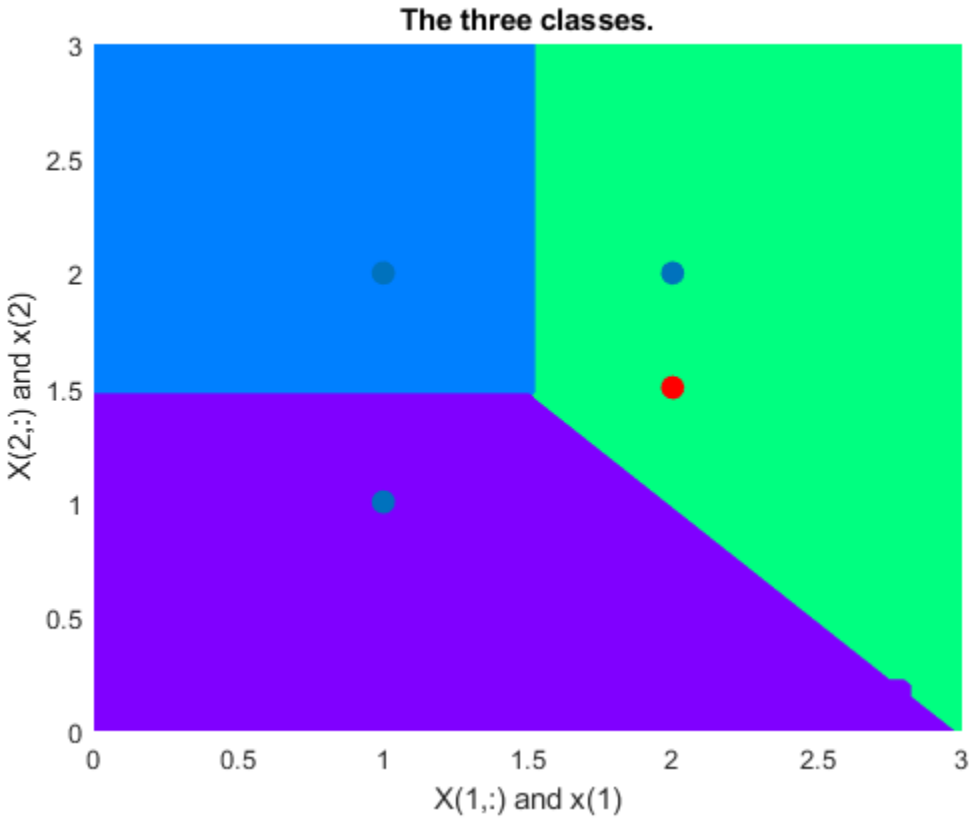
Let's classify y new vector with our network.

```
x = [2; 1.5];
y = net(x);
ac = vec2ind(y);
hold on
plot(x(1),x(2),'.','markersize',30,'color',[1 0 0])
text(x(1)+0.1,x(2),sprintf('class %g',ac))
hold off
title('Classifying y new vector.')
xlabel('X(1,:) and x(1)')
ylabel('X(2,:) and x(2)')
```



This diagram shows how the probabilistic neural network divides the input space into the three classes.

```
x1 = 0:.05:3;
x2 = x1;
[X1,X2] = meshgrid(x1,x2);
xx = [X1(:) X2(:)]';
yy = net(xx);
yy = full(yy);
m = mesh(X1,X2,reshape(yy(1,:),length(x1),length(x2)));
m.FaceColor = [0 0.5 1];
m.LineStyle = 'none';
hold on
m = mesh(X1,X2,reshape(yy(2,:),length(x1),length(x2)));
m.FaceColor = [0 1.0 0.5];
m.LineStyle = 'none';
m = mesh(X1,X2,reshape(yy(3,:),length(x1),length(x2)));
m.FaceColor = [0.5 0 1];
m.LineStyle = 'none';
plot3(X(1,:),X(2,:),[1 1 1]+0.1,'.','markersize',30)
plot3(x(1),x(2),1.1,'.','markersize',30,'color',[1 0 0])
hold off
view(2)
title('The three classes.')
xlabel('X(1,:) and x(1)')
ylabel('X(2,:) and x(2)')
```


The three classes.
X(2,:) and x(2)
X(1,:) and x(1)
0
0.5
1
1.5
2
2.5
3

Learning Vector Quantization

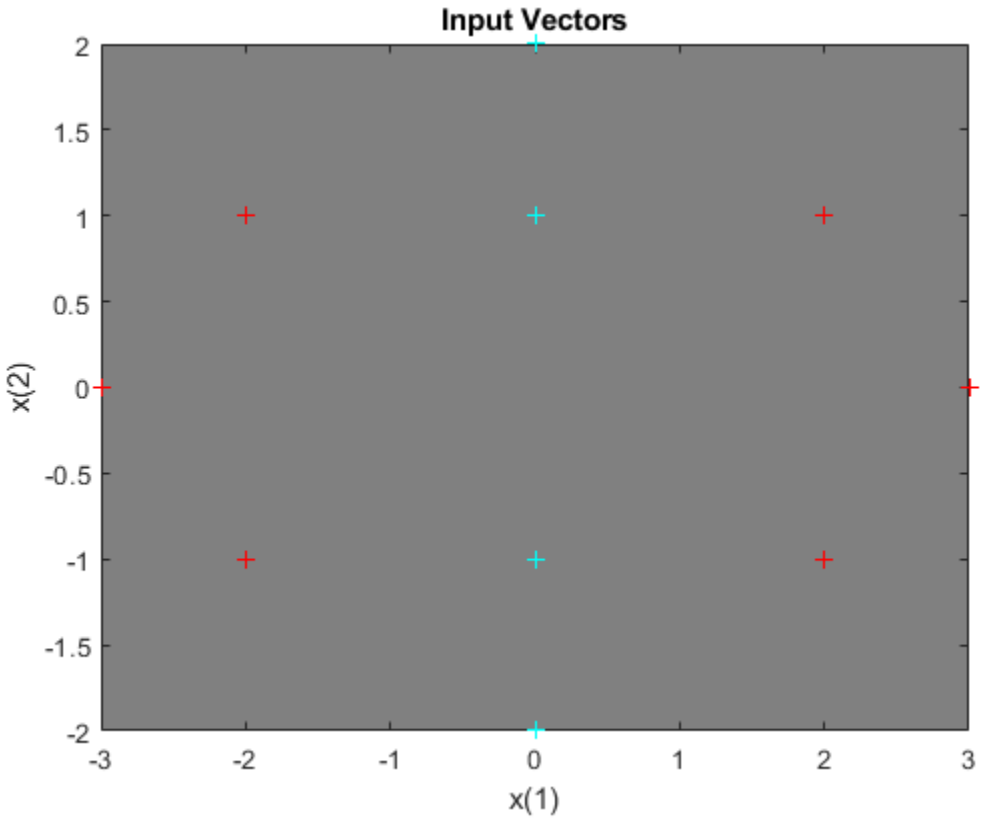
An LVQ network is trained to classify input vectors according to given targets.

Let X be 10 2-element example input vectors and C be the classes these vectors fall into. These classes can be transformed into vectors to be used as targets, T, with IND2VEC.

```
x = [-3 -2 -2  0  0  0  0 +2 +2 +3;
      0 +1 -1 +2 +1 -1 -2 +1 -1  0];
c = [1 1 1 2 2 2 2 1 1 1];
t = ind2vec(c);
```

Here the data points are plotted. Red = class 1, Cyan = class 2. The LVQ network represents clusters of vectors with hidden neurons, and groups the clusters with output neurons to form the desired classes.

```
colormap(hsv);
plotvec(x,c)
title('Input Vectors');
xlabel('x(1)');
ylabel('x(2)');
```

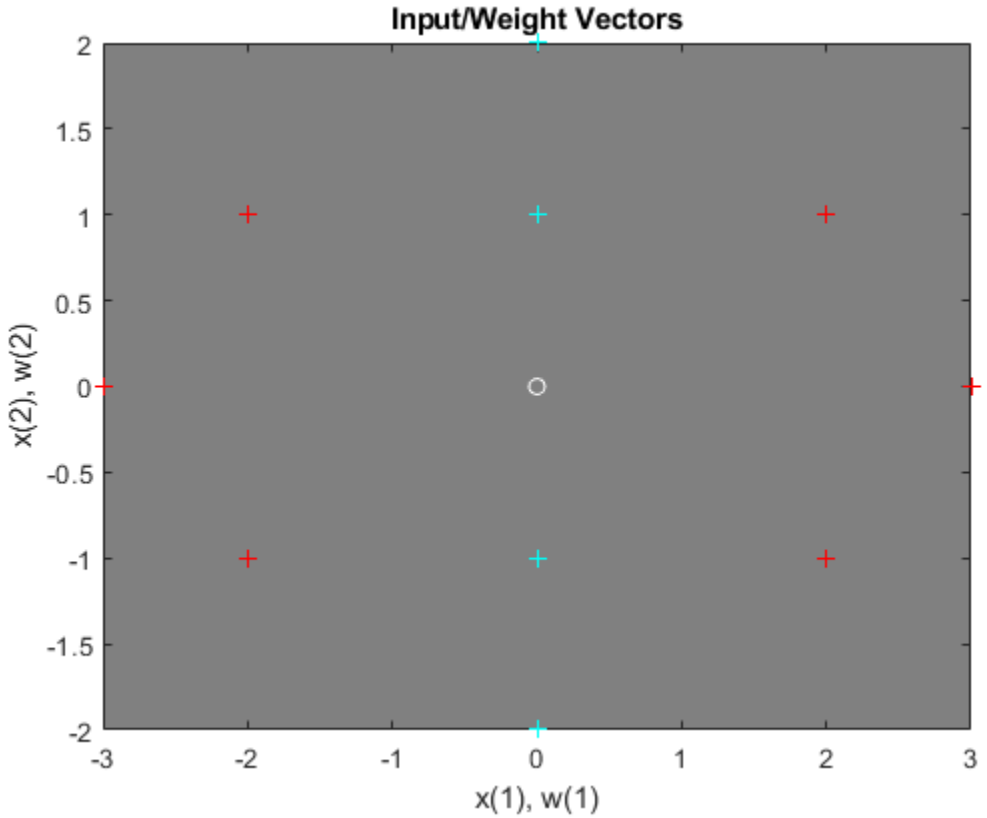


Here LVQNET creates an LVQ layer with four hidden neurons and a learning rate of 0.1. The network is then configured for inputs X and targets T. (Configuration normally an unnecessary step as it is done automatically by TRAIN.)

```
net = lvqnet(4,0.1);
net = configure(net,x,t);
```

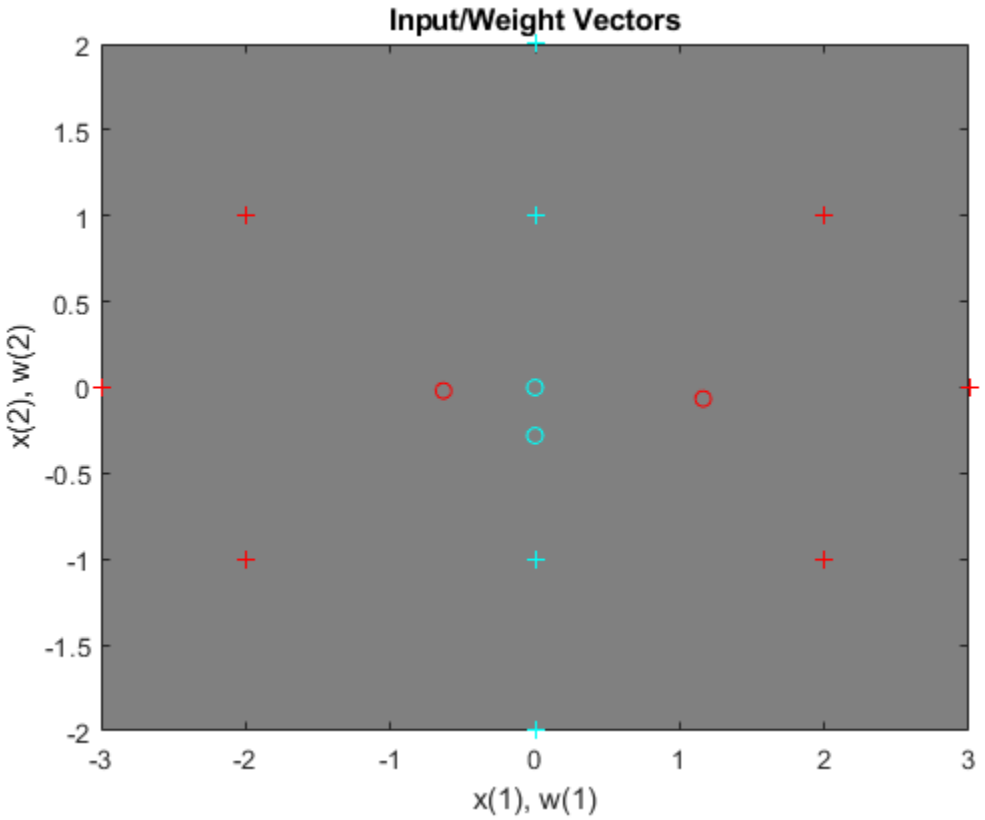
The competitive neuron weight vectors are plotted as follows.

```
hold on
w1 = net.IW{1};
plot(w1(1,1),w1(1,2),'ow')
title('Input/Weight Vectors');
xlabel('x(1), w(1)');
ylabel('x(2), w(2)');
```



To train the network, first override the default number of epochs, and then train the network. When it is finished, replot the input vectors '+' and the competitive neurons' weight vectors 'o'. Red = class 1, Cyan = class 2.

```
net.trainParam.epochs=150;
net=train(net,x,t);

cla;
plotvec(x,c);
hold on;
plotvec(net.IW{1}',vec2ind(net.LW{2}),'o');
```



Now use the LVQ network as a classifier, where each neuron corresponds to a different category. Present the input vector [0.2; 1]. Red = class 1, Cyan = class 2.

```
x1 = [0.2; 1];
y1 = vec2ind(net(x1))
```

```
y1 = 2
```

Linear Prediction Design

This example illustrates how to design a linear neuron to predict the next value in a time series given the last five values.

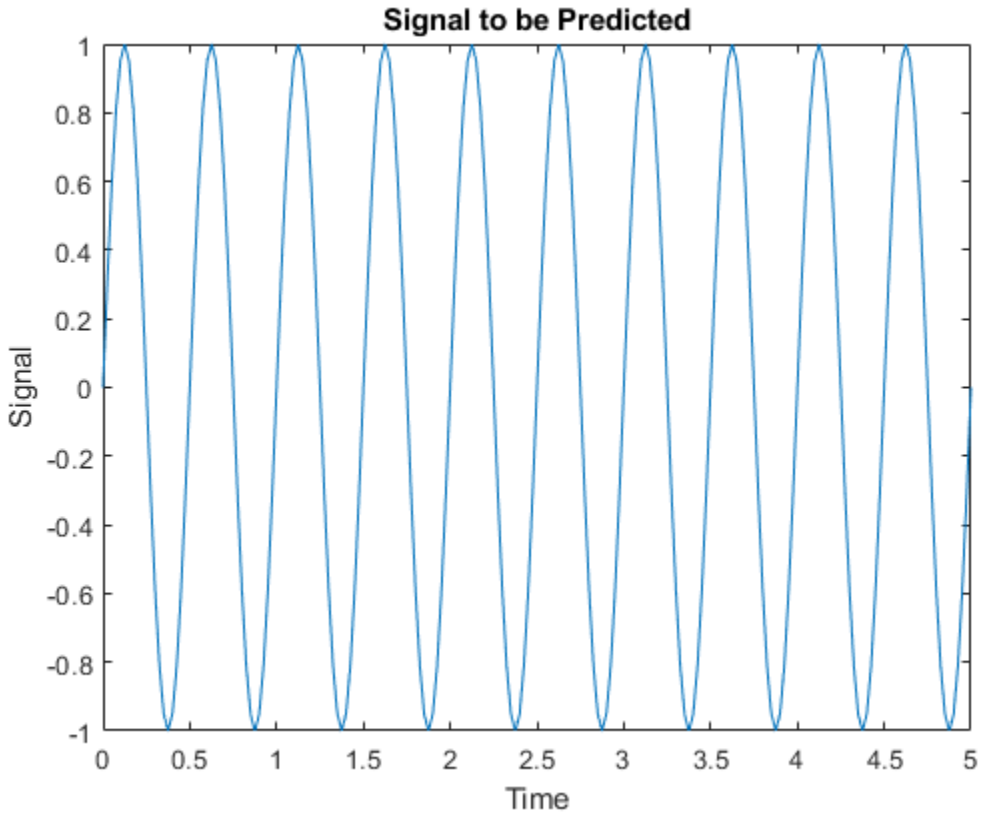
Defining a Wave Form

Here time is defined from 0 to 5 seconds in steps of 1/40 of a second.

```
time = 0:0.025:5;
```

We can define a signal with respect to time.

```
signal = sin(time*4*pi);
plot(time,signal)
xlabel('Time');
ylabel('Signal');
title('Signal to be Predicted');
```



Setting up the Problem for a Neural Network

The signal convert is then converted to a cell array. Neural Networks represent timesteps as columns of a cell array, do distinguish them from different samples at a given time, which are represented with columns of matrices.

```
signal = con2seq(signal);
```

To set up the problem we will use the first four values of the signal as initial input delay states, and the rest except for the last step as inputs.

```
Xi = signal(1:4);
X = signal(5:(end-1));
timex = time(5:(end-1));
```

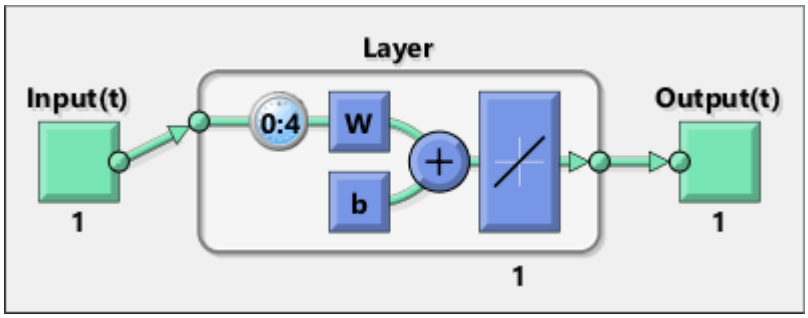
The targets are now defined to match the inputs, but shifted earlier by one timestep.

```
T = signal(6:end);
```

Designing the Linear Layer

The function **newlind** will now design a linear layer with a single neuron which predicts the next timestep of the signal given the current and four past values.

```
net = newlind(X,T,Xi);
view(net)
```



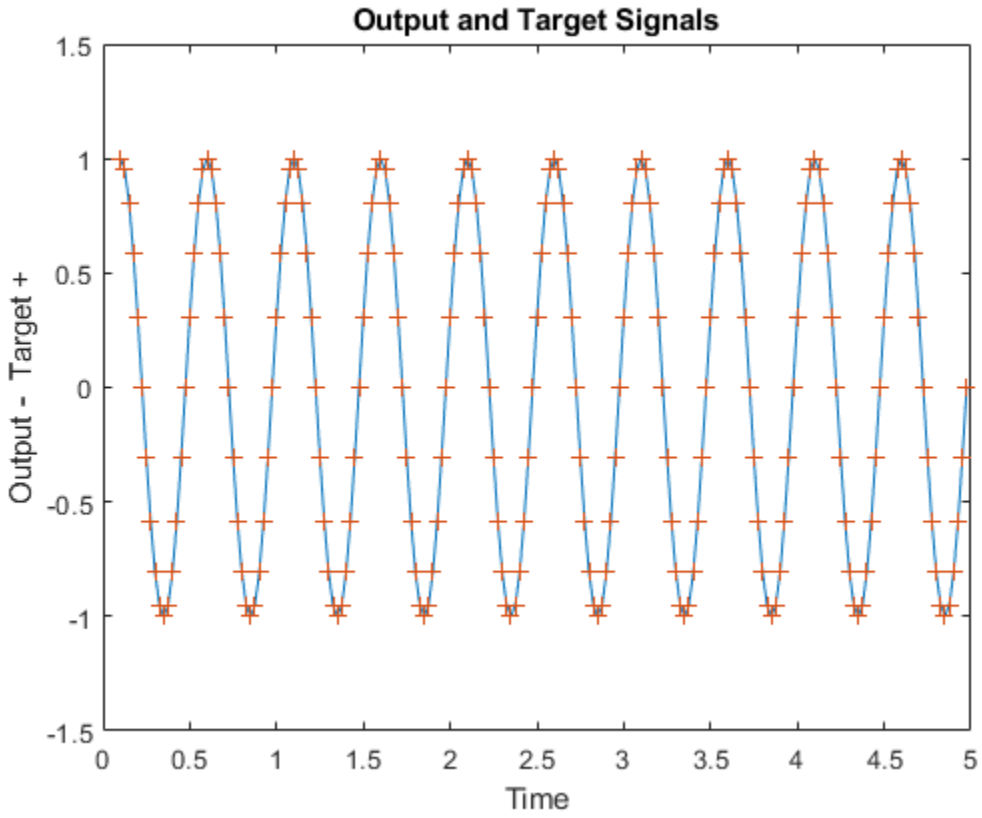
Testing the Linear Layer

The network can now be called like a function on the inputs and delayed states to get its time response.

```
Y = net(X,Xi);
```

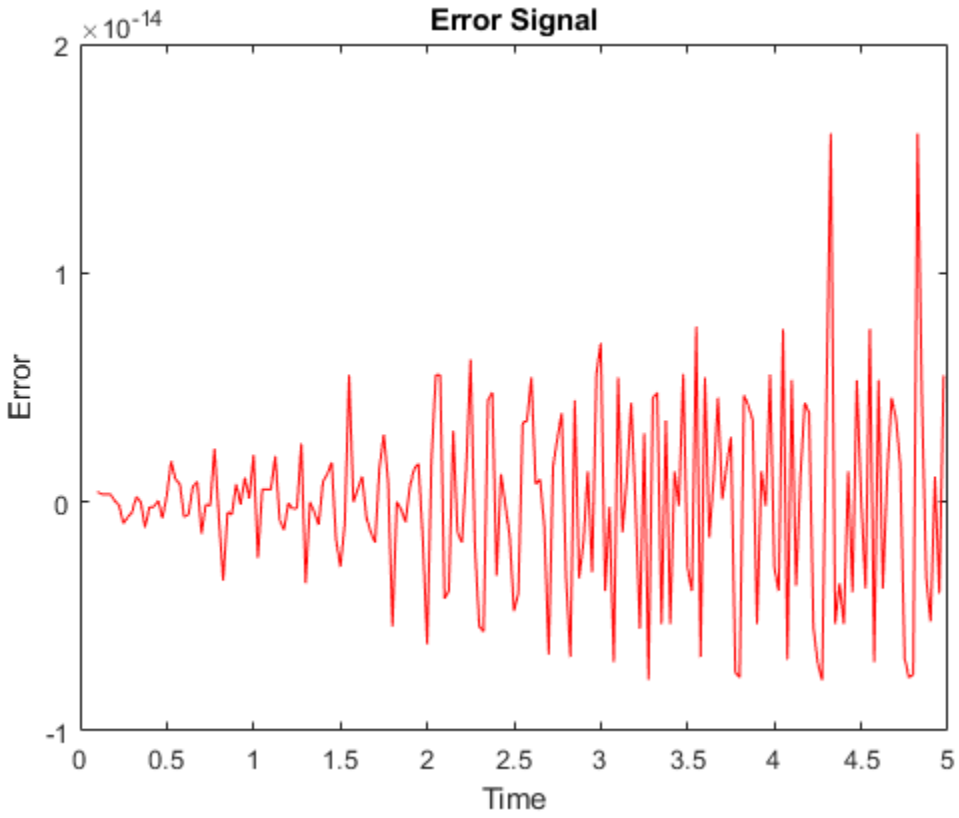
The output signal is plotted with the targets.

```
figure
plot(timex,cell2mat(Y),timex,cell2mat(T),'+')
xlabel('Time');
ylabel('Output -  Target +');
title('Output and Target Signals');
```



The error can also be plotted.

```
figure
E = cell2mat(T)-cell2mat(Y);
plot(timex,E,'r')
hold off
xlabel('Time');
ylabel('Error');
title('Error Signal');
```



Notice how small the error is!

This example illustrated how to design a dynamic linear network which can predict a signal's next value from current and past values.

Adaptive Linear Prediction

This example shows how an adaptive linear layer can learn to predict the next value in a signal, given the current and last four values.

To learn how to forecast time series data using a deep learning network, see “Time Series Forecasting Using Deep Learning” on page 4-15.

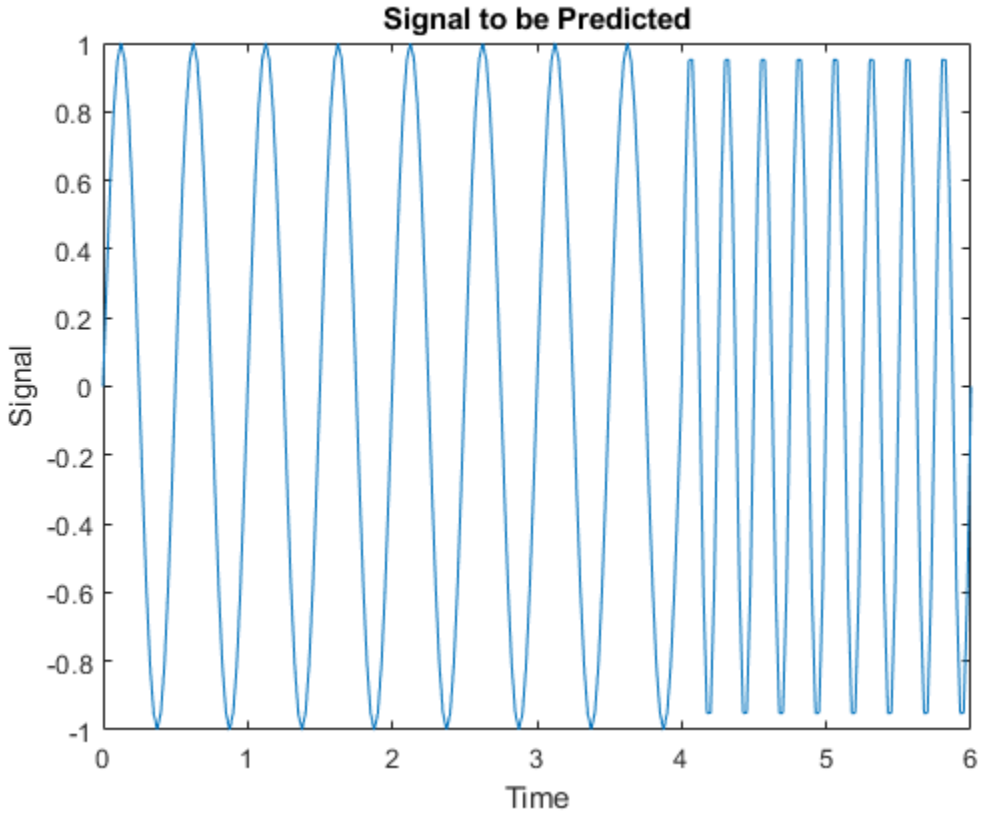
Defining a Wave Form

Here two time segments are defined from 0 to 6 seconds in steps of 1/40 of a second.

```
time1 = 0:0.025:4;      % from 0 to 4 seconds
time2 = 4.025:0.025:6;  % from 4 to 6 seconds
time = [time1 time2];  % from 0 to 6 seconds
```

Here is a signal which starts at one frequency but then transitions to another frequency.

```
signal = [sin(time1*4*pi) sin(time2*8*pi)];
plot(time,signal)
xlabel('Time');
ylabel('Signal');
title('Signal to be Predicted');
```



Setting up the Problem for a Neural Network

The signal convert is then converted to a cell array. Neural Networks represent timesteps as columns of a cell array, do distinguish them from different samples at a given time, which are represented with columns of matrices.

```
signal = con2seq(signal);
```

To set up the problem we will use the first five values of the signal as initial input delay states, and the rest for inputs.

```
Xi = signal(1:5);
X = signal(6:end);
timex = time(6:end);
```

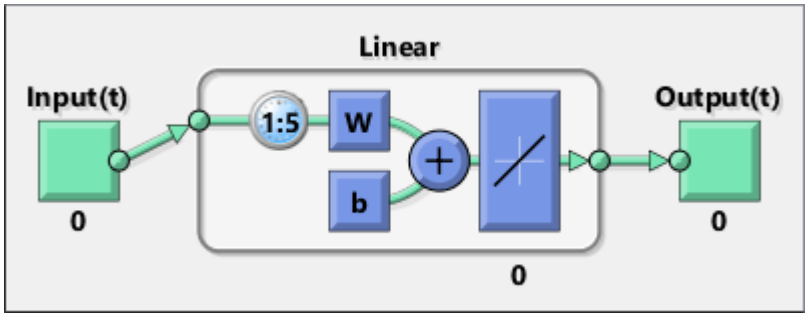
The targets are now defined to match the inputs. The network is to predict the current input, only using the last five values.

```
T = signal(6:end);
```

Creating the Linear Layer

The function **linearlayer** creates a linear layer with a single neuron with a tap delay of the last five inputs.

```
net = linearlayer(1:5,0.1);
view(net)
```

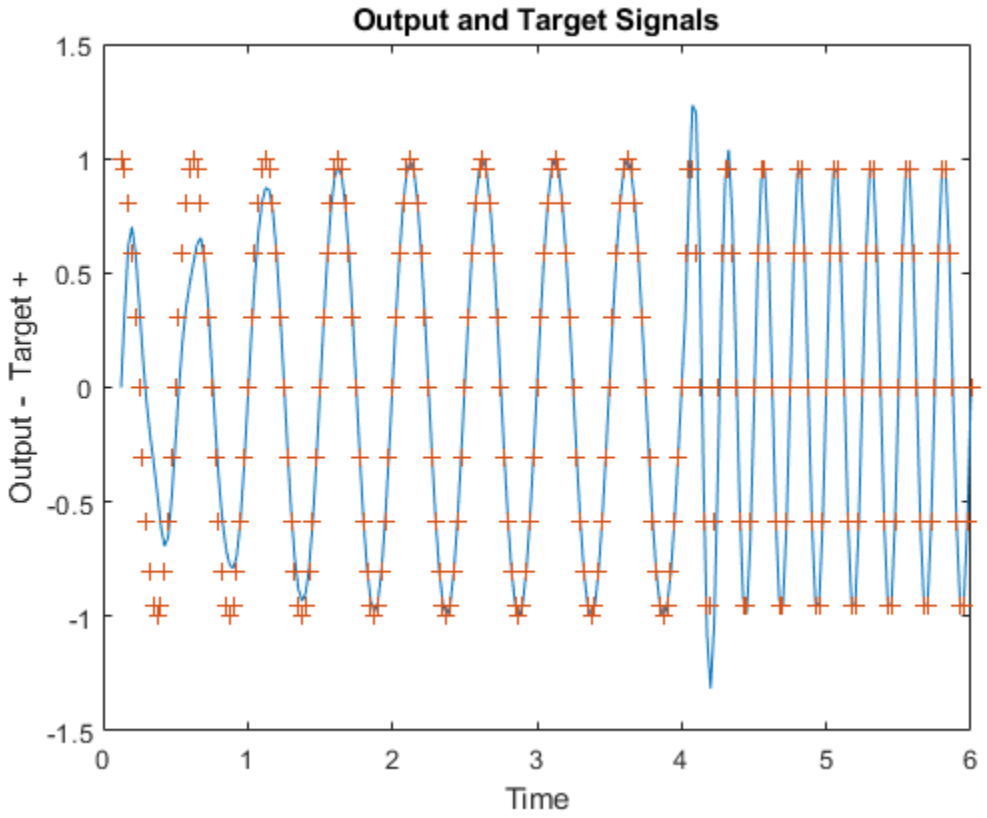


Adapting the Linear Layer

```
The function *adapt* simulates the network on the input, while
adjusting its weights and biases after each timestep in response
to how closely its output matches the target.

It returns the update networks, it outputs, and its errors.

[net,Y] = adapt(net,X,T,Xi);
```

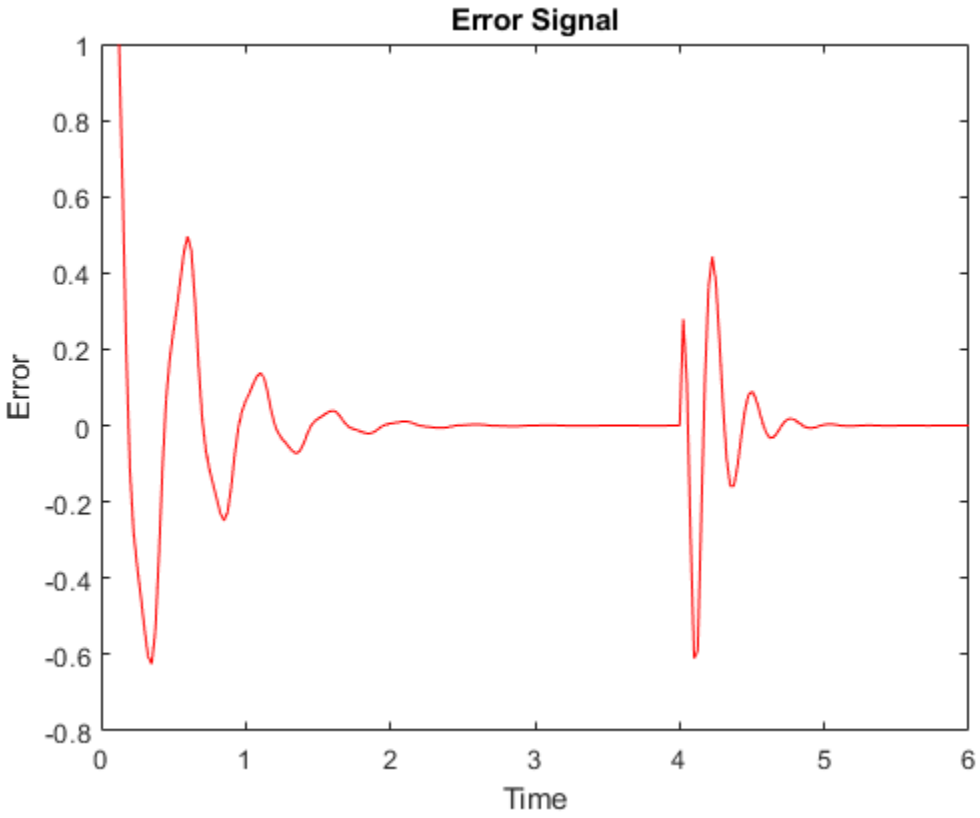
The output signal is plotted with the targets.

```
figure
plot(timex,cell2mat(Y),timex,cell2mat(T),'+')
xlabel('Time');
ylabel('Output -  Target +');
title('Output and Target Signals');
```



The error can also be plotted.

```
figure
E = cell2mat(T)-cell2mat(Y);
plot(timex,E,'r')
hold off
xlabel('Time');
ylabel('Error');
title('Error Signal');
```



Notice how small the error is except for initial errors and the network learns the systems behavior at the beginning and after the system transition.

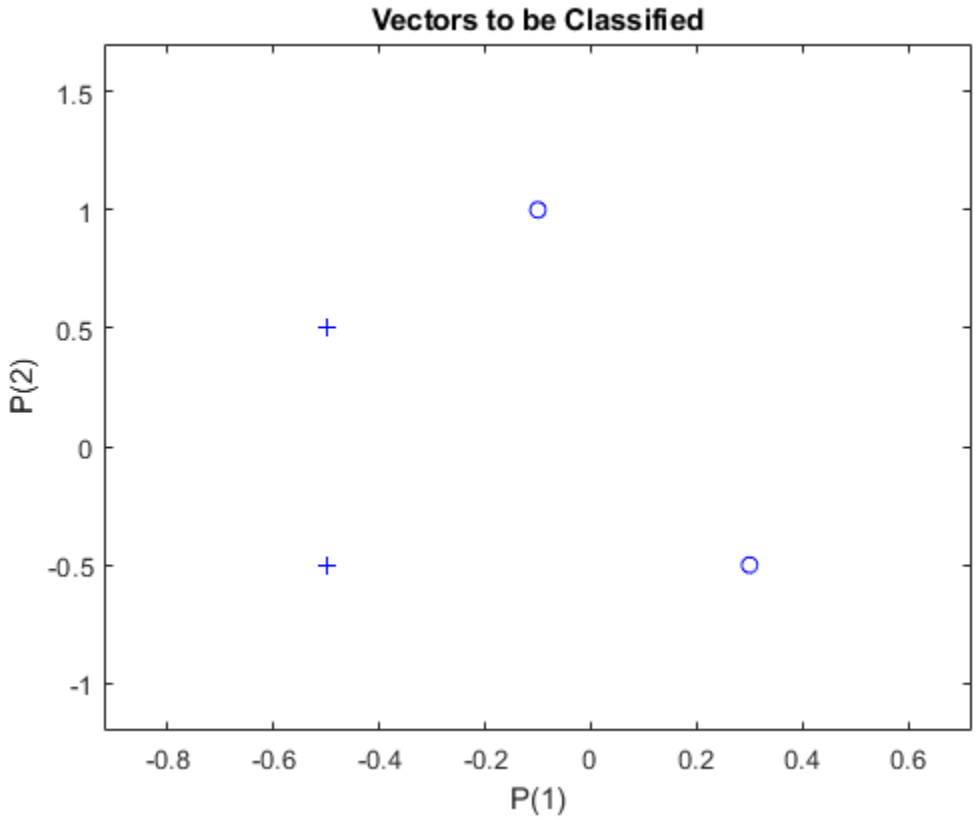
This example illustrated how to simulate an adaptive linear network which can predict a signal's next value from current and past values despite changes in the signals behavior.

Classification with a Two-Input Perceptron

A two-input hard limit neuron is trained to classify four input vectors into two categories.

Each of the four column vectors in X defines a two-element input vectors and a row vector T defines the vector's target categories. We can plot these vectors with PLOTPV.

```
X = [ -0.5 -0.5 +0.3 -0.1;  ...
      -0.5 +0.5 -0.5 +1.0];
T = [1 1 0 0];
plotpv(X,T);
```



The perceptron must properly classify the four input vectors in X into the two categories defined by T. Perceptrons have HARDLIM neurons. These neurons are capable of separating an input space with a straight line into two categories (0 and 1).

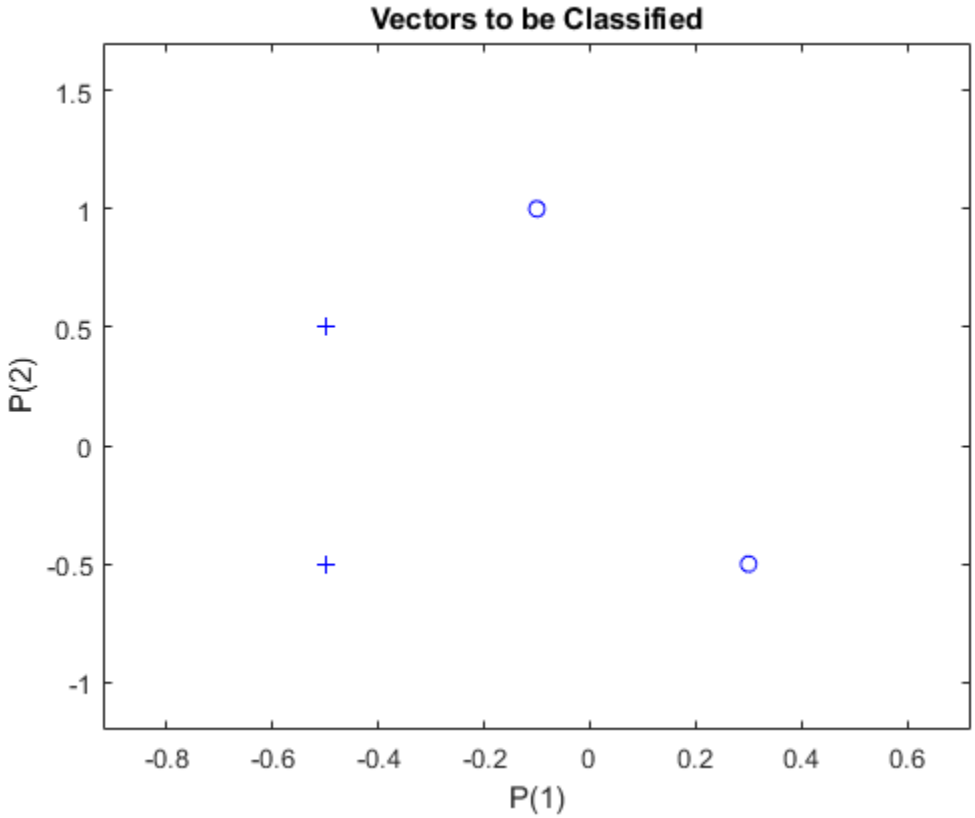
Here PERCEPTRON creates a new neural network with a single neuron. The network is then configured to the data, so we can examine its initial weight and bias values. (Normally the configuration step can be skipped as it is automatically done by ADAPT or TRAIN.)

```
net = perceptron;
net = configure(net,X,T);
```

The input vectors are replotted with the neuron's initial attempt at classification.

The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not ... we are going to train it!

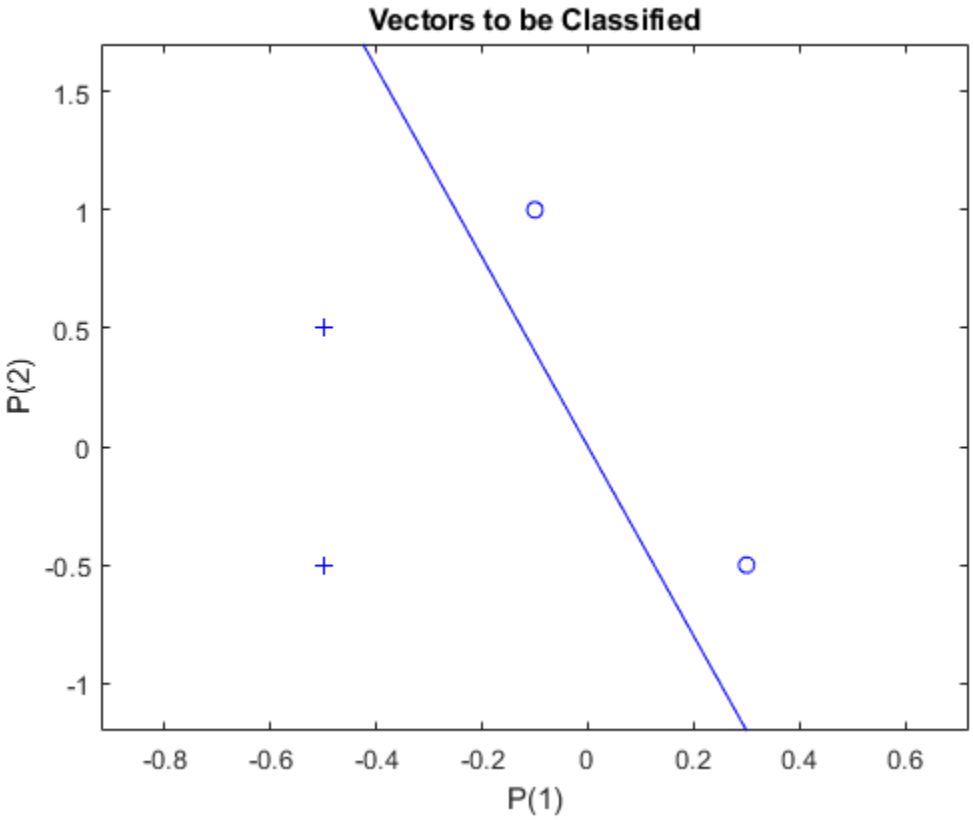
```
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
```



Here the input and target data are converted to sequential data (cell array where each column indicates a timestep) and copied three times to form the series XX and TT.

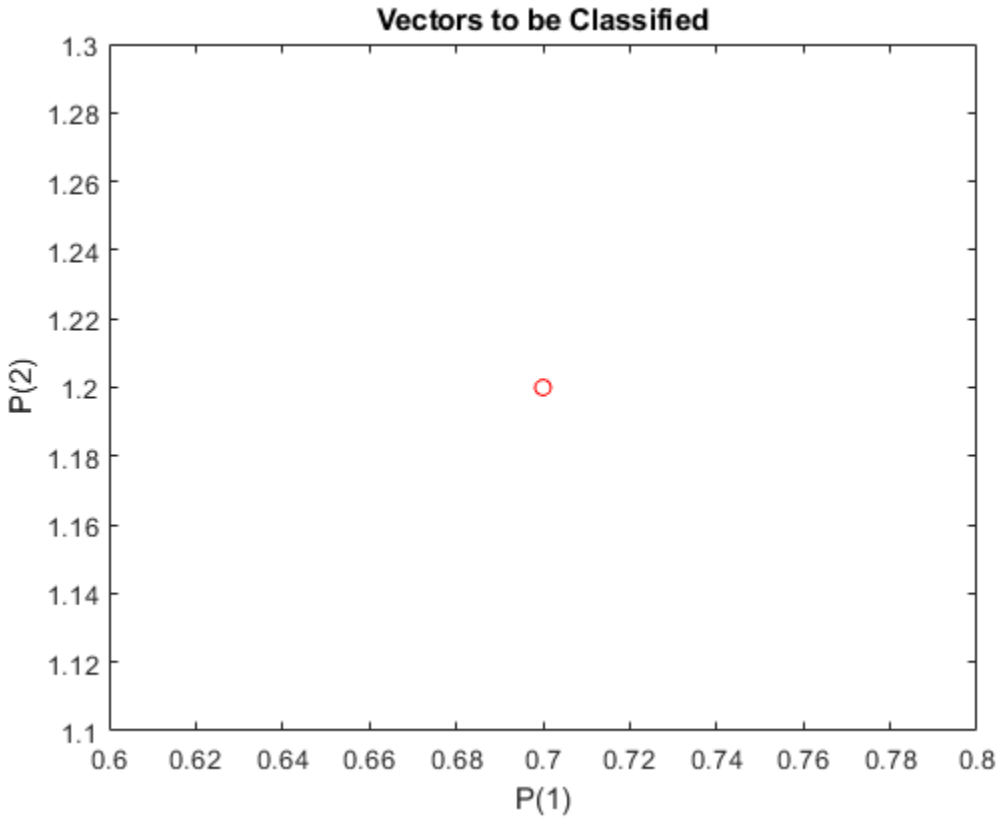
ADAPT updates the network for each timestep in the series and returns a new network object that performs as a better classifier.

```
XX = repmat(con2seq(X),1,3);
TT = repmat(con2seq(T),1,3);
net = adapt(net,XX,TT);
plotpc(net.IW{1},net.b{1});
```



Now SIM is used to classify any other input vector, like [0.7; 1.2]. A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

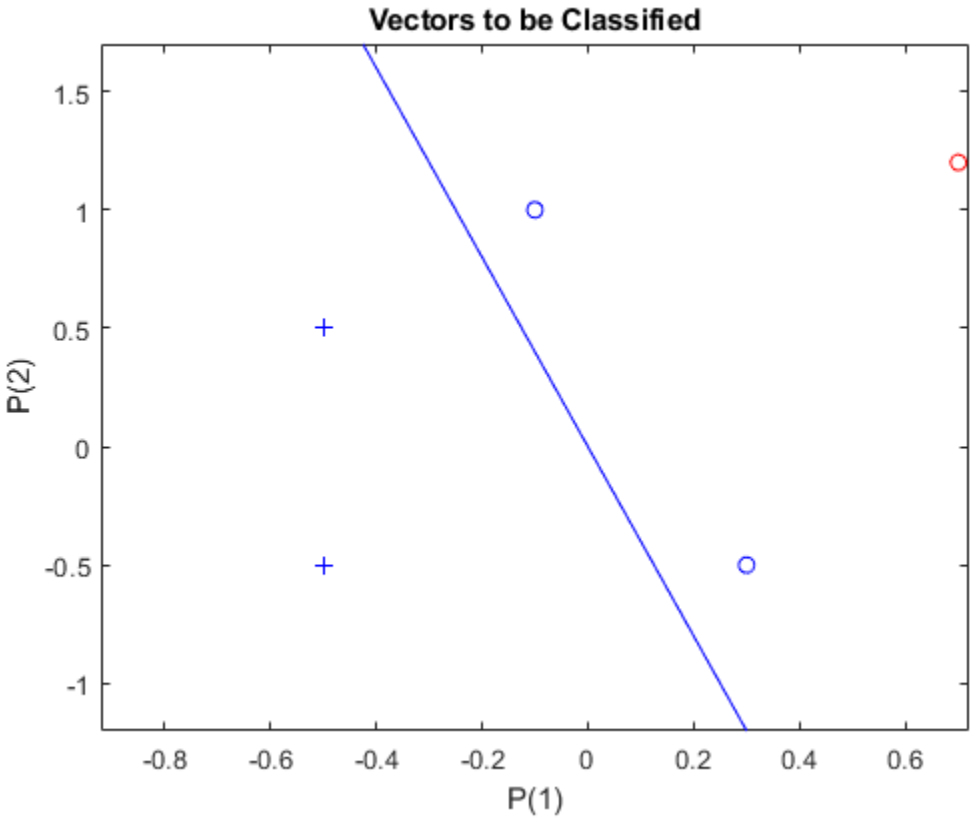
```
x = [0.7; 1.2];
y = net(x);
plotpv(x,y);
point = findobj(gca,'type','line');
point.Color = 'red';
```



Turn on "hold" so the previous plot is not erased and plot the training set and the classification line.

The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus).

```
hold on;
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
hold off;
```

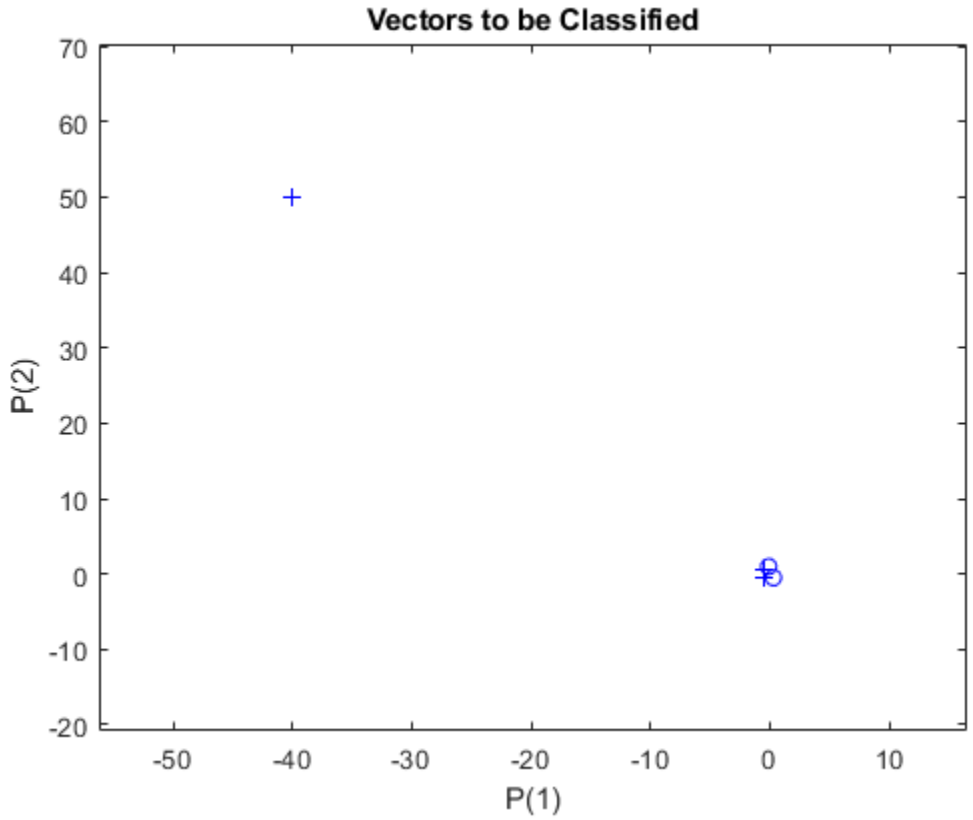

Vectors to be Classified
1.5
1
0.5
0
-0.5
-1
P(2)
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
P(1)

Outlier Input Vectors

A 2-input hard limit neuron is trained to classify 5 input vectors into two categories. However, because 1 input vector is much larger than all of the others, training takes a long time.

Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

```
X = [-0.5 -0.5 +0.3 -0.1 -40; -0.5 +0.5 -0.5 +1.0 50];
T = [1 1 0 0 1];
plotpv(X,T);
```



Note that 4 input vectors have much smaller magnitudes than the fifth vector in the upper left of the plot. The perceptron must properly classify the 5 input vectors in X into the two categories defined by T.

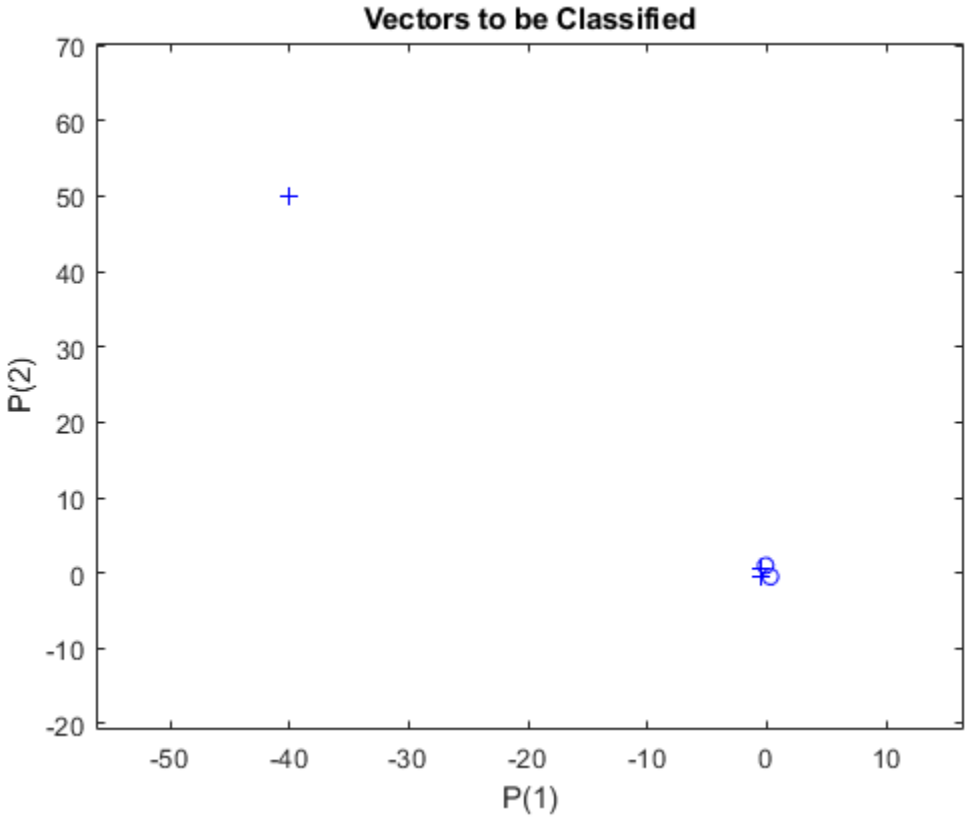
PERCEPTRON creates a new network which is then configured with the input and target data which results in initial values for its weights and bias. (Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

```
net = perceptron;
net = configure(net,X,T);
```

Add the neuron's initial attempt at classification to the plot.

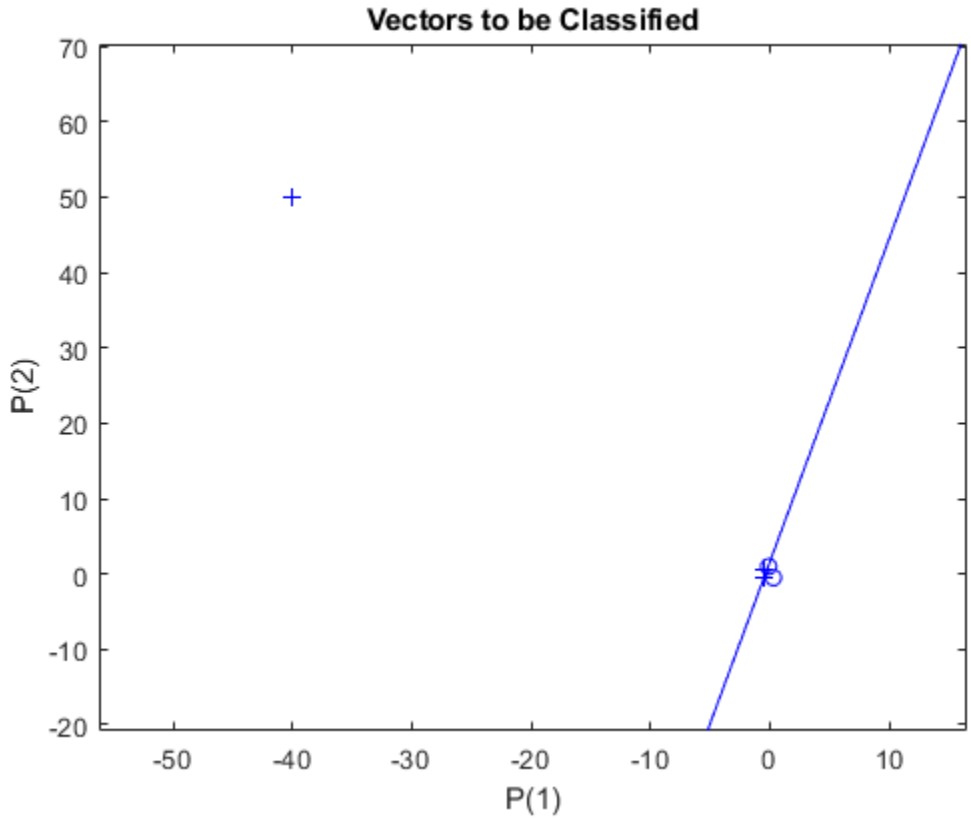
The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
hold on
linehandle = plotpc(net.IW{1},net.b{1});
```



ADAPT returns a new network object that performs as a better classifier, the network output, and the error. This loop adapts the network and plots the classification line, until the error is zero.

```
E = 1;
while (sse(E))
   [net,Y,E] = adapt(net,X,T);
   linehandle = plotpc(net.IW{1},net.b{1},linehandle);
   drawnow;
end
```

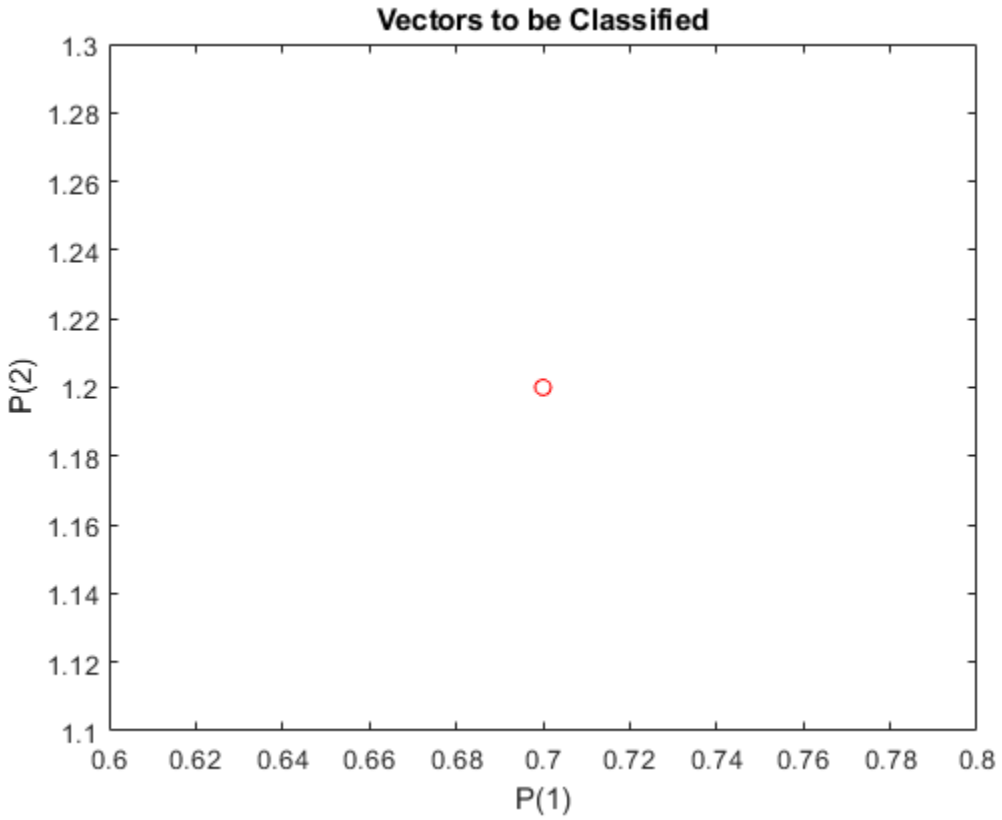


Note that it took the perceptron three passes to get it right. This a long time for such a simple problem. The reason for the long training time is the outlier vector. Despite the long training time, the perceptron still learns properly and can be used to classify other inputs.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

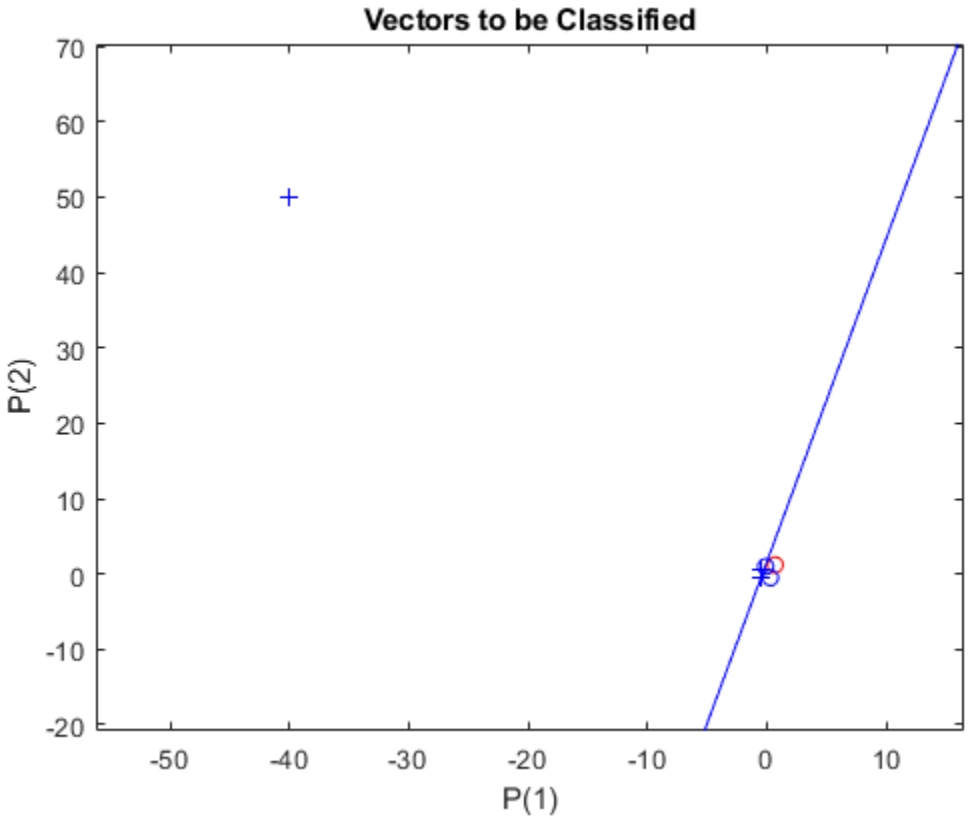
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
x = [0.7; 1.2];
y = net(x);
plotpv(x,y);
circle = findobj(gca,'type','line');
circle.Color = 'red';
```



Turn on "hold" so the previous plot is not erased. Add the training set and the classification line to the plot.

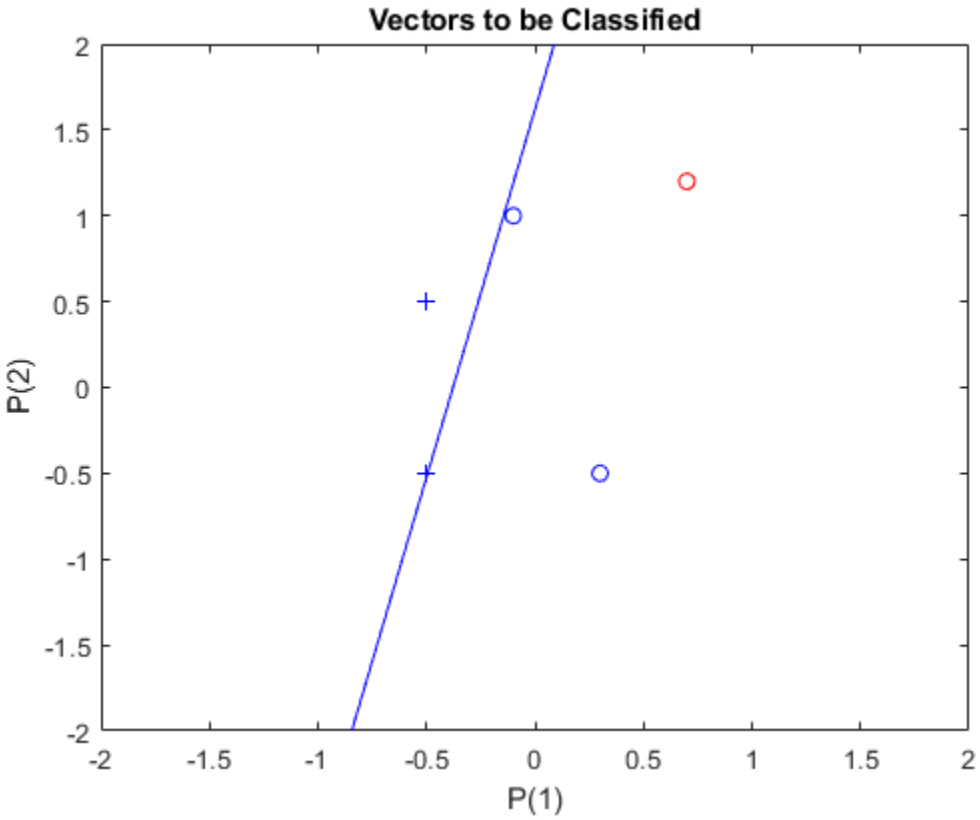
```
hold on;
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
hold off;
```



Finally, zoom into the area of interest.

The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus). Despite the long training time, the perceptron still learns properly. To see how to reduce training times associated with outlier vectors, see the "Normalized Perceptron Rule" example.

```
axis([-2 2 -2 2]);
```

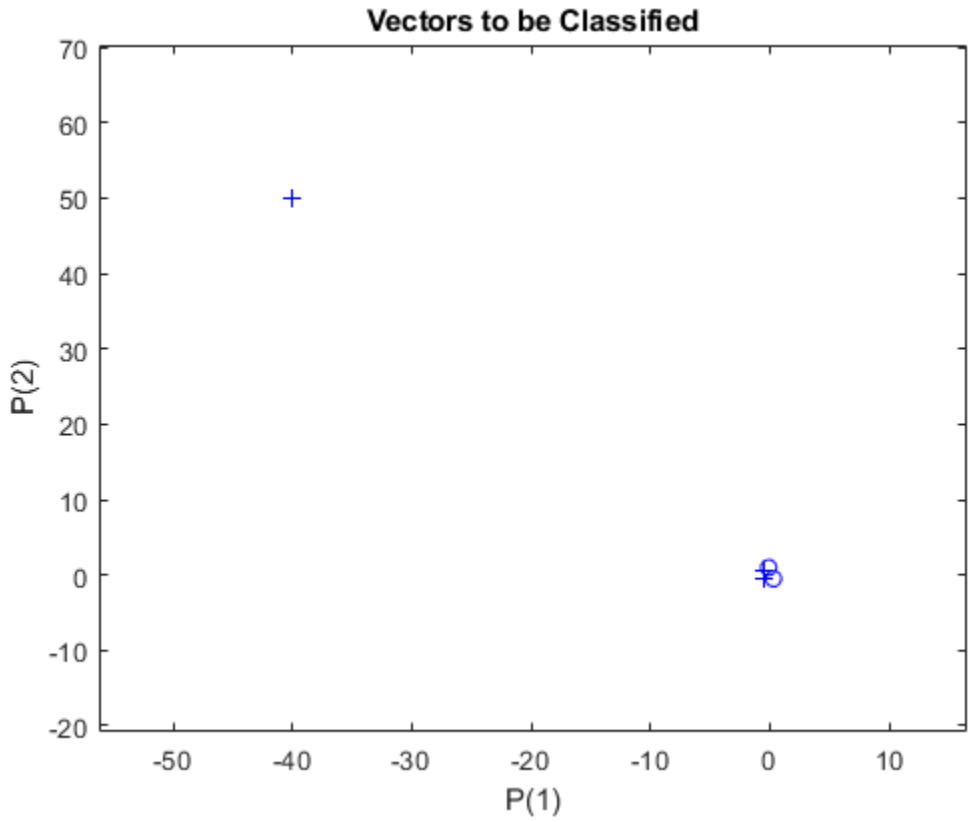

Vectors to be Classified
P(2)
P(1)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

Normalized Perceptron Rule

A 2-input hard limit neuron is trained to classify 5 input vectors into two categories. Despite the fact that one input vector is much bigger than the others, training with LEARNPN is quick.

Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

```
X = [ -0.5 -0.5 +0.3 -0.1 -40; ...
      -0.5 +0.5 -0.5 +1.0 50];
T = [1 1 0 0 1];
plotpv(X,T);
```



Note that 4 input vectors have much smaller magnitudes than the fifth vector in the upper left of the plot. The perceptron must properly classify the 5 input vectors in X into the two categories defined by T.

PERCEPTRON creates a new network with LEARPN learning rule, which is less sensitive to large variations in input vector size than LEARNP (the default).

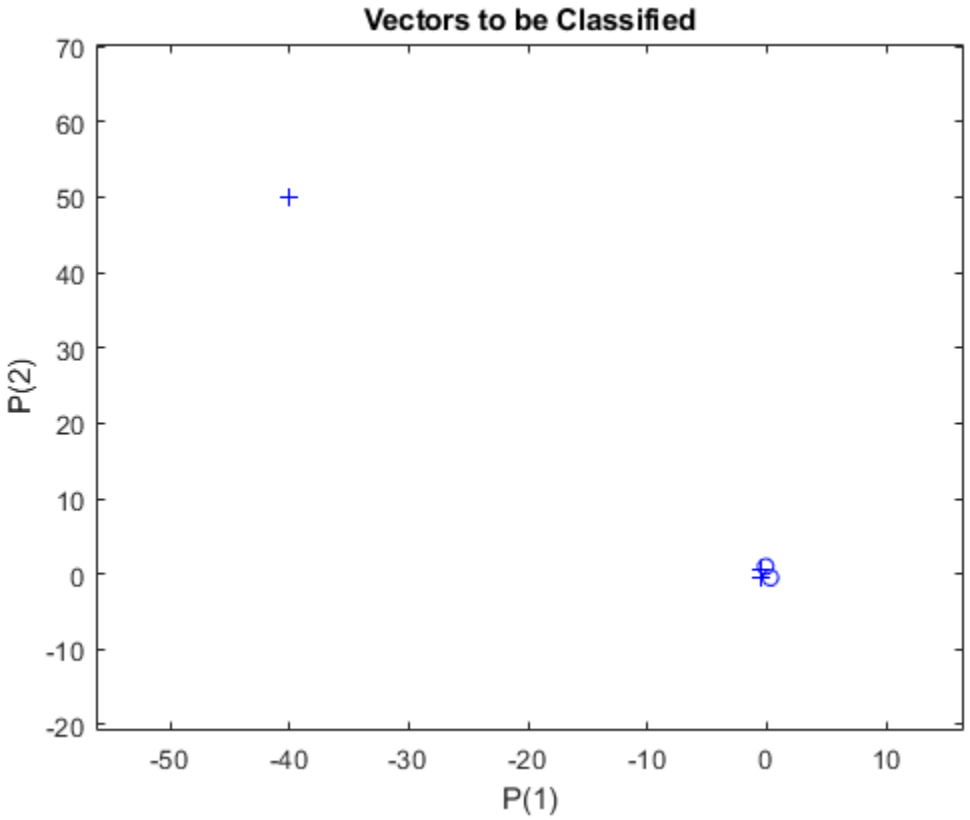
The network is then configured with the input and target data which results in initial values for its weights and bias. (Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

```
net = perceptron('hardlim','learnpn');
net = configure(net,X,T);
```

Add the neuron's initial attempt at classification to the plot.

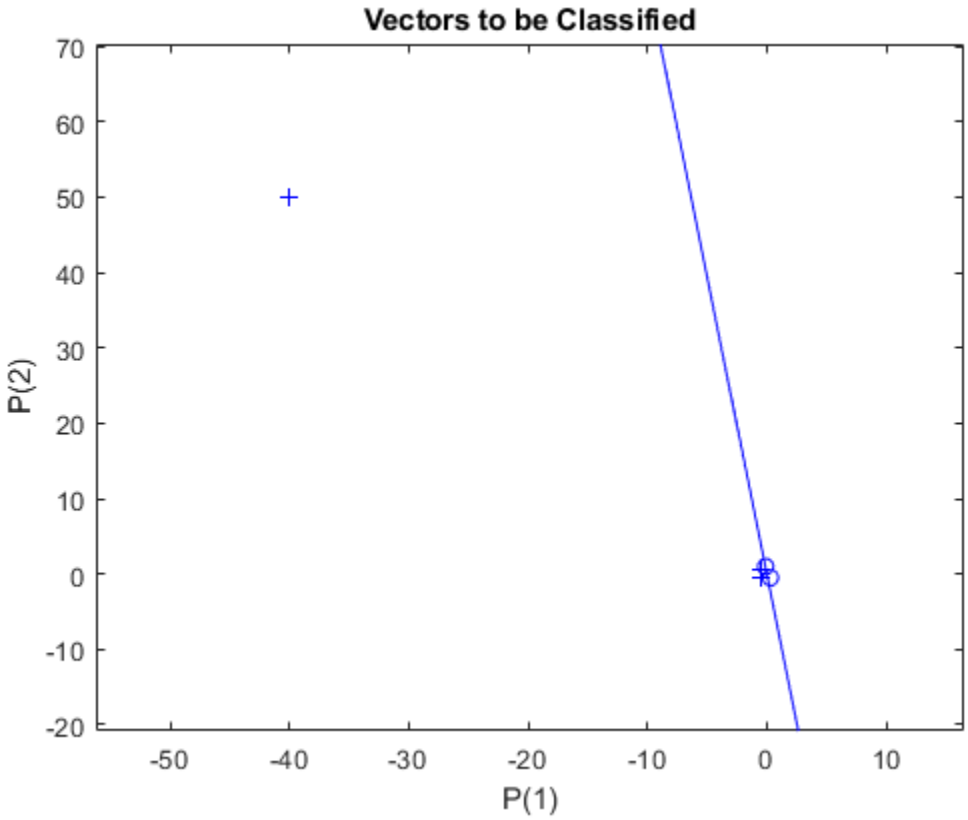
The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
hold on
linehandle = plotpc(net.IW{1},net.b{1});
```



ADAPT returns a new network object that performs as a better classifier, the network output, and the error. This loop allows the network to adapt, plots the classification line, and continues until the error is zero.

```
E = 1;
while (sse(E))
   [net,Y,E] = adapt(net,X,T);
   linehandle = plotpc(net.IW{1},net.b{1},linehandle);
   drawnow;
end
```

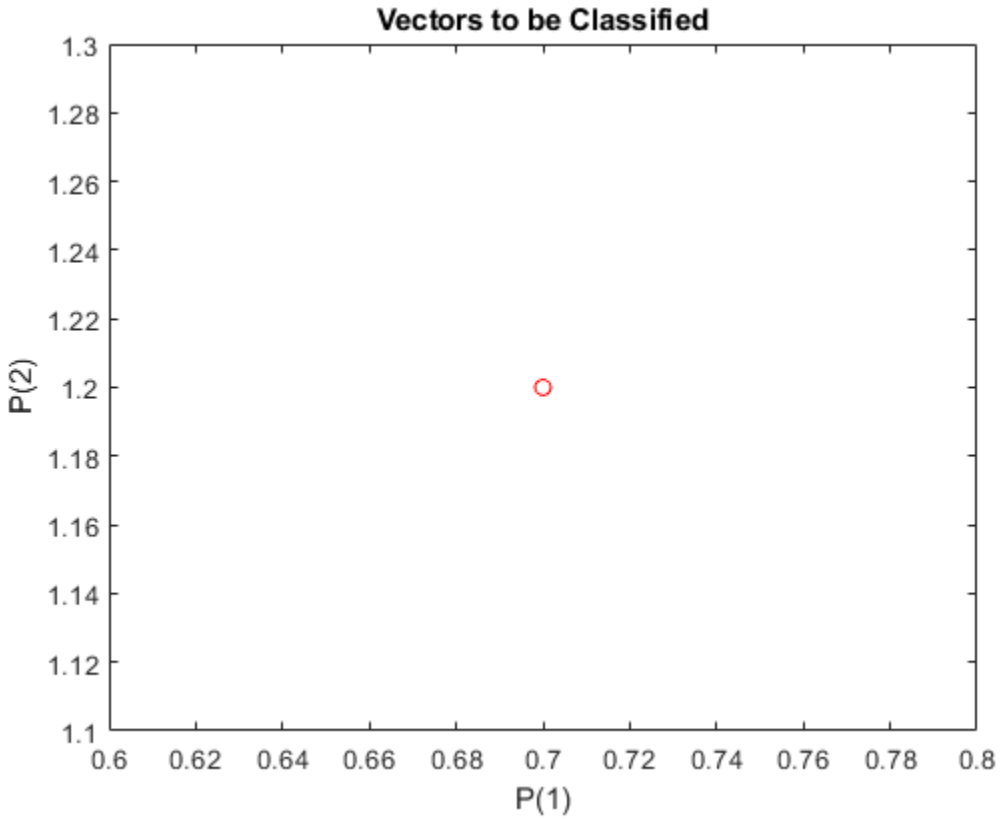


Note that training with LEARNP took only 3 epochs, while solving the same problem with LEARNPN required 32 epochs. Thus, LEARNPN does much better job than LEARNP when there are large variations in input vector size.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

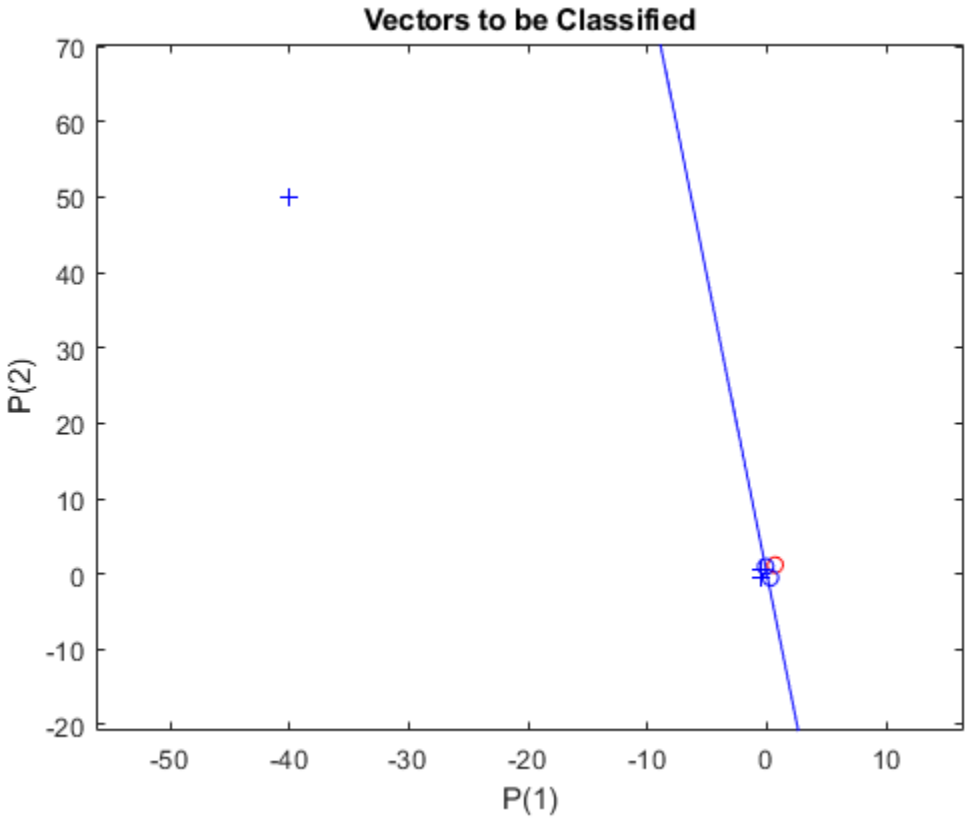
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
x = [0.7; 1.2];
y = net(x);
plotpv(x,y);
circle = findobj(gca,'type','line');
circle.Color = 'red';
```



Turn on "hold" so the previous plot is not erased. Add the training set and the classification line to the plot.

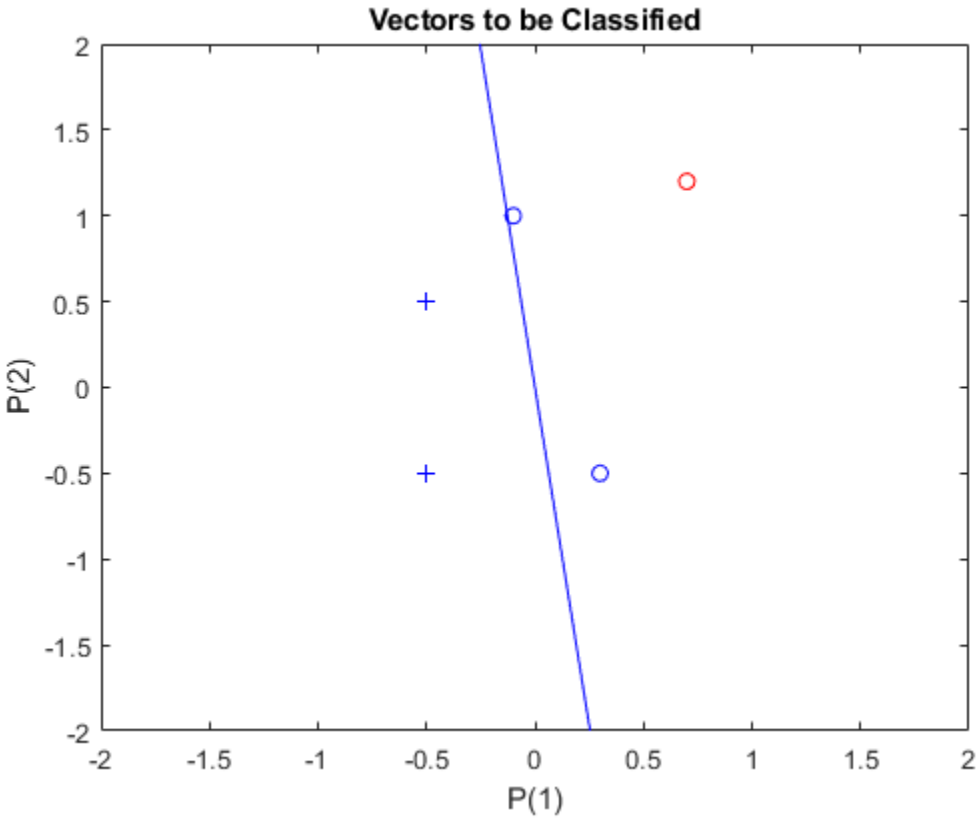
```
hold on;
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
hold off;
```



Finally, zoom into the area of interest.

The perceptron correctly classified our new point (in red) as category "zero" (represented by a circle) and not a "one" (represented by a plus). The perceptron learns properly in much shorter time in spite of the outlier (compare with the "Outlier Input Vectors" example).

```
axis([-2 2 -2 2]);
```

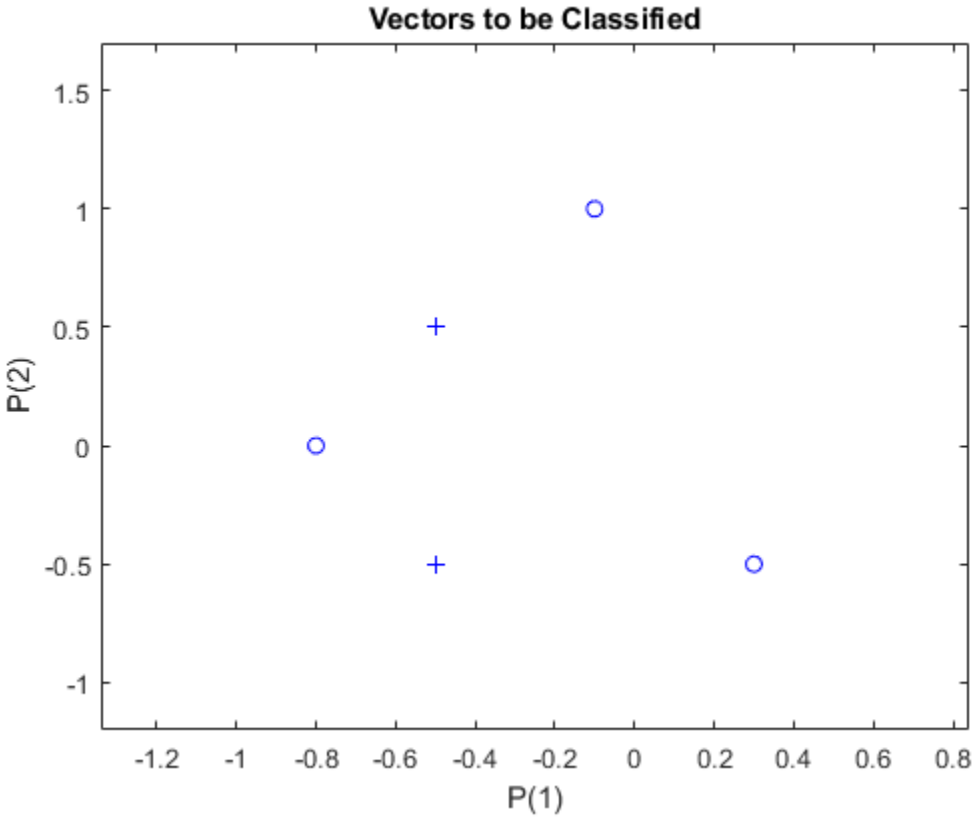

Vectors to be Classified
P(2)
P(1)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

Linearly Non-separable Vectors

A 2-input hard limit neuron fails to properly classify 5 input vectors because they are linearly non-separable.

Each of the five column vectors in X defines a 2-element input vectors, and a row vector T defines the vector's target categories. Plot these vectors with PLOTPV.

```
X = [ -0.5 -0.5 +0.3 -0.1 -0.8; ...
      -0.5 +0.5 -0.5 +1.0 +0.0 ];
T = [1 1 0 0 0];
plotpv(X,T);
```



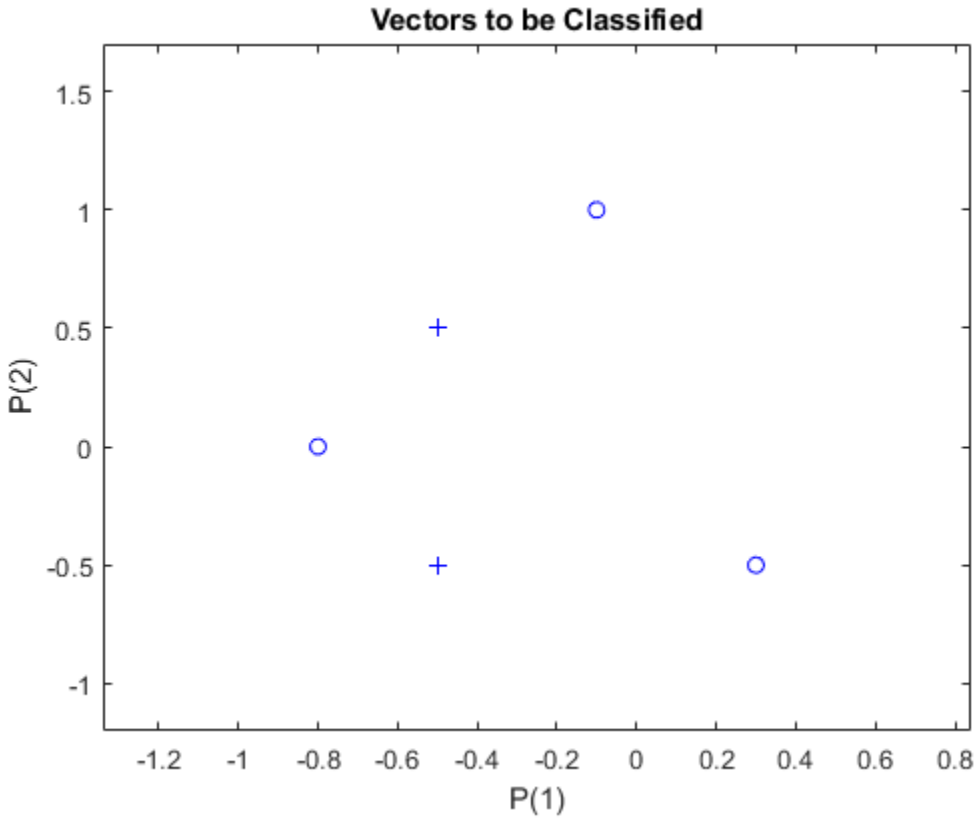
The perceptron must properly classify the input vectors in X into the categories defined by T. Because the two kinds of input vectors cannot be separated by a straight line, the perceptron will not be able to do it.

Here the initial perceptron is created and configured. (The configuration step is normally optional, as it is performed automatically by ADAPT and TRAIN.)

```
net = perceptron;
net = configure(net,X,T);
```

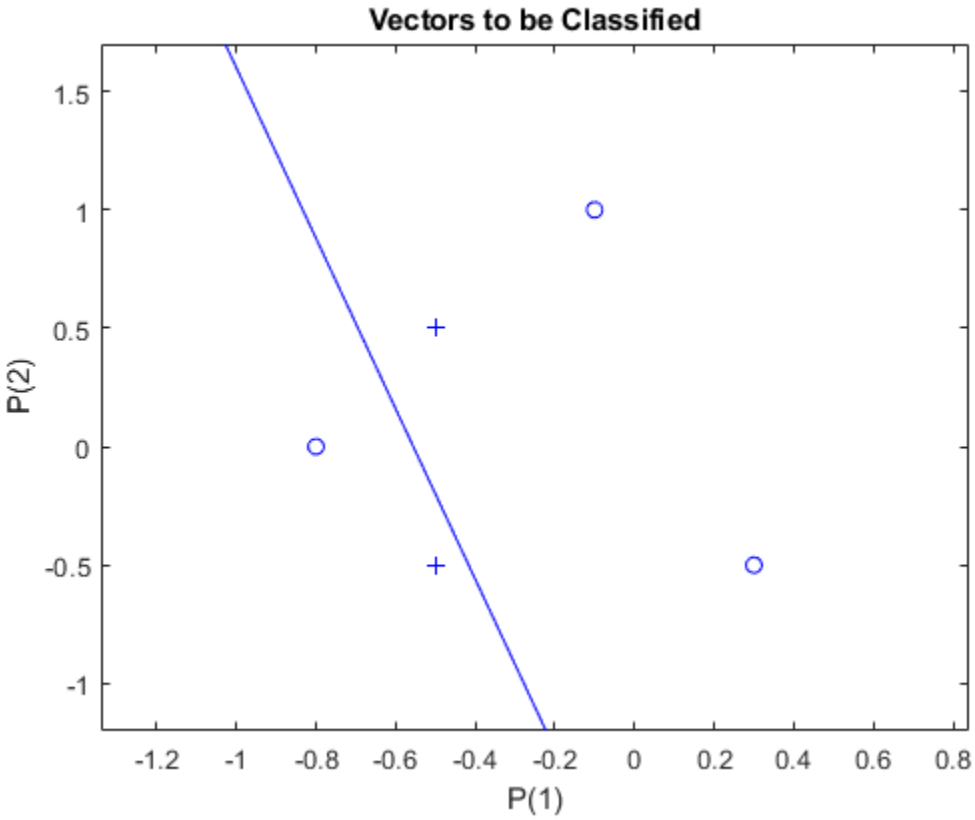
Add the neuron's initial attempt at classification to the plot. The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot.

```
hold on
plotpv(X,T);
linehandle = plotpc(net.IW{1},net.b{1});
```



ADAPT returns a new network after learning on the input and target data, the outputs and error. The loop allows the network to repeatedly adapt, plots the classification line, and stops after 25 iterations.

```
for a = 1:25
   [net,Y,E] = adapt(net,X,T);
   linehandle = plotpc(net.IW{1},net.b{1},linehandle);  drawnow;
end;
```



Note that zero error was never obtained. Despite training, the perceptron has not become an acceptable classifier. Only being able to classify linearly separable data is the fundamental limitation of perceptrons.

Pattern Association Showing Error Surface

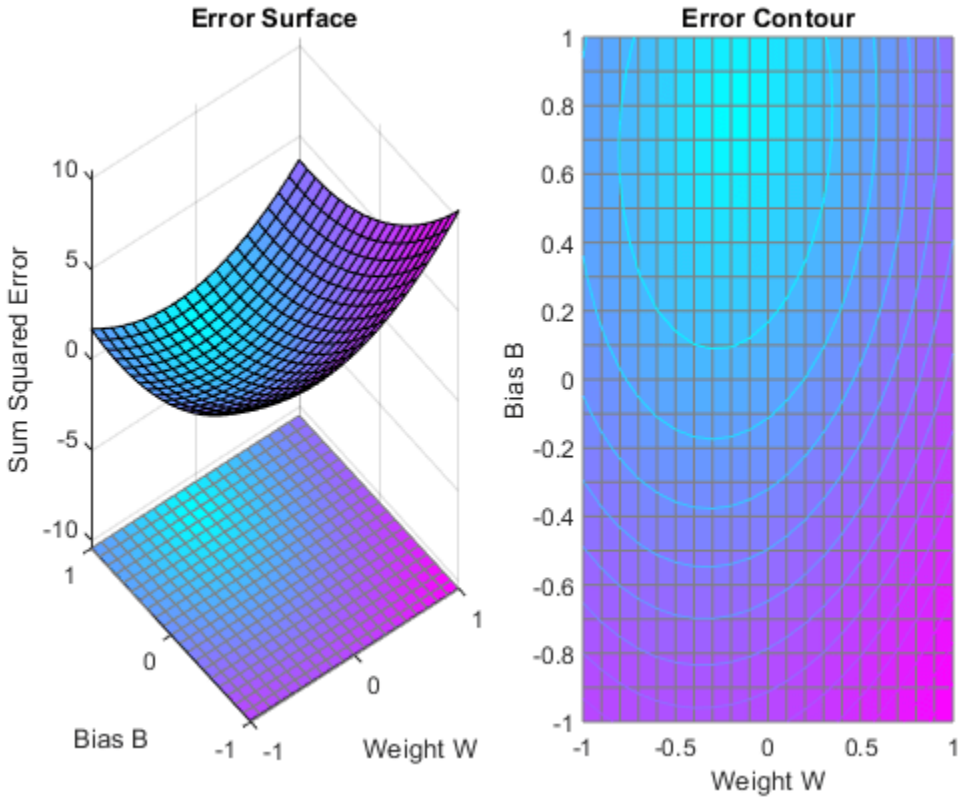
A linear neuron is designed to respond to specific inputs with target outputs.

X defines two 1-element input patterns (column vectors). T defines the associated 1-element targets (column vectors).

```
X = [1.0 -1.2];
T = [0.5 1.0];
```

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath. The best weight and bias values are those that result in the lowest point on the error surface.

```
w_range = -1:0.1:1;
b_range = -1:0.1:1;
ES = errsurf(X,T,w_range,b_range,'purelin');
plotes(w_range,b_range,ES);
```



The function NEWLIND will design y network that performs with the minimum error.

```
net = newlind(X,T);
```

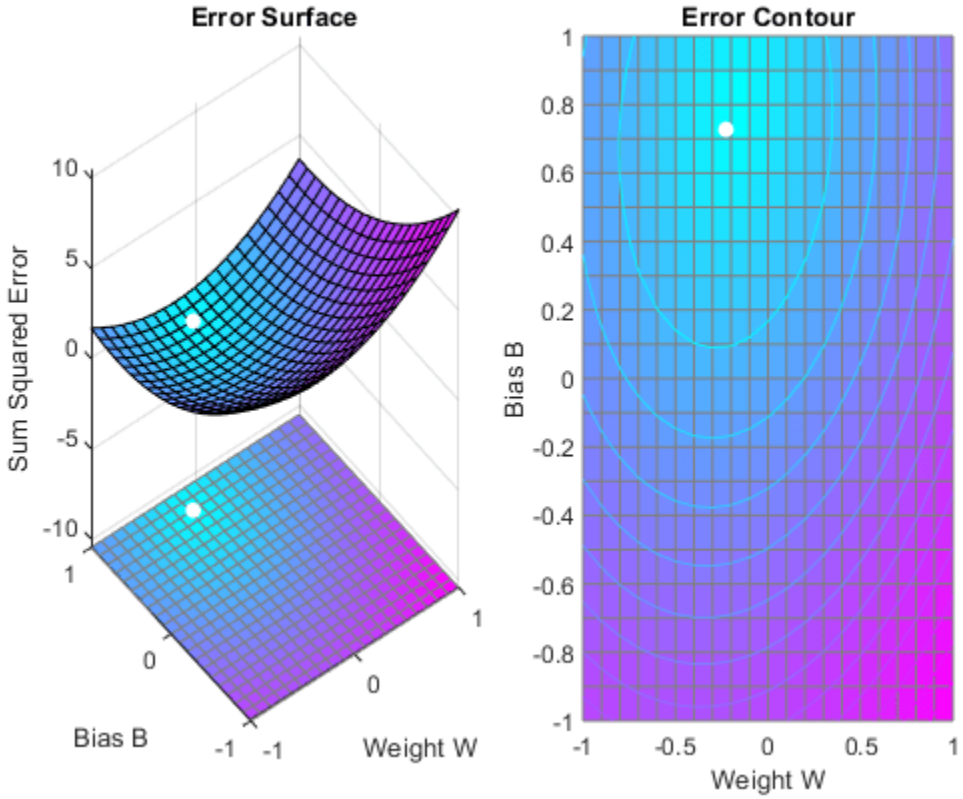
SIM is used to simulate the network for inputs X. We can then calculate the neurons errors. SUMSQR adds up the squared errors.

```
A = net(X)
```

```
A = 1×2

    0.5000    1.0000

E = T - A

E = 1×2

     0     0

SSE = sumsqr(E)

SSE = 0
```

PLOTES replots the error surface. PLOTEP plots the "position" of the network using the weight and bias values returned by SOLVELIN. As can be seen from the plot, SOLVELIN found the minimum error solution.

```
plotes(w_range,b_range,ES);
plotep(net.IW{1,1},net.b{1},SSE);
```



We can now test the associator with one of the original inputs, -1.2, and see if it returns the target, 1.0.

```
x = -1.2;
y = net(x)
```

```
y = 1
```

Training a Linear Neuron

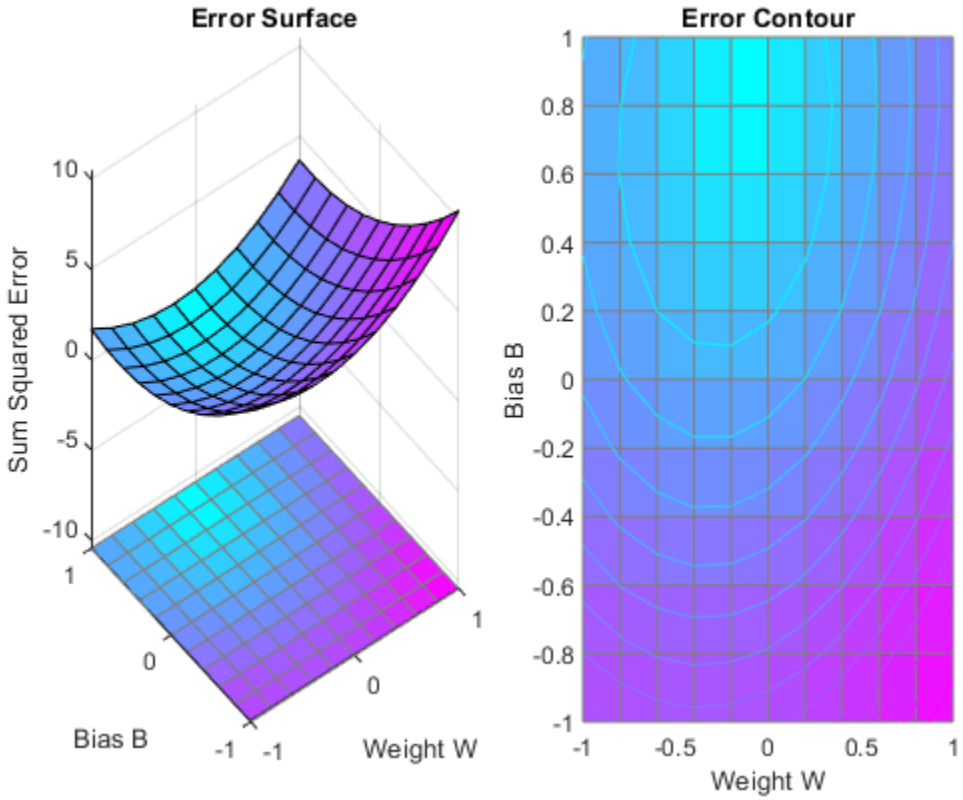
A linear neuron is trained to respond to specific inputs with target outputs.

X defines two 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors). A single input linear neuron with y bias can be used to solve this problem.

```
X = [1.0 -1.2];
T = [0.5 1.0];
```

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath. The best weight and bias values are those that result in the lowest point on the error surface.

```
w_range = -1:0.2:1;  b_range = -1:0.2:1;
ES = errsurf(X,T,w_range,b_range,'purelin');
plotes(w_range,b_range,ES);
```



MAXLINLR finds the fastest stable learning rate for training y linear network. For this example, this rate will only be 40% of this maximum. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

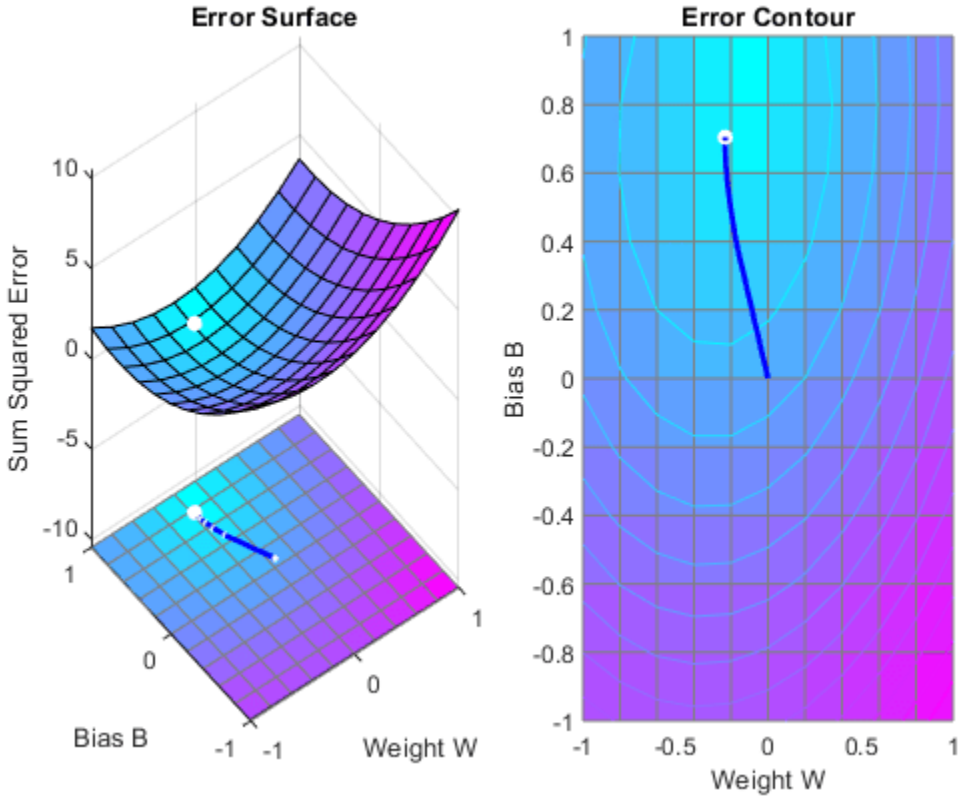
```
maxlr = 0.40*maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

Override the default training parameters by setting the performance goal.

```
net.trainParam.goal = .001;
```

To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch. The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

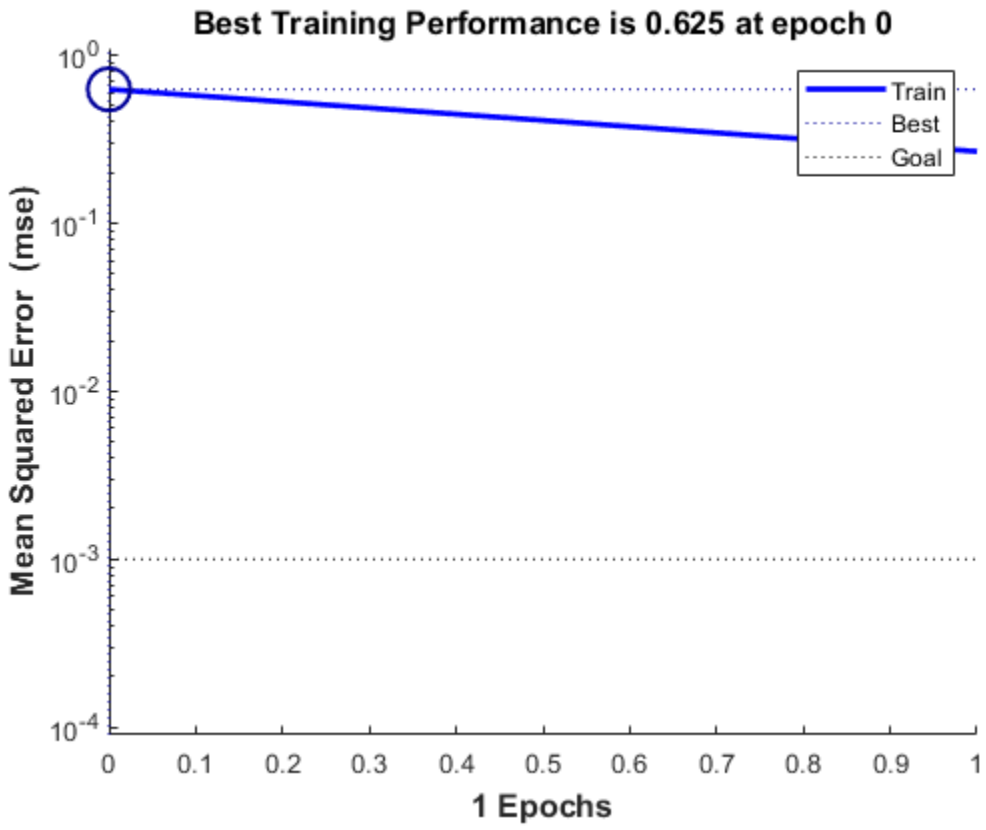
```
% [net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while true
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      r.epoch=[r.epoch epoch];
      r.perf=[r.perf tr.perf(2)];
      r.vperf=[r.vperf NaN];
      r.tperf=[r.tperf NaN];
   else
      break
   end
end
```




```
tr=r;
```

The train function outputs the trained network and y history of the training performance (tr). Here the errors are plotted with respect to training epochs: The error dropped until it fell beneath the error goal (the black line). At that point training stopped.

```
plotperform(tr);
```



Now use SIM to test the associator with one of the original inputs, -1.2, and see if it returns the target, 1.0. The result is very close to 1, the target. This could be made even closer by lowering the performance goal.

```
x = -1.2;
y = net(x)
```

```
y = 0.9817
```

Linear Fit of Nonlinear Problem

A linear neuron is trained to find the minimum sum-squared error linear fit to y nonlinear input/output problem.

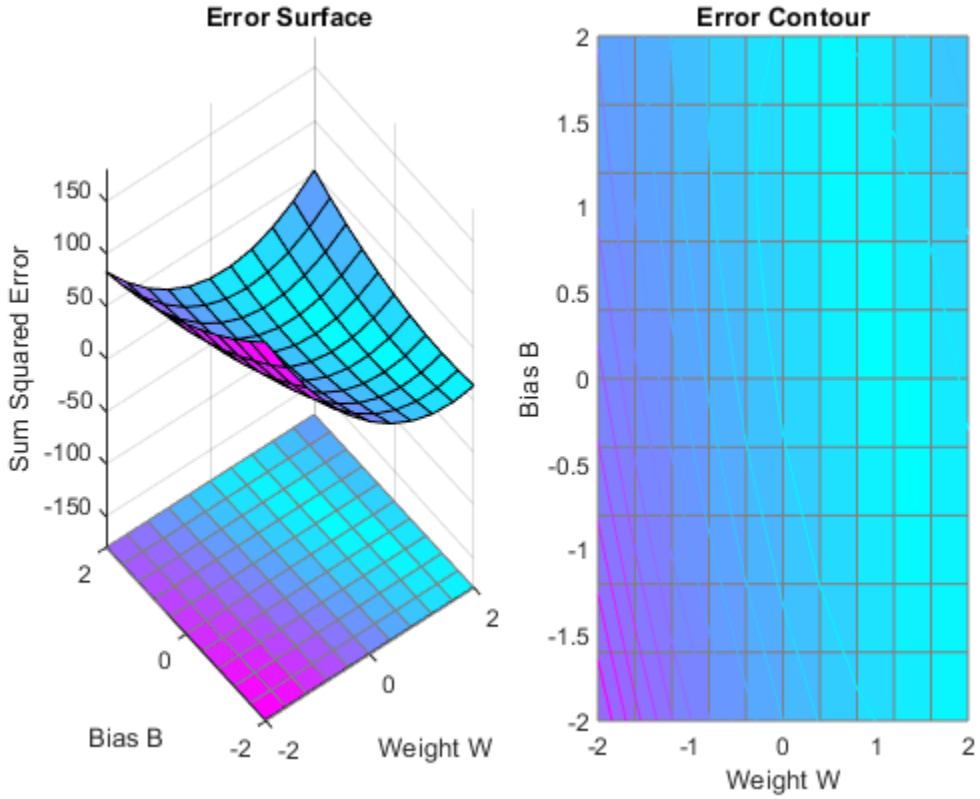
X defines four 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors). Note that the relationship between values in X and in T is nonlinear. I.e. No W and B exist such that X*W+B = T for all of four sets of X and T values above.

```
X = [+1.0 +1.5 +3.0 -1.2];
T = [+0.5 +1.1 +3.0 -1.0];
```

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath.

The best weight and bias values are those that result in the lowest point on the error surface. Note that because y perfect linear fit is not possible, the minimum has an error greater than 0.

```
w_range =-2:0.4:2;  b_range = -2:0.4:2;
ES = errsurf(X,T,w_range,b_range,'purelin');
plotes(w_range,b_range,ES);
```



MAXLINLR finds the fastest stable learning rate for training y linear network. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

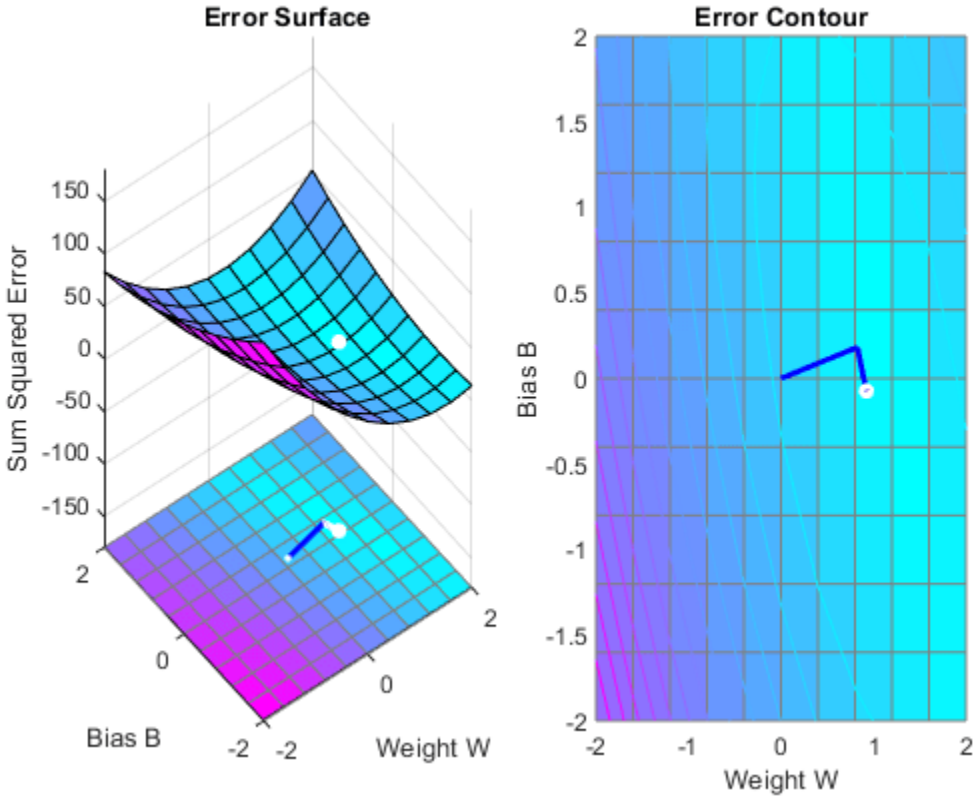
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

Override the default training parameters by setting the maximum number of epochs. This ensures that training will stop.

```
net.trainParam.epochs = 15;
```

To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch (code not shown here). The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while epoch < 15
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      r.epoch=[r.epoch epoch];
      r.perf=[r.perf tr.perf(2)];
      r.vperf=[r.vperf NaN];
      r.tperf=[r.tperf NaN];
   else
      break
   end
end
```

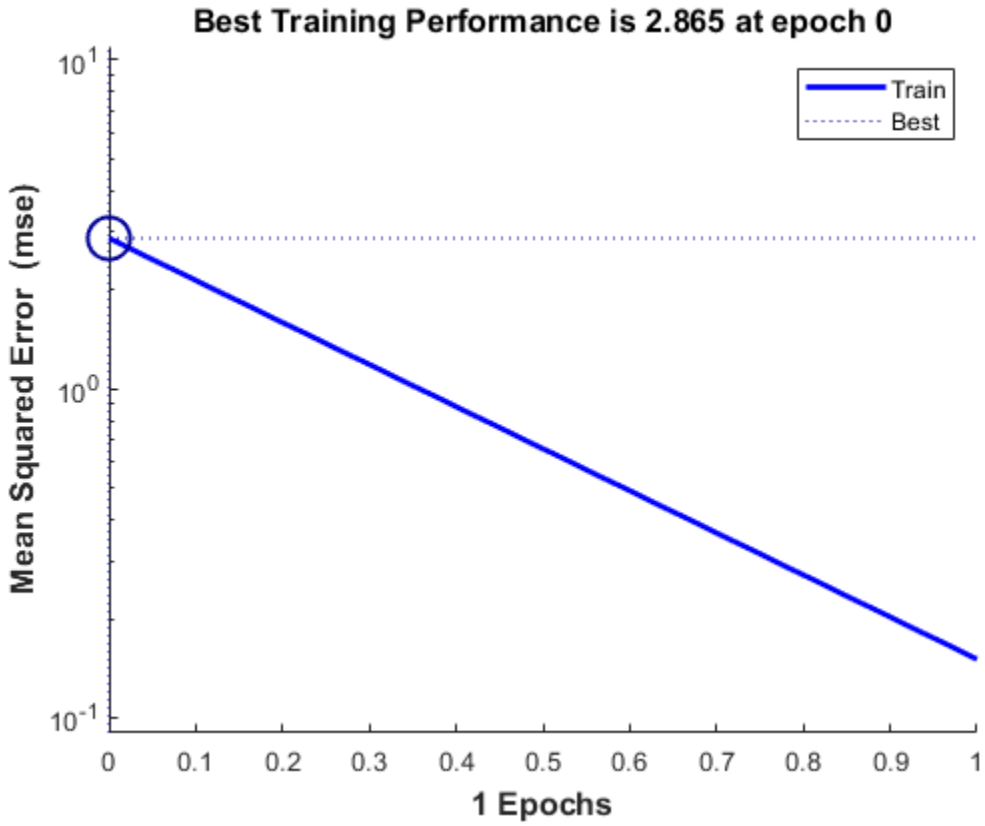



```
tr=r;
```

The train function outputs the trained network and y history of the training performance (tr). Here the errors are plotted with respect to training epochs.

Note that the error never reaches 0. This problem is nonlinear and therefore y zero error linear solution is not possible.

```
plotperform(tr);
```



Now use SIM to test the associator with one of the original inputs, -1.2, and see if it returns the target, 1.0.

The result is not very close to 0.5! This is because the network is the best linear fit to y nonlinear problem.

```
x = -1.2;
y = net(x)
```

```
y = -1.1803
```

Underdetermined Problem

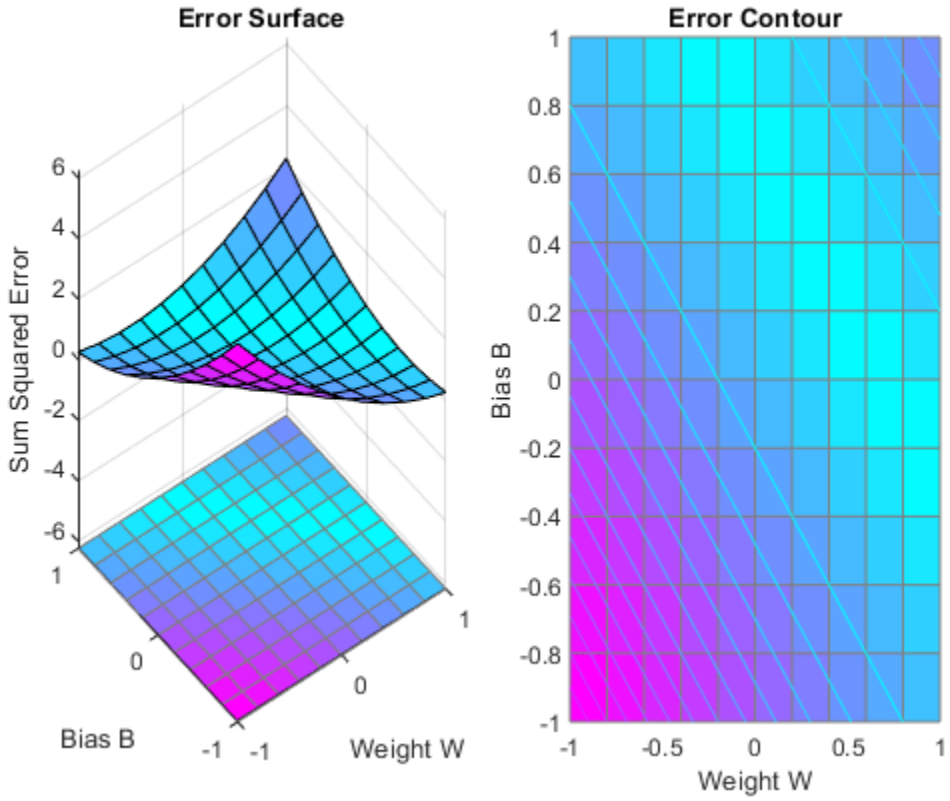
A linear neuron is trained to find y non-unique solution to an undetermined problem.

X defines one 1-element input patterns (column vectors). T defines an associated 1-element target (column vectors). Note that there are infinite values of W and B such that the expression W*X+B = T is true. Problems with multiple solutions are called underdetermined.

```
X = [+1.0];
T = [+0.5];
```

ERRSURF calculates errors for y neuron with y range of possible weight and bias values. PLOTES plots this error surface with y contour plot underneath. The bottom of the valley in the error surface corresponds to the infinite solutions to this problem.

```
w_range = -1:0.2:1;  b_range = -1:0.2:1;
ES = errsurf(X,T,w_range,b_range,'purelin');
plotes(w_range,b_range,ES);
```



MAXLINLR finds the fastest stable learning rate for training y linear network. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

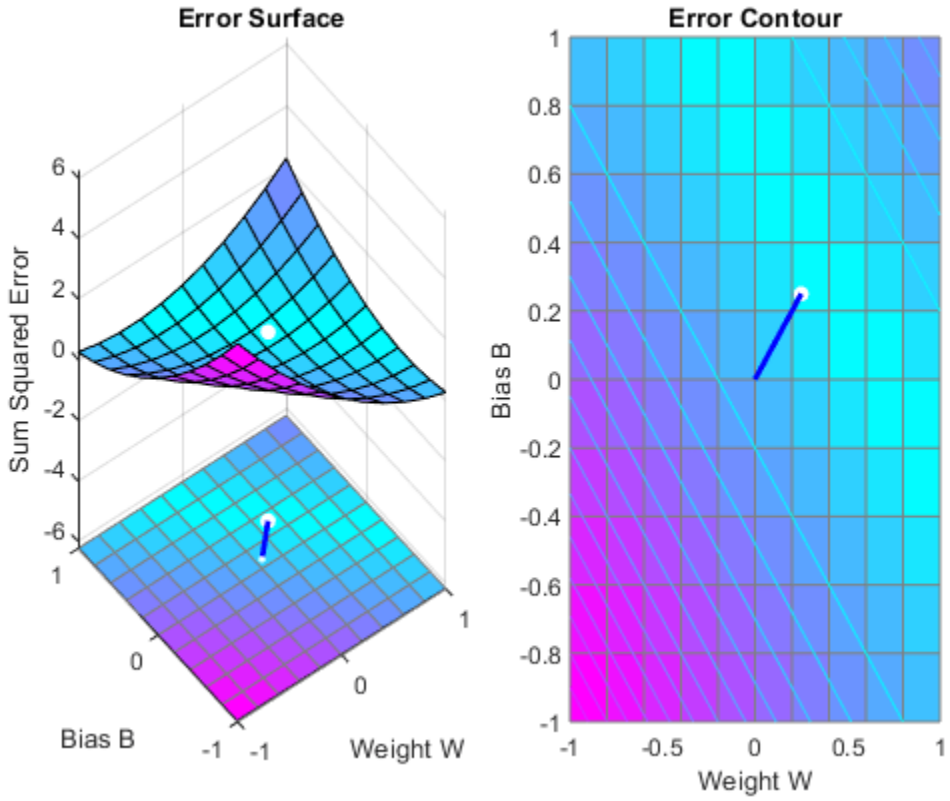
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

Override the default training parameters by setting the performance goal.

```
net.trainParam.goal = 1e-10;
```

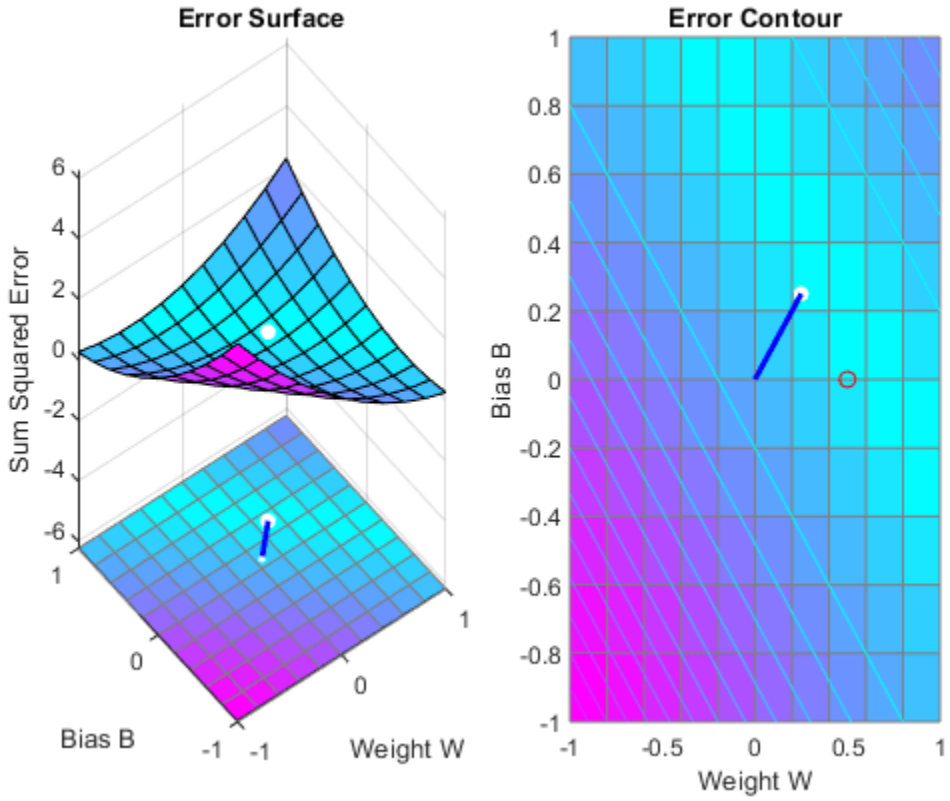
To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch. The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while true
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      r.epoch=[r.epoch epoch];
      r.perf=[r.perf tr.perf(2)];
      r.vperf=[r.vperf NaN];
      r.tperf=[r.tperf NaN];
   else
      break
   end
end
tr=r;
```



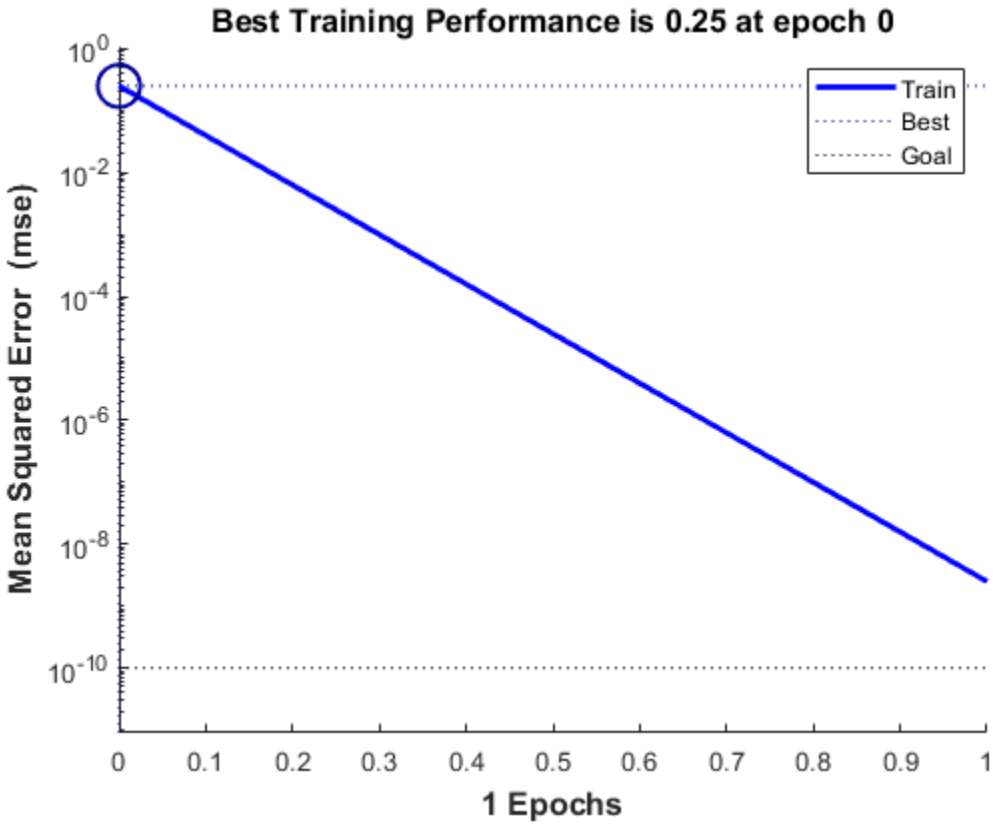
Here we plot the NEWLIND solution. Note that the TRAIN (white dot) and SOLVELIN (red circle) solutions are not the same. In fact, TRAINWH will return y different solution for different initial conditions, while SOLVELIN will always return the same solution.

```
solvednet = newlind(X,T);
hold on;
plot(solvednet.IW{1,1},solvednet.b{1},'ro')
hold off;
```



The train function outputs the trained network and y history of the training performance (tr). Here the errors are plotted with respect to training epochs: Once the error reaches the goal, an adequate solution for W and B has been found. However, because the problem is underdetermined, this solution is not unique.

```
subplot(1,2,1);
plotperform(tr);
```



We can now test the associator with one of the original inputs, 1.0, and see if it returns the target, 0.5. The result is very close to 0.5. The error can be reduced further, if required, by continued training with TRAINWH using y smaller error goal.

```
x = 1.0;
y = net(x)
```

```
y =

    0.5000
```

Linearly Dependent Problem

A linear neuron is trained to find the minimum error solution for y problem with linearly dependent input vectors. If y linear dependence in input vectors is not matched in the target vectors, the problem is nonlinear and does not have y zero error linear solution.

X defines three 2-element input patterns (column vectors). Note that 0.5 times the sum of (column) vectors 1 and 3 results in vector 2. This is called linear dependence.

```
X = [ 1.0   2.0   3.0; ...
      4.0   5.0   6.0];
```

T defines an associated 1-element target (column vectors). Note that 0.5 times the sum of -1.0 and 0.5 does not equal 1.0. Because the linear dependence in X is not matched in T this problem is nonlinear and does not have y zero error linear solution.

```
T = [0.5 1.0 -1.0];
```

MAXLINLR finds the fastest stable learning rate for TRAINWH. NEWLIN creates y linear neuron. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

```
maxlr = maxlinlr(X,'bias');
net = newlin([0 10;0 10],1,[0],maxlr);
```

TRAIN uses the Widrow-Hoff rule to train linear networks by default. We will display each 50 epochs and train for y maximum of 500 epochs.

```
net.trainParam.show = 50;     % Frequency of progress displays (in epochs).
net.trainParam.epochs = 500;  % Maximum number of epochs to train.
net.trainParam.goal = 0.001;  % Sum-squared error goal.
```

Now the network is trained on the inputs X and targets T. Note that, due to the linear dependence between input vectors, the problem did not reach the error goal represented by the black line.

```
[net,tr] = train(net,X,T);
```

We can now test the associator with one of the original inputs, [1; 4] , and see if it returns the target, 0.5. The result is not 0.5 as the linear network could not fit the nonlinear problem caused by the linear dependence between input vectors.

```
p = [1.0; 4];
y = net(p)
```

```
y = 0.8971
```

Too Large a Learning Rate

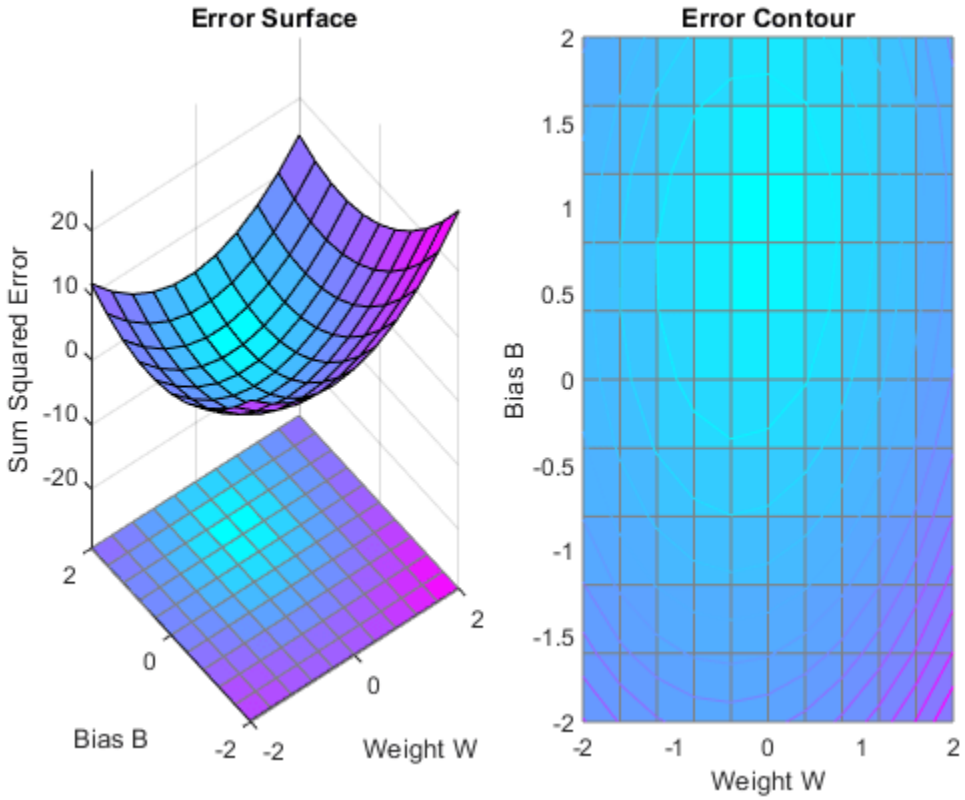
A linear neuron is trained to find the minimum error solution for a simple problem. The neuron is trained with the learning rate larger than the one suggested by MAXLINLR.

X defines two 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors).

```
X = [+1.0 -1.2];
T = [+0.5 +1.0];
```

ERRSURF calculates errors for a neuron with a range of possible weight and bias values. PLOTES plots this error surface with a contour plot underneath. The best weight and bias values are those that result in the lowest point on the error surface.

```
w_range = -2:0.4:2;
b_range = -2:0.4:2;
ES = errsurf(X,T,w_range,b_range,'purelin');
plotes(w_range,b_range,ES);
```



MAXLINLR finds the fastest stable learning rate for training a linear network. NEWLIN creates a linear neuron. To see what happens when the learning rate is too large, increase the learning rate to 225% of the recommended value. NEWLIN takes these arguments: 1) Rx2 matrix of min and max values for R input elements, 2) Number of elements in the output vector, 3) Input delay vector, and 4) Learning rate.

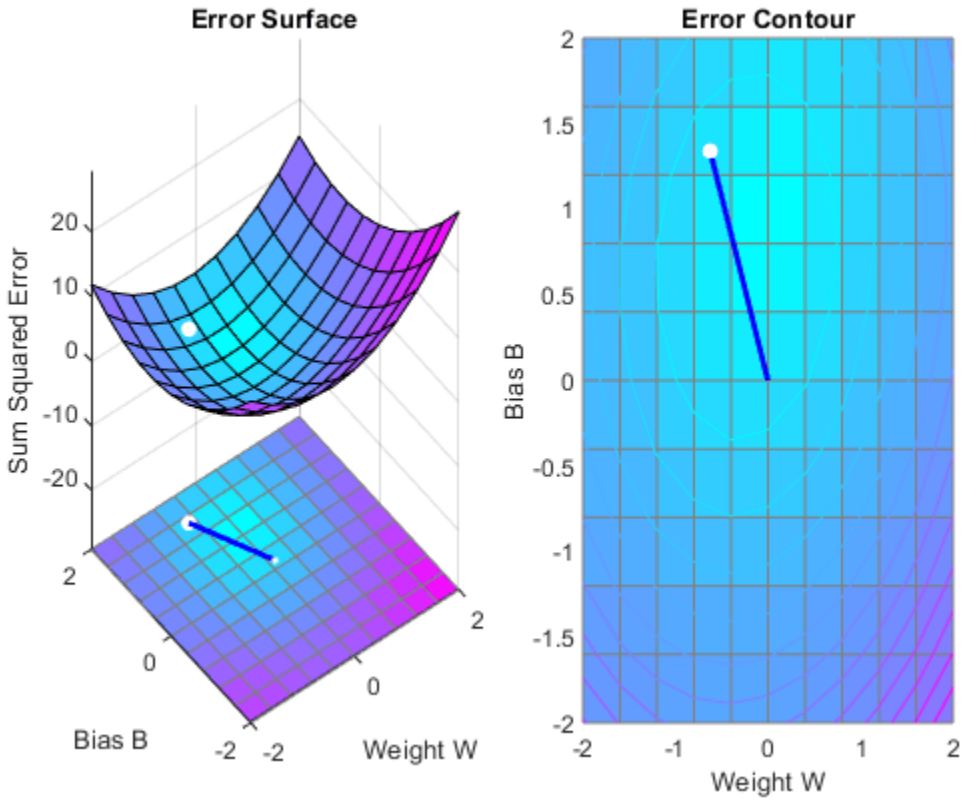
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr*2.25);
```

Override the default training parameters by setting the maximum number of epochs. This ensures that training will stop:

```
net.trainParam.epochs = 20;
```

To show the path of the training we will train only one epoch at a time and call PLOTEP every epoch (code not shown here). The plot shows a history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

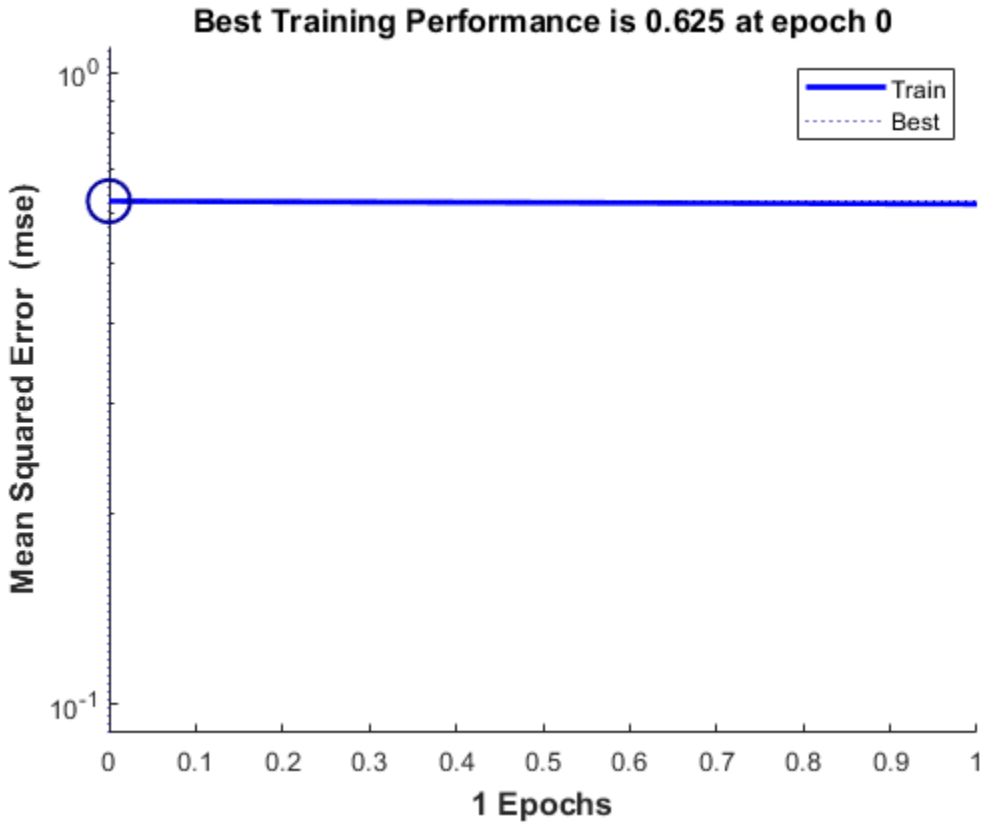
```
%[net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while epoch < 20
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      r.epoch=[r.epoch epoch];
      r.perf=[r.perf tr.perf(2)];
      r.vperf=[r.vperf NaN];
      r.tperf=[r.tperf NaN];
   else
      break
   end
end
```




```
tr=r;
```

The train function outputs the trained network and a history of the training performance (tr). Here the errors are plotted with respect to training epochs.

```
plotperform(tr);
```



We can now use SIM to test the associator with one of the original inputs, -1.2, and see if it returns the target, 1.0. The result is not very close to 0.5! This is because the network was trained with too large a learning rate.

```
x = -1.2;
y = net(x)
```

```
y = 2.0913
```

Adaptive Noise Cancellation

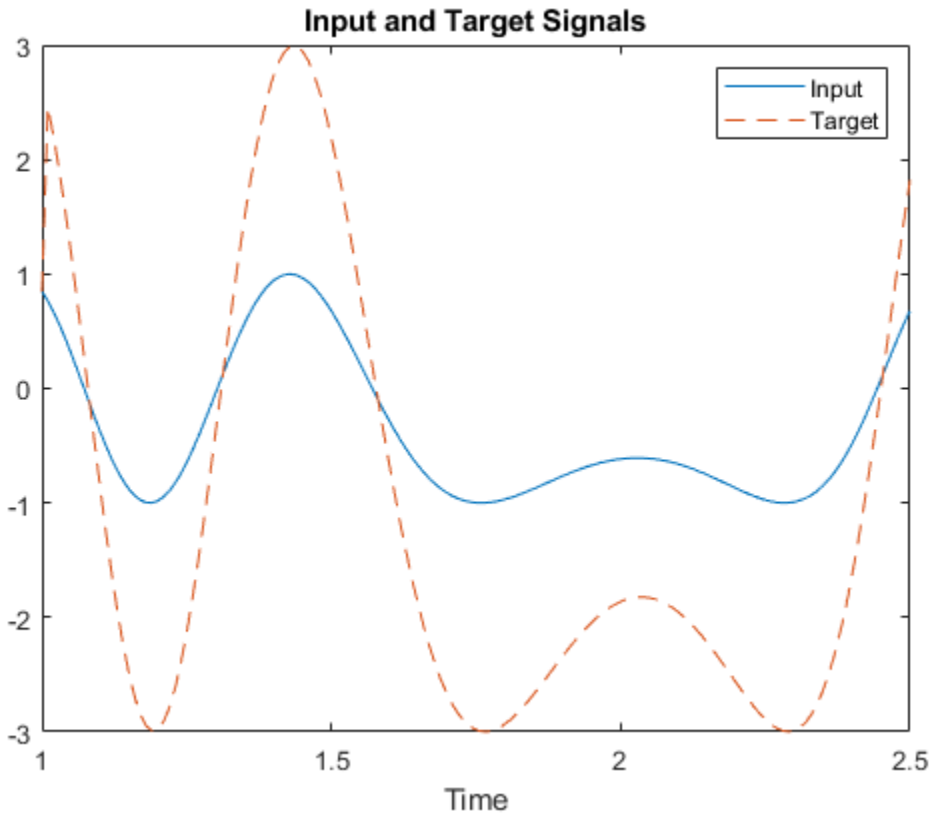
A linear neuron is allowed to adapt so that given one signal, it can predict a second signal.

TIME defines the time steps of this simulation. P defines a signal over these time steps. T is a signal derived from P by shifting it to the left, multiplying it by 2 and adding it to itself.

```
time = 1:0.01:2.5;
X = sin(sin(time).*time*10);
P = con2seq(X);
T = con2seq(2*[0 X(1:(end-1))] + X);
```

Here is how the two signals are plotted:

```
plot(time,cat(2,P{:}),time,cat(2,T{:}),'--')
title('Input and Target Signals')
xlabel('Time')
legend({'Input','Target'})
```



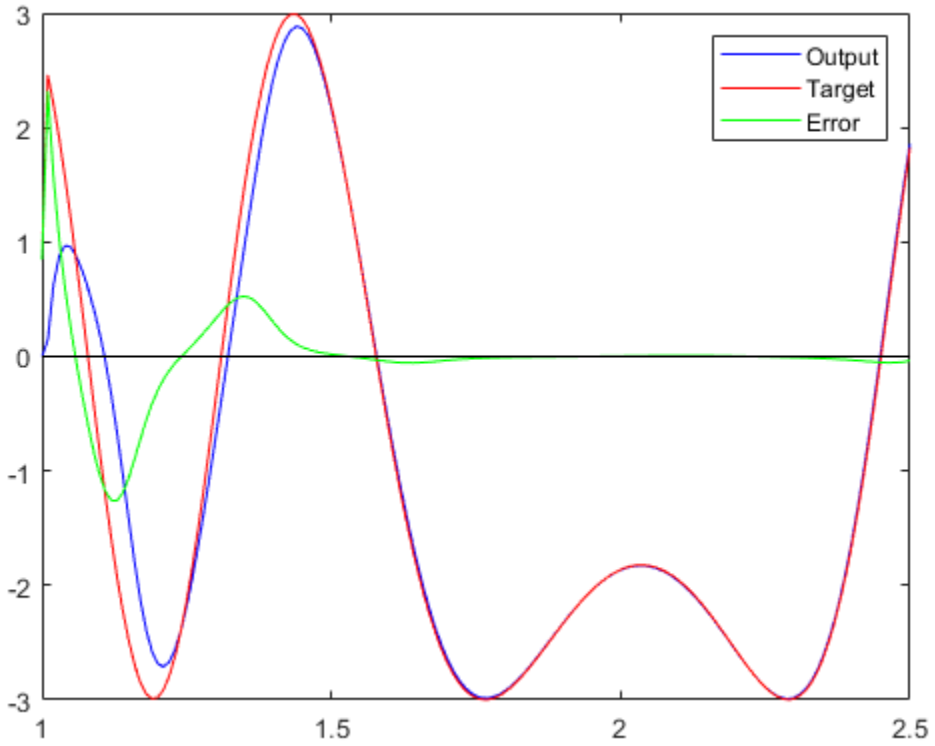
The linear network must have tapped delay in order to learn the time-shifted correlation between P and T. NEWLIN creates a linear layer. [-3 3] is the expected input range. The second argument is the number of neurons in the layer. [0 1] specifies one input with no delay and one input with a delay of one. The last argument is the learning rate.

```
net = newlin([-3 3],1,[0 1],0.1);
```

ADAPT simulates adaptive networks. It takes a network, a signal, and a target signal, and filters the signal adaptively. Plot the output Y in blue, the target T in red and the error E in green. By t=2 the

network has learned the relationship between the input and the target and the error drops to near zero.

```
[net,Y,E,Pf]=adapt(net,P,T);
plot(time,cat(2,Y{:}),'b', ...
   time,cat(2,T{:}),'r', ...
   time,cat(2,E{:}),'g',[1 2.5],[0 0],'k')
legend({'Output','Target','Error'})
```



Shallow Neural Networks Bibliography

Shallow Neural Networks Bibliography

[Batt92] Battiti, R., “First and second order methods for learning: Between steepest descent and Newton's method,” *Neural Computation*, Vol. 4, No. 2, 1992, pp. 141–166.

[Beal72] Beale, E.M.L., “A derivation of conjugate gradients,” in F.A. Lootsma, Ed., *Numerical methods for nonlinear optimization*, London: Academic Press, 1972.

[Bren73] Brent, R.P., *Algorithms for Minimization Without Derivatives*, Englewood Cliffs, NJ: Prentice-Hall, 1973.

[Caud89] Caudill, M., *Neural Networks Primer*, San Francisco, CA: Miller Freeman Publications, 1989.

This collection of papers from the *AI Expert Magazine* gives an excellent introduction to the field of neural networks. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[CaBu92] Caudill, M., and C. Butler, *Understanding Neural Networks: Computer Explorations, Vols. 1 and 2*, Cambridge, MA: The MIT Press, 1992.

This is a two-volume workbook designed to give students “hands on” experience with neural networks. It is written for a laboratory course at the senior or first-year graduate level. Software for IBM PC and Apple Macintosh computers is included. The material is well written, clear, and helpful in understanding a field that traditionally has been buried in mathematics.

[Char92] Charalambous, C.,“Conjugate gradient algorithm for efficient training of artificial neural networks,” *IEEE Proceedings*, Vol. 139, No. 3, 1992, pp. 301–310.

[ChCo91] Chen, S., C.F.N. Cowan, and P.M. Grant, “Orthogonal least squares learning algorithm for radial basis function networks,” *IEEE Transactions on Neural Networks*, Vol. 2, No. 2, 1991, pp. 302–309.

This paper gives an excellent introduction to the field of radial basis functions. The papers use a minimum of mathematics to explain the main results clearly. Several good suggestions for further reading are included.

[ChDa99] Chengyu, G., and K. Danai, “Fault diagnosis of the IFAC Benchmark Problem with a model-based recurrent neural network,” *Proceedings of the 1999 IEEE International Conference on Control Applications*, Vol. 2, 1999, pp. 1755–1760.

[DARP88] *DARPA Neural Network Study*, Lexington, MA: M.I.T. Lincoln Laboratory, 1988.

This book is a compendium of knowledge of neural networks as they were known to 1988. It presents the theoretical foundations of neural networks and discusses their current applications. It contains sections on associative memories, recurrent networks, vision, speech recognition, and robotics. Finally, it discusses simulation tools and implementation technology.

[DeHa01a] De Jesús, O., and M.T. Hagan, “Backpropagation Through Time for a General Class of Recurrent Network,” *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2638–2642.

[DeHa01b] De Jesús, O., and M.T. Hagan, “Forward Perturbation Algorithm for a General Class of Recurrent Network,” *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2626–2631.

[DeHa07] De Jesús, O., and M.T. Hagan, "Backpropagation Algorithms for a Broad Class of Dynamic Networks," IEEE Transactions on Neural Networks, Vol. 18, No. 1, January 2007, pp. 14 -27.

This paper provides detailed algorithms for the calculation of gradients and Jacobians for arbitrarily-connected neural networks. Both the backpropagation-through-time and real-time recurrent learning algorithms are covered.

[DeSc83] Dennis, J.E., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[DHH01] De Jesús, O., J.M. Horn, and M.T. Hagan, "Analysis of Recurrent Network Training and Suggestions for Improvements," *Proceedings of the International Joint Conference on Neural Networks*, Washington, DC, July 15–19, 2001, pp. 2632–2637.

[Elma90] Elman, J.L., "Finding structure in time," *Cognitive Science*, Vol. 14, 1990, pp. 179–211.

This paper is a superb introduction to the Elman networks described in Chapter 10, "Recurrent Networks."

[FeTs03] Feng, J., C.K. Tse, and F.C.M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, Vol. 50, No. 7, 2003, pp. 954–957.

[FlRe64] Fletcher, R., and C.M. Reeves, "Function minimization by conjugate gradients," *Computer Journal*, Vol. 7, 1964, pp. 149–154.

[FoHa97] Foresee, F.D., and M.T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Joint Conference on Neural Networks*, 1997, pp. 1930–1935.

[GiMu81] Gill, P.E., W. Murray, and M.H. Wright, *Practical Optimization*, New York: Academic Press, 1981.

[GiPr02] Gianluca, P., D. Przybylski, B. Rost, P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," *Proteins: Structure, Function, and Genetics*, Vol. 47, No. 2, 2002, pp. 228–235.

[Gros82] Grossberg, S., *Studies of the Mind and Brain*, Drodrecht, Holland: Reidel Press, 1982.

This book contains articles summarizing Grossberg's theoretical psychophysiology work up to 1980. Each article contains a preface explaining the main points.

[HaDe99] Hagan, M.T., and H.B. Demuth, "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642–1656.

[HaJe99] Hagan, M.T., O. De Jesus, and R. Schultz, "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, pp. 311–340.

[HaMe94] Hagan, M.T., and M. Menhaj, "Training feed-forward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1999, pp. 989–993, 1994.

This paper reports the first development of the Levenberg-Marquardt algorithm for neural networks. It describes the theory and application of the algorithm, which trains neural networks at a rate 10 to 100 times faster than the usual gradient descent backpropagation method.

[HaRu78] Harrison, D., and Rubinfeld, D.L., “Hedonic prices and the demand for clean air,” *J. Environ. Economics & Management*, Vol. 5, 1978, pp. 81-102.

This data set was taken from the StatLib library, which is maintained at Carnegie Mellon University.

[HDB96] Hagan, M.T., H.B. Demuth, and M.H. Beale, *Neural Network Design*, Boston, MA: PWS Publishing, 1996.

This book provides a clear and detailed survey of basic neural network architectures and learning rules. It emphasizes mathematical analysis of networks, methods of training networks, and application of networks to practical engineering problems. It has example programs, an instructor’s guide, and transparency overheads for teaching.

[HDH09] Horn, J.M., O. De Jesús and M.T. Hagan, “Spurious Valleys in the Error Surface of Recurrent Networks - Analysis and Avoidance,” IEEE Transactions on Neural Networks, Vol. 20, No. 4, pp. 686-700, April 2009.

This paper describes spurious valleys that appear in the error surfaces of recurrent networks. It also explains how training algorithms can be modified to avoid becoming stuck in these valleys.

[Hebb49] Hebb, D.O., *The Organization of Behavior*, New York: Wiley, 1949.

This book proposed neural network architectures and the first learning rule. The learning rule is used to form a theory of how collections of cells might form a concept.

[Himm72] Himmelblau, D.M., *Applied Nonlinear Programming*, New York: McGraw-Hill, 1972.

[HuSb92] Hunt, K.J., D. Sbarbaro, R. Zbikowski, and P.J. Gawthrop, Neural Networks for Control System — A Survey,” *Automatica*, Vol. 28, 1992, pp. 1083–1112.

[JaRa04] Jayadeva and S.A.Rahman, “A neural network with O(N) neurons for ranking N numbers in O(1/N) time,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 51, No. 10, 2004, pp. 2044–2051.

[Joll86] Jolliffe, I.T., *Principal Component Analysis*, New York: Springer-Verlag, 1986.

[KaGr96] Kamwa, I., R. Grondin, V.K. Sood, C. Gagnon, Van Thich Nguyen, and J. Mereb, “Recurrent neural networks for phasor detection and adaptive identification in power system control and protection,” *IEEE Transactions on Instrumentation and Measurement*, Vol. 45, No. 2, 1996, pp. 657–664.

[Koho87] Kohonen, T., *Self-Organization and Associative Memory*, *2nd Edition*, Berlin: Springer-Verlag, 1987.

This book analyzes several learning rules. The Kohonen learning rule is then introduced and embedded in self-organizing feature maps. Associative networks are also studied.

[Koho97] Kohonen, T., *Self-Organizing Maps*, Second Edition, Berlin: Springer-Verlag, 1997.

This book discusses the history, fundamentals, theory, applications, and hardware of self-organizing maps. It also includes a comprehensive literature survey.

[LiMi89] Li, J., A.N. Michel, and W. Porod, “Analysis and synthesis of a class of neural networks: linear systems operating on a closed hypercube,” *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 11, 1989, pp. 1405–1422.

This paper discusses a class of neural networks described by first-order linear differential equations that are defined on a closed hypercube. The systems considered retain the basic structure of the Hopfield model but are easier to analyze and implement. The paper presents an efficient method for determining the set of asymptotically stable equilibrium points and the set of unstable equilibrium points. Examples are presented. The method of Li, et. al., is implemented in Advanced Topics in the *User's Guide*.

[Lipp87] Lippman, R.P., "An introduction to computing with neural nets," *IEEE ASSP Magazine*, 1987, pp. 4–22.

This paper gives an introduction to the field of neural nets by reviewing six neural net models that can be used for pattern classification. The paper shows how existing classification and clustering algorithms can be performed using simple components that are like neurons. This is a highly readable paper.

[MacK92] MacKay, D.J.C., "Bayesian interpolation," *Neural Computation*, Vol. 4, No. 3, 1992, pp. 415–447.

[Marq63] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

[McPi43] McCulloch, W.S., and W.H. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, 1943, pp. 115–133.

A classic paper that describes a model of a neuron that is binary and has a fixed threshold. A network of such neurons can perform logical operations.

[MeJa00] Medsker, L.R., and L.C. Jain, *Recurrent neural networks: design and applications*, Boca Raton, FL: CRC Press, 2000.

[Moll93] Moller, M.F., "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, Vol. 6, 1993, pp. 525–533.

[MuNe92] Murray, R., D. Neumerkel, and D. Sbarbaro, "Neural Networks for Modeling and Control of a Non-linear Dynamic System," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, 1992, pp. 404–409.

[NaMu97] Narendra, K.S., and S. Mukhopadhyay, "Adaptive Control Using Neural Networks and Approximate Models," *IEEE Transactions on Neural Networks*, Vol. 8, 1997, pp. 475–485.

[NaPa91] Narendra, Kumpati S. and Kannan Parthasarathy, "Learning Automata Approach to Hierarchical Multiobjective Analysis," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, No. 1, January/February 1991, pp. 263–272.

[NgWi89] Nguyen, D., and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, 1989, pp. 357–363.

This paper describes a two-layer network that first learned the truck dynamics and then learned how to back the truck to a specified position at a loading dock. To do this, the neural network had to solve a highly nonlinear control systems problem.

[NgWi90] Nguyen, D., and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1990, pp. 21–26.

Nguyen and Widrow show that a two-layer sigmoid/linear network can be viewed as performing a piecewise linear approximation of any learned function. It is shown that weights and biases generated with certain constraints result in an initial network better able to form a function approximation of an arbitrary function. Use of the Nguyen-Widrow (instead of purely random) initial conditions often shortens training time by more than an order of magnitude.

[Powe77] Powell, M.J.D., "Restart procedures for the conjugate gradient method," *Mathematical Programming*, Vol. 12, 1977, pp. 241–254.

[Pulu92] Purdie, N., E.A. Lucas, and M.B. Talley, "Direct measure of total cholesterol and its distribution among major serum lipoproteins," *Clinical Chemistry*, Vol. 38, No. 9, 1992, pp. 1645–1647.

[RiBr93] Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, 1993.

[Robin94] Robinson, A.J., "An application of recurrent nets to phone probability estimation," *IEEE Transactions on Neural Networks*, Vol. 5 , No. 2, 1994.

[RoJa96] Roman, J., and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, Vol. 2, 1996, pp. 454–460.

[Rose61] Rosenblatt, F., *Principles of Neurodynamics*, Washington, D.C.: Spartan Press, 1961.

This book presents all of Rosenblatt's results on perceptrons. In particular, it presents his most important result, the *perceptron learning theorem*.

[RuHi86a] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," in D.E. Rumelhart and J.L. McClelland, Eds., *Parallel Data Processing*, *Vol. 1*, Cambridge, MA: The M.I.T. Press, 1986, pp. 318–362.

This is a basic reference on backpropagation.

[RuHi86b] Rumelhart, D.E., G.E. Hinton, and R.J. Williams, "Learning representations by back-propagating errors," *Nature*, Vol. 323, 1986, pp. 533–536.

[RuMc86] Rumelhart, D.E., J.L. McClelland, and the PDP Research Group, Eds., *Parallel Distributed Processing*, *Vols. 1 and 2*, Cambridge, MA: The M.I.T. Press, 1986.

These two volumes contain a set of monographs that present a technical introduction to the field of neural networks. Each section is written by different authors. These works present a summary of most of the research in neural networks to the date of publication.

[Scal85] Scales, L.E., *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[SoHa96] Soloway, D., and P.J. Haley, "Neural Generalized Predictive Control," *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996, pp. 277–281.

[VoMa88] Vogl, T.P., J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics*, Vol. 59, 1988, pp. 256–264.

Backpropagation learning can be speeded up and made less sensitive to small features in the error surface such as shallow local minima by combining techniques such as batching, adaptive learning rate, and momentum.

[WaHa89] Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 37, 1989, pp. 328–339.

[Wass93] Wasserman, P.D., *Advanced Methods in Neural Computing*, New York: Van Nostrand Reinhold, 1993.

[WeGe94] Weigend, A. S., and N. A. Gershenfeld, eds., *Time Series Prediction: Forecasting the Future and Understanding the Past*, Reading, MA: Addison-Wesley, 1994.

[WiHo60] Widrow, B., and M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record, New York IRE*, 1960, pp. 96–104.

[WiSt85] Widrow, B., and S.D. Sterns, *Adaptive Signal Processing*, New York: Prentice-Hall, 1985.

This is a basic paper on adaptive signal processing.

A

Mathematical Notation

Mathematics and Code Equivalents

In this section...

The transition from mathematics to code or vice versa can be made with the aid of a few rules. They are listed here for reference.

Mathematics Notation to MATLAB Notation

To change from mathematics notation to MATLAB notation:

- Change superscripts to cell array indices. For example,

  $$p^1 \rightarrow p\{1\}$$

- Change subscripts to indices within parentheses. For example,

  $$p_2 \rightarrow p(2)$$

  and

  $$p_2^1 \rightarrow p\{1\}(2)$$

- Change indices within parentheses to a second cell array index. For example,

  $$p^1(k-1) \rightarrow p\{1, k-1\}$$

- Change mathematics operators to MATLAB operators and toolbox functions. For example,

  $$ab \rightarrow a * b$$

Figure Notation

The following equations illustrate the notation used in figures.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + 2_{1,R}p_R + b$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,R} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,R} \\ w_{S,1} & w_{S,2} & \ldots & w_{S,R} \end{bmatrix}$$

B

Neural Network Blocks for the Simulink Environment

Neural Network Simulink Block Library

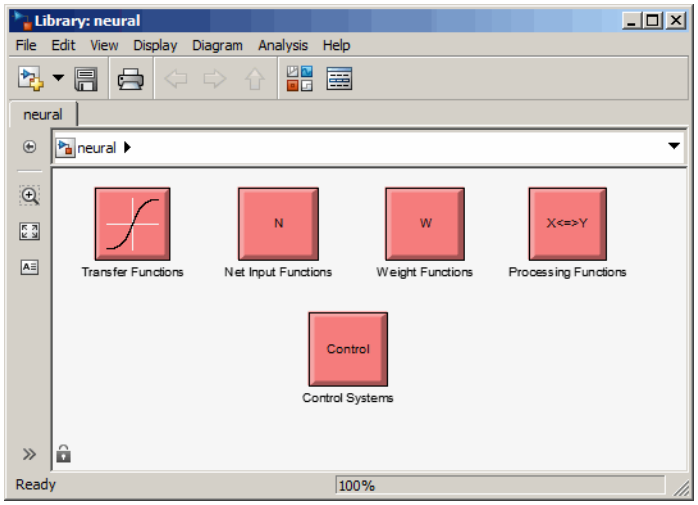
In this section...

The Deep Learning Toolbox product provides a set of blocks you can use to build neural networks using Simulink software, or that the function `gensim` can use to generate the Simulink version of any network you have created using MATLAB software.

Open the Deep Learning Toolbox block library with the command:

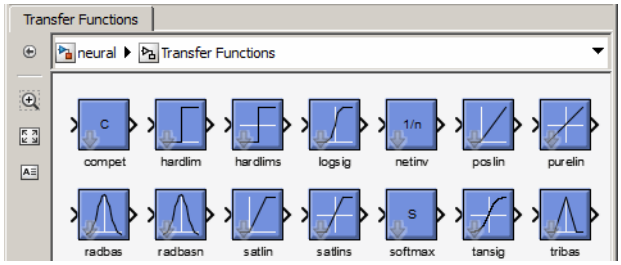
```
neural
```

This opens a library window that contains five blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

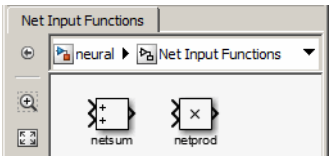
Double-click the Transfer Functions block in the Neural library window to open a window containing several transfer function blocks.



Each of these blocks takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector.

Net Input Blocks

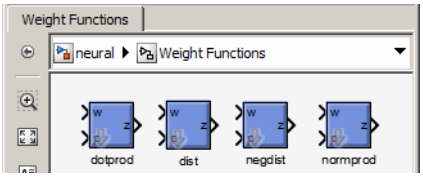
Double-click the Net Input Functions block in the Neural library window to open a window containing two net-input function blocks.



Each of these blocks takes any number of weighted input vectors, weight layer output vectors, and bias vectors, and returns a net-input vector.

Weight Blocks

Double-click the Weight Functions block in the Neural library window to open a window containing three weight function blocks.



Each of these blocks takes a neuron's weight vector and applies it to an input vector (or a layer output vector) to get a weighted input value for a neuron.

It is important to note that these blocks expect the neuron's weight vector to be defined as a column vector. This is because Simulink signals can be column vectors, but cannot be matrices or row vectors.

It is also important to note that because of this limitation you have to create *S* weight function blocks (one for each row), to implement a weight matrix going to a layer with *S* neurons.

This contrasts with the other two kinds of blocks. Only one net input function and one transfer function block are required for each layer.

Processing Blocks

Double-click the Processing Functions block in the Neural library window to open a window containing processing blocks and their corresponding reverse-processing blocks.

Each of these blocks can be used to preprocess inputs and postprocess outputs.

Deploy Shallow Neural Network Simulink Diagrams

In this section...

The function `gensim` generates block descriptions of networks so you can simulate them using Simulink software.

```
gensim(net,st)
```

The second argument to `gensim` determines the sample time, which is normally chosen to be some positive real value.

If a network has no delays associated with its input weights or layer weights, this value can be set to -1. A value of -1 causes `gensim` to generate a network with continuous sampling.

Example

Here is a simple problem defining a set of inputs `p` and corresponding targets `t`.

```
p = [1 2 3 4 5];
t = [1 3 5 7 9];
```

The code below designs a linear layer to solve this problem.

```
net = newlind(p,t)
```

You can test the network on your original inputs with `sim`.

```
y = sim(net,p)
```

The results show the network has solved the problem.

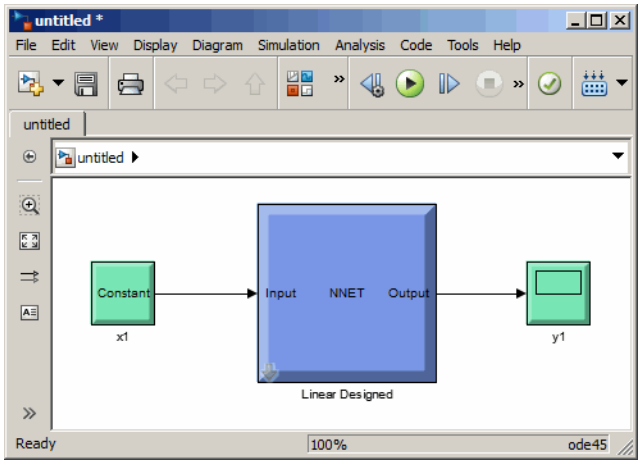
```
y =
      1.0000    3.0000    5.0000    7.0000    9.0000
```

Call `gensim` as follows to generate a Simulink version of the network.

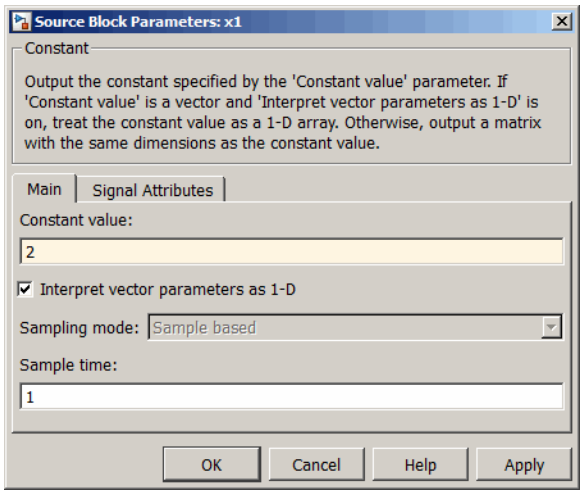
```
gensim(net,-1)
```

The second argument is -1, so the resulting network block samples continuously.

The call to `gensim` opens the following Simulink Editor, showing a system consisting of the linear network connected to a sample input and a scope.



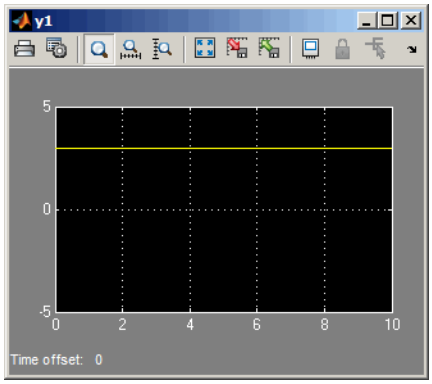
To test the network, double-click the input Constant `x1` block on the left.



The input block is actually a standard Constant block. Change the constant value from the initial randomly generated value to 2, and then click **OK**.

Select the menu option **Simulation** > **Run**. Simulink takes a moment to simulate the system.

When the simulation is complete, double-click the output `y1` block on the right to see the following display of the network's response.



Note that the output is 3, which is the correct output for an input of 2.

Suggested Exercises

Here are a couple exercises you can try.

Change the Input Signal

Replace the constant input block with a signal generator from the standard Simulink Sources blockset. Simulate the system and view the network's response.

Use a Discrete Sample Time

Recreate the network, but with a discrete sample time of 0.5, instead of continuous sampling.

```
gensim(net,0.5)
```

Again, replace the constant input with a signal generator. Simulate the system and view the network's response.

Generate Functions and Objects

For information on simulating and deploying shallow neural networks with MATLAB functions, see "Deploy Shallow Neural Network Functions" on page 28-48.

Code Notes

Deep Learning Toolbox Data Conventions

In this section...

Dimensions

The following code dimensions are used in describing both the network signals that users commonly see, and those used by the utility functions:

`Ni` = Number of network inputs	`= net.numInputs`
`Ri` = Number of elements in input `i`	`= net.inputs{i}.size`
`Nl` = Number of layers	`= net.numLayers`
`Si` = Number of neurons in layer `i`	`= net.layers{i}.size`
`Nt` = Number of targets	
`Vi` = Number of elements in target `i`, equal to `Sj`, where `j` is the `i`th layer with a target. (A layer `n` has a target if `net.targets(n) == 1`.)	
`No` = Number of network outputs	
`Ui` = Number of elements in output `i`, equal to `Sj`, where `j` is the `i`th layer with an output (A layer `n` has an output if `net.outputs(n) == 1`.)	
`ID` = Number of input delays	`= net.numInputDelays`
`LD` = Number of layer delays	`= net.numLayerDelays`
`TS` = Number of time steps	
`Q` = Number of concurrent vectors or sequences	

Variables

The variables a user commonly uses when defining a simulation or training session are

`P`	Network inputs	`Ni`-by-`TS` cell array, where each element `P{i,ts}` is an `Ri`-by-`Q` matrix
`Pi`	Initial input delay conditions	`Ni`-by-`ID` cell array, where each element `Pi{i,k}` is an `Ri`-by-`Q` matrix
`Ai`	Initial layer delay conditions	`Nl`-by-`LD` cell array, where each element `Ai{i,k}` is an `Si`-by-`Q` matrix
`T`	Network targets	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-by-`Q` matrix

These variables are returned by simulation and training calls:

`Y`	Network outputs	`No`-by-`TS` cell array, where each element `Y{i,ts}` is a `Ui`-by-`Q` matrix
`E`	Network errors	`Nt`-by-`TS` cell array, where each element `P{i,ts}` is a `Vi`-by-`Q` matrix
`perf`	Network performance	

Utility Function Variables

These variables are used only by the utility functions.

`Pc`	Combined inputs	`Ni`-by-`(ID+TS)` cell array, where each element `P{i,ts}` is an `Ri`-by-`Q` matrix `Pc = [Pi P]` = Initial input delay conditions and network inputs
`Pd`	Delayed inputs	Ni-by-Nj-by-TS cell array, where each element Pd{i,j,ts} is an (Ri*IWD(i,j))-by-Q matrix, and where IWD(i,j) is the number of delay taps associated with the input weight to layer i from input j Equivalently, `IWD(i,j) = length(net.inputWeights{i,j}.delays)` `Pd` is the result of passing the elements of `P` through each input weight's tap delay lines. Because inputs are always transformed by input delays in the same way, it saves time to do that operation only once instead of for every training step.
`BZ`	Concurrent bias vectors	`Nl`-by-1 cell array, where each element `BZ{i}` is an `Si`-by-Q matrix Each matrix is simply `Q` copies of the `net.b{i}` bias vector.
`IWZ`	Weighted inputs	`Ni`-by-`Nl`-by-`TS` cell array, where each element `IWZ{i,j,ts}` is an `Si`-by-`???`-by-`Q` matrix
`LWZ`	Weighted layer outputs	`Ni`-by-`Nl`-by-`TS` cell array, where each element `LWZ{i,j,ts}` is an `Si`-by-`Q` matrix
`N`	Net inputs	`Ni`-by-`TS` cell array, where each element `N{i,ts}` is an `Si`-by-`Q` matrix
`A`	Layer outputs	`Nl`-by-`TS` cell array, where each element `A{i,ts}` is an `Si`-by-`Q` matrix
`Ac`	Combined layer outputs	`Nl`-by-`(LD+TS)` cell array, where each element `A{i,ts}` is an `Si`-by-`Q` matrix `Ac = [Ai A]` = Initial layer delay conditions and layer outputs.

`Tl`	Layer targets	`Nl`-by-`TS` cell array, where each element `Tl{i,ts}` is an `Si`-by-`Q` matrix `Tl` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`El`	Layer errors	`Nl`-by-`TS` cell array, where each element `El{i,ts}` is an `Si`-by-`Q` matrix `El` contains empty matrices `[]` in rows of layers `i` not associated with targets, indicated by `net.targets(i) == 0`.
`X`	Column vector of all weight and bias values	