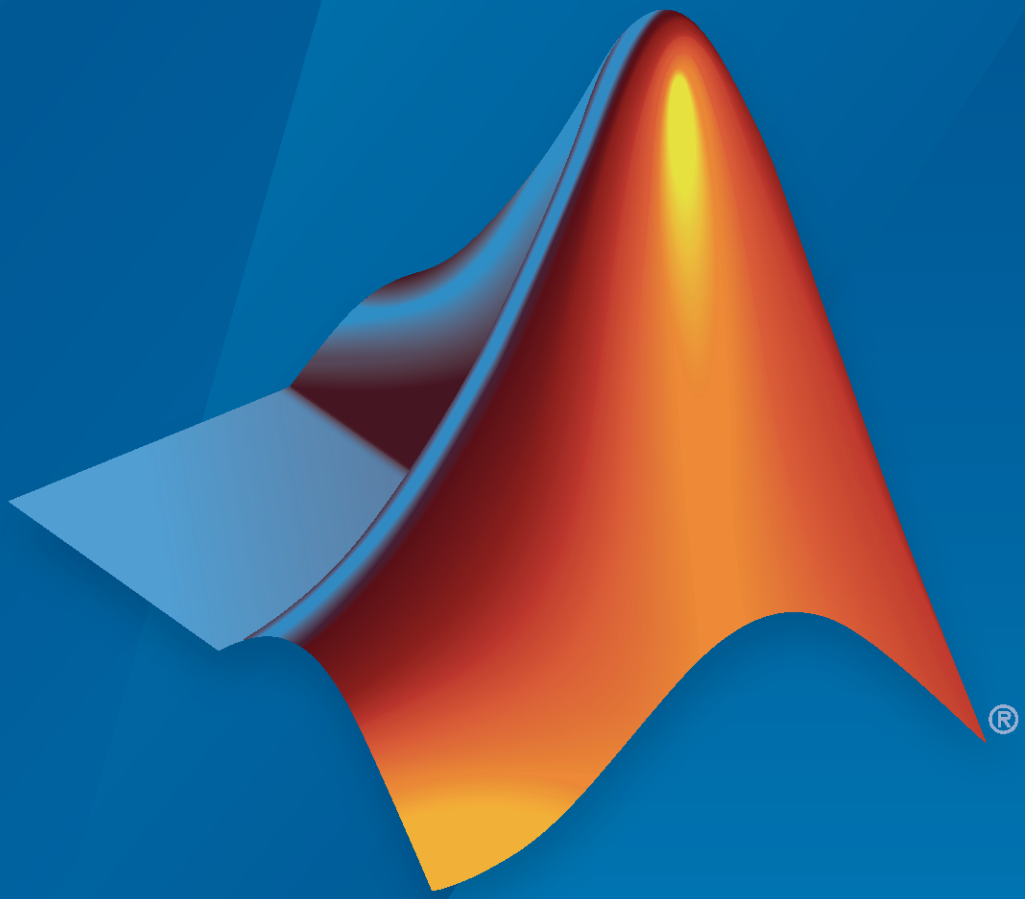
Deep Learning Toolbox™

User's Guide

Mark Hudson Beale
Martin T. Hagan
Howard B. Demuth



MATLAB®

R2020a

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)

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 Signal Processing Examples

12 Audio Examples

13 Reinforcement Learning Examples

14 Predictive Maintenance Examples

15 Automatic Differentiation

16 Deep Learning Data Preprocessing

17 Deep Learning Code Generation

Neural Network Design Book

18 Neural Network Objects, Data, and Training Styles

19 Multilayer Shallow Neural Networks and Backpropagation Training

20 Dynamic Neural Networks

21 Control Systems

22 Radial Basis Neural Networks

23 Self-Organizing and Learning Vector Quantization Networks

24 Adaptive Filters and Adaptive Training

25 Advanced Topics

26 Historical Neural Networks

27 Neural Network Object Reference

28 Function Approximation, Clustering, and Control Examples

29 Shallow Neural Networks Bibliography

A Mathematical Notation

B Neural Network Blocks for the Simulink Environment

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 1-12
Create a new deep neural network for classification or regression	“Create Simple Deep Learning Network for Classification” on page 3-40 “Train Convolutional Neural Network for Regression” on page 3-46
Resize, rotate, or preprocess images for training or prediction	“Preprocess Images for Deep Learning” on page 16-8
Label your image data automatically based on folder names, or interactively using an app	“Train Network for Image Classification” Image Labeler

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 4-9
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	"Deep Learning, Semantic Segmentation, and Detection" (Computer Vision Toolbox)
Classify text data	"Classify Text Data Using Deep Learning" on page 4-74
Classify audio data for speech recognition	"Speech Command Recognition Using Deep Learning" on page 4-17
Visualize what features networks have learned	"Deep Dream Images Using GoogLeNet" on page 5-2 "Visualize Activations of a Convolutional Neural Network" on page 5-75
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	"Deep Learning with Big Data on GPUs and in Parallel" on page 1-8

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.

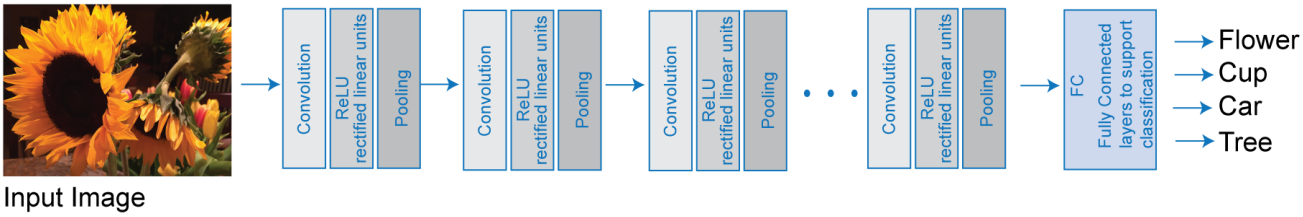
	Use a Pretrained Network for Transfer Learning	Create a New Deep Network
Training Data	Hundreds to thousands of labeled images (small)	Thousands to millions of labeled images
Computation	Moderate computation (GPU optional)	Compute intensive (requires GPU for speed)
Training Time	Seconds to minutes	Days 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 1-45.

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.

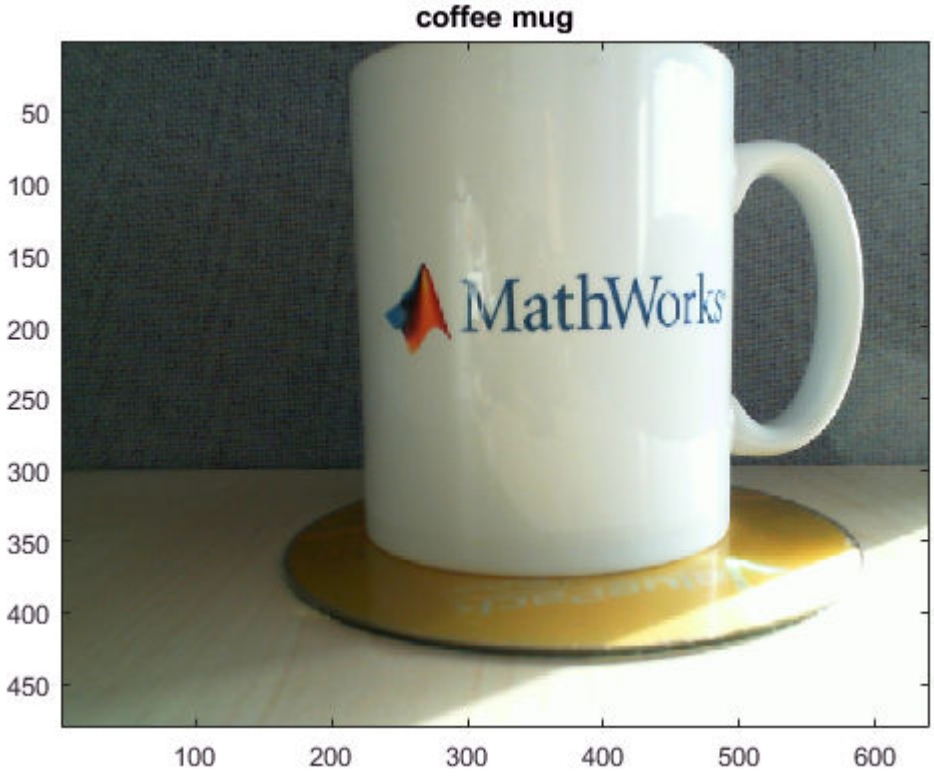
   ```
   camera = webcam; % Connect to the camera
   net = alexnet;   % Load the neural network
   ```

   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 1-12.

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.

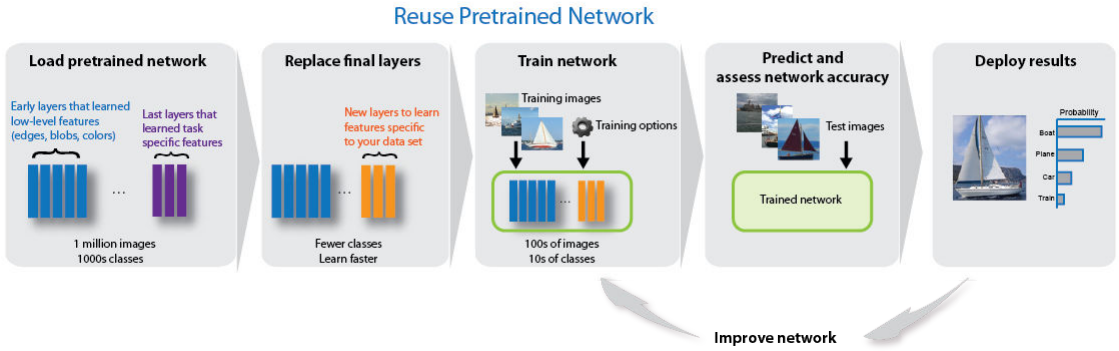
For example, if you take a network trained on thousands or millions of images, you can retrain it for new object detection using only hundreds of images. You can effectively fine-tune a pretrained network with much smaller data sets than the original training data. If you have a very large dataset, then 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 hardware and memory settings, see “Deep Learning with Big Data on GPUs and in Parallel” on page 1-8.

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 1-12
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Deep Learning with Big Data on GPUs and in Parallel” on page 1-8
- “Deep Learning, Semantic Segmentation, and Detection” (Computer Vision Toolbox)
- “Classify Text Data Using Deep Learning” on page 4-74
- “Deep Learning Tips and Tricks” on page 1-45

Deep Learning with Big Data on GPUs and in Parallel

Training deep networks is computationally intensive; however, neural networks are inherently parallel algorithms. You can usually accelerate training of convolutional neural networks by distributing training in parallel across multicore CPUs, high-performance GPUs, and clusters with multiple CPUs and GPUs. Using GPU or parallel options requires Parallel Computing Toolbox.

Tip GPU support is automatic if you have Parallel Computing Toolbox. By default, the `trainNetwork` function uses a GPU if available.

If you have access to a machine with multiple GPUs, then simply specify the training option `'ExecutionEnvironment','multi-gpu'`.

You do not need multiple computers to solve problems using data sets too large to fit in memory. You can use the `augmentedImageDatastore` function to work with batches of data without needing a cluster of machines. For an example, see “Train Network with Augmented Images”. 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.

Deep Learning Hardware and Memory Considerations	Recommendations	Required Products
Data too large to fit in memory	To import data from image collections that are too large to fit in memory, use the `augmentedImageDatastore` function. This function is designed to read batches of images for faster processing in machine learning and computer vision applications.	MATLAB Deep Learning Toolbox
CPU	If you do not have a suitable GPU, then you can train on a CPU instead. By default, the `trainNetwork` function uses the CPU if no GPU is available.	MATLAB Deep Learning Toolbox
GPU	By default, the `trainNetwork` function uses a GPU if available. Requires a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. Check your GPU using `gpuDevice`. Specify the execution environment using the `trainingOptions` function.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox

Deep Learning Hardware and Memory Considerations	Recommendations	Required Products
Parallel on your local machine using multiple GPUs or CPU cores	Take advantage of multiple workers by specifying the execution environment with the `trainingOptions` function. If you have more than one GPU on your machine, specify `'multi-gpu'`. Otherwise, specify `'parallel'`.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox
Parallel on a cluster or in the cloud	Scale up to use workers on clusters or in the cloud to accelerate your deep learning computations. Use `trainingOptions` and specify `'parallel'` to use a compute cluster. For more information, see “Deep Learning in the Cloud” on page 1-10.	MATLAB Deep Learning Toolbox Parallel Computing Toolbox MATLAB Parallel Server™

Tip To learn more, see “Scale Up Deep Learning in Parallel and in the Cloud” on page 7-2.

All functions for deep learning training, prediction, and validation in Deep Learning Toolbox perform computations using single-precision, floating-point arithmetic. Functions for deep learning include `trainNetwork`, `predict`, `classify`, and `activations`. The software uses single-precision arithmetic when you train networks using both CPUs and GPUs.

Because single-precision and double-precision performance of GPUs can differ substantially, it is important to know in which precision computations are performed. 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).

Training with Multiple GPUs

MATLAB supports training a single network using multiple GPUs in parallel. This can be achieved using multiple GPUs on your local machine, or on a cluster or cloud with workers with GPUs. To speed up training using multiple GPUs, try increasing the mini-batch size and learning rate.

- Enable multi-GPU training on your local machine by setting the “`'ExecutionEnvironment'`” option to `'multi-gpu'` with the `trainingOptions` function.
- On a cluster or cloud, set the “`'ExecutionEnvironment'`” option to `'parallel'` with the `trainingOptions` function.

Convolutional neural networks are typically trained iteratively using batches of images. This is done because the whole dataset is too large to fit into GPU memory. For optimum performance, you can experiment with the `MiniBatchSize` option that you specify with the `trainingOptions` function.

The optimal 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. Because it improves the significance of each batch, you can increase the learning rate. A good general guideline is to increase the learning rate proportionally to the increase in batch size. Depending on your application, a larger batch size and learning rate can speed up training without a decrease in accuracy, up to some limit.

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 7-4.

To learn more, see “Scale Up Deep Learning in Parallel and in the Cloud” on page 7-2 and “Select Particular GPUs to Use for Training” on page 7-5.

Deep Learning in the Cloud

If you do not have a suitable GPU available for faster training of a convolutional neural network, you can try your deep learning applications with multiple high-performance GPUs in the cloud, such as on Amazon® Elastic Compute Cloud (Amazon EC2®). MATLAB Deep Learning Toolbox provides examples that show you how to perform deep learning in the cloud using Amazon EC2 with P2 or P3 machine instances and data stored in the cloud.

You can accelerate training by using multiple GPUs on a single machine or in a cluster of machines with multiple GPUs. Train a single network using multiple GPUs, or train multiple models at once on the same data.

For more information on the complete cloud workflow, see “Deep Learning in Parallel and in the Cloud”.

Fetch and Preprocess Data in Background

When training a network in parallel, you can fetch and preprocess data in the background. To perform data dispatch in the background, enable background dispatch in the mini-batch datastore used by `trainNetwork`. 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 16-28.

To enable background dispatch, set the `DispatchInBackground` property of the datastore to `true`.

You can fine-tune the training computation and data dispatch loads between workers by specifying the `'WorkerLoad'` name-value pair argument of `trainingOptions`. For advanced options, you can try modifying the number of workers of the parallel pool. For more information, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox)

See Also

`trainNetwork` | `trainingOptions`

See Also

Related Examples

- “Scale Up Deep Learning in Parallel and in the Cloud” on page 7-2

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.

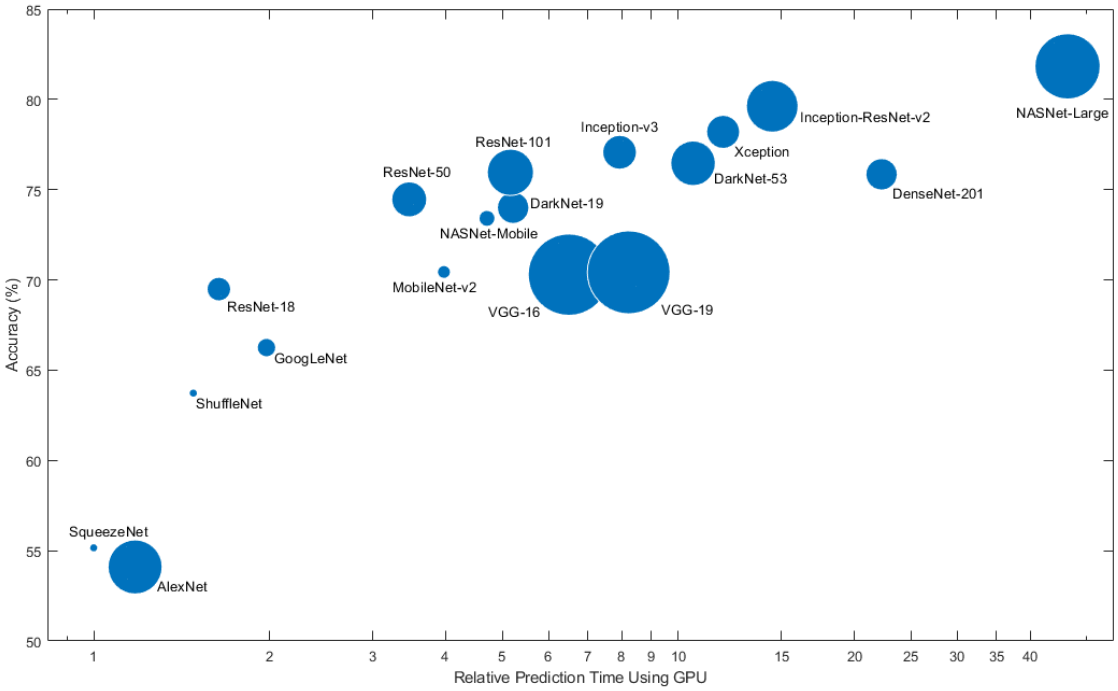
You can use previously trained networks for the following tasks:

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 1-15. 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 1-15. 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	4.6 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	6.3 MB	1.4	224-by-224
`nasnetmobile`	*	20 MB	5.3	224-by-224
`nasnetlarge`	*	360 MB	88.9	331-by-331
`darknet19`	19	72.5 MB	21.0	256-by-256
`darknet53`	53	145 MB	41.0	256-by-256
`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.

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.

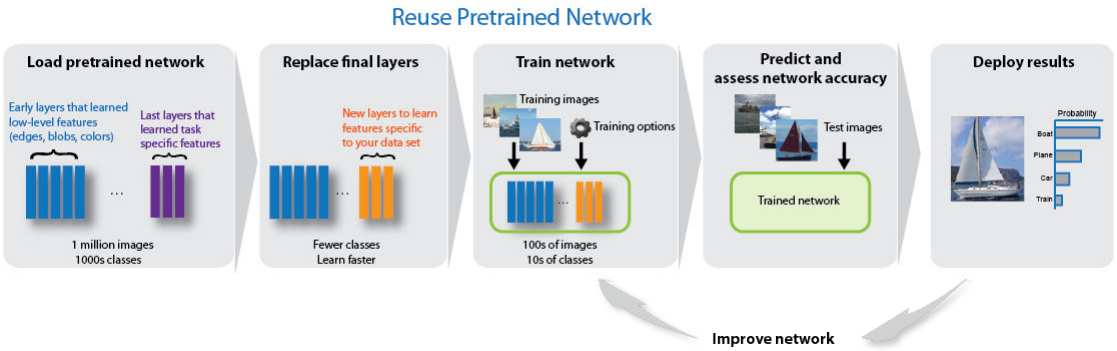
Transfer Learning

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`.

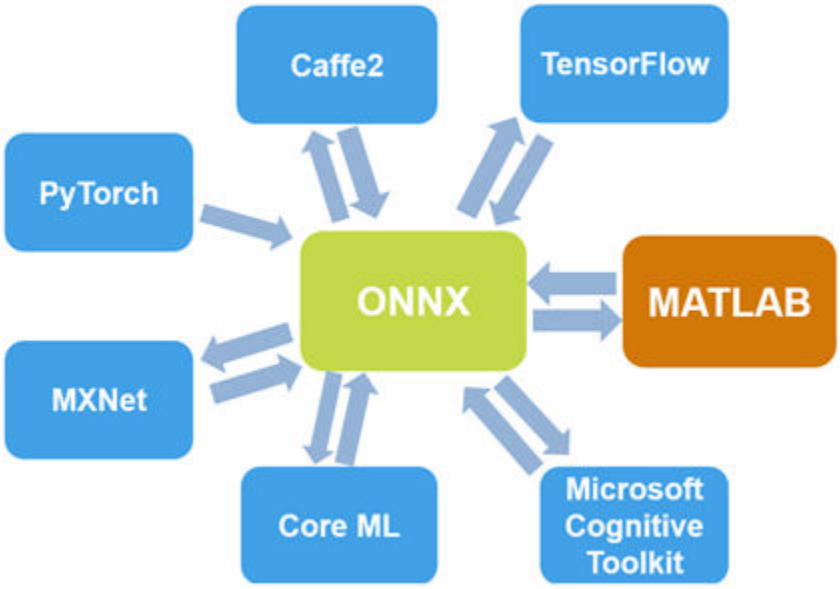
You can import network architectures of Caffe networks. Download the desired `.prototxt` file and use `importCaffeLayers` to import the network layers into MATLAB. For more information, see `importCaffeLayers`.

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`.



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

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

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 5-90
- “Visualize Activations of a Convolutional Neural Network” on page 5-75
- “Deep Dream Images Using GoogLeNet” on page 5-2

Learn About Convolutional Neural Networks

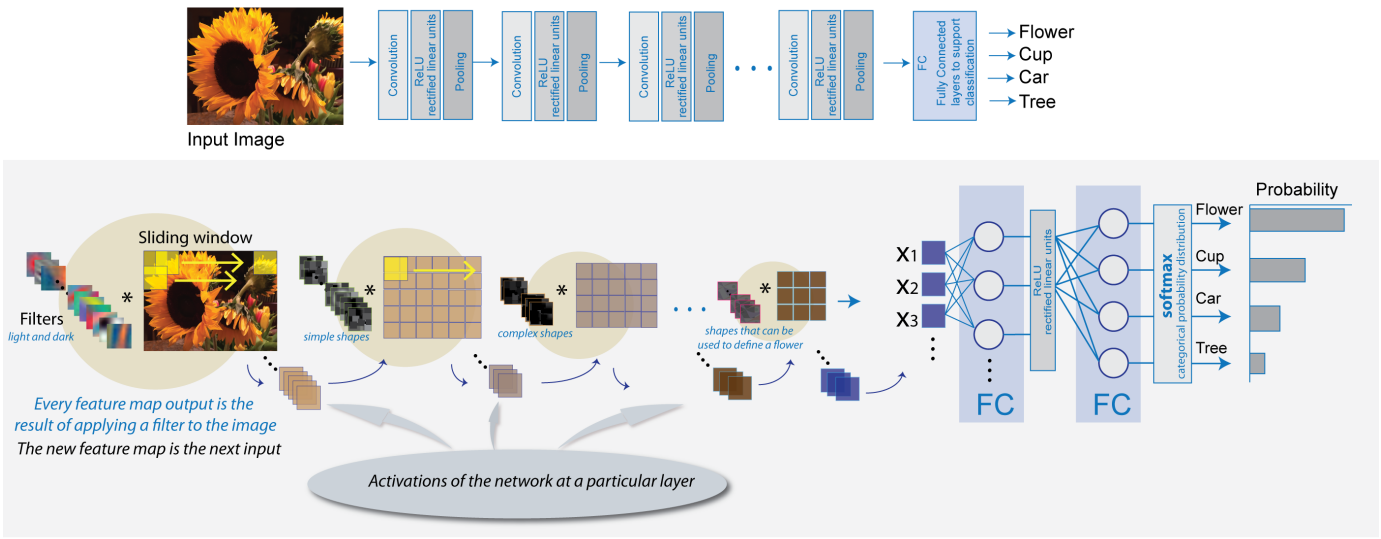
Convolutional neural networks (ConvNets) are widely used tools for deep learning. They are specifically suitable for images as inputs, although they are also used for other applications such as text, signals, and other continuous responses. They differ from other types of neural networks in a few ways:

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 1-30. For setting up training parameters, see “Set Up Parameters and Train Convolutional Neural Network” on page 1-41.

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 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Get Started with Transfer Learning”
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Pretrained Deep Neural Networks” on page 1-12

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 by specifying the multiple inputs using a `combinedDatastore` or `transformedDatastore` object.

For networks with multiple inputs, the datastore must be a combined or transformed datastore that returns a cell array with (`numInputs`+1) columns containing the predictors and the responses, where `numInputs` is the number of network inputs and `numResponses` is the number of responses. For `i` less than or equal to `numInputs`, the `i`th element of the cell array corresponds to the input `layers.InputNames(i)`, where `layers` is the layer graph defining the network architecture. The last column of the cell array corresponds to the responses.

Tip If the network also has multiple outputs, then you must define the network as a function and train the network using a custom training loop. for more information, see “Multiple-Output Networks” on page 1-21.

Prediction

To make predictions on a trained deep learning network with multiple inputs, use either the `predict` or `classify` functions and specify the multiple inputs using a `combinedDatastore` or `transformedDatastore` object.

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, define the network as a function and train it using a custom training loop. For an example, see “Train Network with Multiple Outputs” on page 3-54.

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 15-173.

Alternatively, convert the model function to a `DAGNetwork` object using the `functionToLayerGraph` and `assembleNetwork` functions. With the assembled network, you can use the `predict` function for `DAGNetwork` objects which allows you to:

- Make predictions with datastore input directly.
- Save the network in a MAT file.
- Use options provided by the `predict` function for `DAGNetwork` objects.

For an example, see “Assemble Multiple-Output Network for Prediction” on page 15-106.

See Also

`assembleNetwork` | `functionToLayerGraph` | `predict`

More About

- “Train Network with Multiple Outputs” on page 3-54
- “Assemble Multiple-Output Network for Prediction” on page 15-106
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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 3-40 “Train Convolutional Neural Network for Regression” on page 3-46 “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 4-9
Create deep learning network for audio data.	“Speech Command Recognition Using Deep Learning” on page 4-17
Create deep learning network for text data.	“Classify Text Data Using Deep Learning” on page 4-74 “Generate Text Using Deep Learning” on page 4-131

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 15-2.

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
 `roiInputLayer` (Computer Vision Toolbox™)	An ROI input layer inputs images to a Fast R-CNN object detection network.

Convolution and Fully Connected Layers

Layer	Description
 `convolution2dLayer`	A 2-D convolutional layer applies sliding convolutional filters to the input.
 `convolution3dLayer`	A 3-D convolutional layer applies sliding cuboidal convolution filters to three-dimensional 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.
 `gruLayer`	A GRU layer learns dependencies between time steps in time series and sequence data.
 `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.

Layer	Description
 `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` (Text Analytics Toolbox™)	A word embedding layer maps word indices to vectors.

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.
 `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.
 `preluLayer` on page 15-17 (Custom layer example)	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.

Normalization, Dropout, and Cropping Layers

Layer	Description
 `batchNormalizationLayer`	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.
 `crossChannelNormalizationLayer`	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.

Layer	Description
 `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.

Pooling and Unpooling Layers

Layer	Description
 `averagePooling2dLayer`	An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.
 `averagePooling3dLayer`	A 3-D average pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions and computing the average values of each region.
 `globalAveragePooling2dLayer`	A global average pooling layer performs down-sampling by computing the mean of the height and width dimensions of the input.
 `globalAveragePooling3dLayer`	A 3-D global average pooling layer performs down-sampling by computing the mean of the height, width, and depth dimensions of the input.
 `maxPooling2dLayer`	A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.
 `maxPooling3dLayer`	A 3-D max pooling layer performs down-sampling by dividing three-dimensional input into cuboidal pooling regions, and computing the maximum of each region.
 `globalMaxPooling2dLayer`	A global max pooling layer performs down-sampling by computing the maximum of the height and width dimensions of the input.
 `globalMaxPooling3dLayer`	A 3-D global max pooling layer performs down-sampling by computing the maximum of the height, width, and depth dimensions of the input.
 `maxUnpooling2dLayer`	A max unpooling layer unpools the output of a max pooling layer.

Combination Layers

Layer	Description
 `additionLayer`	An addition layer adds 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 15-28 (Custom layer example)	A weighted addition layer scales and adds inputs from multiple neural network layers element-wise.

Object Detection Layers

Layer	Description
 `roiInputLayer` (Computer Vision Toolbox)	An ROI input layer inputs images to a Fast R-CNN object detection network.
 `roiMaxPooling2dLayer` (Computer Vision Toolbox)	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.
 `anchorBoxLayer` (Computer Vision Toolbox)	An anchor box layer stores anchor boxes for a feature map used in object detection networks.
 `regionProposalLayer` (Computer Vision Toolbox)	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` (Computer Vision Toolbox)	An SSD merge layer merges the outputs of feature maps for subsequent regression and classification loss computation.
 `rpnSoftmaxLayer` (Computer Vision Toolbox)	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` (Computer Vision Toolbox)	A focal loss layer predicts object classes using focal loss.

Layer	Description
 `rpnClassificationLayer` (Computer Vision Toolbox)	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` (Computer Vision Toolbox)	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.

Generative Adversarial Network Layers

Layer	Description
 `projectAndReshapeLayer` on page 3-72 (Custom layer example)	A project and reshape layer takes as input 1-by-1-by-`numLatentInputs` arrays and converts them to images of the specified size. Use project and reshape layers to reshape the noise input to GANs.
 `embedAndReshapeLayer` on page 3-83 (Custom layer example)	An embed and reshape layer takes as input numeric indices of categorical elements and converts them to images of the specified size. Use embed and reshape layers to input categorical data into conditional GANs.

Output Layers

Layer	Description
 `softmaxLayer`	A softmax layer applies a softmax function to the input.
 `classificationLayer`	A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.
 `regressionLayer`	A regression layer computes the half-mean-squared-error loss for regression problems.
 `pixelClassificationLayer` (Computer Vision Toolbox)	A pixel classification layer provides a categorical label for each image pixel or voxel.
 `dicePixelClassificationLayer` (Computer Vision Toolbox)	A Dice pixel classification layer provides a categorical label for each image pixel or voxel using generalized Dice loss.
 `focalLossLayer` (Computer Vision Toolbox)	A focal loss layer predicts object classes using focal loss.

Layer	Description
 `rpnSoftmaxLayer` (Computer Vision Toolbox)	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` (Computer Vision Toolbox)	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` (Computer Vision Toolbox)	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.
 `weightedClassificationLayer` on page 15-47 (Custom layer example)	A weighted classification layer computes the weighted cross entropy loss for classification problems.
 `tverskyPixelClassificationLayer` on page 8-124 (Custom layer example)	A Tversky pixel classification layer provides a categorical label for each image pixel or voxel using Tversky loss.
 `sseClassificationLayer` on page 15-39 (Custom layer example)	A classification SSE layer computes the sum of squares error loss for classification problems.
 `maeRegressionLayer` on page 15-54 (Custom layer example)	A regression MAE layer computes the mean absolute error loss for regression problems.

See Also

Deep Network Designer | `trainNetwork` | `trainingOptions`

More About

- “Build Networks with Deep Network Designer” on page 2-15
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Define Custom Deep Learning Layers” on page 15-2
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Sequence Classification Using Deep Learning” on page 4-2
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Learning Tips and Tricks” on page 1-45

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 1-23. To learn about LSTM networks for sequence classification and regression, see “Long Short-Term Memory Networks” on page 1-53. To learn how to create your own custom layers, see “Define Custom Deep Learning Layers” on page 15-2.

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, if you have categorical responses, you must have a softmax layer 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 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 the input. Create a 2-D convolutional layer using `convolution2dLayer`.

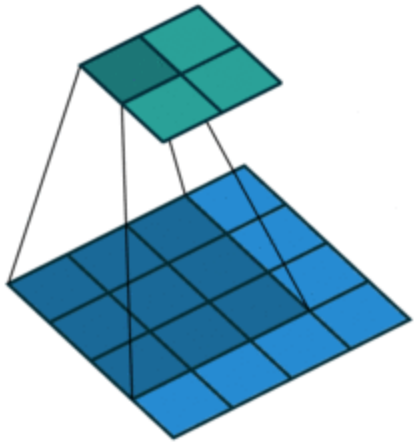
The convolutional layer consists of various components.[1]

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.

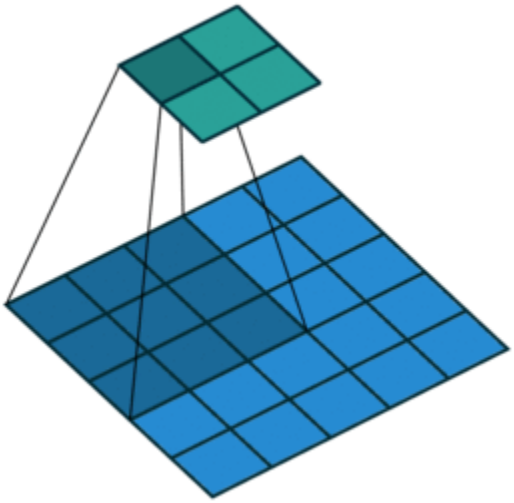
This image shows a 3-by-3 filter scanning through the input. The lower map represents the input and the upper map represents the output.



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.

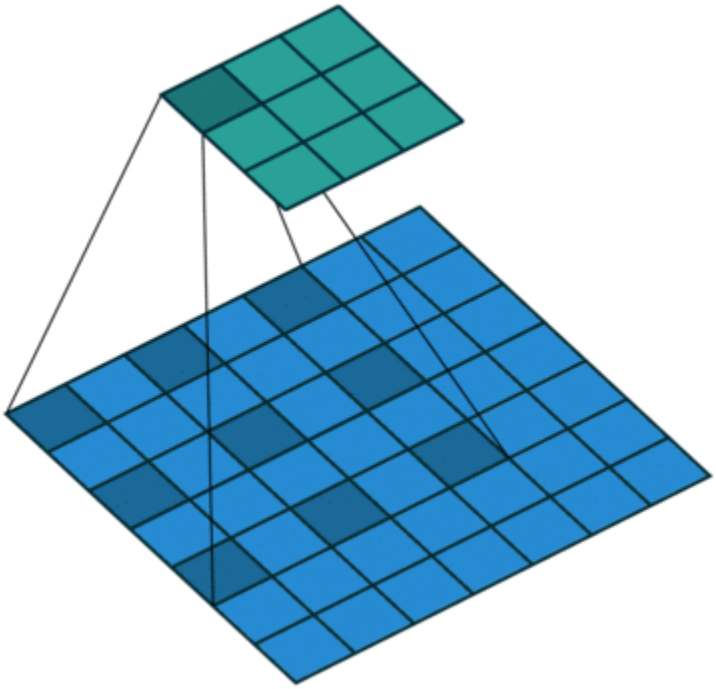
Dilated Convolution

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.



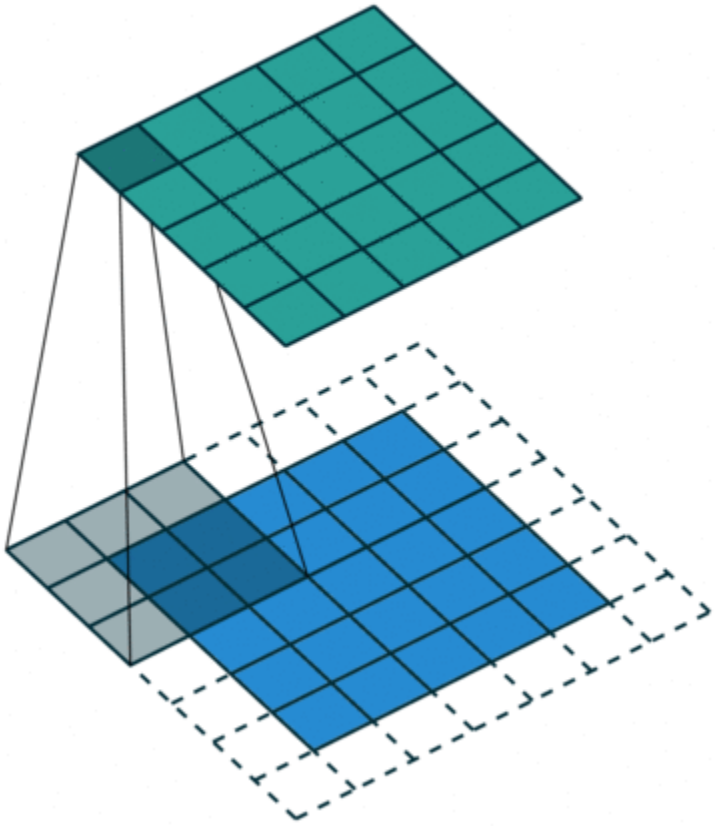
Feature Maps

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.

Zero Padding

You can also apply zero padding to input image borders vertically and horizontally using the `'Padding'` name-value pair argument. Padding is rows or columns of zeros added to the borders of an image input. 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 zero 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 1-41.

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 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.

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 1-25.

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 max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region. Create a max pooling layer using `maxPooling2dLayer`.

An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and 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 1-41.

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 multi-class classification problems with mutually exclusive classes. Create a classification layer using `classificationLayer`.

For classification problems, a softmax layer and then a classification layer must 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 must follow the 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]:

$$\text{loss} = -\sum_{i=1}^{N}\sum_{j=1}^{K} t_{ij}\ln y_{ij},$$

where N is the number of samples, K is the number of classes, t_{ij} is the indicator that the ith sample belongs to the jth class, and y_{ij} is the output for sample i for class j, which in this case, is the value from the softmax function. That is, it is the probability that the network associates the ith input with class j.

Regression Layer

Create a regression layer using `regressionLayer`.

A regression layer computes the half-mean-squared-error loss for regression problems. 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, arXiv:1502.03167* (2015).

See Also

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

More About

- “List of Deep Learning Layers” on page 1-23
- “Learn About Convolutional Neural Networks” on page 1-19
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Resume Training from Checkpoint Network” on page 5-30
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Pretrained Deep Neural Networks” on page 1-12
- “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 1-30, 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 1-43.

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 CUDA enabled GPU with compute capability 3.0 or higher.

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 5-30.

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 training 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

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

More About

- “Learn About Convolutional Neural Networks” on page 1-19
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Resume Training from Checkpoint Network” on page 5-30

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 1-12. 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 3-66.
	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 4-17.

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 4-34 and “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39.
	Time series forecasting	For an example, see “Time Series Forecasting Using Deep Learning” on page 4-9.
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 4-74.
	Text generation	For an example, see “Generate Text Using Deep Learning” on page 4-131.
Audio	Audio classification and regression	For an example, see “Speech Command Recognition Using Deep Learning” on page 4-17.

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 1-57.
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 1-41.

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 1-41.

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 15-76.
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 semantic segmentation tasks, you can specify class weights in `pixelClassificationLayer` using the `ClassWeights` property. For image classification tasks, you can use the example custom classification layer provided in “Define Custom Weighted Classification Layer” on page 15-47.

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 16-8.

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` 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 3-46.

Preprocess Sequence Data

For more information about working with LSTM networks, see “Long Short-Term Memory Networks” on page 1-53.

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 1-60.
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 1-57.
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 on Multiple GPUs” on page 7-2.

For more information, see “Scale Up Deep Learning in Parallel 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

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

More About

- “Pretrained Deep Neural Networks” on page 1-12
- “Preprocess Images for Deep Learning” on page 16-8
- “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 3-66

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.

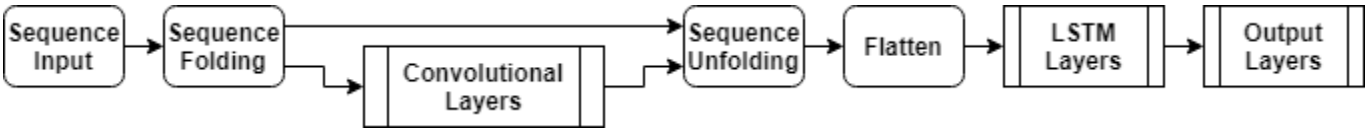
LSTM Network Architecture

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 4-39.

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 4-48.

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
 `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` (Text Analytics Toolbox)	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 4-9.

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`.

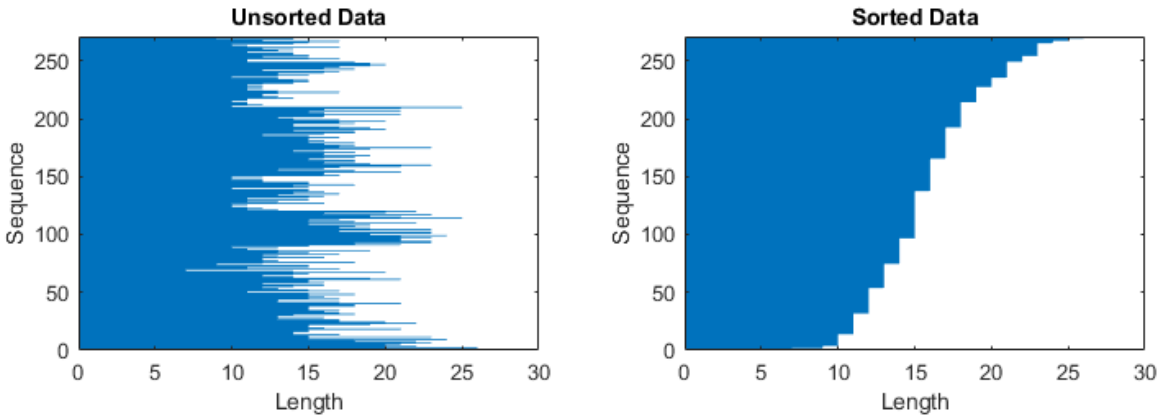
After training the network, use the same mini-batch size and padding options when using the `classify`, `predict`, `classifyAndUpdateState`, `predictAndUpdateState`, and `activations` functions.

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);
```

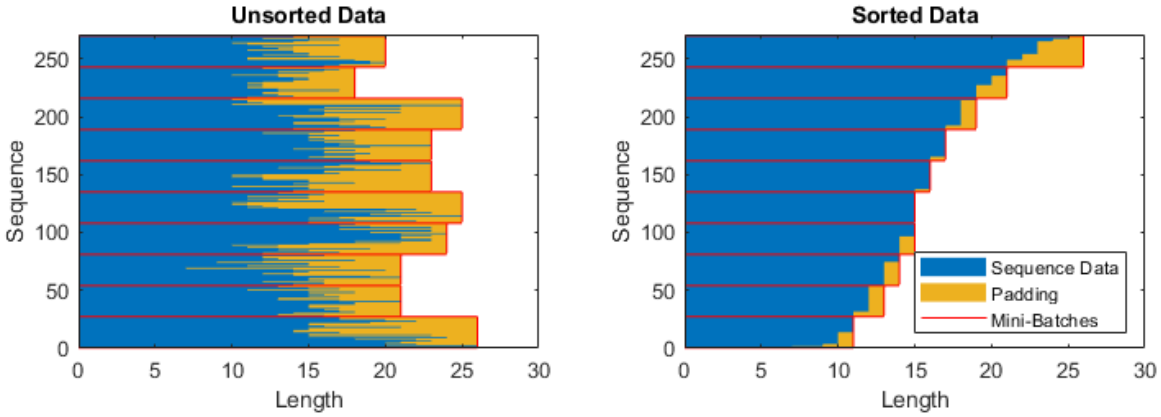
The following figures show the sequence lengths of the sorted and unsorted data in bar charts.



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.

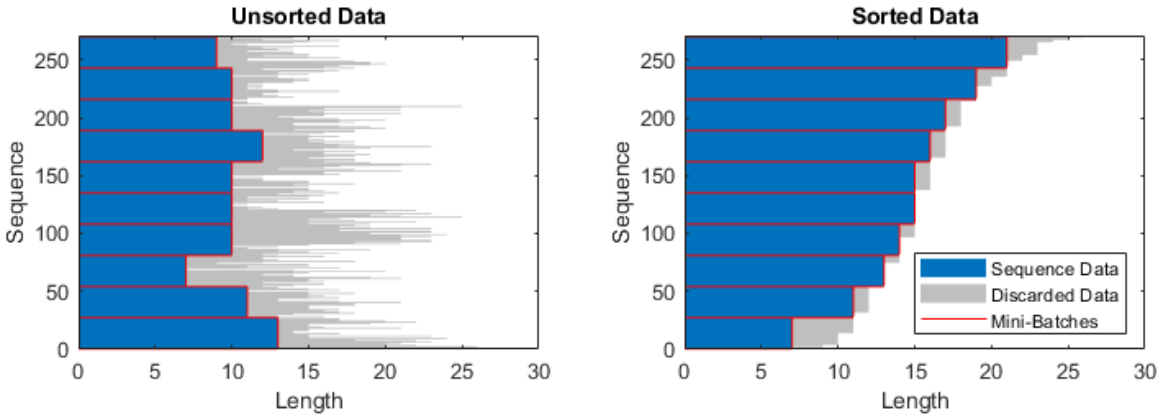
The following figures illustrate the effect of setting `'SequenceLength'` to `'longest'`.



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'`.



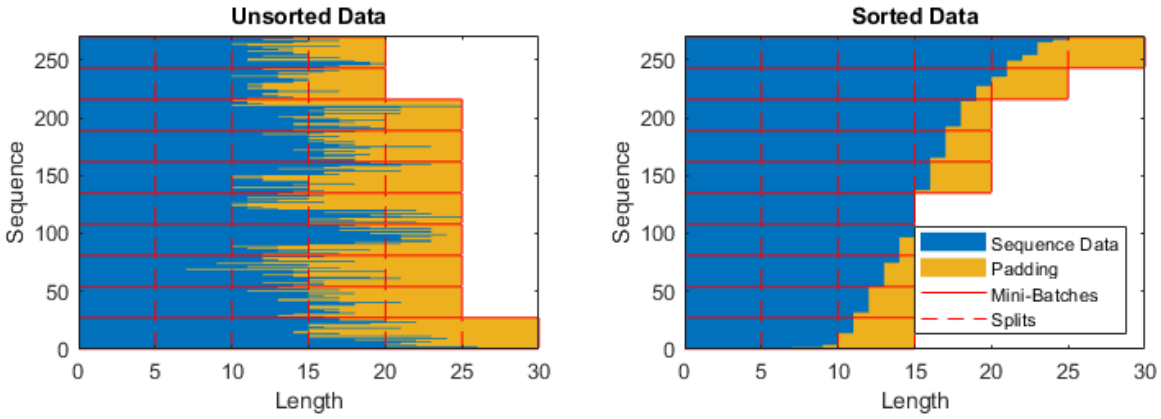
Split Sequences

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

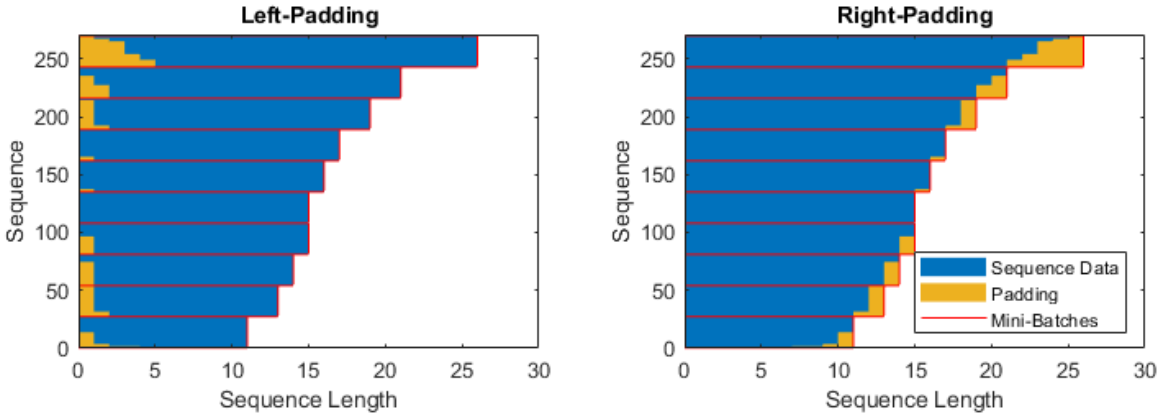
The location of the padding and truncation can impact training, classification, and prediction accuracy. Try setting the `'SequencePaddingDirection'` option in `trainingOptions` to `'left'` or `'right'` and see which is best for your data.

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

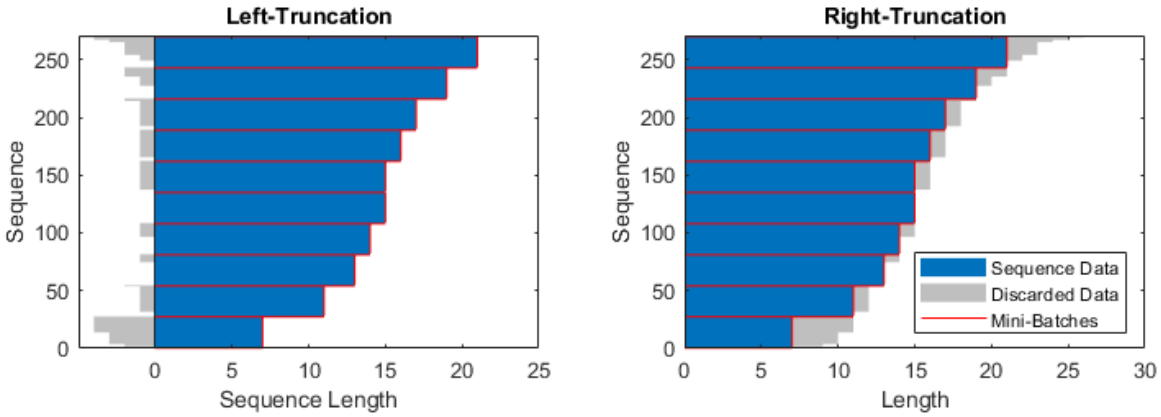
pad or truncate sequence data on the left, set the `'SequencePaddingDirection'` option to `'left'`.

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.

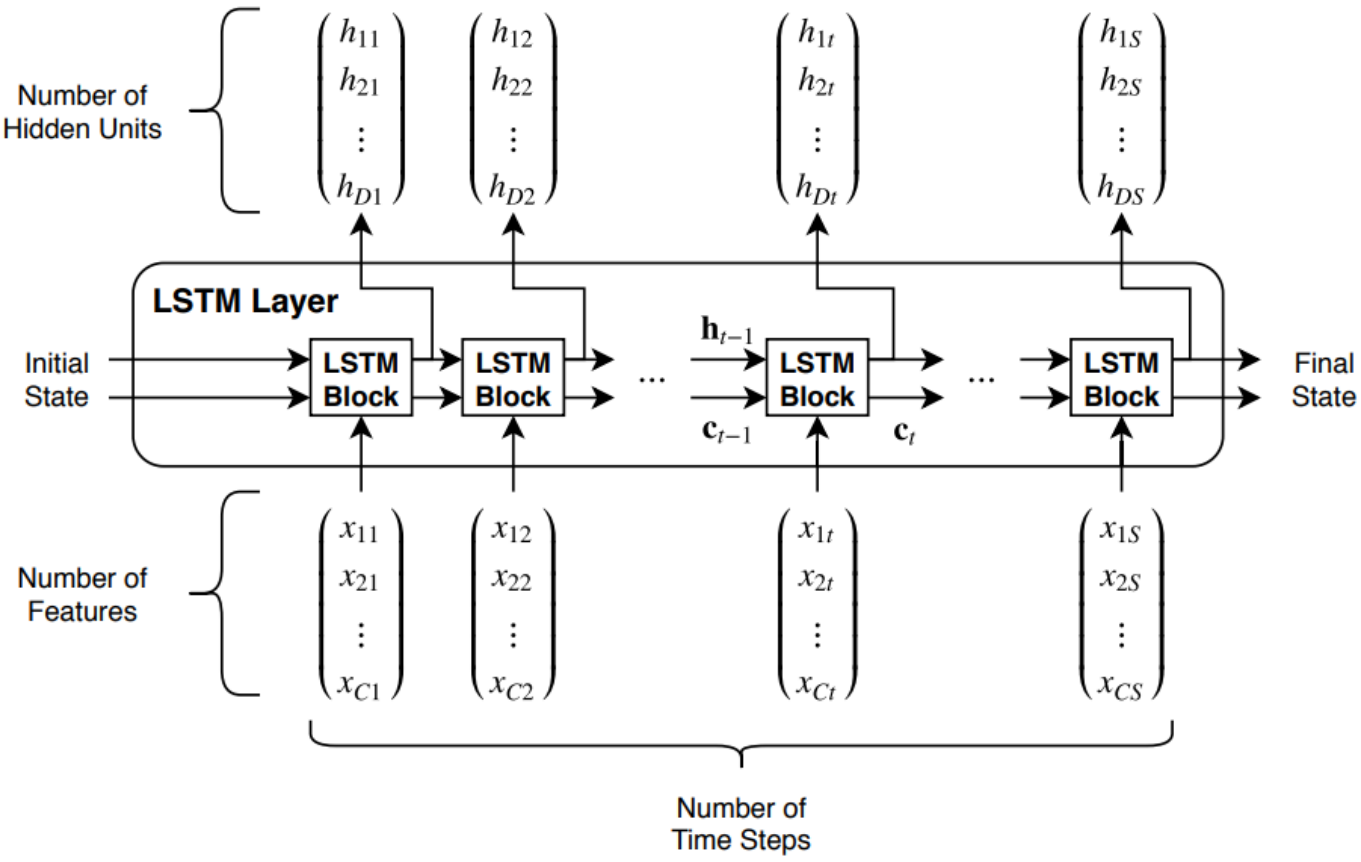
To learn more, see “Train Network Using Out-of-Memory Sequence Data” on page 16-89 and “Classify Out-of-Memory Text Data Using Deep Learning” on page 16-98.

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 5-86.

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.



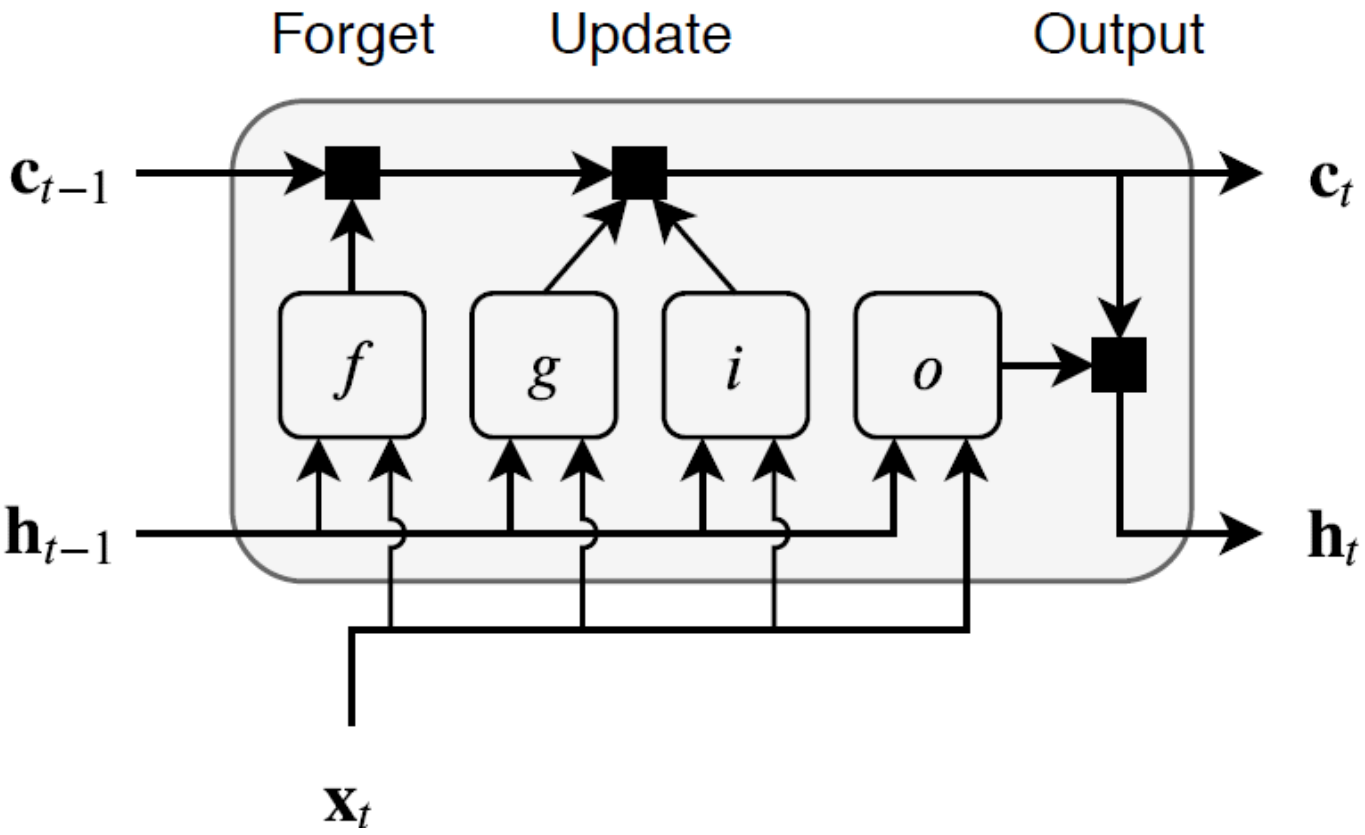
The first LSTM block uses the initial state of the network and the first time step of the sequence to compute the first output and the updated cell state. At time step t, the block uses the current state of the network ($\mathbf{c}_{t-1}, \mathbf{h}_{t-1}$) and the next time step of the sequence to compute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

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

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39

- “Classify Videos Using Deep Learning” on page 4-48
- “Visualize Activations of LSTM Network” on page 5-86
- “Develop Custom Mini-Batch Datastore” on page 16-28
- “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:

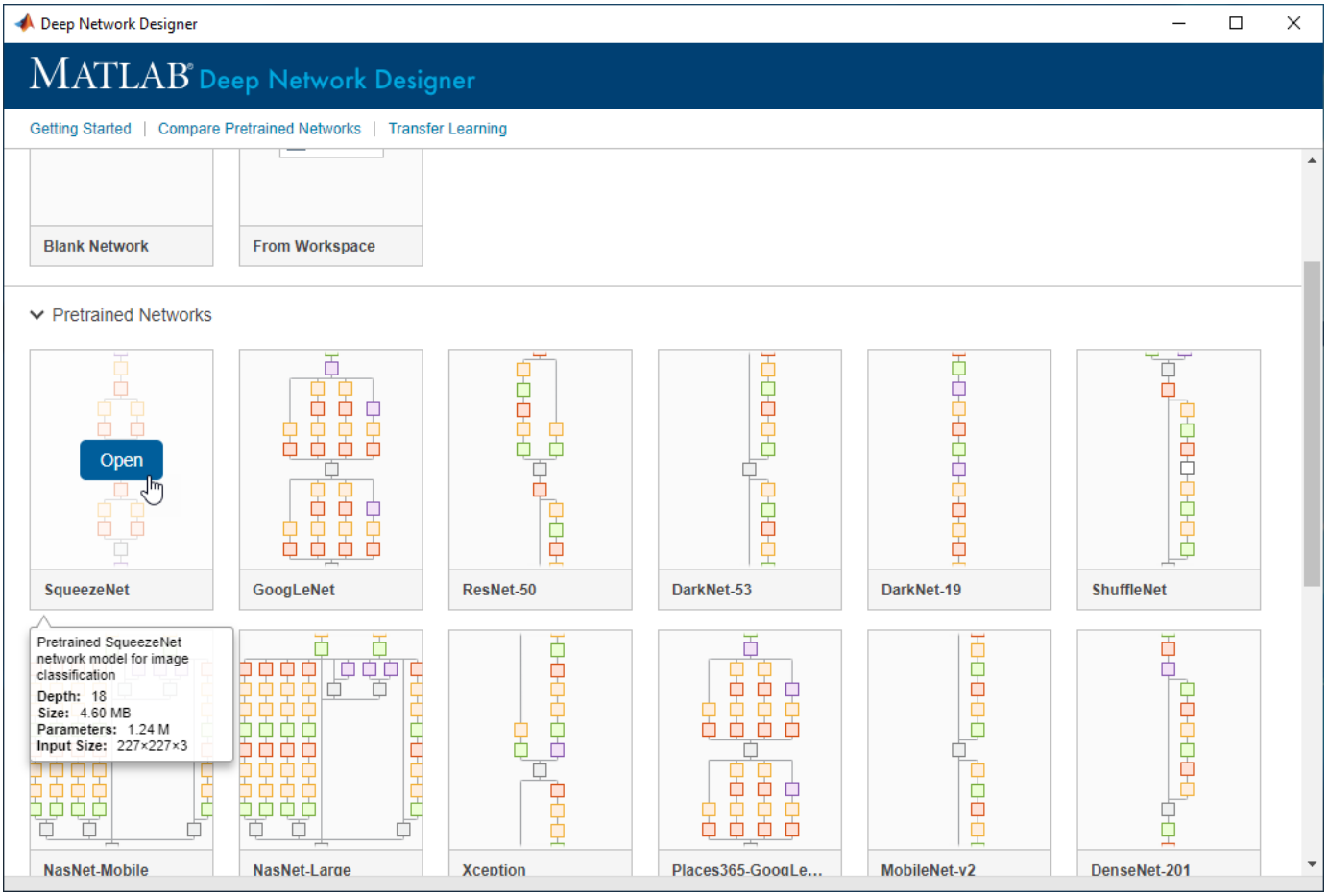
```
deepNetworkDesigner
```

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 1-12.

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 an example showing how to train a network using nonimage data, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-22.

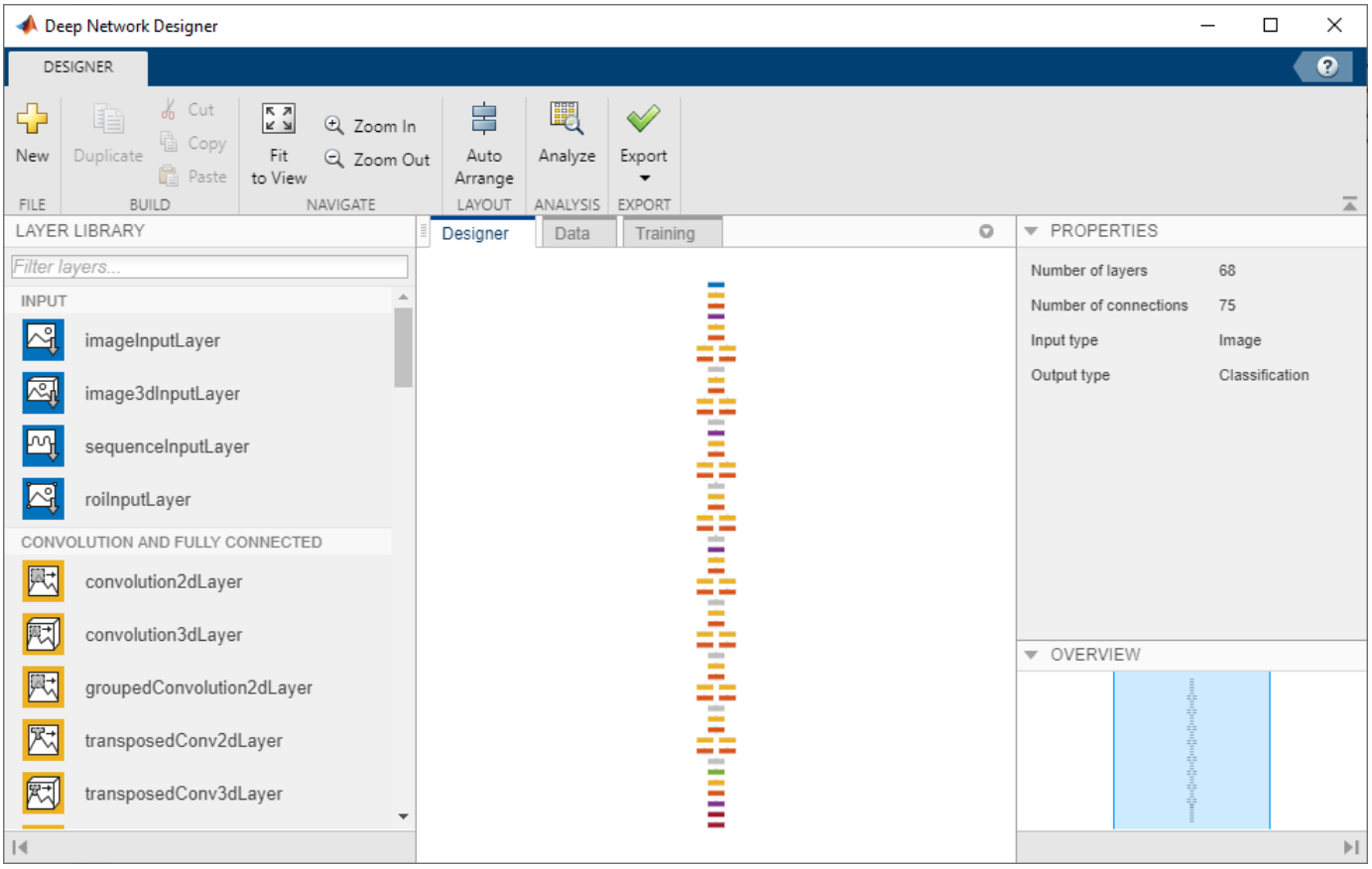
SqueezeNet does not require an additional support package. For other pretrained networks, if you do not have the required support package installed, then the app provides the **Install** option.

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**. The Import Data dialog box opens.

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

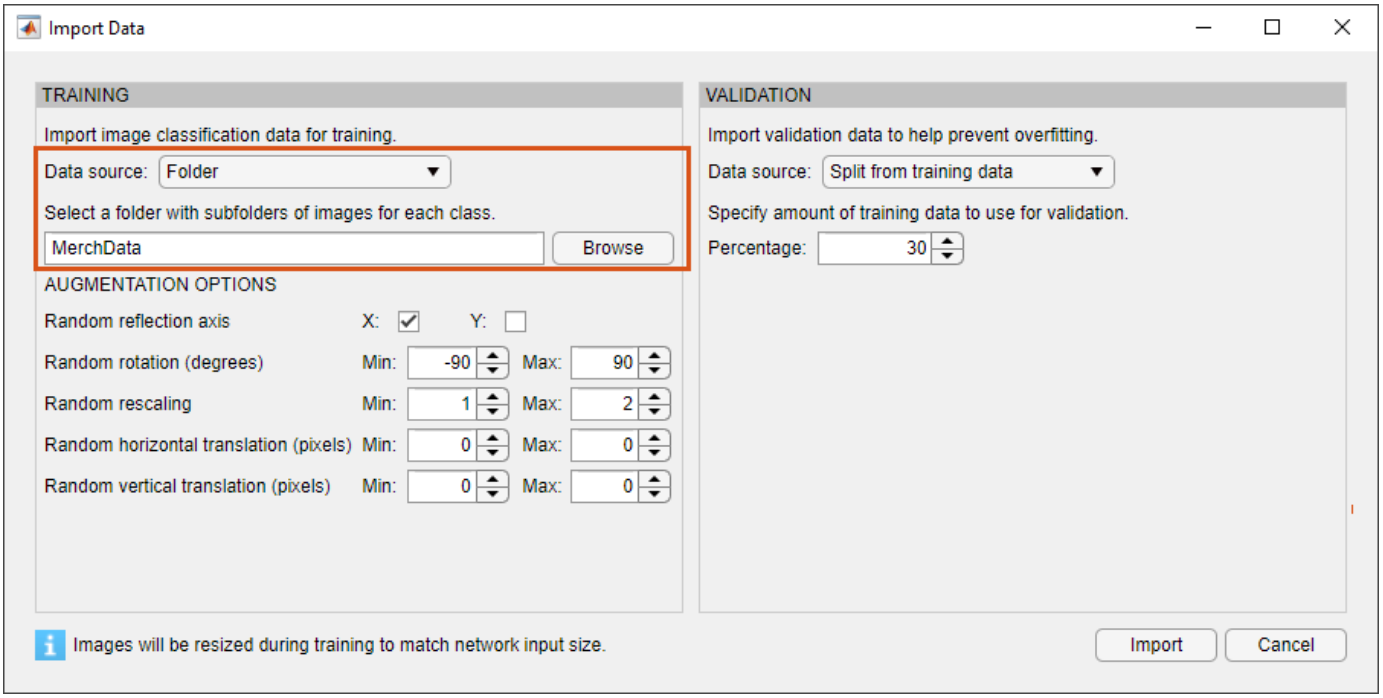


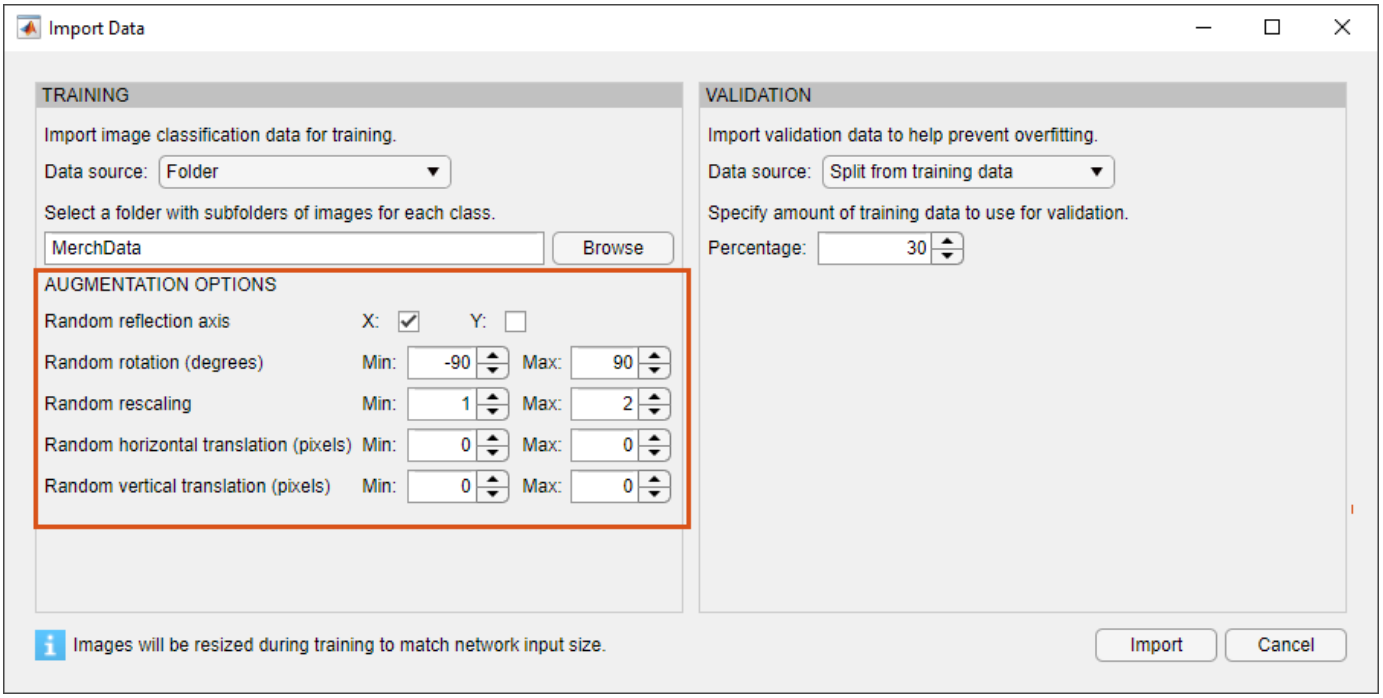
Image Augmentation

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.

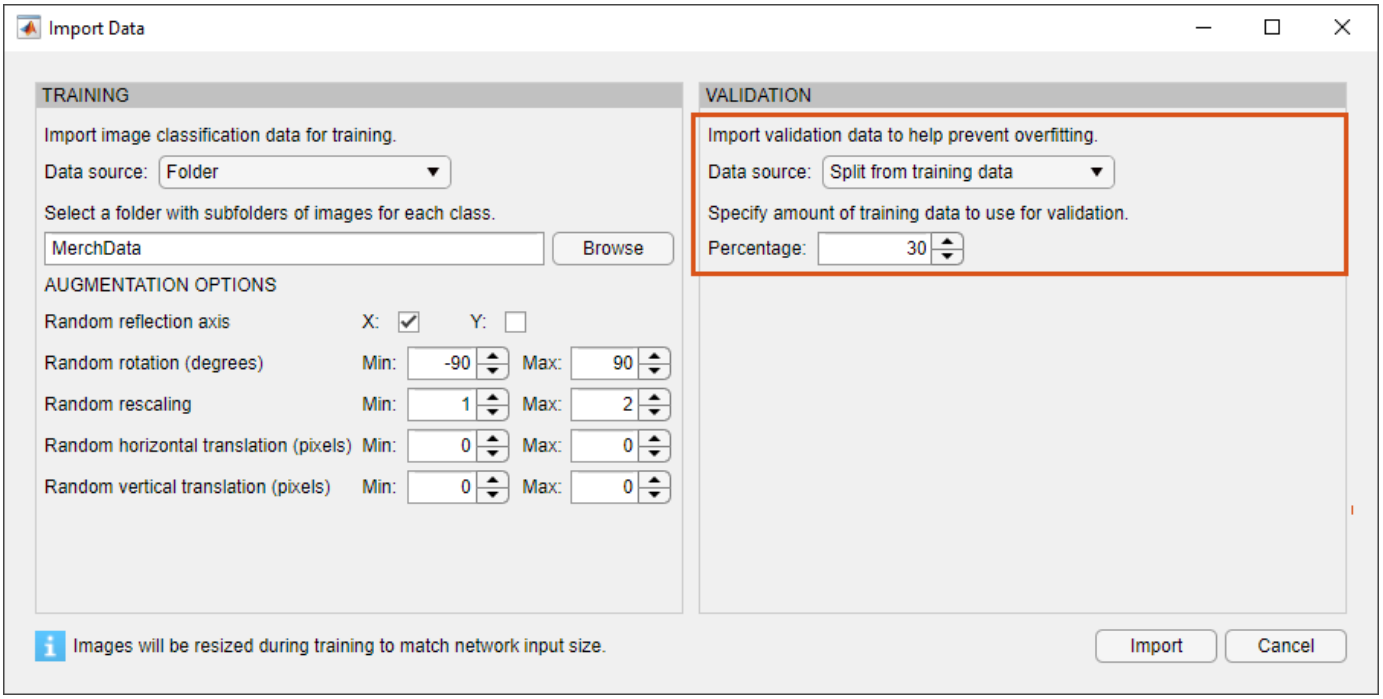
For this example, apply a random reflection in the x-axis, a random rotation from the range [-90,90] degrees, and a random rescaling from the range [1,2].



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.

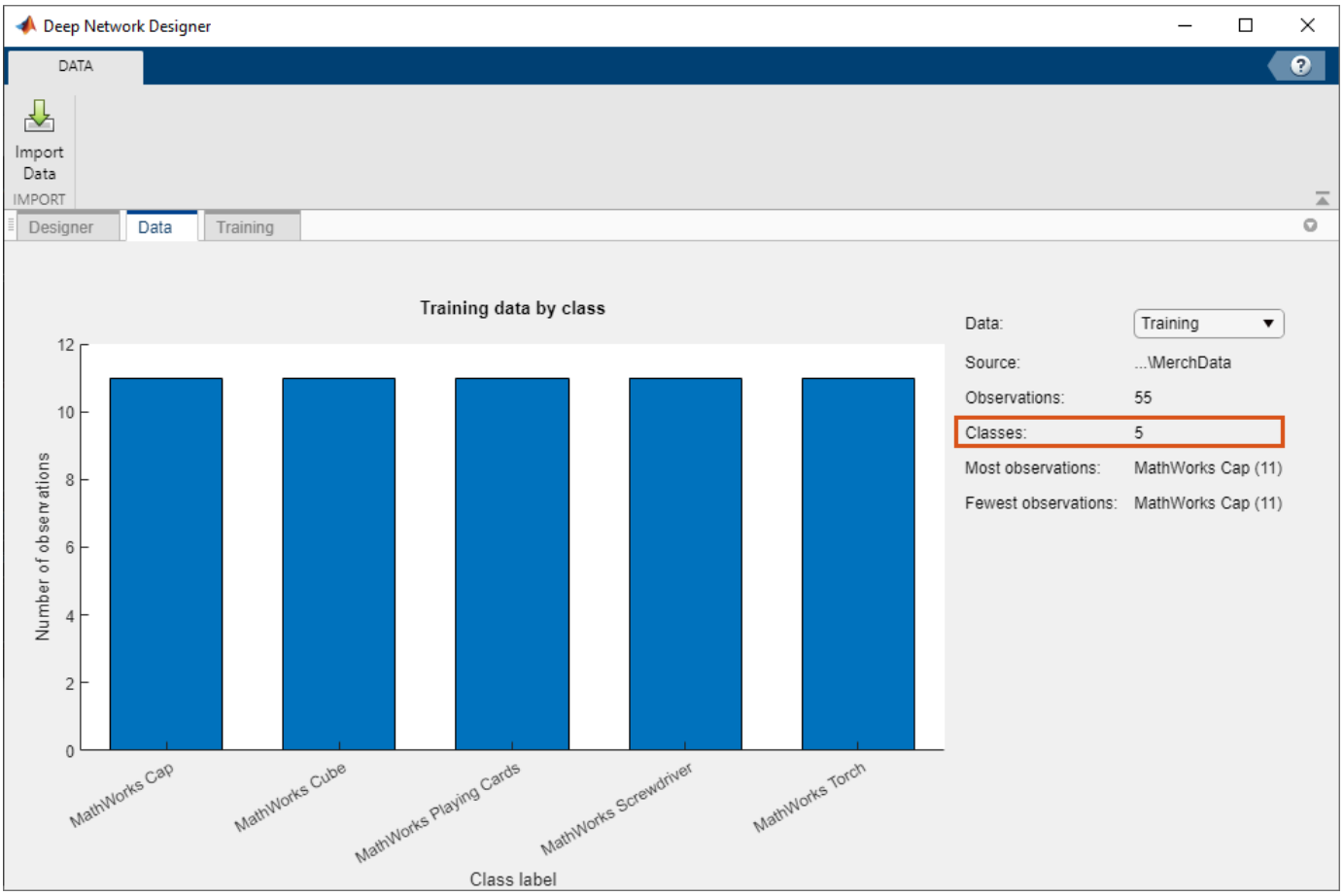
For this example, use 30% of the images for validation.



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** pane. You can see that, in this example, there are five classes in the data set.



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

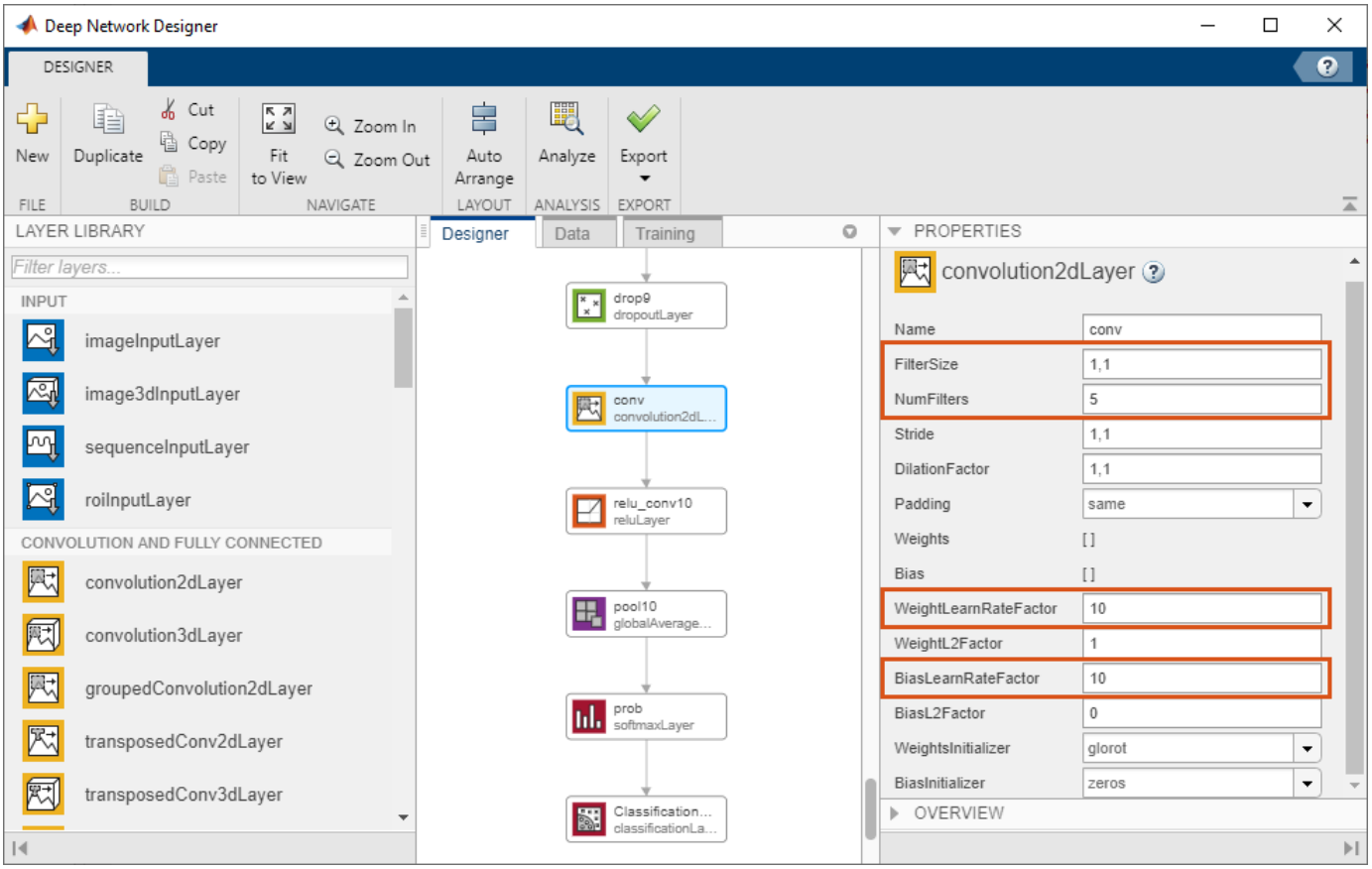
To use a pretrained network for transfer learning, you must change the number of classes to match your new data set. First, find the last learnable layer in the network. For SqueezeNet, the last learnable layer is the last convolutional layer, '`conv10`'. In this case, replace the convolutional layer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

Drag a new `convolutional2dLayer` 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 set `WeightLearnRateFactor` and `BiasLearnRateFactor` to 10.

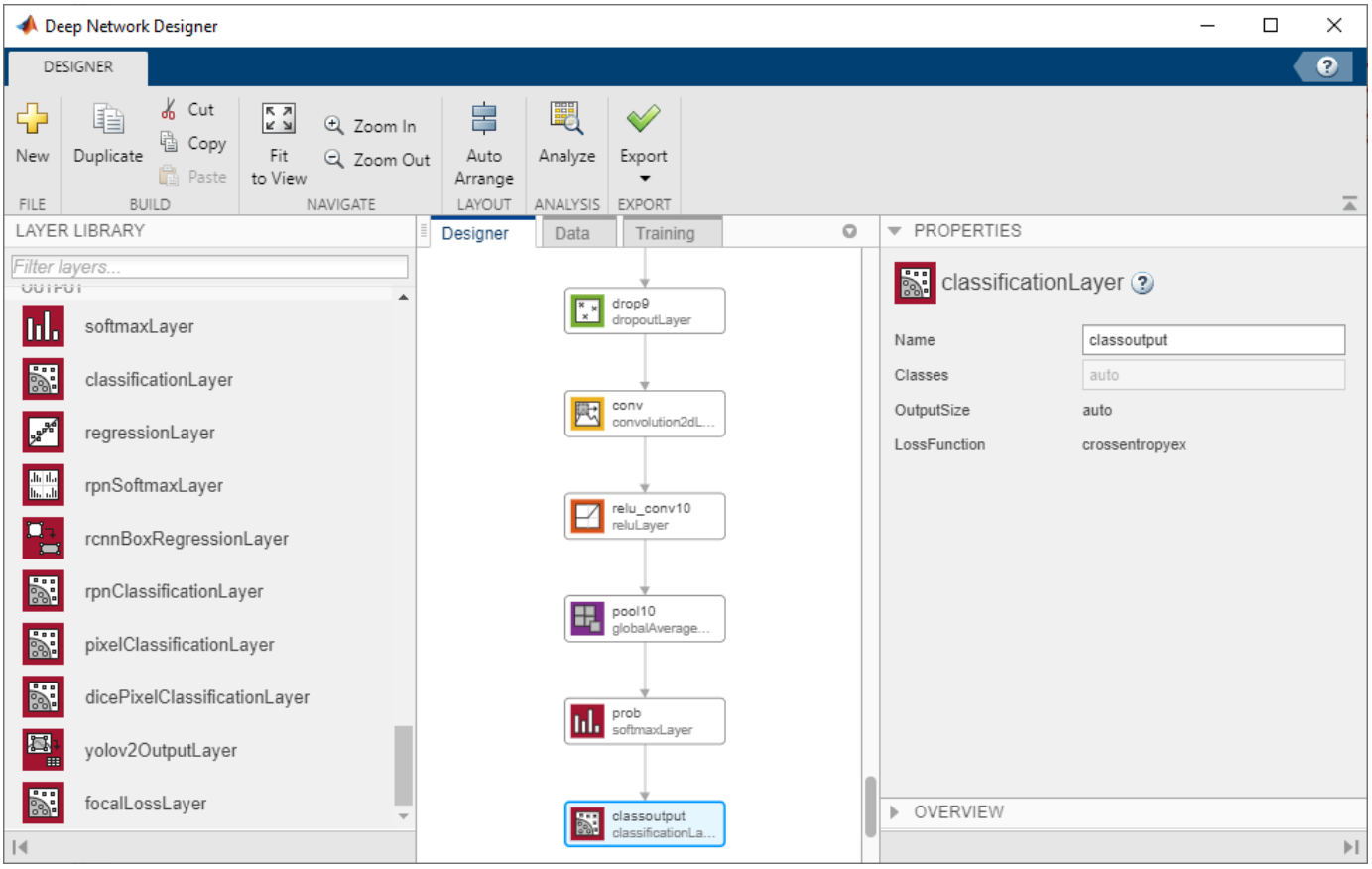
Delete the last 2-D convolutional layer and connect your new layer instead.



Replace Output Layer

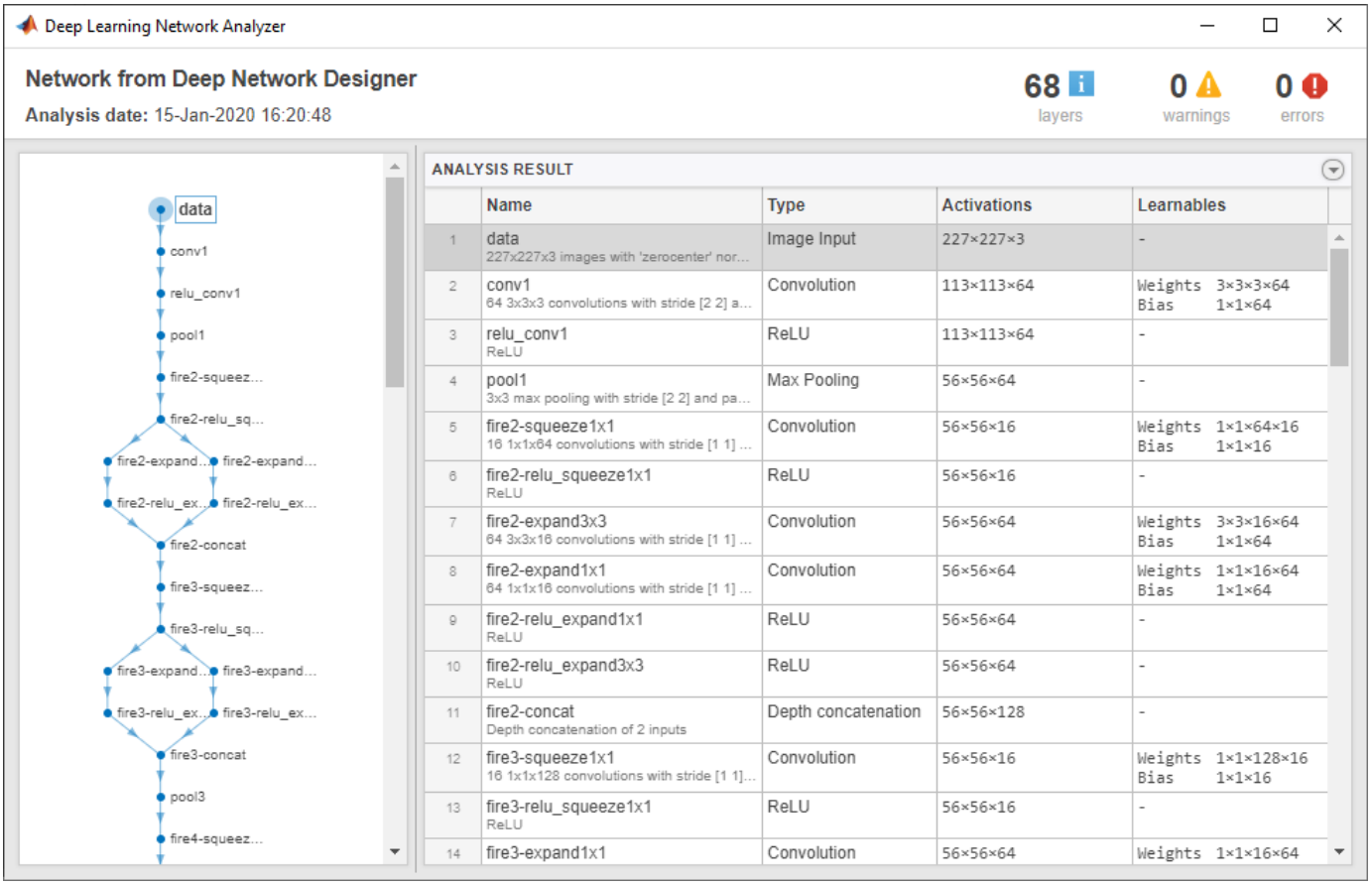
For transfer learning, you need to replace the output layer. Scroll to the end of the **Layer Library** and drag a new `classificationLayer` onto the canvas. Delete the original `output` layer and connect your new layer in its place.

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

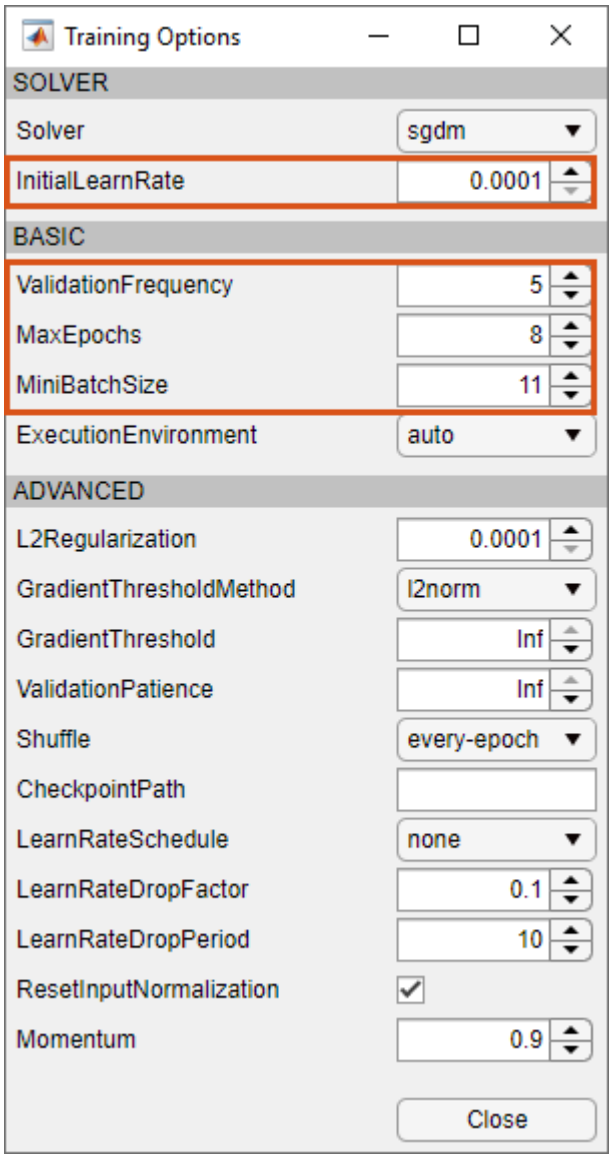
The Deep Network Designer app enables you to train image classification networks imported or created in the app. For other types of data, you can construct the network in Deep Network Designer and then export the network for training. For an example of showing how to use Deep Network Designer to construct a sequence network and then export the results for training, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-22.

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 `MiniBatchSize` and `ValidationFrequency`.

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

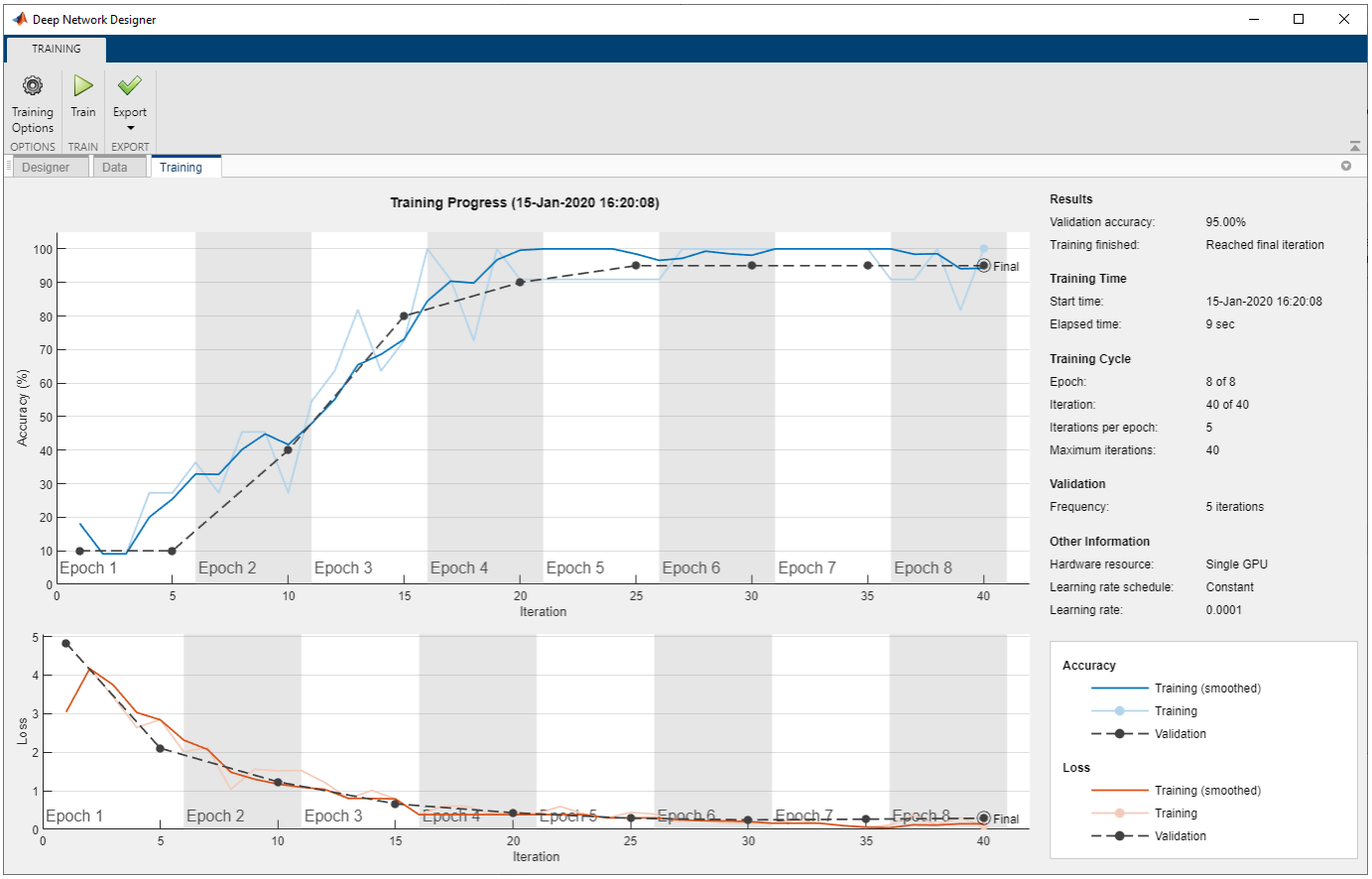
- Set `InitialLearnRate` to a small value to slow down learning in the transferred layers.
- Specify `ValidationFrequency` so that the accuracy on the validation data is calculated once every epoch.
- Specify a small number of epochs. An epoch is a full training cycle on the entire training data set. 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 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: [1×40 double]
          TrainingAccuracy: [1×40 double]
            ValidationLoss: [4.8267 NaN NaN NaN 2.1034 NaN NaN NaN NaN 1.2332 NaN NaN NaN NaN 0
        ValidationAccuracy: [10 NaN NaN NaN 10 NaN NaN NaN NaN 40 NaN NaN NaN NaN 80 NaN NaN Na
            BaseLearnRate: [1×40 double]
       FinalValidationLoss: 0.2893
   FinalValidationAccuracy: 95
```

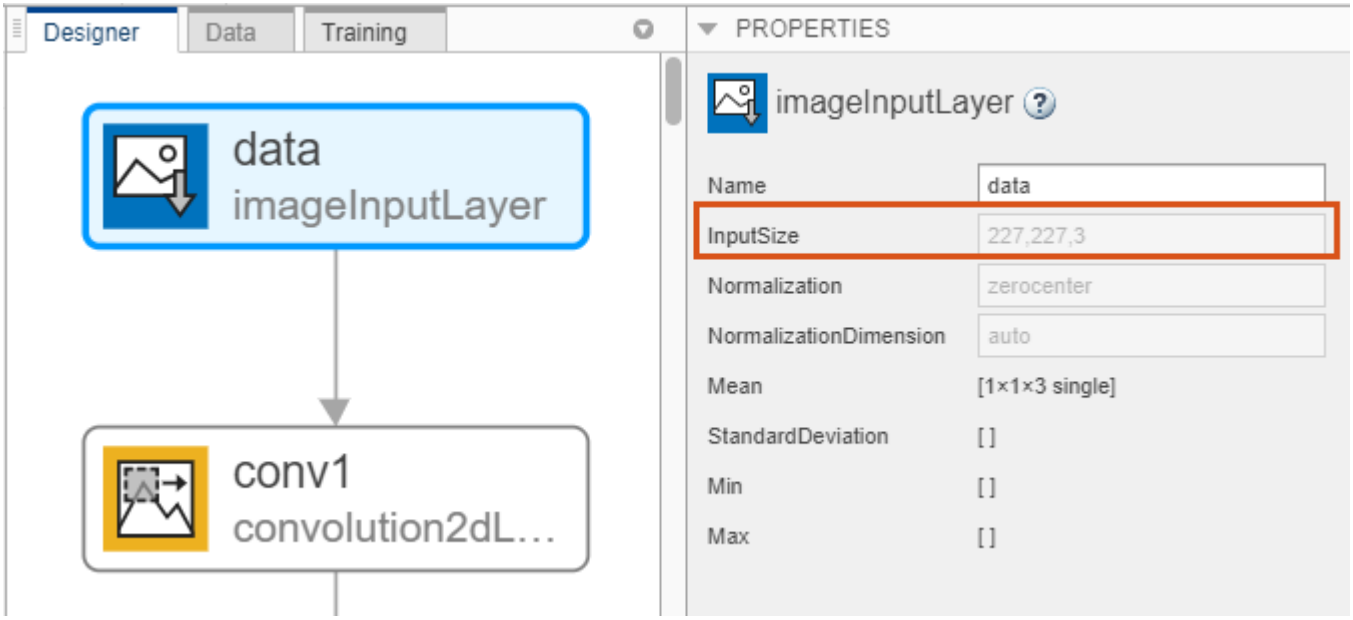
You can also generate MATLAB code, which recreates the network and the training options used. On the **Training** tab, select **Export > Generate Code for Training**. Examine the MATLAB code to learn how to programmatically prepare the data for training, create the network architecture, and train the 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
- “Generate MATLAB Code from Deep Network Designer” on page 2-31
- “Deep Learning Tips and Tricks” on page 1-45
- “List of Deep Learning Layers” on page 1-23

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 and edit networks.
- Build new networks from scratch.
- Drag and drop to add new layers and create new connections.
- View and edit layer properties.
- Generate MATLAB code to create the network architecture.

Tip Starting with a pretrained network and fine-tuning it with transfer learning is usually much faster and easier than training a new network from scratch. For an example showing how to perform transfer learning with a pretrained network, see “Transfer Learning with Deep Network Designer” on page 2-2.

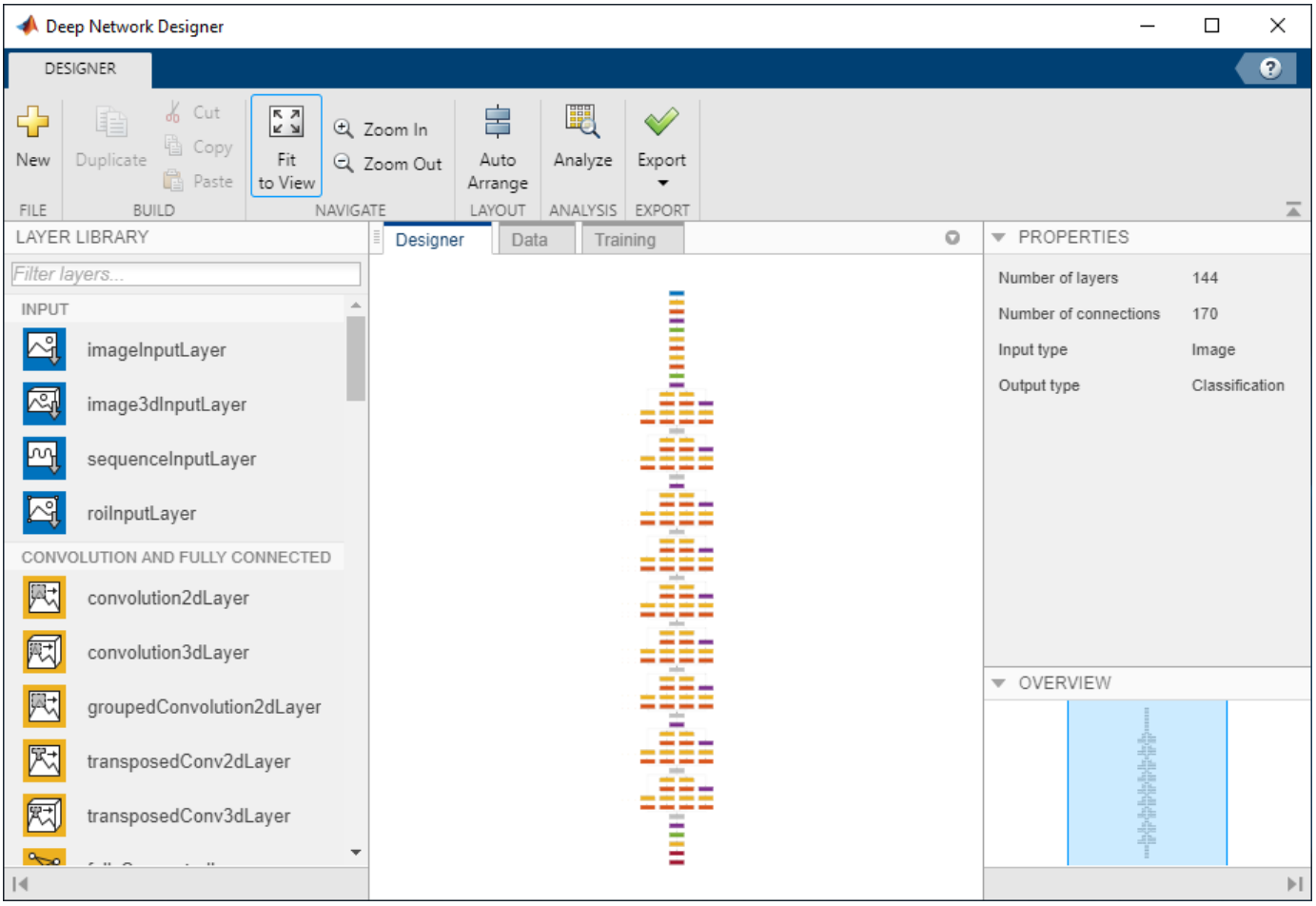
Open App and Import Networks

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
```

If you want to modify or copy an existing pretrained network, you can select it from the start page.

You can also click **New** on the **Designer** tab to load a network from the workspace, select a pretrained network, or build a network from scratch. Deep Network Designer displays a zoomed-out view of the whole network.



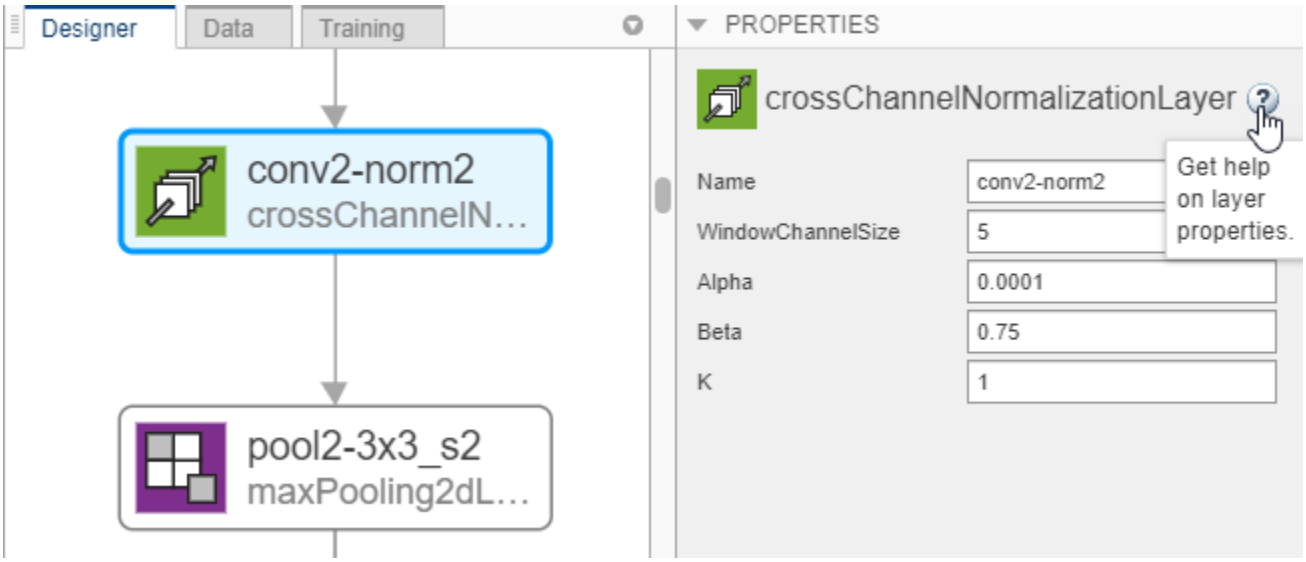
In the app, you can use any of the built-in layers to build a network. In addition, you can work with custom layers by creating them at the command line and then importing the network into the app. For a list of available layers and examples of custom layers, see “List of Deep Learning Layers” on page 1-23.

The **Designer** pane of Deep Network Designer is where you can construct, edit, and analyze your network.

Create and Edit a Network

Assemble a network by dragging blocks from the **Layer Library** and connecting them. You can work with blocks of layers at a time. Select multiple layers, then copy and paste or delete.

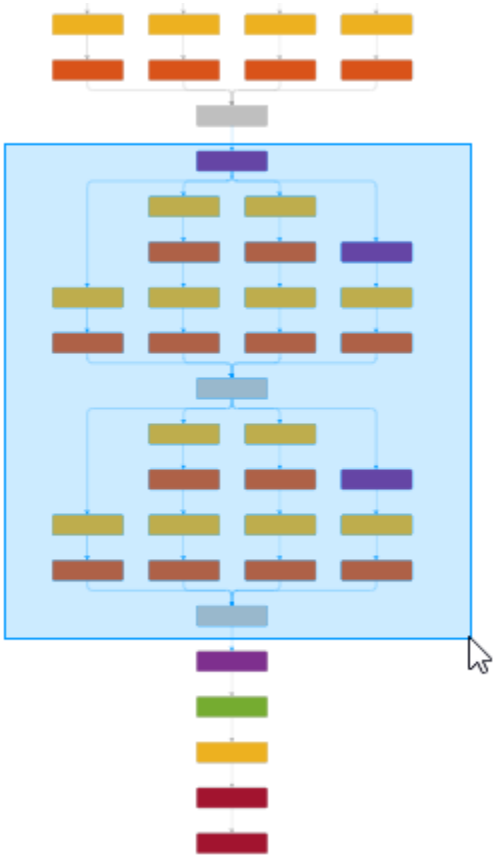
To view and edit layer properties, select a layer. Click the help icon next to the layer name for more information about the properties of the layer.



For information on all layer properties, click the layer name in the table on the “List of Deep Learning Layers” on page 1-23 page.

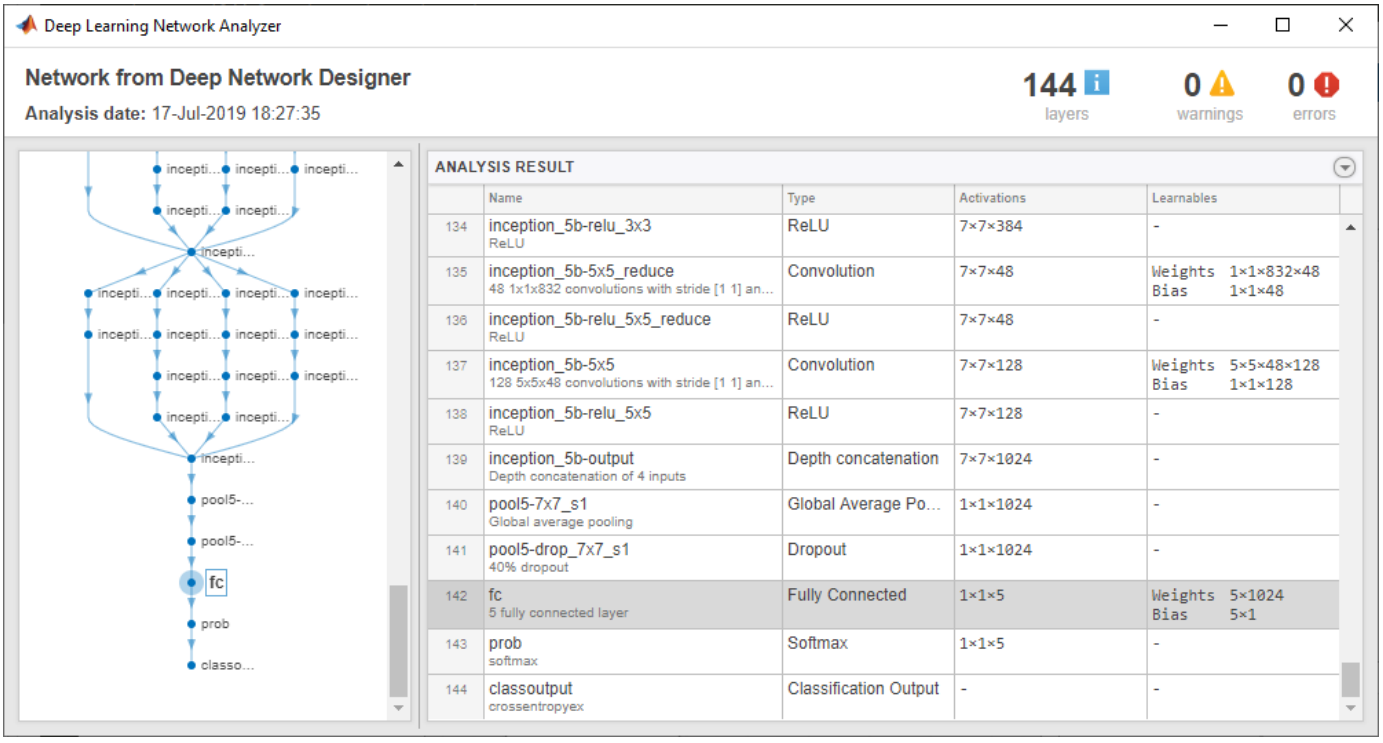
For tips on selecting a suitable network architecture, see “Deep Learning Tips and Tricks” on page 1-45.

Creating blocks of layers to copy and connect repeated units can be useful. For example, you can use blocks of layers to create multiple copies of groups of convolution, batch normalization, and ReLU layers. You can add layers to the end of pretrained networks to make them deeper. Alternatively, if you are working with small input images, you can edit a pretrained network to simplify it. For example, you can create a simpler network by deleting units of layers, such as inception modules, from a GoogLeNet network.



Check Network

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



Train Network Using Deep Network Designer

You can train the network for image classification problems, using the Deep Network Designer app. On the **Data** tab, click **Import Data** to select the image data on which you want to train the network.

After you import your data, on the **Training** tab, click **Train**. Deep Network Designer copies the network you construct in the **Designer** pane and then trains the network. If you want greater control over the training, modify the training options by clicking **Training Options**. For more information on importing data and training a network constructed in Deep Network Designer, see “Transfer Learning with Deep Network Designer” on page 2-2.

Tip Deep Network Designer can train image classification networks. For an example showing how to export a network and train it for sequence classification problems, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-22.

Export Network

To export the untrained network to the workspace for training, on the **Designer** tab, click **Export**. The Deep Network Designer app exports the network to a new variable containing the edited network layers. After exporting, you can supply the layer variable to the `trainNetwork` function.

For this example, assume that the layers exported from the app are named `lgraph_1`, the images are in an augmented image datastore called `images` and `options` contains the training options. To train the network, type:

```
trainedNet = trainNetwork(images,lgraph_1,options)
```

For command-line examples showing how to set training options and assess trained network accuracy, see “Create Simple Deep Learning Network for Classification” on page 3-40 or “Train Residual Network for Image Classification” on page 3-13.

To export the trained network to the workspace, on the **Training** tab, click **Export**. The exported network has layers with weights trained by Deep Network Designer.

Generate MATLAB Code

Using Deep Network Designer, you can generate MATLAB code to recreate the network construction and training performed in the app.

For an example showing how to generate MATLAB code that recreates the network architecture, see “Generate MATLAB Code to Recreate Network Layers” on page 2-31.

For an example showing how to generate MATLAB code that recreates the network architecture and the network training, see “Generate MATLAB Code to Train Network” on page 2-31.

See Also

Deep Network Designer

Related Examples

- “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-22
- “List of Deep Learning Layers” on page 1-23
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-31
- “Deep Learning Tips and Tricks” on page 1-45

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 interactively.
- 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] and [2]. 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)
```

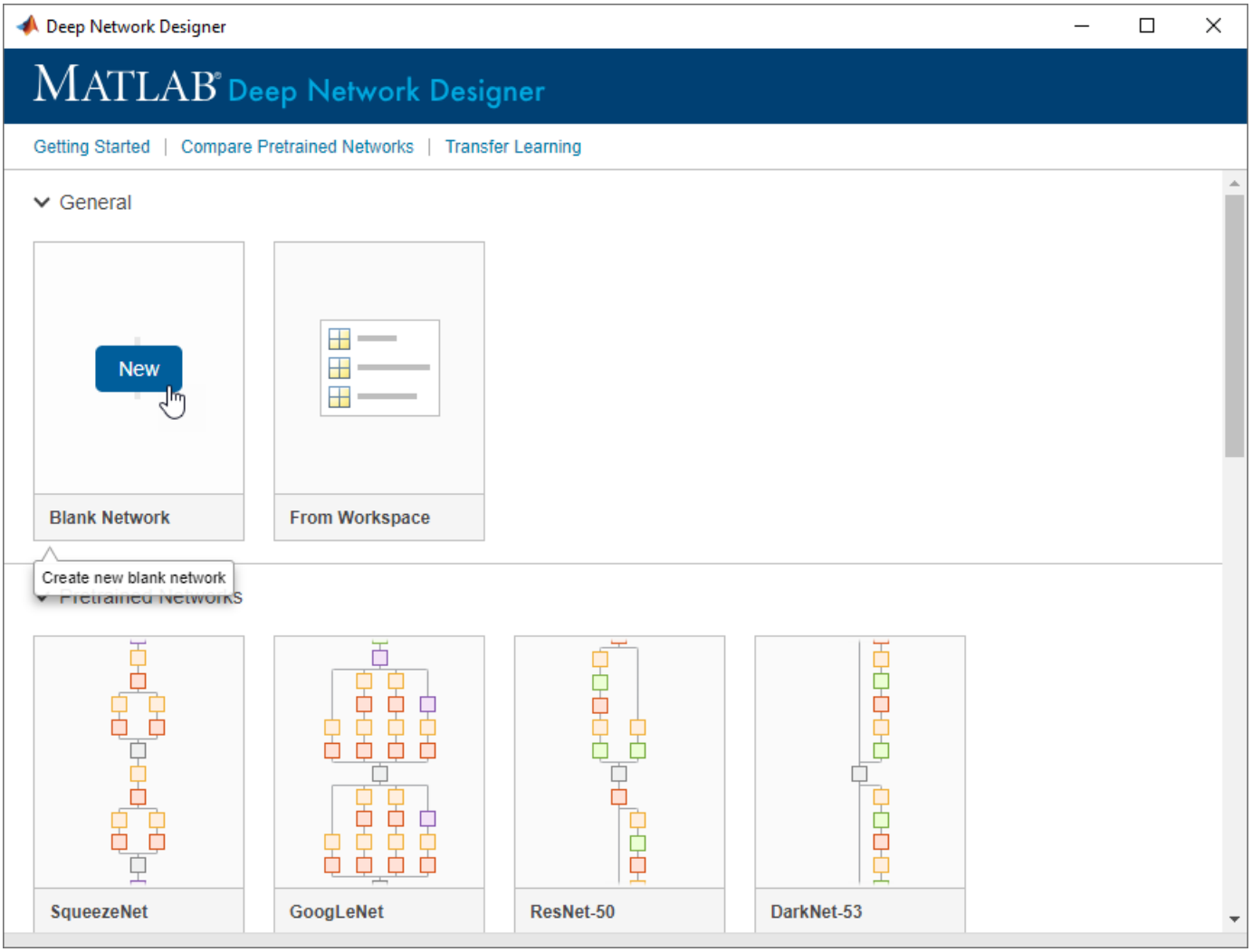
```
ans=5×1 cell array
    {12×20 double}
    {12×26 double}
    {12×22 double}
    {12×20 double}
    {12×21 double}
```

Define Network Architecture

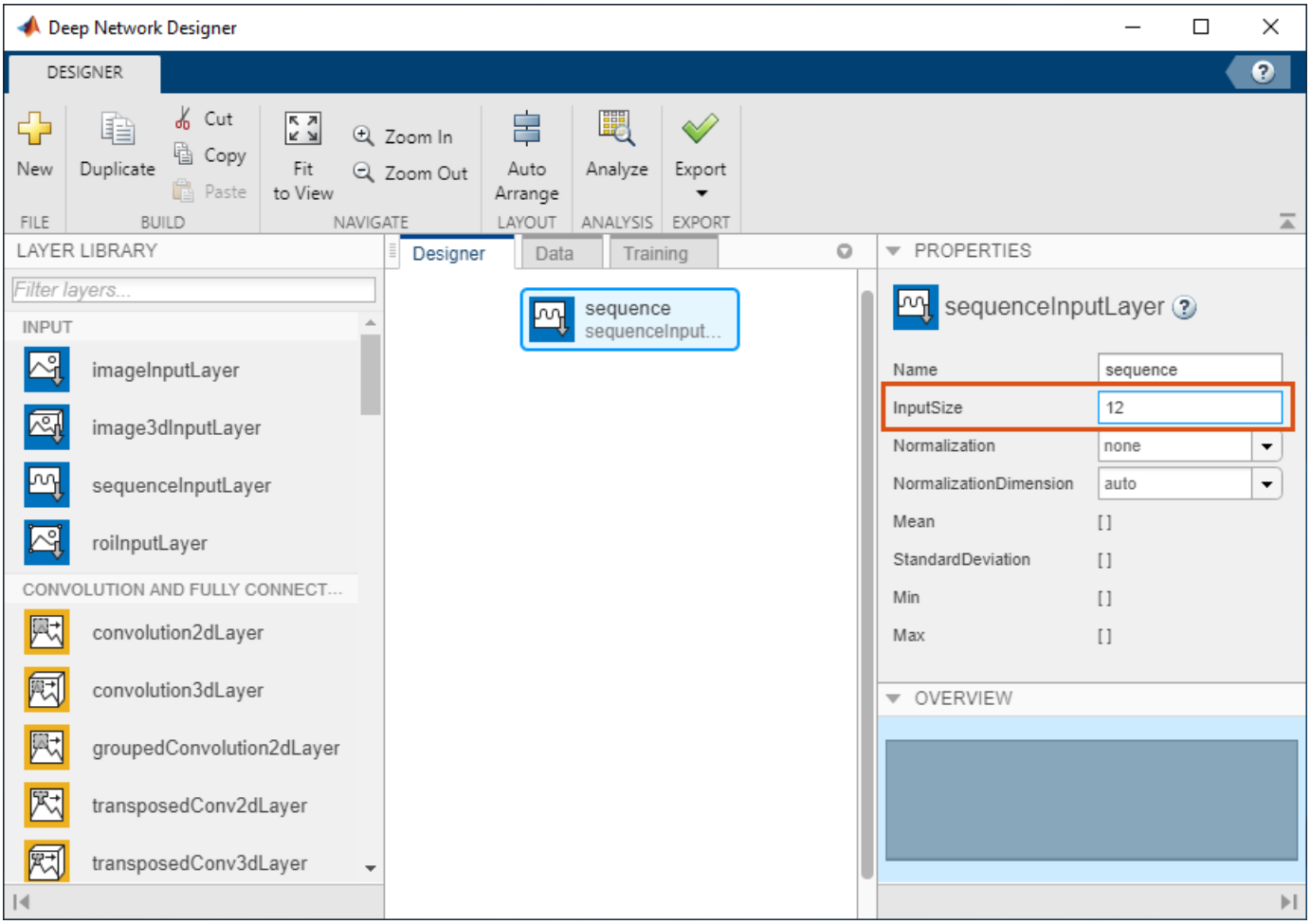
Open Deep Network Designer.

```
deepNetworkDesigner
```

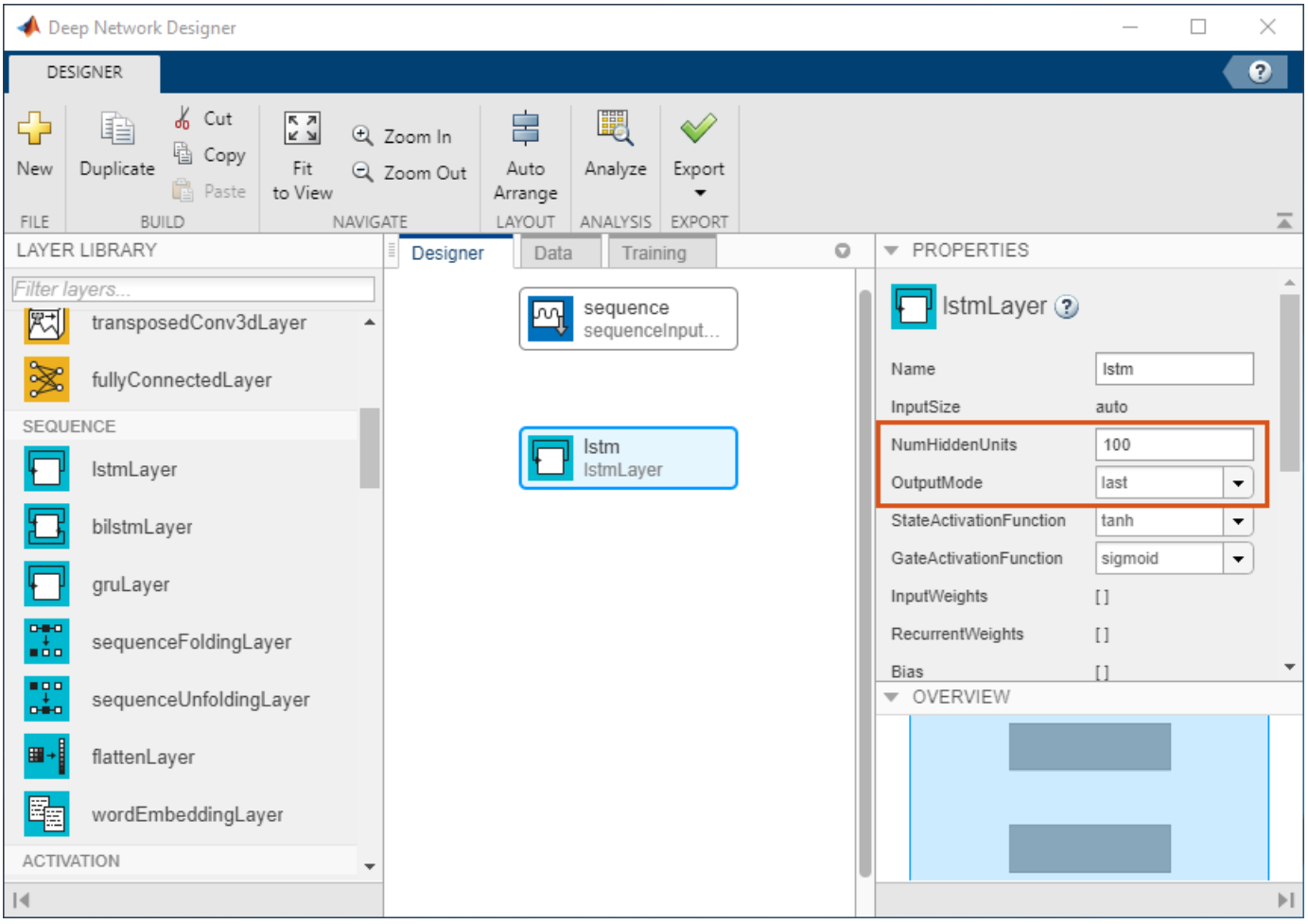
Select **Blank Network**.



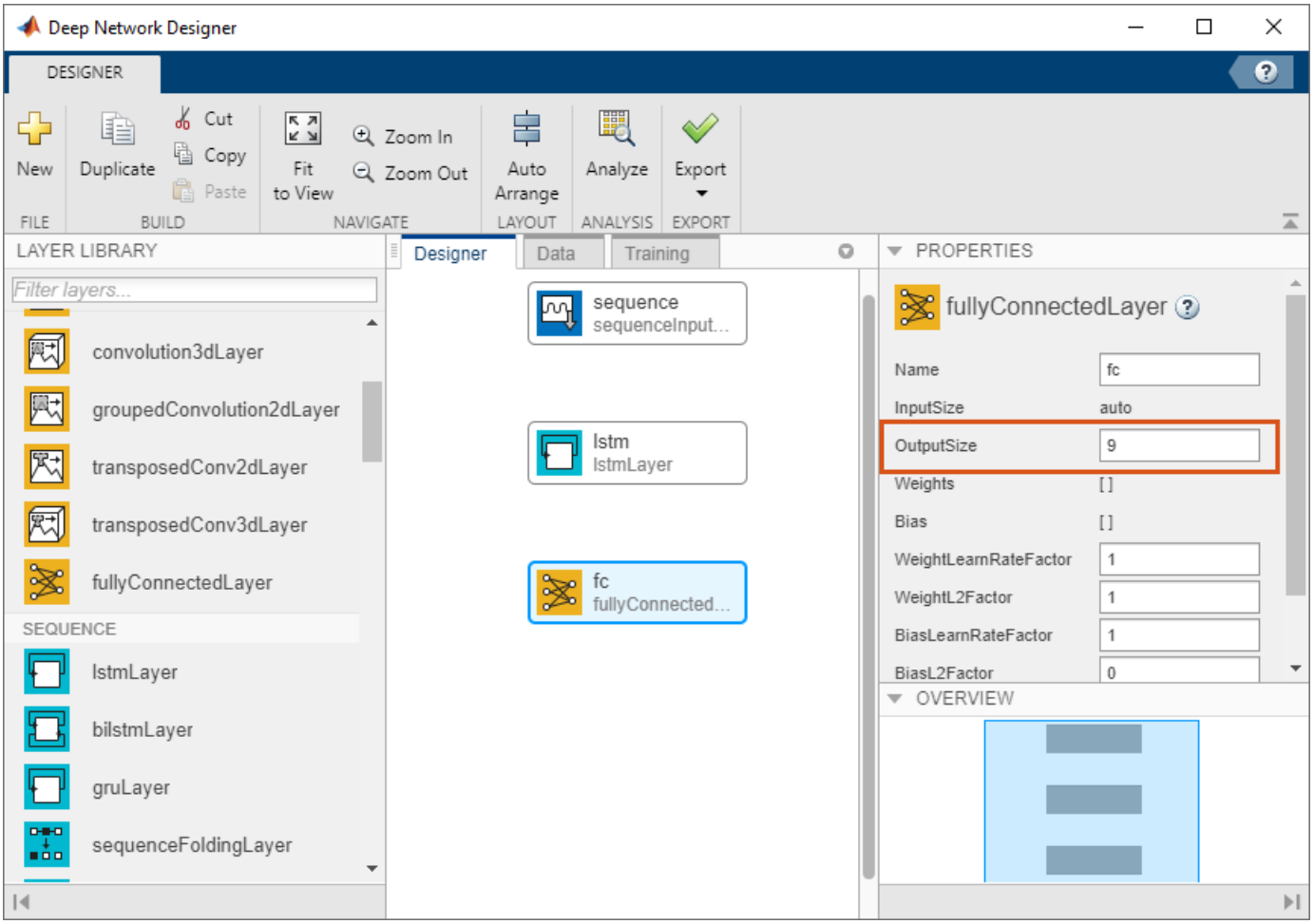
Drag a `sequenceInputLayer` to the canvas and set the `InputSize` to `12`, to match the feature dimension.



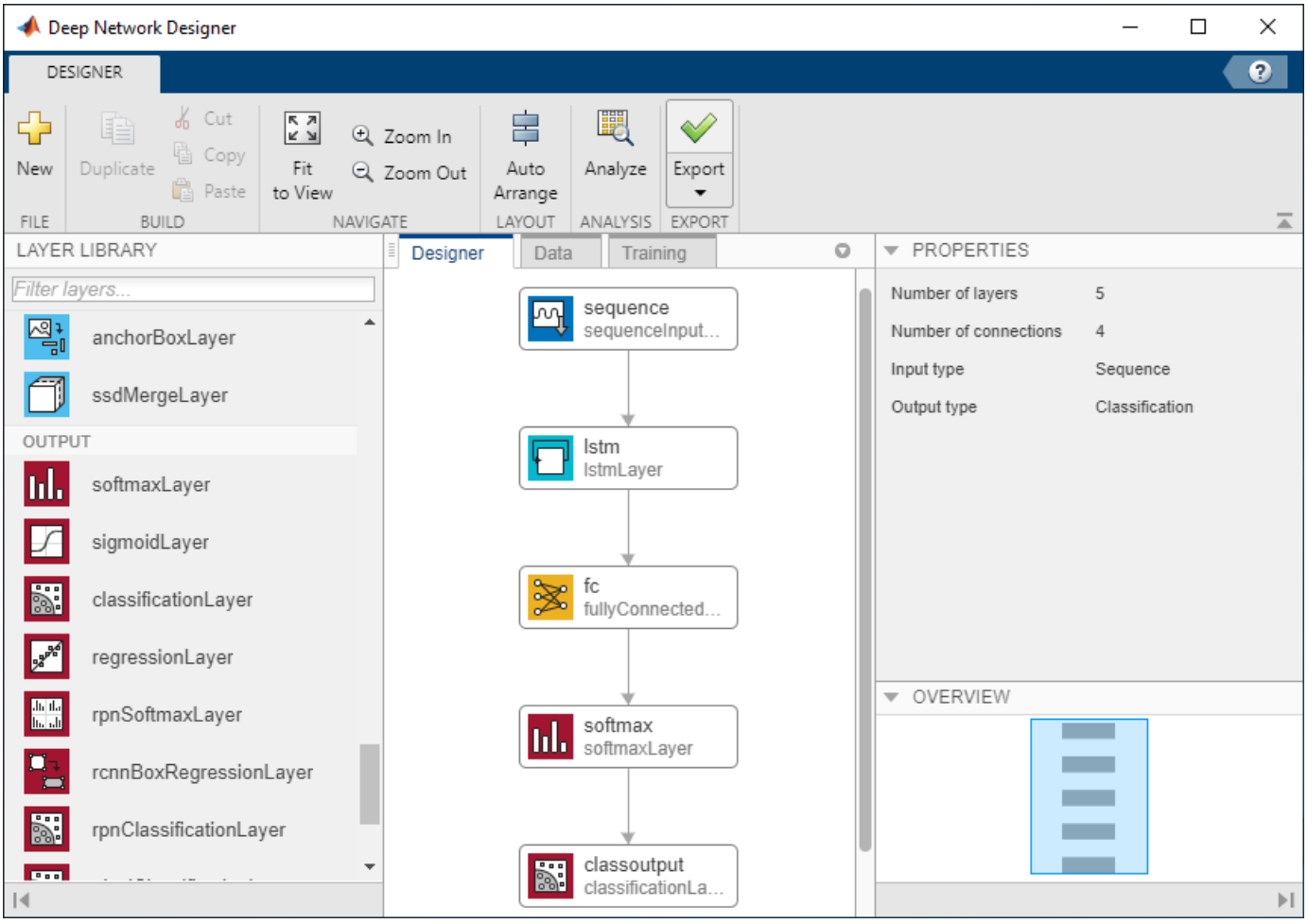
Then, drag an `lstmLayer` to the canvas. Set `NumHiddenUnits` to `100` and `OutputMode` to `last`.



Next, drag a `fullyConnectedLayer` onto the canvas and set `OutputSize` to `9`, the number of classes.

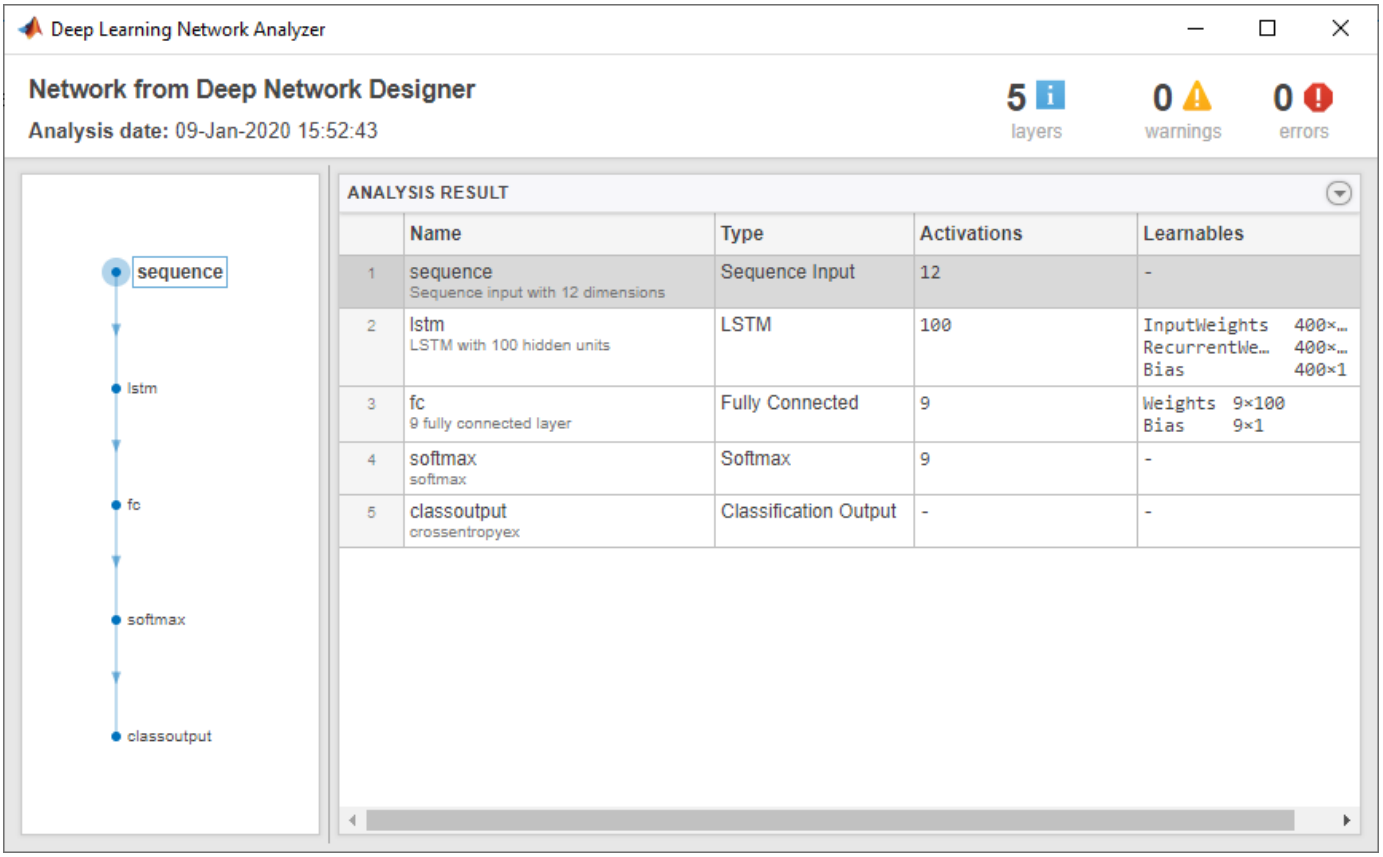


Finally, drag a `softmaxLayer` and a `classificationLayer` onto the canvas. Connect your layers to create a series network.



Check Network Architecture

To check the network and examine more details of the layers, click **Analyze**. If the Deep Learning Network Analyzer reports zero errors, then the edited network is ready for training.



Export Network Architecture

To export the network architecture, 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

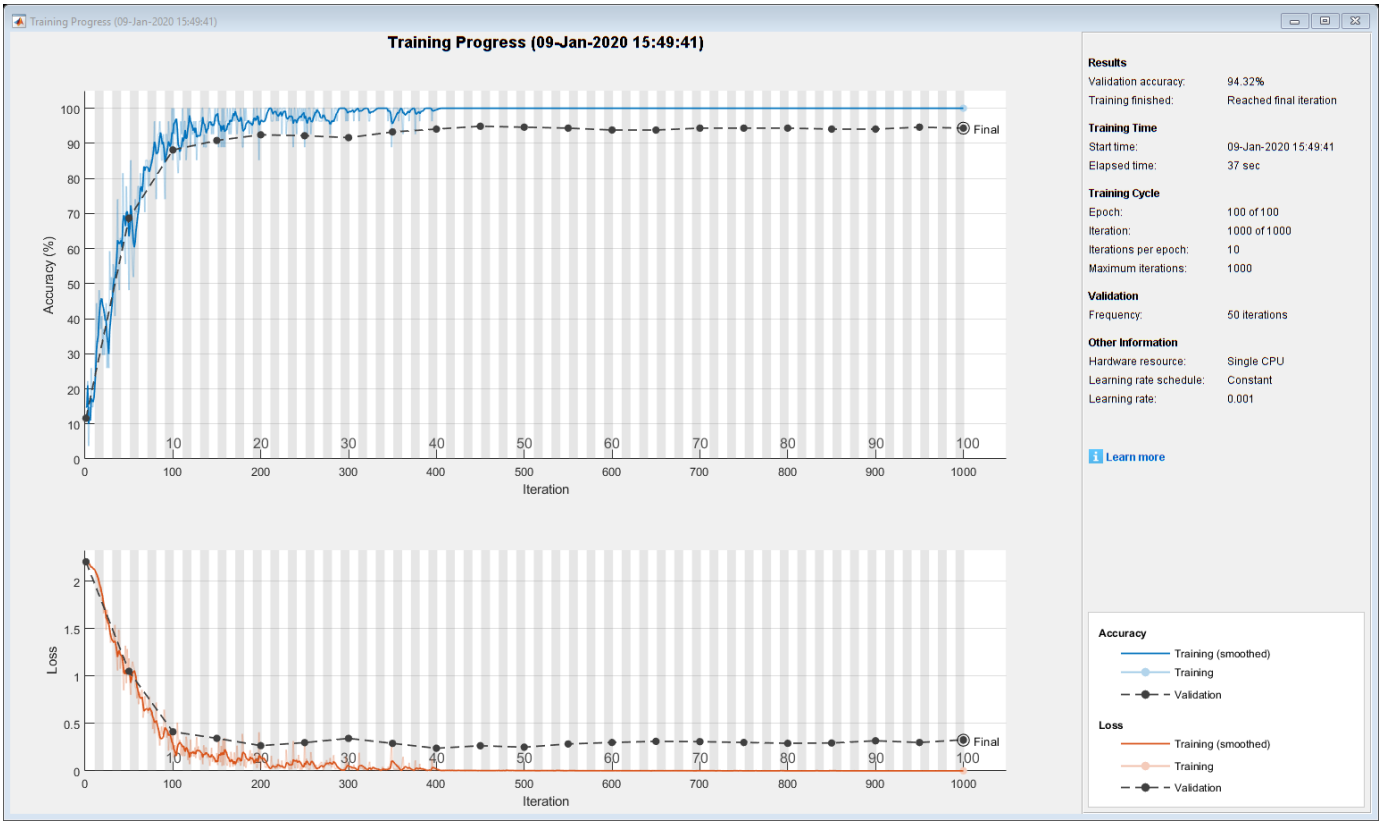
Specify the training options and train the network.

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);
```



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 1-53.

For an example showing how to use convolutional networks to classify sequence data, see “Speech Command Recognition Using Deep Learning” on page 4-17.

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 1-23
- “Transfer Learning with Deep Network Designer” on page 2-2
- “Generate MATLAB Code from Deep Network Designer” on page 2-31
- “Deep Learning Tips and Tricks” on page 1-45

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 an image classification 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`.

You can supply the generated layer graph or layer array to the `trainNetwork` function. For example, assume that the layers produced by the generated code are named `lgraph_1`, the images are in an augmented image datastore called `images` and `options` contains the training options. To train the network, type:

```
trainedNet = trainNetwork(images,lgraph_1,options)
```

For an example of training a network exported from Deep Network Designer, see “Create Simple Sequence Classification Network Using Deep Network Designer” on page 2-22.

Generate MATLAB Code to Train Network

To recreate the construction and training of an image classification network in Deep Network Designer, generate MATLAB code. For an example of using Deep Network Designer to train a 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 as an image datastore 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.

Use Network for Prediction

Suppose that the trained network is contained in the variable `net`. To use the trained image classification network for prediction, use the `predict` function. For example, use the 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 | `trainNetwork` | `trainingOptions`

Related Examples

- “Transfer Learning with Deep Network Designer” on page 2-2
- “Build Networks with Deep Network Designer” on page 2-15

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

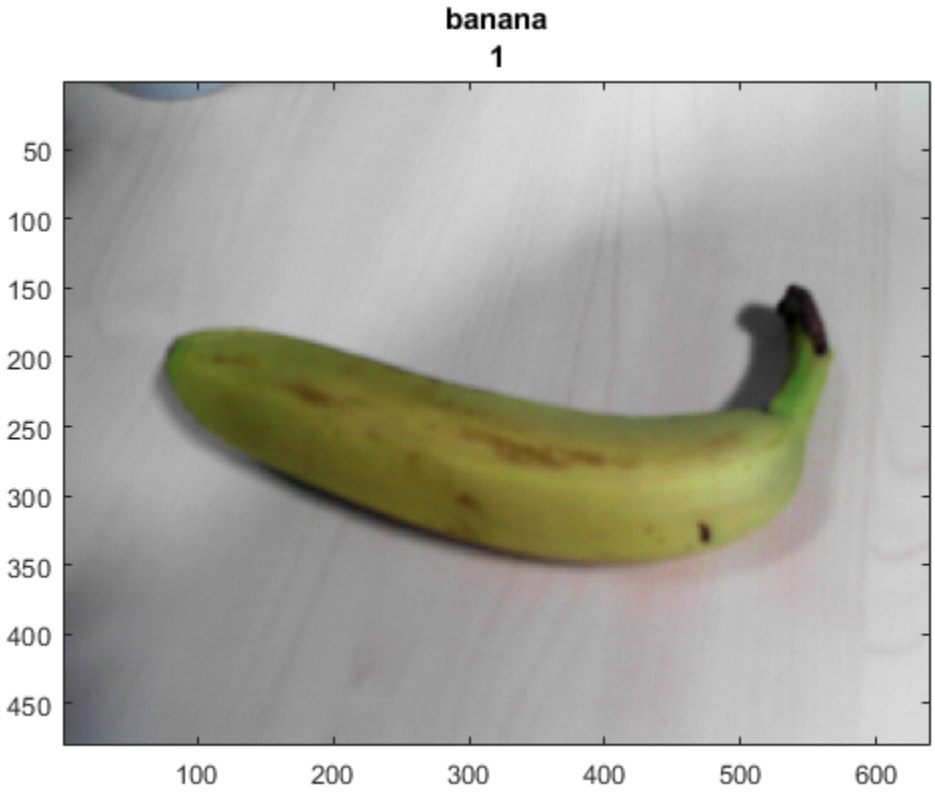
To classify an image, you must resize it to the input size of the network. Get the first two elements of the `InputSize` property of the image input layer of the network. The image input layer is the first layer of the network.

```
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.

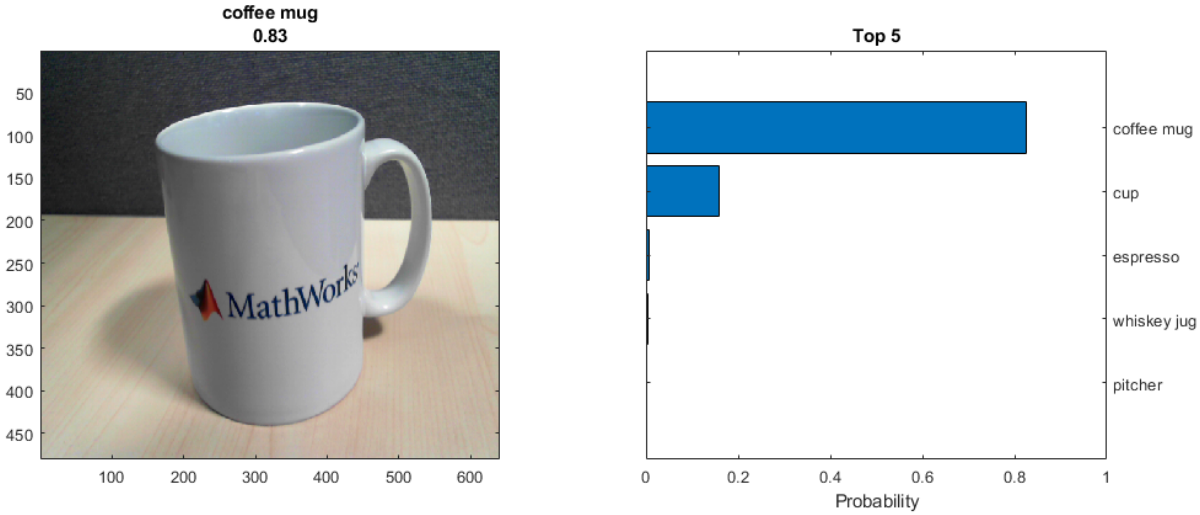
```
im = snapshot(camera);
image(ax1,im)
im = imresize(im,inputSize);
[label,score] = classify(net,im);
title(ax1,{char(label),num2str(max(score),2)});
```

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 `ActivePositionProperty` to `'position'`.

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
ax2.ActivePositionProperty = 'position';
```

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

`alexnet` | `vgg19`

Related Examples

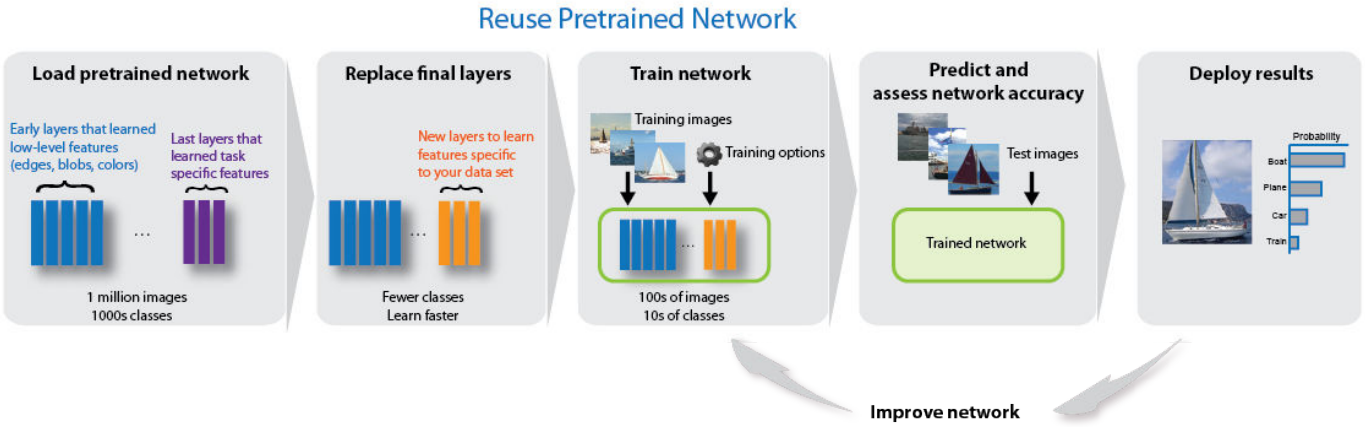
- “Transfer Learning Using AlexNet” on page 3-33
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Learning in MATLAB” on page 1-2

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

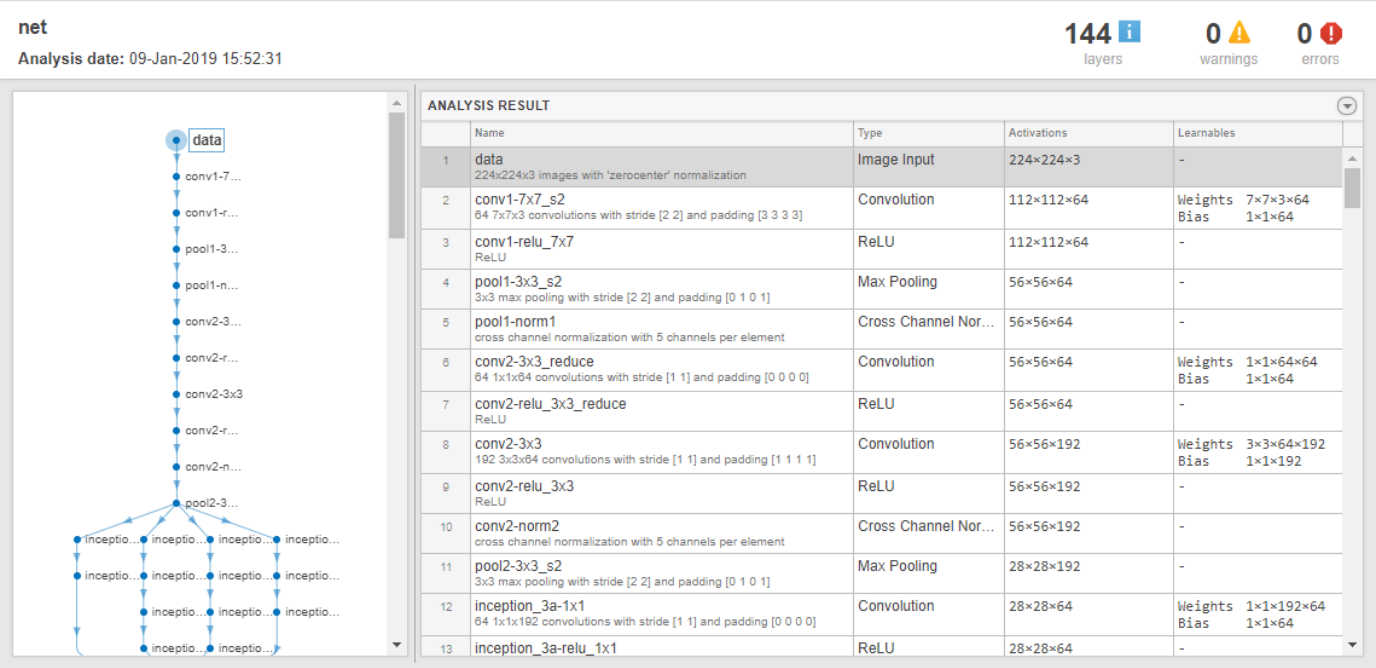
Load a pretrained GoogLeNet network. If the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

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 1-14.

```
net =
```

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'
                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.

Extract the layer graph from the trained network. If the network is a `SeriesNetwork` object, such as AlexNet, VGG-16, or VGG-19, then convert the list of layers in `net.Layers` to a layer graph.

```
if isa(net,'SeriesNetwork')
  lgraph = layerGraph(net.Layers);
else
  lgraph = layerGraph(net);
end
```

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 =
  1x2 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.

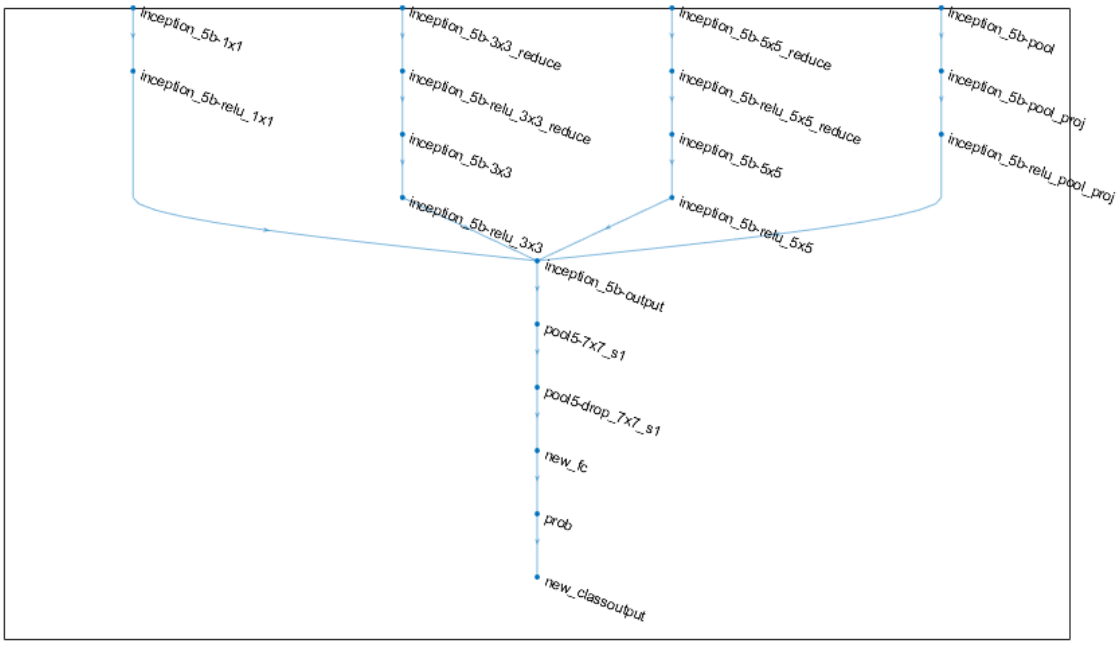
```
numClasses = numel(categories(imdsTrain.Labels));

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);
```

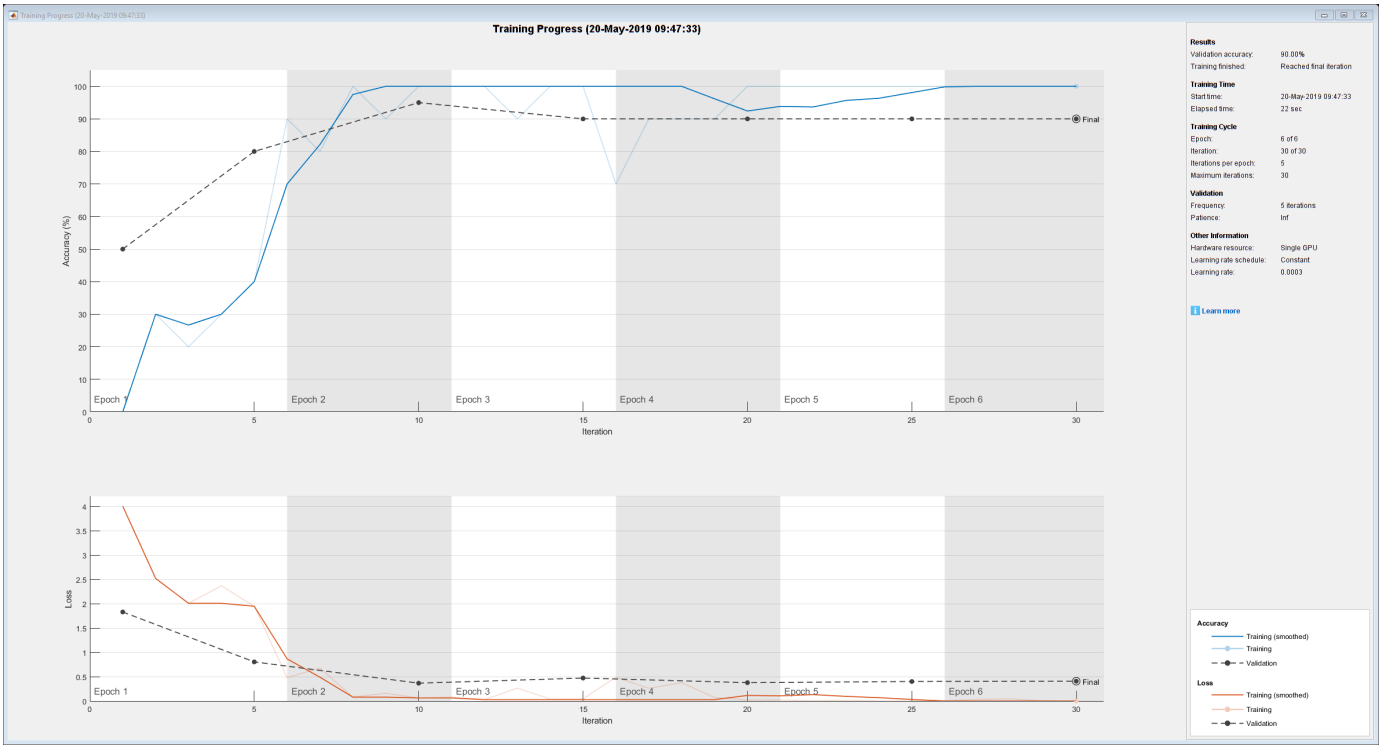
Specify the training options. Set `InitialLearnRate` to a small value to slow down learning in the transferred layers that are not already frozen. In the previous step, you increased the learning rate factors for the last learnable layer to speed up learning in the new final layers. This combination 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 (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). 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

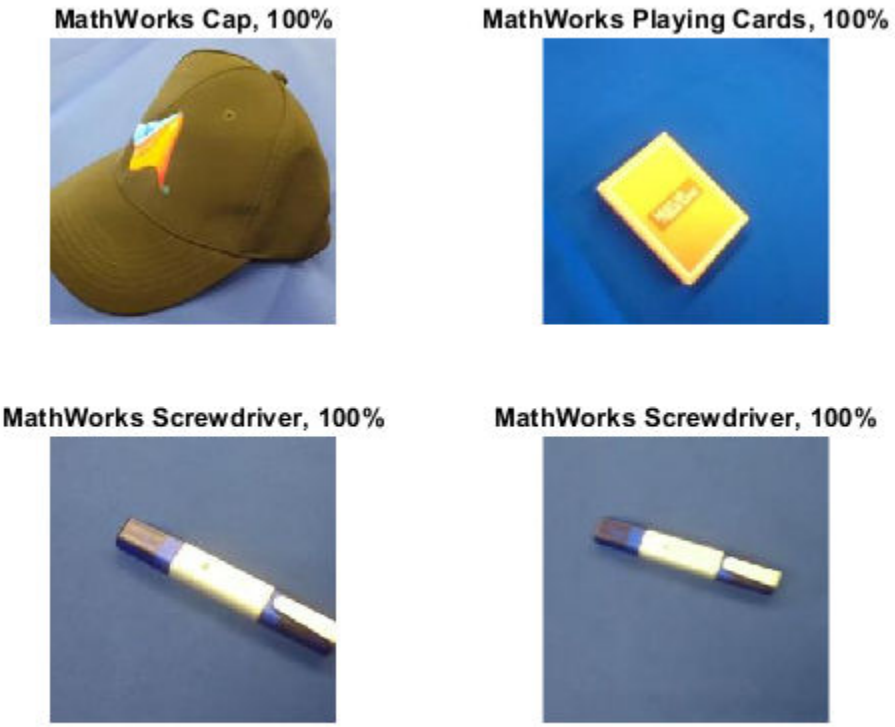
Classify the validation images using the fine-tuned network, and calculate the classification accuracy.

```
[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

`DAGNetwork` | `alexnet` | `analyzeNetwork` | `googlenet` | `importCaffeLayers` | `importCaffeNetwork` | `layerGraph` | `plot` | `trainNetwork` | `vgg16` | `vgg19`

Related Examples

- “Convert Classification Network into Regression Network” on page 3-66
- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-12
- “Transfer Learning Using AlexNet” 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`.

You can also load pretrained networks for image classification. For more information, see “Pretrained Deep Neural Networks” on page 1-12.

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);
```

```
Downloading CIFAR-10 dataset (175 MB). This can take a while...done.
```

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.

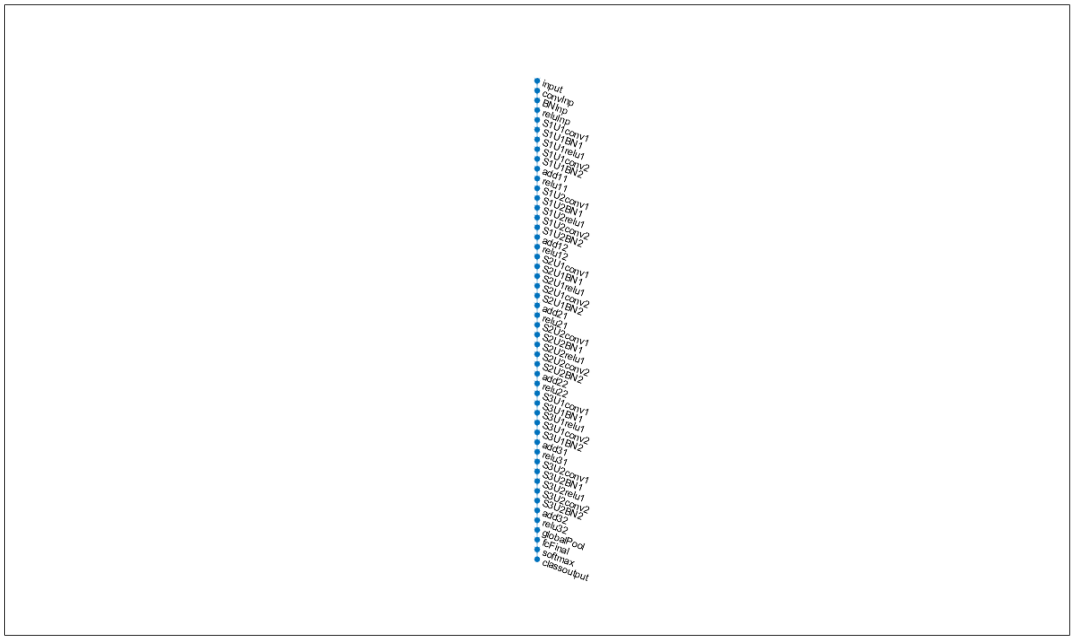
Give unique names to all the layers. The layers in the convolutional units have names starting with `'SjUk'`, where `j` is the stage index and `k` is the index of the convolutional unit within that stage. For example, `'S2U1'` denotes stage 2, unit 1.

```
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);
```



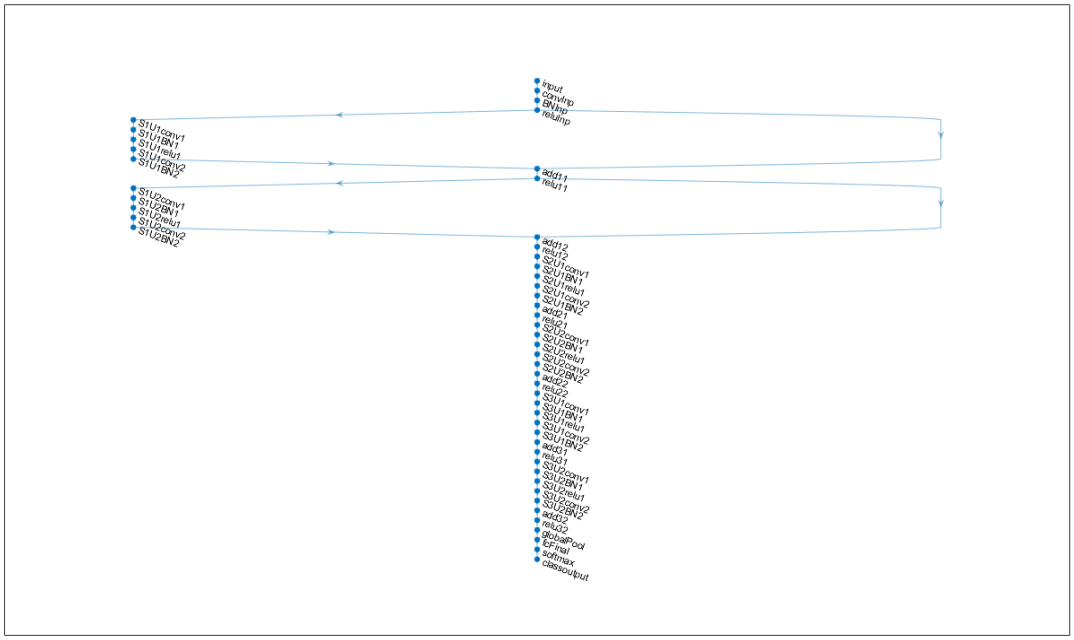
Create Residual Connections

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');
```

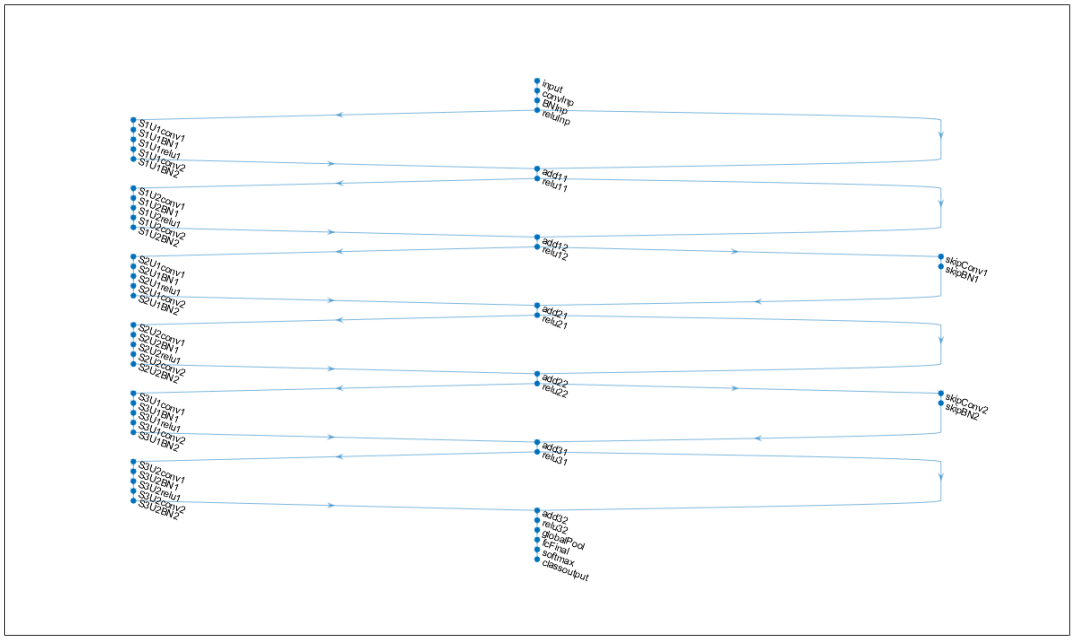
Change the activation size in the residual connection between the second and third stages by another 1-by-1 convolutional layer together with its batch normalization layer.

```
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

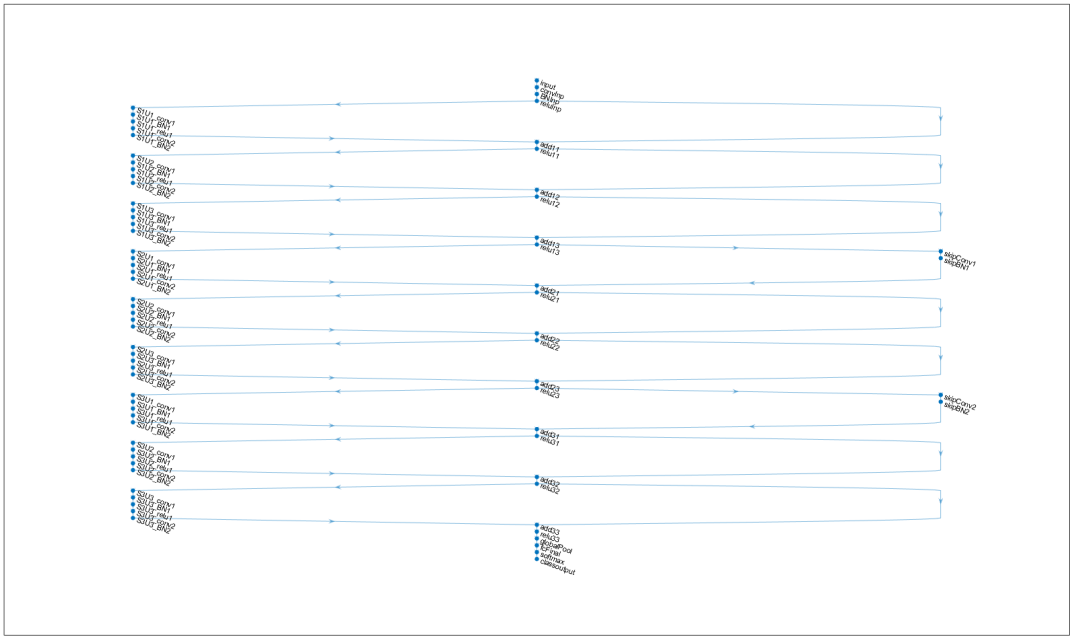
To create a layer graph with residual connections for CIFAR-10 data of arbitrary depth and width, use the supporting function residualCIFARlgraph.

`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

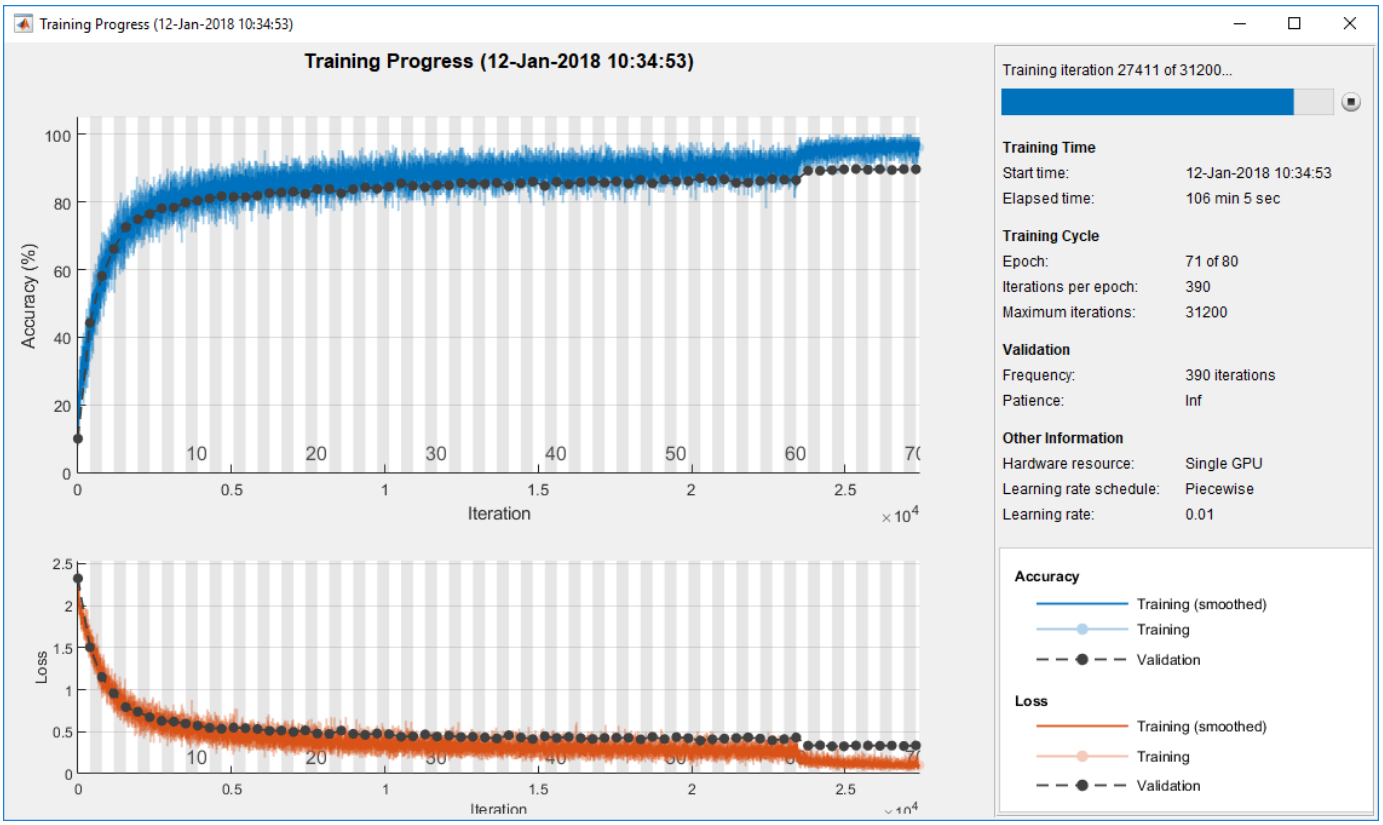
Specify training options. Train the network for 80 epochs. Select a learning rate that is proportional to the mini-batch size and reduce the learning rate by a factor of 10 after 60 epochs. Validate the network once per epoch using the validation data.

```
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 + "%")
```

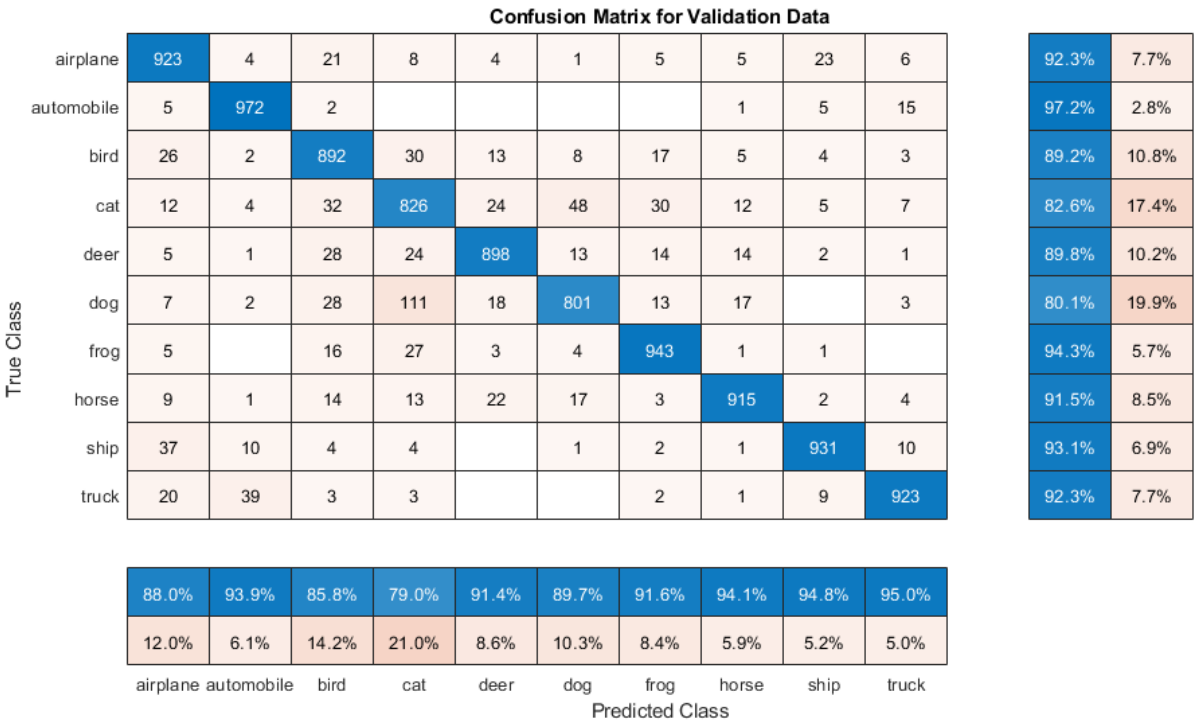
```
Training error: 2.862%
```

```
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

`analyzeNetwork` | `layerGraph` | `trainNetwork` | `trainingOptions`

Related Examples

- “Deep Learning Using Bayesian Optimization” on page 5-34
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Pretrained Deep Neural Networks” on page 1-12
- “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 1-14.

```
net =
```

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
```

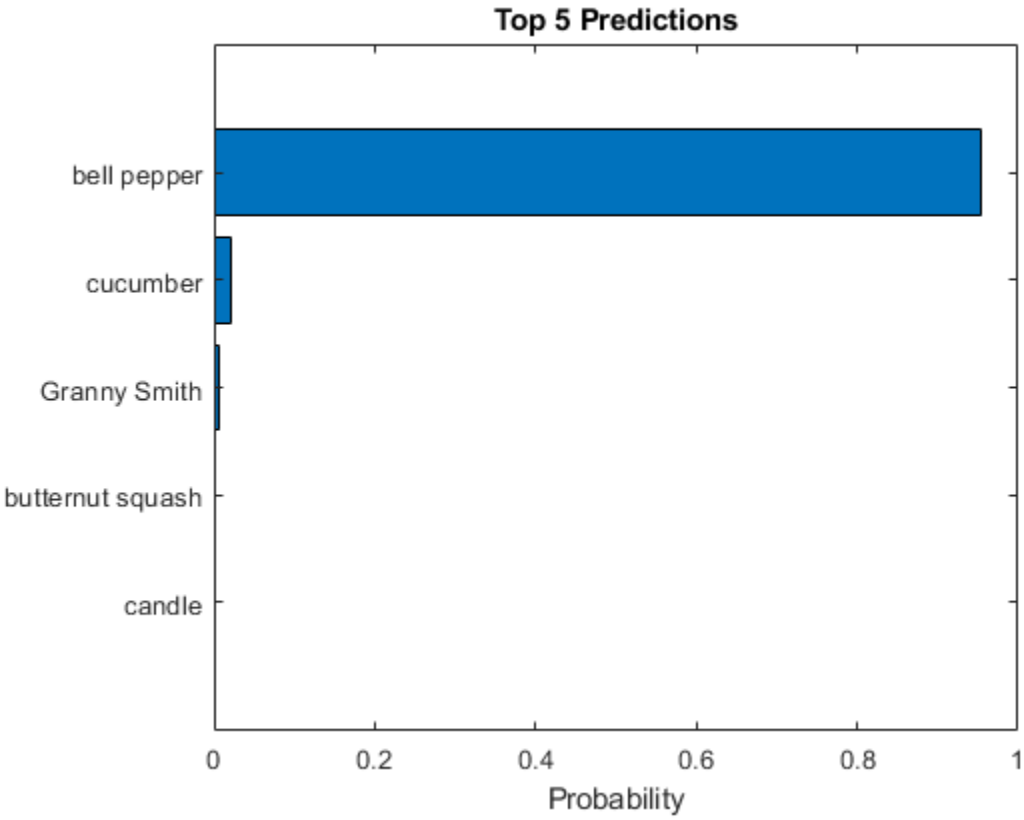
Display the image with the predicted label and the predicted probability of the image having that label.

```
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` | `alexnet` | `classify` | `googlenet` | `predict`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-12
- “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.

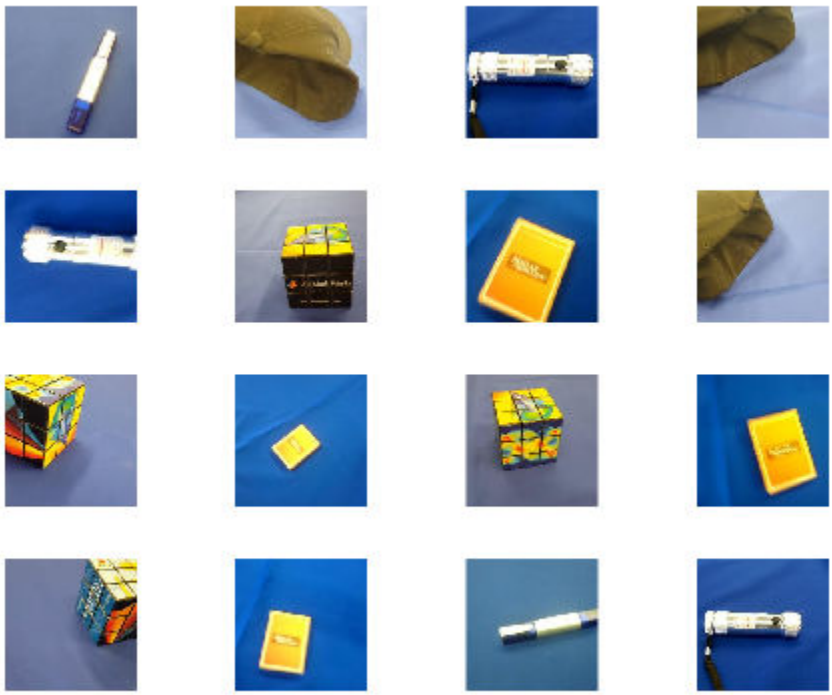
Load Data

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

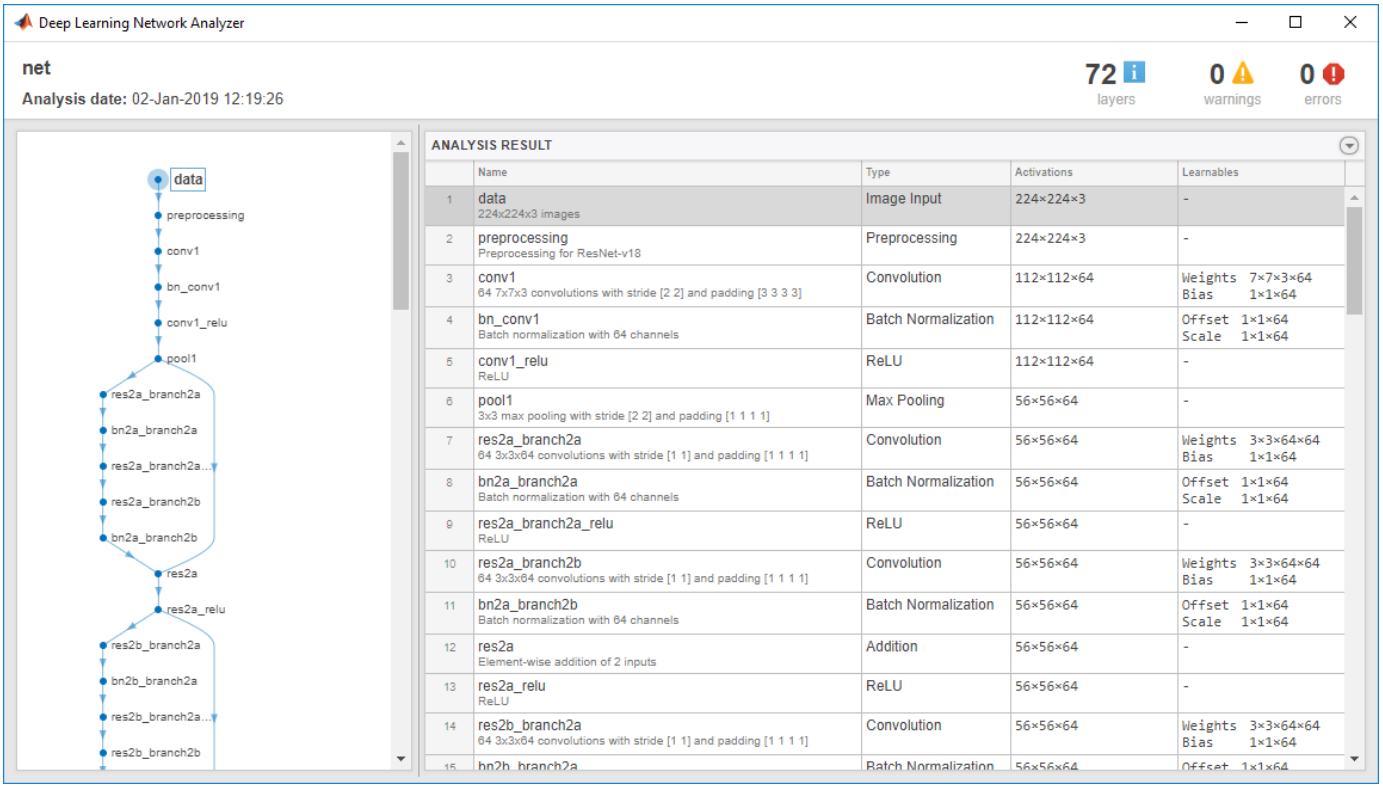
Load a pretrained ResNet-18 network. If the Deep Learning Toolbox Model *for ResNet-18 Network* support package is not installed, then the software provides a download link. ResNet-18 is trained on more than a million images and can classify images into 1000 object categories, such as keyboard, mouse, 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: [71×1 nnet.cnn.layer.Layer]
    Connections: [78×2 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);
```

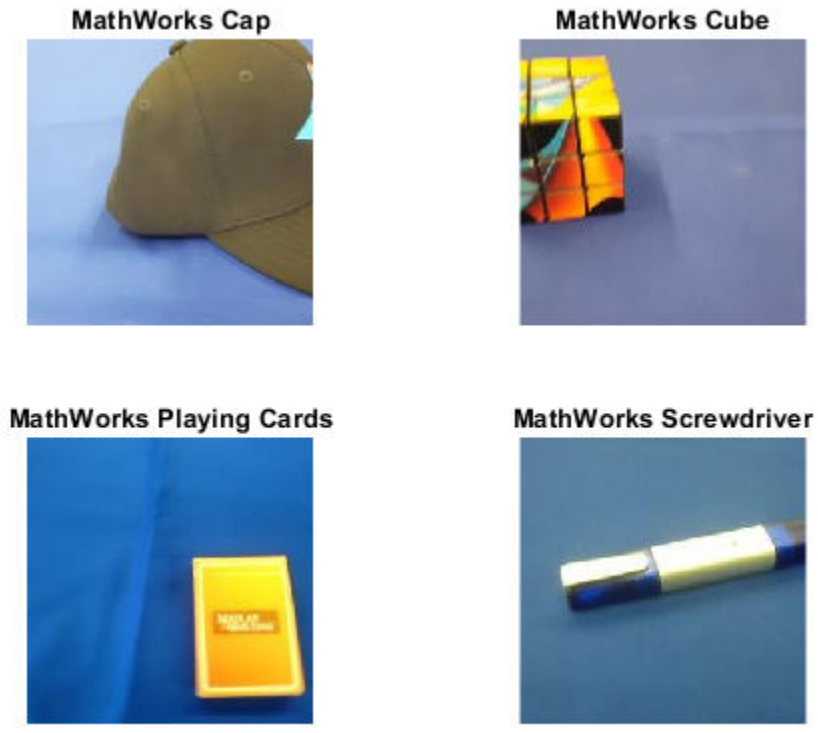
Classify Test Images

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 1-12.

See Also

`fitcecoc` | `resnet50`

Related Examples

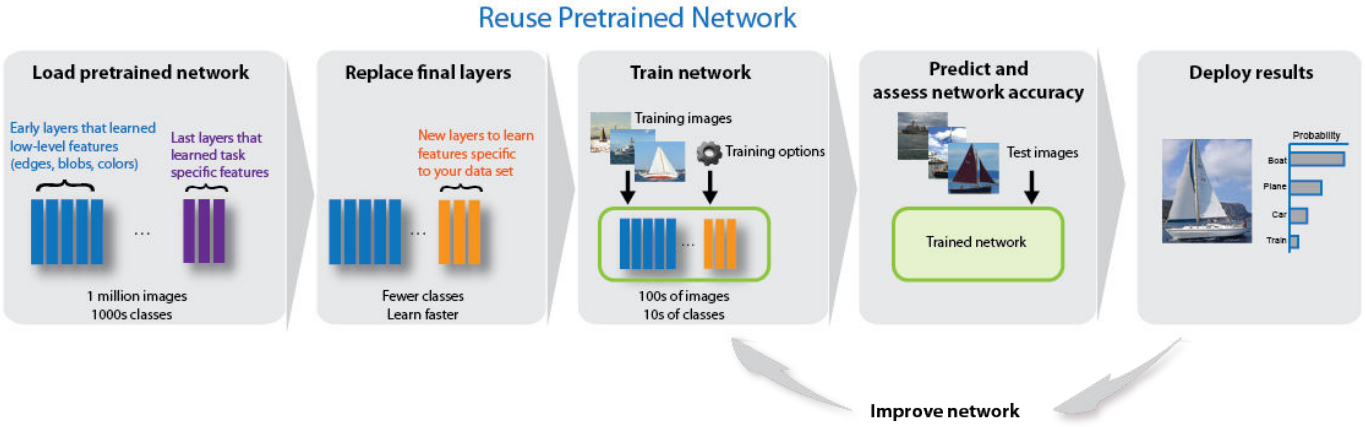
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Learning in MATLAB” on page 1-2

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');
```

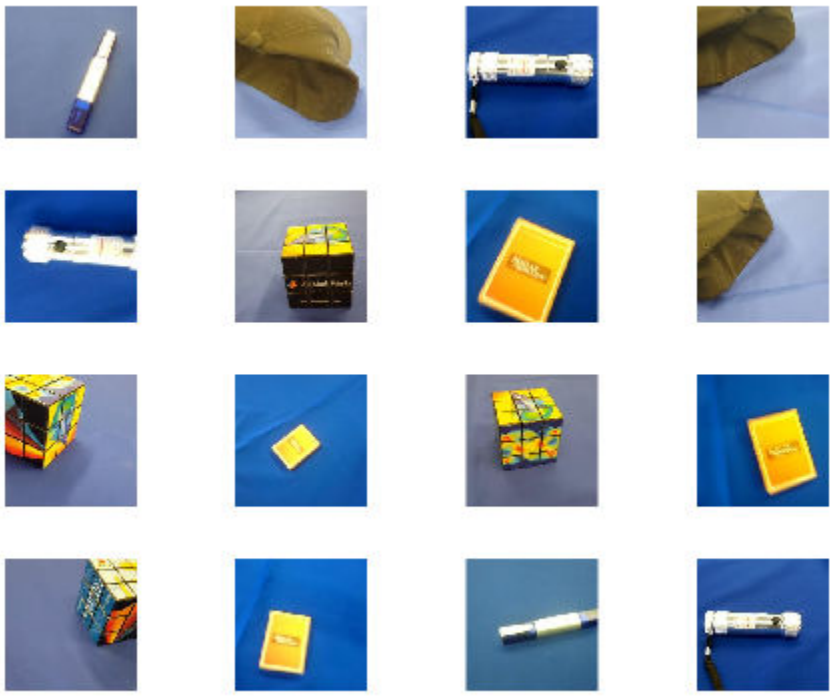
Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `images` datastore into two new datastores.

```
[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
```



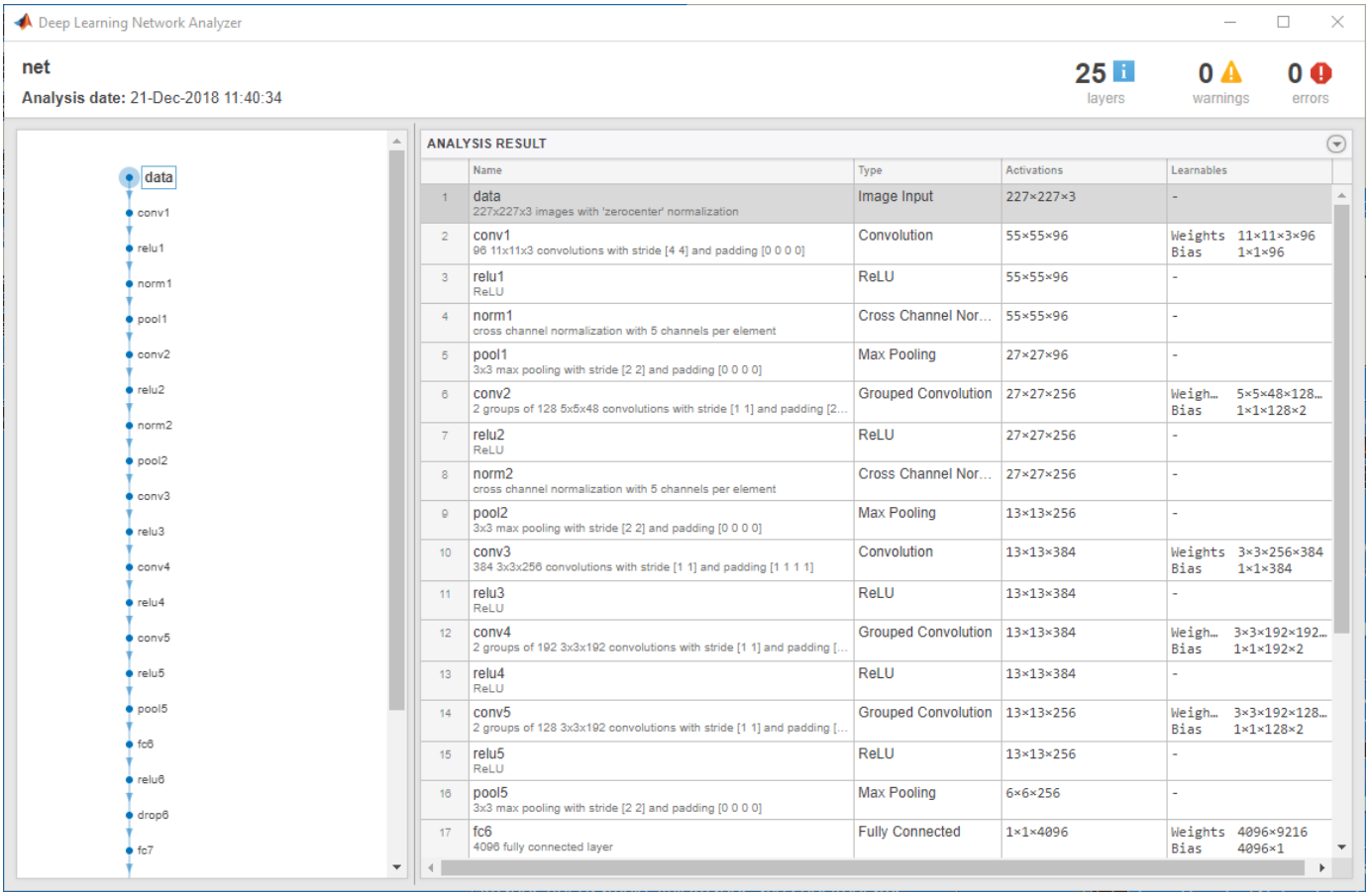
Load Pretrained Network

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.

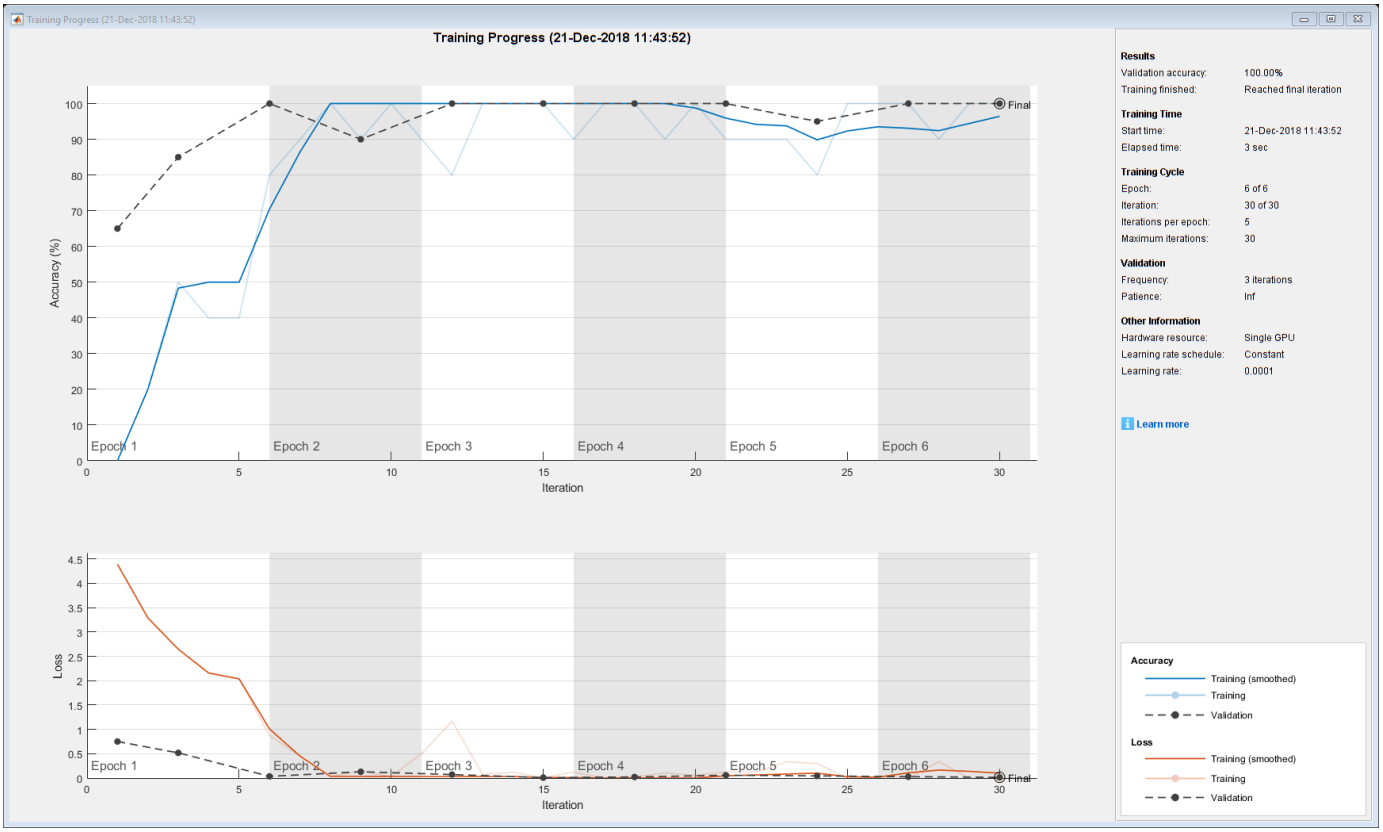
```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

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 (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). 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,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 1-45.

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

`alexnet` | `analyzeNetwork` | `trainNetwork` | `trainingOptions`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-19
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41

- “Extract Image Features Using Pretrained Network” on page 3-28
- “Pretrained Deep Neural Networks” on page 1-12
- “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.

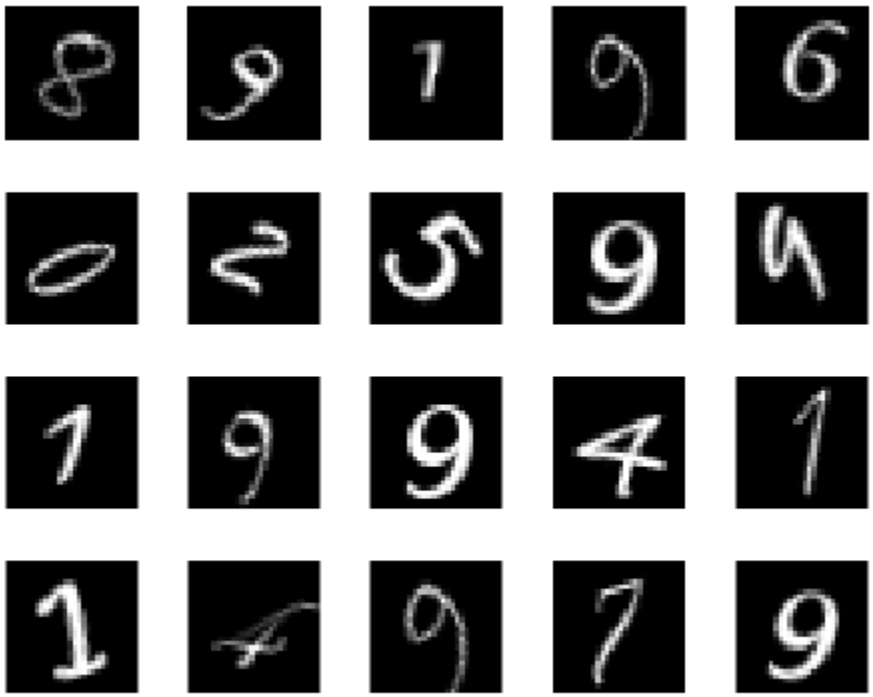
Load and Explore Image Data

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', ...
```

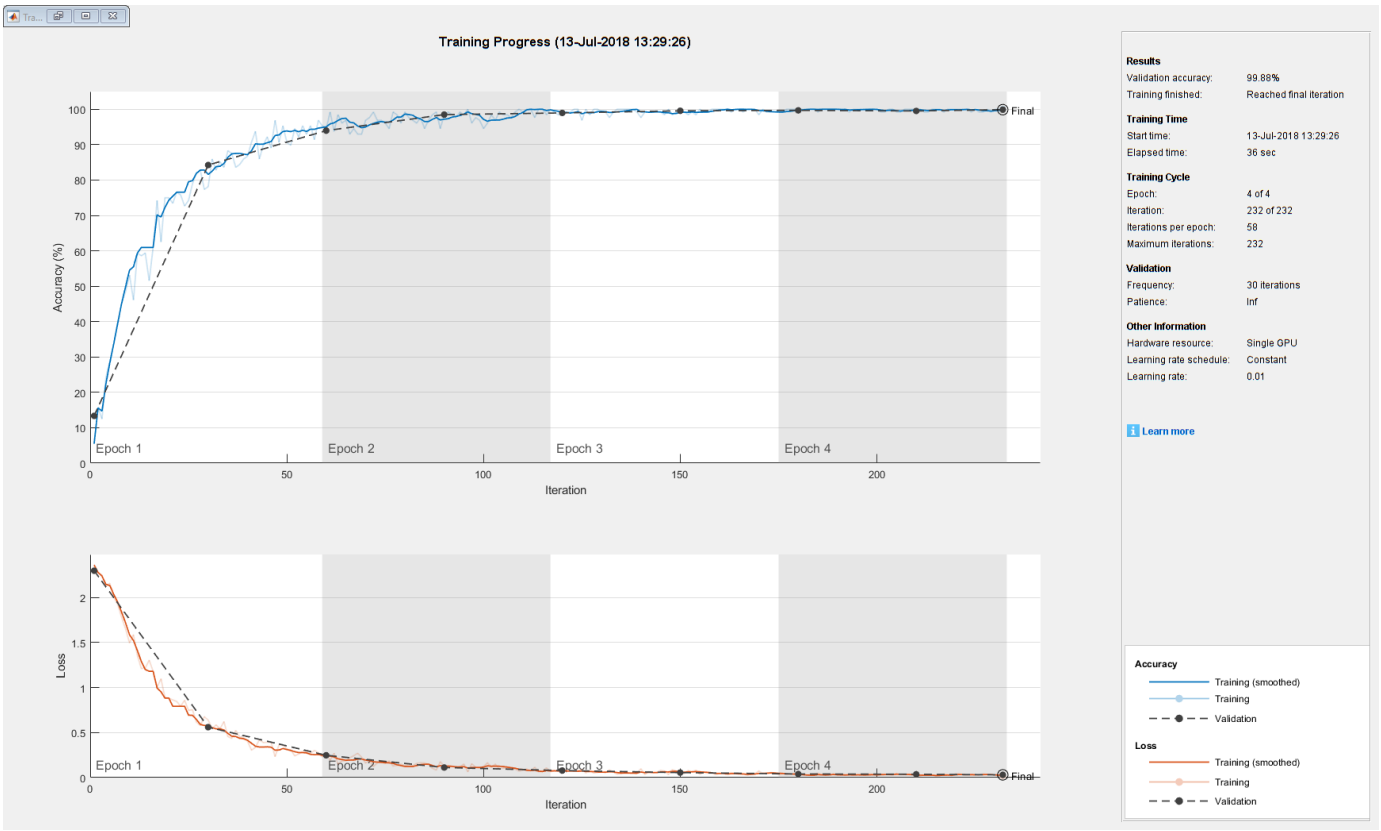
```
'ValidationData',imdsValidation, ...
'ValidationFrequency',30, ...
'Verbose',false, ...
'Plots','training-progress');
```

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 (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. 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 5-49. 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

Deep Network Designer | `analyzeNetwork` | `trainNetwork` | `trainingOptions`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-19
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Pretrained Deep Neural Networks” on page 1-12
- “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

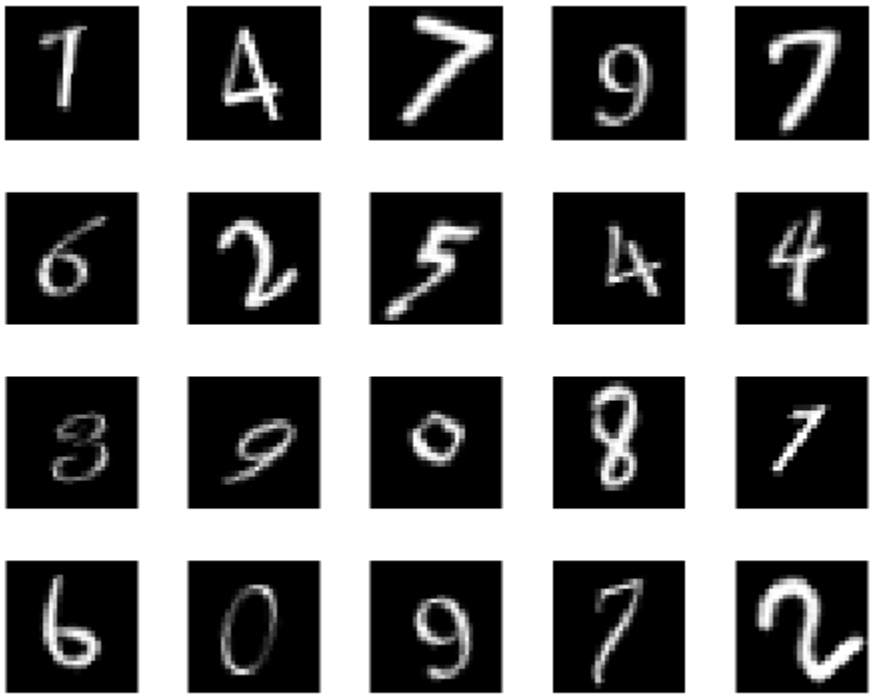
The data set contains synthetic images of handwritten digits together with the corresponding angles (in degrees) by which each image is rotated.

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)))
    drawnow
end
```



Check Data Normalization

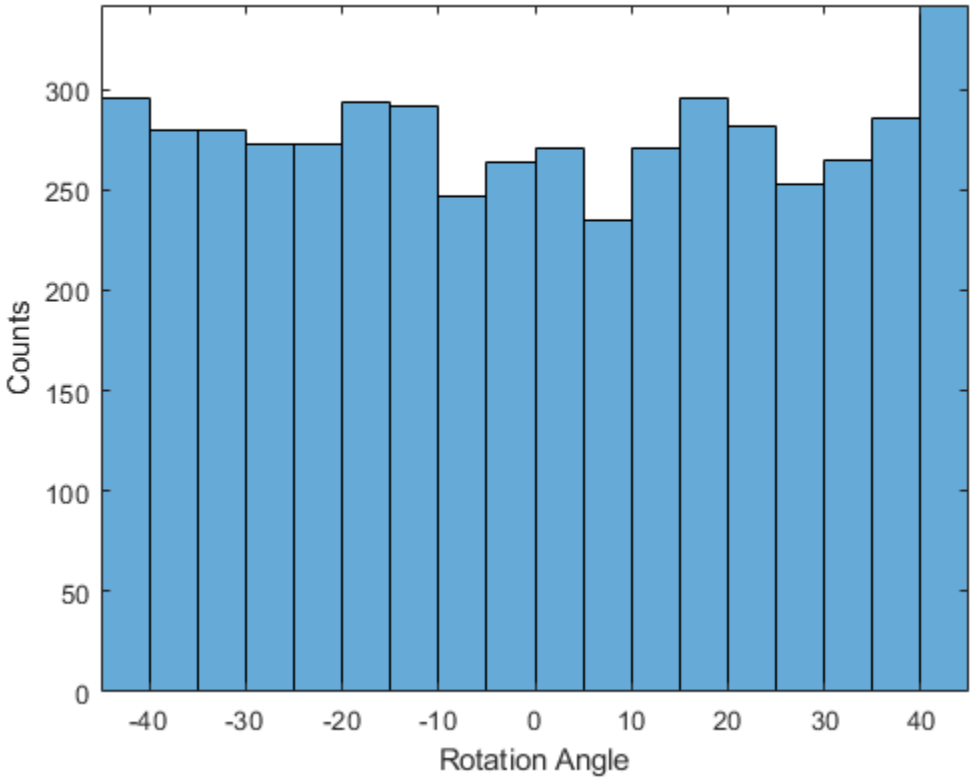
When training neural networks, it often helps to make sure that your data is normalized in all stages of the network. Normalization helps stabilize and speed up network training using gradient descent. If your data is poorly scaled, then the loss can become `NaN` and the network parameters can diverge 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];
```

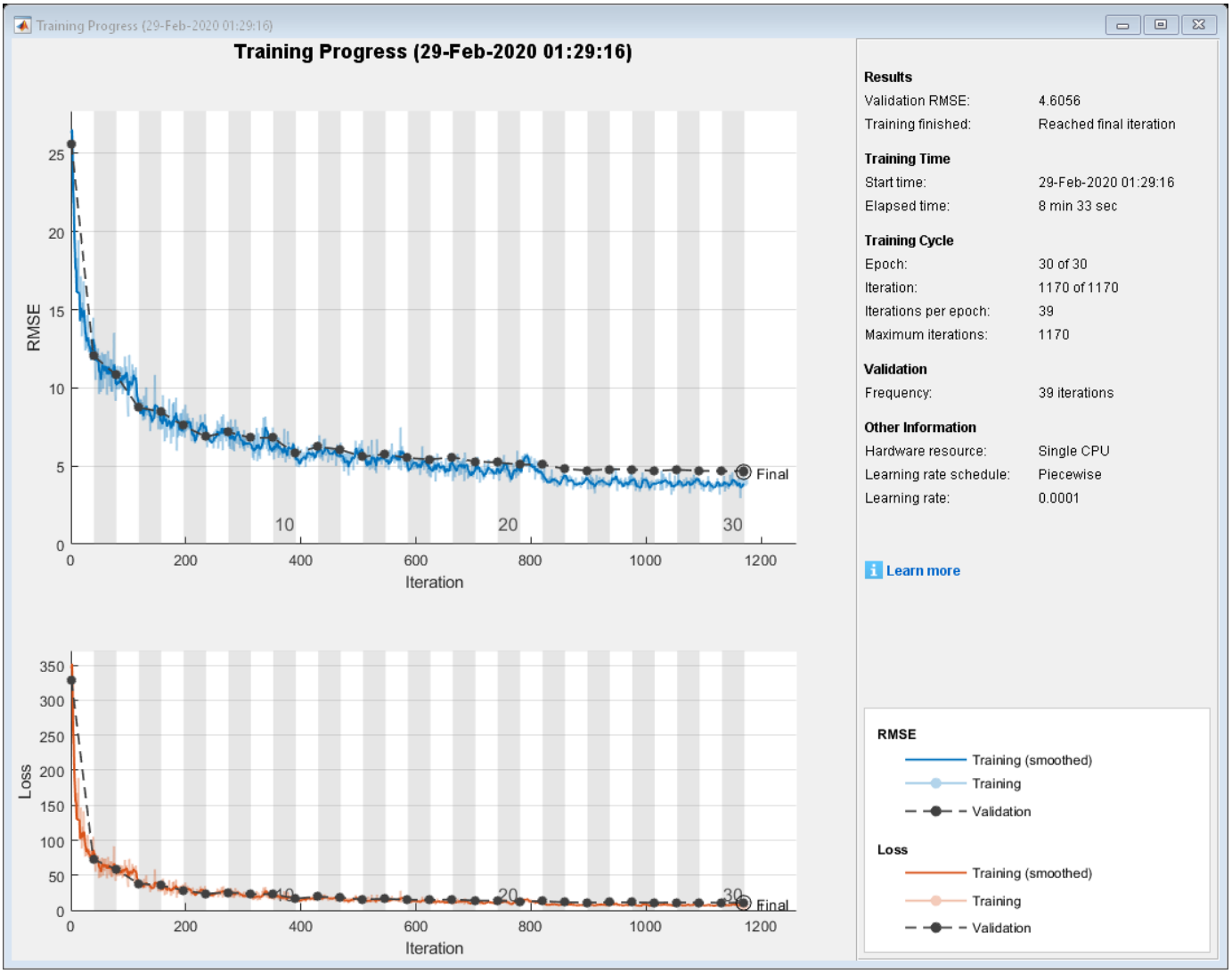
Train Network

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. Otherwise, `trainNetwork` uses the CPU. A CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher is required for training on a GPU.

```
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 = 0.9704
```

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
    4.6056
```

Display Box Plot of Residuals for Each Digit Class

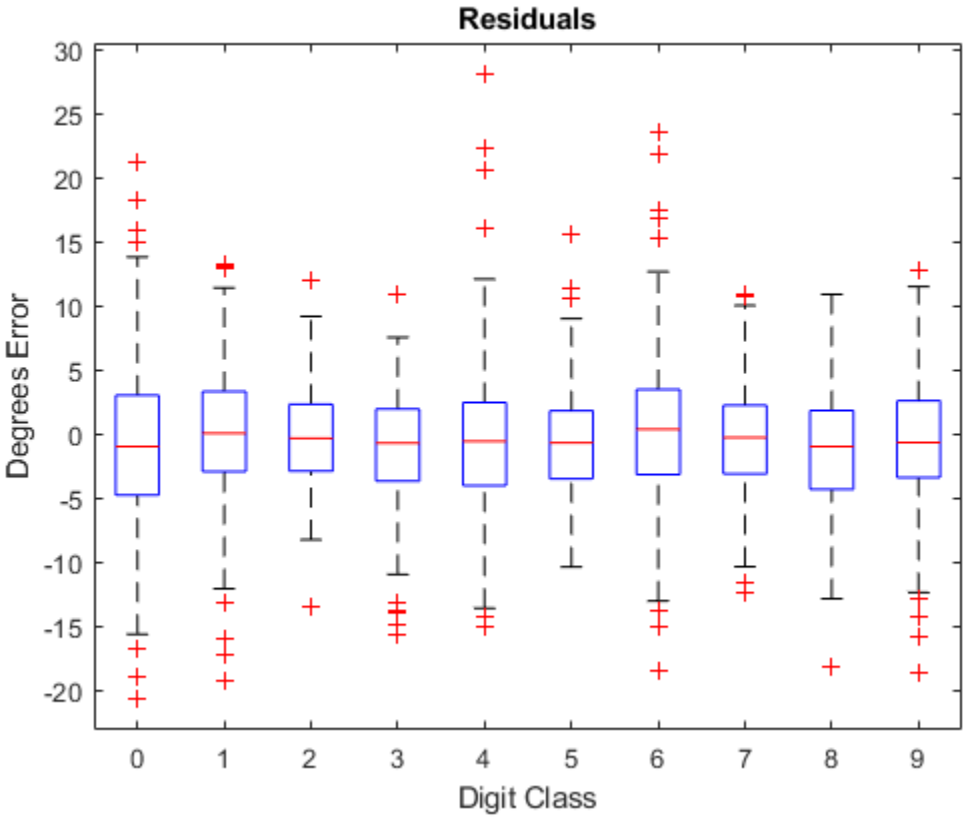
The `boxplot` function requires a matrix where each column corresponds to the residuals for each digit class.

The validation data groups images by digit classes 0–9 with 500 examples of each. Use `reshape` to group the residuals by digit class.

```
residualMatrix = reshape(predictionError,500,10);
```

Each column of `residualMatrix` corresponds to the residuals of each digit. Create a residual box plot for each digit using `boxplot` (Statistics and Machine Learning Toolbox).

```
figure
boxplot(residualMatrix,...
    'Labels',{'0','1','2','3','4','5','6','7','8','9'})
xlabel('Digit Class')
ylabel('Degrees Error')
title('Residuals')
```



The digit classes with highest accuracy have a mean close to zero and little variance.

Correct Digit Rotations

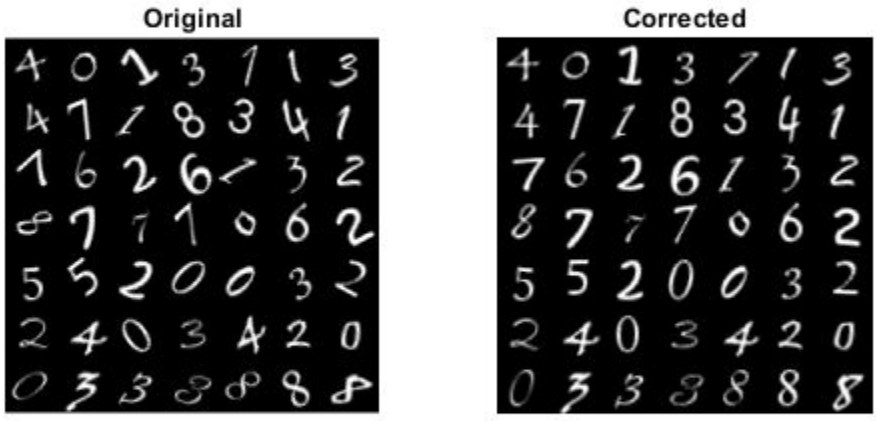
You can use functions from Image Processing Toolbox to straighten the digits and display them together. Rotate 49 sample digits according to their predicted angles of rotation using `imrotate` (Image Processing Toolbox).

```
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

`classificationLayer` | `regressionLayer`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Convert Classification Network into Regression Network” on page 3-66

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 specify the network as a function and train it 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.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;
classNames = categories(YTrain);
numClasses = numel(classNames);
numObservations = numel(YTrain);
```

View some images from the training data.

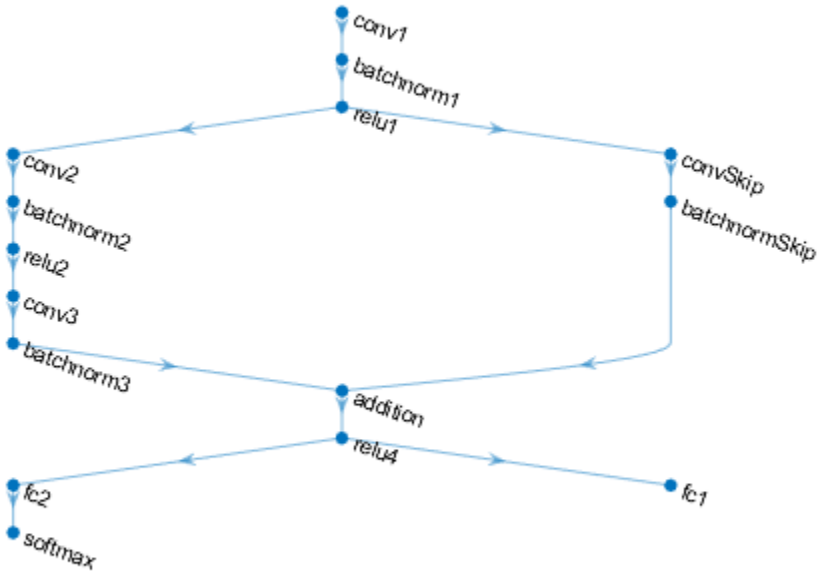
```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

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.
- 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 "conv_1") and `ParameterName` is the name of the parameter (for example, "Weights").

Create a struct `parameters` containing the model parameters. Initialize the learnable layer weights using the example function `initializeGaussian`, listed at the end of the example. Initialize the learnable layer biases with zeros. Initialize the batch normalization offset and scale parameters with zeros and ones, 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. Initialize the batch normalization trained mean and trained variance states with zeros and ones, respectively.

```
parameters.conv1.Weights = dlarray(initializeGaussian([5,5,1,16]));
parameters.conv1.Bias = dlarray(zeros(16,1,'single'));
```

```
parameters.batchnorm1.Offset = dlarray(zeros(16,1,'single'));
parameters.batchnorm1.Scale = dlarray(ones(16,1,'single'));
state.batchnorm1.TrainedMean = zeros(16,1,'single');
state.batchnorm1.TrainedVariance  = ones(16,1,'single');

parameters.convSkip.Weights = dlarray(initializeGaussian([1,1,16,32]));
parameters.convSkip.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnormSkip.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnormSkip.Scale = dlarray(ones(32,1,'single'));
state.batchnormSkip.TrainedMean = zeros(32,1,'single');
state.batchnormSkip.TrainedVariance = ones(32,1,'single');

parameters.conv2.Weights = dlarray(initializeGaussian([3,3,16,32]));
parameters.conv2.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm2.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm2.Scale = dlarray(ones(32,1,'single'));
state.batchnorm2.TrainedMean = zeros(32,1,'single');
state.batchnorm2.TrainedVariance  = ones(32,1,'single');

parameters.conv3.Weights = dlarray(initializeGaussian([3,3,32,32]));
parameters.conv3.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm3.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm3.Scale = dlarray(ones(32,1,'single'));
state.batchnorm3.TrainedMean = zeros(32,1,'single');
state.batchnorm3.TrainedVariance  = ones(32,1,'single');

parameters.fc2.Weights = dlarray(initializeGaussian([10,6272]));
parameters.fc2.Bias = dlarray(zeros(numClasses,1,'single'));

parameters.fc1.Weights = dlarray(initializeGaussian([1,6272]));
parameters.fc1.Bias = dlarray(zeros(1,1,'single'));
```

View the struct of the parameters.

```
parameters
```

```
parameters = struct with fields:
            conv1: [1×1 struct]
       batchnorm1: [1×1 struct]
         convSkip: [1×1 struct]
    batchnormSkip: [1×1 struct]
            conv2: [1×1 struct]
       batchnorm2: [1×1 struct]
            conv3: [1×1 struct]
       batchnorm3: [1×1 struct]
              fc2: [1×1 struct]
              fc1: [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]
    batchnormSkip: [1×1 struct]
       batchnorm2: [1×1 struct]
       batchnorm3: [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 input data `dlX`, the model parameters `parameters`, 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 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 = 30;
miniBatchSize = 128;
plots = "training-progress";
```

Train on a GPU if one is available. This requires Parallel Computing Toolbox™. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

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 epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert to `gpuArray` objects.
- 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
```

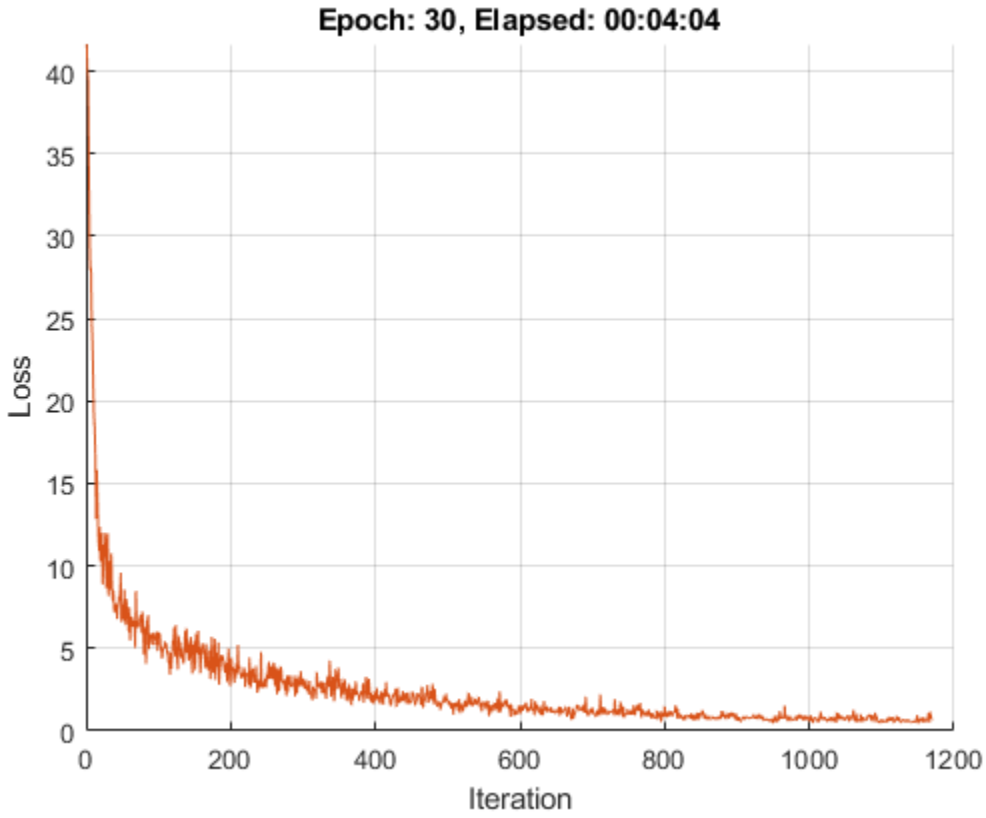
Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model.

```
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    idx = randperm(numObservations);
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);
    anglesTrain = anglesTrain(idx);

    % 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.
        X = XTrain(:,:,:,idx);

        Y1 = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y1(c,YTrain(idx)==classNames(c)) = 1;
        end
```

```
            Y2 = anglesTrain(idx)';
            Y2 = single(Y2);

            % Convert mini-batch of data to dlarray.
            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, state, and loss using dlfeval and the
            % modelGradients function.
            [gradients,state,loss] = dlfeval(@modelGradients, dlX, Y1, Y2, parameters, 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

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

Convert the data to a `dlarray` object with dimension format `'SSCB'`. For GPU prediction, also convert the data to `gpuArray`.

```
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

To predict the labels and angles of the validation data, use the model function with the `doTraining` option set to `false`.

```
doTraining = false;
[dlYPred,anglesPred] = model(dlXTest, parameters,doTraining,state);
```

Evaluate the classification accuracy.

```
[~,idx] = max(extractdata(dlYPred),[],1);
labelsPred = classNames(idx);
accuracy = mean(labelsPred==YTest)
```

```
accuracy = 0.9644
```

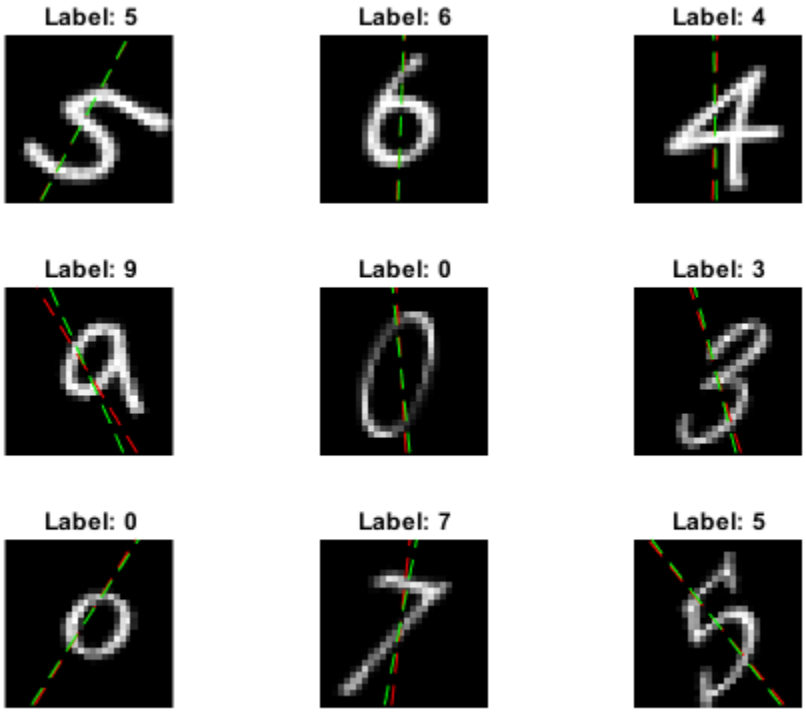
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean((extractdata(anglesPred) - anglesTest').^2))
```

```
angleRMSE =

  gpuArray single

    5.8081
```

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 = extractdata(anglesPred(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(labelsPred(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes the input data `dlX`, the model parameters `parameters`, 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(dlX,parameters,doTraining,state)

% Convolution
W = parameters.conv1.Weights;
B = parameters.conv1.Bias;
dlY = dlconv(dlX,W,B,'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,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)
W = parameters.convSkip.Weights;
B = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,W,B,'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
W = parameters.conv2.Weights;
B = parameters.conv2.Bias;
dlY = dlconv(dlY,W,B,'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,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,Offset,Scale,trainedMean,trainedVariance);
end
dlY = relu(dlY);

% Convolution
W = parameters.conv3.Weights;
B = parameters.conv3.Bias;
dlY = dlconv(dlY,W,B,'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,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 (angles)
W = parameters.fc1.Weights;
B = parameters.fc1.Bias;
dlY2 = fullyconnect(dlY,W,B);

% Fully connect, softmax (labels)
W = parameters.fc2.Weights;
B = parameters.fc2.Bias;
dlY1 = fullyconnect(dlY,W,B);
dlY1 = softmax(dlY1);

end
```

Model Gradients Function

The `modelGradients` function, takes 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(dlX,T1,T2,parameters,state)

doTraining = true;
[dlY1,dlY2,state] = model(dlX,parameters,doTraining,state);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,parameters);

end
```

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
```

See Also

`batchnorm` | `dlarray` | `dlconv` | `dlfeval` | `dlgradient` | `fullyconnect` | `relu` | `sgdmupdate` | `softmax`

More About

- “Multiple-Input and Multiple-Output Networks” on page 1-21
- “Make Predictions Using Model Function” on page 15-173
- “Assemble Multiple-Output Network for Prediction” on page 15-106
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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

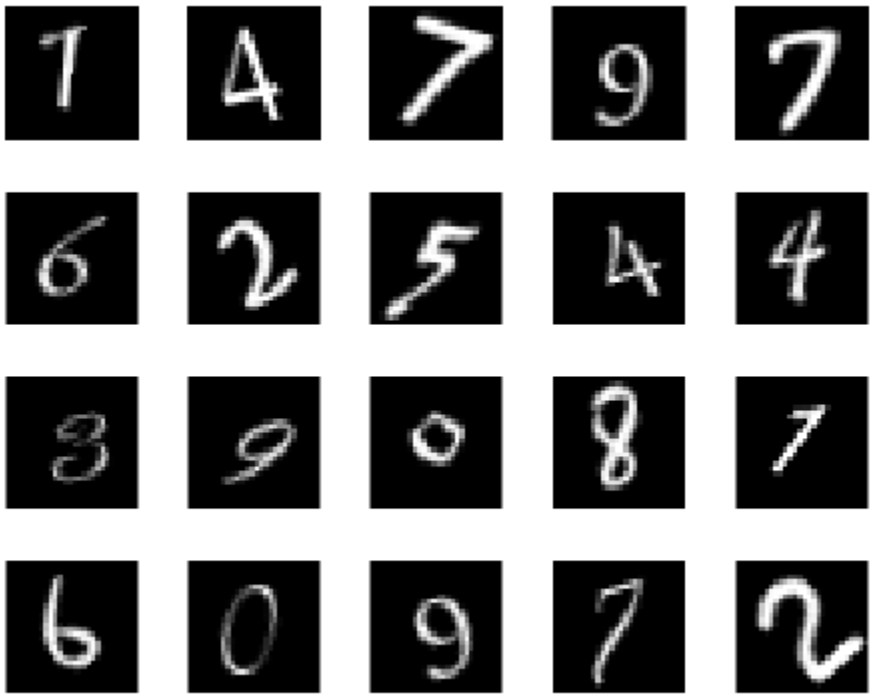
The data set contains synthetic images of handwritten digits together with the corresponding angles (in degrees) by which each image is rotated.

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)))
    drawnow
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));
```

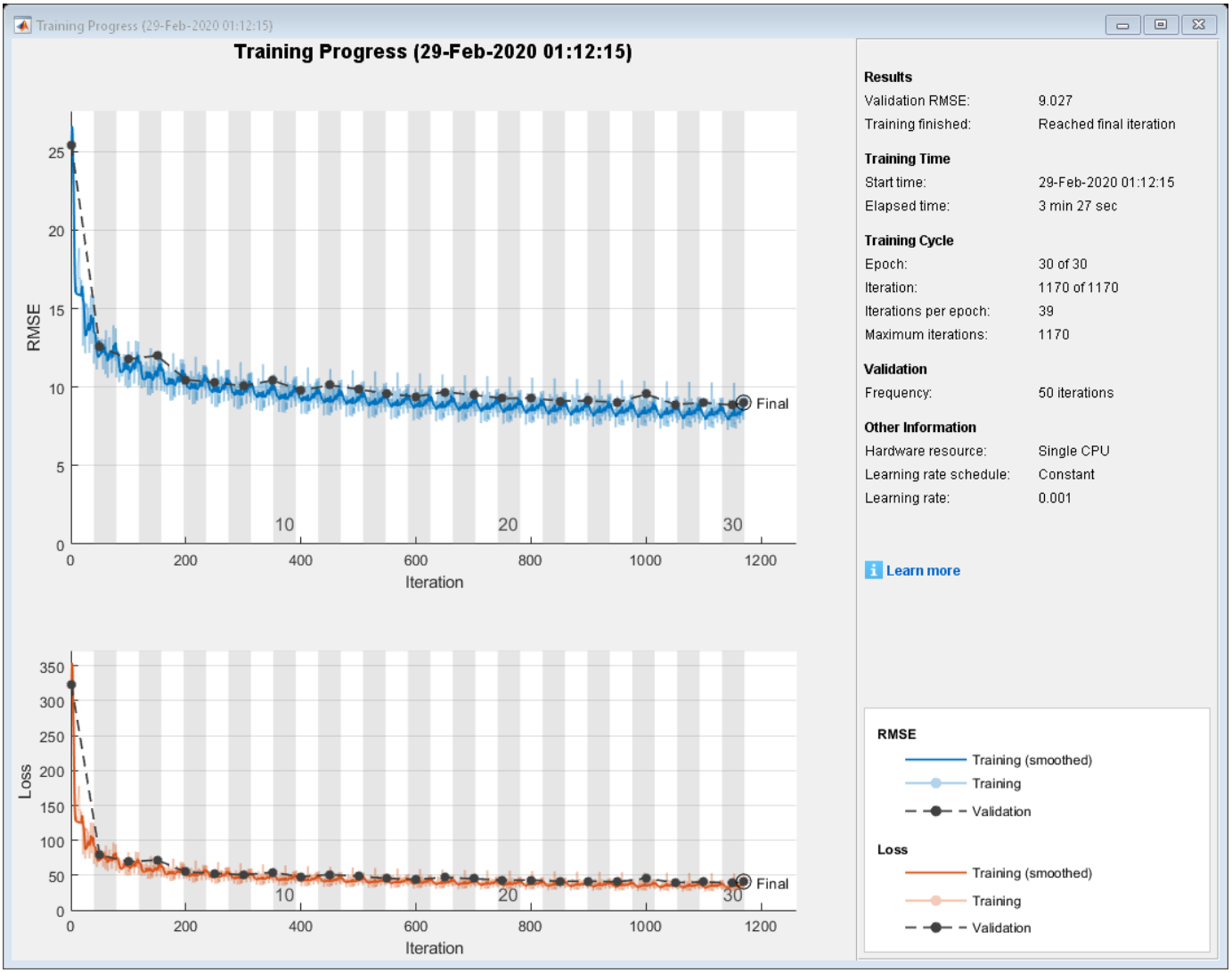
Train Network

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. Otherwise, `trainNetwork` uses the CPU. A CUDA®-enabled NVIDIA® GPU with compute capability 3.0 or higher is required for training on a GPU.

```
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
    9.0270
```

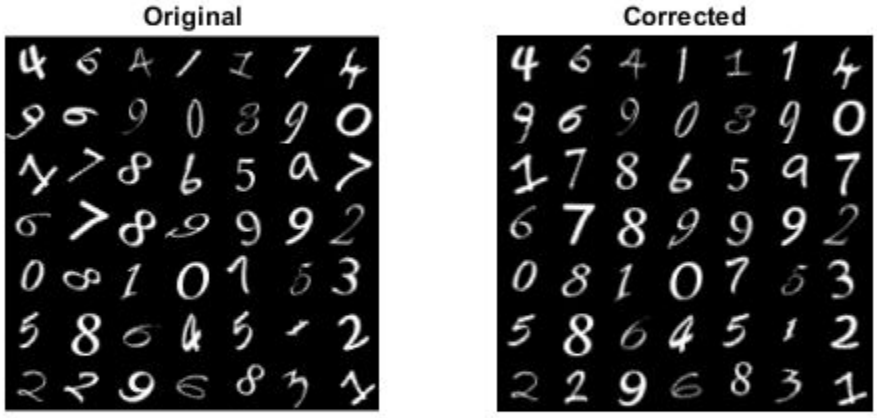
Correct Digit Rotations

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

`classificationLayer` | `regressionLayer`

Related Examples

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

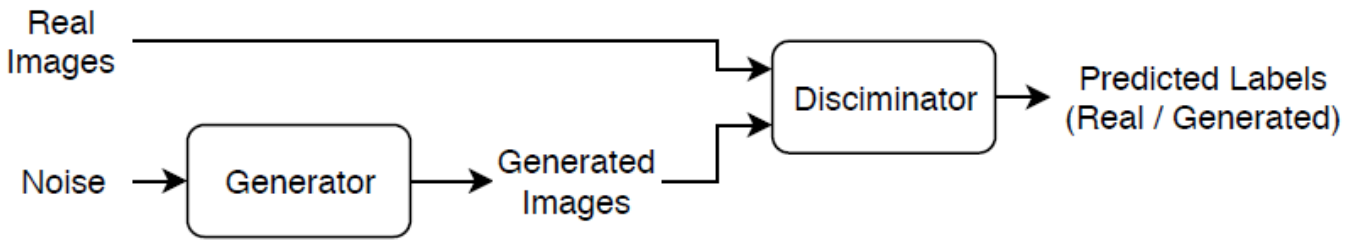
Train Generative Adversarial Network (GAN)

This example shows how to train a generative adversarial network (GAN) 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.

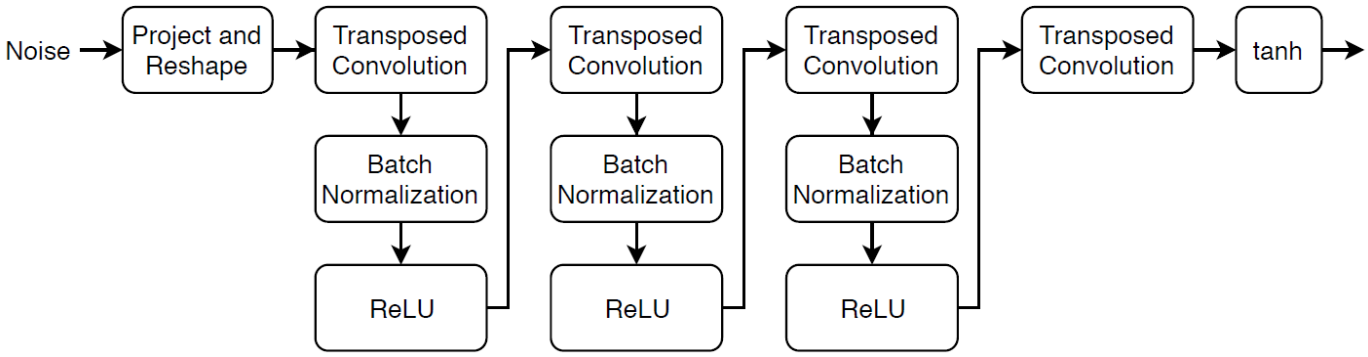
```
datasetFolder = fullfile(imageFolder);

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 1-by-1-by-100 arrays of random values:



This network:

- Converts the 1-by-1-by-100 arrays of noise 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 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 the custom layer `projectAndReshapeLayer`, attached to this example as a supporting file. The `projectAndReshapeLayer` 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];

layersGenerator = [
```

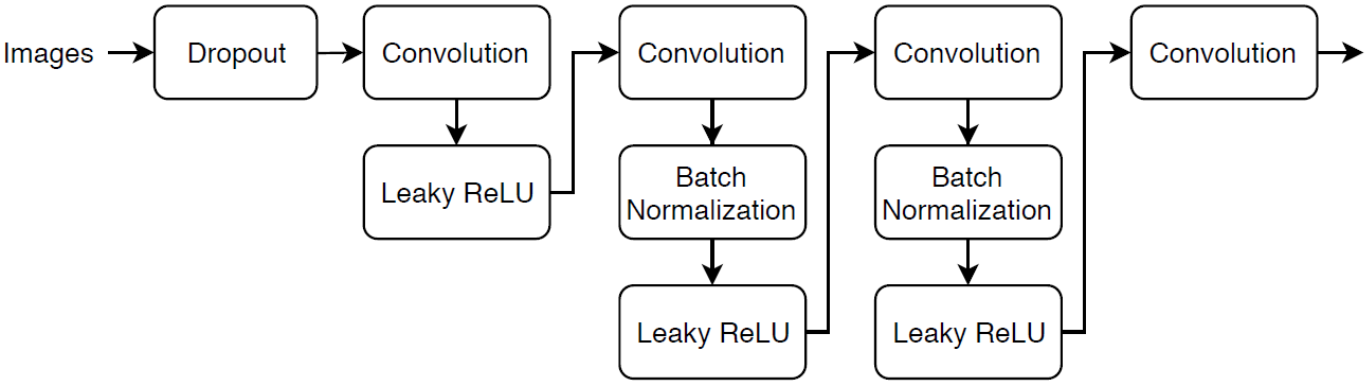
```
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','in')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj');
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bnorm1')
    reluLayer('Name','relu1')
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','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);
```

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 a 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 Model Gradients Function on page 3-0 .

```
dropoutProb = 0.5;
numFilters = 64;
scale = 0.2;
```

```
inputSize = [64 64 3];
filterSize = 5;

layersDiscriminator = [
    imageInputLayer(inputSize,'Normalization','none','Name','in')
    dropoutLayer(0.5,'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);
```

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, Loss Functions and Scores

Create the function `modelGradients`, listed in the Model Gradients Function on page 3-0 section of the example, which takes as input 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. Also set the read size of the augmented image datastore to the mini-batch size. For larger datasets, you might not need to train for as many epochs.

```
numEpochs = 500;
miniBatchSize = 128;
augimds.MiniBatchSize = miniBatchSize;
```

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;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

IIf the discriminator learns to discriminate between real and generated images too quickly, then the generator may 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 to flip 30% of the real labels. This means that 15% of the total number of labels will be flipped. Note that this does not impair the generator as all the generated images are still labelled correctly.

```
flipFactor = 0.3;
```

Display the generated validation images every 100 iterations.

```
validationFrequency = 100;
```

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 as well as a plot of the scores.

Initialize the parameters for Adam.

```
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
```

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(1,1,numLatentInputs,numValidationImages,'single');
```

Convert the data to `dlarray` objects and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).

```
dlZValidation = dlarray(ZValidation,'SSCB');
```

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 newtork 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:

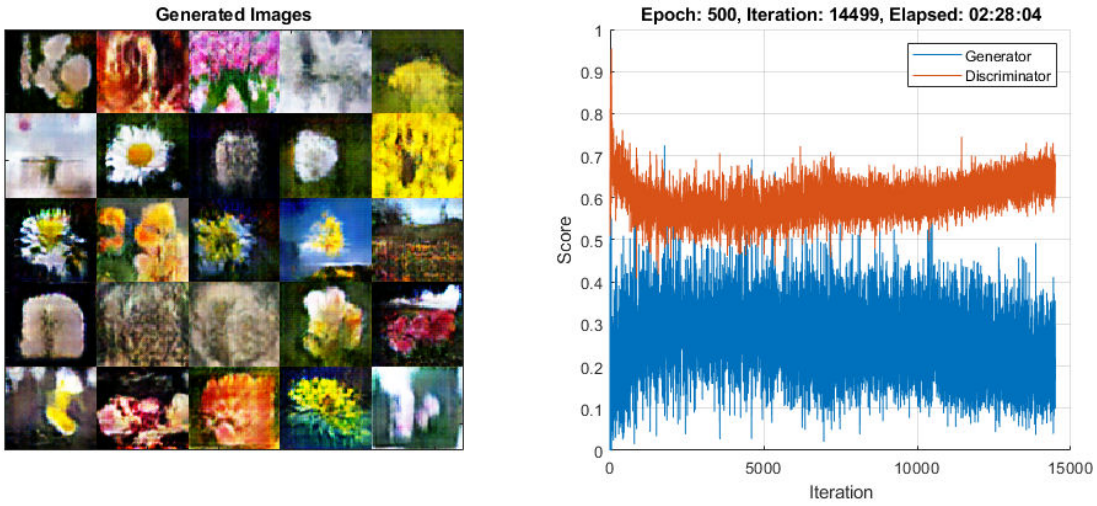
- Rescale the images in the range [-1 1].
- Convert the data to `dlarray` objects with underlying type `single` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- Generate a `dlarray` object containing an array of random values for the generator network.
- For GPU training, convert the data to `gpuArray` objects.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function.
- Update the network parameters using the `adamupdate` function.
- 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.
    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

        % Concatenate mini-batch of data and generate latent inputs for the
        % generator network.
        X = cat(4,data{:,1}{:});
        X = single(X);
        Z = randn(1,1,numLatentInputs,size(X,4),'single');

        % Rescale the images in the range [-1 1].
        X = rescale(X,-1,1,'InputMin',0,'InputMax',255);
```

```
% Convert mini-batch of data to dlarray and specify the dimension labels
% 'SSCB' (spatial, spatial, channel, batch).
dlX = dlarray(X, 'SSCB');
dlZ = dlarray(Z, 'SSCB');

% If training on a GPU, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
    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 realistic looking 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 5-124.

Generate New Images

To generate new images, use the `predict` function on the generator with a `dlarray` object containing a batch of 1-by-1-by-100 arrays of random values. 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 1-by-1-by-100 arrays of random values to input into the generator network.

```
ZNew = randn(1,1,numLatentInputs,25,'single');
dlZNew = dlarray(ZNew,'SSCB');
```

To generate images using the GPU, also convert the data to `gpuArray` objects.

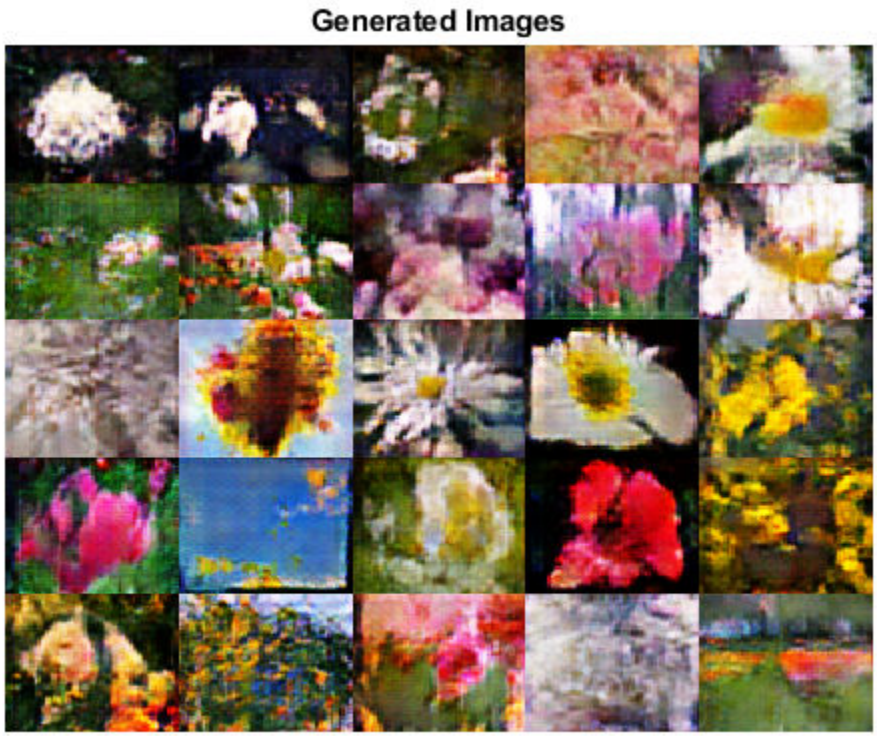
```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZNew = gpuArray(dlZNew);
end
```

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], `modelGradients` 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
```

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." *arXiv preprint arXiv:1511.06434* (2015).

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `forward` | `predict`

More About

- “Train Conditional Generative Adversarial Network (CGAN)” on page 3-83
- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-124
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Train Network Using Custom Training Loop” on page 15-134
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

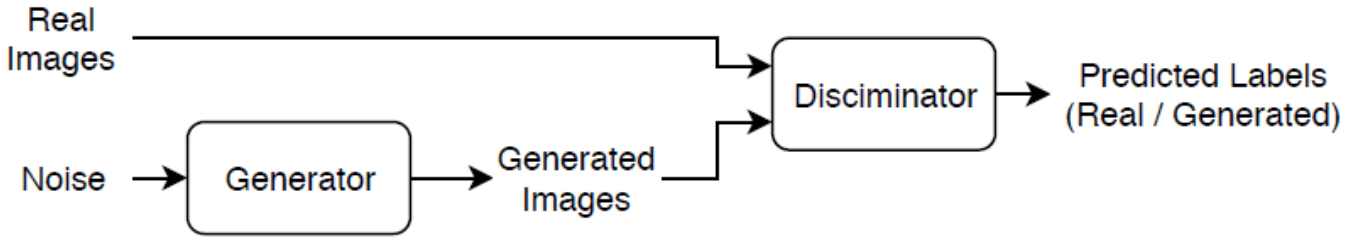
Train Conditional Generative Adversarial Network (CGAN)

This example shows how to train a conditional generative adversarial network (CGAN) 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.

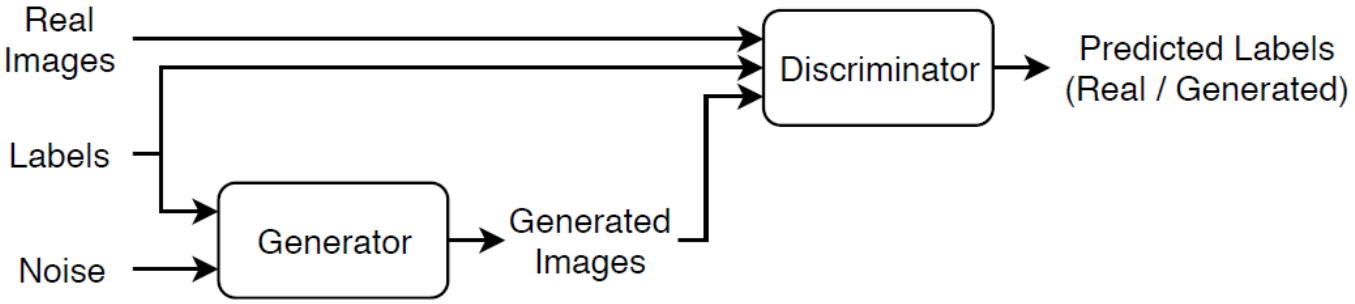
A GAN consists of two networks that train together:

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 is a type of GAN that also takes advantage of labels during the training process.

1. The 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. The 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.

```
datasetFolder = fullfile(imageFolder);

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

View the number of classes.

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

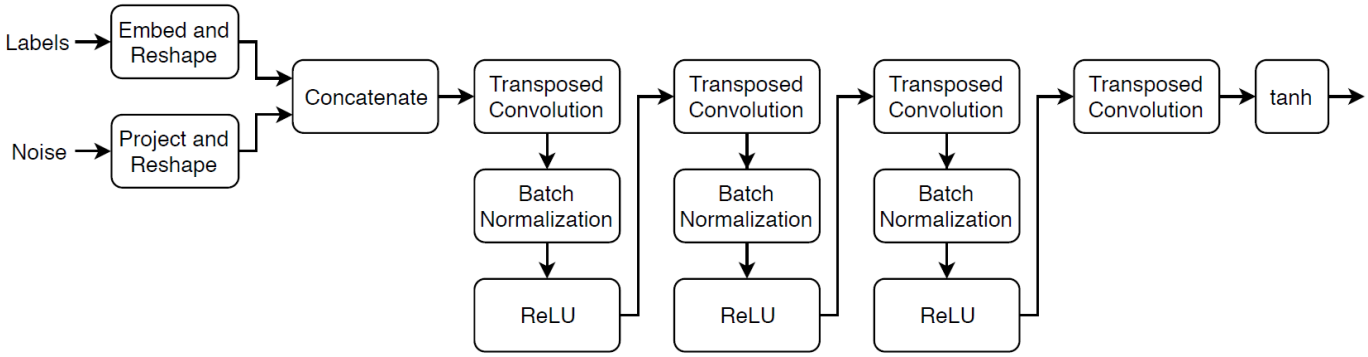
```
numClasses = 5
```

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 1-by-1-by-100 arrays of random values and corresponding labels:



This network:

- Converts the 1-by-1-by-100 arrays of noise to 4-by-4-by-1024 arrays.
- 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-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 channel of the generated images.
- 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 `projectAndReshapeLayer` object upscales the input using a fully connected operation and reshapes the output to the specified size.

To input the labels into the network, use an `imageInputLayer` object and specify an image 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 image of the specified size using an embedding and a fully connected operation.

```
numLatentInputs = 100;
embeddingDimension = 50;
numFilters = 64;

filterSize = 5;
projectionSize = [4 4 1024];

layersGenerator = [
    imageInputLayer([1 1 numLatentInputs],'Normalization','none','Name','noise')
    projectAndReshapeLayer(projectionSize,numLatentInputs,'proj');
    concatenationLayer(3,2,'Name','cat');
    transposedConv2dLayer(filterSize,4*numFilters,'Name','tconv1')
    batchNormalizationLayer('Name','bn1')
```

```
    reluLayer('Name','relu1')
    transposedConv2dLayer(filterSize,2*numFilters,'Stride',2,'Cropping','same','Name','tconv2')
    batchNormalizationLayer('Name','bn2')
    reluLayer('Name','relu2')
    transposedConv2dLayer(filterSize,numFilters,'Stride',2,'Cropping','same','Name','tconv3')
    batchNormalizationLayer('Name','bn3')
    reluLayer('Name','relu3')
    transposedConv2dLayer(filterSize,3,'Stride',2,'Cropping','same','Name','tconv4')
    tanhLayer('Name','tanh')];

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');
```

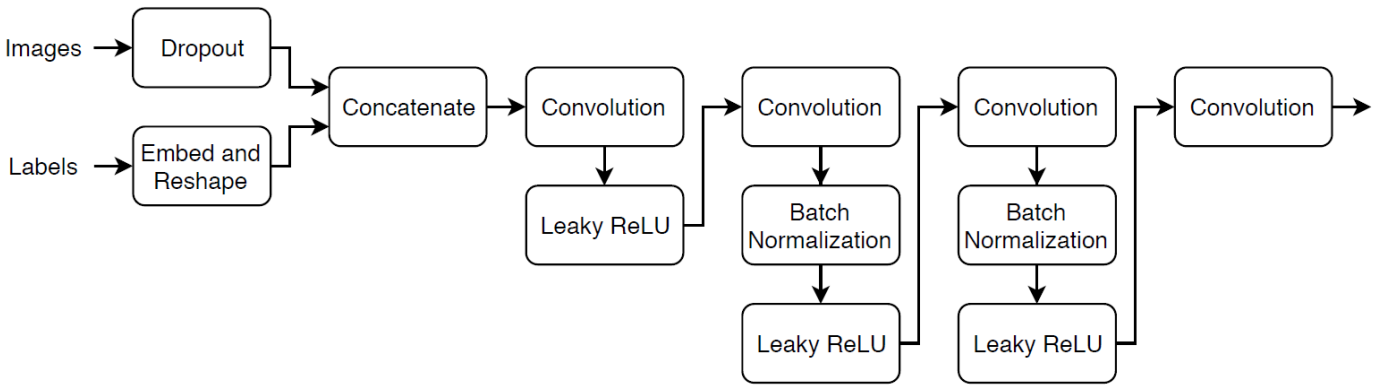
To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator)
```

```
dlnetGenerator =
  dlnetwork with properties:

         Layers: [16×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
     Learnables: [19×3 table]
          State: [6×3 table]
     InputNames: {'noise'  'labels'}
    OutputNames: {'tanh'}
```

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 0.25.
- 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 to 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 = 0.25;
numFilters = 64;
scale = 0.2;

inputSize = [64 64 3];
filterSize = 5;

layersDiscriminator = [
    imageInputLayer(inputSize,'Normalization','none','Name','images')
    dropoutLayer(dropoutProb,'Name','dropout')
    concatenationLayer(3,2,'Name','cat')
    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);

layers = [
    imageInputLayer([1 1],'Name','labels','Normalization','none')
    embedAndReshapeLayer(inputSize,embeddingDimension,numClasses,'emb')];

lgraphDiscriminator = addLayers(lgraphDiscriminator,layers);
lgraphDiscriminator = connectLayers(lgraphDiscriminator,'emb','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: [17×1 nnet.cnn.layer.Layer]
    Connections: [16×2 table]
     Learnables: [19×3 table]
          State: [6×3 table]
     InputNames: {'images'  'labels'}
    OutputNames: {'conv5'}
```

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;
augimds.MiniBatchSize = miniBatchSize;
```

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;
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

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 may 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.

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 1-by-1-by-`numLatentInputs` arrays of random values and a corresponding set of labels 1 through 5 (corresponding to the classes) repeated 5 times, where the labels are in the fourth dimension of the array.

```
numValidationImagesPerClass = 5;
ZValidation = randn(1,1,numLatentInputs,numValidationImagesPerClass*numClasses,'single');
TValidation = single(repmat(1:numClasses,[1 numValidationImagesPerClass]));
TValidation = permute(TValidation,[1 3 4 2]);
```

Convert the data to `dlarray` objects and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).

```
dlZValidation = dlarray(ZValidation, 'SSCB');
dlTValidation = dlarray(TValidation, 'SSCB');
```

For GPU training, convert the data to `gpuArray` objects.

```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZValidation = gpuArray(dlZValidation);
    dlTValidation = gpuArray(dlTValidation);
end
```

Train the GAN. For each epoch, shuffle the data and loop over mini-batches of data.

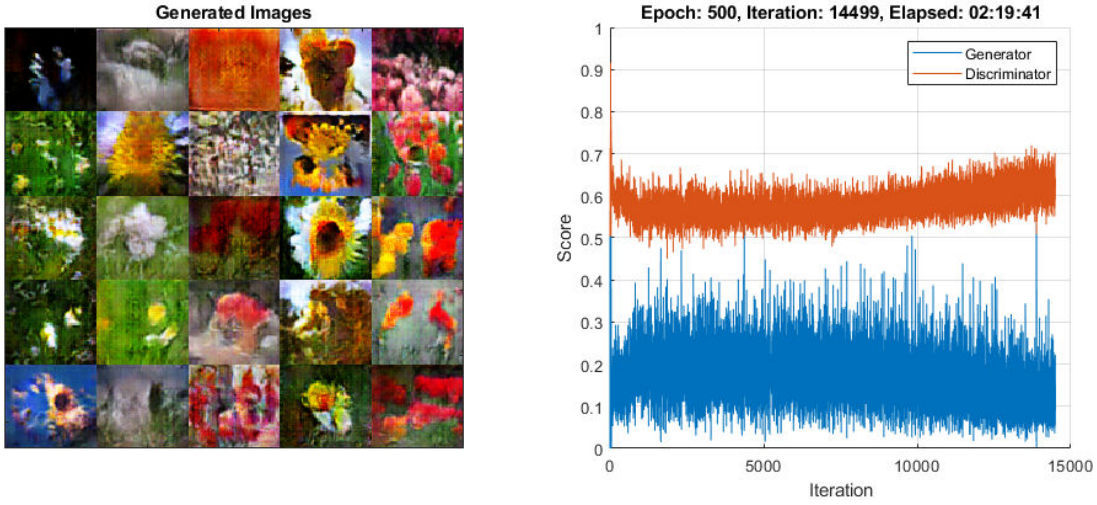
For each mini-batch:

- To ensure that the inputs to the discriminator match the outputs of the generator, rescale the real images so that the pixels take values in the range [-1, 1].
- Convert the image data and labels to `dlarray` objects with underlying type `single` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- Generate a `dlarray` object containing an array of random values for the generator network.
- 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.
- 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.
    reset(augimds);
    augimds = shuffle(augimds);

    % Loop over mini-batches.
    while hasdata(augimds)
        iteration = iteration + 1;

        % Read mini-batch of data and generate latent inputs for the
        % generator network.
        data = read(augimds);

        % Ignore last partial mini-batch of epoch.
        if size(data,1) < miniBatchSize
            continue
        end

        X = cat(4,data{:,1}{:});
        X = single(X);

        T = single(data.response);
        T = permute(T,[2 3 4 1]);

        Z = randn(1,1,numLatentInputs,miniBatchSize,'single');

        % Rescale the images in the range [-1 1].
        X = rescale(X,-1,1,'InputMin',0,'InputMax',255);

        % Convert mini-batch of data to dlarray and specify the dimension labels
        % 'SSCB' (spatial, spatial, channel, batch).
        dlX = dlarray(X, 'SSCB');
        dlZ = dlarray(Z, 'SSCB');
        dlT = dlarray(T, 'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
            dlZ = gpuArray(dlZ);
            dlT = gpuArray(dlT);
        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 realistic looking 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 5-124.

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 1-by-1-by-`numLatentInputs` arrays of random values and an array of labels corresponding to the desired classes. Convert the data to `dlarray` objects and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch). For GPU prediction, convert the data to `gpuArray`. 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(1,1,numLatentInputs,numObservationsNew,'single');
T = repmat(single(idxClass),[1 1 1 numObservationsNew]);
```

Convert the data to `dlarray` objects with the dimension labels `'SSCB'` (spatial, spatial, channels, batch).

```
dlZ = dlarray(Z,'SSCB');
dlT = dlarray(T,'SSCB');
```

To generate images using the GPU, also convert the data to `gpuArray` objects.

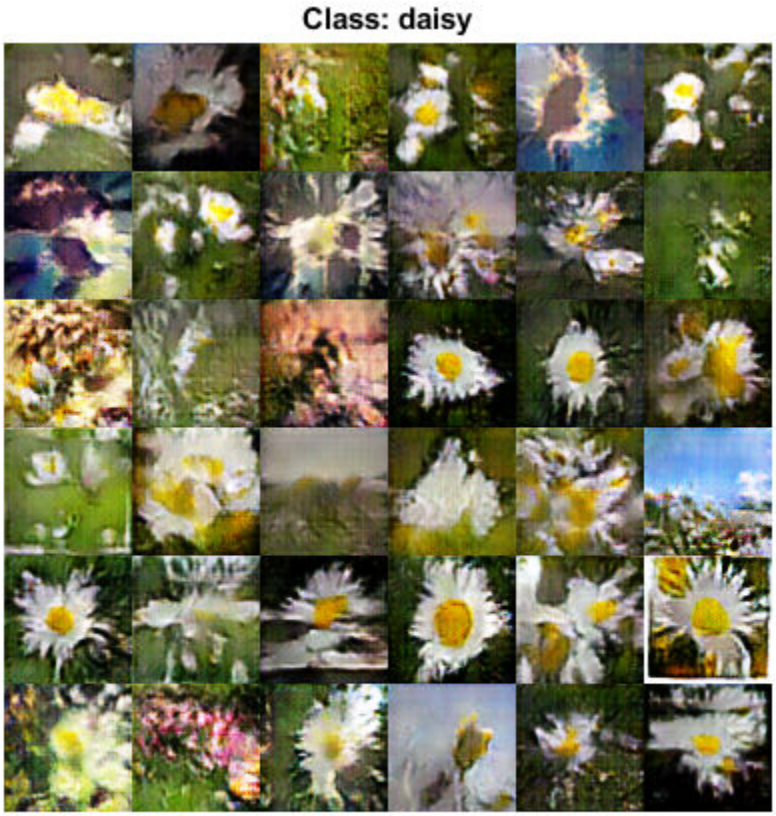
```
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlZ = gpuArray(dlZ);
    dlT = gpuArray(dlT);
end
```

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 `dlnetDiscrimintor`, 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 may 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, scoreDiscri
    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 that the sigmoid operation σ 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
```

References

1 The TensorFlow Team. *Flowers* http://download.tensorflow.org/example_images/flower_photos.tgz

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `forward` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Monitor GAN Training Progress and Identify Common Failure Modes” on page 5-124
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Train Network Using Custom Training Loop” on page 15-134
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

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 3-110.

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.

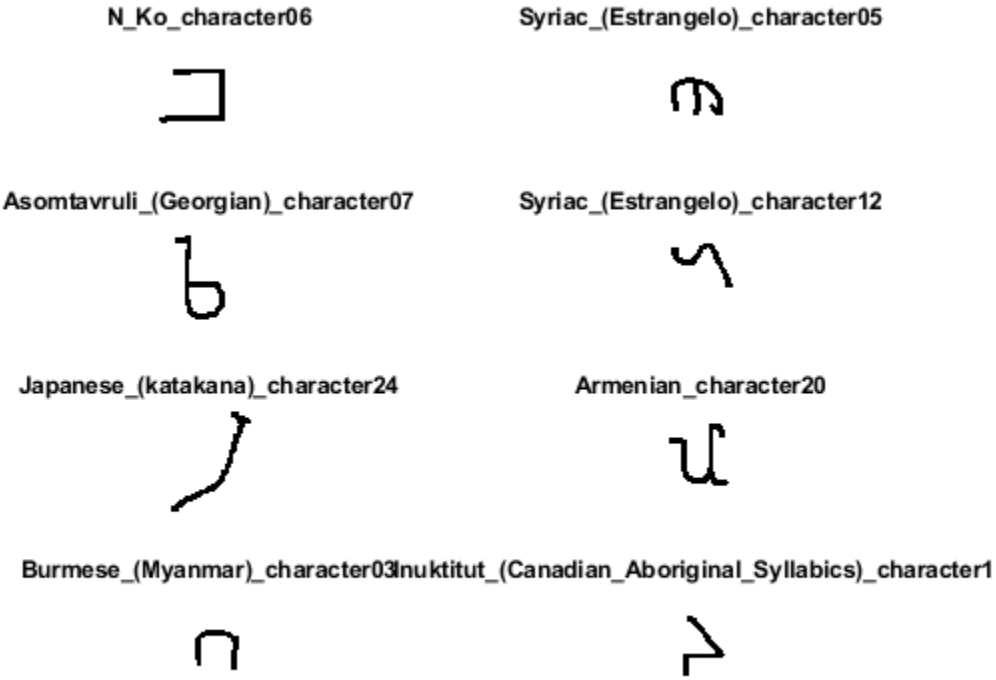
```
imdsTrain = imageDatastore(dataFolderTrain, ...
    'IncludeSubfolders',true, ...
    'LabelSource','none');

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

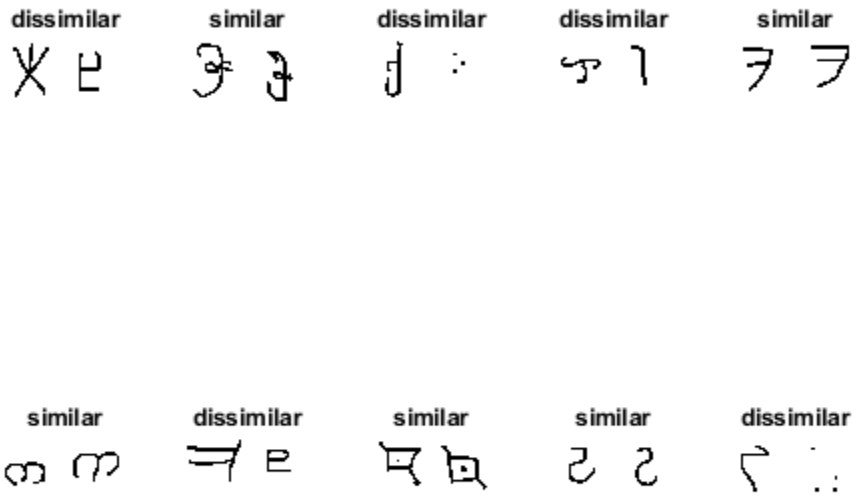
To train the network, the data must be grouped into pairs of images that are either similar or dissimilar. Here, similar images are different handwritten instances of the same character, which have the same label, while dissimilar images of different characters have different labels. The function `getSiameseBatch` (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.

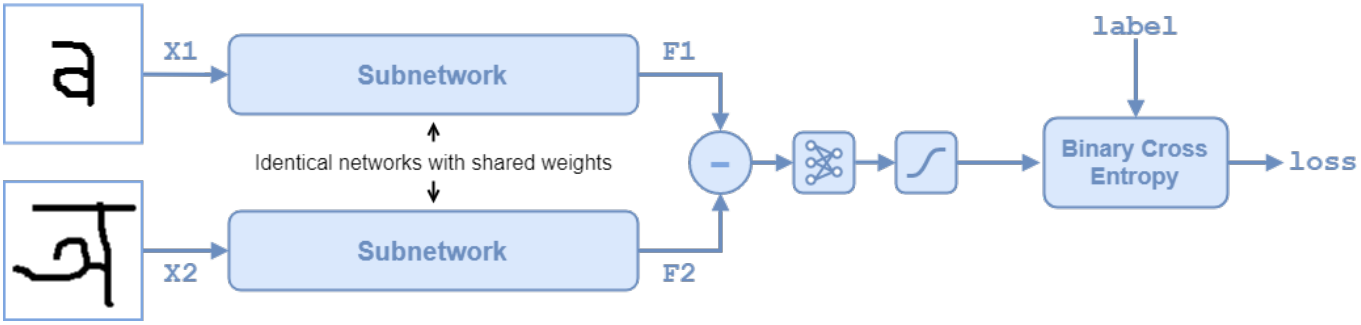
```
for i = 1:batchSize
    if pairLabel(i) == 1
        s = "similar";
    else
        s = "dissimilar";
    end
    subplot(2,5,i)
    imshow([pairImage1(:,:,:,i) pairImage2(:,:,:,i)]);
    title(s)
end
```



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 CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. 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

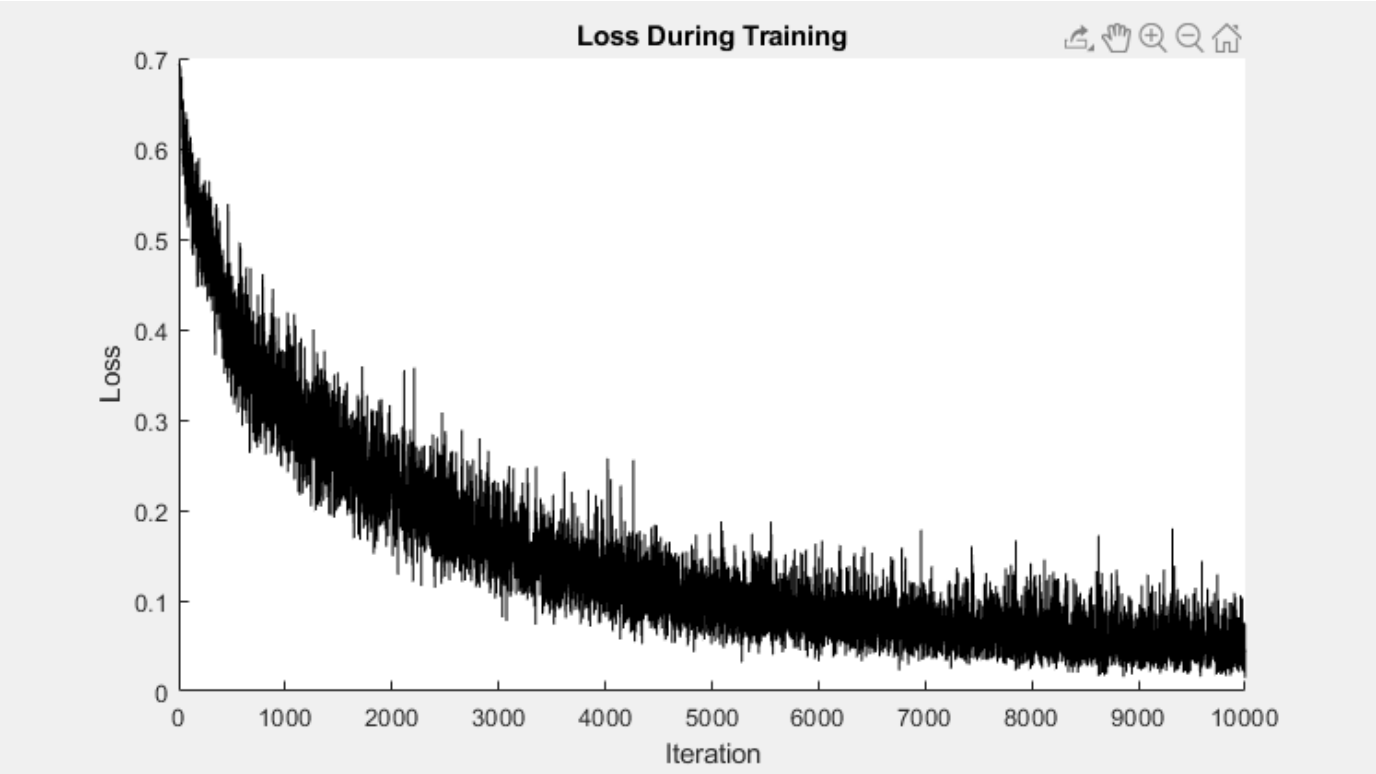
Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

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 accuracye for the minibatch
    accuracy(i) = sum(Y == pairLabelsAcc)/accuracyBatchSize;
end

% Compute accuracy over all minibatches
averageAccuracy = mean(accuracy)*100
```

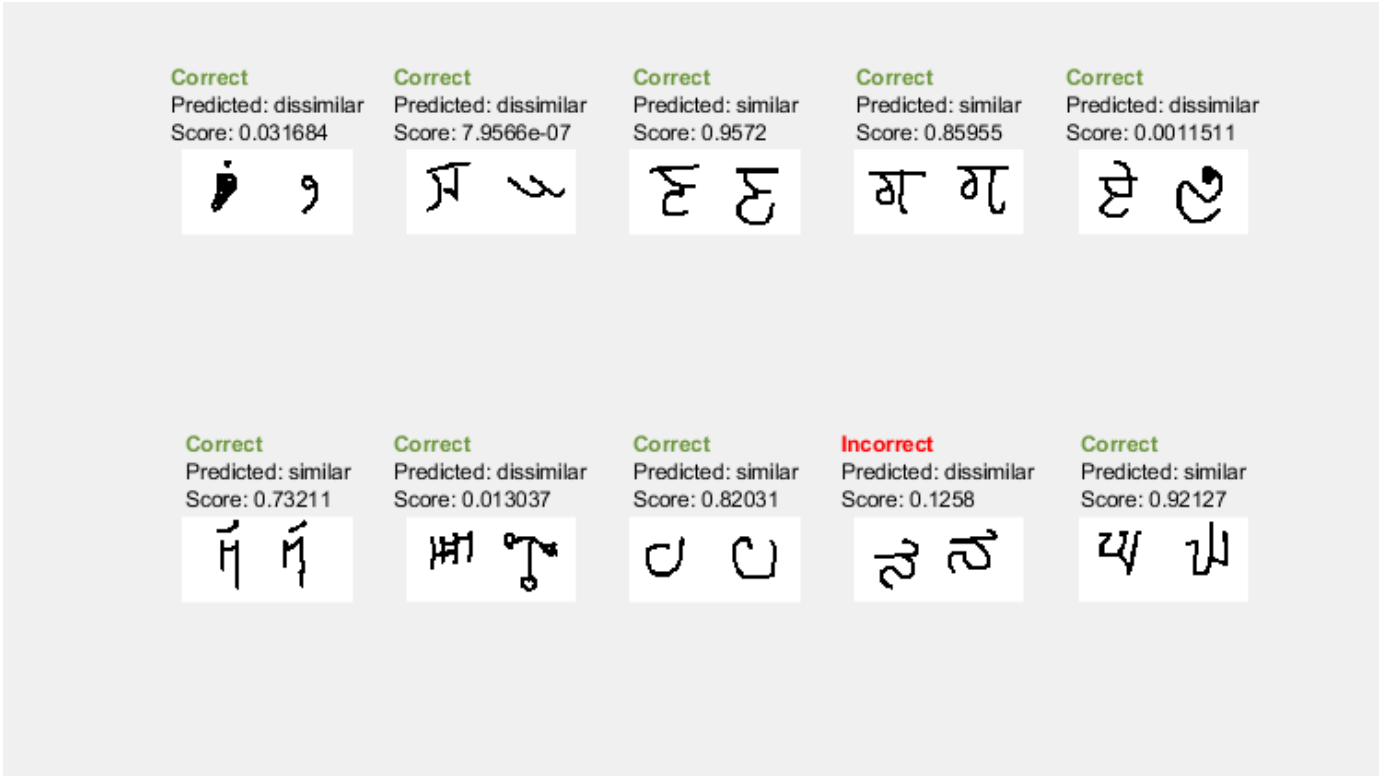
```
averageAccuracy = 88.6667
```

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 the 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,pairLab
% 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, pairLabels, and returns the binary cross-entropy loss value.

    % Get the precision of the prediction to prevent errors due to floating
    % point precision
    y = extractdata(Y);
    if(isa(y,'gpuArray'))
        precision = classUnderlying(y);
    else
        precision = class(y);
    end

    % 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. LeCunn, 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

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Train a Siamese Network for Dimensionality Reduction” on page 3-110
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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 3-96.

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);
```

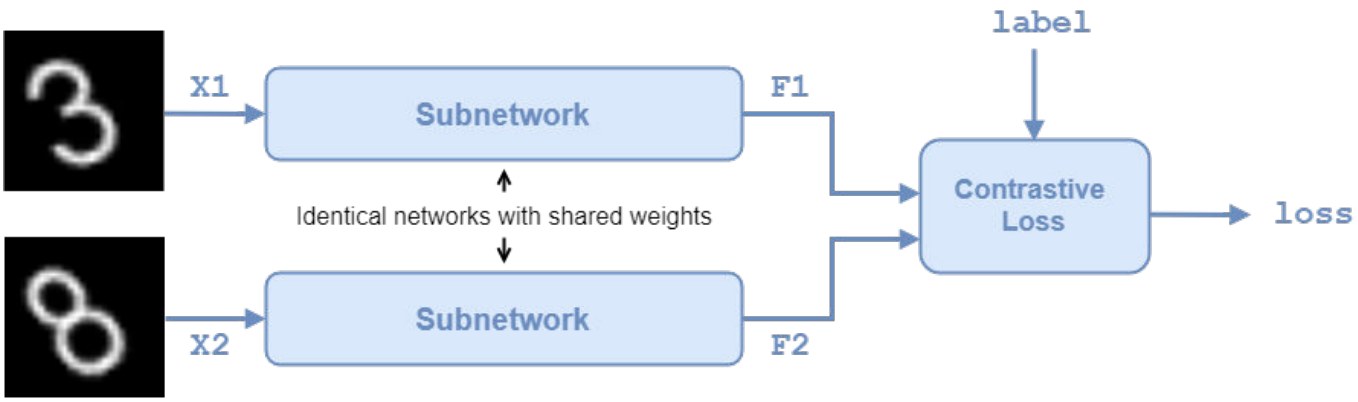
Display the generated pairs of images.

```
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 CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. 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

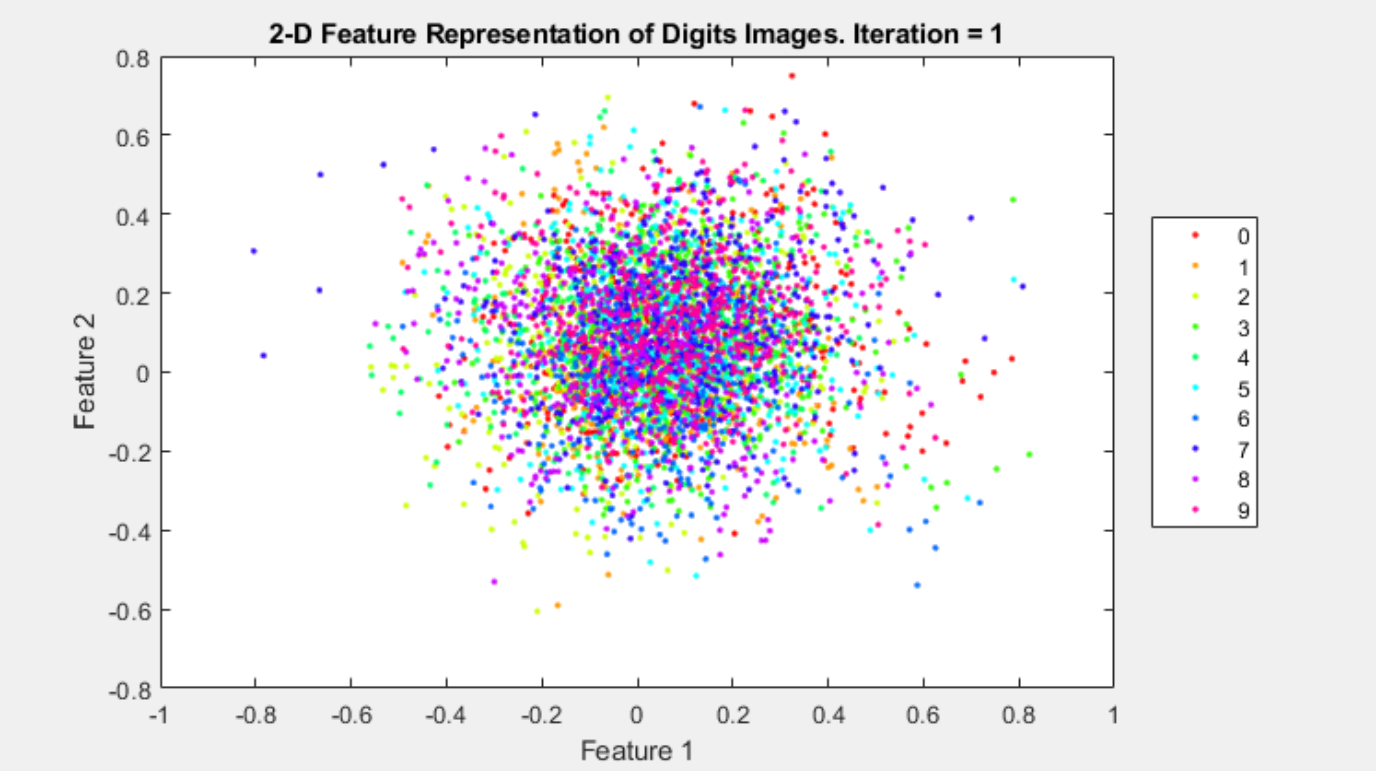
Train the model using a custom training loop. Loop over the training data and update the network parameters at each iteration.

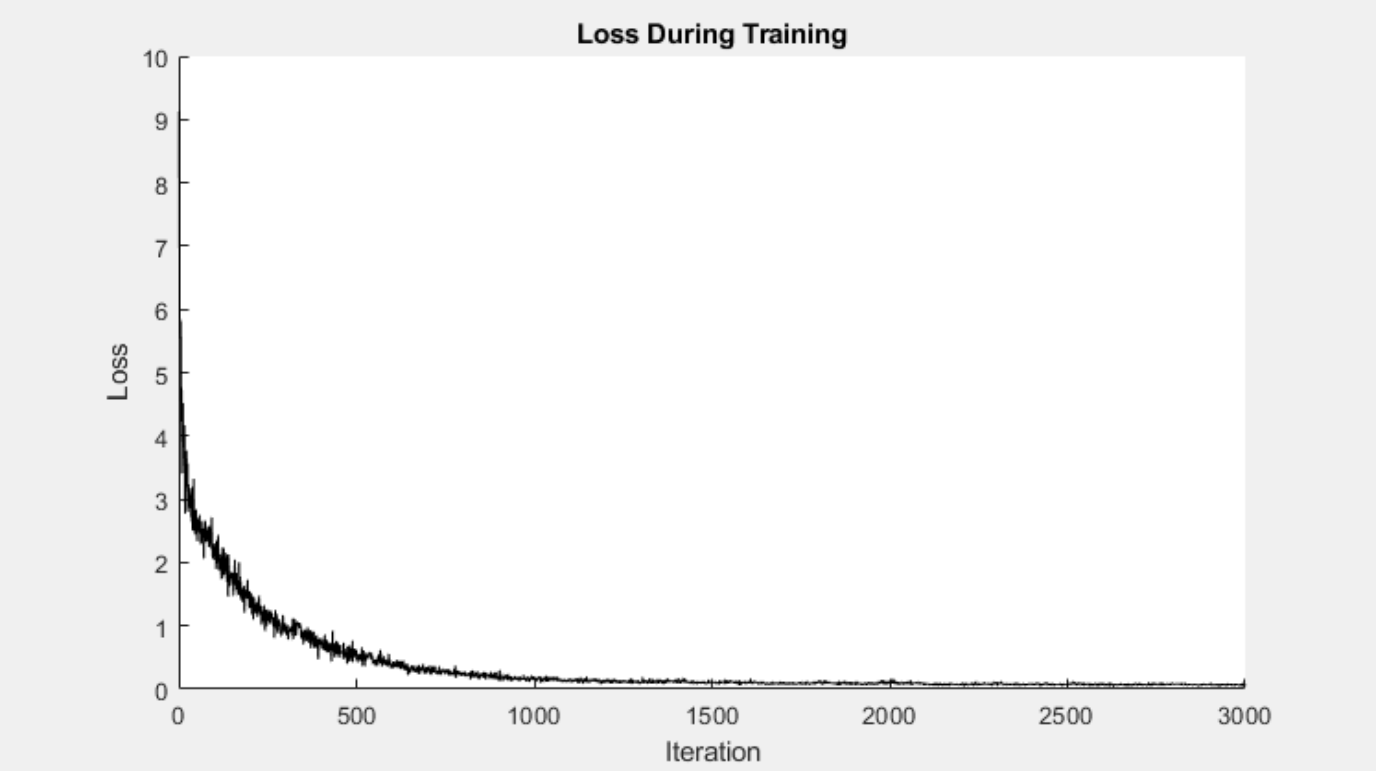
For each iteration:

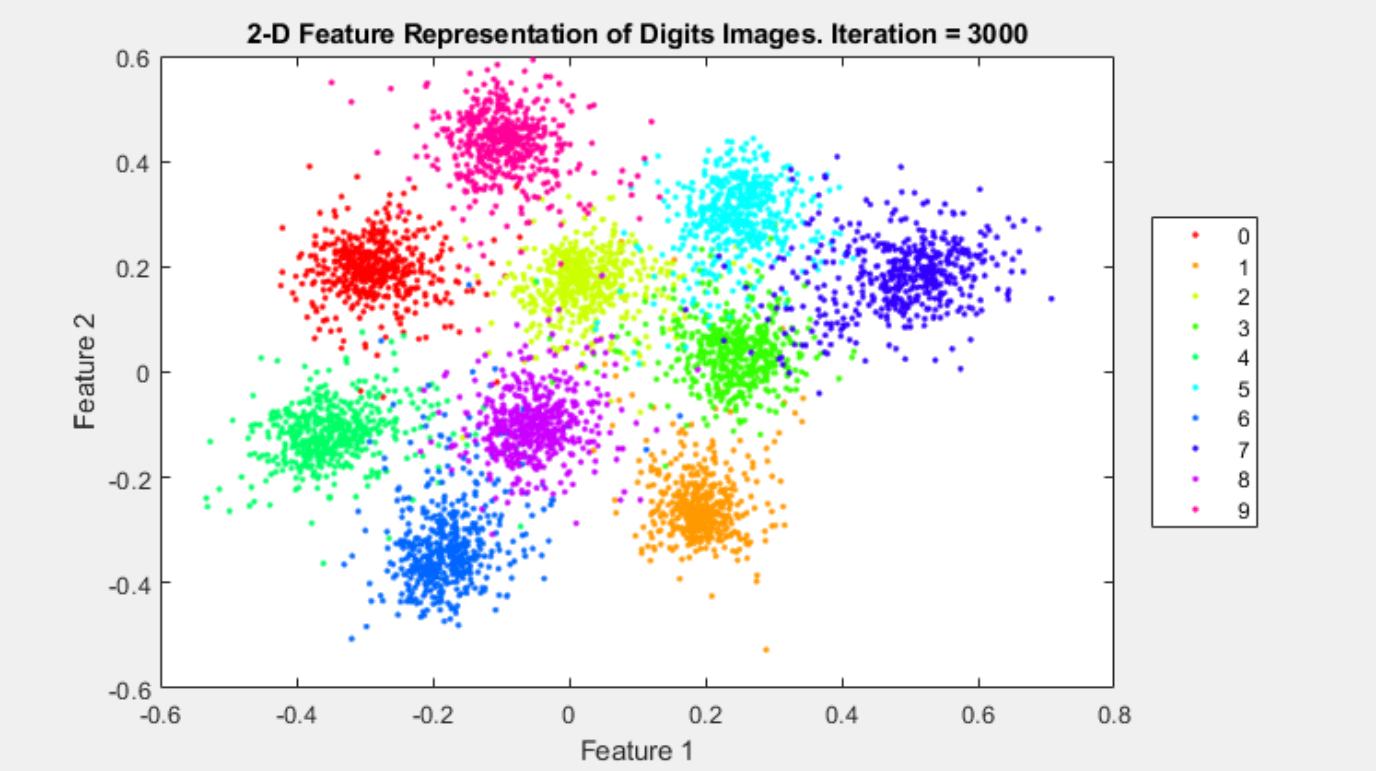
- Extract a batch of image pairs and labels using the `getSiameseBatch` function defined in the section Create Batches of Image Pairs on page 3-0 .
- Convert the image data to `dlarray` objects with underlying type `single` and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- 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.

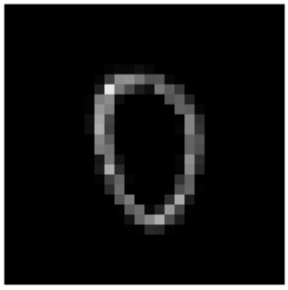
Use the Trained Network to Find Similar Images

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`.

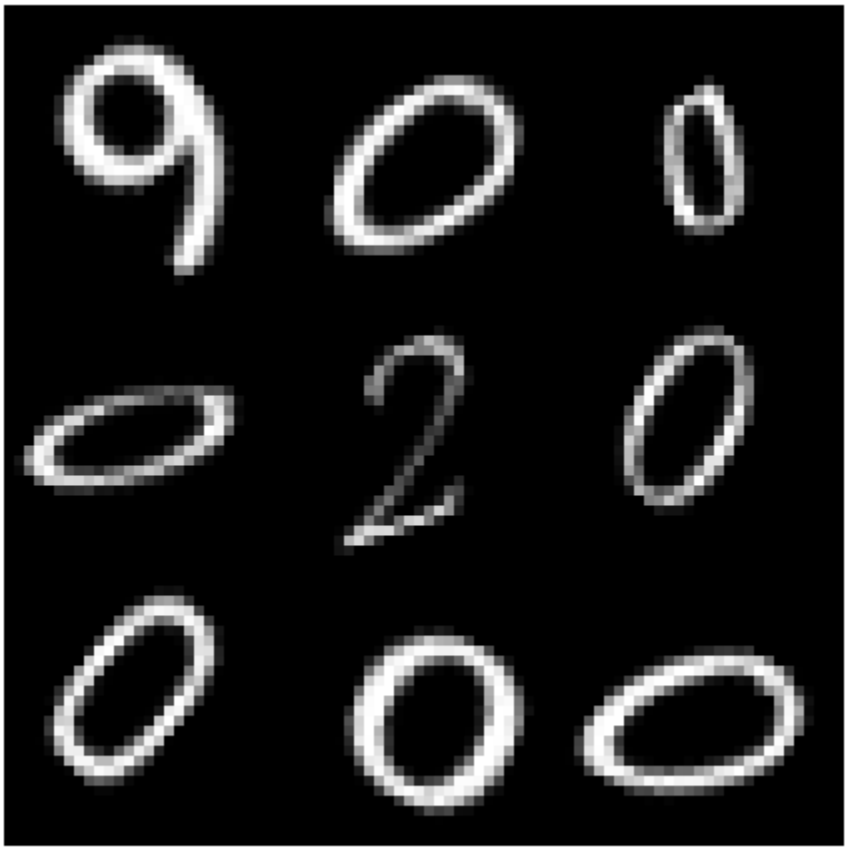
```
trialF = predict(dlnet,testImg);
```

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. LeCunn, 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. LeCunn. "*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

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Train a Siamese Network to Compare Images” on page 3-96
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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]. Extract and place the files in the working directory, then 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';
testImagesFile = 't10k-images.idx3-ubyte';
testLabelsFile = 't10k-labels.idx1-ubyte';

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 `latent_dim`, 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')
    ]);
```

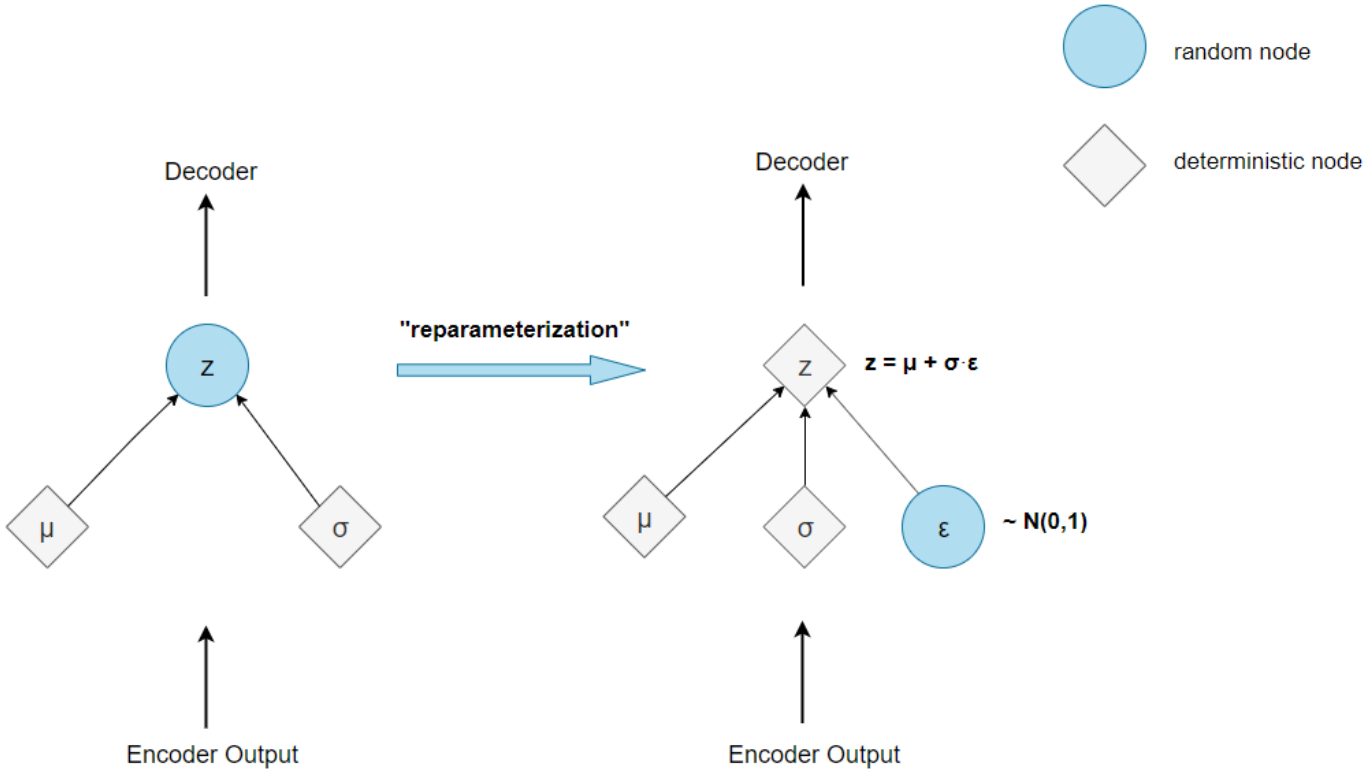
To train both networks with a custom training loop and enable automatic differentiation, convert the layer graphs to `dlnetwork` objects.

```
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

$$\text{KL loss} = -0.5 \cdot \sum_{i=1}^{n} \left(1 + \log(\sigma_i) - \mu_i^2 - \sigma_i^2\right).$$

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 (requires Parallel Computing Toolbox™). If you do not have a GPU, set the `executionEnvironment` to `"cpu"`.

```
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 = 27.0561. Time taken for epoch = 33.0037s
Epoch : 2 Test ELBO loss = 24.414. Time taken for epoch = 32.4167s
Epoch : 3 Test ELBO loss = 23.0166. Time taken for epoch = 32.3244s
Epoch : 4 Test ELBO loss = 20.9078. Time taken for epoch = 32.1268s
Epoch : 5 Test ELBO loss = 20.6519. Time taken for epoch = 32.3451s
Epoch : 6 Test ELBO loss = 20.3201. Time taken for epoch = 32.4371s
Epoch : 7 Test ELBO loss = 19.9266. Time taken for epoch = 32.4551s
Epoch : 8 Test ELBO loss = 19.8448. Time taken for epoch = 32.9919s
Epoch : 9 Test ELBO loss = 19.7485. Time taken for epoch = 33.1783s
Epoch : 10 Test ELBO loss = 19.6295. Time taken for epoch = 33.1623s
Epoch : 11 Test ELBO loss = 19.539. Time taken for epoch = 32.4781s
Epoch : 12 Test ELBO loss = 19.4682. Time taken for epoch = 32.5094s
Epoch : 13 Test ELBO loss = 19.3577. Time taken for epoch = 32.5996s
Epoch : 14 Test ELBO loss = 19.3247. Time taken for epoch = 32.6447s
Epoch : 15 Test ELBO loss = 19.3043. Time taken for epoch = 32.2494s
Epoch : 16 Test ELBO loss = 19.2948. Time taken for epoch = 32.5408s
Epoch : 17 Test ELBO loss = 19.191. Time taken for epoch = 32.8177s
Epoch : 18 Test ELBO loss = 19.1075. Time taken for epoch = 32.5982s
Epoch : 19 Test ELBO loss = 19.0606. Time taken for epoch = 33.7771s
Epoch : 20 Test ELBO loss = 19.0298. Time taken for epoch = 33.6249s
Epoch : 21 Test ELBO loss = 19.0534. Time taken for epoch = 33.4906s
Epoch : 22 Test ELBO loss = 18.9859. Time taken for epoch = 33.1101s
Epoch : 23 Test ELBO loss = 19.0077. Time taken for epoch = 32.7345s
Epoch : 24 Test ELBO loss = 18.9963. Time taken for epoch = 33.0067s
Epoch : 25 Test ELBO loss = 18.9189. Time taken for epoch = 32.891s
Epoch : 26 Test ELBO loss = 18.8925. Time taken for epoch = 33.0905s
Epoch : 27 Test ELBO loss = 18.9182. Time taken for epoch = 32.6203s
Epoch : 28 Test ELBO loss = 18.8664. Time taken for epoch = 32.4095s
Epoch : 29 Test ELBO loss = 18.8512. Time taken for epoch = 32.4317s
Epoch : 30 Test ELBO loss = 18.7983. Time taken for epoch = 32.4s
Epoch : 31 Test ELBO loss = 18.7971. Time taken for epoch = 32.4902s
Epoch : 32 Test ELBO loss = 18.7888. Time taken for epoch = 32.2591s
Epoch : 33 Test ELBO loss = 18.7811. Time taken for epoch = 32.4291s
Epoch : 34 Test ELBO loss = 18.7804. Time taken for epoch = 32.5968s
Epoch : 35 Test ELBO loss = 18.7839. Time taken for epoch = 32.3787s
Epoch : 36 Test ELBO loss = 18.7045. Time taken for epoch = 32.6078s
Epoch : 37 Test ELBO loss = 18.7783. Time taken for epoch = 32.6429s
Epoch : 38 Test ELBO loss = 18.7068. Time taken for epoch = 32.7032s
Epoch : 39 Test ELBO loss = 18.6822. Time taken for epoch = 32.3438s
Epoch : 40 Test ELBO loss = 18.7155. Time taken for epoch = 32.6521s
Epoch : 41 Test ELBO loss = 18.7161. Time taken for epoch = 32.5532s
Epoch : 42 Test ELBO loss = 18.6597. Time taken for epoch = 32.6419s
Epoch : 43 Test ELBO loss = 18.6657. Time taken for epoch = 32.4558s
Epoch : 44 Test ELBO loss = 18.5996. Time taken for epoch = 32.5503s
Epoch : 45 Test ELBO loss = 18.6666. Time taken for epoch = 32.5503s
Epoch : 46 Test ELBO loss = 18.6449. Time taken for epoch = 32.2981s
Epoch : 47 Test ELBO loss = 18.6107. Time taken for epoch = 32.3152s
Epoch : 48 Test ELBO loss = 18.6393. Time taken for epoch = 32.7135s
Epoch : 49 Test ELBO loss = 18.6351. Time taken for epoch = 32.3859s
Epoch : 50 Test ELBO loss = 18.5955. Time taken for epoch = 32.6549s
```

Visualize Results

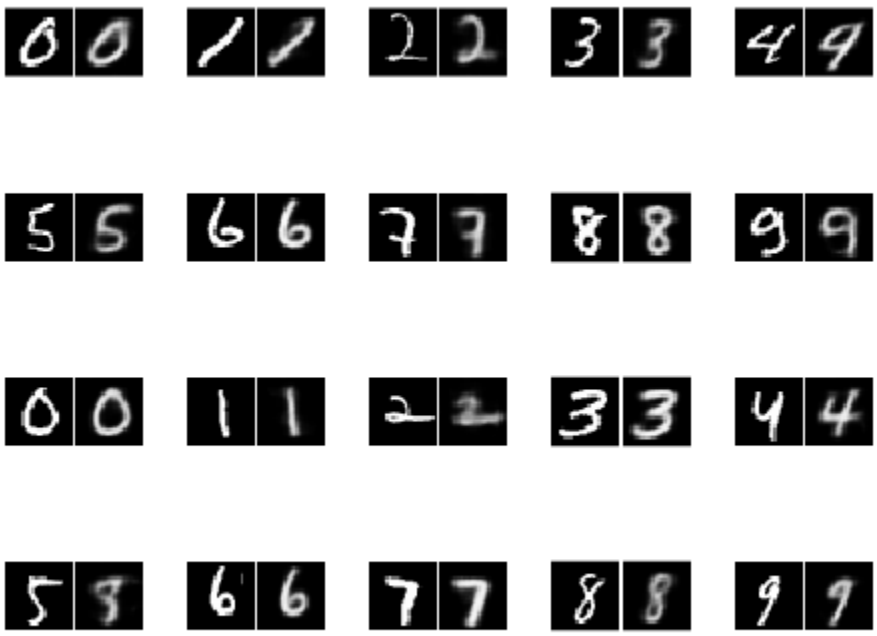
To visualize and interpret the results, use the helper Visualization functions on page 3-0 . These helper functions are defined at the end of this example.

The `VisualizeReconstruction` function shows a randomly chosen digit from each class accompanied by its reconstruction after passing through the autoencoder.

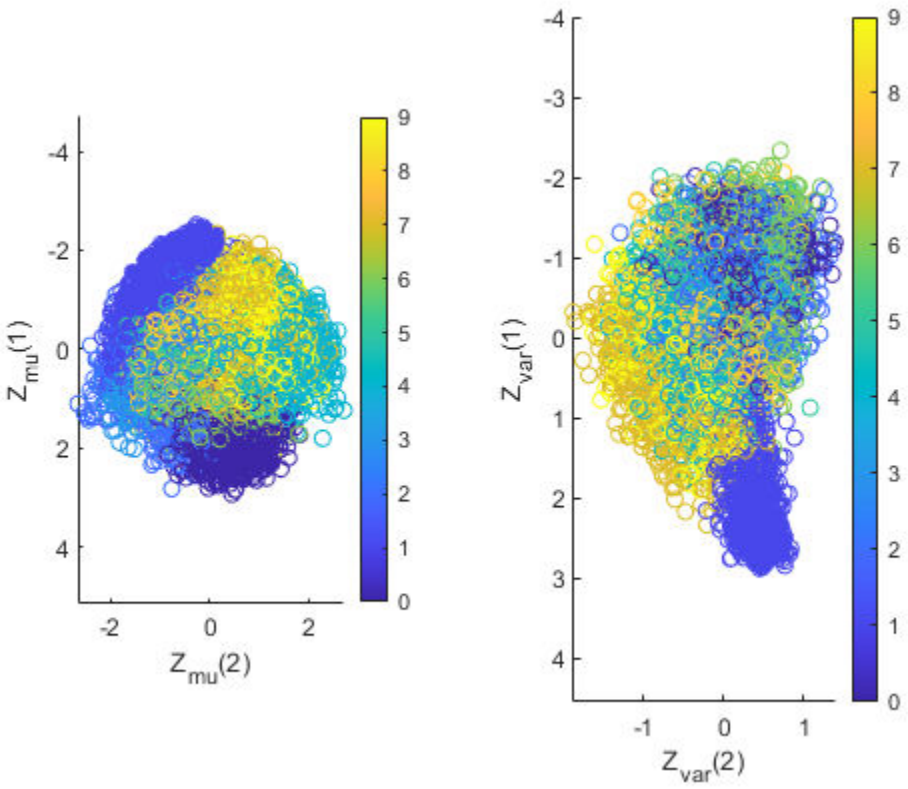
The `VisualizeLatentSpace` function takes the mean and the variance encodings (each of dimension 20) generated after passing the test images through the encoder network, and performs principal component analysis (PCA) on the matrix containing the encodings for each of the images. You can then visualize 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.

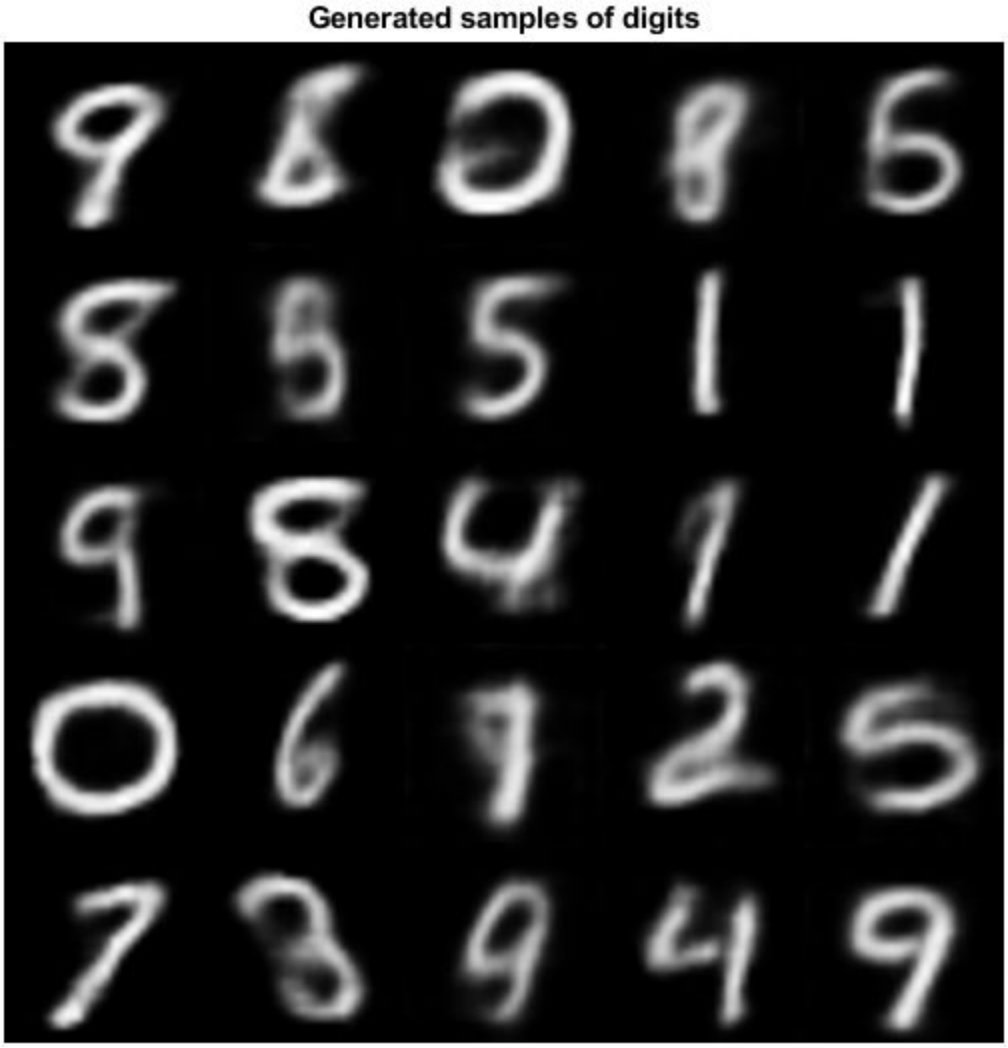
```
visualizeReconstruction(XTest, YTest, encoderNet, decoderNet)
```



```
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 3-72.

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

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `layerGraph` | `sigmoid`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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

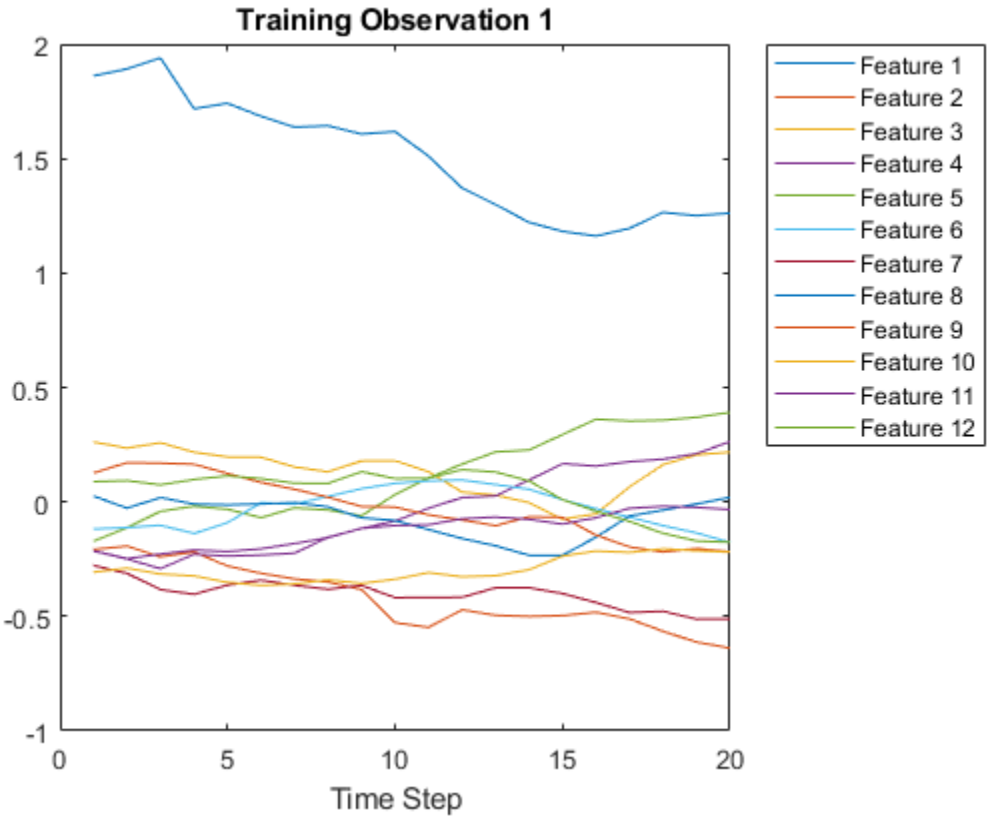
Load the Japanese Vowels training data. `XTrain` is a cell array containing 270 sequences of dimension 12 of varying length. `Y` is a categorical vector of labels "1","2",...,"9", which correspond to the nine speakers. The entries in `XTrain` are matrices with 12 rows (one row for each feature) and varying 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.

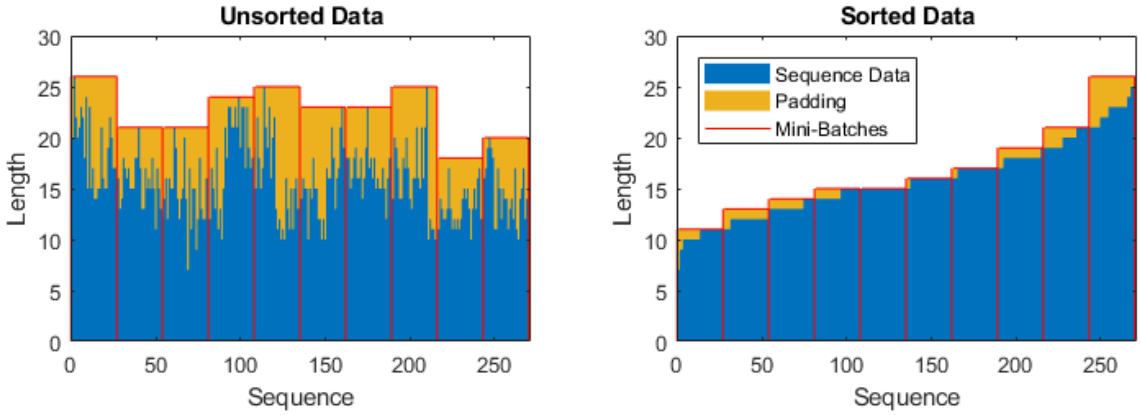
```
figure
plot(XTrain{1}')
xlabel("Time Step")
title("Training Observation 1")
numFeatures = size(XTrain{1},1);
legend("Feature " + string(1:numFeatures),'Location','northeastoutside')
```



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.

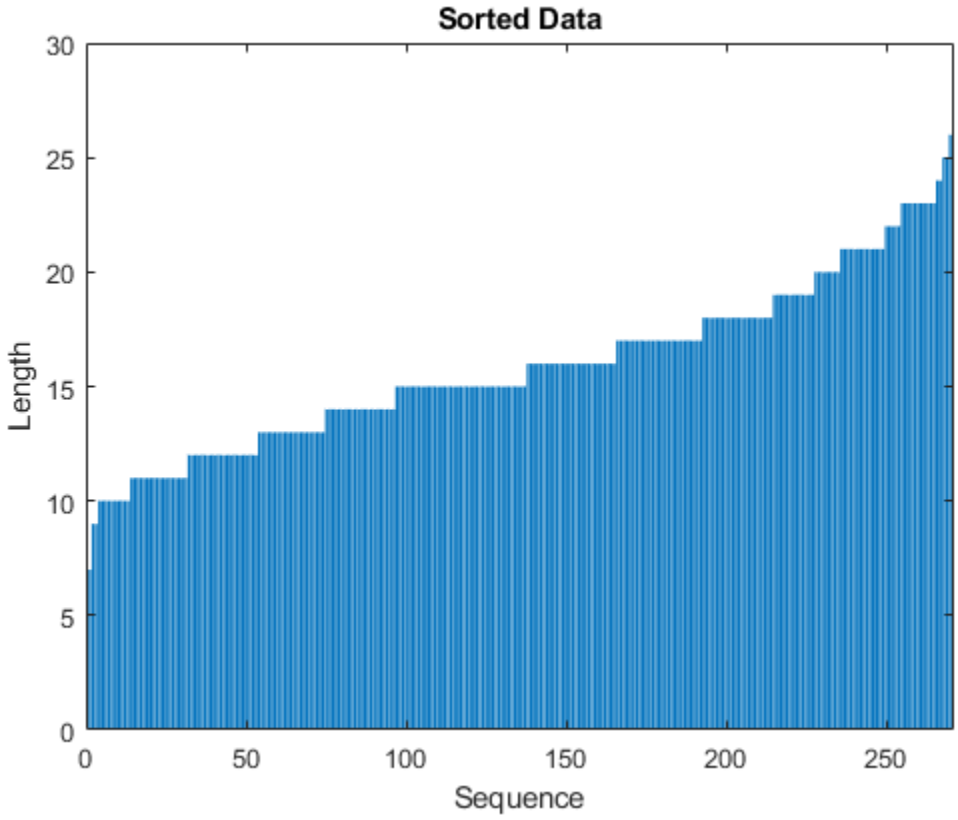
```
numObservations = numel(XTrain);
for i=1:numObservations
    sequence = XTrain{i};
    sequenceLengths(i) = size(sequence,2);
end
```

Sort the data by sequence length.

```
[sequenceLengths,idx] = sort(sequenceLengths);
XTrain = XTrain(idx);
YTrain = YTrain(idx);
```

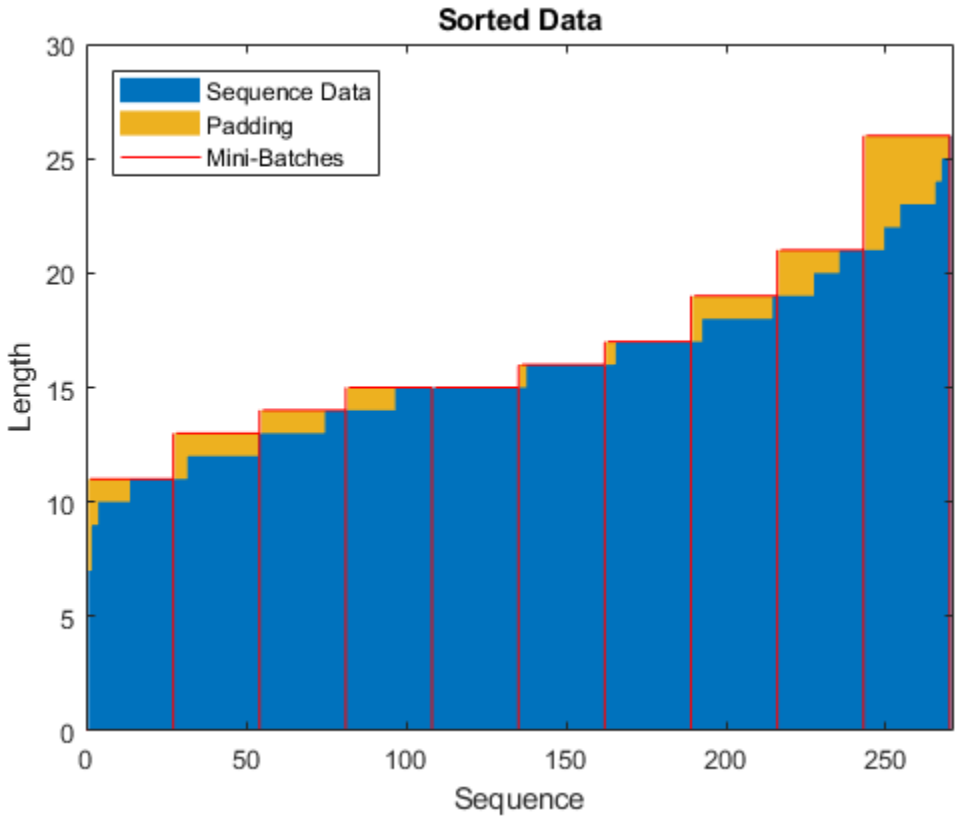
View the sorted sequence lengths in a bar chart.

```
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.

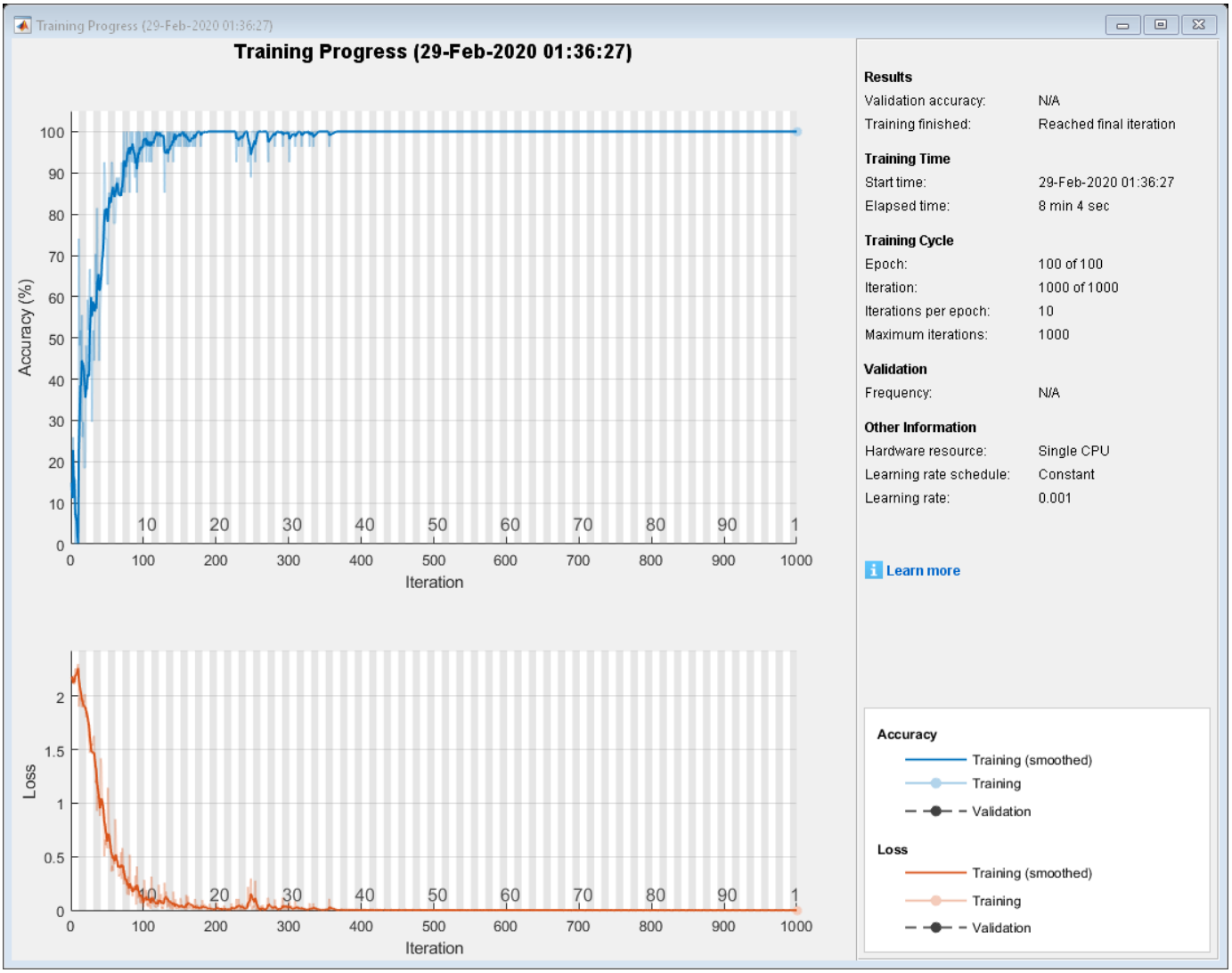
Since the mini-batches are small with short sequences, training is better suited for the CPU. Specify `'ExecutionEnvironment'` to be `'cpu'`. To train on a GPU, if available, set `'ExecutionEnvironment'` to `'auto'` (this is the default value).

```
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

`bilstmLayer` | `lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “Deep Learning in MATLAB” on page 1-2

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.

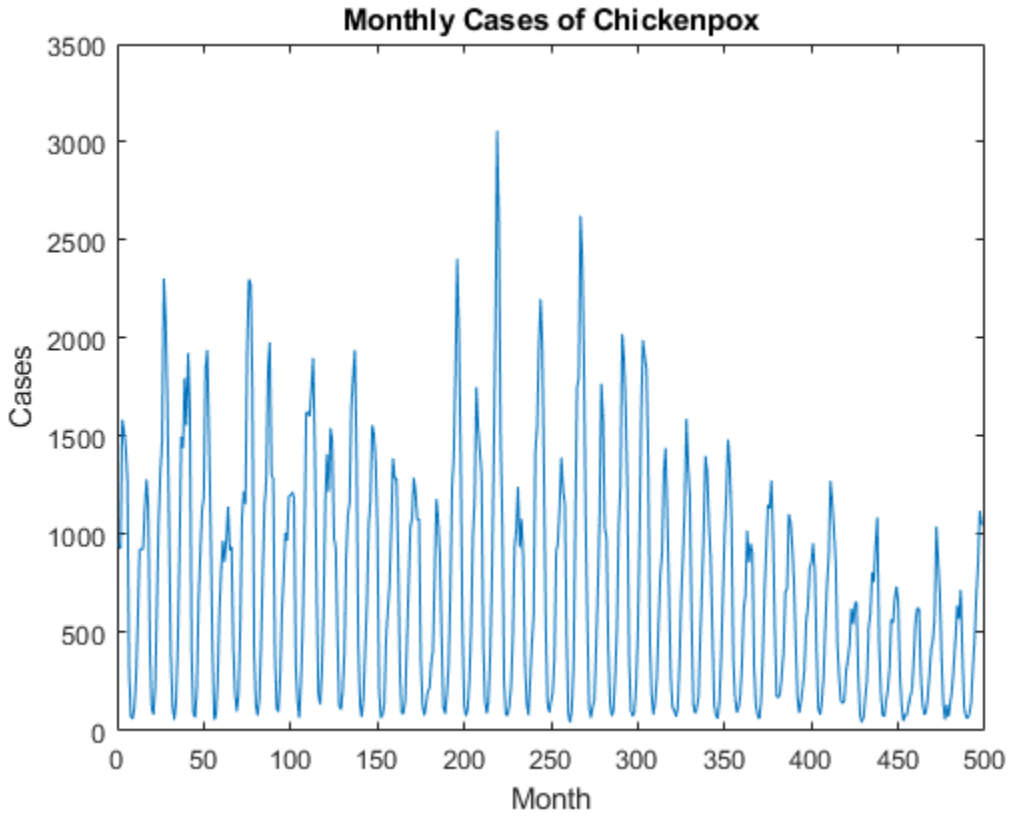
To forecast the values of multiple time steps in the future, use the `predictAndUpdateState` function to predict time steps one at a time and update the network state at each prediction.

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];
```

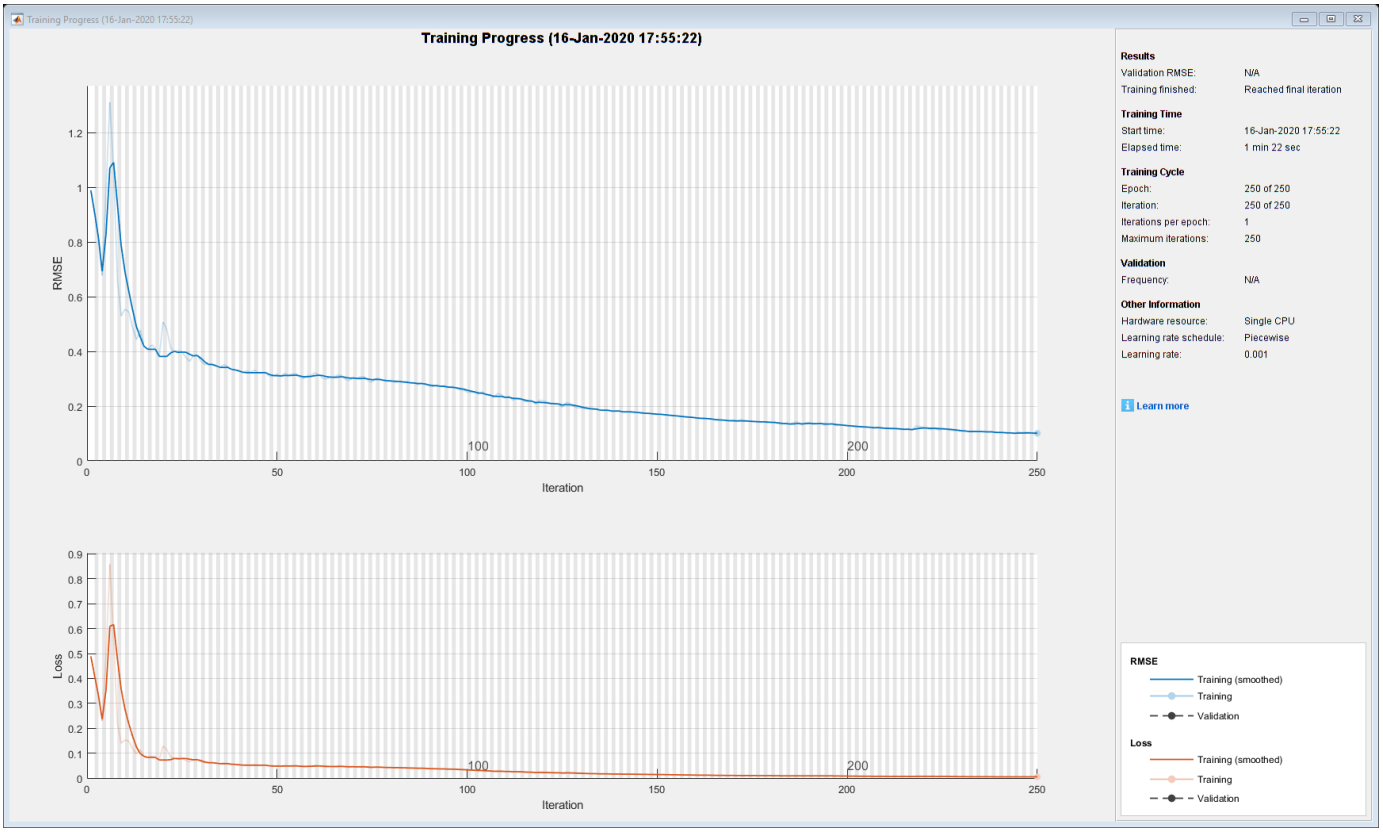
Specify the training options. Set the solver to `'adam'` and train for 250 epochs. To prevent the gradients from exploding, set the gradient threshold to 1. Specify the initial learn rate 0.005, and drop the learn rate after 125 epochs by multiplying by a factor of 0.2.

```
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;
```

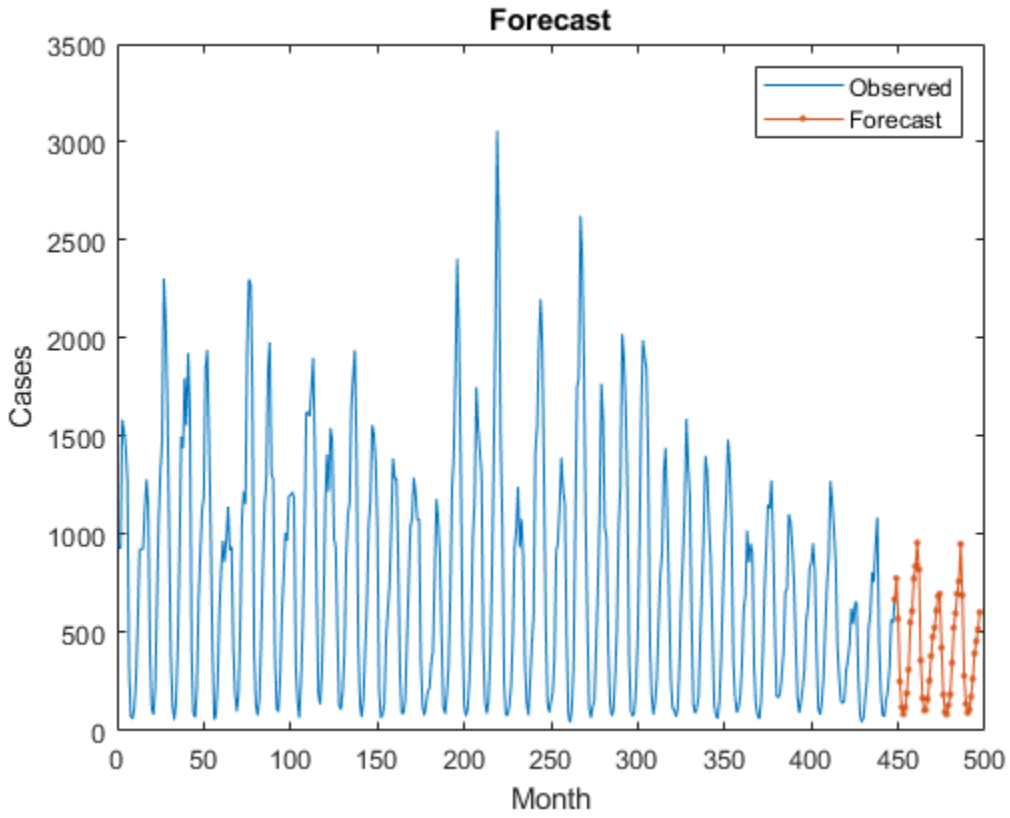
The training progress plot reports the root-mean-square error (RMSE) calculated from the standardized data. Calculate the RMSE from the unstandardized predictions.

```
YTest = dataTest(2:end);
rmse = sqrt(mean((YPred-YTest).^2))
```

```
rmse = single
    248.5531
```

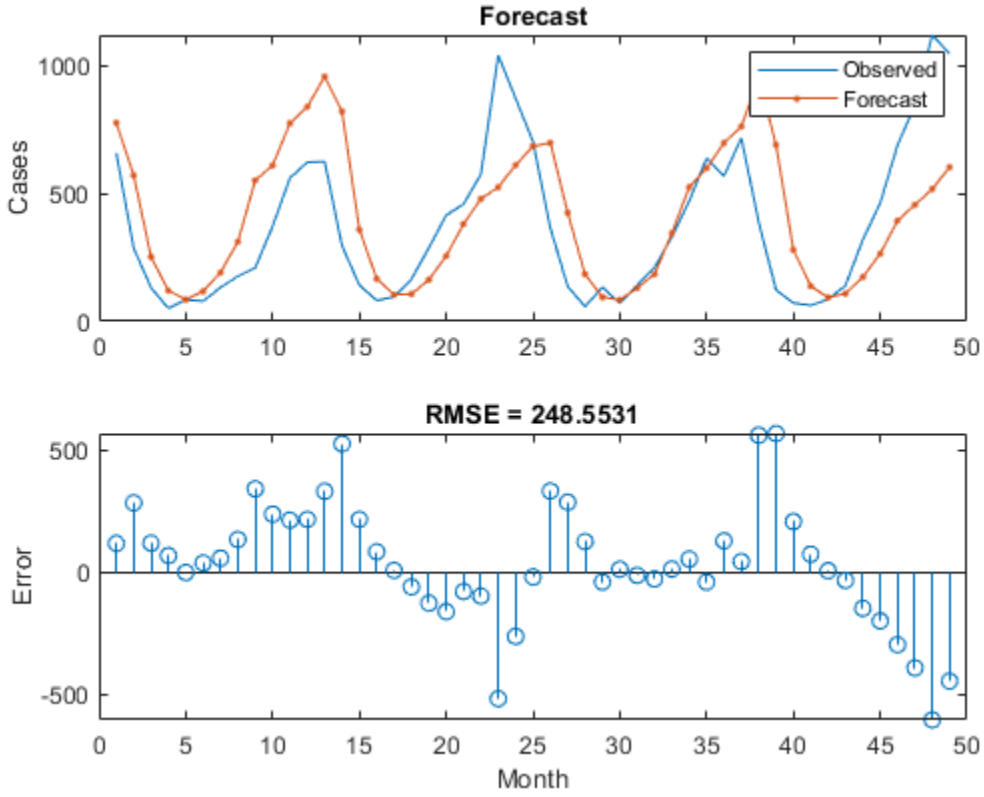
Plot the training time series with the forecasted values.

```
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;
```

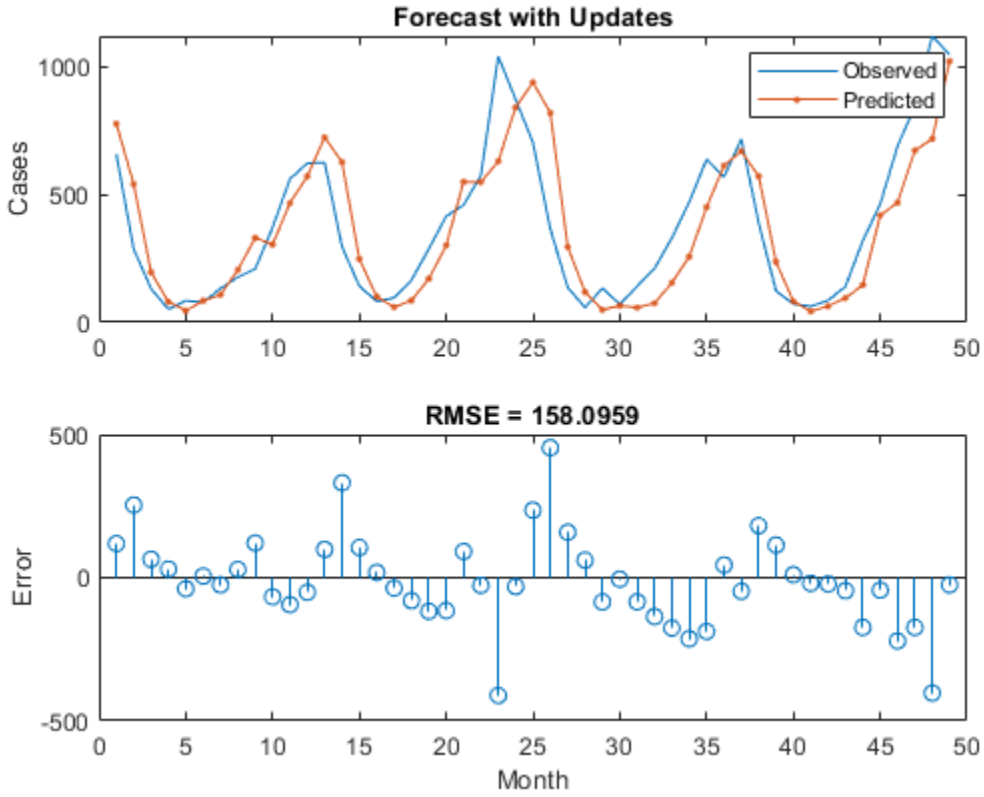
Calculate the root-mean-square error (RMSE).

```
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

`lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Generate Text Using Deep Learning” on page 4-131
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “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 = 0.01 * randn(16e3,1);
```

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));
```

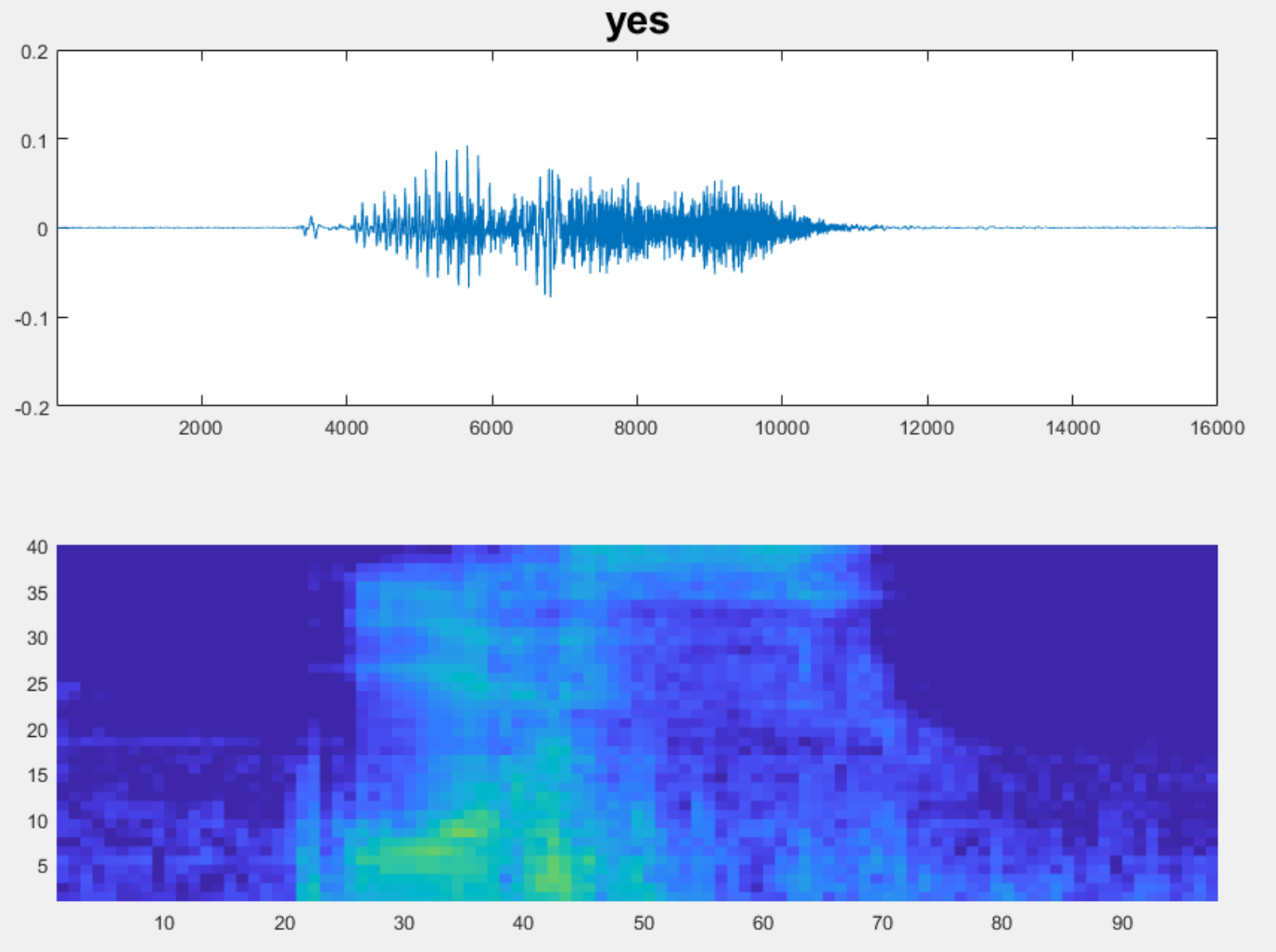
Initialize a buffer for the audio. Extract the classification labels of the network. Initialize buffers of half a second for the labels and classification probabilities of the streaming audio. Use these buffers to compare the classification results over a longer period of time and by that build 'agreement' 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
```



Load Speech Commands Data Set

Download and extract the data set [1].

```
url = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz'

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading speech commands data set (1.5 GB)...')
    untar(url,datasetFolder)
end
```

Create an `audioDatastore` that points to the data set.

```
ads = audioDatastore(datasetFolder, ...
    'IncludeSubfolders',true, ...
    'FileExtensions','.wav', ...
    'LabelSource','foldernames')
```

```
ads =

  audioDatastore with properties:

                      Files: {
                             ' ...\Local\Temp\google_speech\_background_noise_\doing_the_dishes
                             ' ...\AppData\Local\Temp\google_speech\_background_noise_\dude_miac
                             ' ...\AppData\Local\Temp\google_speech\_background_noise_\exercise_
                              ... and 64724 more
                             }
                    Folders: {
                             'C:\Users\bhemmat\AppData\Local\Temp\google_speech'
                             }
                     Labels: [_background_noise_; _background_noise_; _background_noise_ ... and
    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. Do not include the longer files with background noise in the training set. Background noise will be added in a separate step later.

Use `subset` 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 = ~ismember(ads.Labels,[commands,"_background_noise_"]);

includeFraction = 0.2;
mask = rand(numel(ads.Labels),1) < includeFraction;
isUnknown = isUnknown & mask;
ads.Labels(isUnknown) = categorical("unknown");

adsSubset = subset(ads,isCommand|isUnknown);
countEachLabel(adsSubset)
```

```
ans =

  11×2 table

     Label     Count
    _______    _____

    down       2359
    go         2372
```

```
    left       2353
    no         2375
    off        2357
    on         2367
    right      2367
    stop       2380
    unknown    8186
    up         2375
    yes        2377
```

Split Data into Training, Validation, and Test Sets

The data set folder contains text files, which list the audio files to be used as the validation and test sets. These predefined validation and test sets do not contain utterances of the same word by the same person, so it is better to use these predefined sets than to select a random subset of the whole data set.

Because this example trains a single network, it only uses the validation set and not the test set to evaluate the trained model. If you train many networks and choose the network with the highest validation accuracy as your final network, then you can use the test set to evaluate the final network.

Read the list of validation files.

```
c = importdata(fullfile(datasetFolder,'validation_list.txt'));
filesValidation = string(c);
```

Read the list of test files.

```
c = importdata(fullfile(datasetFolder,'testing_list.txt'));
filesTest = string(c);
```

Determine which files in the datastore should go to validation set and which should go to test set.

```
files = adsSubset.Files;
sf = split(files,filesep);
isValidation = ismember(sf(:,end-1) + "/" + sf(:,end),filesValidation);
isTest = ismember(sf(:,end-1) + "/" + sf(:,end),filesTest);

adsValidation = subset(adsSubset,isValidation);
adsTrain = subset(adsSubset,~isValidation & ~isTest);
```

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` 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);
```

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
```

The `audioFeatureExtractor` normalizes auditory spectrograms by the window power so that measurements are independent of the type of window and length of windowing. 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 avoid roundoff error, you will reverse the window normalization.

Determine the denormalization factor to apply.

```
unNorm = 2/(sum(afe.Window)^2);
```

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 =
```

```
        25041
```

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.

```
XTrain = XTrain/unNorm;
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 = XValidation/unNorm;
XValidation = log10(XValidation + epsil);
```

Isolate the train and validation labels. Remove empty categories.

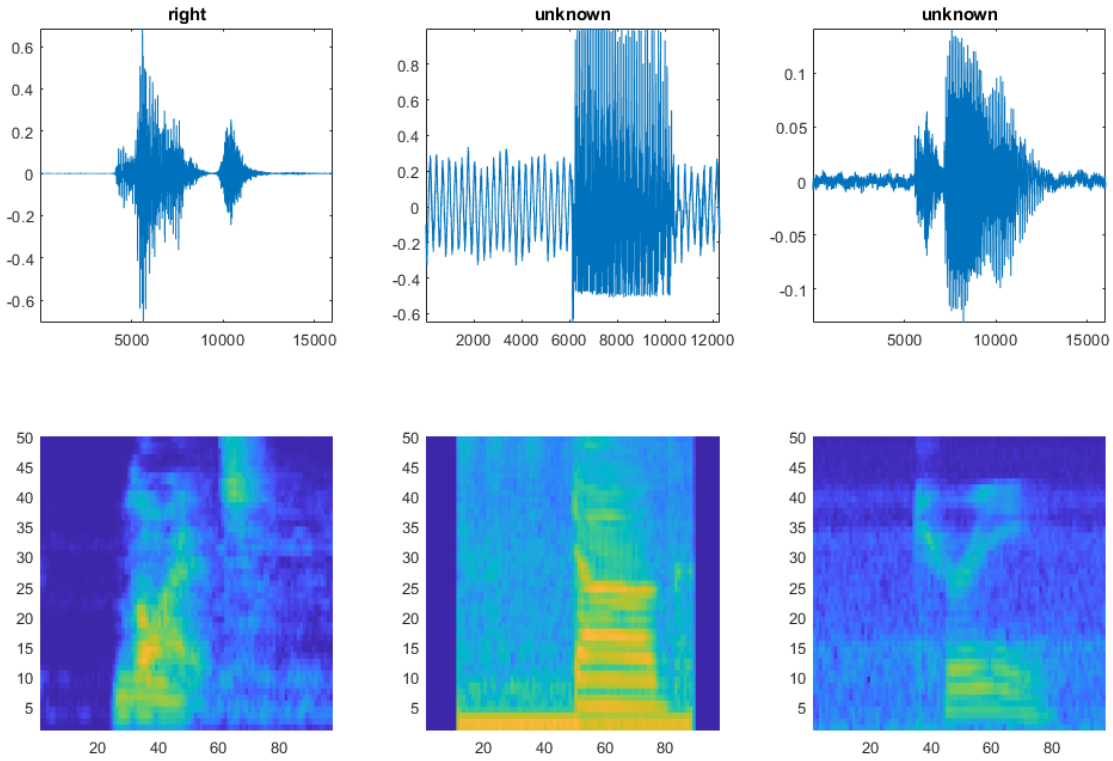
```
YTrain = removecats(adsTrain.Labels);
YValidation = removecats(adsValidation.Labels);
```

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_noise_` 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_noise_` 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 = subset(ads,ads.Labels=="_background_noise_");
 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 = Xbkg/unNorm;
Xbkg = log10(Xbkg + epsil);
```

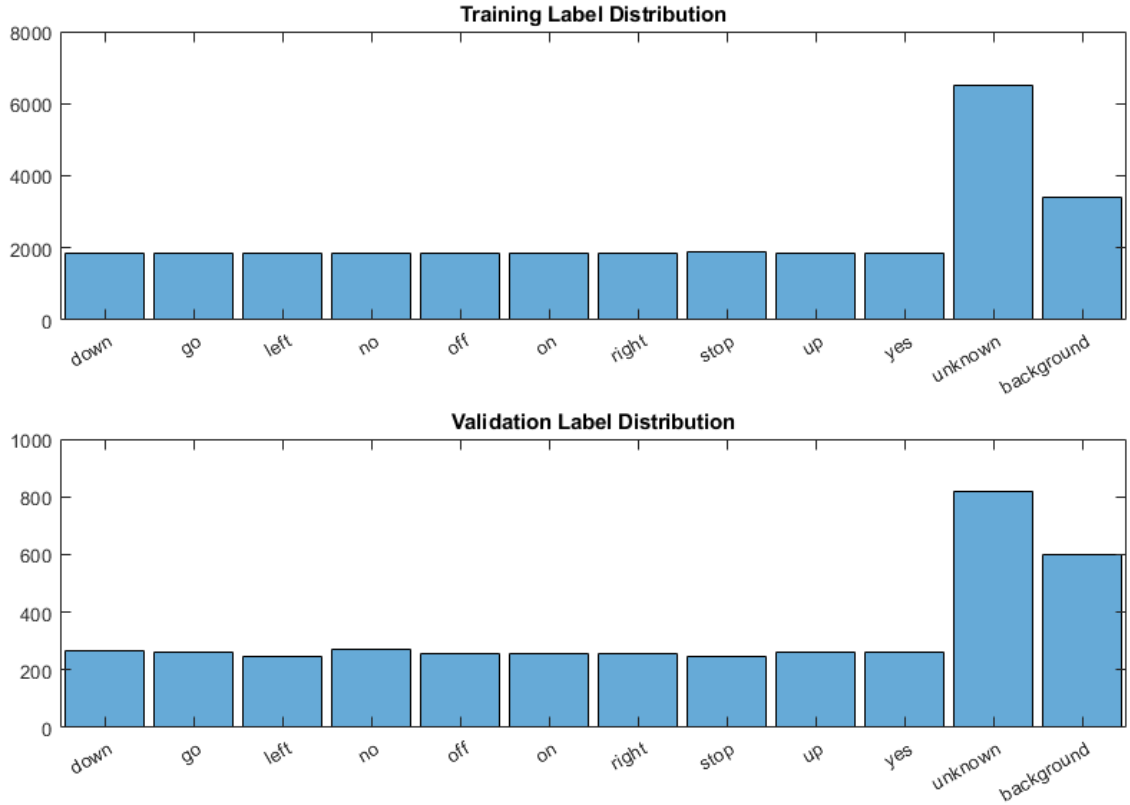
```
Processed 1000 background clips out of 4000
Processed 2000 background clips out of 4000
Processed 3000 background clips out of 4000
Processed 4000 background clips out of 4000
```

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

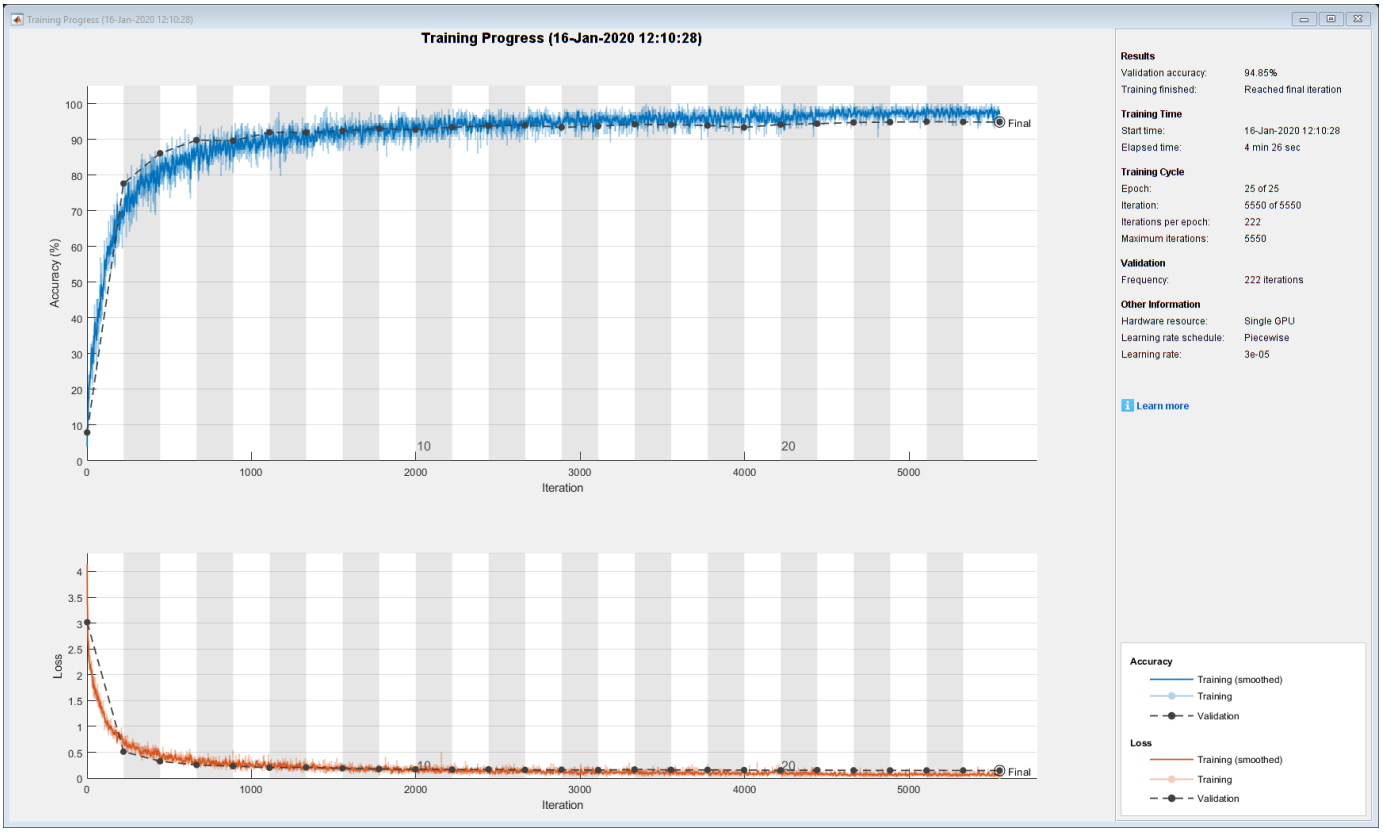
Specify the training options. Use the Adam optimizer with a mini-batch size of 128. Train for 25 epochs and reduce the learning rate by a factor of 10 after 20 epochs.

```
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

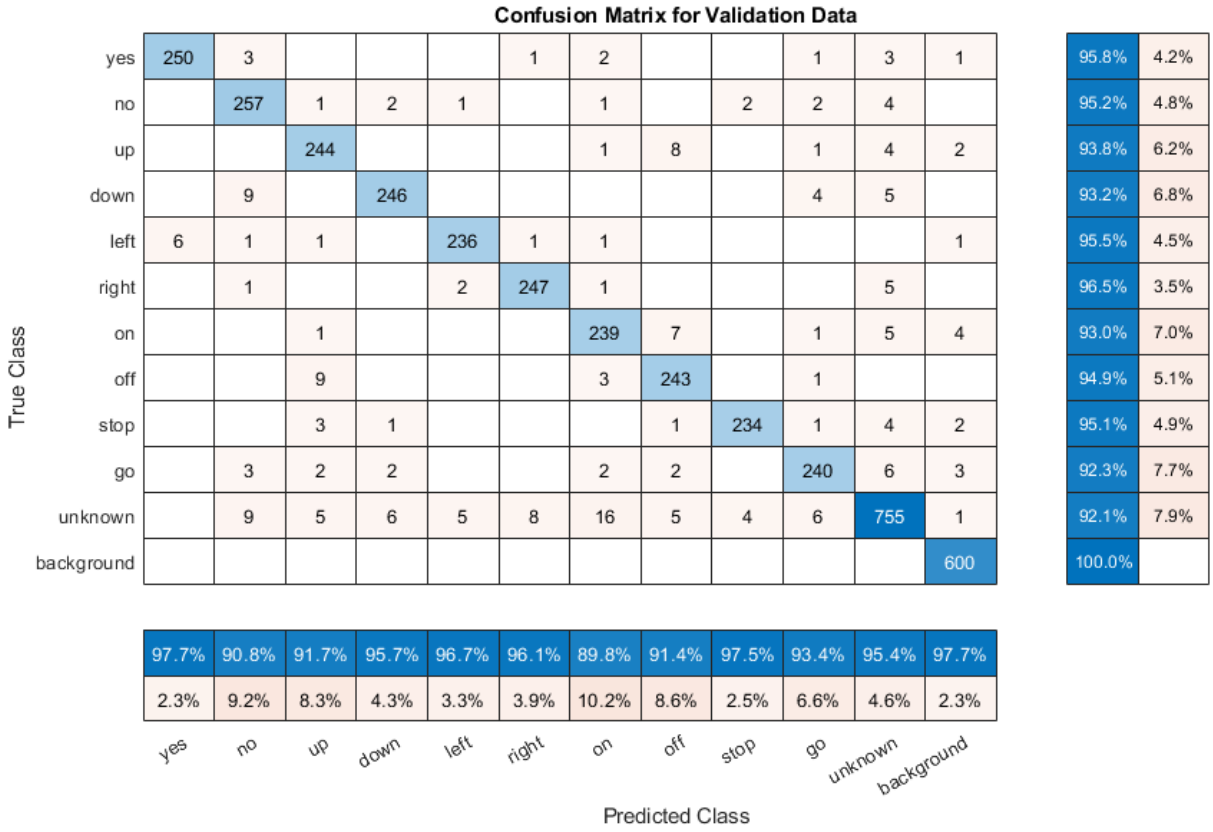
Calculate the final accuracy of the network on the training set (without data augmentation) and validation set. The network is very accurate on this data set. However, the training, validation, and test data all have similar distributions that do not necessarily reflect real-world environments. This 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: 1.526%
Validation error: 5.1539%
```

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: 286.7314 kB
Single-image prediction time on CPU: 3.1647 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

`analyzeNetwork` | `classify` | `trainNetwork`

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

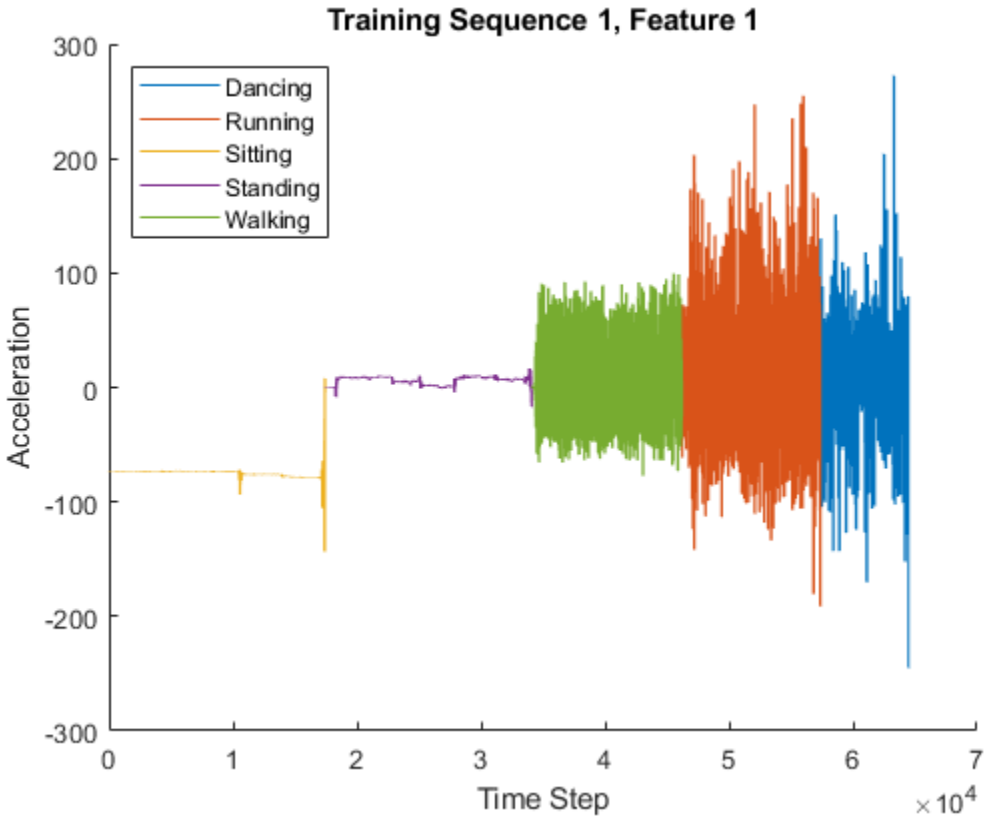
Load the human activity recognition data. The data contains seven time series of sensor data obtained from a smartphone worn on the body. Each sequence has three features and varies in length. The three features correspond to the accelerometer readings in three different directions.

```
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.

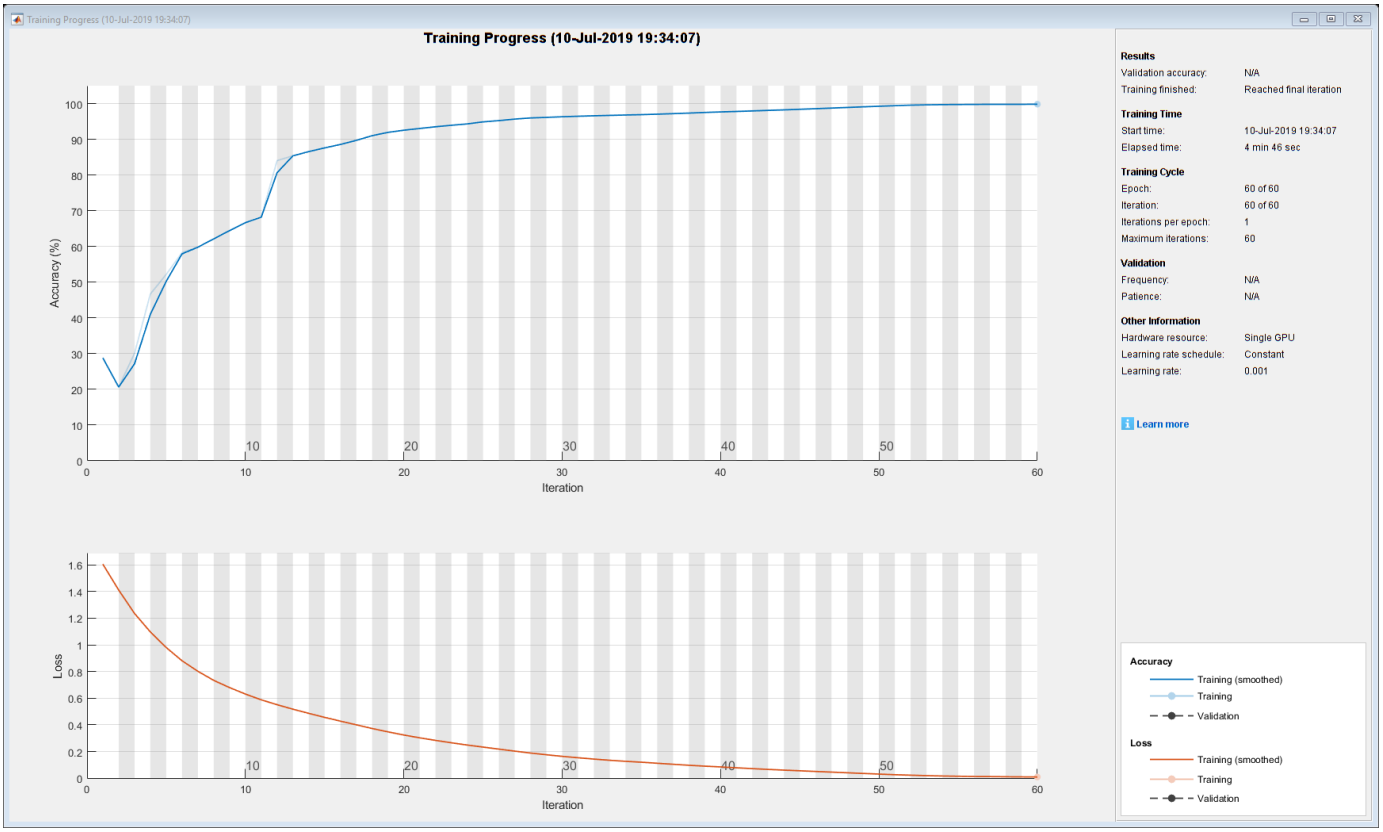
```
numFeatures = 3;
numHiddenUnits = 200;
numClasses = 5;

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);
```

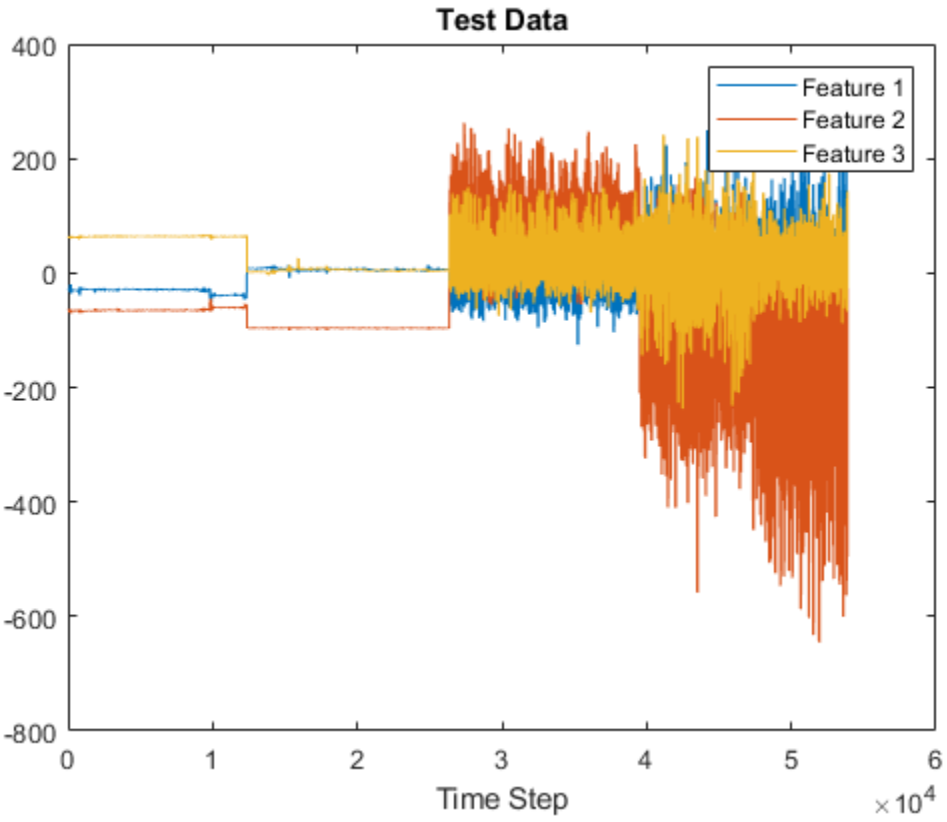


Test LSTM Network

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 4-9.

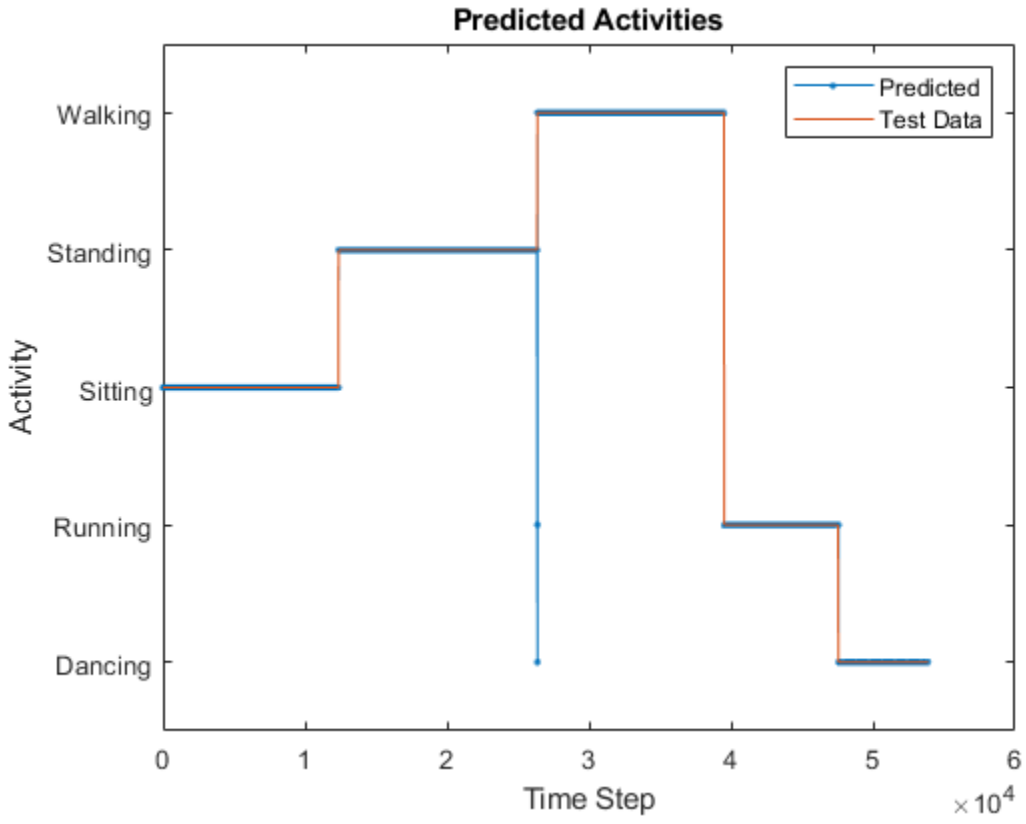
Calculate the accuracy of the predictions.

```
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

`lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “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 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.

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 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 1–17

```
filename = "CMAPSSData.zip";
dataFolder = "data";
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
```

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.

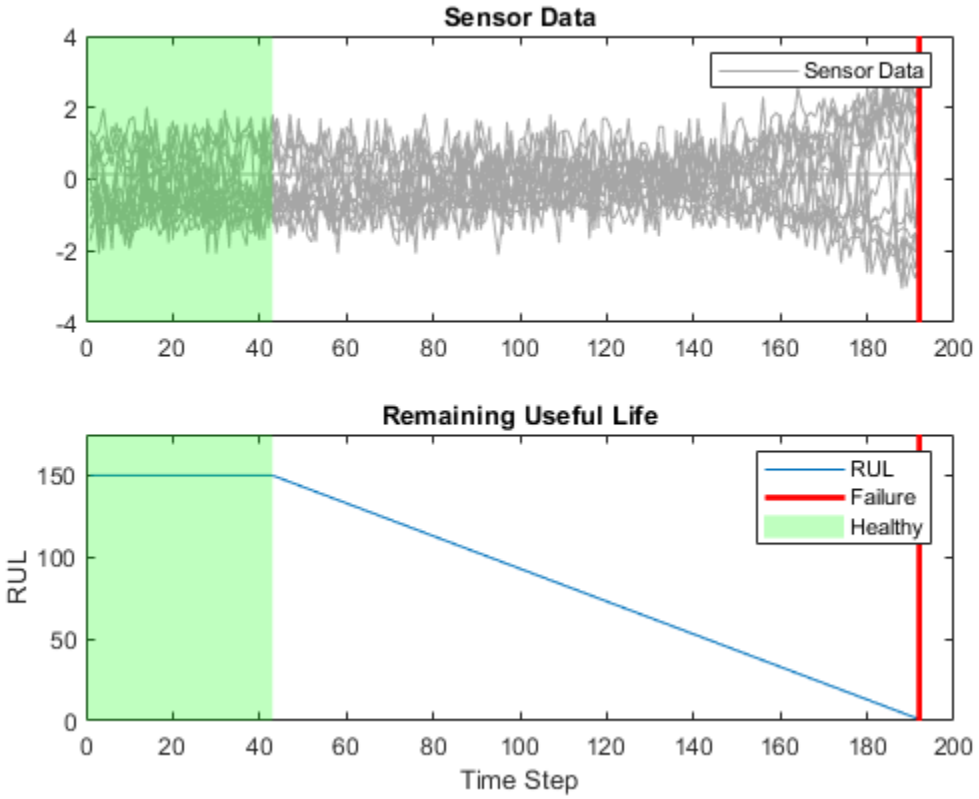
```
mu = mean([XTrain{:}],2);
sig = std([XTrain{:}],0,2);

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

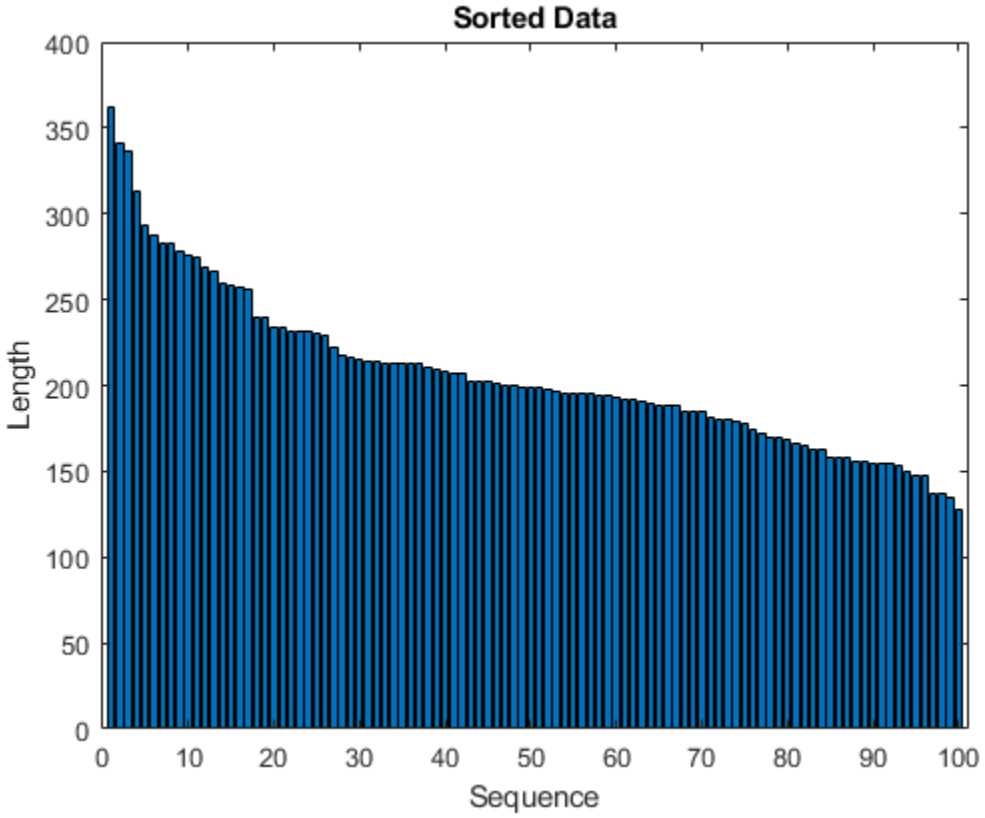
To minimize the amount of padding added to the mini-batches, sort the training data by sequence length. Then, choose a mini-batch size which divides the training data evenly and reduces the amount of padding in the mini-batches.

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);
```

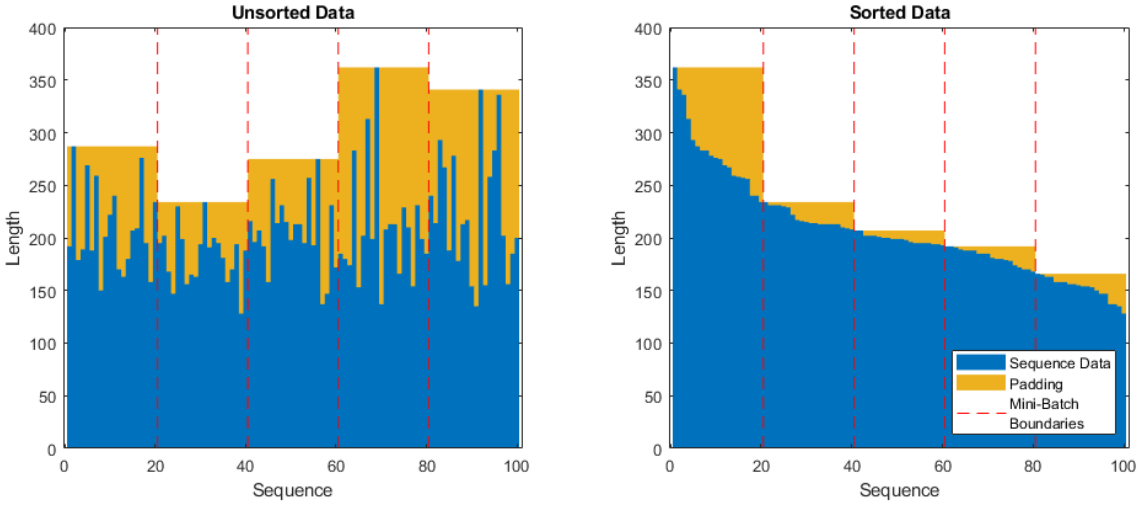
View the sorted sequence lengths in a bar chart.

```
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);
featureDimension = size(XTrain{1},1);
numHiddenUnits = 200;

layers = [ ...
    sequenceInputLayer(featureDimension)
    lstmLayer(numHiddenUnits,'OutputMode','sequence')
    fullyConnectedLayer(50)
    dropoutLayer(0.5)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

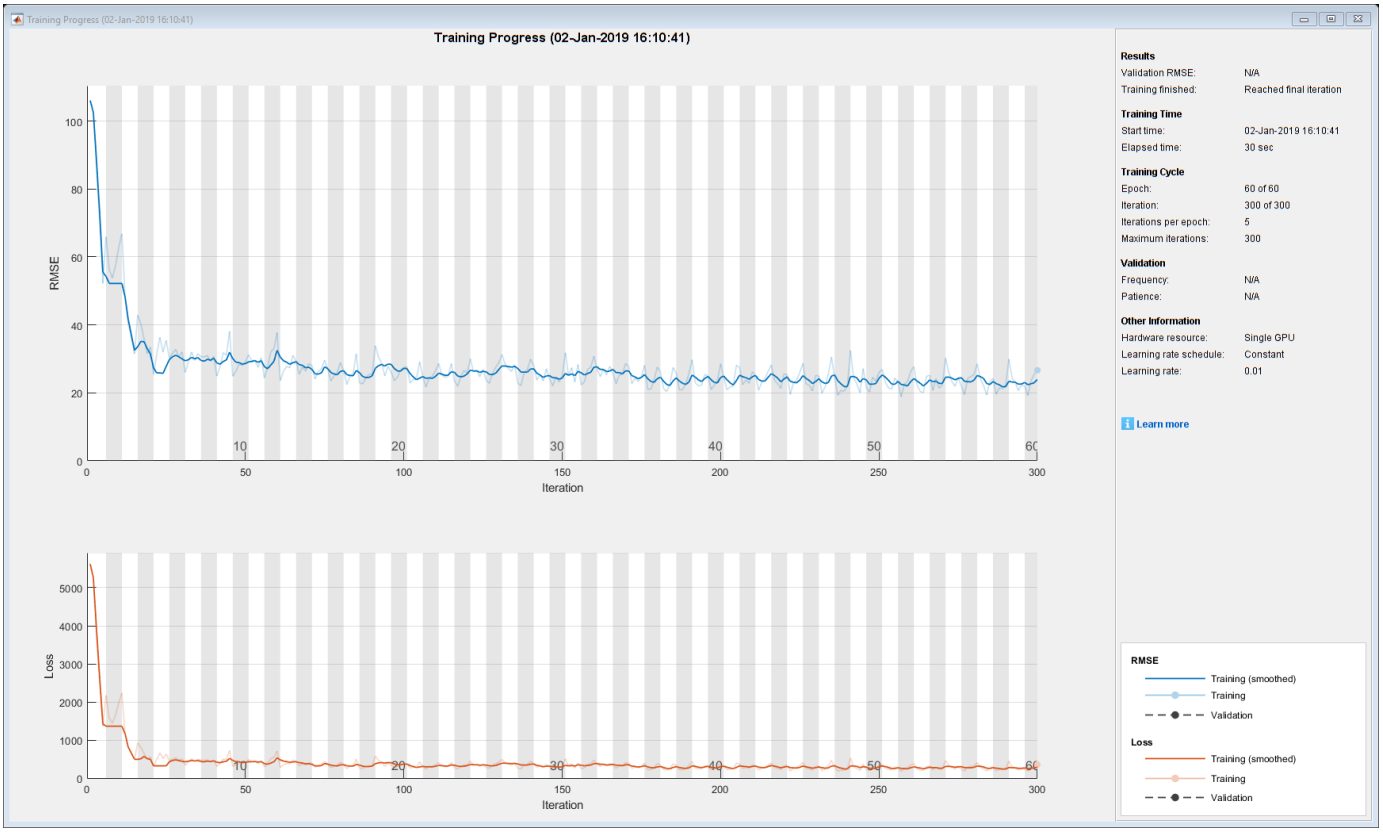
Specify the training options. Train for 60 epochs with mini-batches of size 20 using the solver `'adam'`. Specify the learning rate 0.01. To prevent the gradients from exploding, set the gradient threshold to 1. To keep the sequences sorted by length, set `'Shuffle'` to `'never'`.

```
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);
```

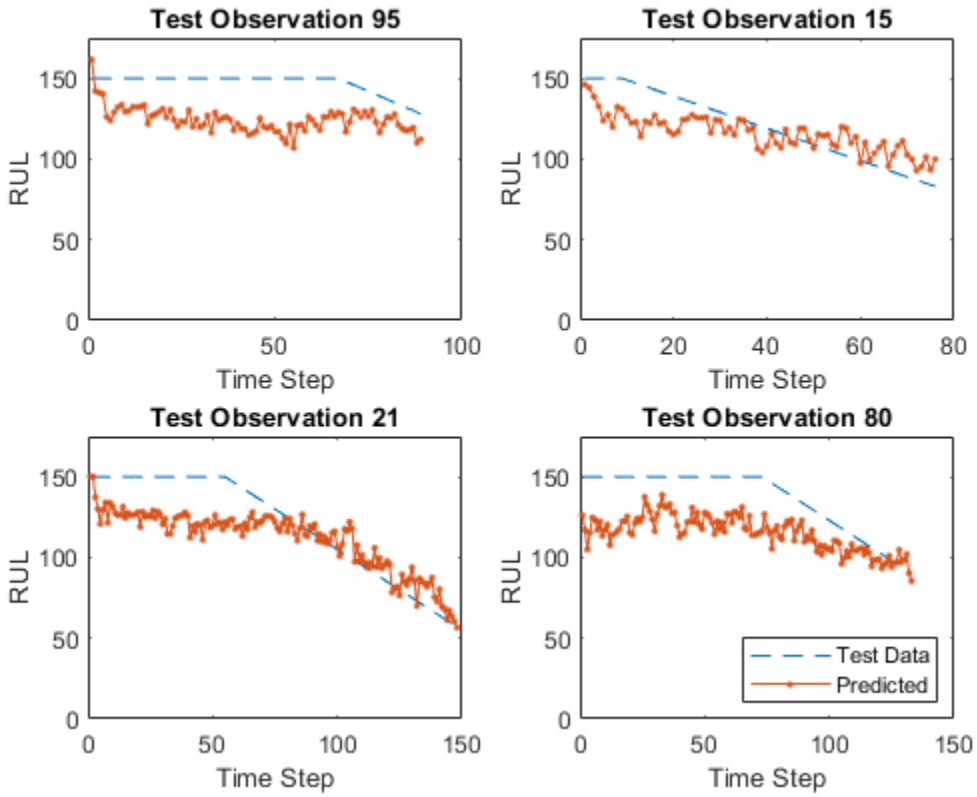
The LSTM network makes predictions on the partial sequence one time step at a time. At each time step, the network predicts using the value at this time step, and the network state calculated from the

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 4-9.

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')
```

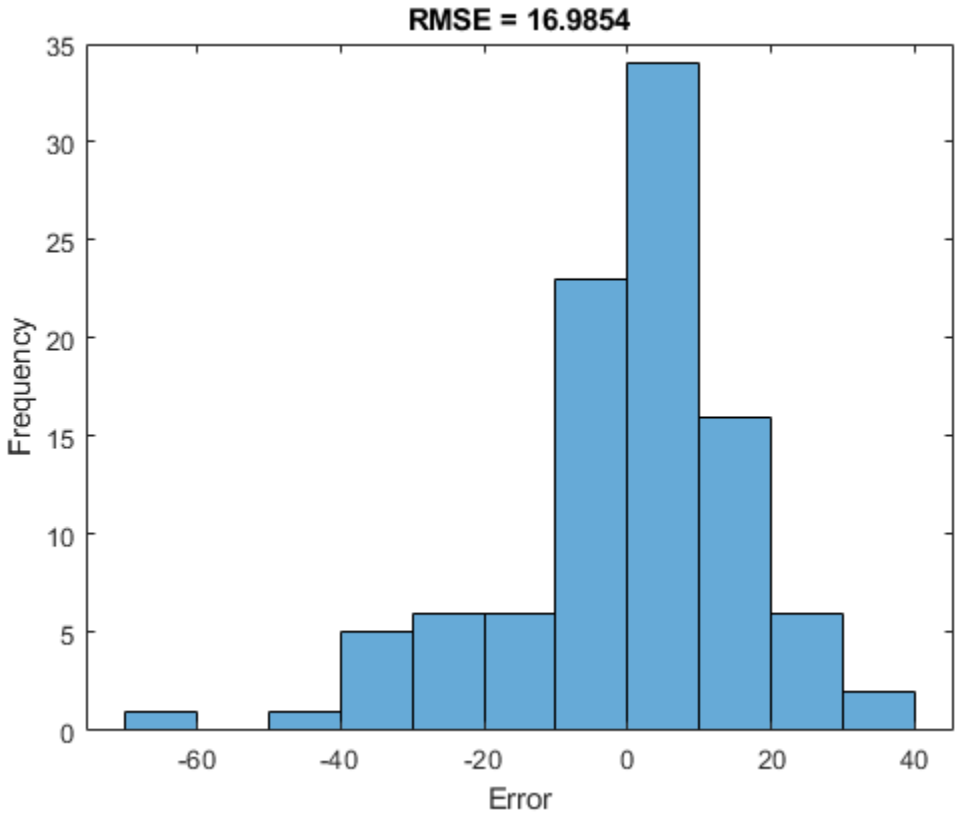


For a given partial sequence, the predicted current RUL is the last element of the predicted sequences. Calculate the root-mean-square error (RMSE) of the predictions, and visualize the prediction error in a histogram.

```
for i = 1:numel(YTest)
    YTestLast(i) = YTest{i}(end);
    YPredLast(i) = YPred{i}(end);
end
figure
rmse = sqrt(mean((YPredLast - YTestLast).^2))
```

```
rmse = single
    16.9854
```

```
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

`lstmLayer` | `predictAndUpdateState` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

See Also

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Long Short-Term Memory Networks” on page 1-53
- “Deep Learning in MATLAB” on page 1-2

Classify Videos Using Deep Learning

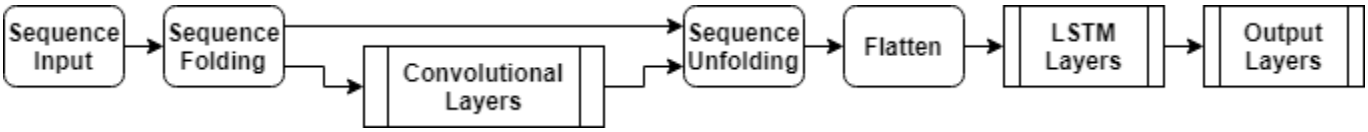
This example shows how to create a network for video classification by combining a pretrained image classification model and an LSTM network.

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
```

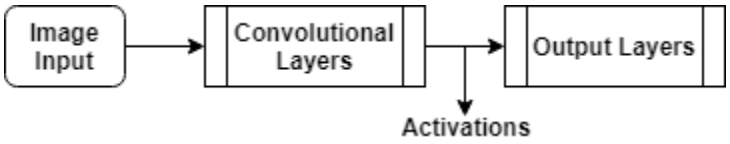
To view the video, use the `implay` function (requires Image Processing Toolbox™). This function expects data in the range [0,1], so you must first divide the data by 255. Alternatively, you can loop over the individual frames and use the `imshow` function.

```
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.

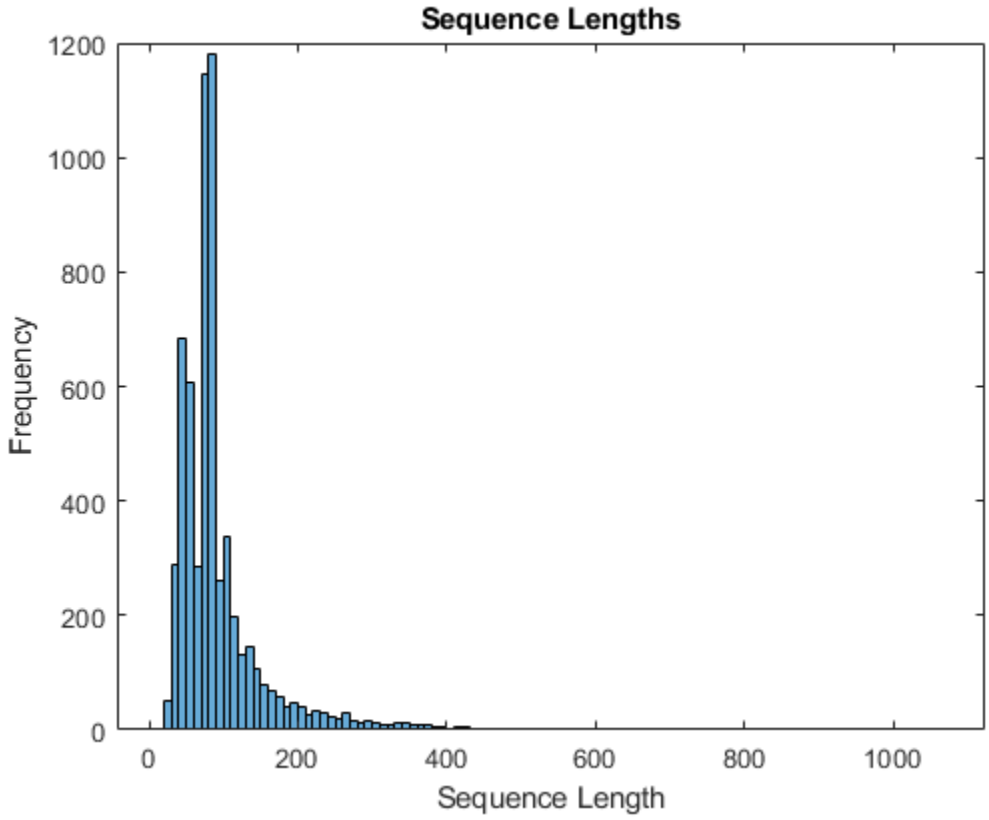
```
numObservations = numel(sequences);
idx = randperm(numObservations);
N = floor(0.9 * numObservations);

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.

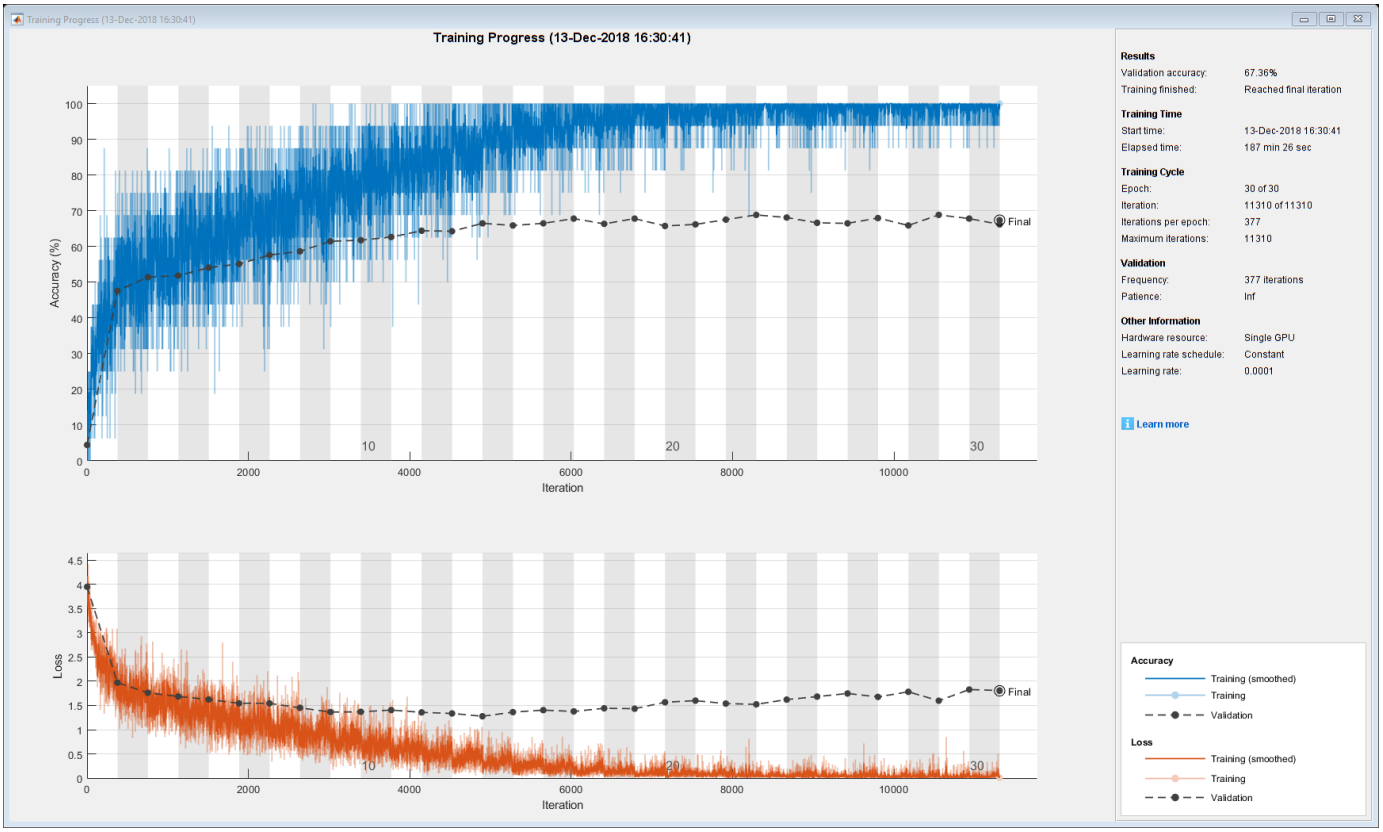
- Set a mini-batch size 16, an initial learning rate of 0.0001, and a gradient threshold of 2 (to prevent the gradients from exploding).
- Truncate the sequences in each mini-batch to have the same length as the shortest sequence.
- Shuffle the data every epoch.
- Validate the network once per epoch.
- 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)
```

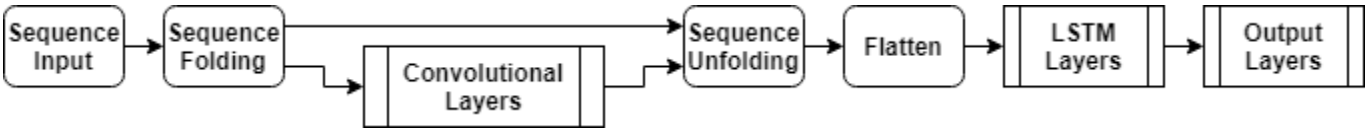
```
accuracy = 0.6647
```

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 folding 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]
```

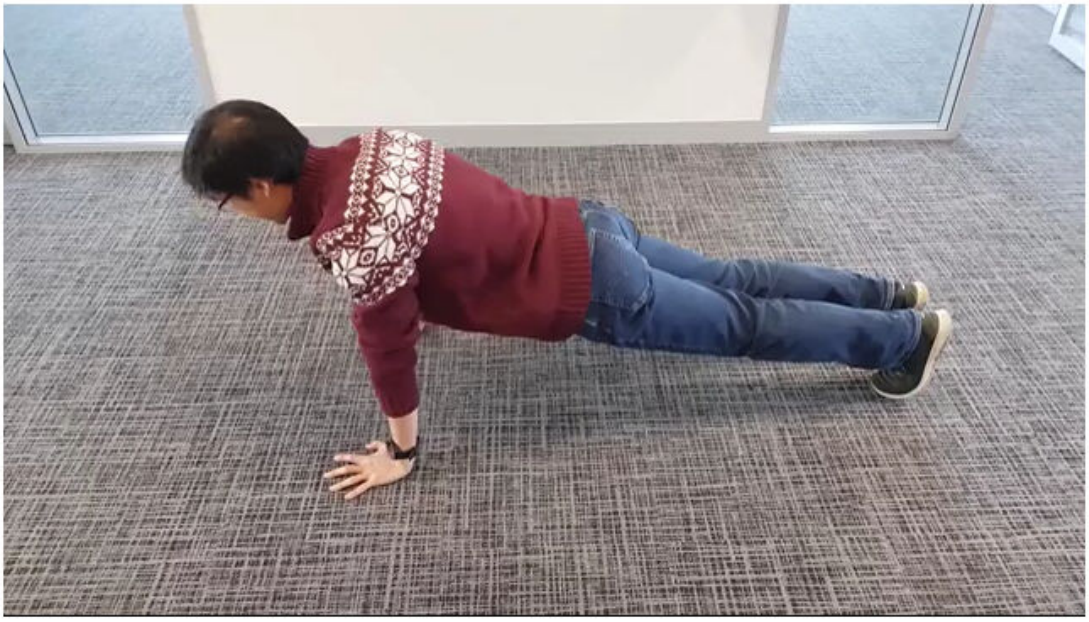
Classify Using New Data

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

`flattenLayer` | `lstmLayer` | `sequenceFoldingLayer` | `sequenceInputLayer` | `sequenceUnfoldingLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “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 can be better parallelism, better control over the receptive field size, better control of the network's memory footprint 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.

This example uses sensor data obtained from a smartphone worn on the body and trains a temporal convolutional network as a function using a custom training loop and automatic differentiation to recognize the activity of the wearer given time series data representing accelerometer readings in three different directions.

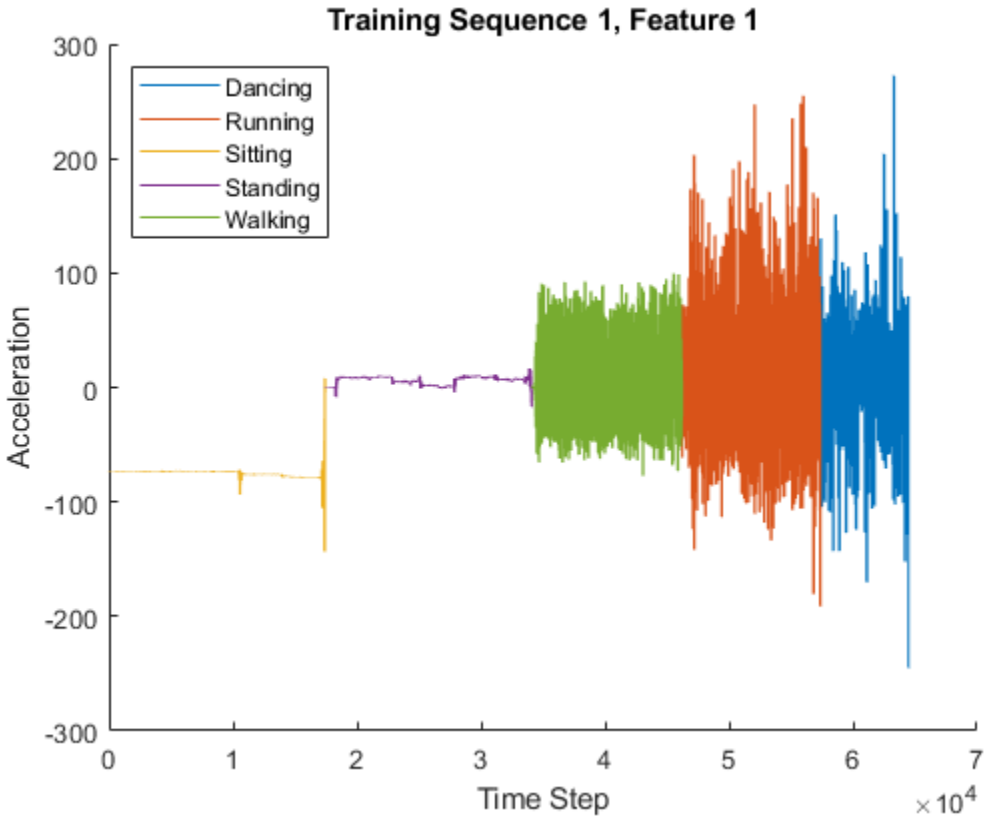
Load Training Data

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 the accelerometer readings in three different directions. Six sequences are used for training and one sequence is used for testing after training.

```
s = load("HumanActivityTrain.mat");
XTrain = s.XTrain;
YTrain = s.YTrain;

numObservations = numel(XTrain);
classes = categories(YTrain{1});
numClasses = numel(classes);
```

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.

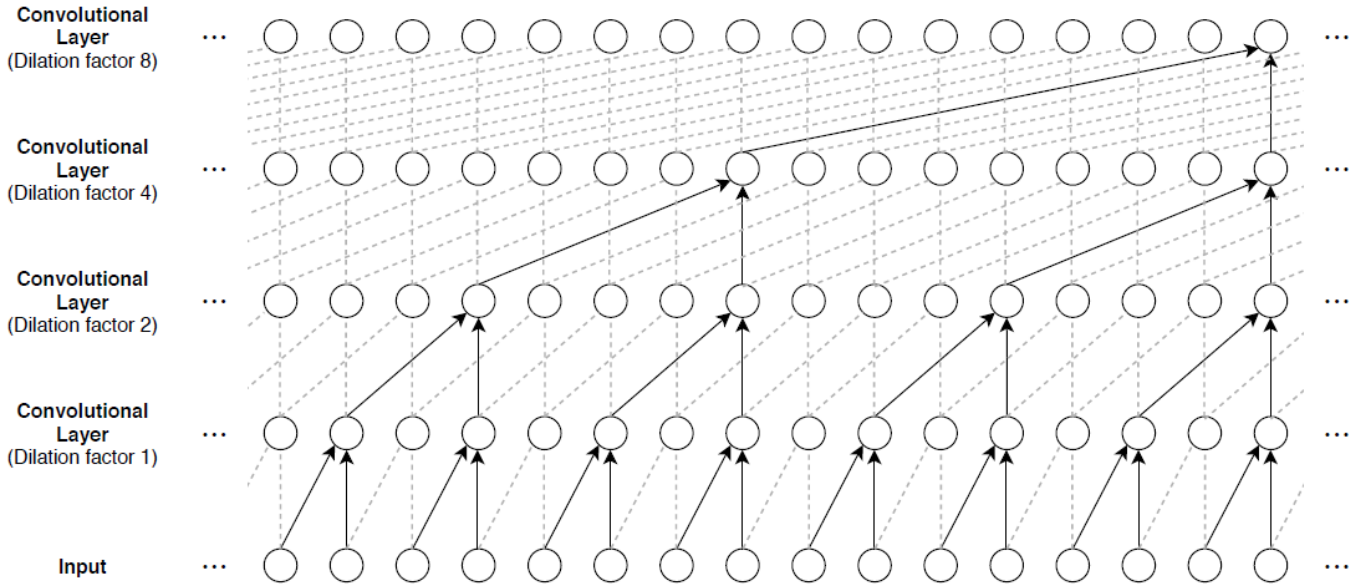
```
X = XTrain{1}(1,:);

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 Deep Learning Model

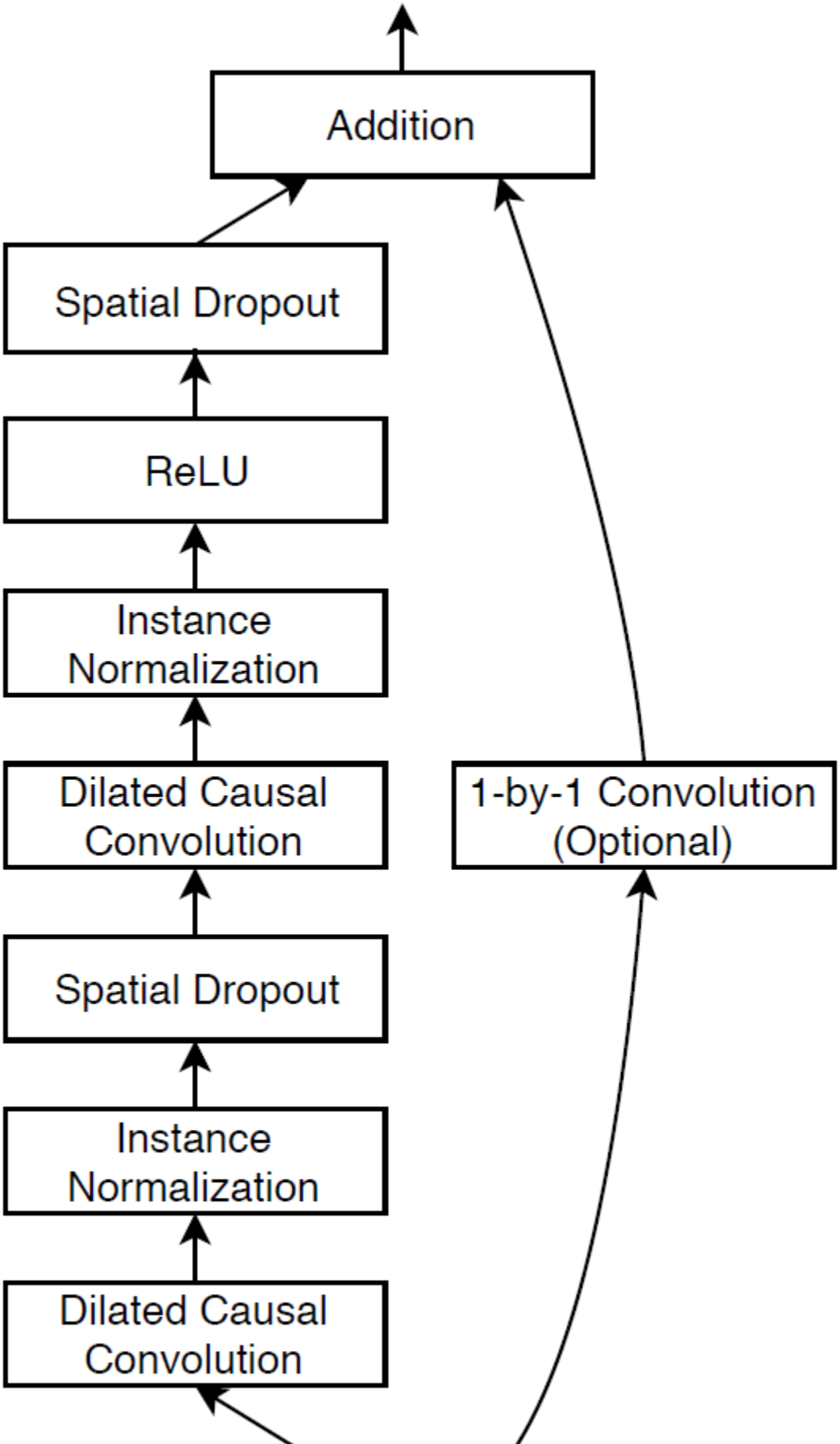
The main building block of a temporal convolutional network is a dilated causal convolution layer which operates over the time steps of each sequence. In this context, "causal" means that the activations computed for a particular time step cannot depend on activations from future time steps.

In order 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 image below. Assuming 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 or learnable parameters as necessary for the data and task at hand.



One of the disadvantages of TCNs compared to RNNs is the higher 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, it can be beneficial to train with the smallest sensible receptive field size R and only perform prediction with the last R time steps of the input sequence.

A 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 input to each block is added 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 a final activation function is applied.


Addition
Spatial Dropout
ReLU
Instance Normalization
Dilated Causal Convolution
Spatial Dropout
Instance Normalization
Dilated Causal Convolution
1-by-1 Convolution (Optional)

Define and Initialize Model Parameters

Specify the parameters for the TCN architecture with four residual blocks containing dilated causal convolution layers with each 175 filters of size 3.

```
numBlocks = 4;
numFilters = 175;
filterSize = 3;
dropoutFactor = 0.05;
```

To pass the model hyperparameters to the model functions (the number of blocks and the dropout factor), create a struct containing these values.

```
hyperparameters = struct;
hyperparameters.NumBlocks = numBlocks;
hyperparameters.DropoutFactor = dropoutFactor;
```

Create a struct containing `dlarray` objects for all the learnable parameters of the model based on the number of input channels and the hyperparameters that define the model architecture. Each residual block requires weights parameters and bias parameters for each of the two convolution operations. The first residual block usually also requires weights and biases for an additional convolution operation with filter size 1. The final fully connected operation requires a weights and bias parameter as well. Initialize the learnable layer weights using the function `initializeGaussian`, listed at the end of the example. Initialize the learnable layer biases with zeros.

```
numInputChannels = 3;

parameters = struct;
numChannels = numInputChannels;

for k = 1:numBlocks
    parametersBlock = struct;
    blockName = "Block"+k;

    weights = initializeGaussian([filterSize, numChannels, numFilters]);
    bias = zeros(numFilters, 1, 'single');
    parametersBlock.Conv1.Weights = dlarray(weights);
    parametersBlock.Conv1.Bias = dlarray(bias);

    weights = initializeGaussian([filterSize, numFilters, numFilters]);
    bias = zeros(numFilters, 1, 'single');
    parametersBlock.Conv2.Weights = dlarray(weights);
    parametersBlock.Conv2.Bias = dlarray(bias);

    % If the input and output of the block have different numbers of
    % channels, then add a convolution with filter size 1.
    if numChannels ~= numFilters
        weights = initializeGaussian([1, numChannels, numFilters]);
        bias = zeros(numFilters, 1, 'single');
        parametersBlock.Conv3.Weights = dlarray(weights);
        parametersBlock.Conv3.Bias = dlarray(bias);
    end
    numChannels = numFilters;

    parameters.(blockName) = parametersBlock;
```

```
end

weights = initializeGaussian([numClasses,numChannels]);
bias = zeros(numClasses,1,'single');

parameters.FC.Weights = dlarray(weights);
parameters.FC.Bias = dlarray(bias);
```

View the network parameters.

```
parameters
```

```
parameters = struct with fields:
    Block1: [1×1 struct]
    Block2: [1×1 struct]
    Block3: [1×1 struct]
    Block4: [1×1 struct]
        FC: [1×1 struct]
```

View the parameters of the first block.

```
parameters.Block1
```

```
ans = struct with fields:
    Conv1: [1×1 struct]
    Conv2: [1×1 struct]
    Conv3: [1×1 struct]
```

View the parameters of the first convolutional operation of the first block.

```
parameters.Block1.Conv1
```

```
ans = struct with fields:
    Weights: [3×3×175 dlarray]
       Bias: [175×1 dlarray]
```

Define Model and Model Gradients Functions

Create the function `model`, listed in the Model Function on page 4-0 section at the end of the example, that computes the outputs of the deep learning model. The function `model` takes the input data, the learnable model parameters, the model hyperparameters, and a flag that specifies whether the model should return outputs for training or prediction. The network outputs the predictions for the labels at each time step of the input sequence.

Create the function `modelGradients`, listed in the Model Gradients Function on page 4-0 section at the end of the example, that takes a mini-batch of input data, the corresponding target sequences, and the parameters of the network, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss.

Specify Training Options

Specify a set of training options used in the custom training loop.

- Train for 30 epochs with mini-batch size 1.

- Start with an initial learn rate of 0.001
- Multiply the learn rate by 0.1 every 12 epochs.
- Clip the gradients using the L_2 norm with threshold 1.

```
maxEpochs = 30;
miniBatchSize = 1;
initialLearnRate = 0.001;
learnRateDropFactor = 0.1;
learnRateDropPeriod = 12;
gradientThreshold = 1;
```

To train on a GPU if one is available, specify the execution environment `"auto"`. To explicitly train on the CPU or GPU, specify `"cpu"` or `"gpu"` respectively. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
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 do not want to plot the training progress, then set this value to `"none"`.

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

Train Model

Train the network via stochastic gradient descent by looping over the sequences in the training dataset, computing parameter gradients, and updating the network parameters via the Adam update rule. This process is repeated multiple times (referred to as *epochs*) until training converges and the maximum number of epochs is reached.

For each epoch, shuffle the training data.

For each mini-batch:

- Convert the labels to dummy variables.
- Preprocess the sequence data using the `transformSequences` function, listed at the end of the example.
- Convert the data to `dlarray` objects with underlying type single
- For GPU training, convert the data to `gpuArray`.
- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Clip the gradients if they are too large, using the function `thresholdL2Norm`, listed at the end of the example, and the `dlupdate` function.
- Update the network parameters using the `adamupdate` function.
- Update the training progress plot.

After completing `learnRateDropPeriod` epochs, reduce the learn rate by multiplying the current learning rate by `learnRateDropFactor`.

Initialize the learning rate which will be multiplied by the `LearnRateDropFactor` value every `LearnRateDropPeriod` epochs.

```
learnRate = initialLearnRate;
```

Initialize the moving average of the parameter gradients and the element-wise squares of the gradients used by the Adam optimizer.

```
trailingAvg = [];
trailingAvgSq = [];
```

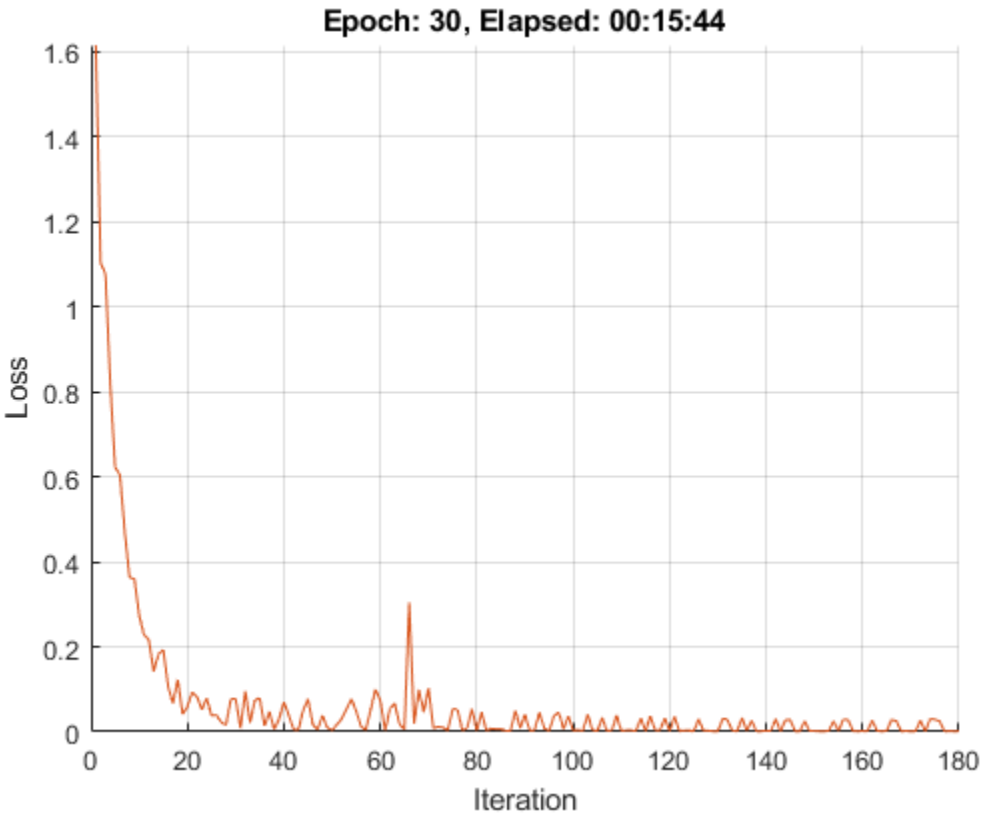
Initialize a plot showing the training progress.

```
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;
numIterationsPerEpoch = floor(numObservations./miniBatchSize);

start = tic;

% Loop over epochs.
for epoch = 1:maxEpochs

    % Shuffle the data.
    idx = randperm(numObservations);
    XTrain = XTrain(idx);
    YTrain = YTrain(idx);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and apply the transformSequences
        % preprocessing function.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        [X,Y,numTimeSteps] = transformSequences(XTrain(idx),YTrain(idx));

        % Convert to dlarray.
        dlX = dlarray(X);

        % If training on a GPU, convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            dlX = gpuArray(dlX);
        end

        % Evaluate the model gradients and loss using dlfeval.
        [gradients, loss] = dlfeval(@modelGradients,dlX,Y,parameters,hyperparameters,numTimeSteps

        % Clip the gradients.
        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);
```

```
        if plots == "training-progress"
            % Plot training progress.
            D = duration(0,0,toc(start),'Format','hh:mm:ss');

            % Normalize the loss over the sequence lengths
            loss = mean(loss ./ numTimeSteps);
            loss = double(gather(extractdata(loss)));
            loss = mean(loss);

            addpoints(lineLossTrain,iteration, mean(loss));

            title("Epoch: " + epoch + ", Elapsed: " + string(D))
            drawnow
        end
    end

    % Reduce the learning rate after learnRateDropPeriod epochs
    if mod(epoch,learnRateDropPeriod) == 0
        learnRate = learnRate*learnRateDropFactor;
    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 for each time step.

```
s = load("HumanActivityTest.mat");
XTest = s.XTest;
YTest = s.YTest;
numObservationsTest = numel(XTest);
```

Preprocess the test data using the same function used for training and convert the data to `dlarray`.

```
[X,Y] = transformSequences(XTest,YTest);
dlXTest = dlarray(X);
```

To predict the labels of the test data, use the model function with the trained parameters, the hyperparameters, and the `doTraining` option set to false.

```
doTraining = false;
dlYPred = model(dlXTest,parameters,hyperparameters,doTraining);
```

To get the categorical labels of the predictions, find the class label with the highest score for each time step. Calculate the accuracy by evaluating the mean classification accuracy for the sequences.

```
YPred = gather(extractdata(dlYPred));

labelsPred = categorical(zeros(numObservationsTest,size(dlYPred,3)));
accuracy = zeros(1,numObservationsTest);

for i = 1:numObservationsTest
    [~,idxPred] = max(YPred(:,i,:),[],1);
    [~,idxTest] = max(Y(:,i,:),[],1);

    labelsPred(i,:) = classes(idxPred)';
    accuracy(i) = mean(idxPred == idxTest);
end
```

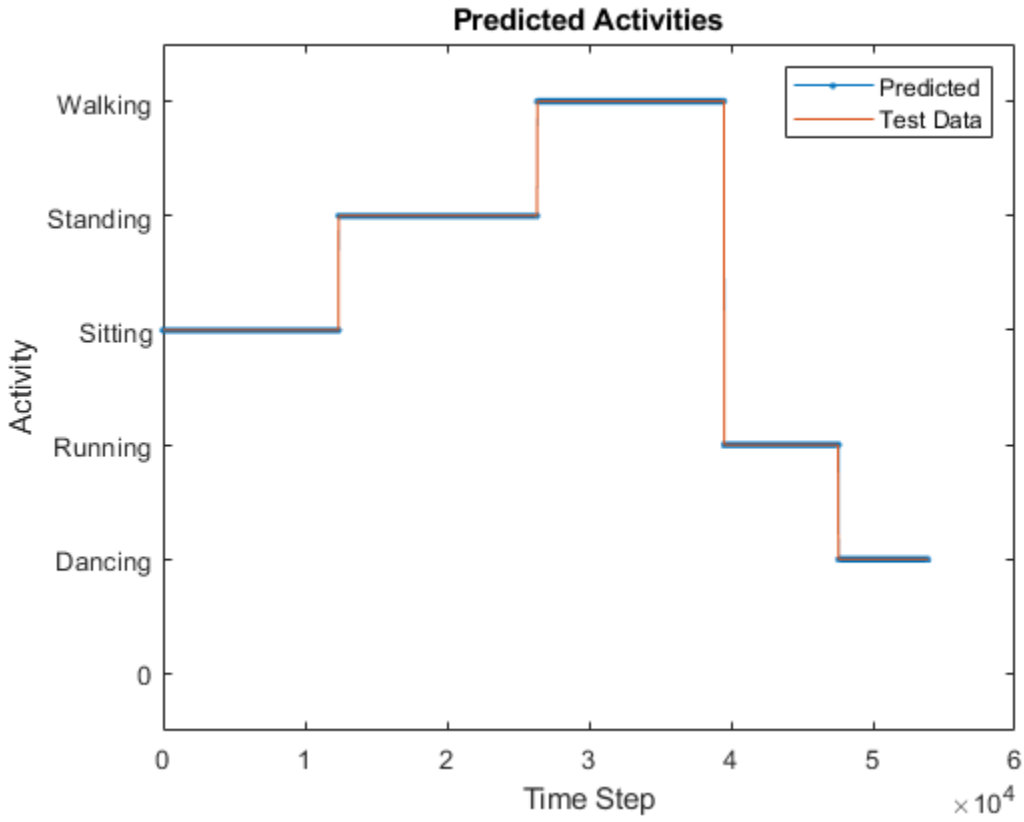
View the mean accuracy over the test set.

```
mean(accuracy)
```

```
ans = 0.9997
```

Compare the predictions for a single sequence with the corresponding test data by using a plot.

```
figure
idx = 1;
plot(categorical(labelsPred(idx,:)),'.-')
hold on
plot(YTest{1})
hold off

xlabel("Time Step")
ylabel("Activity")
title("Predicted Activities")
legend(["Predicted" "Test Data"])
```



Model Function

The function `model` takes the input data `dlX`, the learnable model parameters, the model hyperparameters, and the flag `doTraining` which specifies whether the model should return outputs for training or prediction. The network outputs the predictions for the labels at each time step of the input sequence. The model consists of multiple residual blocks with exponentially increasing dilation factors. After the last residual block, a final `fullyconnect` operation maps the output to the number of classes in the target data.

```
function dlY = model(dlX,parameters,hyperparameters,doTraining)

numBlocks = hyperparameters.NumBlocks;
dropoutFactor = hyperparameters.DropoutFactor;

dlY = dlX;

% Residual blocks.
for k = 1:numBlocks
    dilationFactor = 2^(k-1);
    parametersBlock = parameters.("Block"+k);

    dlY = residualBlock(dlY,dilationFactor,dropoutFactor,parametersBlock,doTraining);
end

% Fully connect.
weights = parameters.FC.Weights;
bias = parameters.FC.Bias;
```

```
dlY = fullyconnect(dlY,weights,bias,'DataFormat','CBT');

end
```

Residual Block Function

The function `residualBlock` implements the core building block of the temporal convolutional network.

To apply 1-D causal dilated convolution, use the `dlconv` function:

- To convolve over the spatial dimensions, set the `'DataFormat'` option to `'CBS'` (use the dimension label `'S'` instead of `'T'`),
- Set the `'DilationFactor'` option according to the dilation factor of the residual block.
- To ensure only the past time steps are used, apply padding only at the beginning of the sequence.

```
function dlY = residualBlock(dlX,dilationFactor,dropoutFactor,parametersBlock,doTraining)

% Convolution options.
filterSize = size(parametersBlock.Conv1.Weights,1);
paddingSize = (filterSize - 1) * dilationFactor;

% Convolution.
weights = parametersBlock.Conv1.Weights;
bias =  parametersBlock.Conv1.Bias;
dlY = dlconv(dlX,weights,bias, ...
    'DataFormat','CBS', ...
    'Stride', 1, ...
    'DilationFactor', dilationFactor, ...
    'Padding', [paddingSize; 0] );

% Instance normalization, ReLU, spatial dropout.
dlY = instanceNormalization(dlY,'CBS');
dlY = relu(dlY);
dlY = spatialDropout(dlY,dropoutFactor,'CBS',doTraining);

% Convolution.
weights = parametersBlock.Conv2.Weights;
bias = parametersBlock.Conv2.Bias;
dlY = dlconv(dlY,weights,bias, ...
    'DataFormat','CBS', ...
    'Stride', 1, ...
    'DilationFactor', dilationFactor, ...
    'Padding',[paddingSize; 0] );

% Instance normalization, ReLU, spatial dropout.
dlY = instanceNormalization(dlY,'CBS');
dlY = relu(dlY);
dlY = spatialDropout(dlY,dropoutFactor,'CBS',doTraining);

% Optional 1-by-1 convolution.
if ~isequal(size(dlX),size(dlY))
    weights = parametersBlock.Conv3.Weights;
    bias = parametersBlock.Conv3.Bias;
    dlX = dlconv(dlX,weights,bias,'DataFormat','CBS');
end
```

```
% Addition and ReLU
dlY = relu(dlX+dlY);

end
```

Model Gradients Function

The `modelGradients` function takes a mini-batch of input data `dlX`, the corresponding target sequences `T`, the learnable parameters, and the hyperparameters, and returns the gradients of the loss with respect to the learnable parameters and the corresponding loss. To compute the gradients, evaluate the `modelGradients` function using the `dlfeval` function in the training loop.

```
function [gradients,loss] = modelGradients(dlX,T,parameters,hyperparameters,numTimeSteps)

dlY = model(dlX,parameters,hyperparameters,true);
dlY = softmax(dlY,'DataFormat','CBT');

dlT = dlarray(T,'CBT');
loss = maskedCrossEntropyLoss(dlY, dlT, numTimeSteps);

gradients = dlgradient(mean(loss),parameters);

end
```

Masked Cross-Entropy Loss Function

The `maskedCrossEntropyLoss` function computes the cross-entropy loss for mini-batches of sequences, where the sequences are different lengths.

```
function loss = maskedCrossEntropyLoss(dlY, dlT, numTimeSteps)

numObservations = size(dlY,2);
loss = dlarray(zeros(1,numObservations,'like',dlY));

for i = 1:numObservations
    idx = 1:numTimeSteps(i);
    loss(i) = crossentropy(dlY(:,i,idx),dlT(:,i,idx),'DataFormat','CBT');
end

end
```

Instance Normalization Function

The `instanceNormalization` function normalizes the input `dlX` by first calculating the mean μ and the variance σ^2 for each observation over each input channel. Then it calculates the normalized activations as

$$\widehat{X} = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}}.$$

In comparison to batch normalization, the mean and variance is different for each observation in the mini-batch. Use normalization, such as instance normalization, between convolutional layers and nonlinearities to speed up training of convolutional neural networks and improve convergence.

```
function dlY = instanceNormalization(dlX,fmt)

reductionDims = find(fmt == 'S');
```

```
mu = mean(dlX,reductionDims);
sigmaSq = var(dlX,1,reductionDims);

epsilon = 1e-5;
dlY = (dlX-mu) ./ sqrt(sigmaSq+epsilon);

end
```

Spatial Dropout Function

The `spatialDropout` function performs spatial dropout [3] on the input `dlX` with dimension labels `fmt` when the `doTraining` flag is `true`. Spatial dropout drops an entire channel of the input data. That is, all time steps of a certain channel are dropped with the probability specified by `dropoutFactor`. The channels are dropped out independently in the batch dimension.

```
function dlY = spatialDropout(dlX,dropoutFactor,fmt,doTraining)

if doTraining
    maskSize = size(dlX);
    maskSize(fmt=='S') = 1;

    dropoutScaleFactor = single(1 - dropoutFactor);
    dropoutMask = (rand(maskSize,'like',dlX) > dropoutFactor) / dropoutScaleFactor;

    dlY = dlX .* dropoutMask;
else
    dlY = dlX;
end

end
```

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
```

Sequence Transform Function

The `sequenceTransform` function takes a cell array of N sequences and returns a C-by-N-by-S numeric array of left-padded 1-D sequences and the number of time steps in each sequence, where C corresponds to the number of features of the sequences and S corresponds to the number of time steps of the longest sequence.

```
function [XTransformed, YTransformed, numTimeSteps] = transformSequences(X,Y)

numTimeSteps = cellfun(@(sequence) size(sequence,2),X);

miniBatchSize = numel(X);
numFeatures = size(X{1},1);
sequenceLength = max(cellfun(@(sequence) size(sequence,2),X));
classes = categories(Y{1});
numClasses = numel(classes);
```

```
sz = [numFeatures miniBatchSize sequenceLength];
XTransformed = zeros(sz,'single');

sz = [numClasses miniBatchSize sequenceLength];
YTransformed = zeros(sz,'single');

for i = 1:miniBatchSize
    predictors = X{i};

    % Create dummy labels.
    responses = zeros(numClasses, numTimeSteps(i), 'single');
    for c = 1:numClasses
        responses(c,Y{i}==classes(c)) = 1;
    end

    % Left pad.
    XTransformed(:,i,:) = leftPad(predictors,sequenceLength);
    YTransformed(:,i,:) = leftPad(responses,sequenceLength);
end

end
```

Left Padding Function

The `leftPad` function takes a sequence and left-pads it with zeros to have the specified sequence length.

```
function sequencePadded = leftPad(sequence,sequenceLength)

[numFeatures,numTimeSteps] = size(sequence);

paddingSize = sequenceLength - numTimeSteps;
padding = zeros(numFeatures,paddingSize);

sequencePadded = [padding sequence];

end
```

Gradient Clipping Function

The `thresholdL2Norm` function scales the gradient `g` so that its L_2 norm equals `gradientThreshold` when the L_2 norm of the gradient is larger than `gradientThreshold`.

```
function g = thresholdL2Norm(g,gradientThreshold)

gradientNorm = sqrt(sum(g.^2,'all'));
if gradientNorm > gradientThreshold
    g = g * (gradientThreshold / gradientNorm);
end

end
```

References

[1] Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling." *arXiv preprint arXiv:1803.01271* (2018).

[2] Van Den Oord, Aäron, et al. "WaveNet: A generative model for raw audio." *SSW* 125 (2016).

[3] Tompson, Jonathan, et al. "Efficient object localization using convolutional networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015.

See Also

`adamupdate` | `crossentropy` | `dlarray` | `dlconv` | `dlfeval` | `dlgradient` | `fullyconnect` | `relu` | `softmax`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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)
```

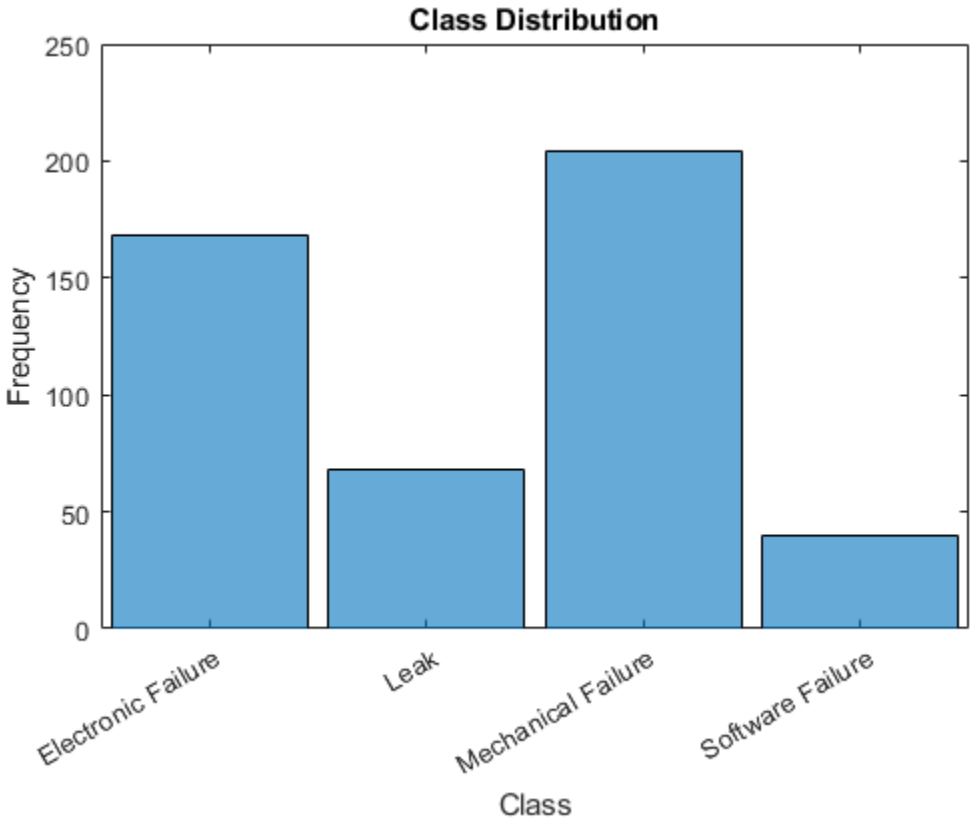
```
ans=8×5 table
                                Description                                       Category
    ____________________________________________________________________________    ____________________

    "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.

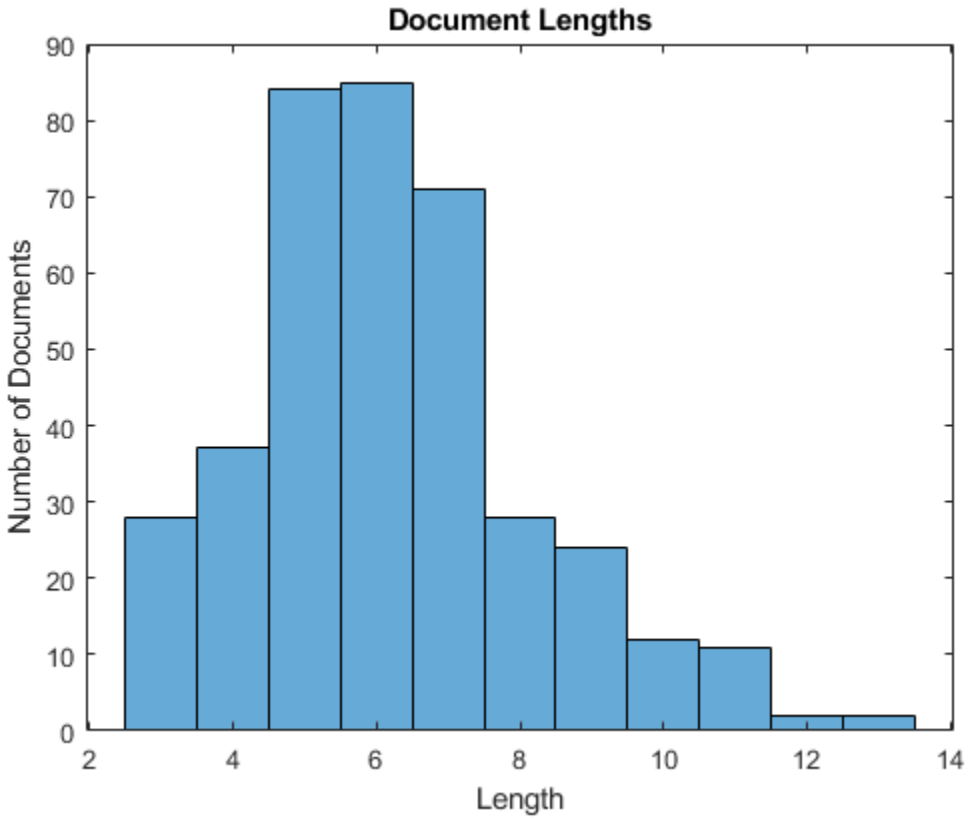
To create a word encoding, use the `wordEncoding` function.

```
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:

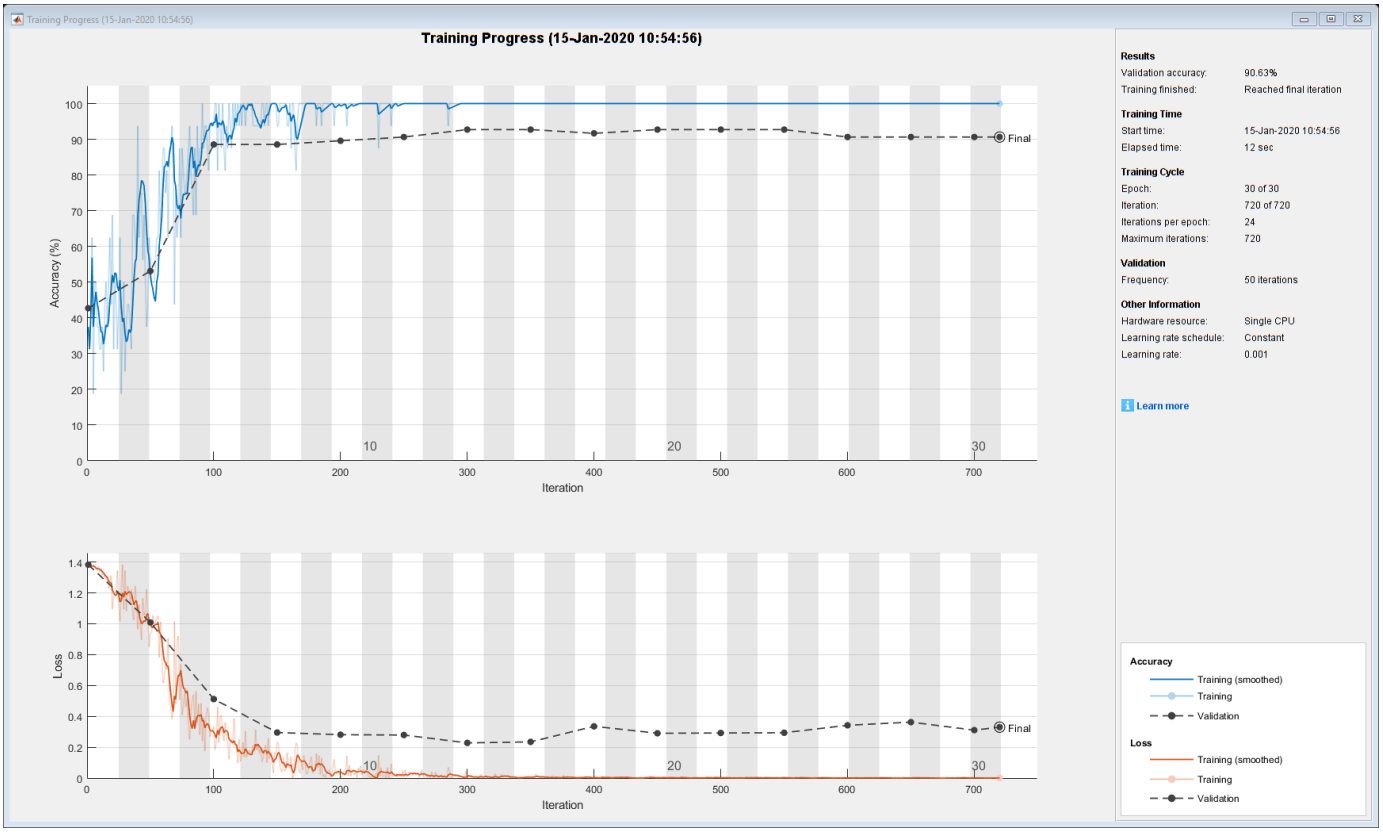
- Train using the Adam solver.
- Specify a mini-batch size of 16.
- Shuffle the data every epoch.
- Monitor the training progress by setting the `'Plots'` option to `'training-progress'`.
- Specify the validation data using the `'ValidationData'` option.
- Suppress verbose output by setting the `'Verbose'` option to `false`.

By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). 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.

```
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

`doc2sequence` | `fastTextWordEmbedding` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” on page 4-131

- “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'`.

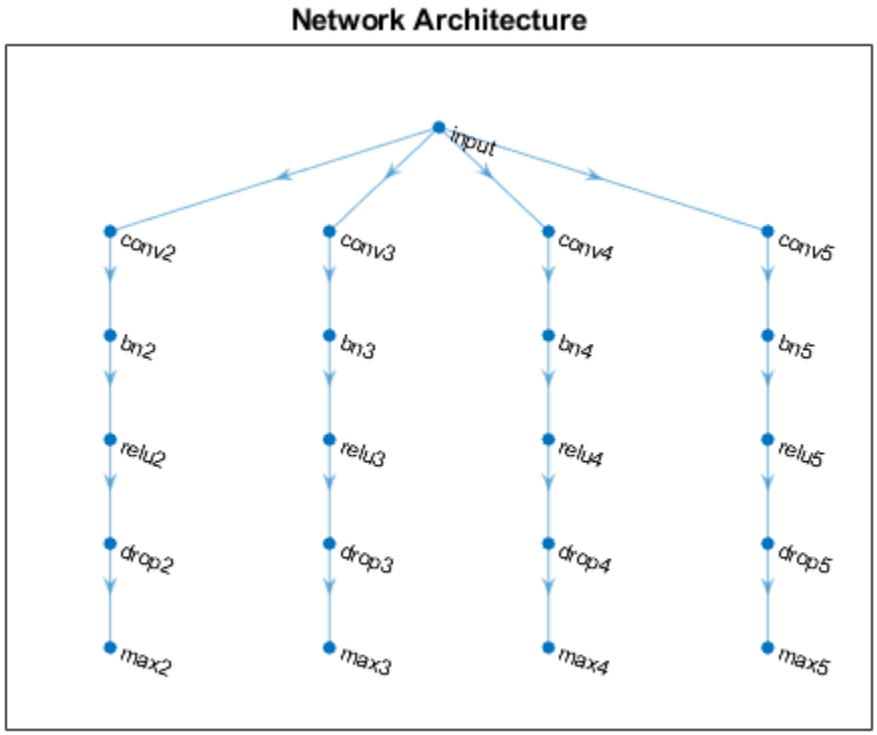
```
layer = imageInputLayer(inputSize,'Normalization','none','Name','input');
lgraph = layerGraph(layer);
```

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
```

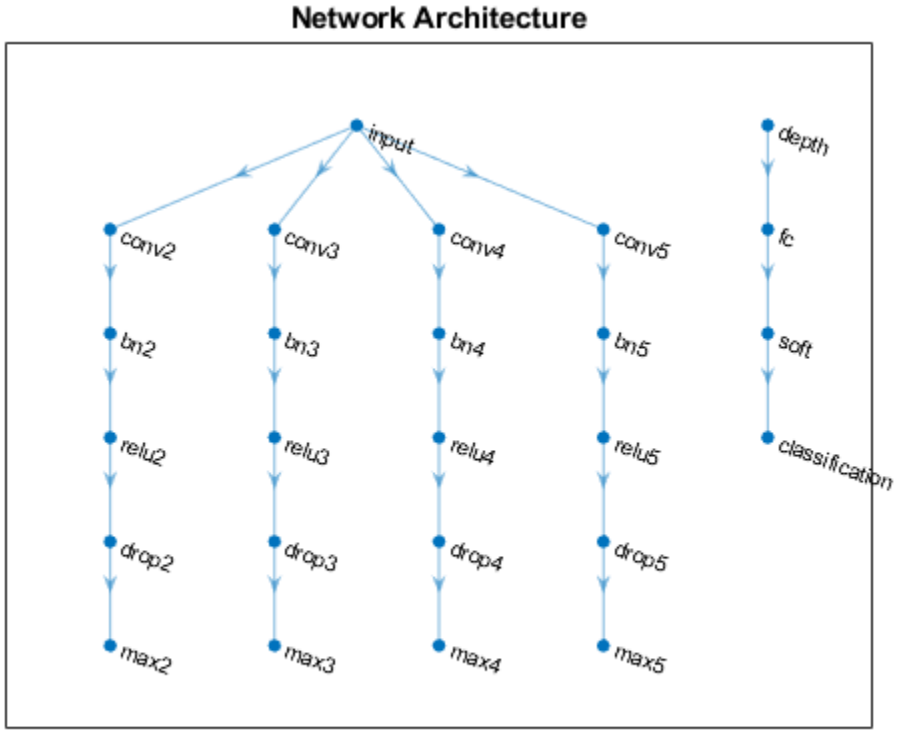
View the network architecture in a plot.

```
figure
plot(lgraph)
title("Network Architecture")
```



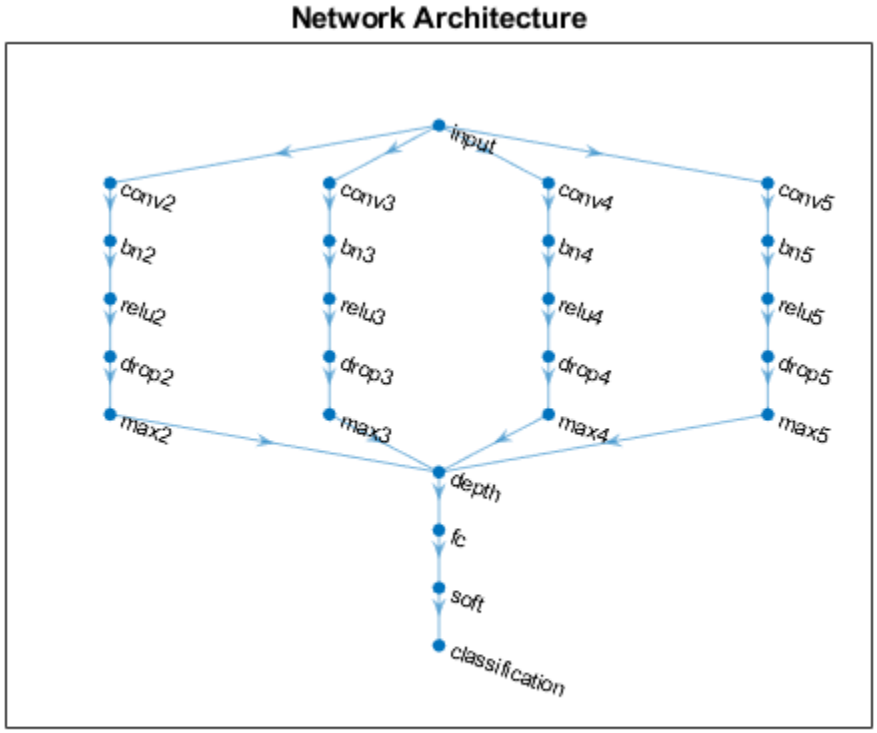
Add the depth concatenation layer, the fully connected layer, the softmax layer, and the classification layer.

```
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

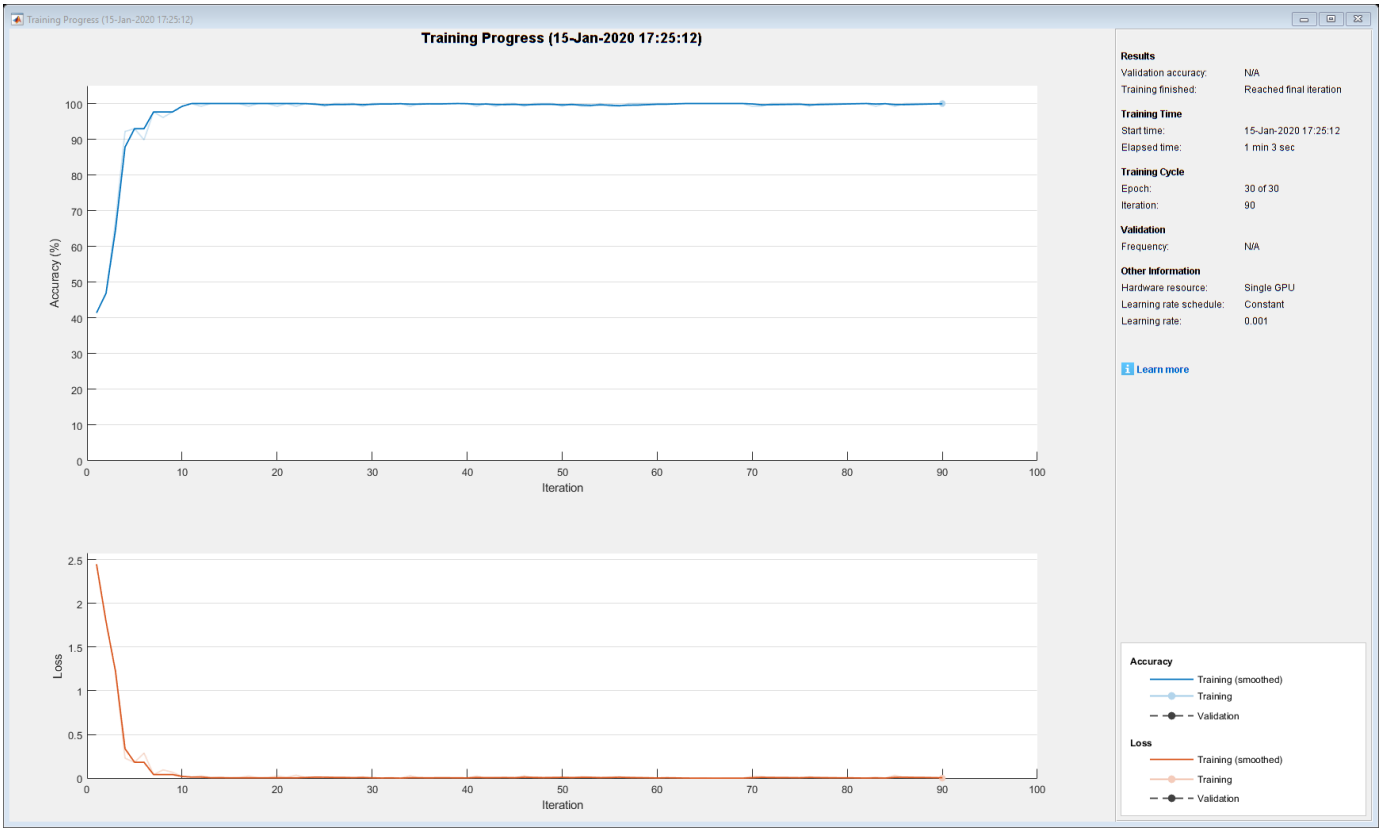
Specify the training options:

- 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

`batchNormalizationLayer` | `convolution2dLayer` | `doc2sequence` | `fastTextWordEmbedding` | `layerGraph` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `transform` | `wordEmbedding` | `wordcloud`

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 16-2
- “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.

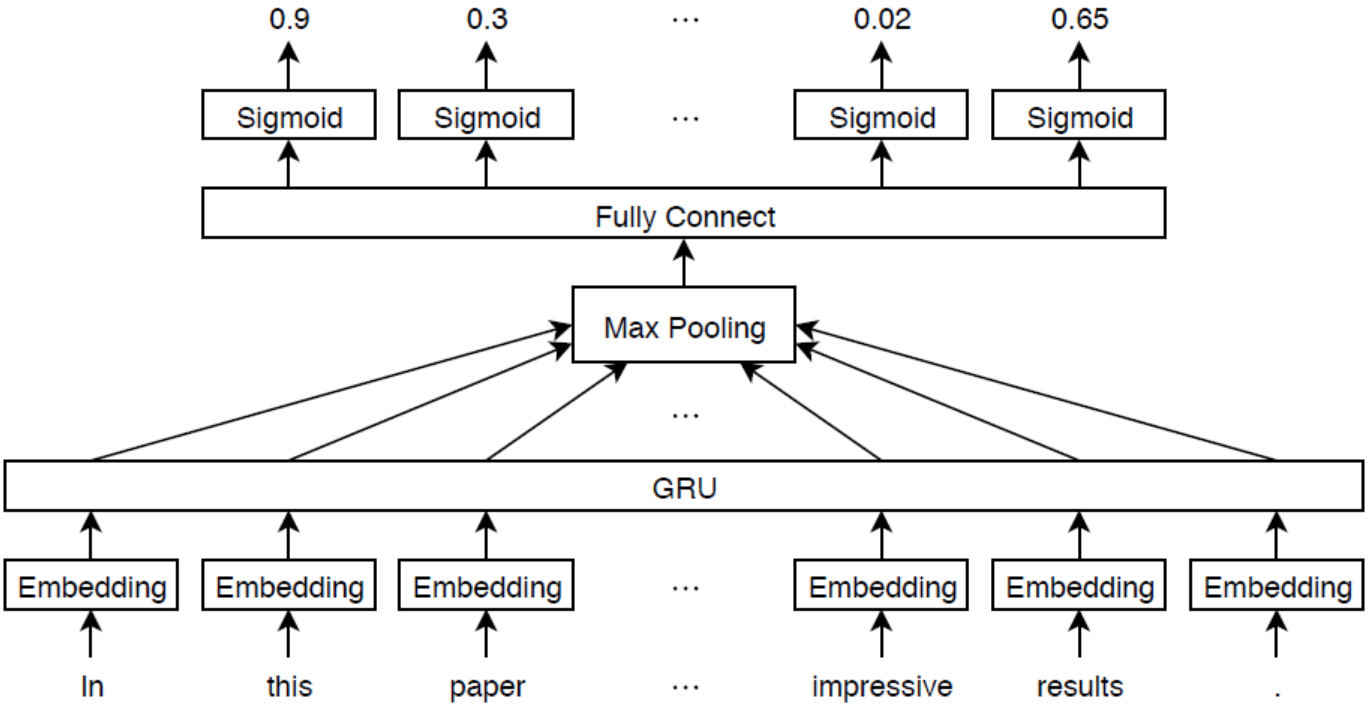
This example defines a deep learning model that classifies subject areas given the abstracts of mathematical papers collected using the arXiv API [1]. The model consists of a word embedding and GRU, max pooling operation, fully connected, and sigmoid operations.

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. Note that the arXiv API is rate limited to querying 1000 articles at a time and requires waiting between requests.

```
importSize = 50000;
```

Import the first set of records.

```
url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
    "&set=math" + ...
    "&metadataPrefix=arXiv";
options = weboptions('Timeout',160);
code = webread(url,options);
```

Parse the returned XML content and create an array of `htmlTree` objects containing the record information.

```
tree = htmlTree(code);
subtrees = findElement(tree,"record");
numel(subtrees)
```

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 `resumptionToken` attribute from the previous result. 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(subtrees) < importSize
    subtreeResumption = findElement(tree,"resumptionToken");
```

```
    if isempty(subtreeResumption)
        break
    end

    resumptionToken = extractHTMLText(subtreeResumption);

    url = "https://export.arxiv.org/oai2?verb=ListRecords" + ...
        "&resumptionToken=" + resumptionToken;

    pause(20)
    code = webread(url,options);

    tree = htmlTree(code);

    subtrees = [subtrees; findElement(tree,"record")];
end
```

Extract and Preprocess Text Data

Extract the abstracts and labels from the parsed HTML trees.

Find the `"<abstract>"` and `"<categories>"` elements using the `findElement` function.

```
subtreeAbstract = htmlTree("");
subtreeCategory = htmlTree("");

for i = 1:numel(subtrees)
    subtreeAbstract(i) = findElement(subtrees(i),"abstract");
    subtreeCategory(i) = findElement(subtrees(i),"categories");
end
```

Extract the text data from the subtrees containing the abstracts using the `extractHTMLText` function.

```
textData = extractHTMLText(subtreeAbstract);
```

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" "$\alpha\ge 0$" "dyadic" "gri
    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
```

Extract the labels from the subtrees containing the labels.

```
strLabels = extractHTMLText(subtreeCategory);
labelsAll = arrayfun(@split,strLabels,'UniformOutput',false);
```

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:

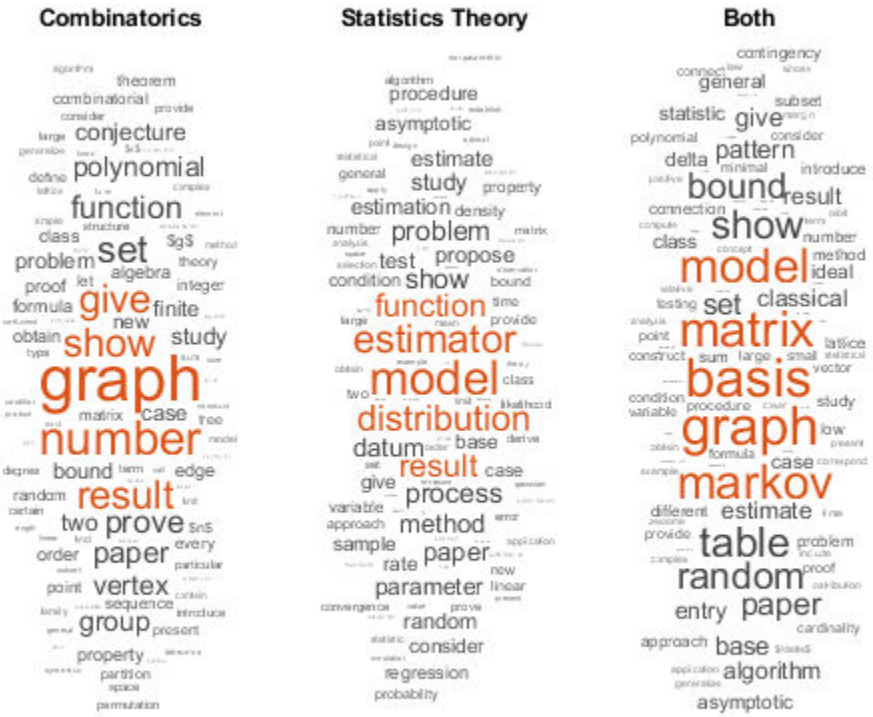
- Abstracts tagged with "Combinatorics" and not tagged with `"Statistics Theory"`
- Abstracts tagged with "Statistics Theory" and not tagged with `"Combinatorics"`
- Abstracts tagged with both `"Combinatorics"` and `"Statistics Theory"`

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")
```



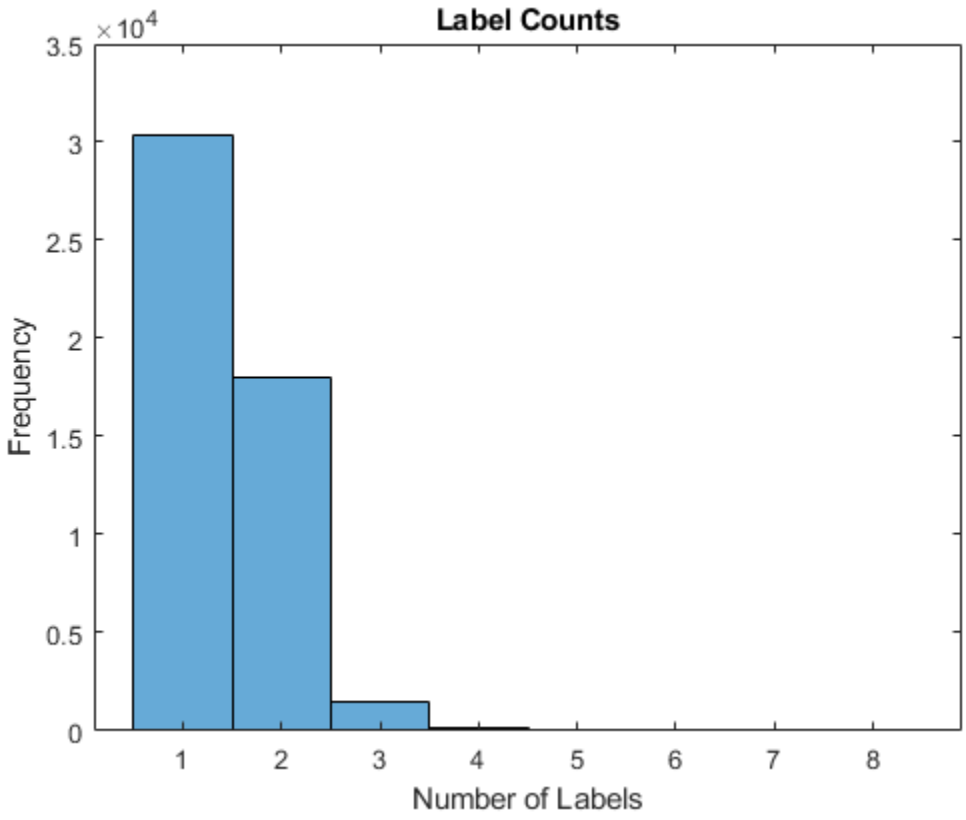
View the number of classes.

```
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)
```

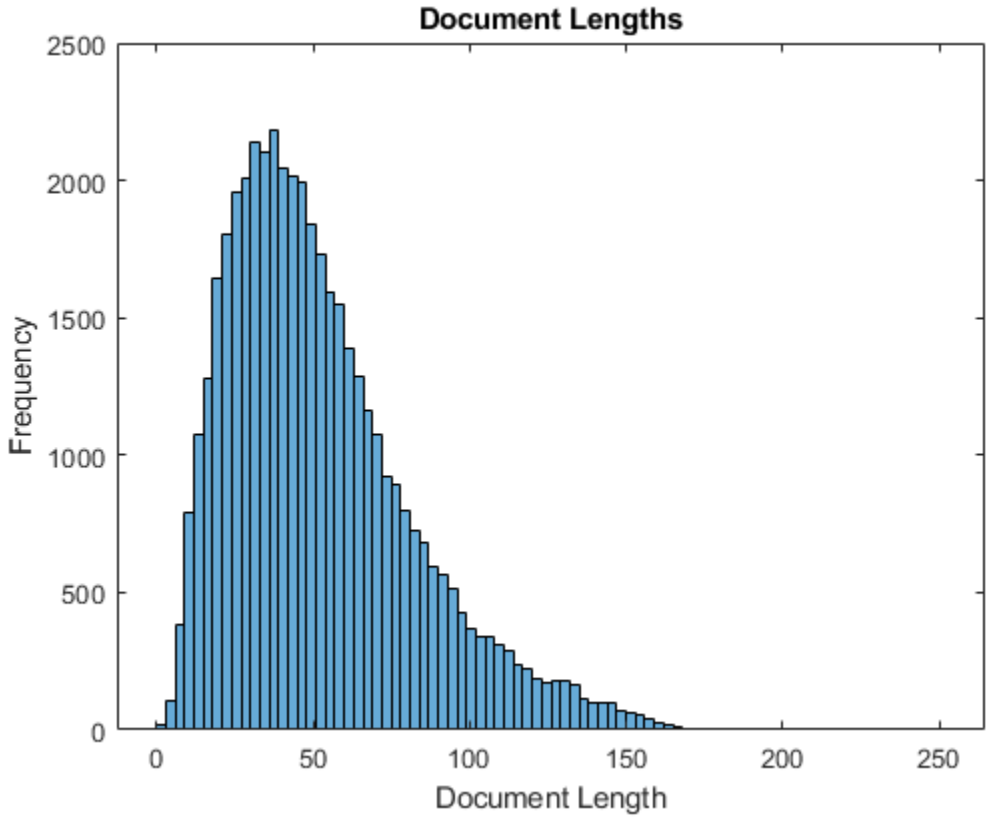
```
enc =
  wordEncoding with properties:

      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 CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
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 `'BCT'` (batch, channel, time).

- 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.

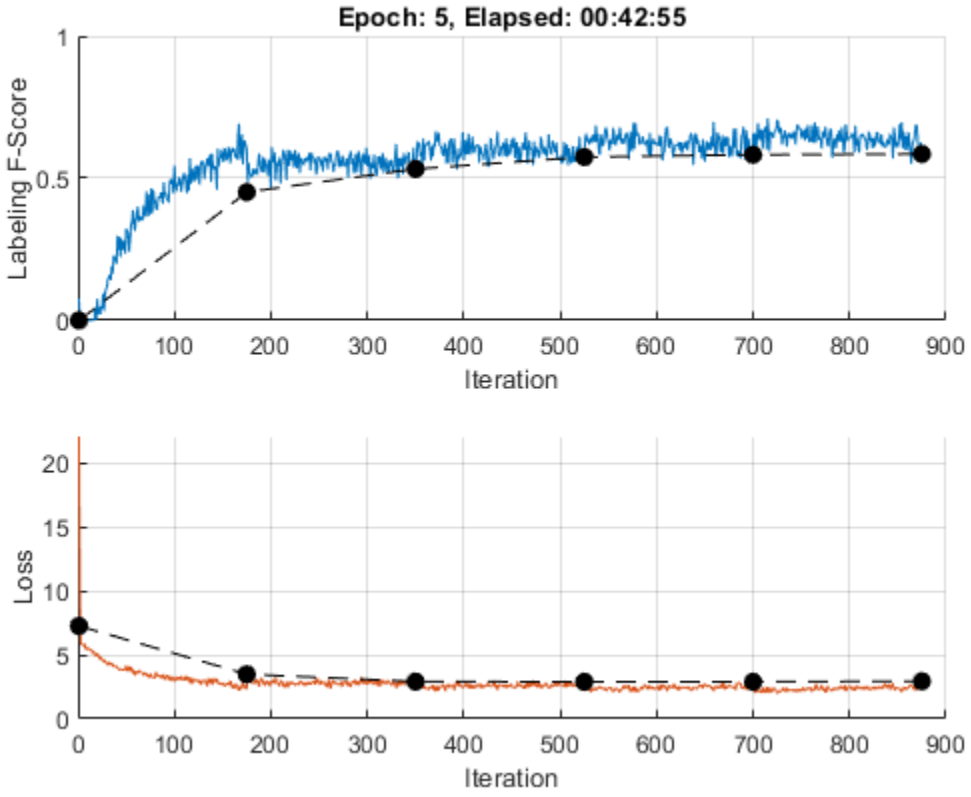
```
numObservationsValidation = numel(documentsValidation);
TValidation = zeros(numClasses, numObservationsValidation, 'single');
for i = 1:numObservationsValidation
    [~,idx] = ismember(labelsValidation{i},classNames);
    TValidation(idx,i) = 1;
end
```

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.

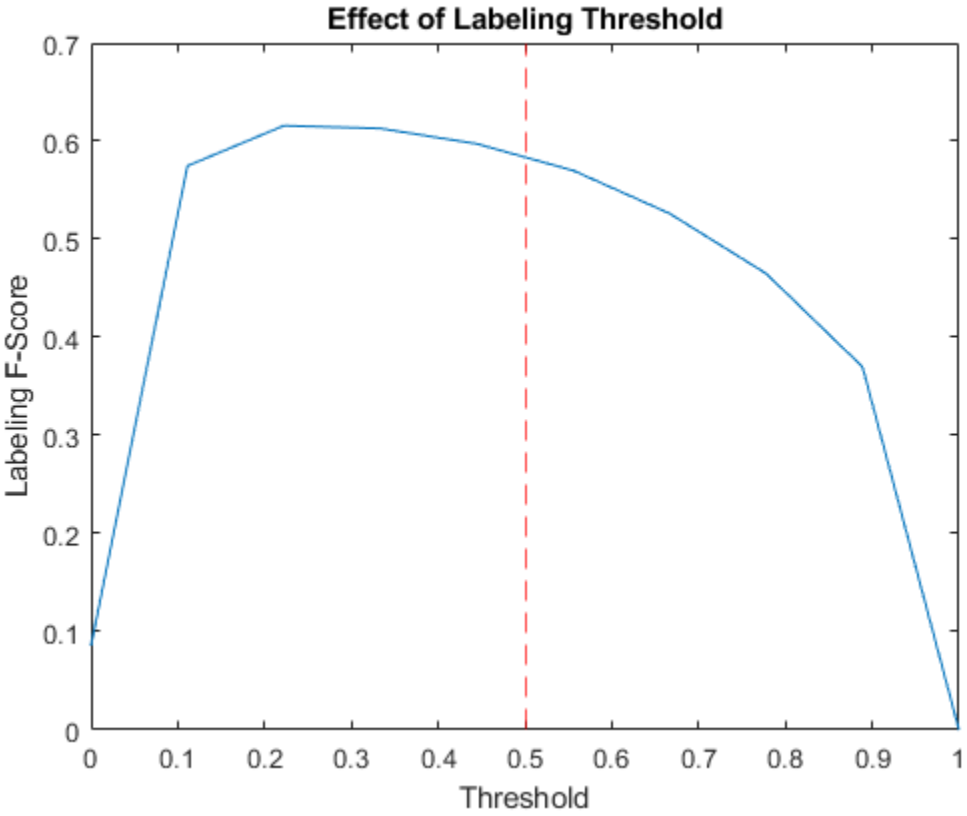
```
score = labelingFScore(YPredValidation,TValidation)
```

```
score = single
    0.5852
```

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")
```



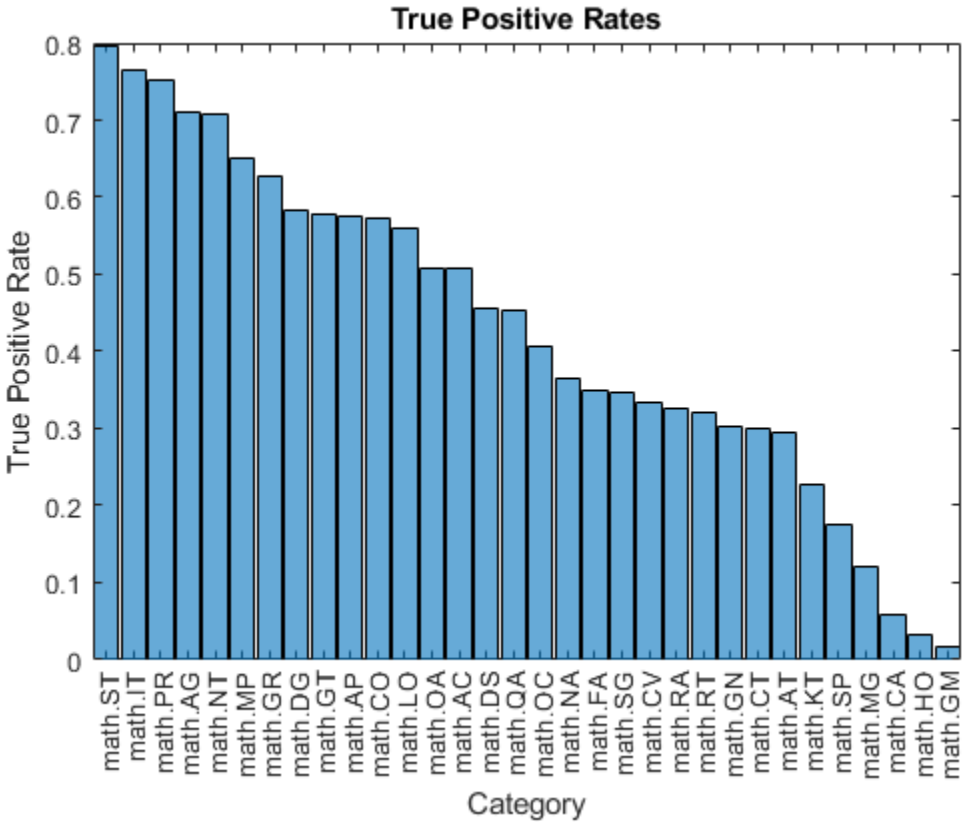
Visualize Predictions

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:

$$\mathrm{TPFN}_{ij} = \begin{cases} \mathrm{numTruePositives}(i), & \text{if } i = j \\ \mathrm{numFalsePositives}(j \,|\, i \text{ is a false negative}), & \text{if } i \neq j \end{cases}$$

Calculate the false negatives and false positives.

```
falseNegatives = T & ~Y;
falsePositives = ~T & Y;
```

Calculate the off-diagonal elements.

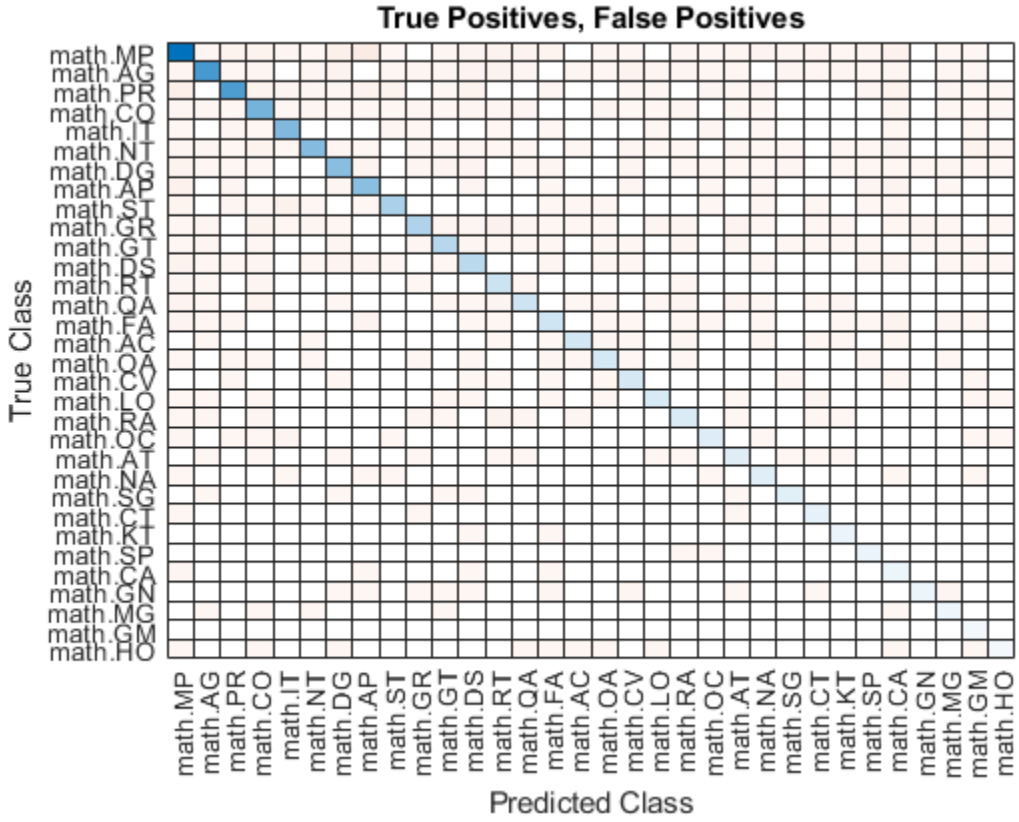
```
falseNegatives = permute(falseNegatives,[3 2 1]);
numConditionalFalsePositives = sum(falseNegatives & falsePositives, 2);
numConditionalFalsePositives = squeeze(numConditionalFalsePositives);

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

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlupdate` | `doc2sequence` | `extractHTMLText` | `fullyconnect` | `gru` | `htmlTree` | `tokenizedDocument` | `wordEncoding`

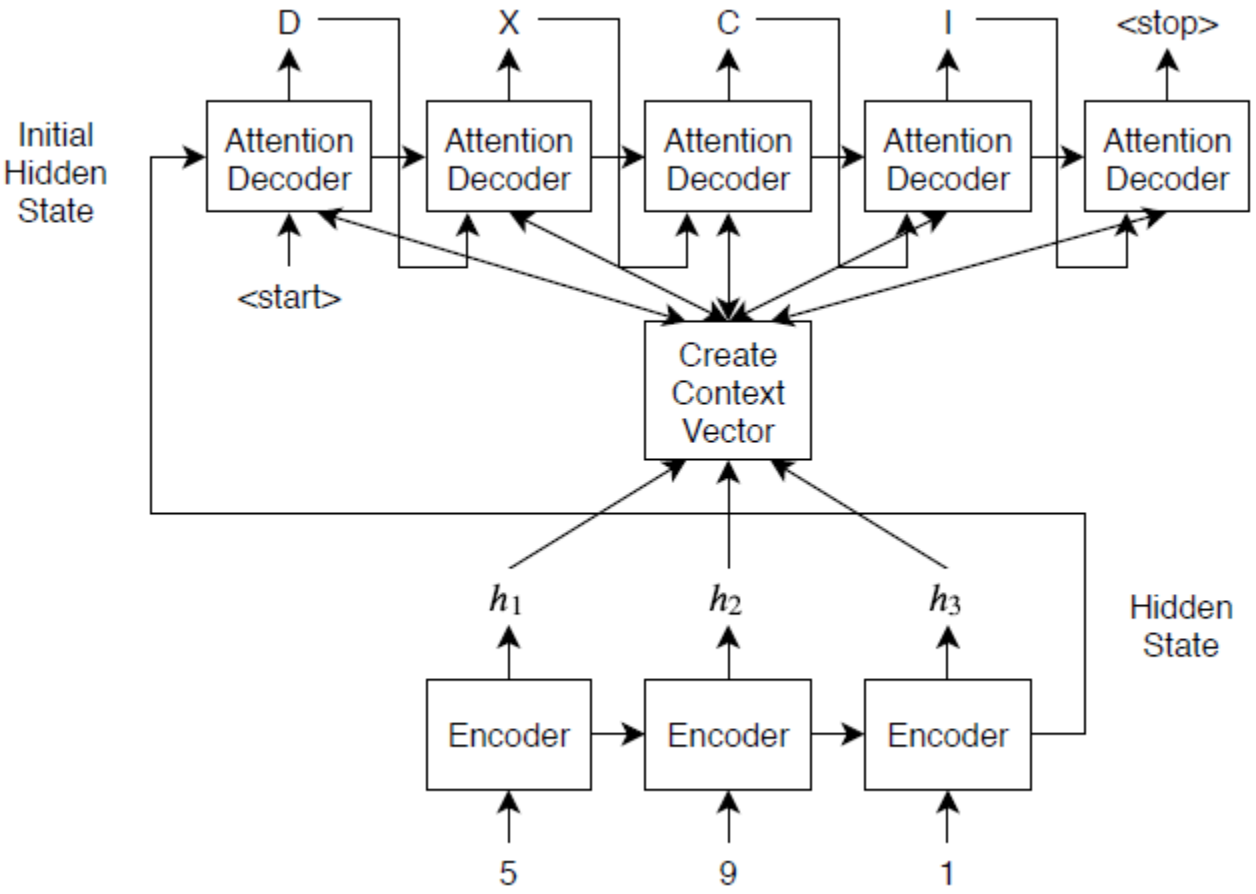
Related Examples

- “Train Network Using Custom Training Loop” on page 15-134
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Sequence-to-Sequence Translation Using Attention” on page 4-111
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Classify Text Data Using Deep Learning” on page 4-74
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

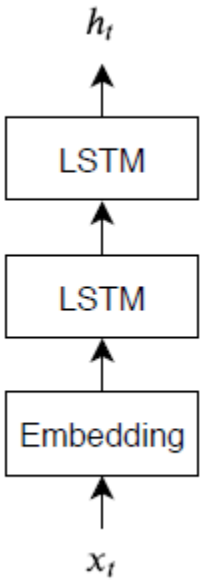
Sequence-to-Sequence Translation Using Attention

This example shows how to convert decimal strings to Roman numerals using a recurrent sequence-to-sequence encoder-decoder model with attention.

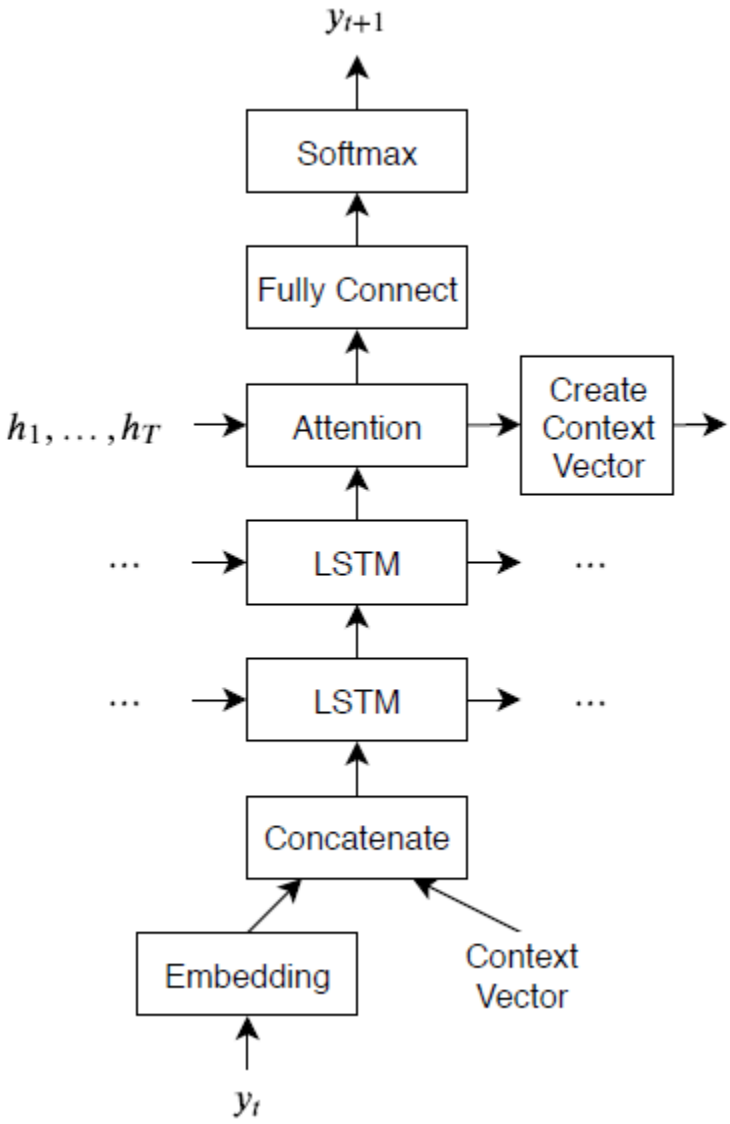
Recurrent encoder-decoder models have proven successful at tasks like abstractive text summarization and neural machine translation. The models consistent of an *encoder* which typically processes input data with a recurrent layer such as LSTM, and a *decoder* which maps the encoded input into the desired 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.



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
    ______    _________

    "168"     "CLXVIII"
    "154"     "CLIV"
    "765"     "DCCLXV"
    "714"     "DCCXIV"
    "649"     "DCXLIX"
    "346"     "CCCXLVI"
    "77"      "LXXVII"
    "83"      "LXXXIII"
```

Preprocess Data

Preprocess the training data using the `preprocessSourceTargetPairs` function, listed at the end of the example. The `preprocessSourceTargetPairs` function converts the input text data to numeric sequences. The elements of the sequences are positive integers that index into a corresponding `wordEncoding` object. The `wordEncoding` maps tokens to a numeric index and vice-versa using a vocabulary. To highlight the beginning and the ends of sequences, the encoding also encapsulates the special tokens `"<start>"` and `"<stop>"`.

```
startToken = "<start>";
stopToken = "<stop>";
[sequencesSource, sequencesTarget, encSource, encTarget] = preprocessSourceTargetPairs(dataTrain
```

Representing Text as Numeric Sequences

For example, the decimal string `"441"` is encoded as follows:

```
strSource = "441";
```

Insert spaces between the characters.

```
strSource = strip(replace(strSource,"","" "));
```

Add the special start and stop tokens.

```
strSource = startToken + strSource + stopToken
```

```
strSource =
"<start>4 4 1<stop>"
```

Tokenize the text using the `tokenizedDocument` function and set the `'CustomTokens'` option to the special tokens.

```
documentSource = tokenizedDocument(strSource,'CustomTokens',[startToken stopToken])
```

```
documentSource =
  tokenizedDocument:
```

```
5 tokens: <start> 4 4 1 <stop>
```

Convert the document to a sequence of token indices using the `word2ind` function with the corresponding `wordEncoding` object.

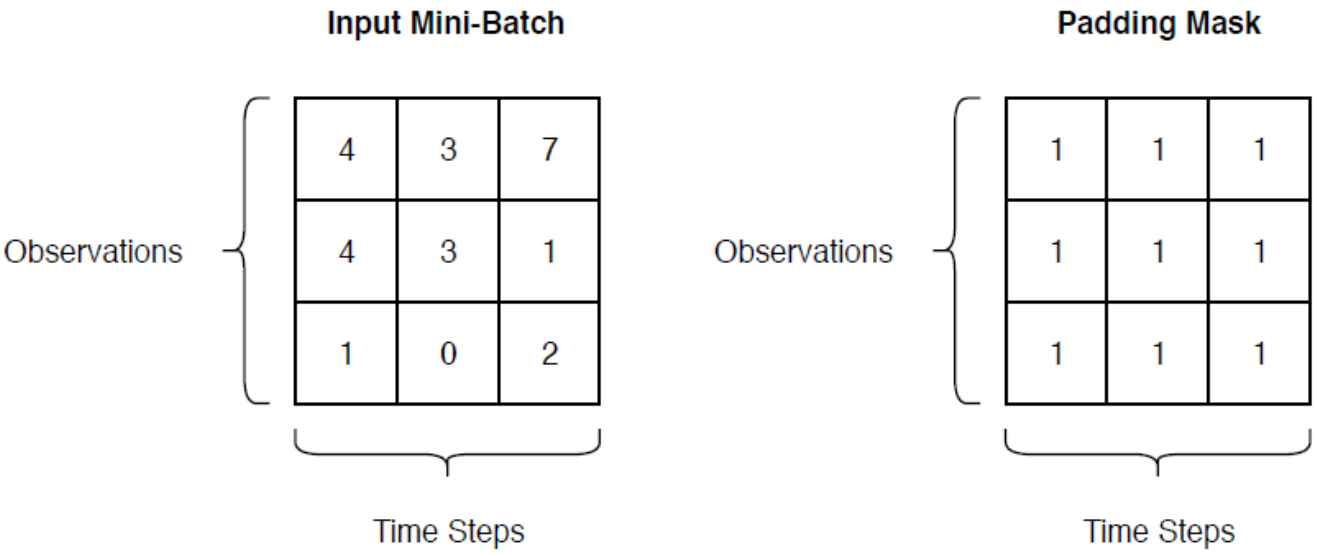
```
tokens = string(documentSource);
sequenceSource = word2ind(encSource,tokens)
```

```
sequenceSource = 1×5

     1     7     7     2     5
```

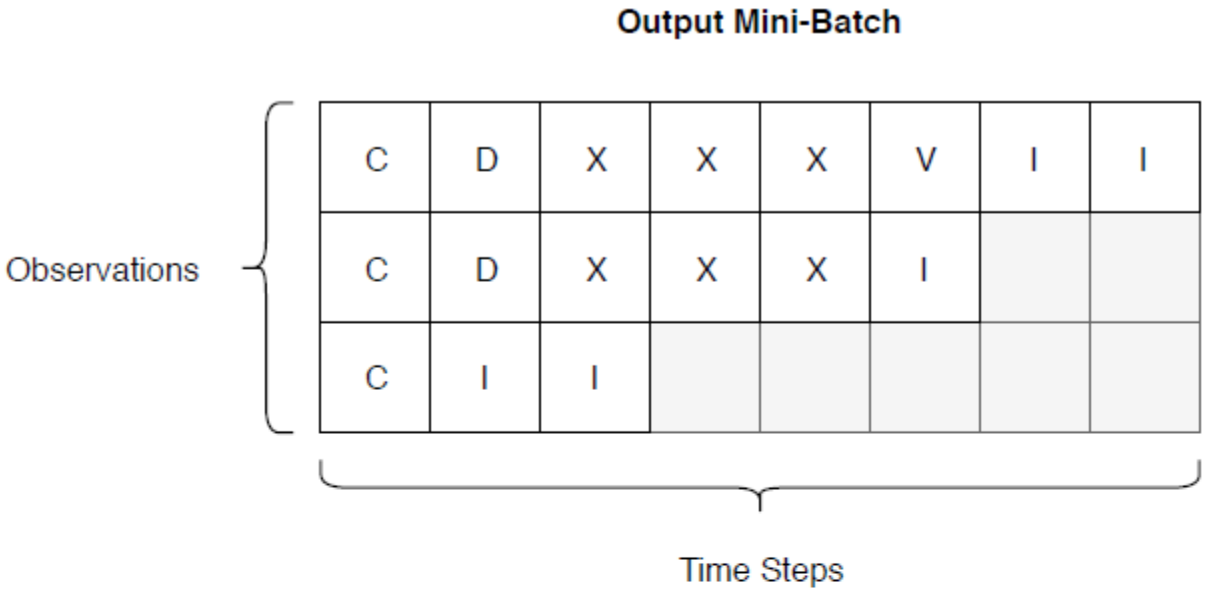
Padding and Masking

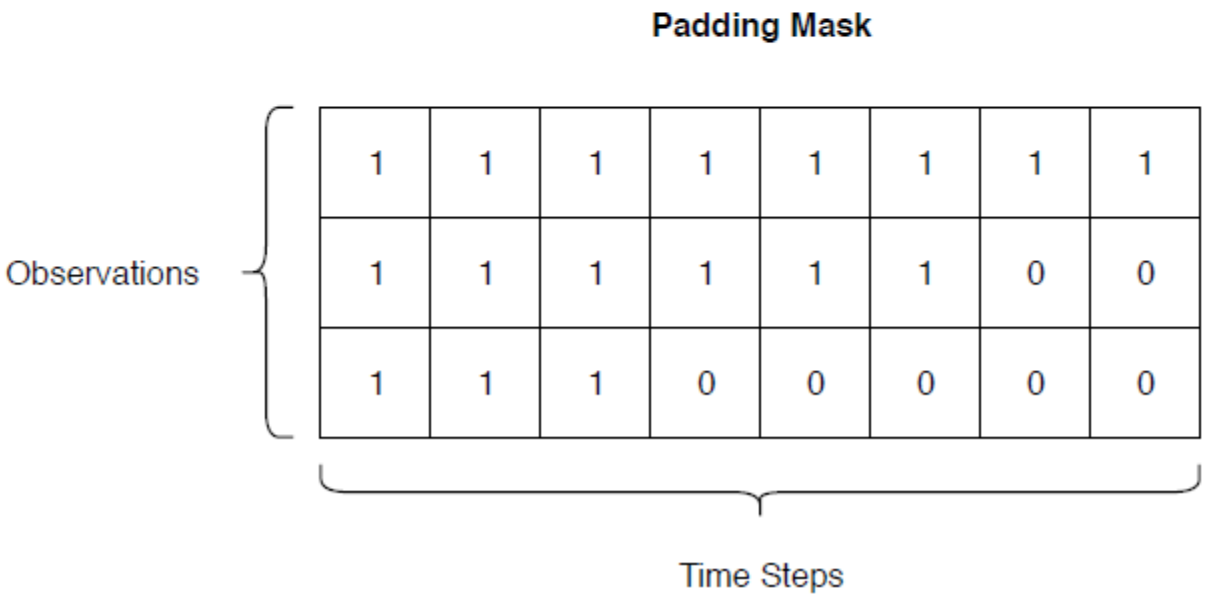
Sequence data such as text naturally have different sequence lengths. To train a model using variable length sequences, pad the mini-batches of input data to have the same length. To ensure that the padding values do not impact the loss calculations, create a mask which records which sequence elements are real, and which are just padding.

For example, consider a mini-batch containing the decimal strings "437", "431", and "102" with the corresponding Roman numeral strings "CDXXXVII", "CDXXXI", and "CII". For character-by-character sequences, the input sequences have the same length and do not need to be padded. The corresponding mask is an array of ones.



The output sequences have different lengths, so they require padding. The corresponding padding mask contains zeros where the corresponding time steps are padding values.





Initialize Model Parameters

Initialize the model parameters. for both the encoder and decoder, specify an embedding dimension of 256, two LSTM layers with 200 hidden units, and dropout layers with random dropout with probability 0.05.

```
embeddingDimension = 256;
numHiddenUnits = 200;
dropout = 0.05;
```

Initialize the encoder model parameters:

- Specify an embedding dimension of 256 and the vocabulary size of the source vocabulary plus 1, where the extra value corresponds to the padding token.
- Specify two LSTM operations with 200 hidden units.
- Initialize the embedding weights by sampling from a random normal distribution.
- Initialize the LSTM weights and biases by sampling from a uniform distribution using the `uniformNoise` function, listed at the end of the example.

```
inputSize = encSource.NumWords + 1;
parametersEncoder.emb.Weights = dlarray(randn([embeddingDimension inputSize]));

parametersEncoder.lstm1.InputWeights = dlarray(uniformNoise([4*numHiddenUnits embeddingDimension]
parametersEncoder.lstm1.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits]
parametersEncoder.lstm1.Bias = dlarray(uniformNoise([4*numHiddenUnits 1],1/numHiddenUnits));

parametersEncoder.lstm2.InputWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits],1/n
parametersEncoder.lstm2.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits]
parametersEncoder.lstm2.Bias = dlarray(uniformNoise([4*numHiddenUnits 1],1/numHiddenUnits));
```

Initialize the decoder model parameters.

- Specify an embedding dimension of 256 and the vocabulary size of the target vocabulary plus 1, where the extra value corresponds to the padding token.
- Initialize the attention mechanism weights using the `uniformNoise` function.
- Initialize the embedding weights by sampling from a random normal distribution.
- Initialize the LSTM weights and biases by sampling from a uniform distribution using the `uniformNoise` function.

```
outputSize = encTarget.NumWords + 1;
parametersDecoder.emb.Weights = dlarray(randn([embeddingDimension outputSize]));

parametersDecoder.attn.Weights = dlarray(uniformNoise([numHiddenUnits numHiddenUnits],1/numHidden

parametersDecoder.lstm1.InputWeights = dlarray(uniformNoise([4*numHiddenUnits embeddingDimension+
parametersDecoder.lstm1.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits]
parametersDecoder.lstm1.Bias = dlarray( uniformNoise([4*numHiddenUnits 1],1/numHiddenUnits));

parametersDecoder.lstm2.InputWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits],1/n
parametersDecoder.lstm2.RecurrentWeights = dlarray(uniformNoise([4*numHiddenUnits numHiddenUnits]
parametersDecoder.lstm2.Bias = dlarray(uniformNoise([4*numHiddenUnits 1], 1/numHiddenUnits));

parametersDecoder.fc.Weights = dlarray(uniformNoise([outputSize 2*numHiddenUnits],1/(2*numHiddenU
parametersDecoder.fc.Bias = dlarray(uniformNoise([outputSize 1], 1/(2*numHiddenUnits)));
```

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 40 epochs. Specify a learning rate of 0.002 and clip the gradients with a threshold of 5.

```
miniBatchSize = 32;
numEpochs = 40;
learnRate = 0.002;
gradientThreshold = 5;
```

Initialize the options from Adam.

```
gradientDecayFactor = 0.9;
squaredGradientDecayFactor = 0.999;
```

Specify to plot the training progress. To disable the training progress plot, set the `plots` value to `"none"`.

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

Train Model

Train the model using a custom training loop.

For the first epoch, train with the sequences sorted by increasing sequence length. This 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. For subsequent epochs, shuffle the data.

For each mini-batch:

- Convert the data to `dlarray`.
- Compute loss and gradients.
- Clip the gradients.
- Update the encoder and decoder model parameters using the `adamupdate` function.
- Update the training progress plot.

Sort the sequences for the first epoch.

```
sequenceLengthsEncoder = cellfun(@(sequence) size(sequence,2), sequencesSource);
[~,idx] = sort(sequenceLengthsEncoder);
sequencesSource = sequencesSource(idx);
sequencesTarget = sequencesTarget(idx);
```

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
```

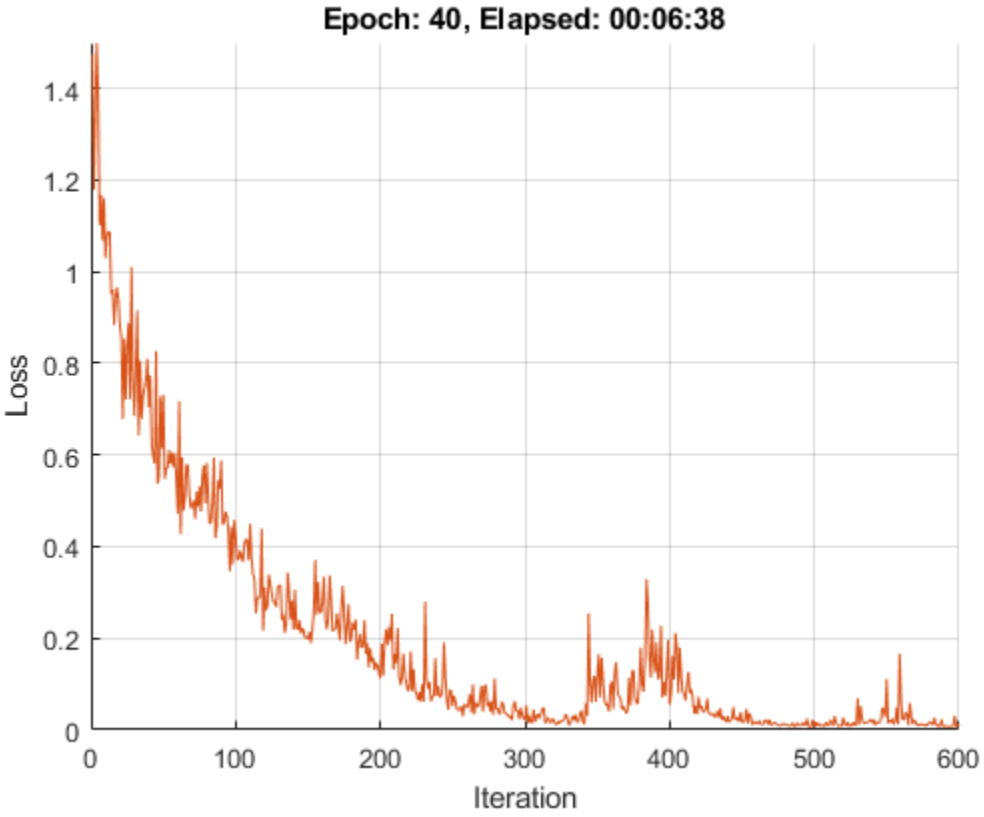
Initialize the values for the `adamupdate` function.

```
trailingAvgEncoder = [];
trailingAvgSqEncoder = [];

trailingAvgDecoder = [];
trailingAvgSqDecoder = [];
```

Train the model.

```
numObservations = numel(sequencesSource);
numIterationsPerEpoch = floor(numObservations/miniBatchSize);

iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        [XSource, XTarget, maskSource, maskTarget] = createBatch(sequencesSource(idx), ...
            sequencesTarget(idx), inputSize, outputSize);

        % Convert mini-batch of data to dlarray.
        dlXSource = dlarray(XSource);
        dlXTarget = dlarray(XTarget);

        % Compute loss and gradients.
        [gradientsEncoder, gradientsDecoder, loss] = dlfeval(@modelGradients, parametersEncoder,
            parametersDecoder, dlXSource, dlXTarget, maskSource, maskTarget, dropout);

        % Gradient clipping.
        gradientsEncoder = dlupdate(@(w) clipGradient(w,gradientThreshold), gradientsEncoder);
        gradientsDecoder = dlupdate(@(w) clipGradient(w,gradientThreshold), gradientsDecoder);

        % Update encoder using adamupdate.
        [parametersEncoder, trailingAvgEncoder, trailingAvgSqEncoder] = adamupdate(parametersEnco
            gradientsEncoder, trailingAvgEncoder, trailingAvgSqEncoder, iteration, learnRate, ..
            gradientDecayFactor, squaredGradientDecayFactor);

        % Update decoder using adamupdate.
        [parametersDecoder, trailingAvgDecoder, trailingAvgSqDecoder] = adamupdate(parametersDeco
            gradientsDecoder, trailingAvgDecoder, trailingAvgSqDecoder, iteration, learnRate, ..
```

```
            gradientDecayFactor, squaredGradientDecayFactor);

        % Display the training progress.
        if plots == "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

    % Shuffle data.
    idx = randperm(numObservations);
    sequencesSource = sequencesSource(idx);
    sequencesTarget = sequencesTarget(idx);
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.

Prepare Data for Translation

Select a mini-batch of test observations.

```
numObservationsTest = 16;
idx = randperm(size(dataTest,1),numObservationsTest);
dataTest(idx,:)
```

```
ans=16×2 table
    Source      Target
    ______    ___________

    "857"     "DCCCLVII"
    "991"     "CMXCI"
    "143"     "CXLIII"
    "924"     "CMXXIV"
    "752"     "DCCLII"
    "85"      "LXXXV"
    "131"     "CXXXI"
    "124"     "CXXIV"
    "858"     "DCCCLVIII"
    "103"     "CIII"
    "497"     "CDXCVII"
    "76"      "LXXVI"
    "815"     "DCCCXV"
    "829"     "DCCCXXIX"
    "940"     "CMXL"
    "94"      "XCIV"
```

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{idx,1};
strTarget = dataTest{idx,2};

documentsSource = transformText(strSource,startToken,stopToken);
```

Convert the tokenized text into a batch of padded sequences by using the `doc2sequence` function. To automatically pad the sequences, set the `'PaddingDirection'` option to `'right'` and set the padding value to the input size (the token index of the padding token).

```
sequencesSource = doc2sequence(encSource,documentsSource, ...
    'PaddingDirection','right', ...
    'PaddingValue',inputSize);
```

Concatenate and permute the sequence data into the required shape for the encoder model function (1-by-*N*-by-*S*, where *N* is the number of observations and *S* is the sequence length).

```
XSource = cat(3,sequencesSource{:});
XSource = permute(XSource,[1 3 2]);
```

Convert input data to `dlarray` and calculate the encoder model outputs.

```
dlXSource = dlarray(XSource);
[dlZ, hiddenState] = modelEncoder(dlXSource, parametersEncoder);
```

Translate Source Text

To generate translations for new data input the sequences into the encoder-decoder model and convert the resulting sequences back into text using the token indices.

To initialize the translations, create a vector containing only the indices corresponding to the start token.

```
decoderInput = repmat(word2ind(encTarget,startToken),[1 numObservationsTest]);
decoderInput = dlarray(decoderInput);
```

Initialize the context vector and the cell arrays containing the translated sequences and the attention scores for each observation.

```
context = dlarray(zeros([size(dlZ, 1) numObservationsTest]));
sequencesTranslated = cell(1,numObservationsTest);
attentionScores = cell(1,numObservationsTest);
```

Loop over time steps and translate the sequences. Keep looping over the time steps until all sequences translated. For each observation, when the translation is finished (when the decoder predicts the stop token), set a flag to stop translating that sequence.

```
stopIdx = word2ind(encTarget,stopToken);

stopTranslating = false(1, numObservationsTest);
maxSequenceLength = 10;

while ~all(stopTranslating)

    % Forward through decoder.
    [dlY, context, hiddenState, attn] = modelDecoder(decoderInput, parametersDecoder, context, .
        hiddenState, dlZ);

    % Loop over observations.
    for i = 1:numObservationsTest
        % Skip already-translated sequences.
        if stopTranslating(i)
            continue
        end


        % Update attention scores.
        attentionScores{i} = [attentionScores{i} extractdata(attn(:,i))];

        % Predict next time step.
        prob = softmax(dlY(:,i), 'DataFormat', 'CB');
        [~, idx] = max(prob(1:end-1,:), [], 1);

        % Set stopTranslating flag when translation done.
        if idx == stopIdx || numel(sequencesTranslated{i}) == maxSequenceLength
            stopTranslating(i) = true;
        else
            sequencesTranslated{i} = [sequencesTranslated{i} extractdata(idx)];
            decoderInput(i) = idx;
        end
    end
end
```

View the source text, target text, and translations in a table.

```
tbl = table;
tbl.Source = strSource;
tbl.Target = strTarget;
```

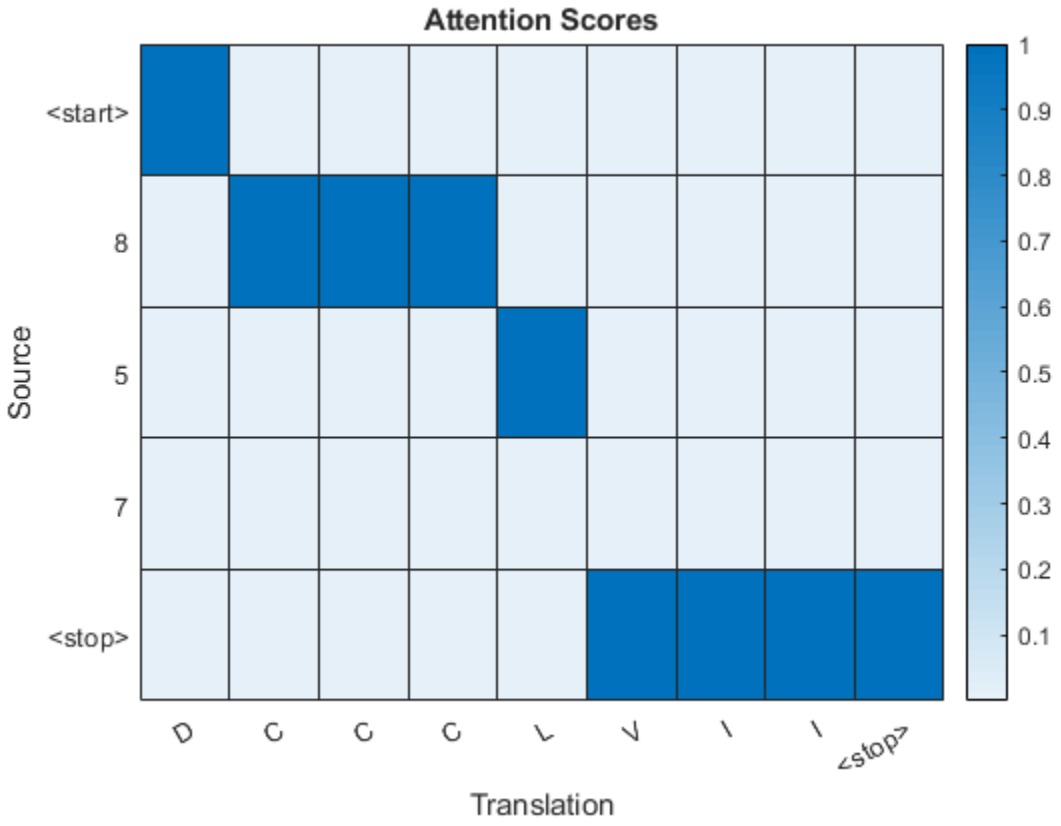
```
tbl.Translated = cellfun(@(sequence) join(ind2word(encTarget,sequence),""),sequencesTranslated)'
tbl
```

```
tbl=16×3 table
    Source      Target       Translated
    ______    ___________    ___________

    "857"     "DCCCLVII"     "DCCCLVII"
    "991"     "CMXCI"        "CMXCI"
    "143"     "CXLIII"       "CXLIII"
    "924"     "CMXXIV"       "CMXXIV"
    "752"     "DCCLII"       "DCCLII"
    "85"      "LXXXV"        "DCCCLVI"
    "131"     "CXXXI"        "CXXXI"
    "124"     "CXXIV"        "CXXIV"
    "858"     "DCCCLVIII"    "DCCCLVIII"
    "103"     "CIII"         "CIII"
    "497"     "CDXCVII"      "CDXCVII"
    "76"      "LXXVI"        "DCCLVII"
    "815"     "DCCCXV"       "DCCCXV"
    "829"     "DCCCXXIX"     "DCCCXXIX"
    "940"     "CMXL"         "CMXL"
    "94"      "XCIV"         "CMXLVI"
```

Plot Attention Scores

Plot the attention scores of the first sequence in a heat map. The attention scores highlight which areas of the source and translated sequences the model attends to when processing the translation.

```
idx = 1;
figure
xlabs = [ind2word(encTarget,sequencesTranslated{idx}) stopToken];
ylabs = string(documentsSource(idx));

heatmap(attentionScores{idx}, ...
    'CellLabelColor','none', ...
    'XDisplayLabels',xlabs, ...
    'YDisplayLabels',ylabs);

xlabel("Translation")
ylabel("Source")
title("Attention Scores")
```



Preprocessing Function

The `preprocessSourceTargetPairs` takes a table `data` containing the source-target pairs in two columns and for each column returns sequences of token indices and a corresponding `wordEncoding` object that maps the indices to words and vice versa.

```
function [sequencesSource, sequencesTarget, encSource, encTarget] = preprocessSourceTargetPairs(d

% Extract text data.
strSource = data{:,1};
strTarget = data{:,2};

% Create tokenized document arrays.
documentsSource = transformText(strSource,startToken,stopToken);
documentsTarget = transformText(strTarget,startToken,stopToken);

% Create word encodings.
encSource = wordEncoding(documentsSource);
encTarget = wordEncoding(documentsTarget);

% Convert documents to numeric sequences.
sequencesSource = doc2sequence(encSource, documentsSource,'PaddingDirection','none');
sequencesTarget = doc2sequence(encTarget, documentsTarget,'PaddingDirection','none');

end
```

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)

% Split text into characters.
str = strip(replace(str,"","" ""));

% Add start and stop tokens.
str = startToken + str + stopToken;

% Create tokenized document array.
documents = tokenizedDocument(str,'CustomTokens',[startToken stopToken]);

end
```

Batch Creation Function

The `createBatch` function takes a mini-batch of source and target sequences and returns padded sequences with the corresponding padding masks.

```
function [XSource, XTarget, maskSource, maskTarget] = createBatch(sequencesSource, sequencesTarge
    paddingValueSource, paddingValueTarget)

numObservations = size(sequencesSource,1);
sequenceLengthSource = max(cellfun(@(x) size(x,2), sequencesSource));
sequenceLengthTarget = max(cellfun(@(x) size(x,2), sequencesTarget));

% Initialize masks.
maskSource = false(numObservations, sequenceLengthSource);
maskTarget = false(numObservations, sequenceLengthTarget);

% Initialize mini-batch.
XSource = zeros(1,numObservations,sequenceLengthSource);
XTarget = zeros(1,numObservations,sequenceLengthTarget);

% Pad sequences and create masks.
for i = 1:numObservations

    % Source
    L = size(sequencesSource{i},2);
    paddingSize = sequenceLengthSource - L;
    padding = repmat(paddingValueSource, [1 paddingSize]);

    XSource(1,i,:) = [sequencesSource{i} padding];
    maskSource(i,1:L) = true;

    % Target
    L = size(sequencesTarget{i},2);
    paddingSize = sequenceLengthTarget - L;
    padding = repmat(paddingValueTarget, [1 paddingSize]);

    XTarget(1,i,:) = [sequencesTarget{i} padding];
    maskTarget(i,1:L) = true;
end
```

```
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.

```
function [dlZ, hiddenState] = modelEncoder(dlX, parametersEncoder, maskSource)

% Embedding
weights = parametersEncoder.emb.Weights;
dlZ = embedding(dlX,weights);

% LSTM
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
inputWeights = parametersEncoder.lstm2.InputWeights;
recurrentWeights = parametersEncoder.lstm2.RecurrentWeights;
bias = parametersEncoder.lstm2.Bias;

[dlZ, hiddenState] = lstm(dlZ,initialHiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Mask output for training
if nargin > 2
    dlZ = dlZ.*permute(maskSource, [3 1 2]);
    sequenceLengths = sum(maskSource, 2);

    % Mask final hidden state
    for ii = 1:size(dlZ, 2)
        hiddenState(:, ii) = dlZ(:, ii, sequenceLengths(ii));
    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(dlX, parameters, context, .
    hiddenState, encoderOutputs, dropout)

% Embedding
weights = parameters.emb.Weights;
```

```
dlX = embedding(dlX, weights);

% RNN input
dlY = cat(1, dlX, context);

% LSTM 1
initialCellState = dlarray(zeros(size(hiddenState)));

inputWeights = parameters.lstm1.InputWeights;
recurrentWeights = parameters.lstm1.RecurrentWeights;
bias = parameters.lstm1.Bias;

dlY = lstm(dlY, hiddenState, initialCellState, inputWeights, ...
    recurrentWeights, bias, 'DataFormat', 'CBT');

if nargin > 5
    % Dropout
    mask = ( rand(size(dlY), 'like', dlY) > dropout );
    dlY = dlY.*mask;
end

% LSTM 2
inputWeights = parameters.lstm2.InputWeights;
recurrentWeights = parameters.lstm2.RecurrentWeights;
bias = parameters.lstm2.Bias;
[~, hiddenState] = lstm(dlY, hiddenState, initialCellState, ...
    inputWeights, recurrentWeights, bias, 'DataFormat', 'CBT');

% Attention
weights = parameters.attn.Weights;
attentionScores = attention(hiddenState, encoderOutputs, weights);

% Context
encoderOutputs = permute(encoderOutputs, [1 3 2]);
attentionScores = permute(attentionScores,[1 3 2]);
context = dlmtimes(encoderOutputs,attentionScores);
context = squeeze(context);

% Fully connect
weights = parameters.fc.Weights;
bias = parameters.fc.Bias;
dlY = weights*cat(1, hiddenState, context) + bias;

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 out batch and sequence dimensions
Z = reshape(Z, [], N, T);
end
```

Attention Function

The `attention` function computes the attention scores according to Luong "general" scoring.

```
function attentionScores = attention(hiddenState, encoderOutputs, weights)

[N, S] = size(encoderOutputs, 2:3);
attentionEnergies = dlarray(zeros( [S N] ));

% The energy at each time step is the dot product of the hidden state
% and the learnable attention weights times the encoder output
hWX = hiddenState .* dlmtimes(weights,encoderOutputs);
for tt = 1:S
    attentionEnergies(tt, :) = sum(hWX(:, :, tt), 1);
end

% Compute softmax scores
attentionScores = softmax(attentionEnergies, 'DataFormat', 'CB');

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 [gradientsEncoder, gradientsDecoder, maskedLoss] = modelGradients(parametersEncoder, ..
    parametersDecoder, dlXSource, dlXTarget, maskSource, maskTarget, dropout)

% Forward through encoder.
[dlZ, hiddenState] = modelEncoder(dlXSource, parametersEncoder, maskSource);

% Get parameter sizes.
[miniBatchSize, sequenceLength] = size(dlXTarget,2:3);
sequenceLength = sequenceLength - 1;
numHiddenUnits = size(dlZ,1);

% Initialize context vector.
context = dlarray(zeros([numHiddenUnits miniBatchSize]));

% Initialize loss.
loss = dlarray(zeros([miniBatchSize sequenceLength]));

% Get first time step for decoder.
decoderInput = dlXTarget(:,:,1);

% Choose whether to use teacher forcing.
doTeacherForcing = rand < 0.5;

if doTeacherForcing
    for t = 1:sequenceLength
        % Forward through decoder.
```

```
        [dlY, context, hiddenState] = modelDecoder(decoderInput, parametersDecoder, context, ...
            hiddenState, dlZ, dropout);

        % Update loss.
        dlT = dlarray(oneHot(dlXTarget(:,:,t+1), size(dlY,1)));
        loss(:,t) = crossEntropyAndSoftmax(dlY, dlT);

        % Get next time step.
        decoderInput = dlXTarget(:,:,t+1);
    end
else
    for t = 1:sequenceLength
        % Forward through decoder.
        [dlY, context, hiddenState] = modelDecoder(decoderInput, parametersDecoder, context, ...
            hiddenState, dlZ, dropout);

        % Update loss.
        dlT = dlarray(oneHot(dlXTarget(:,:,t+1), size(dlY,1)));
        loss(:,t) = crossEntropyAndSoftmax(dlY, dlT);

        % Greedily update next input time step.
        prob = softmax(dlY,'DataFormat','CB');
        [~, decoderInput] = max(prob,[],1);
    end
end

% Determine masked loss.
maskedLoss = sum(sum(loss.*maskTarget(:,2:end))) / miniBatchSize;

% Update gradients.
[gradientsEncoder, gradientsDecoder] = dlgradient(maskedLoss, parametersEncoder, parametersDecode

% For plotting, return loss normalized by sequence length.
maskedLoss = extractdata(maskedLoss) ./ sequenceLength;

end
```

Cross-Entropy and Softmax Loss Function

The `crossEntropyAndSoftmax` loss computes the cross-entropy and softmax loss.

```
function loss = crossEntropyAndSoftmax(dlY, dlT)

offset = max(dlY);
logSoftmax = dlY - offset - log(sum(exp(dlY-offset)));
loss = -sum(dlT.*logSoftmax);

end
```

Uniform Noise Function

The `uniformNoise` function samples weights from a uniform distribution.

```
function weights = uniformNoise(sz, k)

weights = -sqrt(k) + 2*sqrt(k).*rand(sz);

end
```

Gradient Clipping Function

The `clipGradient` function clips the model gradients.

```
function g = clipGradient(g, gradientThreshold)

wnorm = norm(extractdata(g));
if wnorm > gradientThreshold
    g = (gradientThreshold/wnorm).*g;
end

end
```

One-Hot Encoding Function

The `oneHot` function encodes word indices as one-hot vectors.

```
function oh = oneHot(idx, numTokens)
tokens = (1:numTokens)';
oh = (tokens == idx);
end
```

See Also

`adamupdate` | `crossentropy` | `dlarray` | `dlfeval` | `dlgradient` | `dlupdate` | `doc2sequence` | `lstm` | `softmax` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Multilabel Text Classification Using Deep Learning” on page 4-91
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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 ano
    {'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,↵W
    {'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];
```

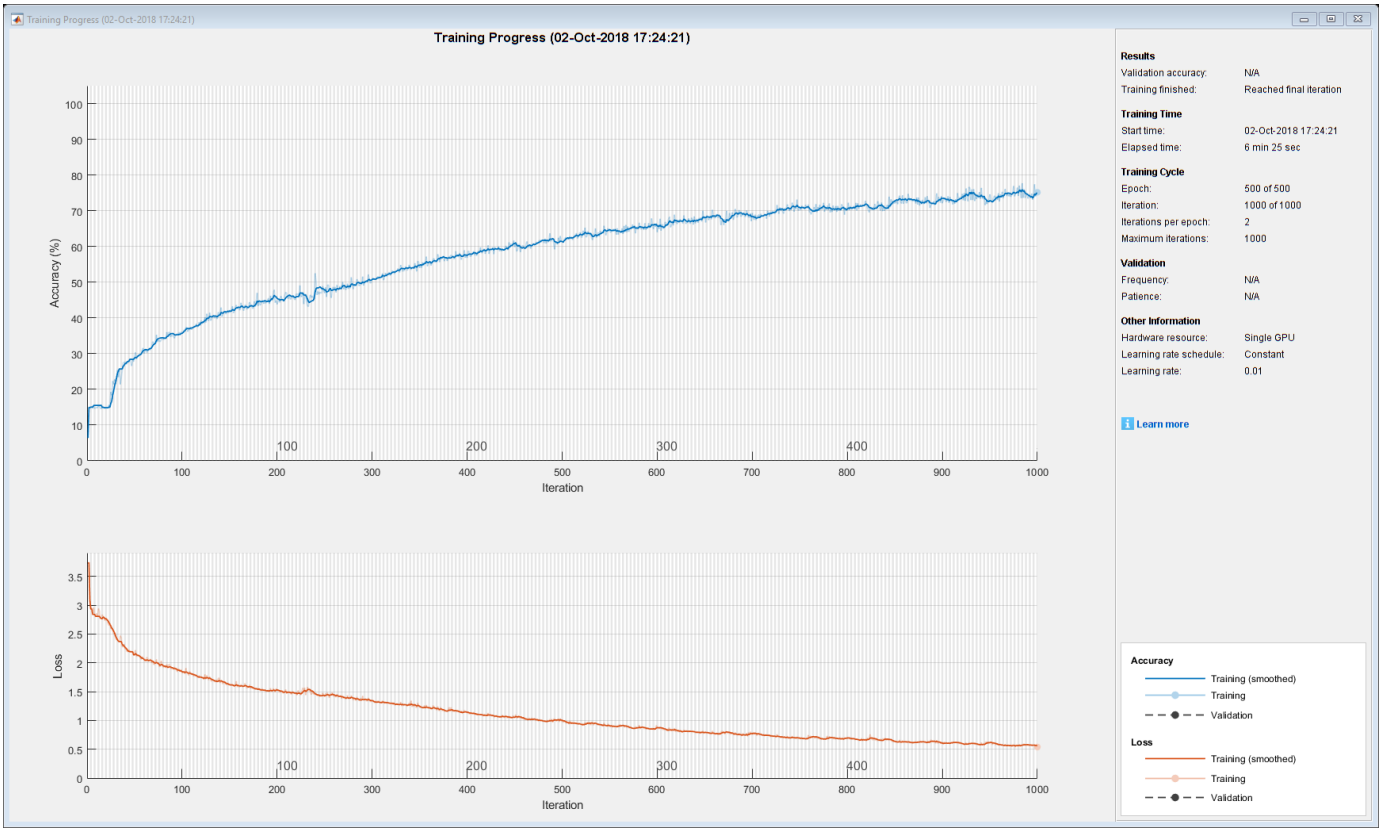
Specify the training options using the `trainingOptions` function. Specify the number of training epochs as 500 and the initial learn rate as 0.01. To prevent the gradients from exploding, set the gradient threshold to 2. Specify to shuffle the data every epoch by setting the `'Shuffle'` option to `'every-epoch'`. 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

`lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

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 4-9
- “Sequence Classification Using Deep Learning” on page 4-2
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “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.

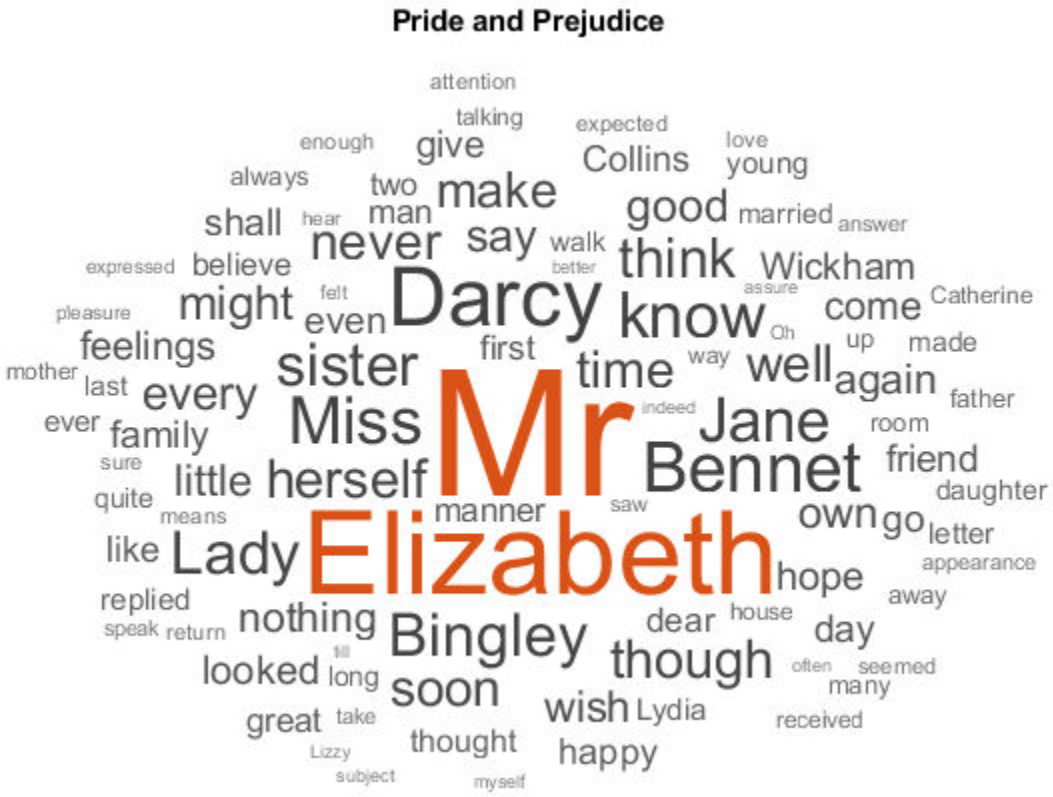
```
textData = extractHTMLText(paragraphs);
textData(textData == "") = [];
```

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.

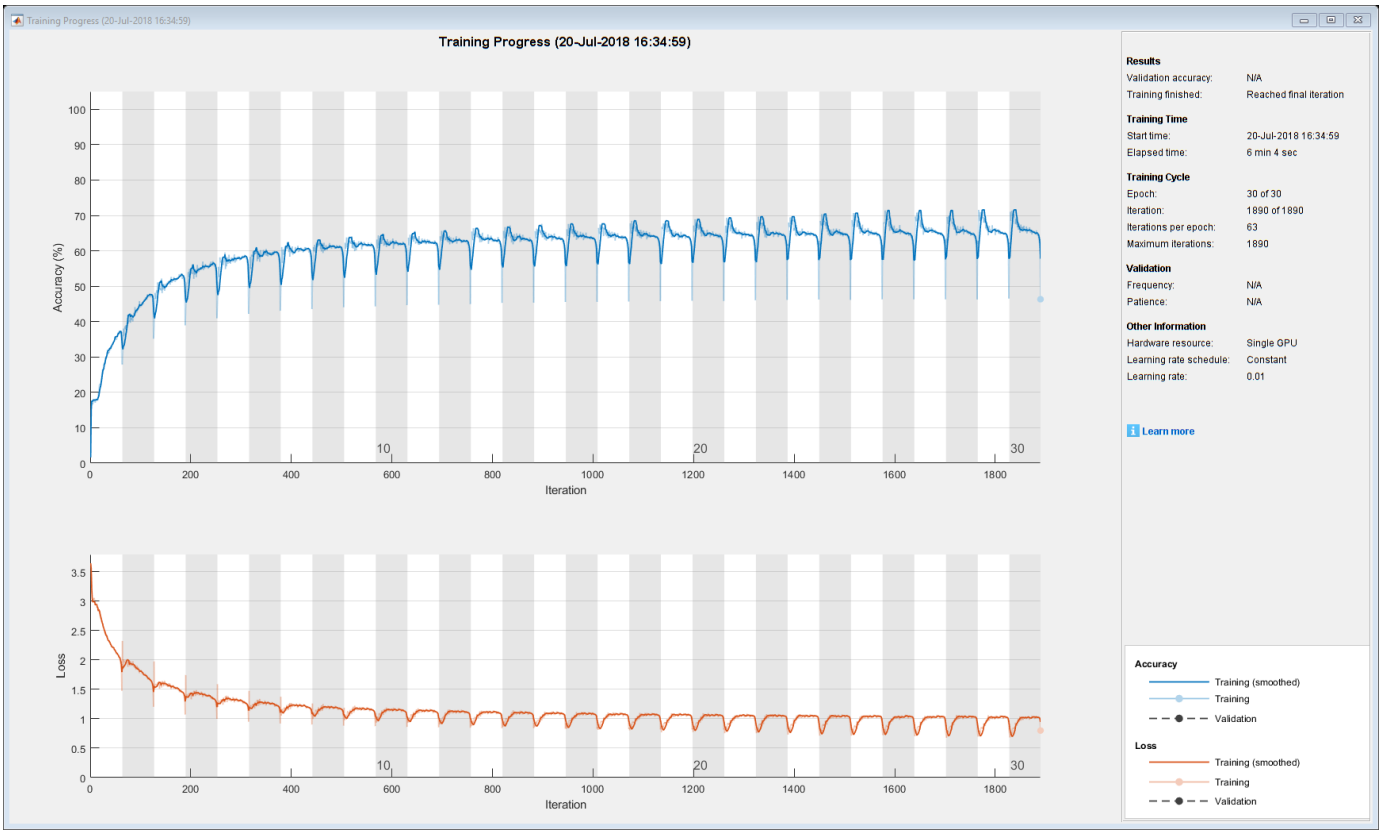
```
inputSize = size(XTrain{1},1);
numClasses = numel(categories([YTrain{:}]));
numCharacters = max([textData{:}]);

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

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

- “Generate Text Using Deep Learning” on page 4-131

- “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 `documentGenerationDatastore.m` 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 16-28.

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 array
    ""
    ""
    ""
    ""
    ""
    ""
    "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
```

Remove the empty paragraphs and view the first 10 remaining paragraphs.

```
textData(textData == "") = [];
textData(1:10)
```

```
ans = 10×1 string array
    "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'll
```

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`. To create the datastore, first save the custom mini-batch datastore `documentGenerationDatastore.m` to the path. 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];
```

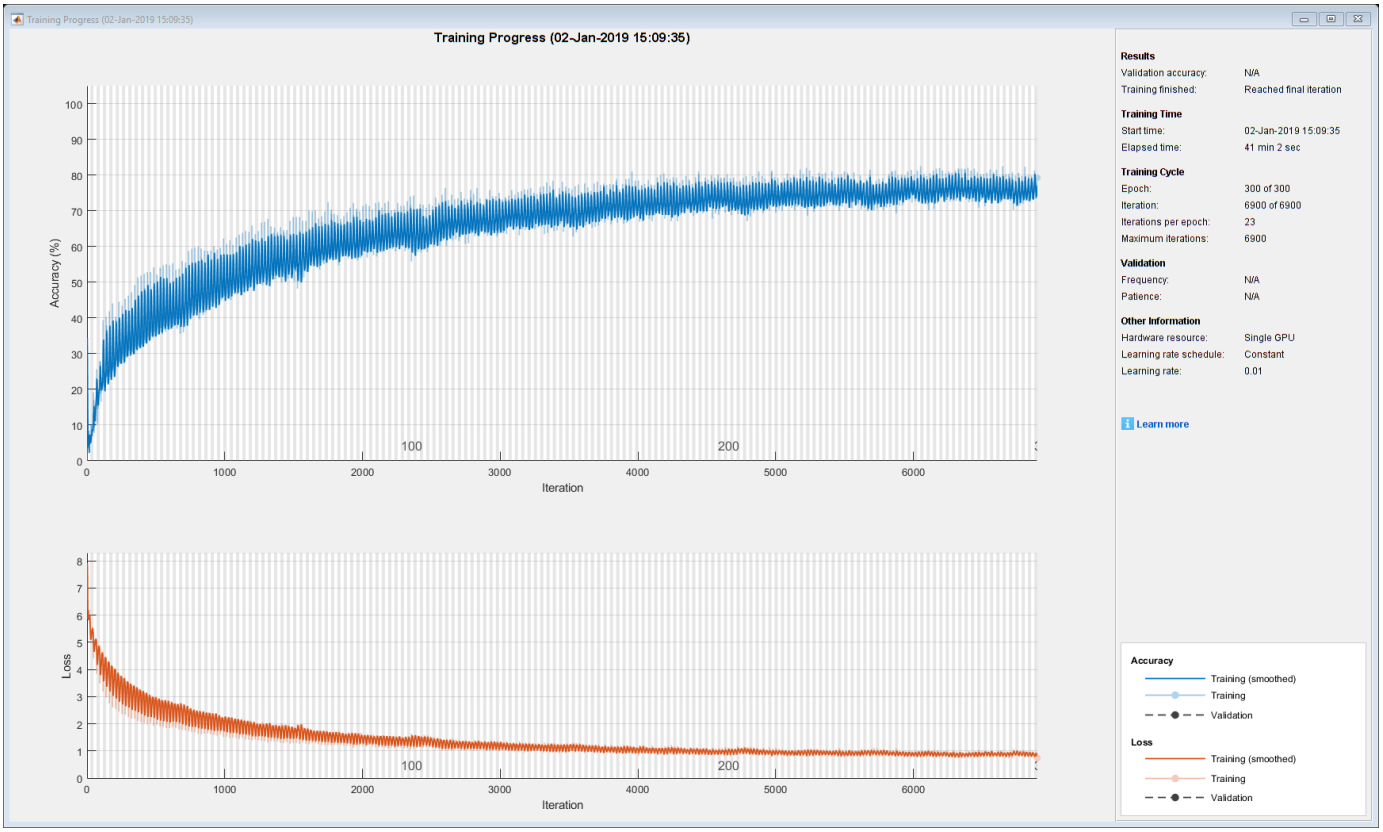
Specify the training options. Specify the solver to be `'adam'`. Train for 300 epochs with learn rate 0.01. Set the mini-batch size to 32. To keep the data sorted by sequence length, set the `'Shuffle'` option to `'never'`. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. 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 replacing 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 =
" ‘Sure, it’s a good Turtle!’ said the Queen in a low, weak voice."
```

To generate multiple pieces of text, reset the network state between generations using `resetState`.

```
net = resetState(net);
```

See Also

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

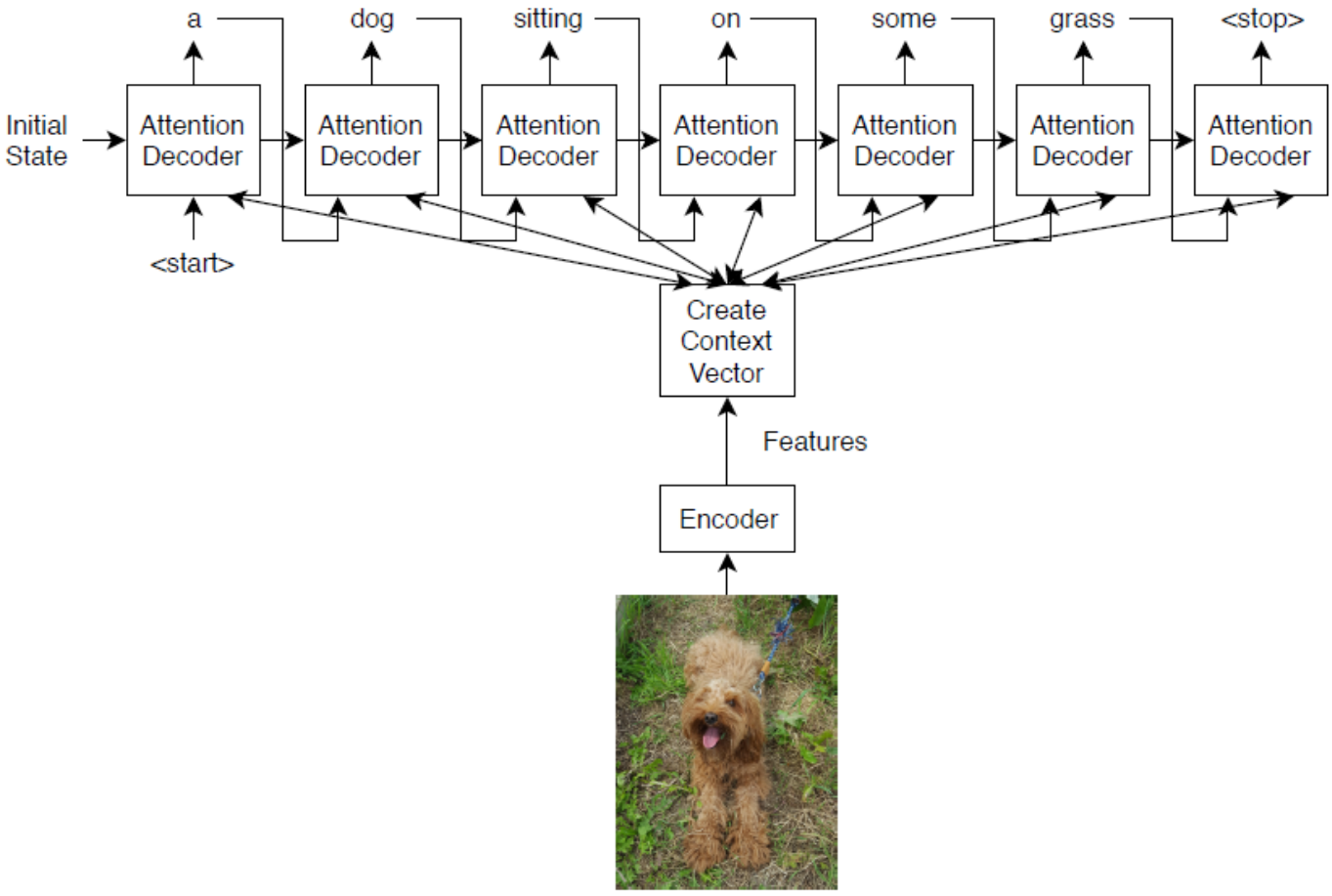
- “Generate Text Using Deep Learning” on page 4-131
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text Analytics Toolbox)
- “Sequence Classification Using Deep Learning” on page 4-2
- “Deep Learning in MATLAB” on page 1-2

Image Captioning Using Attention

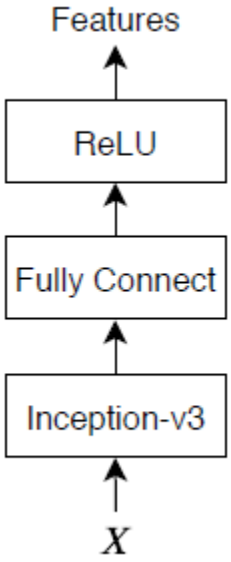
This example shows how to train a deep learning model for image captioning using attention.

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.

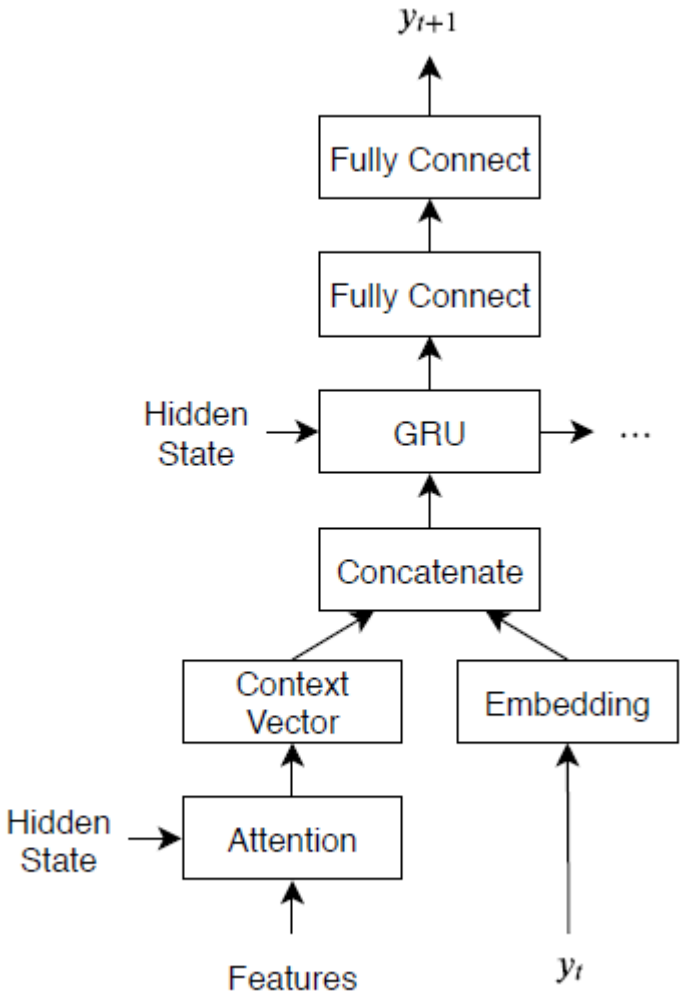
This model trained in this example uses an encoder-decoder architecture. The encoder is a pretrained Inception-v3 network used as a feature extractor. The decoder is a recurrent neural network (RNN) that takes the extracted features as input and generates a caption. The decoder incorporates an *attention 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)
```

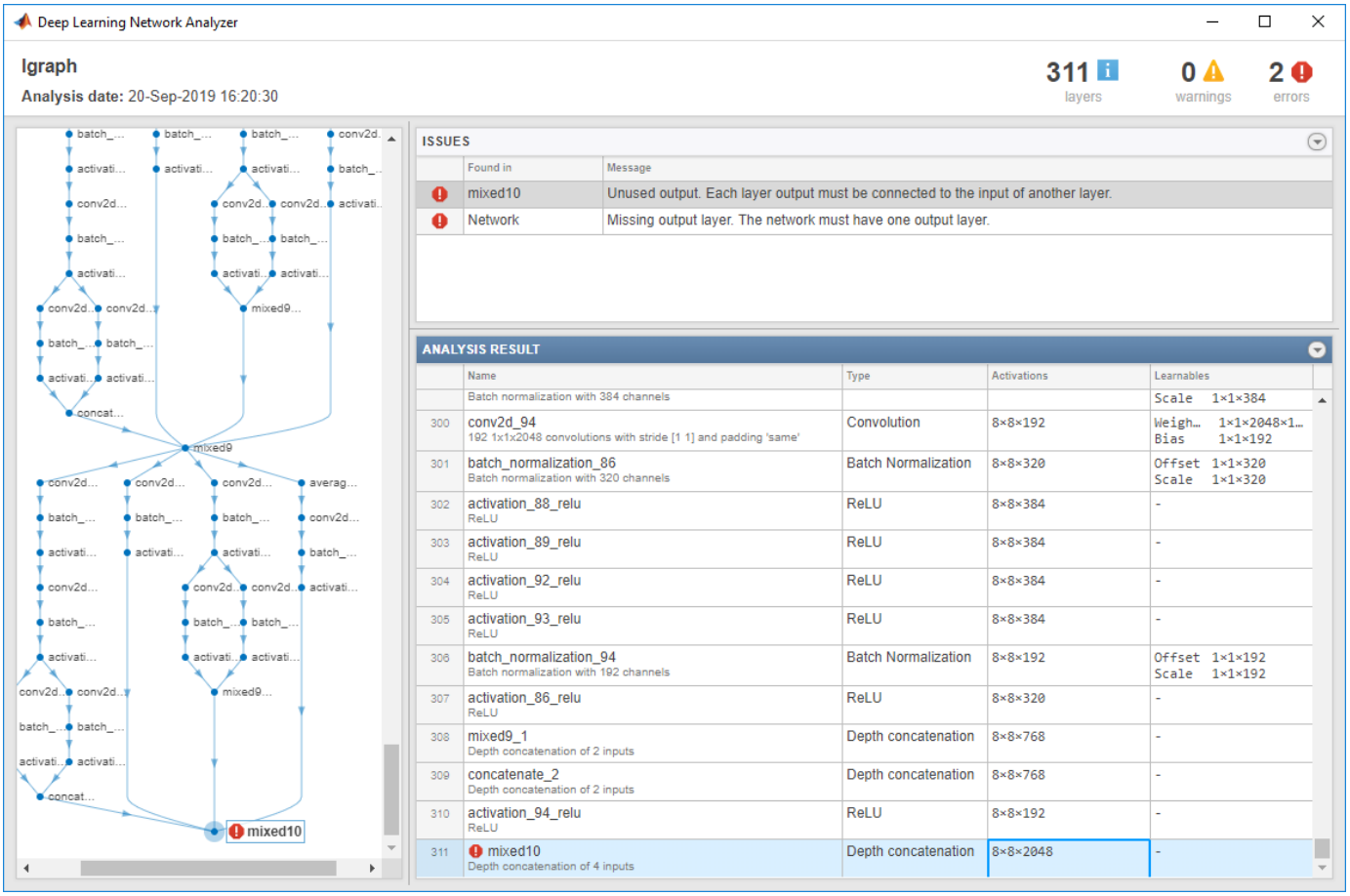
```
ans =
  ImageInputLayer with properties:

                     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. The Deep Learning Network Analyzer shows some issues with the network that can be safely ignored for custom training workflows.

```
analyzeNetwork(lgraph)
```



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 http://cocodataset.org/#download. Extract the images and annotations into a folder named `"coco"`. The COCO 2014 data set belongs to the Coco Consortium and is licensed under the Creative Commons Attribution 4.0 License.

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,embeddingDimensior
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*embeddingDimensi
parametersDecoder.gru.RecurrentWeights = dlarray(initializeGlorot(3*numHiddenUnits,numHiddenUnits
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 CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
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.

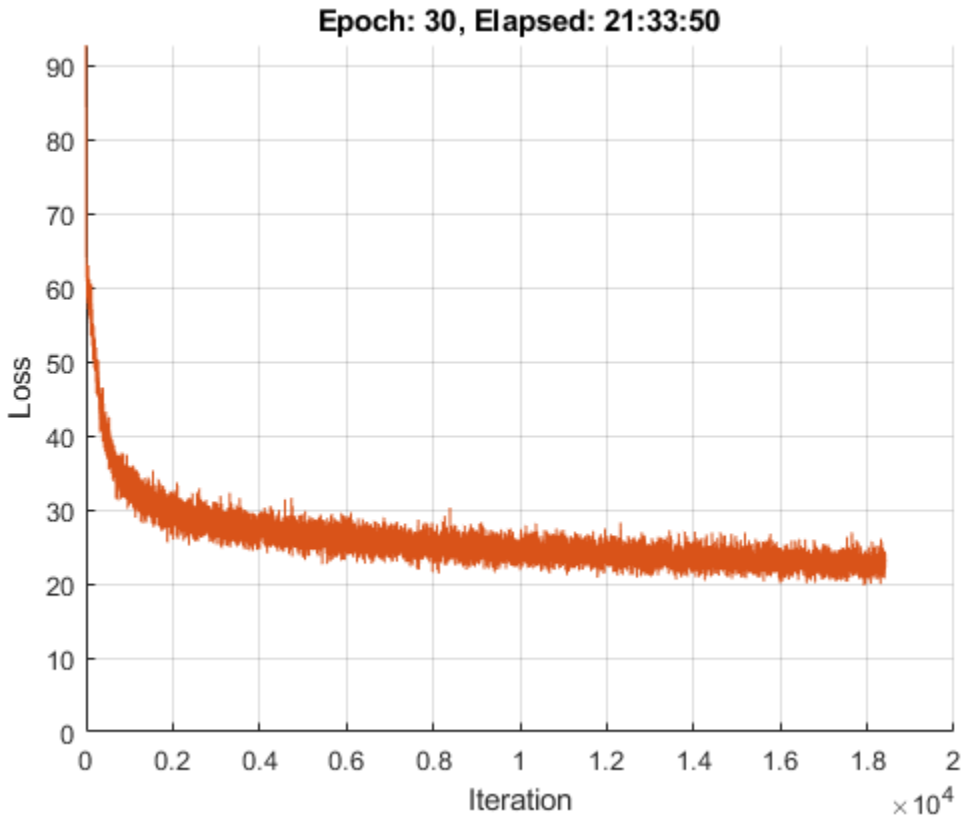
```
if plots == "training-progress"
    figure
    lineLossTrain = animatedline('Color',[0.85 0.325 0.098]);
```

```
    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.

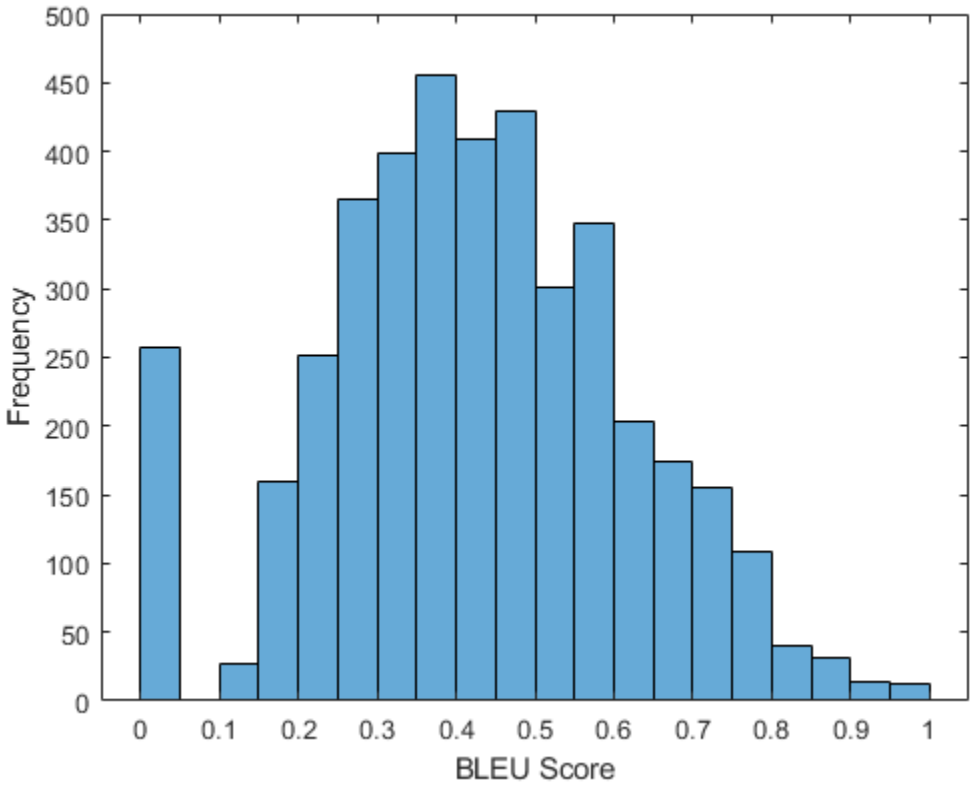
```
scoreMean = mean(scores)
```

```
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

`adamupdate` | `crossentropy` | `dlarray` | `dlfeval` | `dlgradient` | `dlupdate` | `doc2sequence` | `gru` | `lstm` | `softmax` | `tokenizedDocument` | `word2ind` | `wordEncoding`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125

- “Multilabel Text Classification Using Deep Learning” on page 4-91
- “Automatic Differentiation Background” on page 15-112

Deep Learning Tuning and Visualization

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

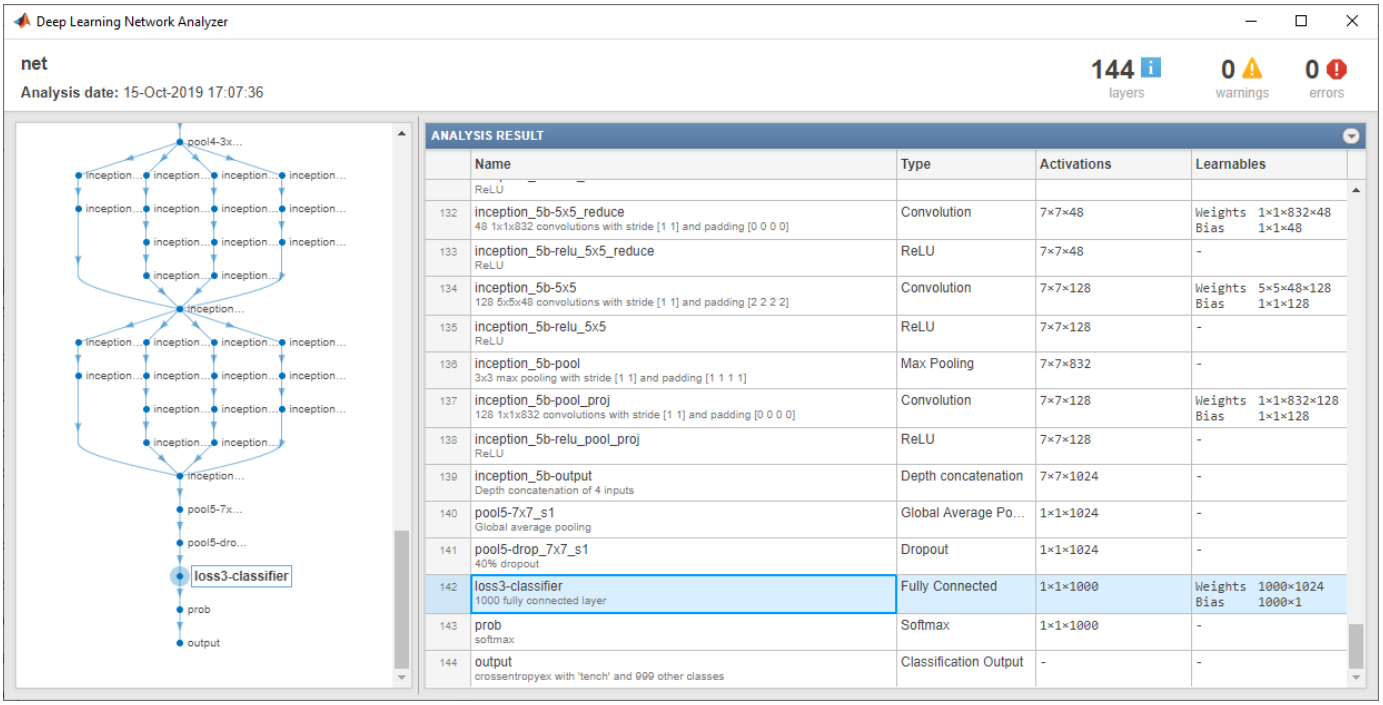
Load a pretrained GoogLeNet Network. If the Deep Learning Toolbox Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

```
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. A CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher is required for running on a GPU.

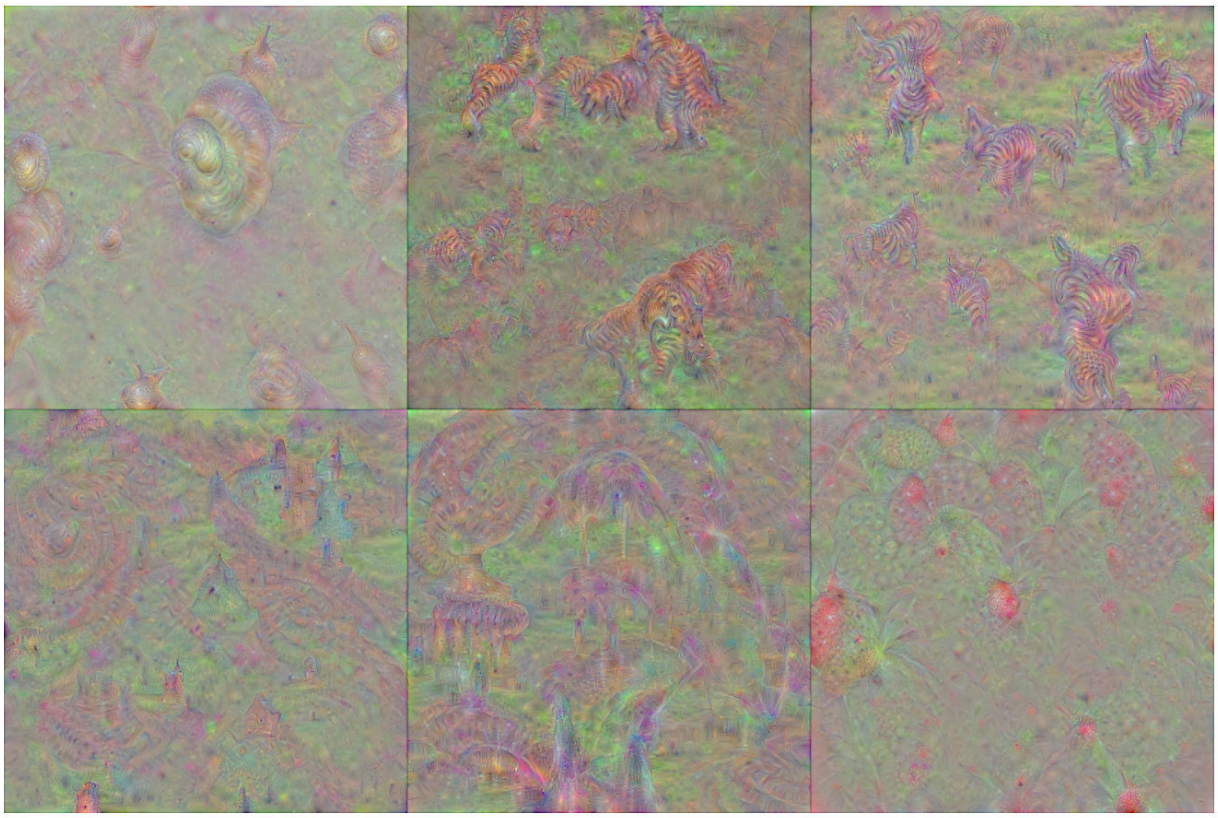
```
I = deepDreamImage(net,layerName,channels);
```

```
|=============================================|
|  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

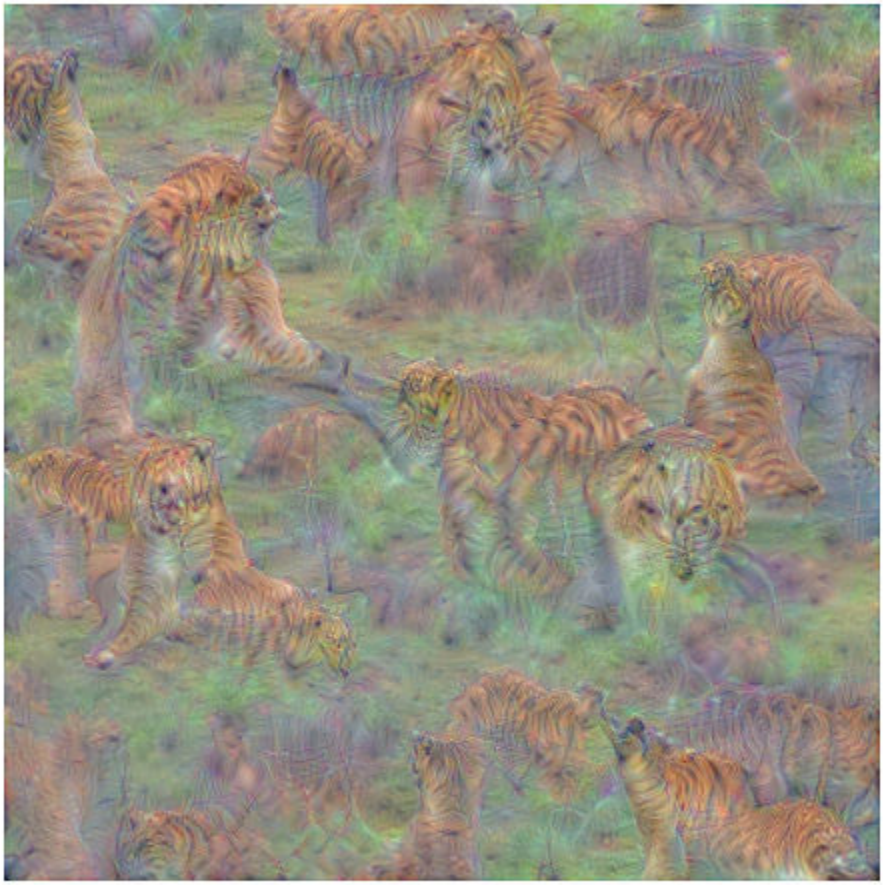
Increasing the number of pyramid levels and iterations per pyramid level can produce more detailed images at the expense of additional computation.

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)
```



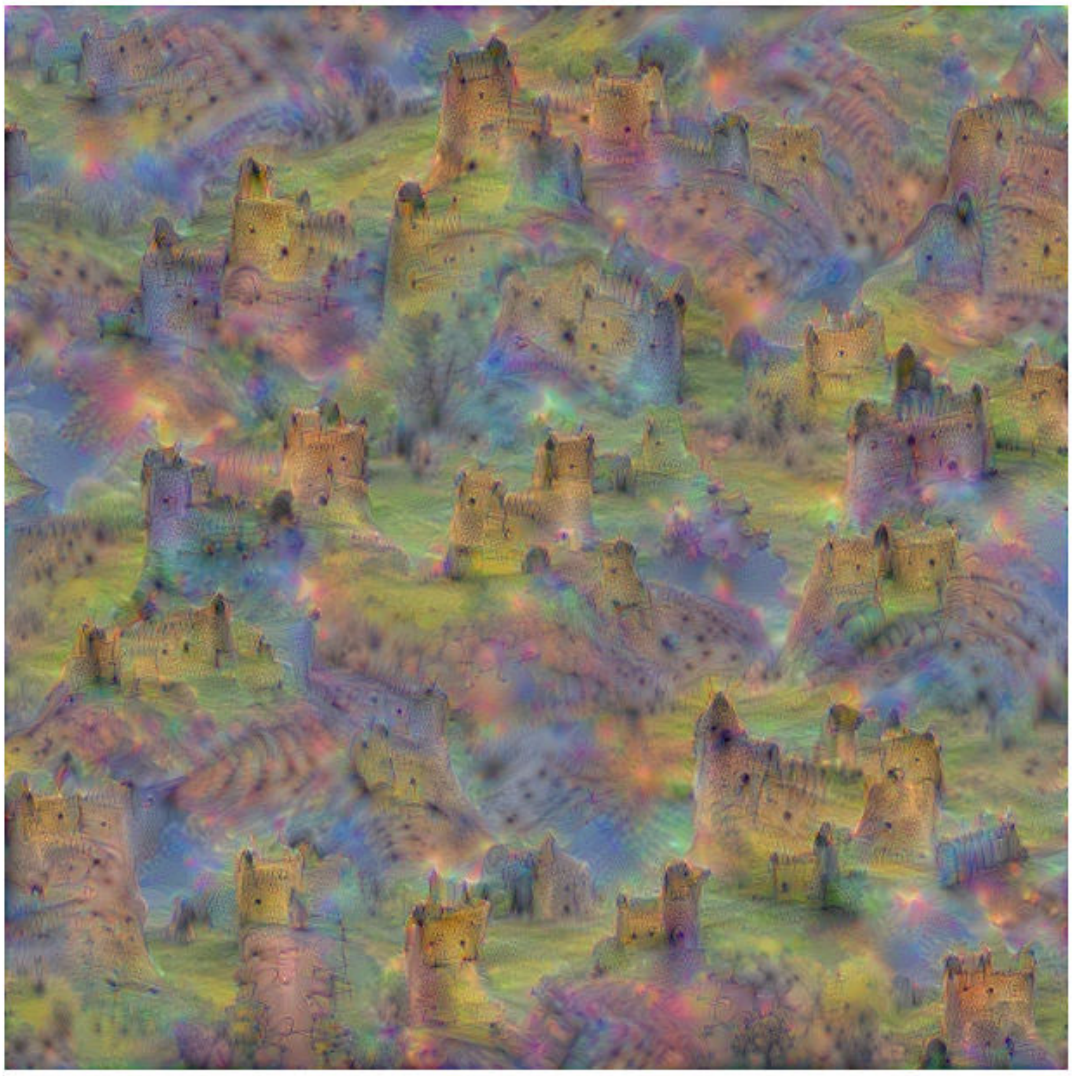
To produce larger and more detailed output images, you can increase both the number of pyramid levels and iterations per pyramid level.

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

deepDreamImage | googlenet

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-12
- “Visualize Activations of a Convolutional Neural Network” on page 5-75
- “Visualize Features of a Convolutional Neural Network” on page 5-90

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. This example shows Grad-CAM using the `dlgradient` automatic differentiation function to perform the required computations easily. For activation mapping techniques on live webcam data, see “Investigate Network Predictions Using Class Activation Mapping” on page 5-57.

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 idea behind Grad-CAM [1] on page 5-0 is to calculate the gradient of the final classification score with respect to the final convolutional feature map. The places where this gradient is large are exactly the places where the final score depends most on the data. The `gradcam` helper function computes the Grad-CAM map for a `dlnetwork`, taking the derivative of the softmax layer score for a given class with respect to a convolutional feature map. For automatic differentiation, the input image `dlImg` must be a `dlarray`.

```
type gradcam.m
```

```
function [featureMap,dScoresdMap] = gradcam(dlnet, dlImg, softmaxName, featureLayerName, classfn)
[scores,featureMap] = predict(dlnet, dlImg, 'Outputs', {softmaxName, featureLayerName});
classScore = scores(classfn);
dScoresdMap = dlgradient(classScore,featureMap);
end
```

The first line of the `gradcam` function obtains the class scores and the feature map from the network. The second line finds the score for the selected classification (golden retriever, in this case). `dlgradient` calculates gradients only for scalar-valued functions. So `gradcam` calculates the gradient of the image score only for the selected classification. The third line uses automatic differentiation to calculate the gradient of the final score with respect to the weights in the feature map layer.

To use Grad-CAM, create a `dlnetwork` from the GoogLeNet network. First, create a layer graph from the network.

```
lgraph = layerGraph(net);
```

To access the data that GoogLeNet uses for classification, remove its final classification layer.

```
lgraph = removeLayers(lgraph, lgraph.Layers(end).Name);
```

Create a `dlnetwork` from the layer graph.

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

Specify the names of the softmax and feature map layers to use with the Grad-CAM helper function. For the feature map layer, specify either the last ReLU layer with non-singleton spatial dimensions, or the last layer that gathers the outputs of ReLU layers (such as a depth concatenation or an addition layer). If your network does not contain any ReLU layers, specify the name of the final convolutional layer that has non-singleton spatial dimensions in the output. Use the function `analyzeNetwork` to examine your network and select the correct layers. For GoogLeNet, the name of the softmax layer is `'prob'` and the depth concatenation layer is `'inception_5b-output'`.

```
softmaxName = 'prob';
featureLayerName = 'inception_5b-output';
```

To use automatic differentiation, convert the sherlock image to a `dlarray`.

```
dlImg = dlarray(single(img),'SSC');
```

Compute the Grad-CAM gradient for the image by calling `dlfeval` on the `gradcam` function.

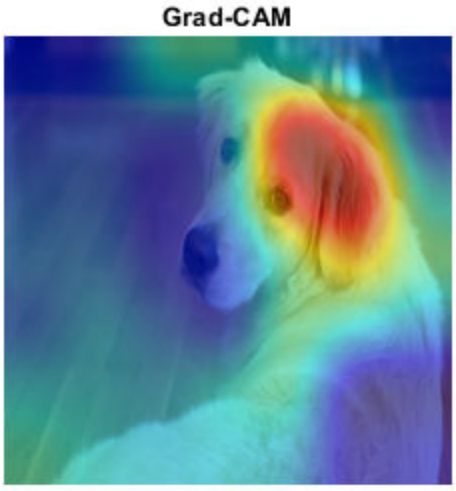
```
[featureMap, dScoresdMap] = dlfeval(@gradcam, dlnet, dlImg, softmaxName, featureLayerName, class
```

Resize the gradient map to the GoogLeNet image size, and scale the scores to the appropriate levels for display.

```
gradcamMap = sum(featureMap .* sum(dScoresdMap, [1 2]), 3);
gradcamMap = extractdata(gradcamMap);
gradcamMap = rescale(gradcamMap);
gradcamMap = imresize(gradcamMap, inputSize, 'Method', 'bicubic');
```

Show the Grad-CAM levels 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(gradcamMap,'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`.

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

`dlarray` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Investigate Network Predictions Using Class Activation Mapping” on page 5-57

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.

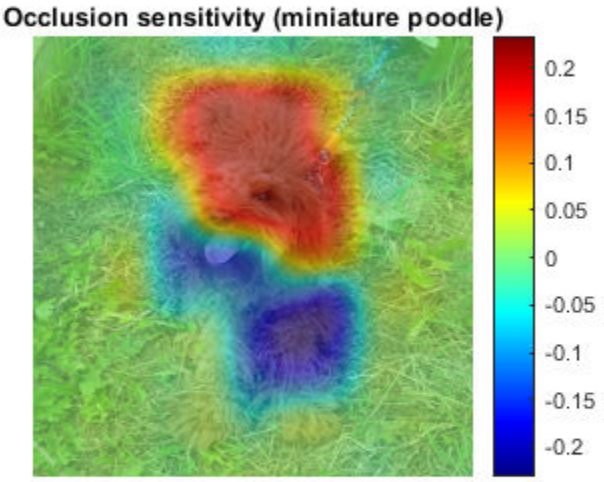
Identify Areas of an Image the Network Uses for Classification

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 5-8.

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

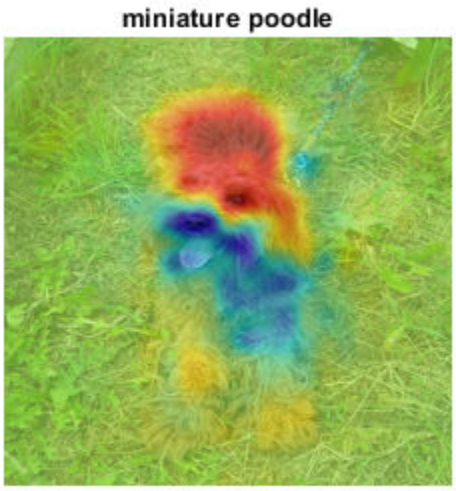
You can use occlusion to compare which parts of the image the network identifies as evidence for different classes. This can be useful in cases where the network is not confident in the classification and gives similar scores to several classes.

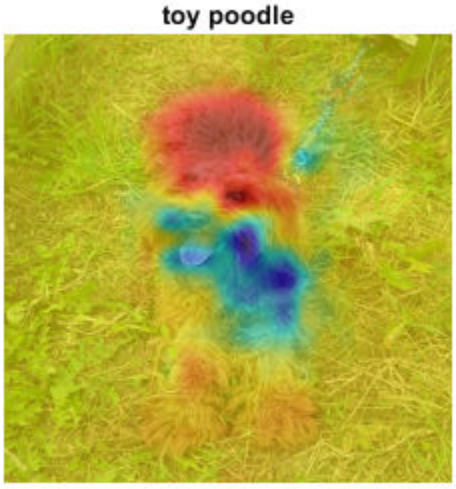
Compute an occlusion map for each of the top three classes. To examine the results of occlusion with higher resolution, reduce the mask size and stride using the `MaskSize` and `Stride` options. A smaller `Stride` leads to a higher-resolution map, but can take longer to compute and use more memory. A 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

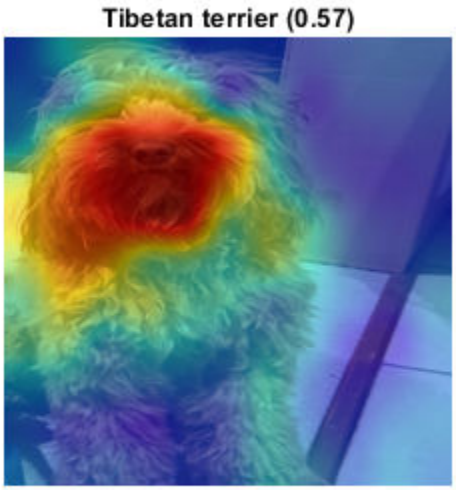
If your network is consistently misclassifying certain types of input data, you can use occlusion sensitivity to determine if particular features of your input data are confusing the network. From the occlusion map of Laika sitting on the grass, you could expect that images of Laika which are more focused 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

- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8
- “Investigate Network Predictions Using Class Activation Mapping” on page 5-57
- “Visualize Features of a Convolutional Neural Network” on page 5-90
- “Visualize Activations of a Convolutional Neural Network” on page 5-75

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 5-12
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8

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 whivh pixels most affect teh 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;
```

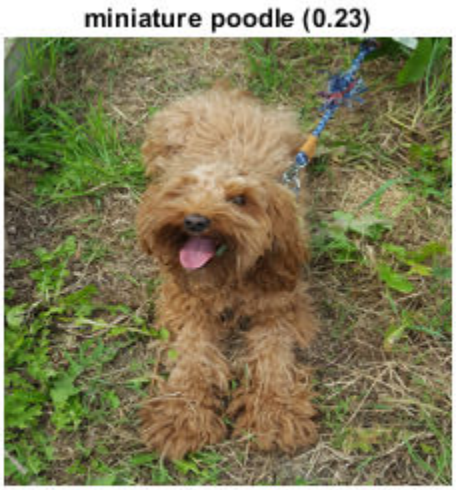
Load the image. The image is of a dog named Laika. Resize the image to the network input size.

```
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');
```

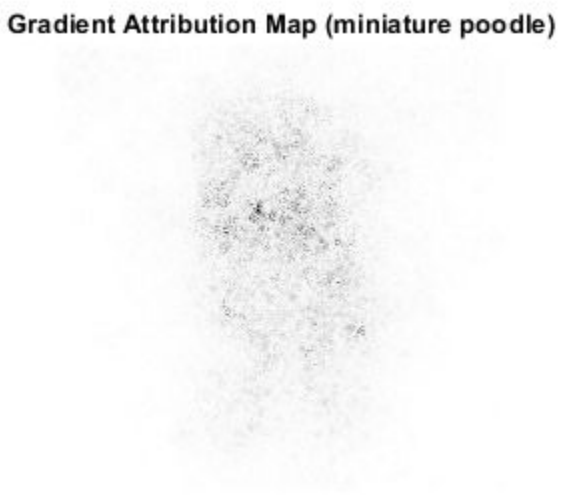
Use `dlfeval` and the `gradientMap` function (defined in the Supporting Functions on page 5-0 section of this example) to compute the derivative $\frac{\partial S^C}{\partial I_{xy}}$. The `gradientMap` function passes the image forward through the network to obtain the class scores and contains a call 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"`.

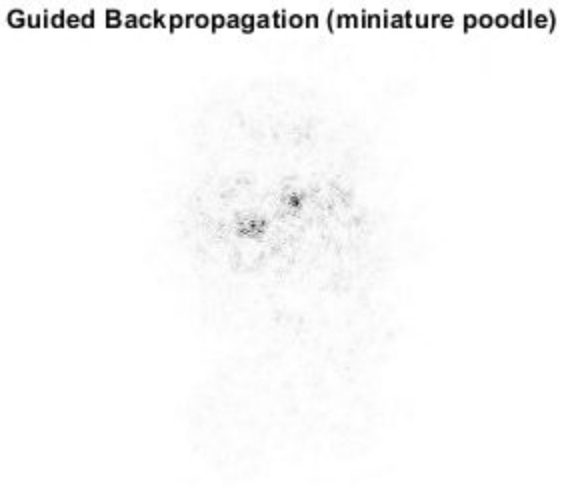
```
customRelu = CustomBackpropReluLayer();
customRelu.BackpropMode = "guided-backprop";

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

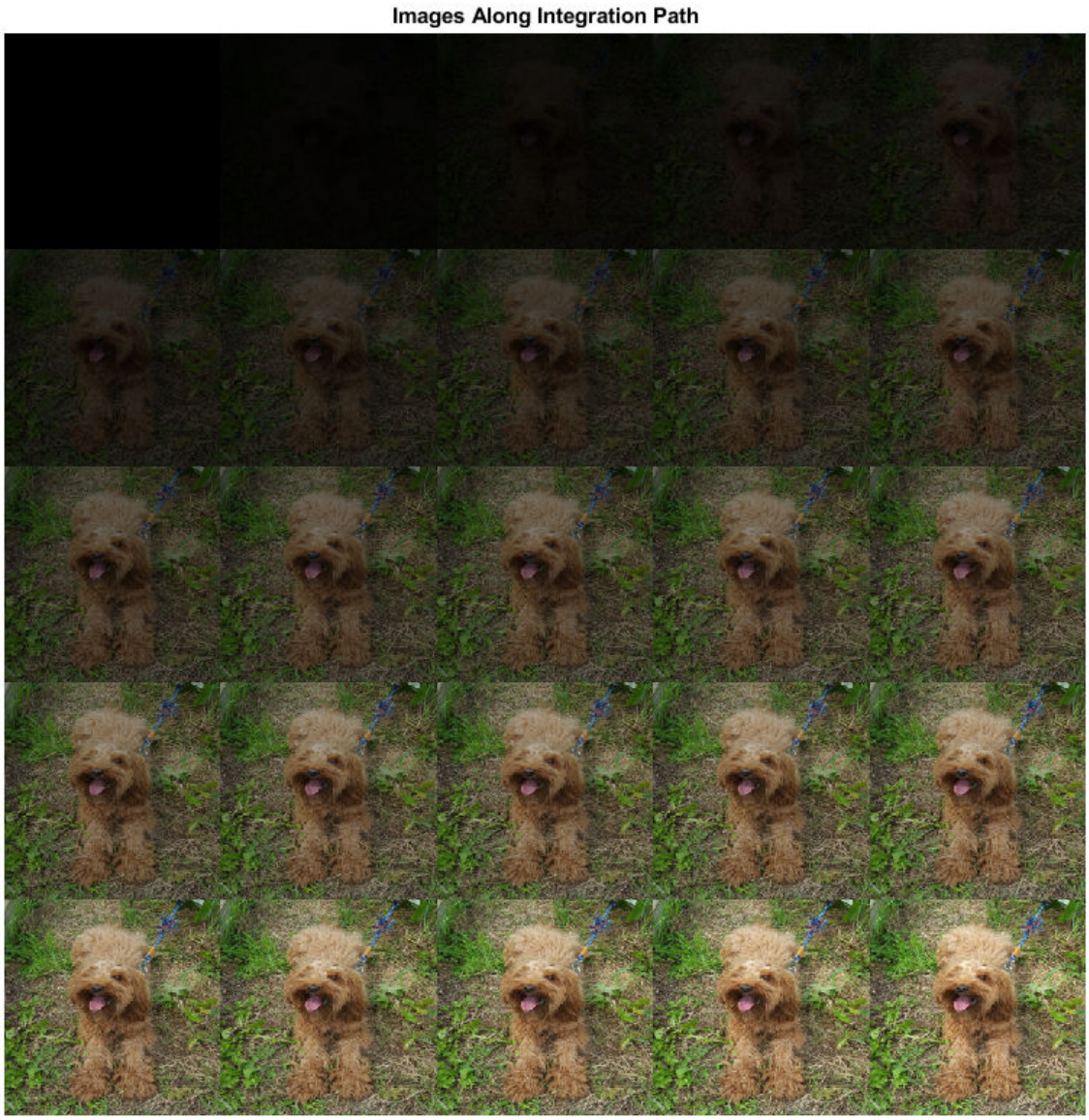
$$I_{\mathrm{xy}}^{n} = I_{\mathrm{xy}}^{0} + \frac{n}{N}\left(I_{\mathrm{xy}} - I_{\mathrm{xy}}^{0}\right).$$

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 = ;

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.

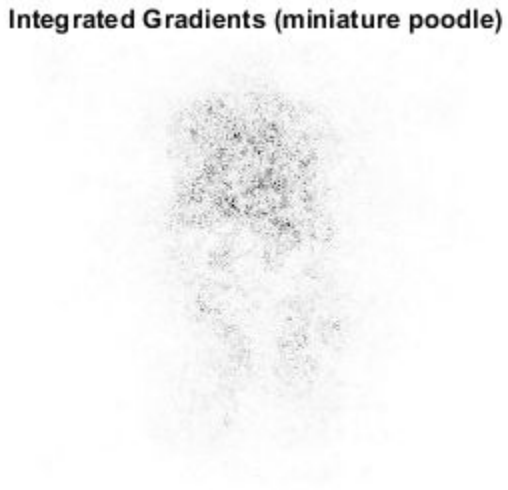
```
dlPathImgs = dlarray(pathImgs, 'SSCB');
dydIIG = dlfeval(@gradientMap, dlnet, dlPathImgs, softmaxName, classIdx);
```

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

`dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `googlenet` | `occlusionSensitivity`

More About

- “Understand Network Predictions Using Occlusion” on page 5-12
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8
- “Investigate Network Predictions Using Class Activation Mapping” on page 5-57
- “Specify Custom Layer Backward Function” on page 15-62

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

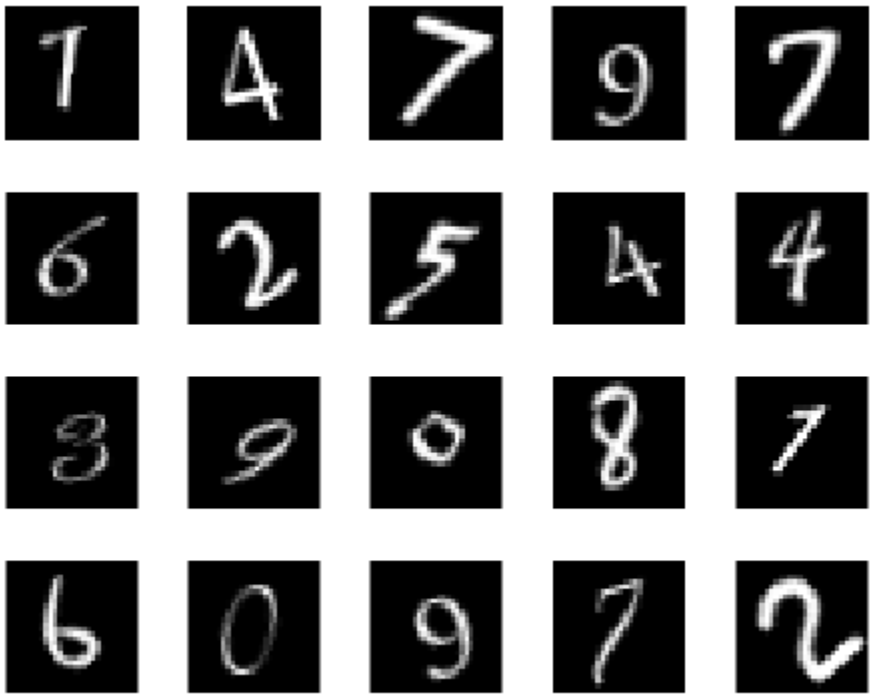
Load the sample data as a 4-D array. `digitTrain4DArrayData` loads the digit training set as 4-D array data. `XTrain` is a 28-by-28-by-1-by-5000 array, where 28 is the height and 28 is the width of the images. 1 is the number of channels and 5000 is the number of synthetic images of handwritten digits. `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];
```

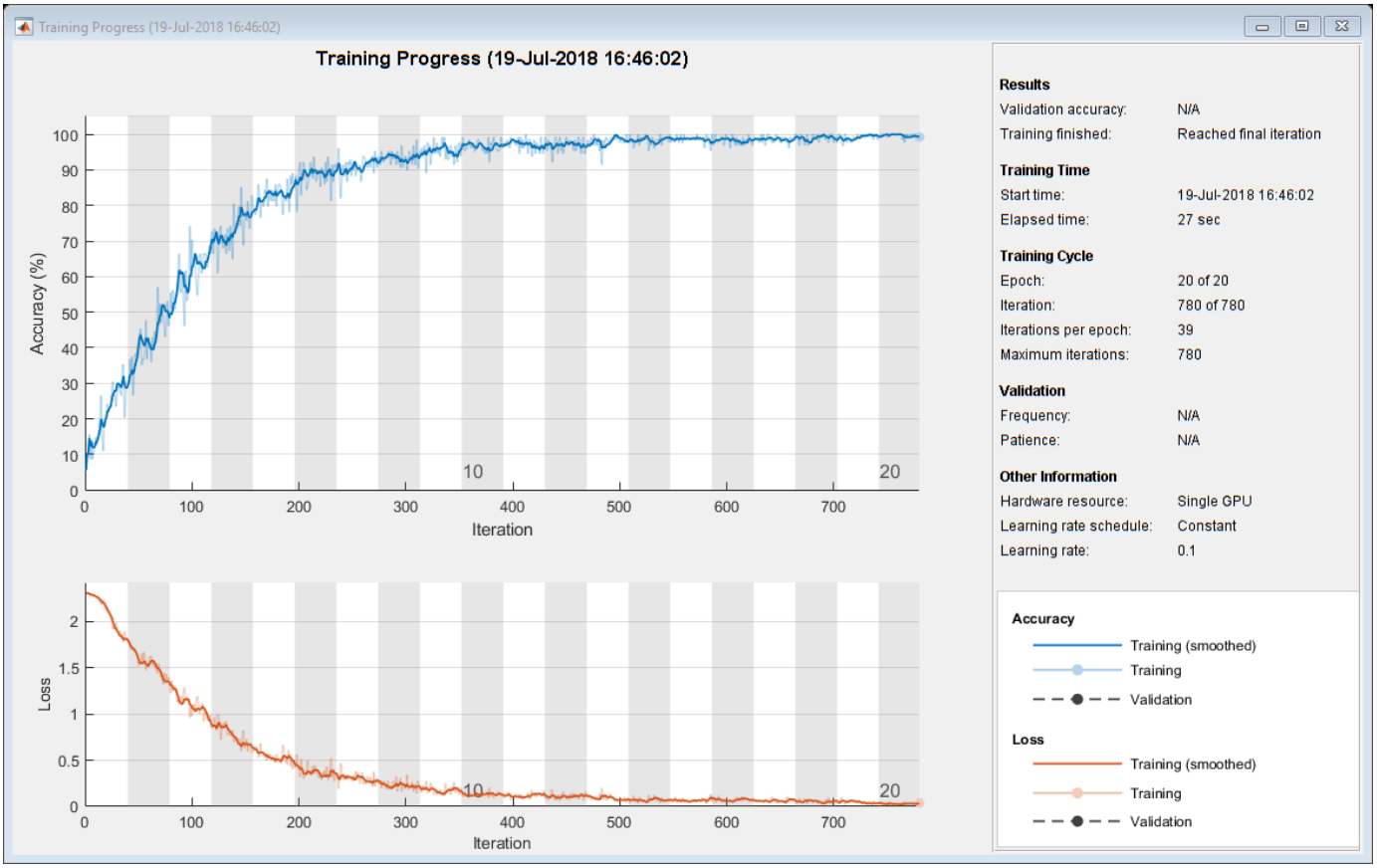
Specify Training Options and Train Network

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.

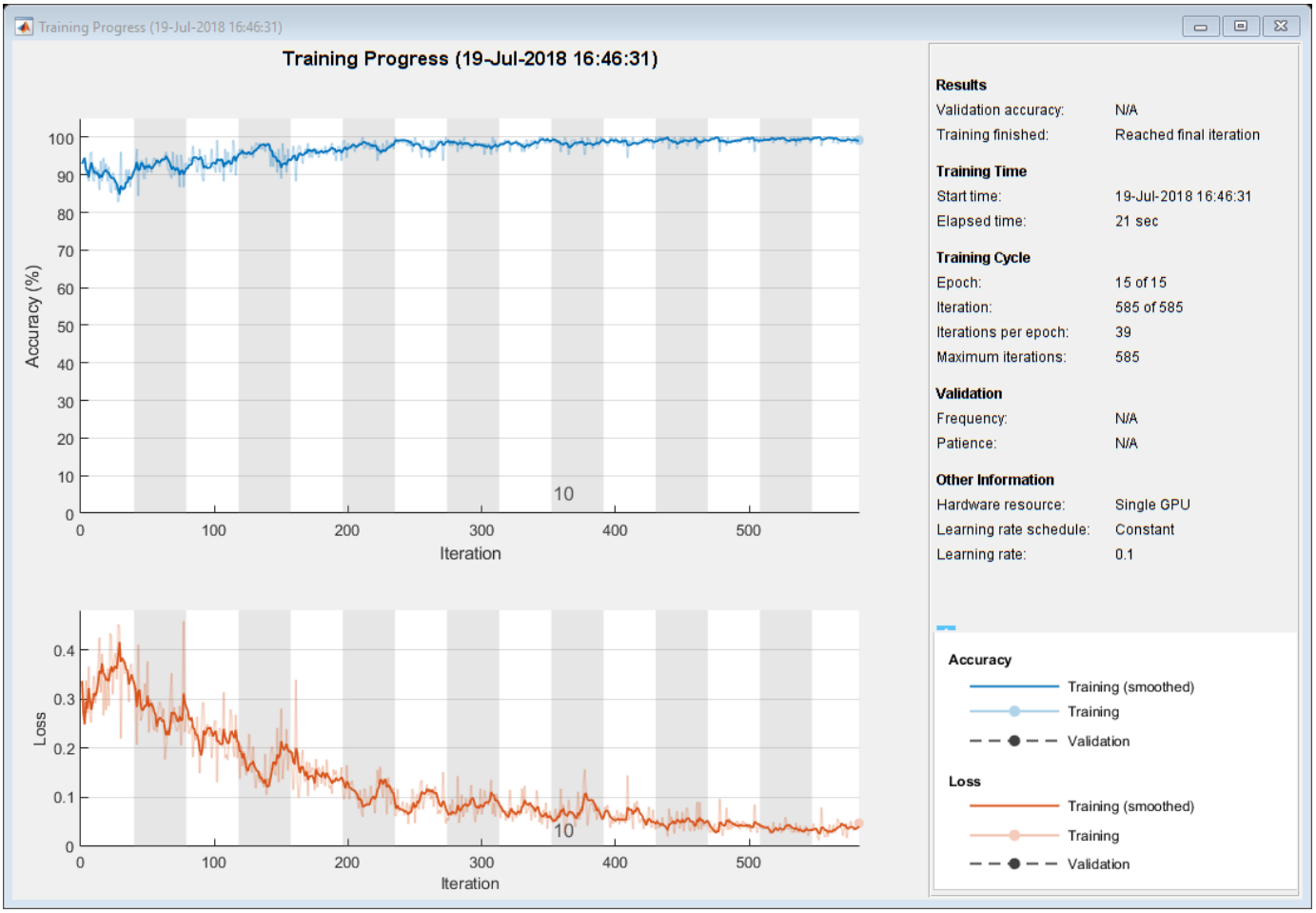
```
load('net_checkpoint__195__2018_07_13__11_59_10.mat','net')
```

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

`trainNetwork` | `trainingOptions`

Related Examples

- “Create Simple Deep Learning Network for Classification” on page 3-40

More About

- “Learn About Convolutional Neural Networks” on page 1-19
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41

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.

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.

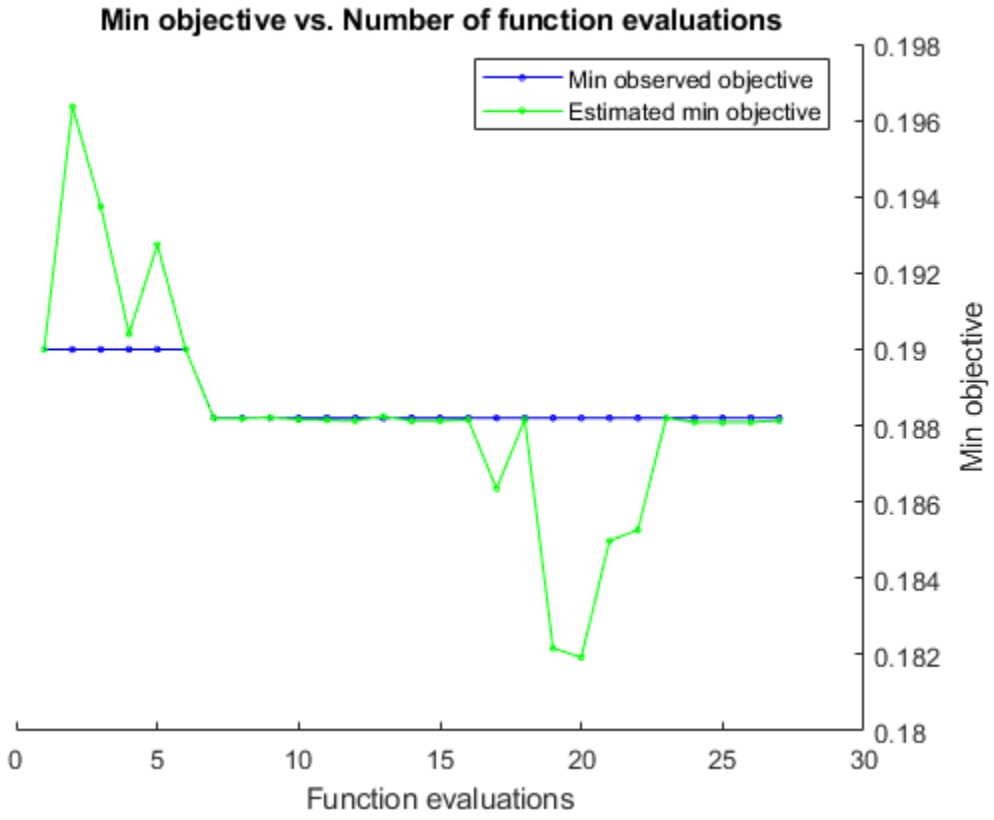
```
ObjFcn = makeObjFcn(XTrain,YTrain,XValidation,YValidation);
```

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   | Objective   | Objective   | BestSoFar   | BestSoFar   | SectionDepth | InitialL
|      | result |             | runtime     | (observed)  | (estim.)    |              | Rate
|=========================================================================================================
|    1 | Best   |        0.19 |        2201 |        0.19 |        0.19 |            3 |     0.01
|    2 | Accept |      0.3224 |      1734.1 |        0.19 |     0.19636 |            1 |     0.00
|    3 | Accept |      0.2076 |      1688.7 |        0.19 |     0.19374 |            2 |     0.02
|    4 | Accept |      0.1908 |      2167.2 |        0.19 |      0.1904 |            3 |      0.9
|    5 | Accept |      0.1972 |      2157.4 |        0.19 |     0.19274 |            3 |      0.2
|    6 | Accept |      0.2594 |      2152.8 |        0.19 |        0.19 |            3 |      0.9
|    7 | Best   |      0.1882 |      2257.5 |      0.1882 |     0.18819 |            3 |        0
|    8 | Accept |      0.8116 |      1989.7 |      0.1882 |     0.18818 |            3 |      0.4
|    9 | Accept |      0.1986 |        1836 |      0.1882 |     0.18821 |            2 |     0.03
|   10 | Accept |      0.2146 |      1909.4 |      0.1882 |     0.18816 |            2 |     0.01
|   11 | Accept |      0.2194 |        1562 |      0.1882 |     0.18815 |            1 |      0.1
|   12 | Accept |      0.2246 |      1591.2 |      0.1882 |     0.18813 |            1 |      0.7
|   13 | Accept |      0.2648 |      1621.8 |      0.1882 |     0.18824 |            1 |     0.01
|   14 | Accept |      0.2222 |        1562 |      0.1882 |     0.18812 |            1 |      0.1
|   15 | Accept |      0.2364 |      1625.7 |      0.1882 |     0.18813 |            1 |     0.07
|   16 | Accept |        0.26 |      1706.2 |      0.1882 |     0.18815 |            1 |     0.01
|   17 | Accept |      0.1986 |      2188.3 |      0.1882 |     0.18635 |            3 |      0.3
|   18 | Accept |      0.1938 |      2169.6 |      0.1882 |     0.18817 |            3 |     0.02
|   19 | Accept |      0.3588 |      1713.7 |      0.1882 |     0.18216 |            1 |     0.01
|   20 | Accept |      0.2224 |      1721.4 |      0.1882 |     0.18193 |            1 |      0.0
|=========================================================================================================
| Iter | Eval   | Objective   | Objective   | BestSoFar   | BestSoFar   | SectionDepth | InitialL
|      | result |             | runtime     | (observed)  | (estim.)    |              | Rate
|=========================================================================================================
|   21 | Accept |      0.1904 |      2184.7 |      0.1882 |     0.18498 |            3 |     0.01
|   22 | Accept |      0.1928 |      2184.4 |      0.1882 |     0.18527 |            3 |      0.0
|   23 | Accept |      0.1934 |      2183.6 |      0.1882 |      0.1882 |            3 |     0.01
|   24 | Accept |       0.303 |      1707.9 |      0.1882 |     0.18809 |            1 |     0.01
|   25 | Accept |       0.194 |      2189.1 |      0.1882 |     0.18808 |            3 |     0.01
|   26 | Accept |      0.2192 |      1752.2 |      0.1882 |     0.18809 |            1 |      0.9
|   27 | Accept |      0.1918 |        2185 |      0.1882 |     0.18813 |            3 |      0.0
```




```
_____________________________________________________________
Optimization completed.
MaxTime of 50400 seconds reached.
Total function evaluations: 27
Total elapsed time: 51962.3666 seconds.
Total objective function evaluation time: 51942.8833

Best observed feasible point:
    SectionDepth    InitialLearnRate    Momentum    L2Regularization
    ____________    ________________    ________    ________________

         3              0.1722          0.8019        4.2149e-06

Observed objective function value = 0.1882
Estimated objective function value = 0.18813
Function evaluation time = 2257.4627

Best estimated feasible point (according to models):
    SectionDepth    InitialLearnRate    Momentum    L2Regularization
    ____________    ________________    ________    ________________

         3              0.1722          0.8019        4.2149e-06

Estimated objective function value = 0.18813
Estimated function evaluation time = 2166.2402
```

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 = 0.1882
```

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)
```

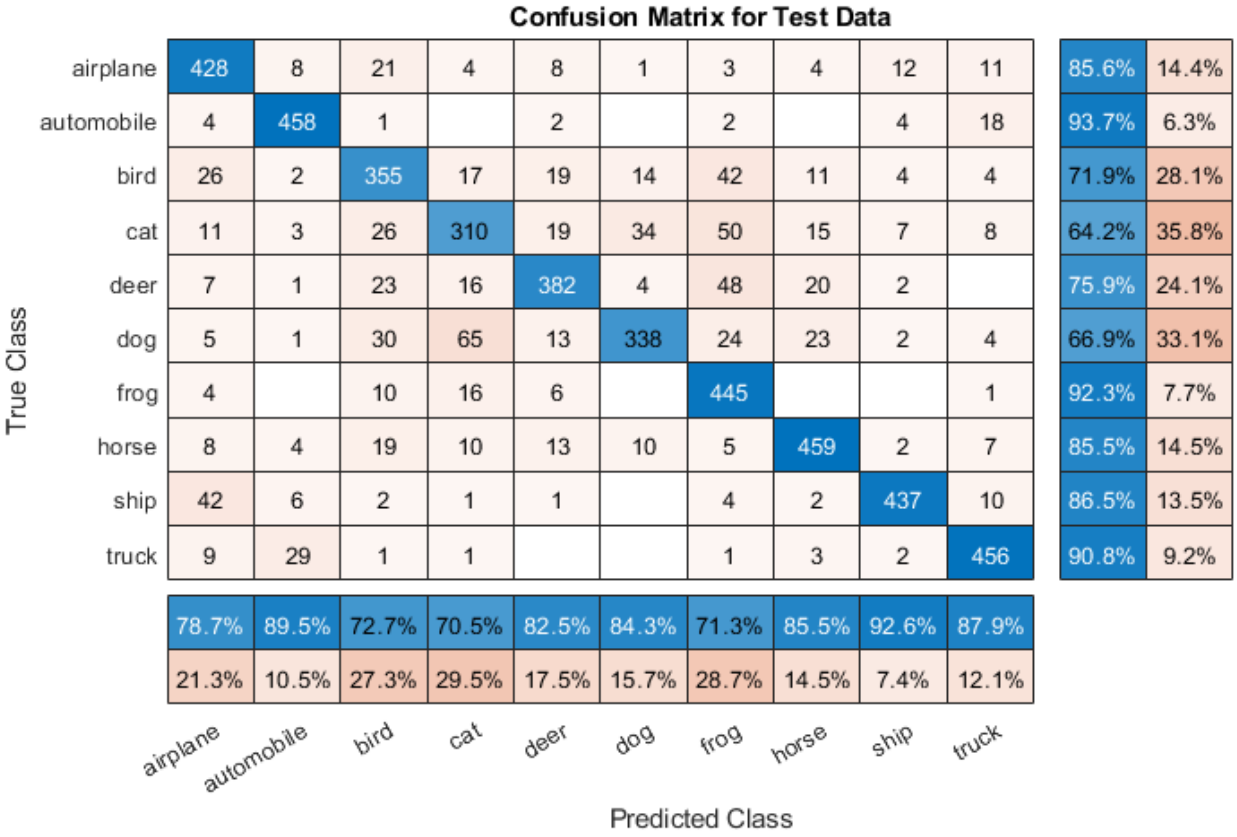
```
testError = 0.1864
```

```
NTest = numel(YTest);
testErrorSE = sqrt(testError*(1-testError)/NTest);
testError95CI = [testError - 1.96*testErrorSE, testError + 1.96*testErrorSE]
```

```
testError95CI = 1×2

    0.1756    0.1972
```

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.

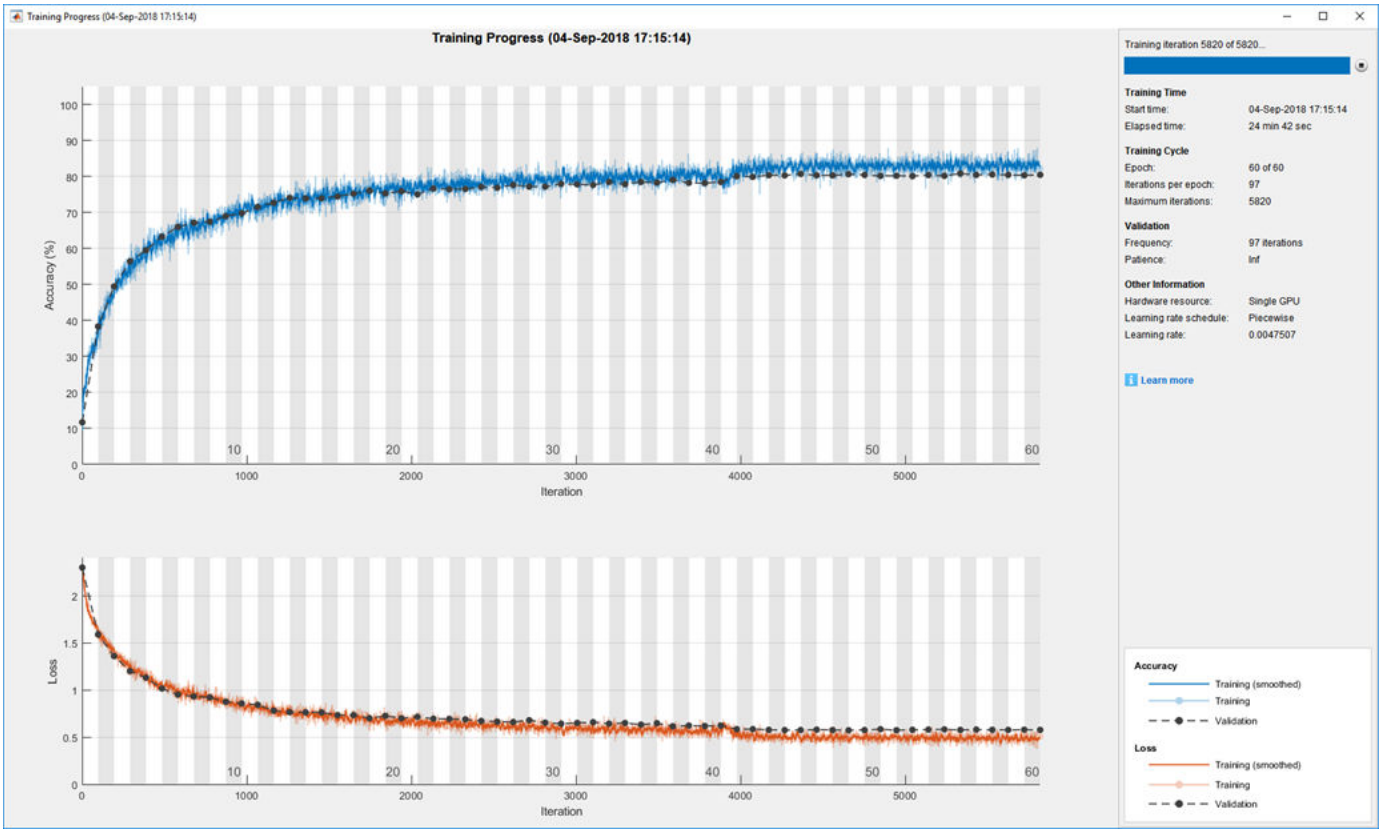
```
miniBatchSize = 256;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'InitialLearnRate',optVars.InitialLearnRate, ...
    'Momentum',optVars.Momentum, ...
    'MaxEpochs',60, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',40, ...
    '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_FIGURE'))
```



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

`bayesopt` | `trainNetwork` | `trainingOptions`

Related Examples

- “Learn About Convolutional Neural Networks” on page 1-19
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Learning in MATLAB” on page 1-2
- “Compare Layer Weight Initializers” on page 15-95
- “Specify Custom Weight Initialization Function” on page 15-89

Run Multiple Deep Learning Experiments 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 interactively run multiple deep learning experiments in serial using Experiment Manager app. For more information, see Experiment Manager.

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” (Parallel Computing Toolbox).
- 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` 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 = 2;
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` 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” (Parallel Computing Toolbox).

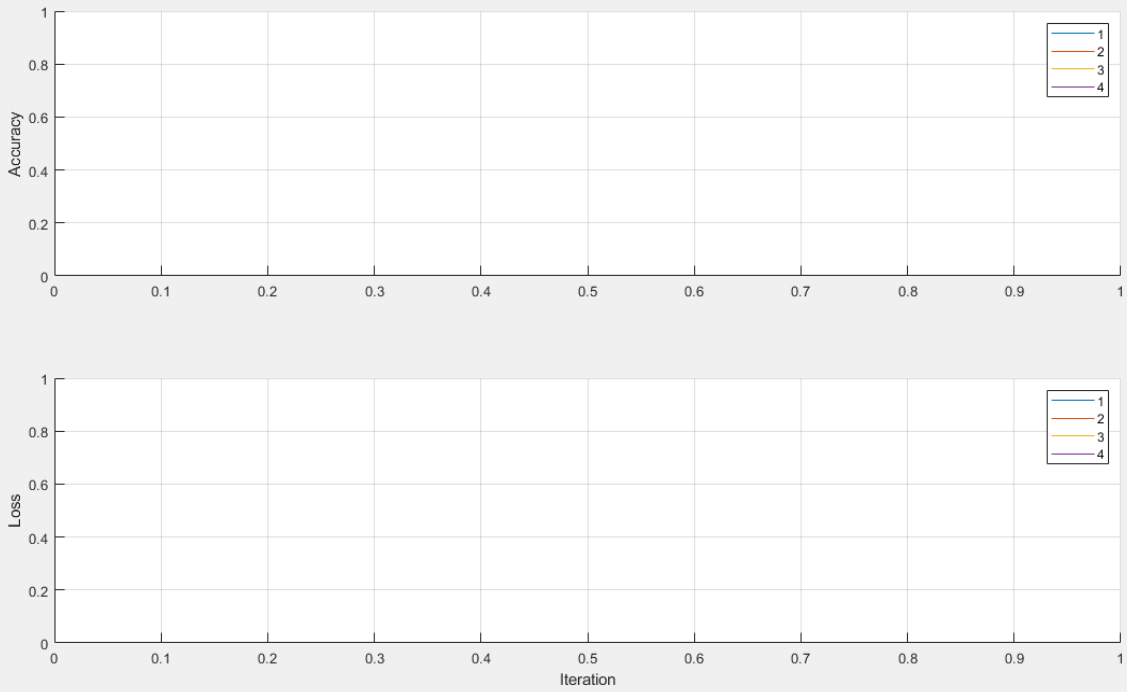
```
dataqueue = parallel.pool.DataQueue;
```

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;
```

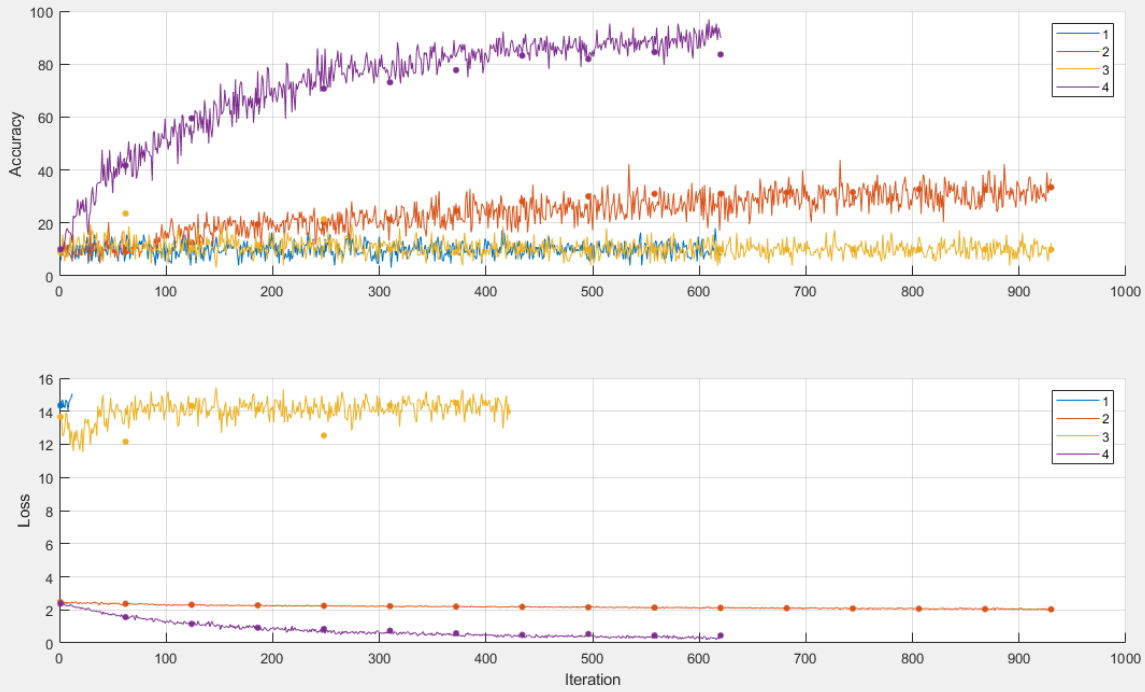
Loop through the network layers and options by using a `for` loop, and use `parfeval` to train the networks on a parallel worker. To request two output arguments from `trainNetwork`, specify `2` as the second input argument to `parfeval`.

```
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.

```
save(['experiment-' datestr(now,'yyyymmddTHHMMSS')],'network','trainingInfo');
```

Plot Results

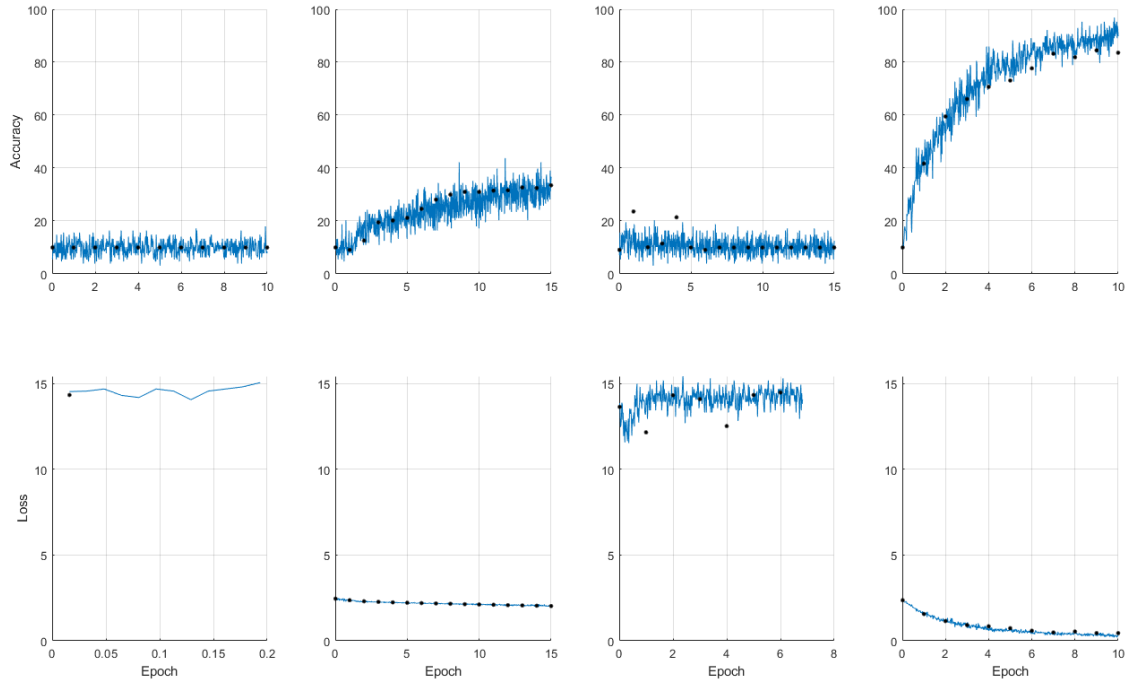
After the networks complete training, plot their training progress by using the information in `trainingInfo`.

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

`augmentedImageDatastore` | `imageDatastore` | `parfeval` | `fetchOutputs` | `trainNetwork` | `trainingOptions`

Related Examples

- “Train Network Using Automatic Multi-GPU Support” (Parallel Computing Toolbox)
- “Use parfeval to Train Multiple Deep Learning Networks” (Parallel Computing Toolbox)

More About

- “Scale Up Deep Learning in Parallel 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 1-38.

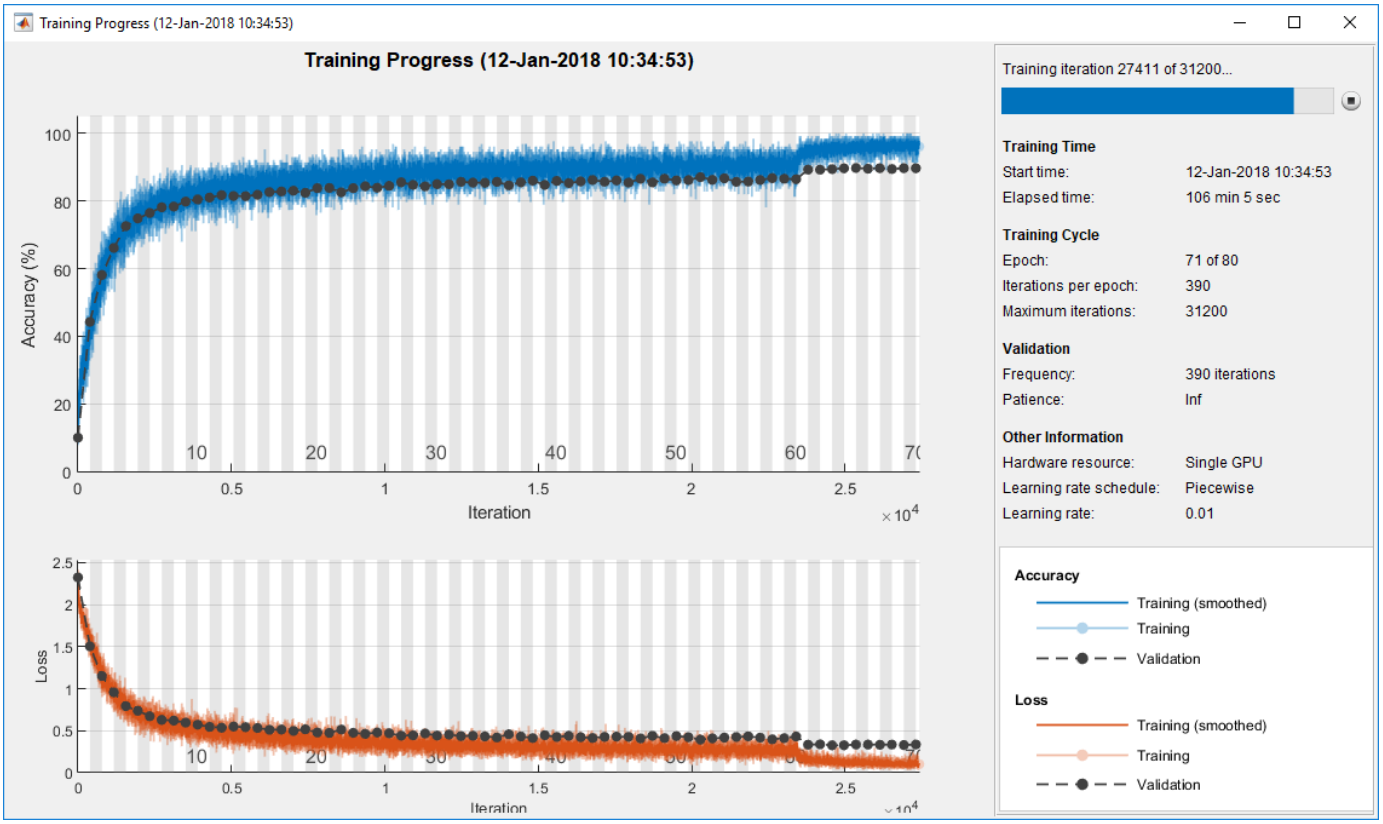
For regression networks, the figure plots the root mean square error (RMSE) instead of the accuracy.

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 final validation accuracy and the reason that training finished. The final validation metrics are labeled **Final** in the plots. If your network contains batch normalization layers, then the final validation metrics are often different from the validation metrics evaluated during training. This is because batch normalization layers in the final network perform different operations than during training.

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 1-41.



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
```

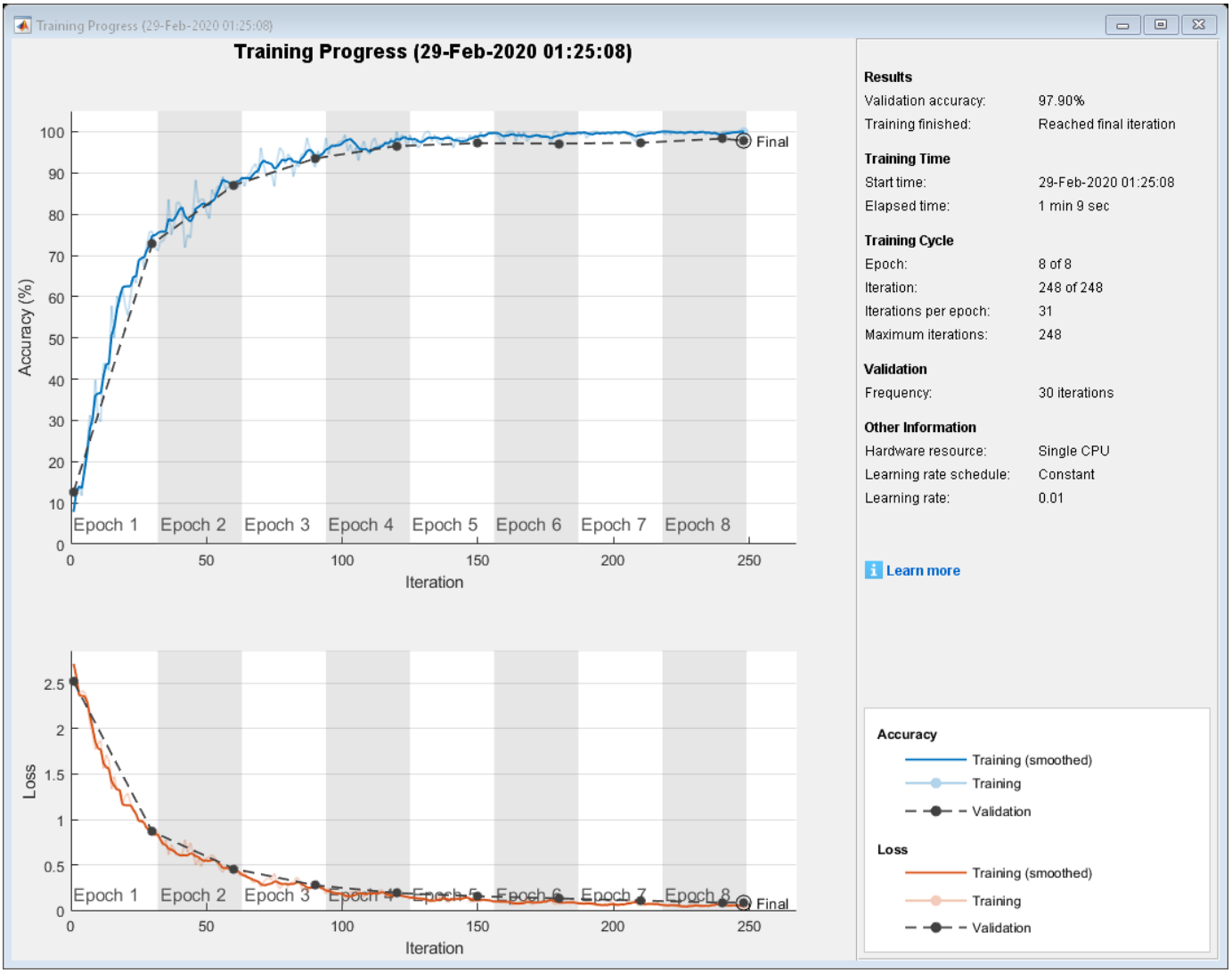
```
    reluLayer

    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 1-19
- “Specify Layers of Convolutional Neural Network” on page 1-30
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Pretrained Deep Neural Networks” on page 1-12
- “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.

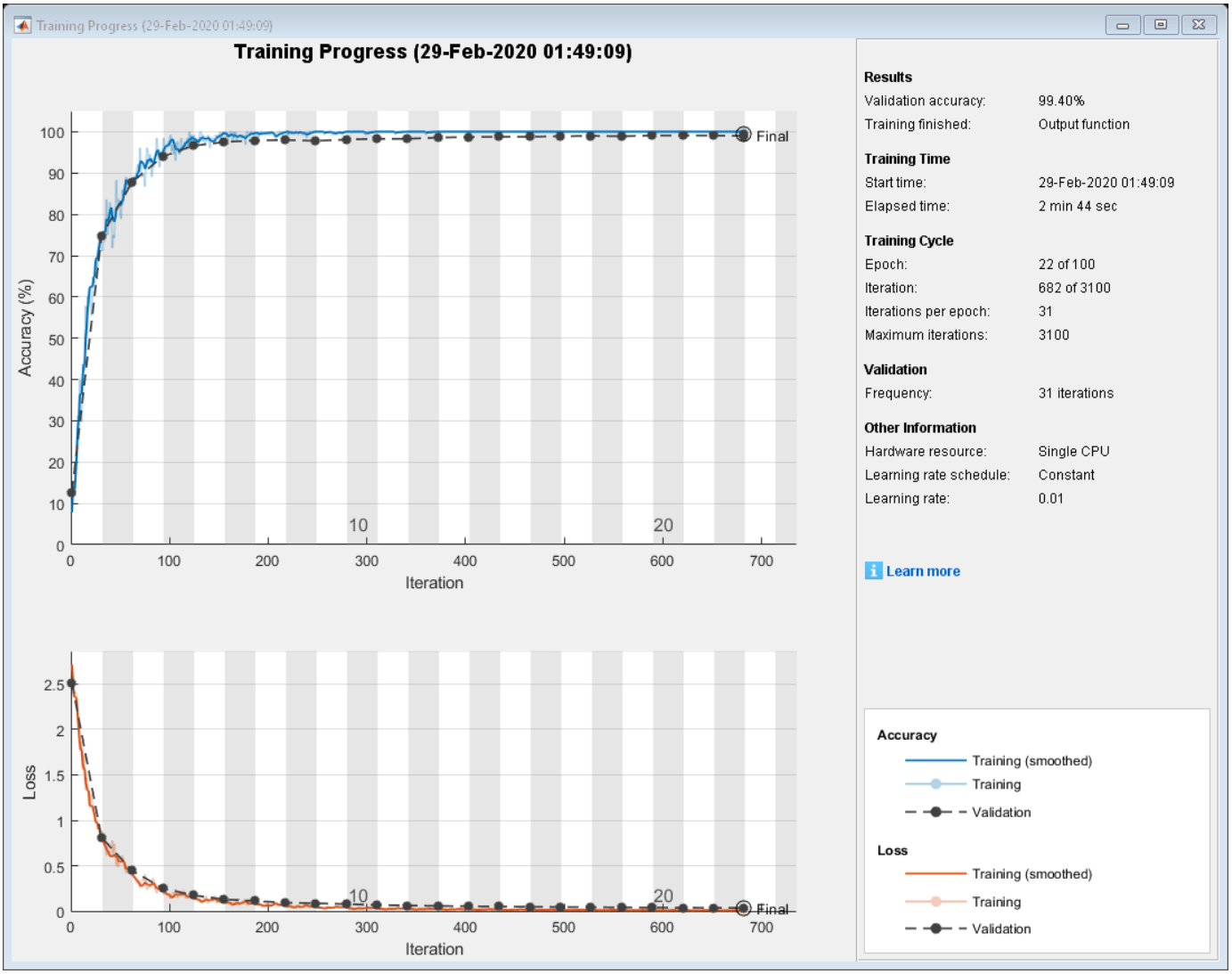
```
miniBatchSize = 128;
validationFrequency = floor(numel(YTrain)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',100, ...
    'MiniBatchSize',miniBatchSize, ...
    'VerboseFrequency',validationFrequency, ...
    'ValidationData',{XValidation,YValidation}, ...
    '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 |       00:00:02 |        7.81% |       12.70% |       2.7155 |       2.5
|       1 |          31 |       00:00:08 |       71.88% |       74.70% |       0.8804 |       0.8
|       2 |          62 |       00:00:17 |       86.72% |       87.80% |       0.3929 |       0.4
|       3 |          93 |       00:00:24 |       94.53% |       94.00% |       0.2230 |       0.2
|       4 |         124 |       00:00:30 |       96.09% |       96.60% |       0.1482 |       0.1
|       5 |         155 |       00:00:39 |       99.22% |       97.50% |       0.1017 |       0.1
|       6 |         186 |       00:00:46 |       99.22% |       97.90% |       0.0783 |       0.1
|       7 |         217 |       00:00:54 |      100.00% |       98.00% |       0.0558 |       0.0
|       8 |         248 |       00:01:01 |      100.00% |       97.80% |       0.0441 |       0.0
|       9 |         279 |       00:01:07 |      100.00% |       98.10% |       0.0349 |       0.0
|      10 |         310 |       00:01:15 |      100.00% |       98.30% |       0.0275 |       0.0
|      11 |         341 |       00:01:22 |      100.00% |       98.30% |       0.0242 |       0.0
|      12 |         372 |       00:01:29 |      100.00% |       98.60% |       0.0217 |       0.0
```

```
|      13 |         403 |       00:01:35 |      100.00% |       98.70% |       0.0191 |       0.0
|      14 |         434 |       00:01:43 |      100.00% |       98.80% |       0.0167 |       0.0
|      15 |         465 |       00:01:49 |      100.00% |       98.80% |       0.0145 |       0.0
|      16 |         496 |       00:01:57 |      100.00% |       98.90% |       0.0127 |       0.0
|      17 |         527 |       00:02:04 |      100.00% |       99.00% |       0.0112 |       0.0
|      18 |         558 |       00:02:10 |      100.00% |       98.90% |       0.0101 |       0.0
|      19 |         589 |       00:02:17 |      100.00% |       99.10% |       0.0092 |       0.0
|      20 |         620 |       00:02:24 |      100.00% |       99.10% |       0.0086 |       0.0
|      21 |         651 |       00:02:37 |      100.00% |       99.10% |       0.0080 |       0.0
|      22 |         682 |       00:02:43 |      100.00% |       99.00% |       0.0076 |       0.0
|=========================================================================================================
```



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 1-19
- “Set Up Parameters and Train Convolutional Neural Network” on page 1-41
- “Deep Learning in MATLAB” on page 1-2
- “Compare Layer Weight Initializers” on page 15-95
- “Specify Custom Weight Initialization Function” on page 15-89

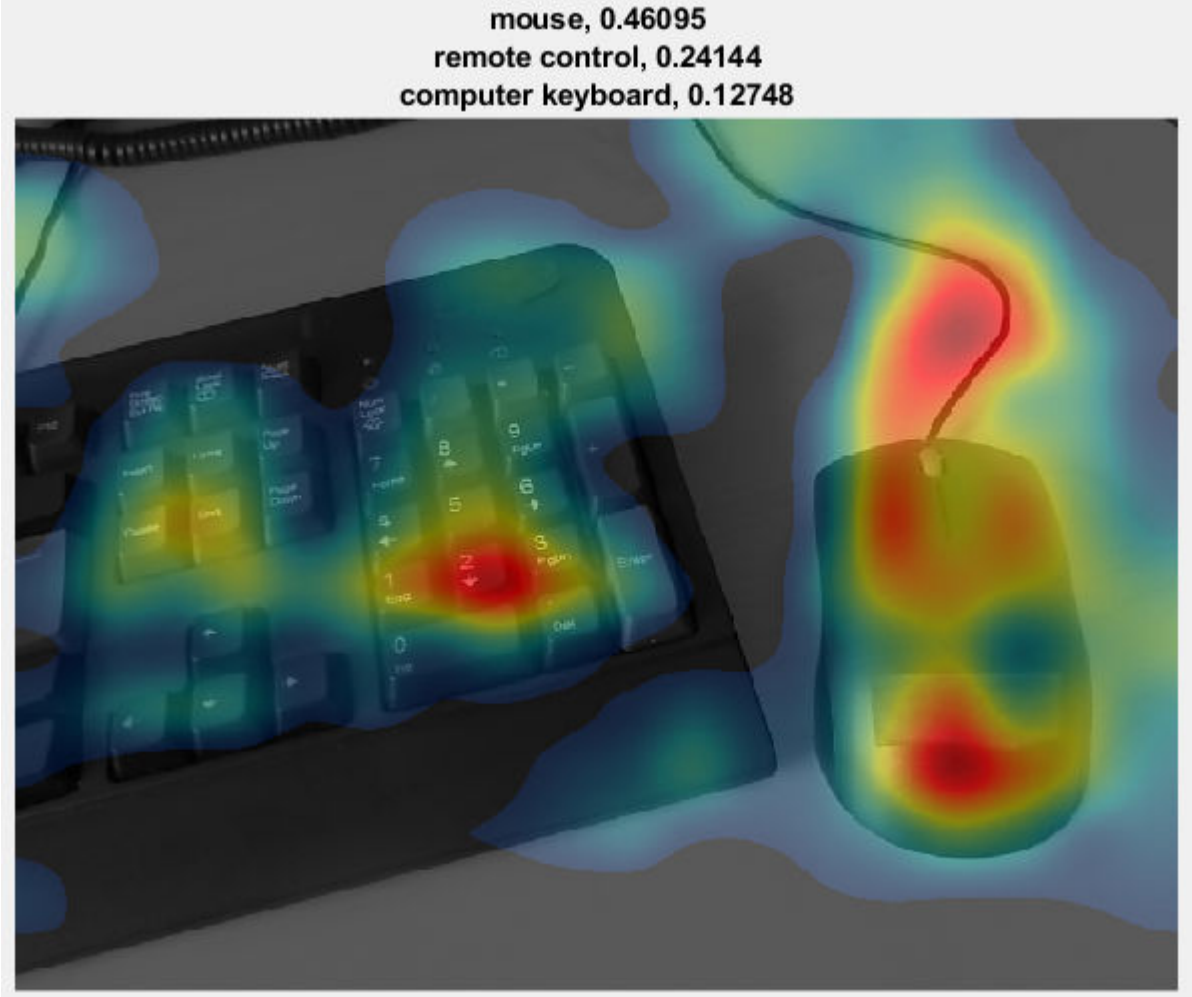
Investigate Network Predictions Using Class Activation Mapping

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 = ;
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.

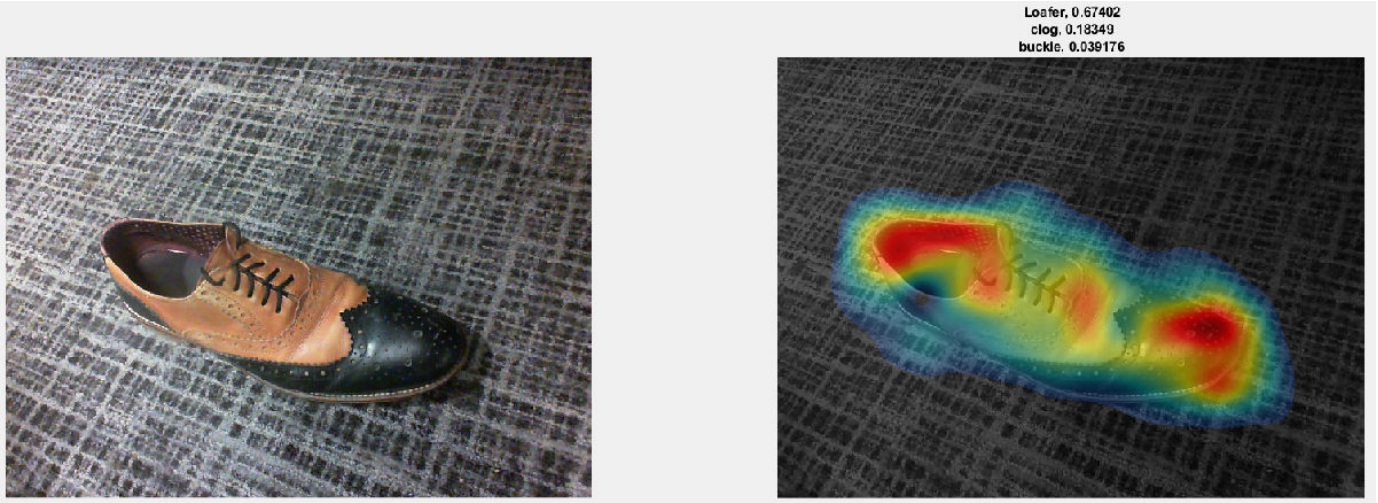
```
    subplot(1,2,1)
    imshow(im)

    subplot(1,2,2)
    CAMshow(im,classActivationMap)
    title(string(labels) + ", " + string(maxScores));

    drawnow

end
```

Clear the webcam object.

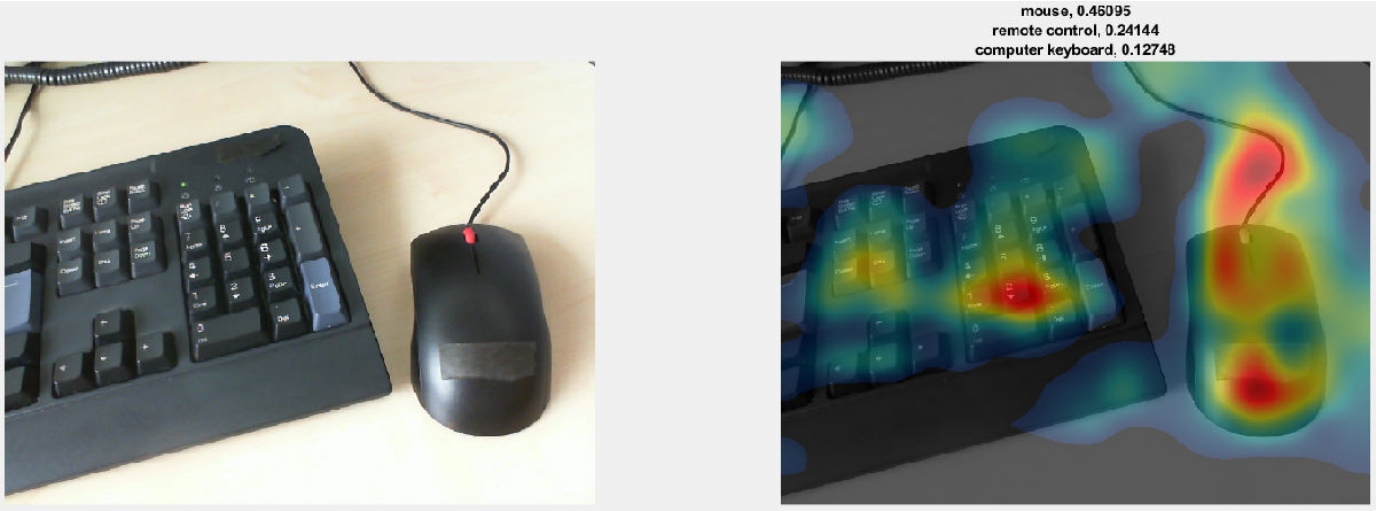
```
clear camera
```

Example Maps

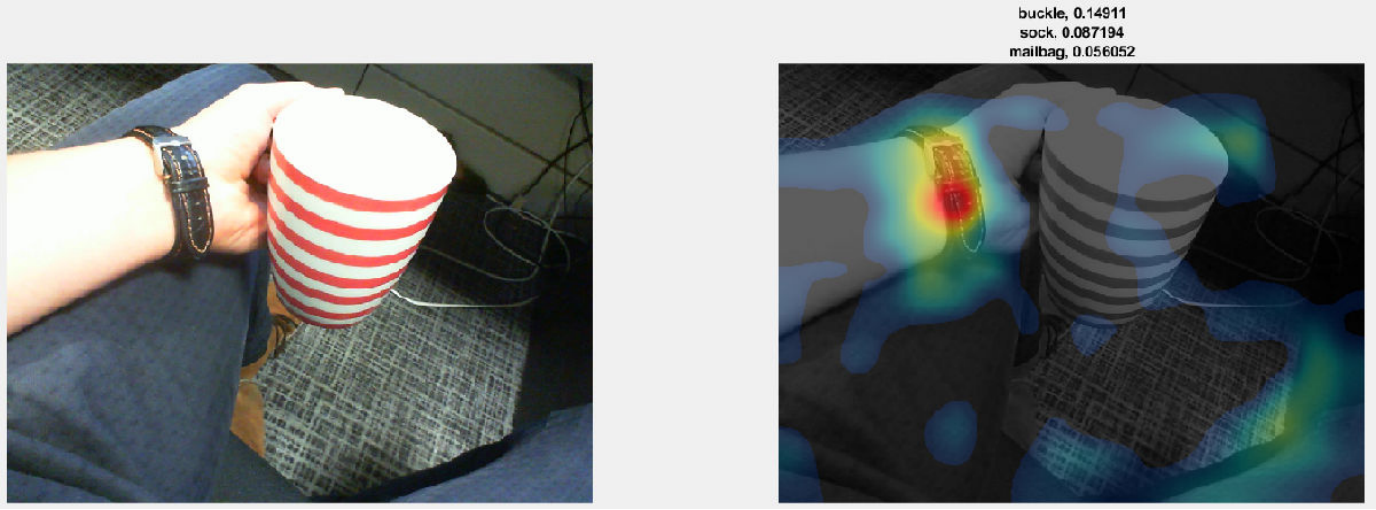
The network correctly identifies the object in this image as a loafer (a type of shoe). The class activation map in the image to the right shows the contribution of each region of the input image to the predicted class `Loafer`. Red regions contribute the most. The network bases its classification on 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`

Related Examples

- “Pretrained Deep Neural Networks” on page 1-12
- “Visualize Activations of a Convolutional Neural Network” on page 5-75
- “Deep Dream Images Using GoogLeNet” on page 5-2
- “Deep Learning in MATLAB” on page 1-2

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` 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"));
```

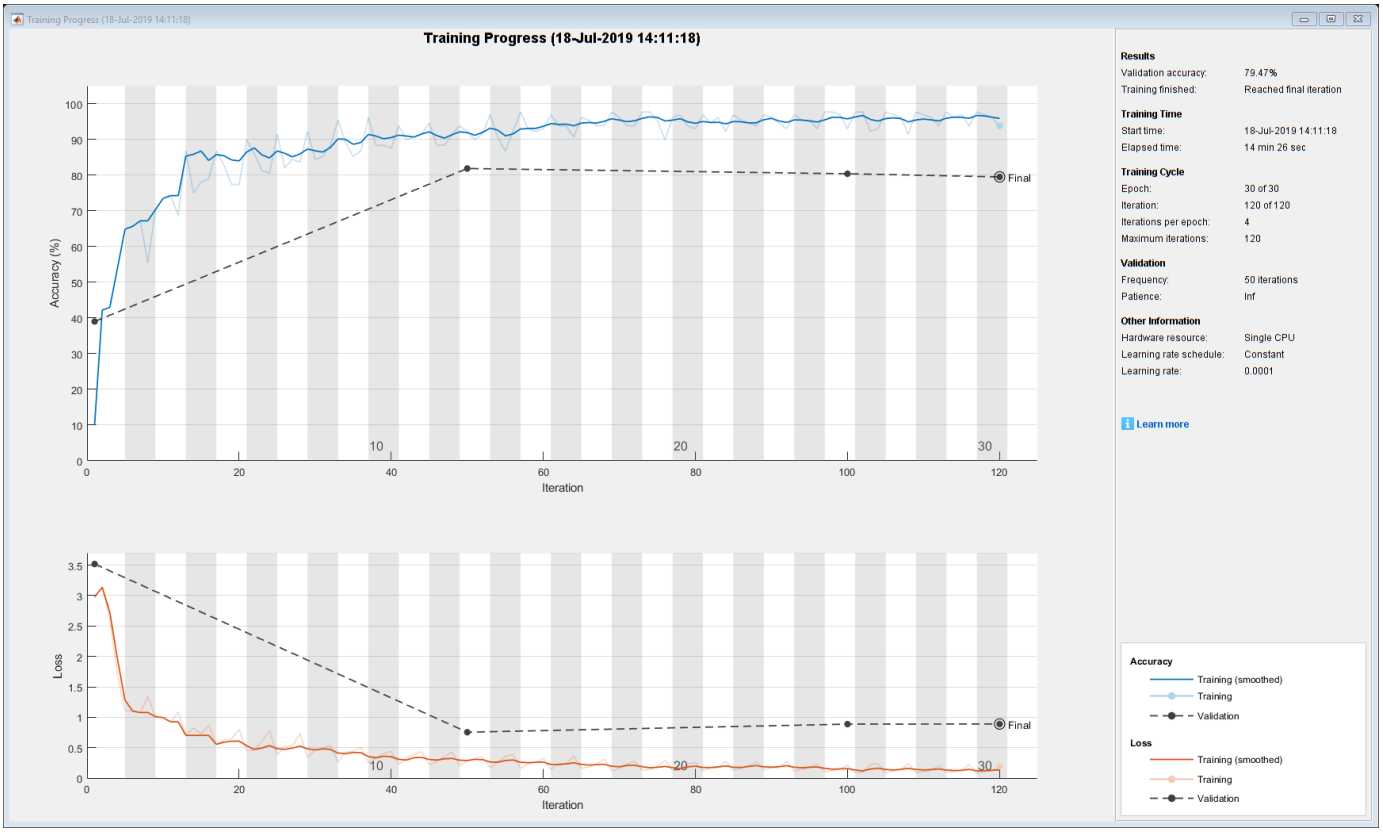
```
newConv =  convolution2dLayer([14 14], 9, "Name", "conv", "Padding", "same");
lgraph = lgraph.replaceLayer("conv10", newConv);
```

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.

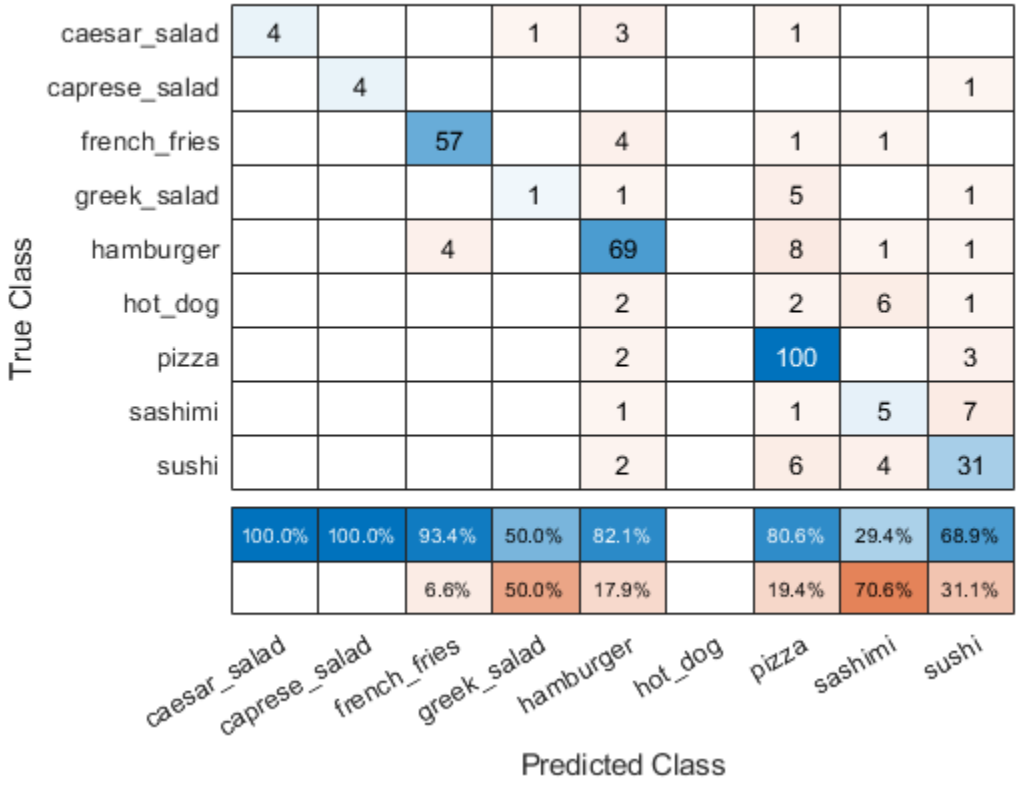
```
opts = trainingOptions("adam", ...
    "InitialLearnRate", 1e-4, ...
    "MaxEpochs", 30, ...
    "ValidationData", augImdsVal, ...
    "Verbose", false,...
    "Plots", "training-progress", ...
    "ExecutionEnvironment","cpu",...
    "MiniBatchSize",128);
rng default
net = trainNetwork(augImdsTrain, 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

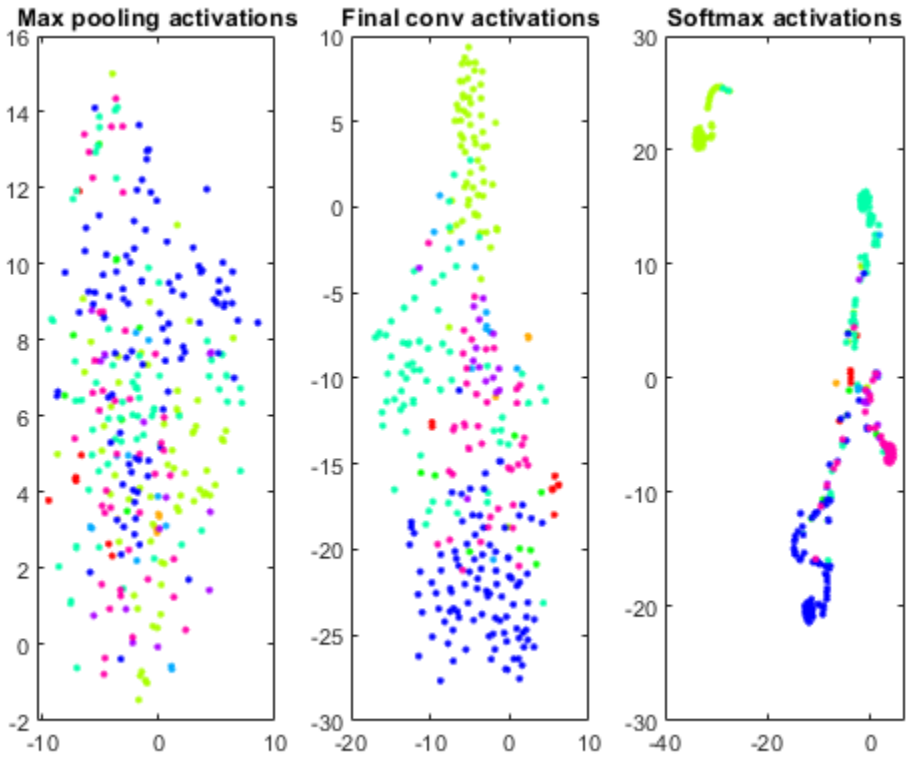
The t-SNE technique tries to preserve distances so that points near each other in the high-dimensional representation are also near each other in the low-dimensional representation. As shown in the confusion matrix, the network is effective at classifying into different classes. Therefore, images that 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");
```

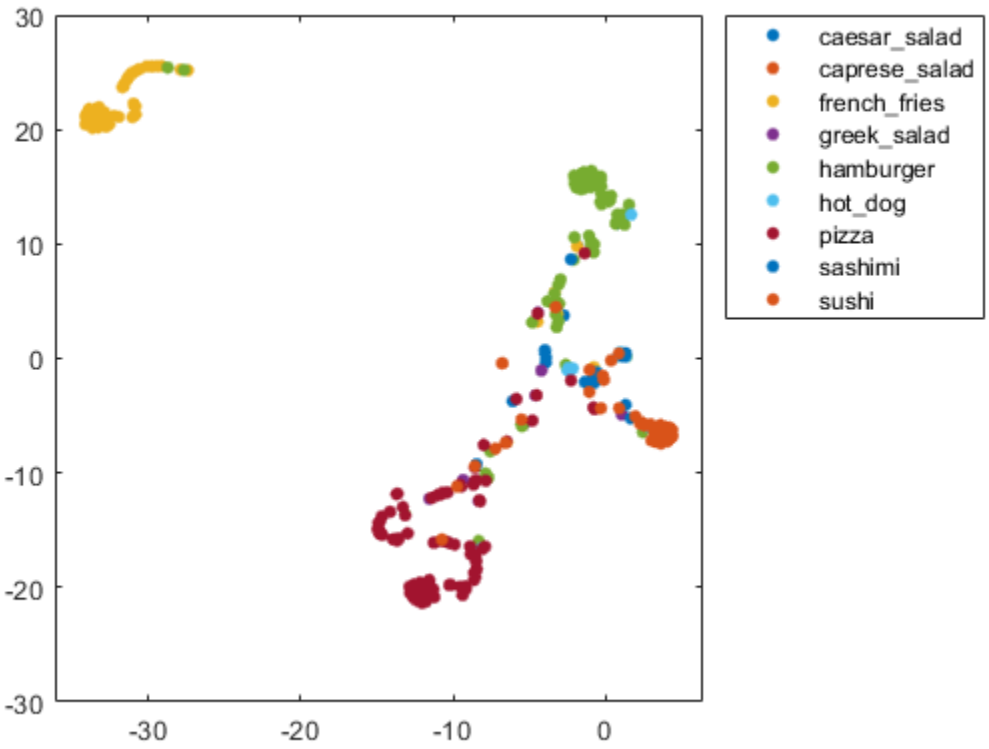


Explore Observations in t-SNE Plot

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";
```



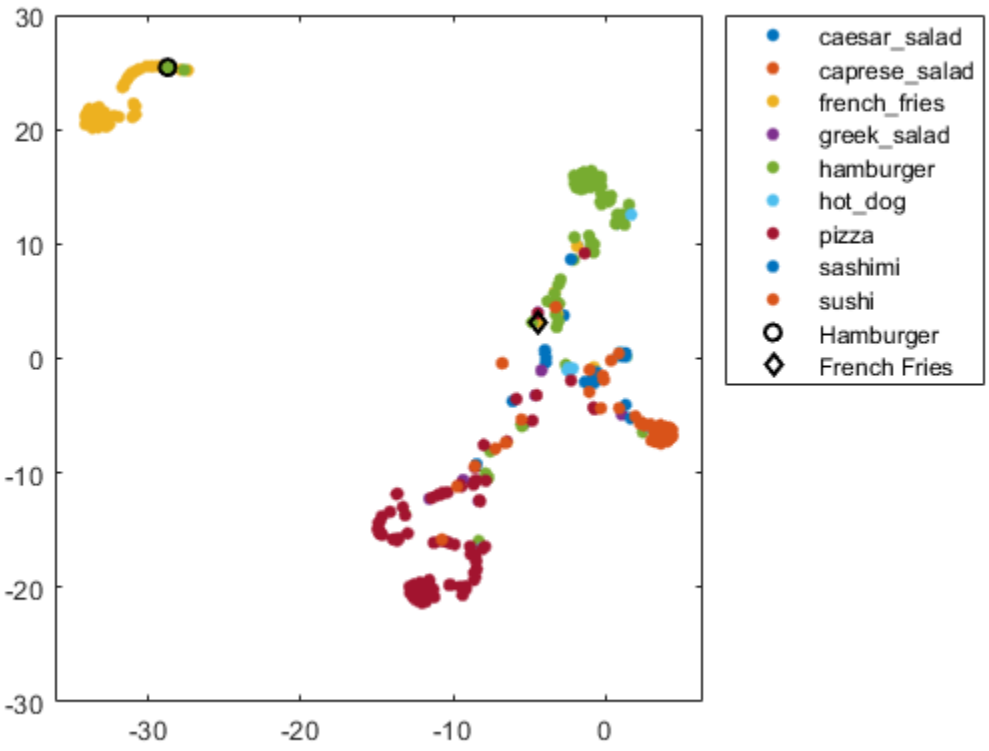
You can also use t-SNE to determine which images are misclassified by the network and why. Incorrect observations are often isolated points of the wrong color for their surrounding cluster. For example, a misclassified image of hamburger is very near the French fries region (the green dot nearest the 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 = ;
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 = ;
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 5-8.

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

`activations` | `classify` | `layerGraph` | `occlusionSensitivity` | `squeezenet` | `trainNetwork` | `trainingOptions` | `tsne`

More About

- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8
- “Investigate Network Predictions Using Class Activation Mapping” on page 5-57
- “Visualize Features of a Convolutional Neural Network” on page 5-90
- “Visualize Activations of a Convolutional Neural Network” on page 5-75

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.

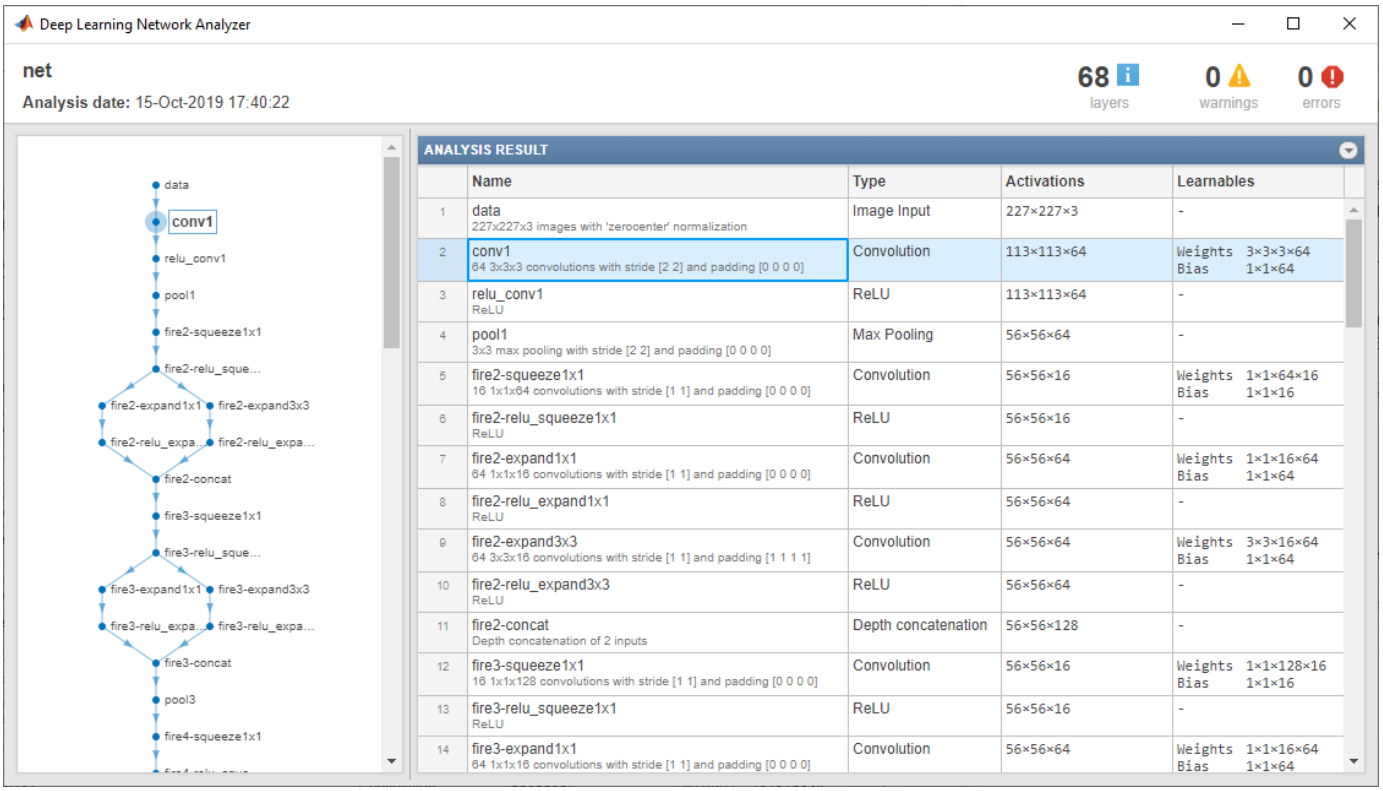
```
im = imread('face.jpg');
imshow(im)
```

```
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.

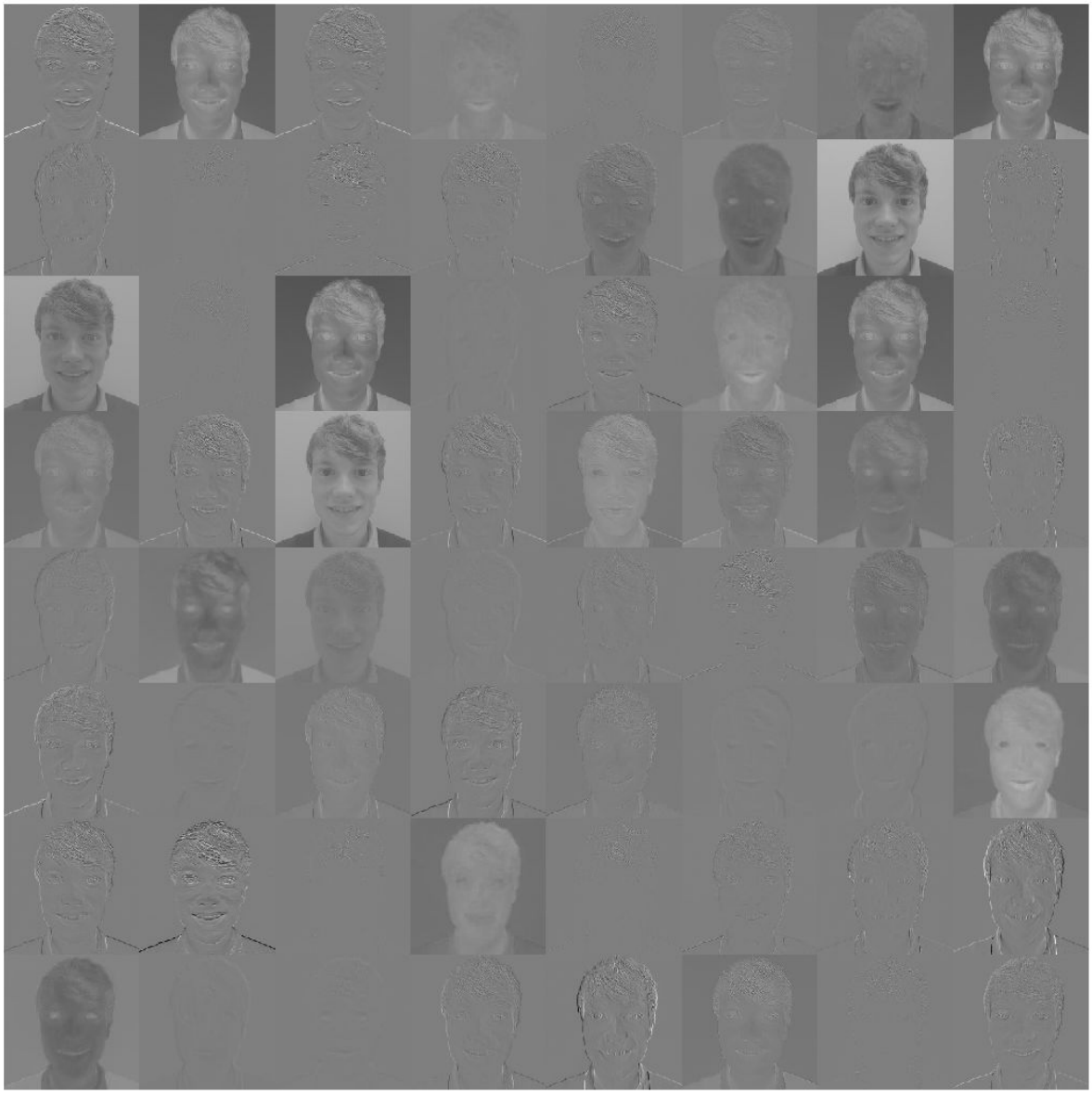
```
act1 = activations(net,im,'conv1');
```

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);
```

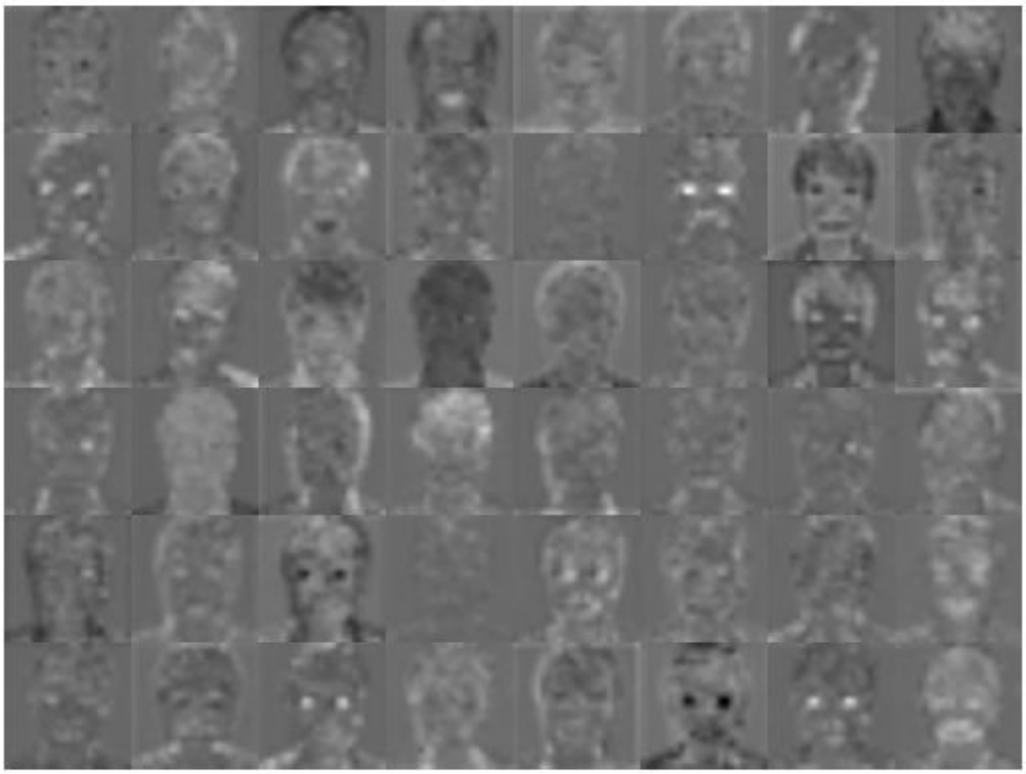
```
I = imtile({im,act1chMax});
imshow(I)
```

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)
```



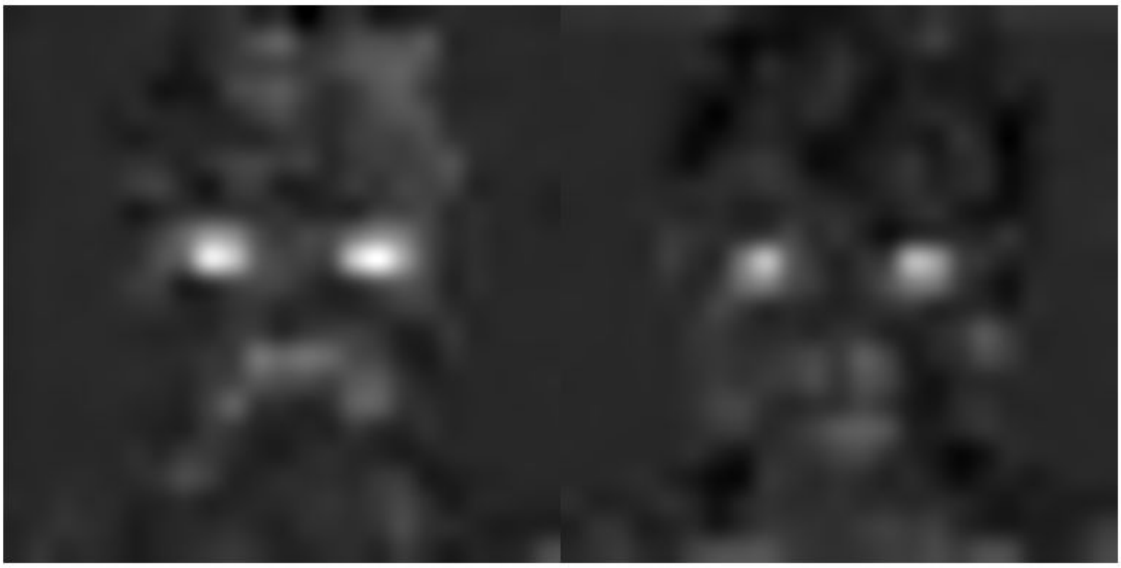
There are too many images to investigate in detail, so focus on some of the more interesting ones. Display the strongest activation in the `fire6-squeeze1x1` layer.

```
[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

`activations` | `deepDreamImage` | `squeezenet`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Dream Images Using GoogLeNet” on page 5-2
- “Visualize Features of a Convolutional Neural Network” on page 5-90

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
```

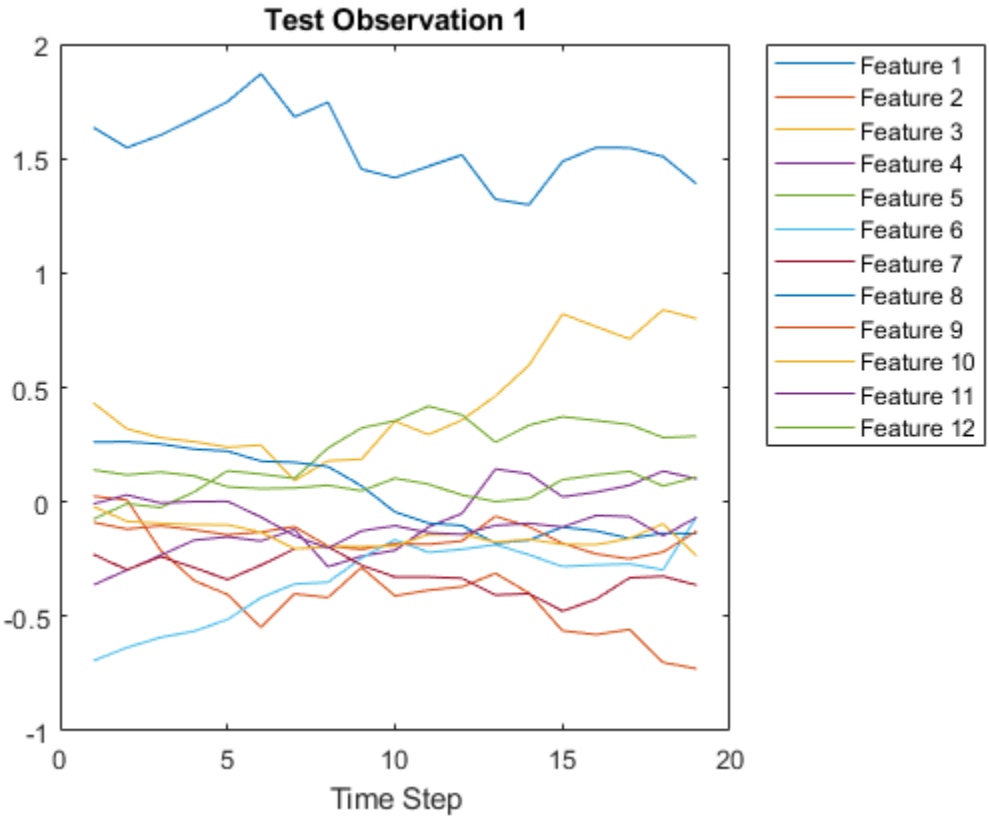
```
ans =
  5x1 Layer array with layers:

     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')
```

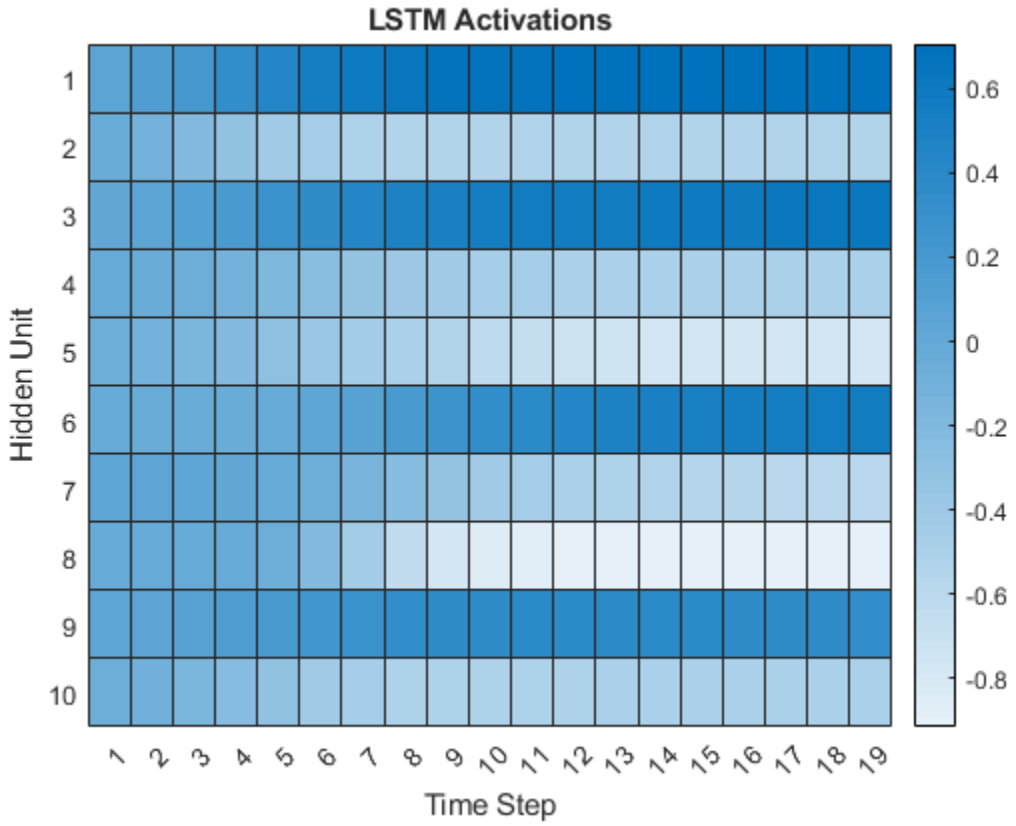


For each time step of the sequences, get the activations output by the LSTM layer (layer 2) for that time step and update the network state.

```
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

`activations` | `bilstmLayer` | `lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53

- “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.

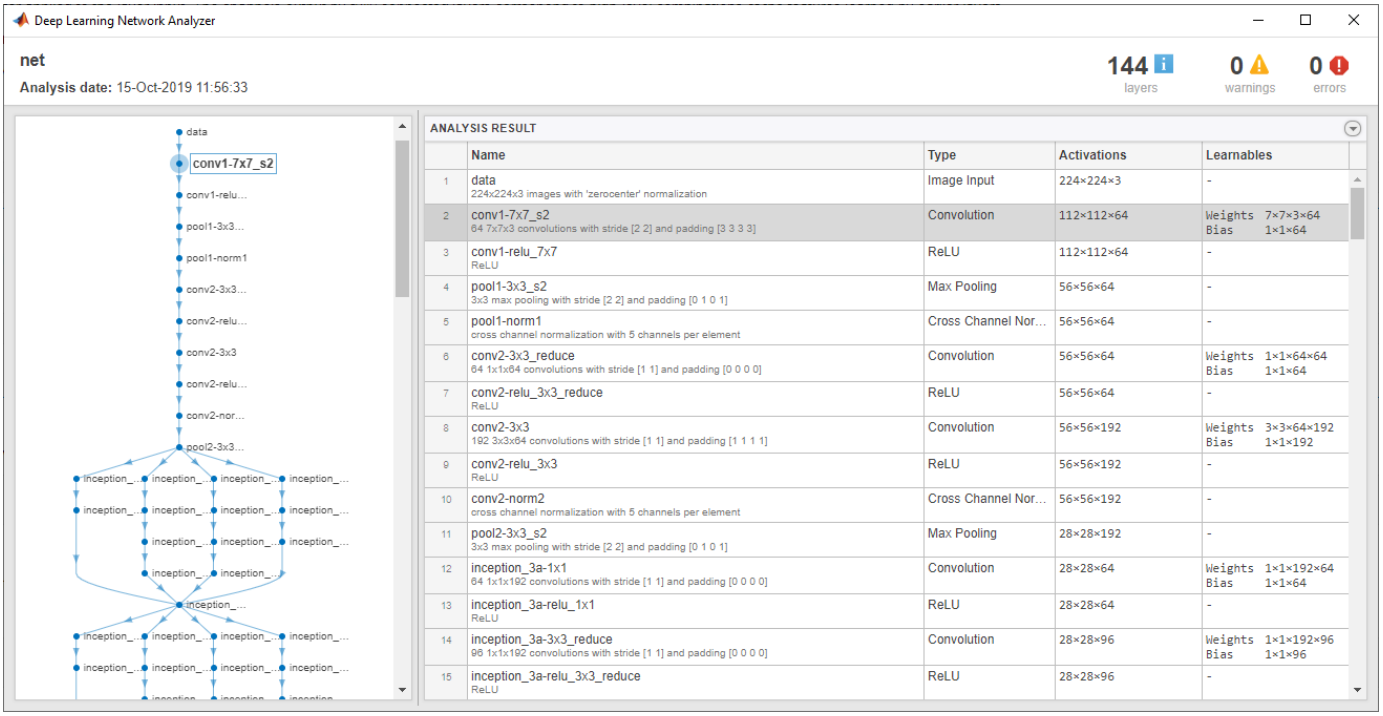
```
net = googlenet;
```

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`.

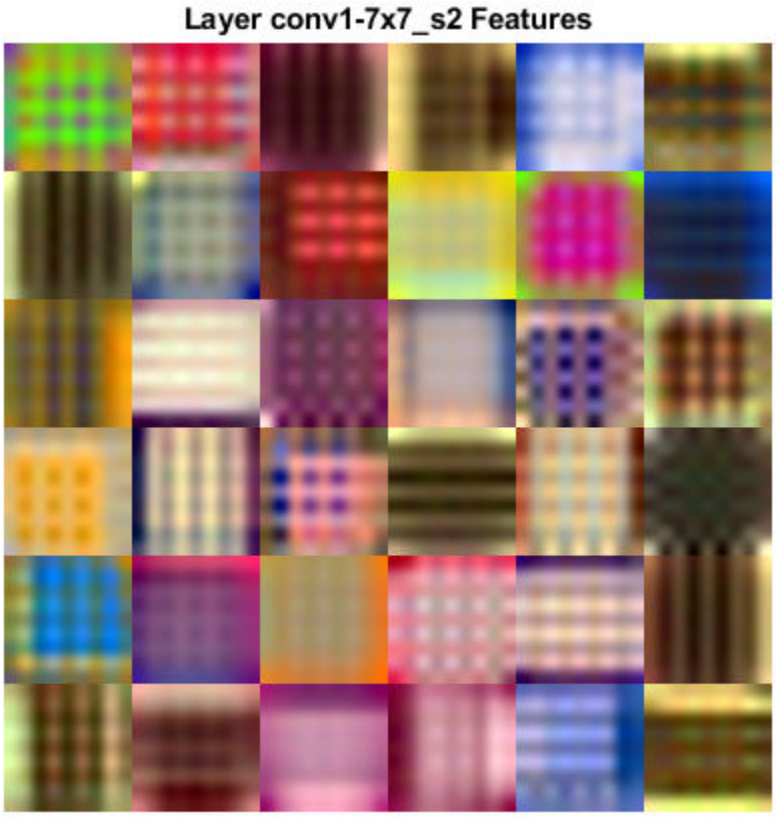
`deepDreamImage` uses a compatible GPU, by default, if available. Otherwise it uses the CPU. A CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher is required for running on a GPU.

```
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

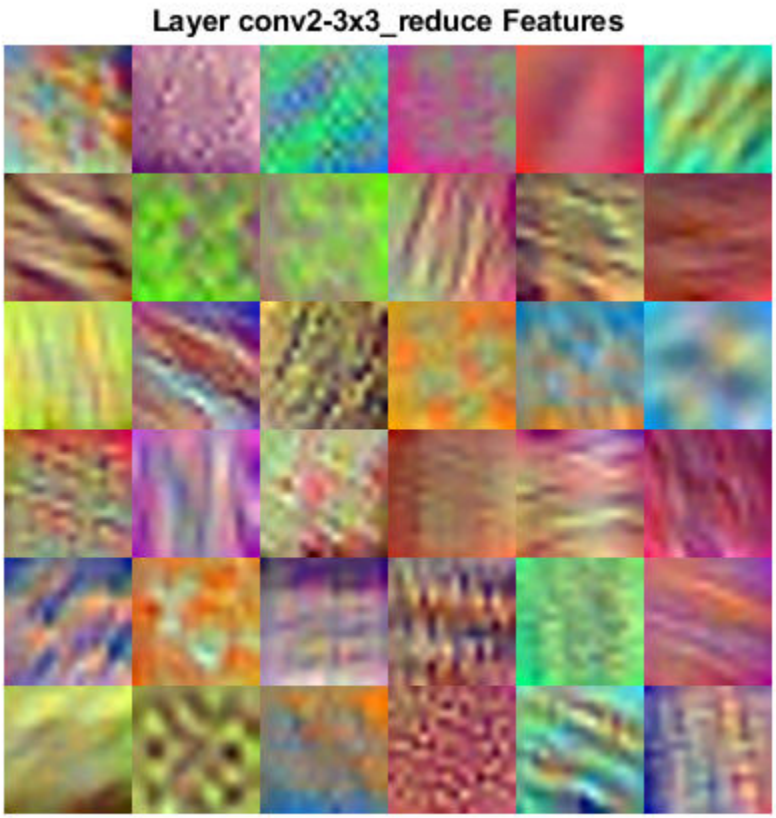
The second convolutional layer is named `'conv2-3x3_reduce'`, which corresponds to layer 6. Visualize the first 36 features learned by this layer by setting `channels` to be the vector of indices `1:36`.

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.

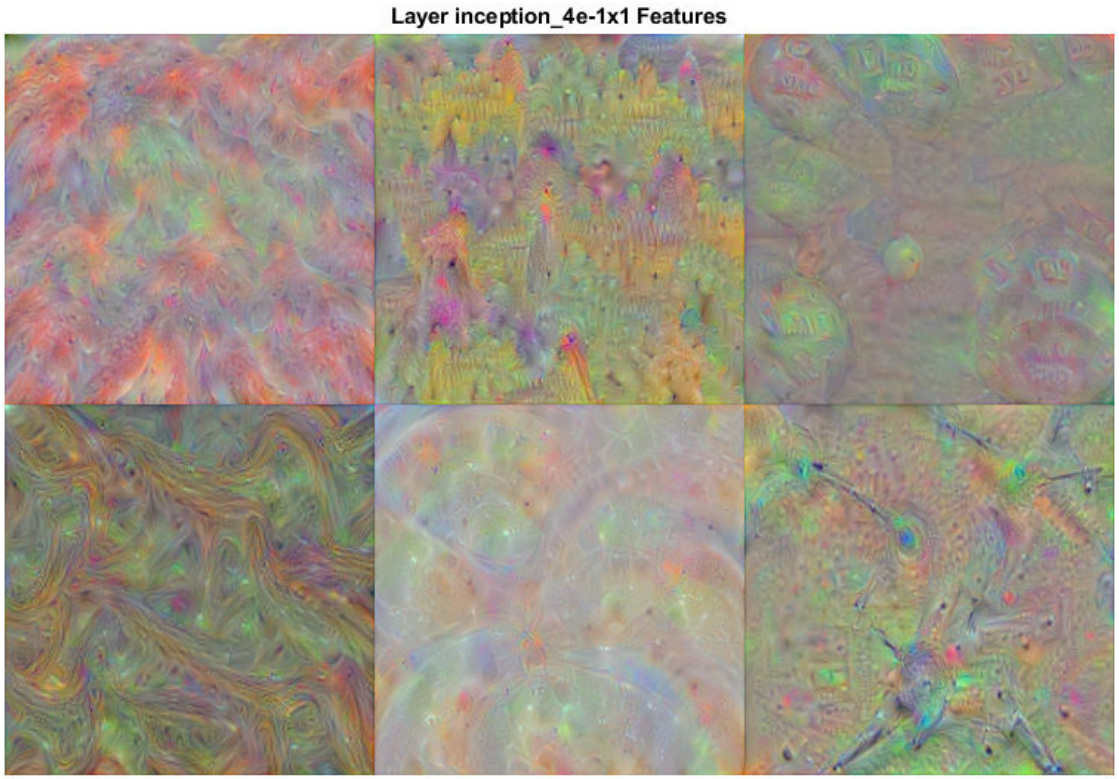
Increasing the number of pyramid levels and iterations per pyramid level can produce more detailed images at the expense of additional computation. You can increase the number of iterations using the

`'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];
```

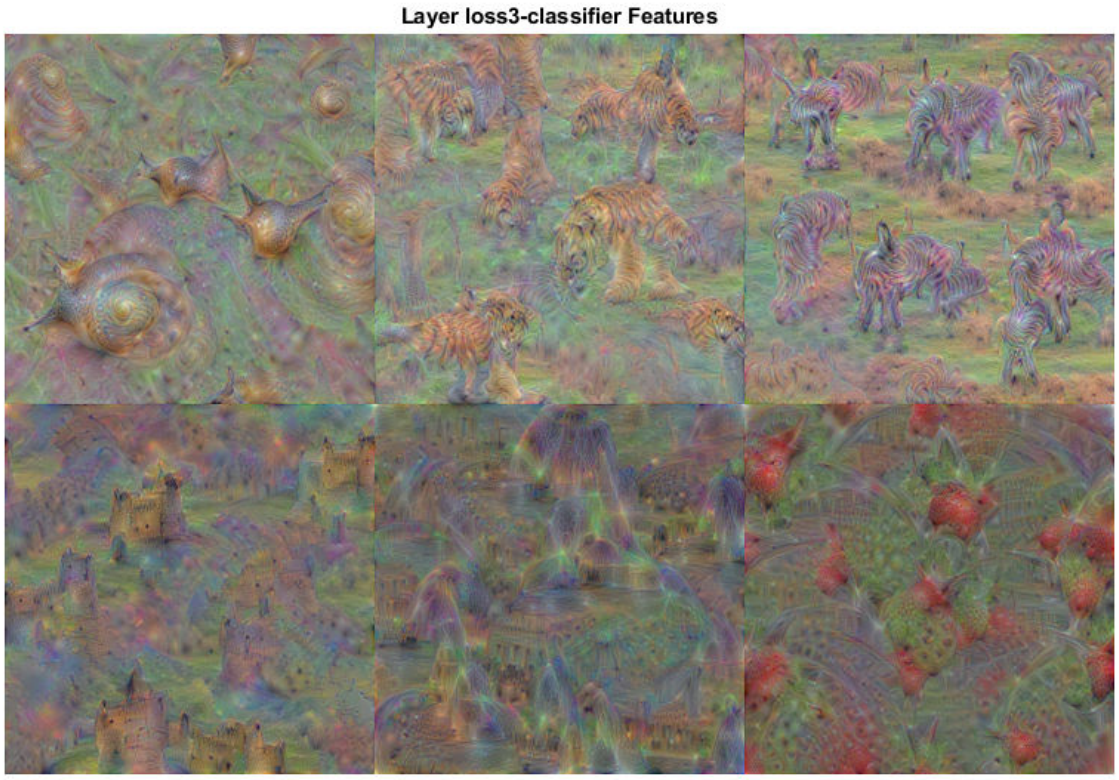
The classes are stored in the `Classes` property of the output layer (the last layer). You can view the names of the selected classes by selecting the entries in `channels`.

```
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

`deepDreamImage` | `googlenet`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Deep Dream Images Using GoogLeNet” on page 5-2
- “Visualize Activations of a Convolutional Neural Network” on page 5-75
- “Pretrained Deep Neural Networks” on page 1-12

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 also 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 learn how to diagnose why a neural network gets classifications wrong using a simple technique based around the class scores for different classes of images.

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 set to prepare for transfer learning on GoogLeNet. Display a few pictures 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 classes 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 Load Pretrained Networks.

```
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;
```

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 again 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 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 the Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher. 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.

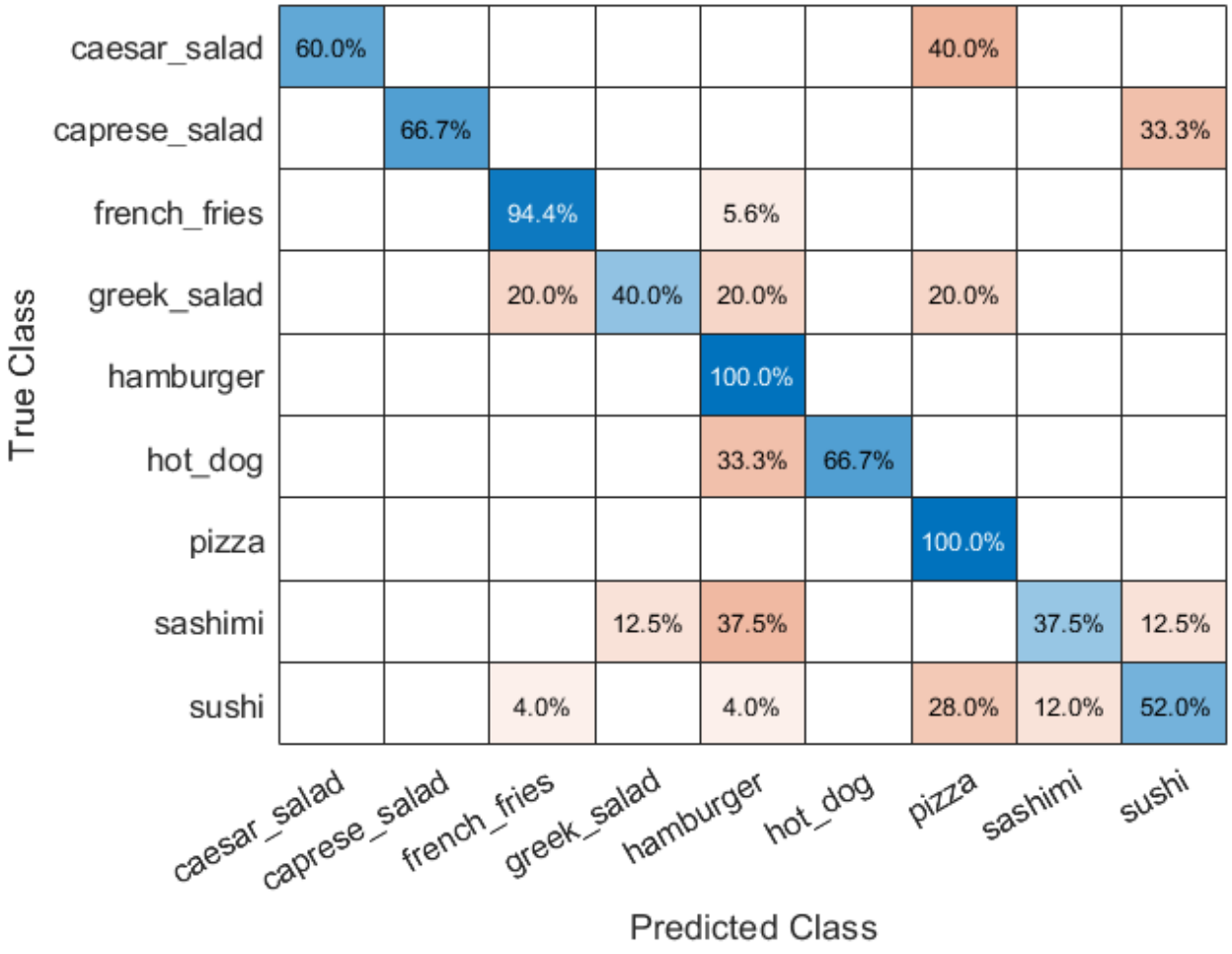
```
augimdsTest = augmentedImageDatastore(inputSize(1:2), imdsTest);
[predictedClasses, predictedScores] = classify(net, augimdsTest);

accuracy = mean(predictedClasses == imdsTest.Labels)
```

```
accuracy = 0.8622
```

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.

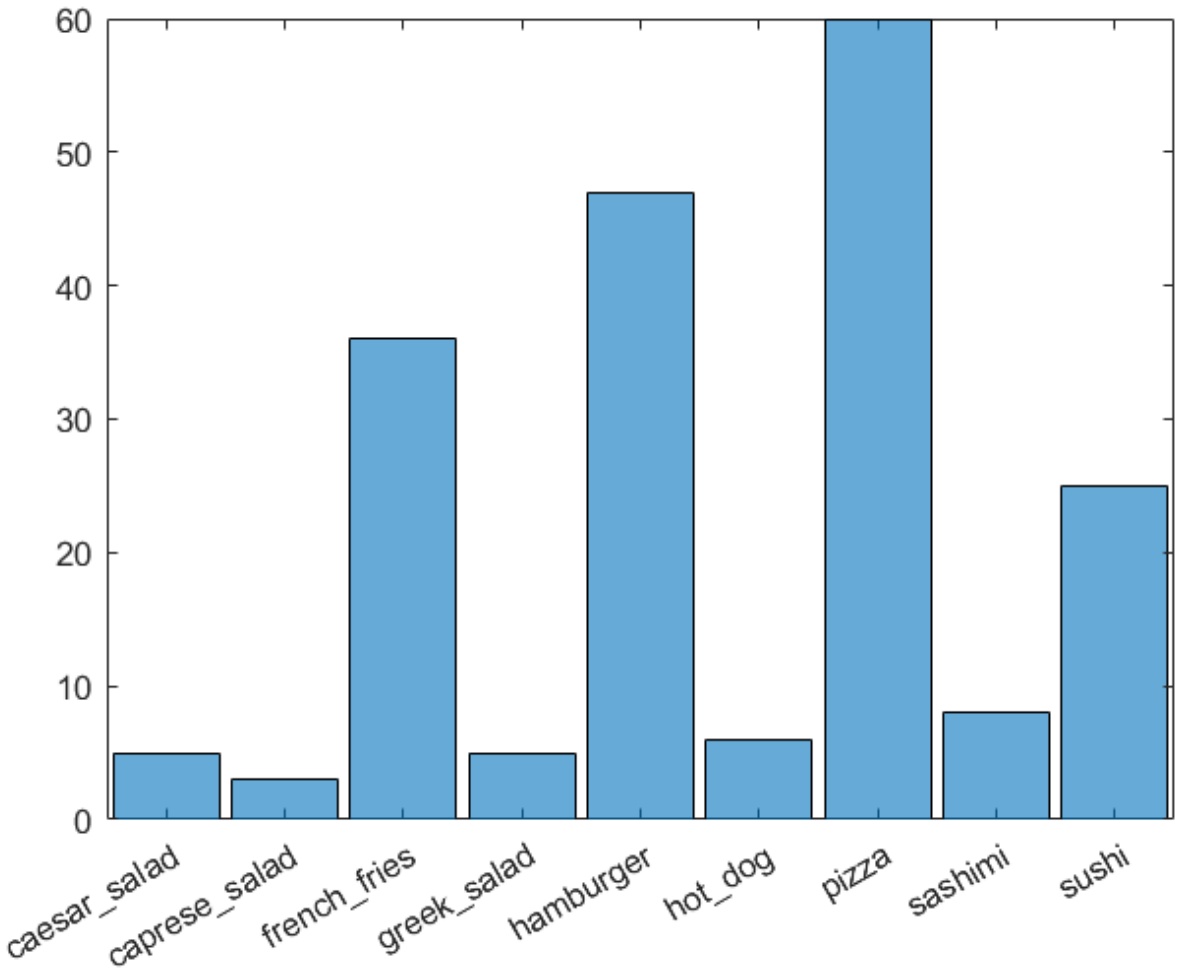
```
figure;
confusionchart(imdsTest.Labels, predictedClasses, 'Normalization', "row-normalized");
```



The confusion matrix shows that, most of the time, the network is incorrectly classifying some classes, such as greek salad, sashimi, and sushi. To better understand why this is happening, run some tests on these 3 classes, while keeping in mind that the data set used in this example underrepresents them.

```
figure();
histogram(imdsValidation.Labels);
```

```
ax = gca();
ax.XAxis.TickLabelInterpreter = "none";
```

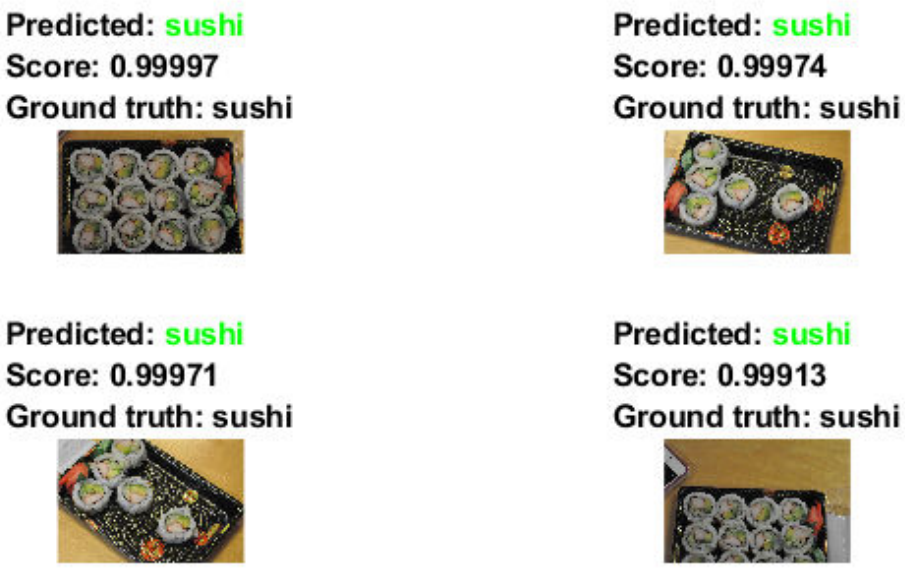


Sushis Most Like Sushi

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?".

To do this, plot the maximally-activating images, or 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, numImgs

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 sushis. 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.

To use Grad-CAM, create a `dlnetwork` from the GoogLeNet network. Specify the name of the softmax layer, `'prob'`. Specify the name of the final convolutional layer, `'inception_5b-output'`.

```
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph, lgraph.Layers(end).Name);
dlnet = dlnetwork(lgraph);
softmaxName = 'prob';
convLayerName = 'inception_5b-output';
```

Read the first resized image from the augmented image datastore , then plot the Grad-CAM visualization.

```
imageNumber = 1;

observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```

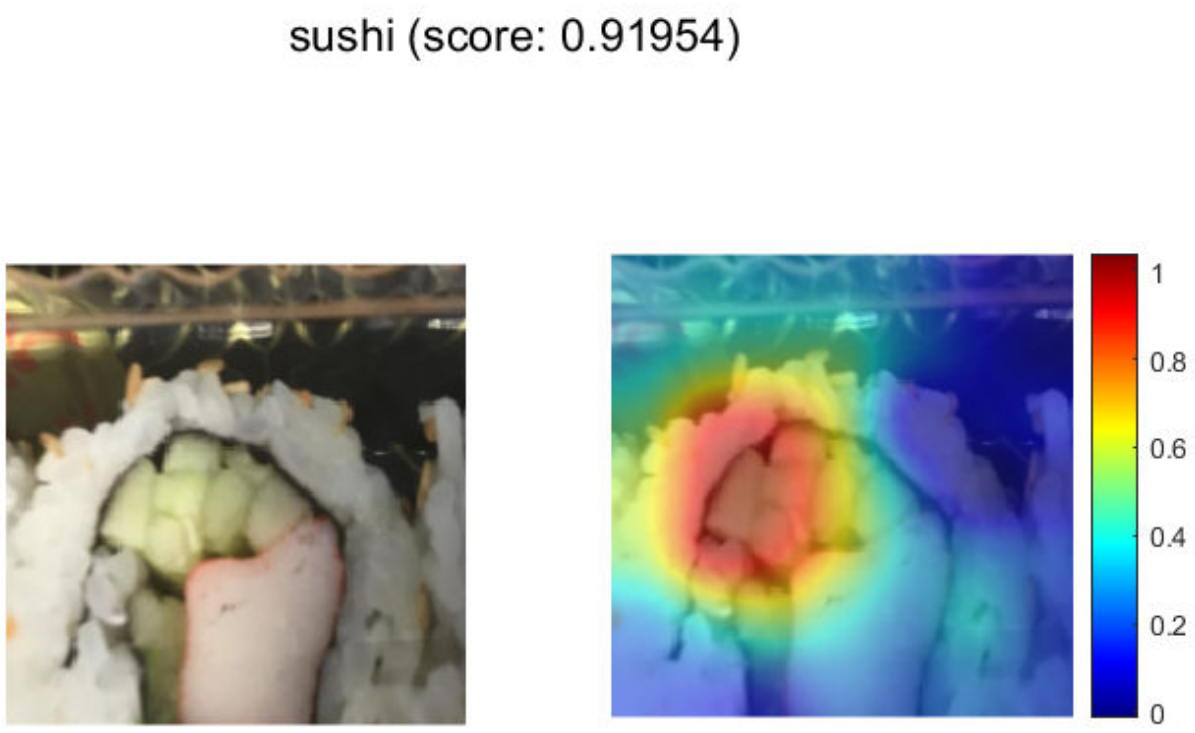
The plot confirms that the network is correctly focusing on sushi instead of the plate or the table. The network classifies this image as sushi because it sees a group of sushis. However, is it able to classify one sushi on its own?

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 = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

alpha = 0.5;

figure
plotGradCAM(img, gradcamMap, alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```

The network associates the sushi class with multiple sushis stacked together. 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 = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
sgtitle(string(label)+" (score: "+ max(score)+")")
```

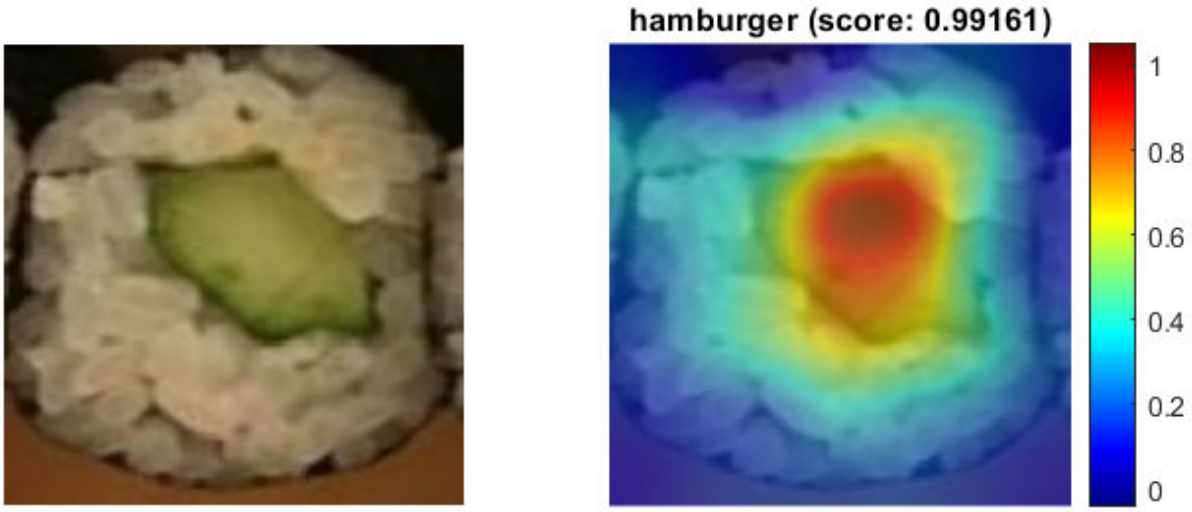


The network is able to classify this lone sushi correctly. However, the GradCAM shows that the network focuses more on the cluster of cucumber in the sushi rather than the whole piece together.

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 = computeGradCAM(dlnet, img, softmaxName, convLayerName, 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.

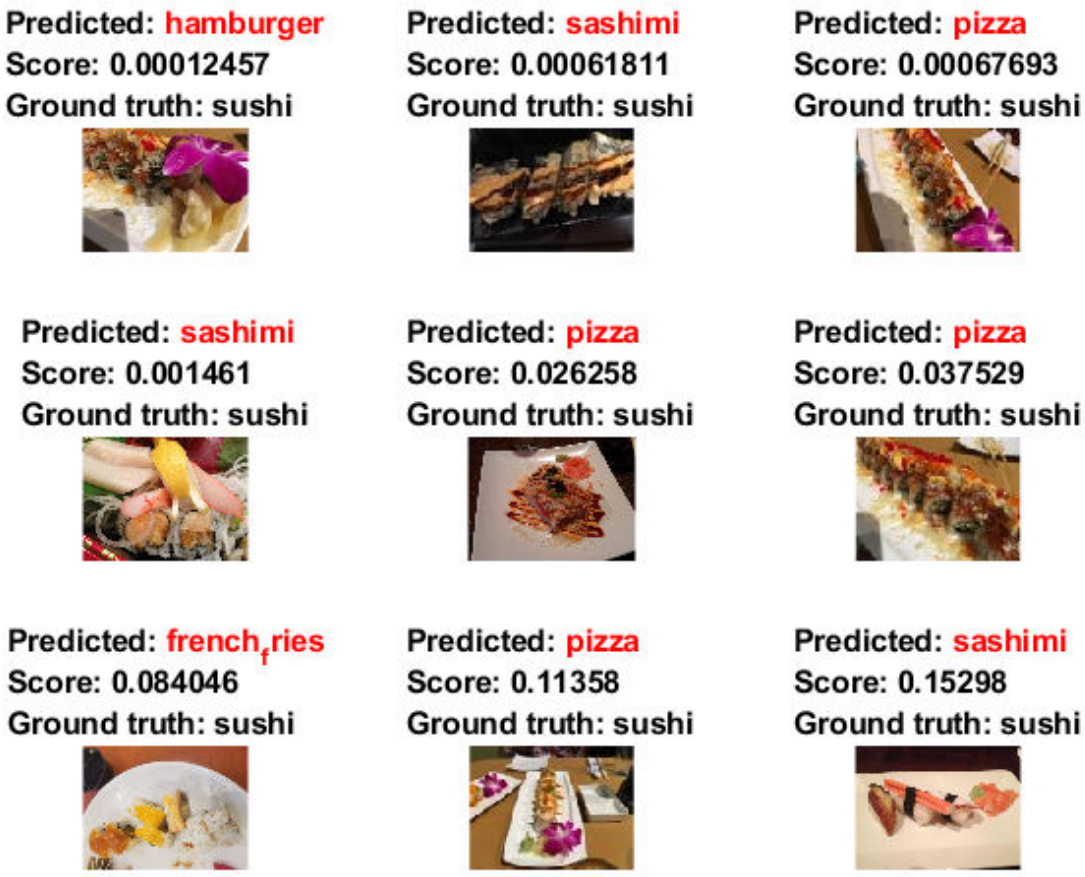
To solve this issue, you have to feed the network with more images of lone sushi during the training process.

Sushis Least Like Sushis

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, numImgs

figure
plotImages(imdsTest, imgIdx, sortedScores, predictedClasses, numImgsToShow)
```



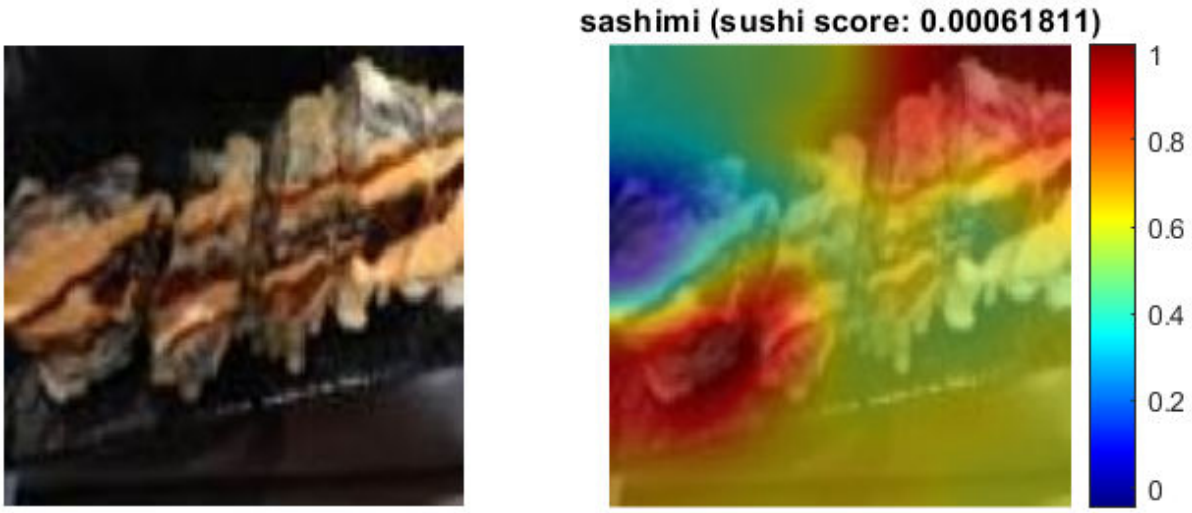
Investigate Sushi Misclassified as Sashimi

Why is the network classifying sushi as sashimi? The network classifies 3 out of the 9 images as sashimi, respectively images 2, 4, and 9. Two of them, 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 = 2;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
```

```
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (sushi score: "+ max(score)+")")
```



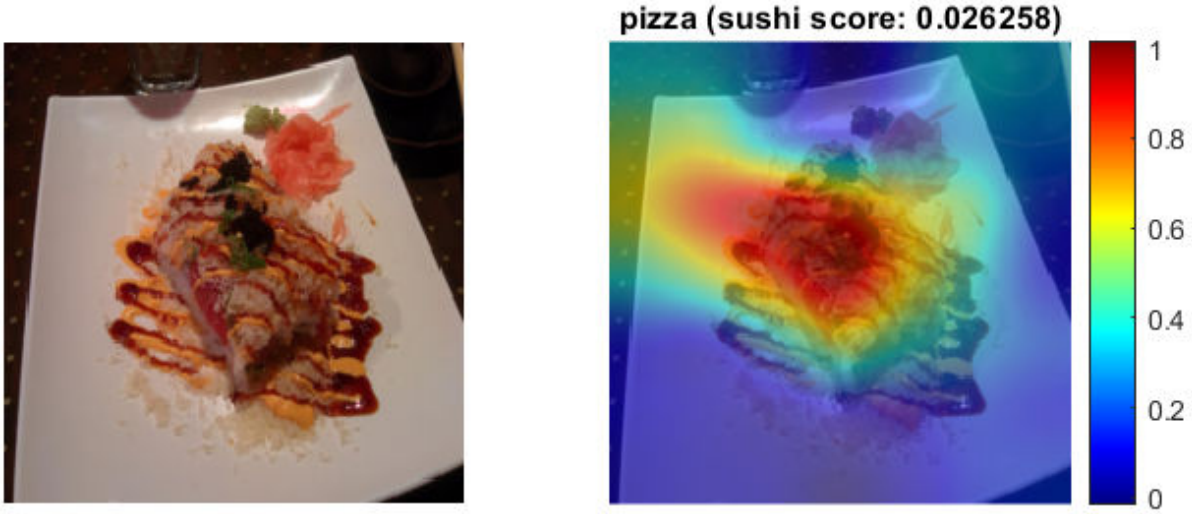
As expected, the network focuses on the sashimi instead of the sushi.

Investigate Sushi Misclassified as Pizza

Why is the network classifying sushi as pizza? The network classifies 4 out of 9 images as pizza instead of sushi. These images have many colorful toppings.

To see which part of the image the network is looking at, run the Grad-CAM technique on one of these images.

```
imageNumber = 5;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (sushi score: "+ max(score)+")")
```



The network strongly focuses on the toppings.

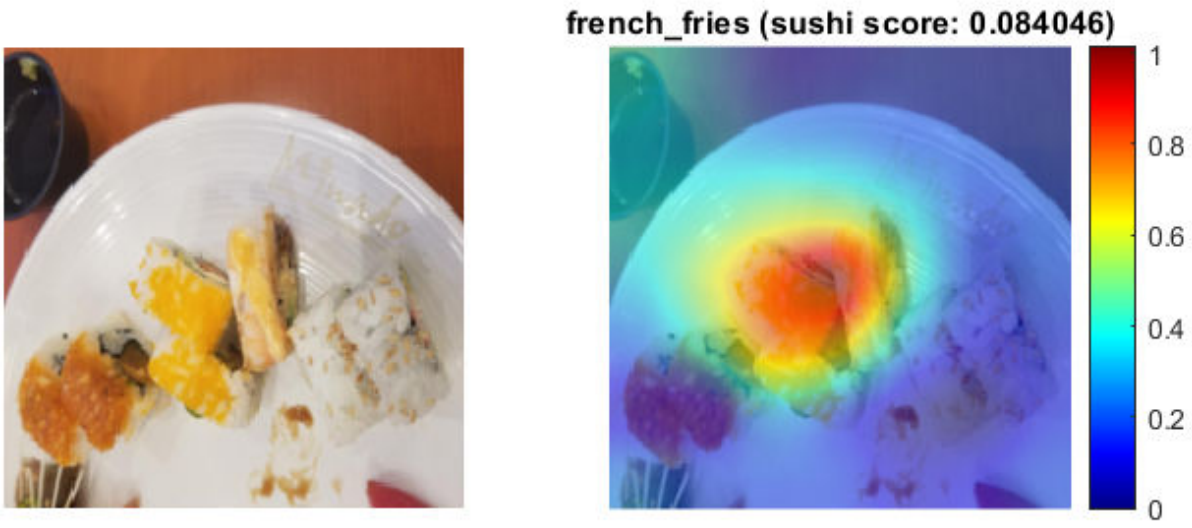
To help the network distinguish pizza from sushi with toppings, add more training images of sushi with toppings.

Investigate Sushi Misclassified as French Fries

Why is the network classifying sushi as french fries? The network classifies the 7th image as french fries instead of sushi. This specific sushi is yellow and the network might associate this color with french fries.

Run Grad-CAM on this image.

```
imageNumber = 7;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, 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.

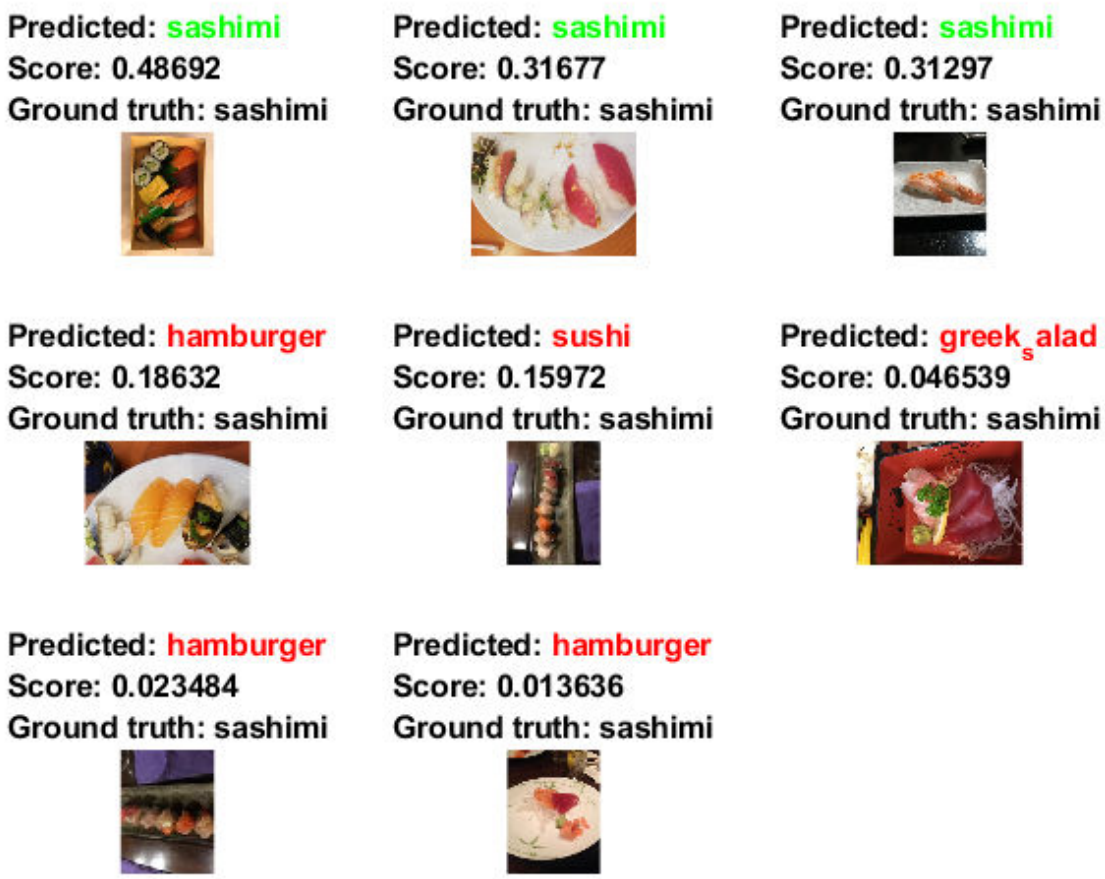
To help the network in this specific case, train it with more images of yellow foods that are not french fries.

Most and Least Like a Sashimi

The food data set comprises only 8 observations of sashimi.

Similar to the sushi class, display the sashimi images in order of most like sashimi to least like sashimi.

```
chosenClass = "sashimi";
numImgsToShow = 8;

[sortedScores, imgIdx] = findMaxActivatingImages(imdsTest, chosenClass, predictedScores, numImgs

figure
plotImages(imdsTest, imgIdx, sortedScores, predictedClasses, numImgsToShow)
```



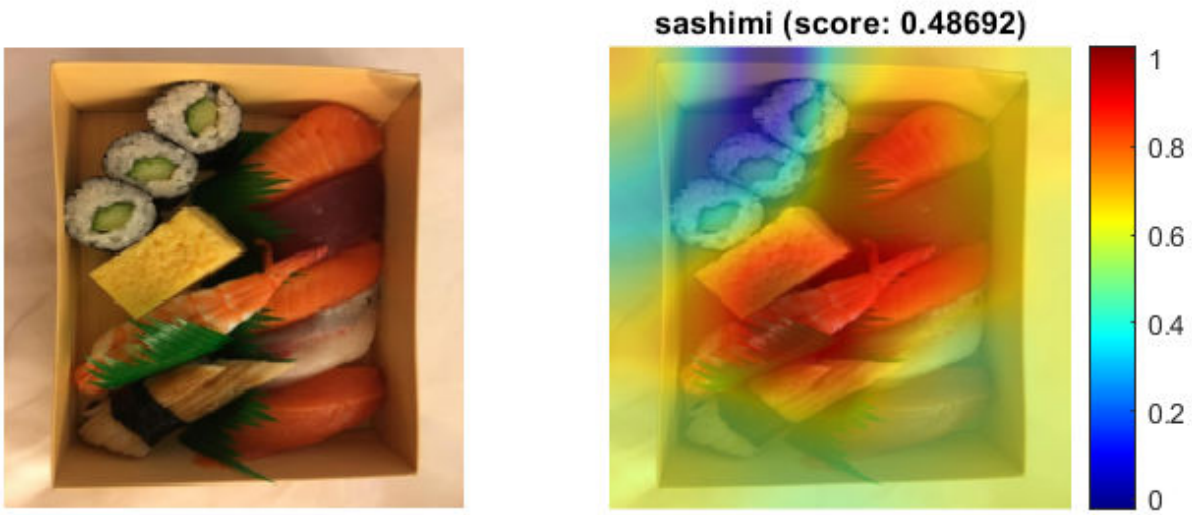
This figure shows that the network is not confident about sashimi accompanied by certain other food, such as sashimi with greenery. However this also shows that there are some flaws in the data where pictures of sushi are mislabeled as sashimi as ground truth such as image number 5.

Visualize Cues for the Sashimi Class

Is the network looking at the right thing for sashimi? Consider only the images that the network correctly classifies as sashimi, and check what it focuses on during the classification process.

Read the highest score sashimi-like image and run Grad-CAM on it.

```
imageNumber = 1;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (score: "+ max(score)+")")
```



This image contains both sashimi and sushi. The network correctly focuses on the sashimi and ignores the sushi. However, the data set contains other images that also have a mix of sushi and sashimi but they are labeled as sushi-only instead. This makes it hard for the network to learn how to properly classify these images, as shown in the "sushi least like sushi" case. This also explains why the prediction score isn't very high in this particular case.

Investigate Sashimi Misclassified as Hamburger

Why is the network classifying sashimi as hamburger? The network classifies 3 out of 8 images of sashimi as hamburger. The data set contains many more observations of hamburger that might bias the network.

Read one of these images and run Grad-CAM on it.

```
imageNumber = 4;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (sashimi score: "+ max(score)+")")
```

Here, the seaweed-wrapped food confuses the network, as it resembles the bun-meat-bun characteristics of a humburger. This again is a flaw in the sashimi data which is not varied enough.

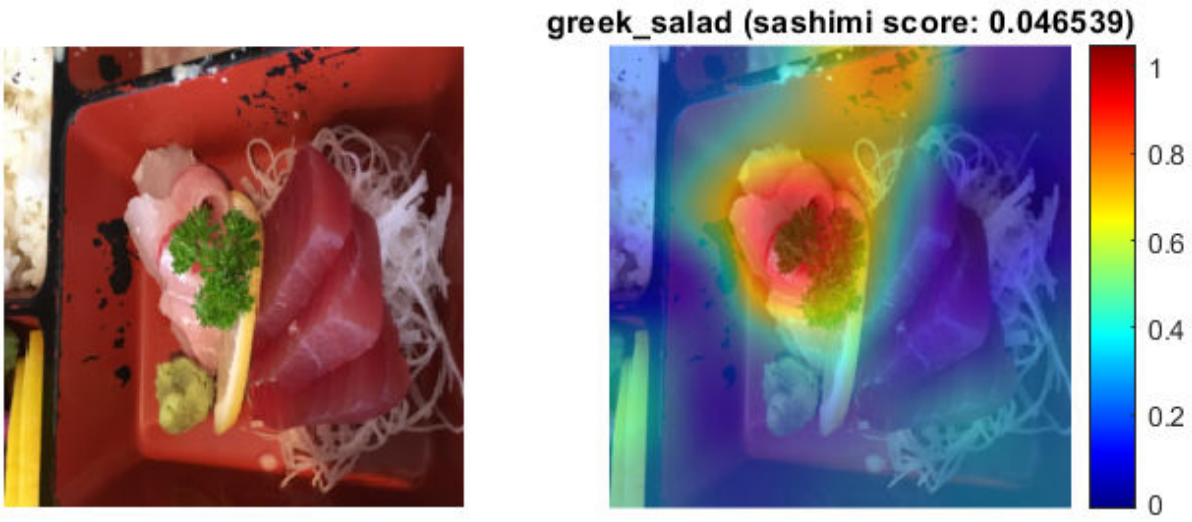
Investigate Sashimi Misclassified as Greek Salad

Why is the network classifying sashimi as greek salad? The network classifies the 6th image as greek instead of sashimi.

Read the image and run the Grad-CAM technique.

```
imageNumber = 6;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
```

```
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (sashimi score: "+ max(score)+")", "Interpreter", "none")
```



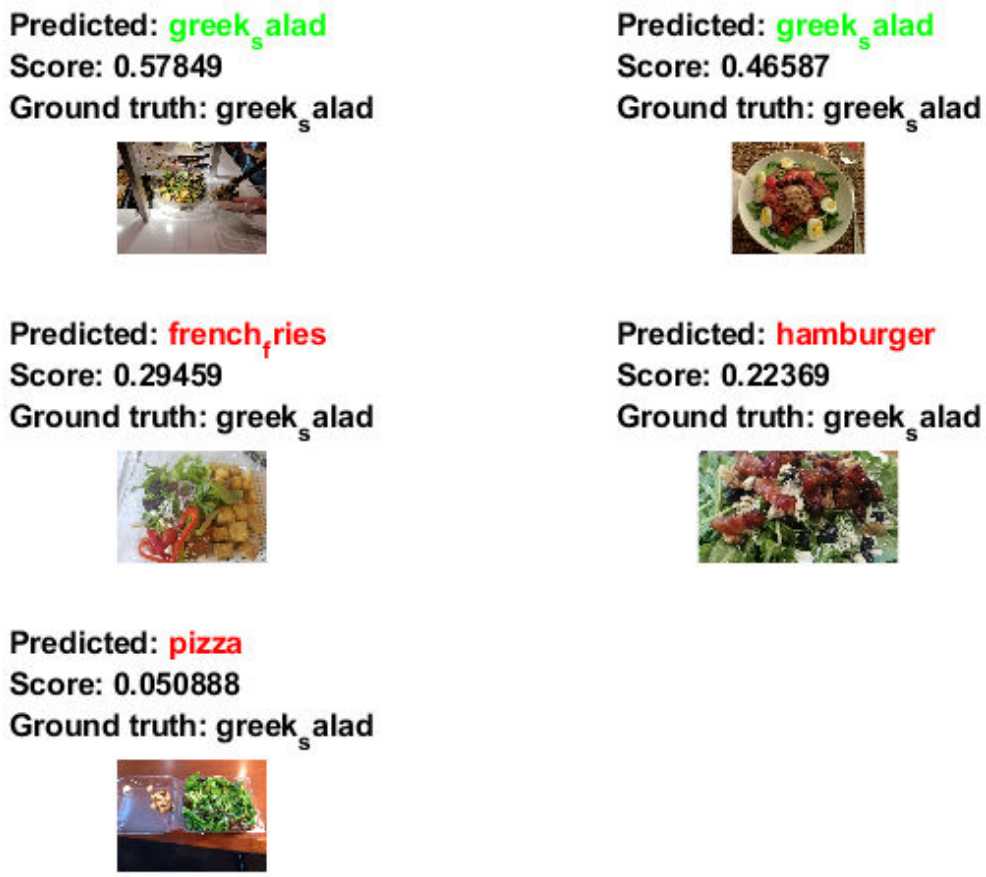
Here, the greenery misleads the network into thinking that the images depicts a plate of sashimi. To help the network recognize the difference between these two classes, add a more varied data set of sashimi accompanied by leaves and vegetables.

Most and Least Like a Greek Salad

The food data set only comprises 5 observations of greek salad.

Similar to the above cases, display the greek salad images in order of most likely greek salad to least likely greek salad.

```
chosenClass = "greek_salad";
numImgsToShow = 5;

[sortedScores, imgIdx] = findMaxActivatingImages(imdsTest, chosenClass, predictedScores, numImgs

figure
plotImages(imdsTest, imgIdx, sortedScores, predictedClasses, numImgsToShow)
```



This figure shows that the network is not confident about greek salad accompanied by certain other food, such as meat.

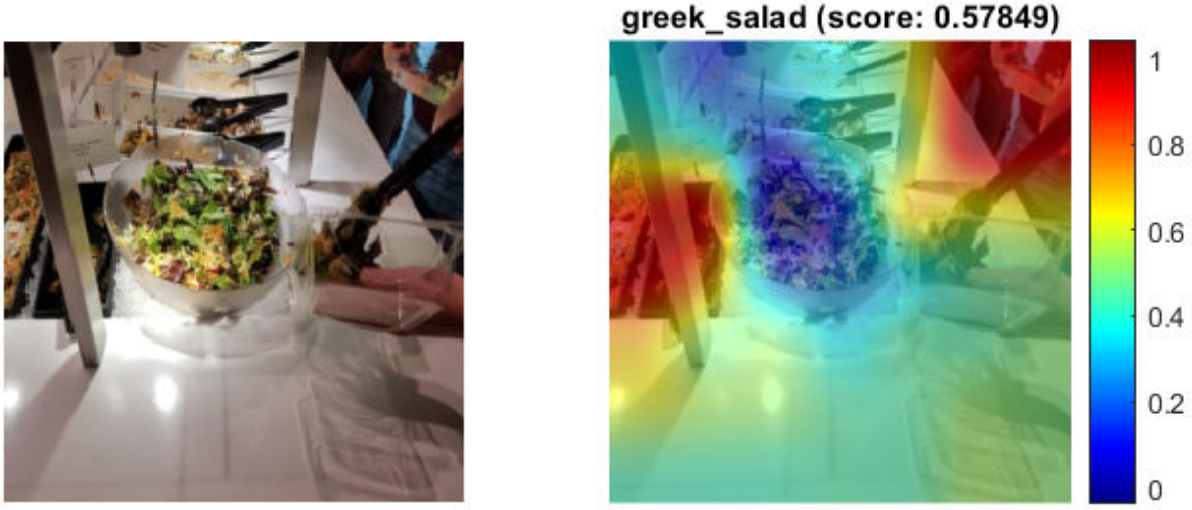
Investigate Cues for the Greek Salad Class

Is the network looking at the right thing for greek salad? Consider only the images that the network correctly classifies as greek salad, and check what it focuses on during the classification process.

Read the highest score greek salad image according to the network, and run Grad-CAM.

```
imageNumber = 1;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
```

```
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (score: "+ max(score)+")", "Interpreter", "none")
```



The network ignores the actual greek salad and rather looks at things in the background instead. This is an indication that the network is classifying this image correctly but for the wrong reasons.

Read the second image that the network classifies correctly as greek salad, and run Grad-CAM.

```
imageNumber = 2;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (score: "+ max(score)+")", "Interpreter", "none")
```

Even though greek salad usually does not have eggs, the network focuses on the tomato and olives which indicates it is looking at the right things for a greek salad.

Investigate Greek Salad Misclassified as Hamburger

Why is the network classifying greek salad as hamburger? The network classifies the fourth image as a hamburger. Is it caused by the chunks of meat in the salad?

Read the image and run Grad-CAM.

```
imageNumber = 4;
observation = augimdsTest.readByIndex(imgIdx(imageNumber));
img = observation.input{1};

label = predictedClasses(imgIdx(imageNumber));
score = sortedScores(imageNumber);

gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label);

figure
alpha = 0.5;
```

```
plotGradCAM(img, gradcamMap, alpha);
title(string(label)+" (greek_salad score: "+ max(score)+")", "Interpreter", "none")
```

The network focuses on the regions with meat.

Conclusion

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 a number of images with incorrect true labels, such as the "sashimi" which is actually "sushi" instead.
- 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.

- Any sushi or sashimi accompanied by salad-like ingredients is likely to be confused with salad classes. Therefore, you must train the network with more images of sushi and sashimi with some salad on the side.
- To learn what makes a greek salad stands out, the network needs more observations of greek salads.

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, numI
% 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, predictedSco

% 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, numI
```

```
% 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, predictedSco

% 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, pred
% 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

    subplot(3,ceil(numImgsToShow./3),i)
    imshow(imread(imgPath));
    title("Predicted: " + color + string(predClass) + "\newline\color{black}Score: " + num2str(sc
end

end

function [convMap,dScoresdMap] = gradcam(dlnet, dlImg, softmaxName, convLayerName, classfn)
% Computes the Grad-CAM map for a dlnetwork, taking the derivative of the softmax layer score
% for a given class with respect to a convolutional feature map.
[scores,convMap] = predict(dlnet, dlImg, 'Outputs', {softmaxName, convLayerName});
classScore = scores(classfn);
dScoresdMap = dlgradient(classScore,convMap);
end

function gradcamMap = computeGradCAM(dlnet, img, softmaxName, convLayerName, label)
```

```
% For automatic differentiation, the input image img must be a dlarray.
dlImg = dlarray(single(img),'SSC');

% Compute the gradCAM map by passing the dlarray image
[convMap, dScoresdMap] = dlfeval(@gradcam, dlnet, dlImg, softmaxName, convLayerName, label);

% Resize the gradient map to the net image size, and scale the scores to the appropriate levels
gradcamMap = sum(convMap .* sum(dScoresdMap, [1 2]), 3);
gradcamMap = extractdata(gradcamMap);
gradcamMap = rescale(gradcamMap);
gradcamMap = imresize(gradcamMap, dlnet.Layers(1).InputSize(1:2), 'Method', 'bicubic');
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

`augmentedImageDatastore` | `classify` | `confusionchart` | `dlnetwork` | `googlenet` | `imageDatastore`

Related Examples

- “Visualize Activations of a Convolutional Neural Network” on page 5-75

More About

- “Deep Learning in MATLAB” on page 1-2
- “Pretrained Deep Neural Networks” on page 1-12

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 3-72.

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:

$$\text{scoreDisriminator} = \frac{1}{2}\text{mean}(\widehat{Y}_{\text{Real}}) + \frac{1}{2}\text{mean}(1 - \widehat{Y}_{\text{Generated}}),$$

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.

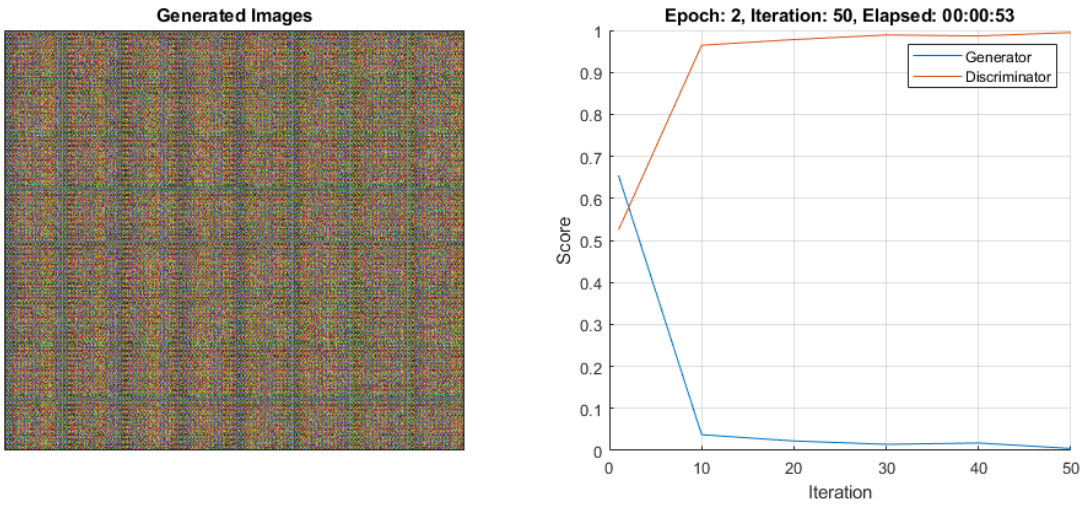
Convergence Failure

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:

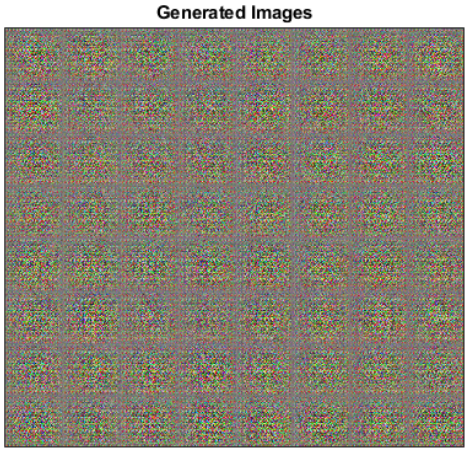
- Impairing the discriminator by randomly giving false labels to real images (one-sided label flipping)
- Impairing the discriminator by adding dropout layers
- Improving the generator's ability to create more features by increasing the number of filters in its convolution layers
- Impairing the discriminator by reducing its number of filters

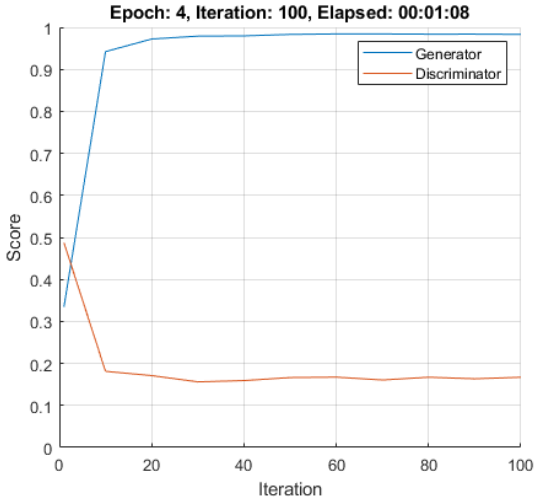
For an example showing how to flip the labels of the real images, see “Train Generative Adversarial Network (GAN)” on page 3-72.

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.





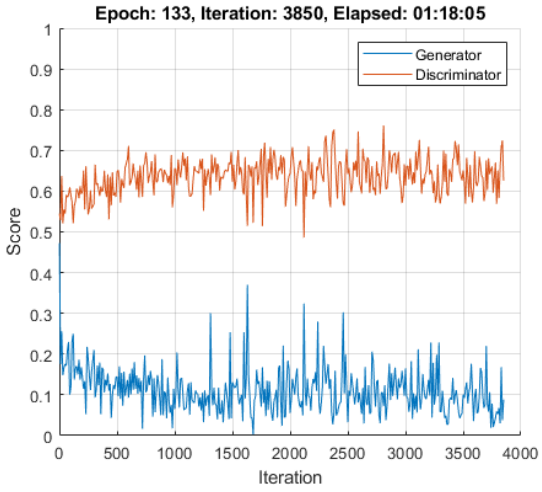
If the score does not recover from these values for many iterations, then it is better to stop the training. If this happens, then try balancing the performance of generator and the discriminator by:

- 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 3-72.

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `forward` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Train Conditional Generative Adversarial Network (CGAN)” on page 3-83
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Train Network Using Custom Training Loop” on page 15-134
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

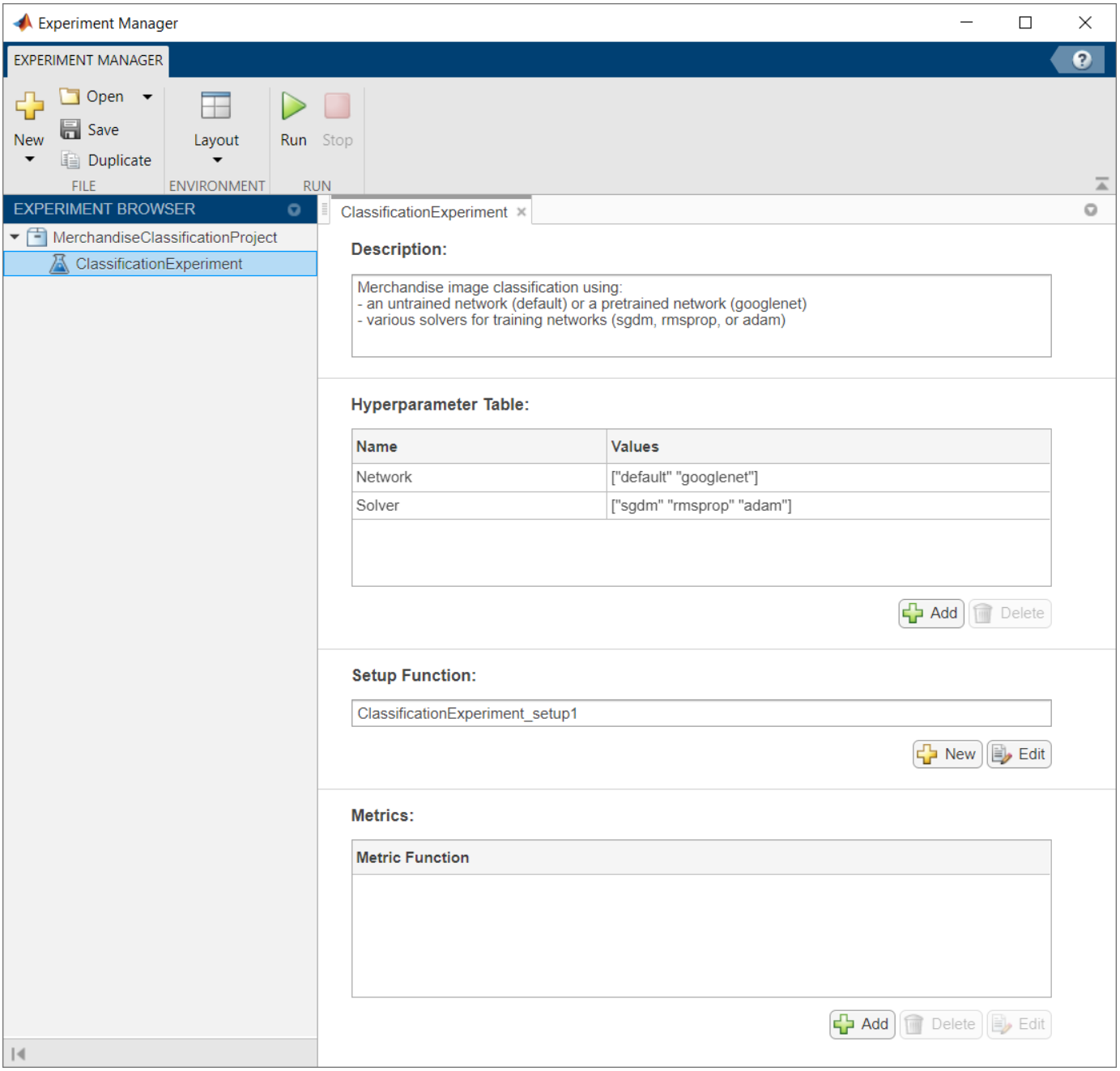
Manage Deep Learning Experiments

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.

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**, double-click the name of the experiment (`ClassificationExperiment`).



An experiment definition consists of a description, a hyperparameter table, a setup function, and (optionally) a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Deep Learning Experiment”.

The **Description** box 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 **Hyperparameter Table** contains the names and values of the hyperparameters used in the experiment. When you run the experiment, Experiment Manager sweeps through the hyperparameter values and trains the network multiple times. Each trial uses a different combination of the hyperparameter values specified in the table. This example uses two hyperparameters:

- `Network` specifies the network to train. The options include `"default"` (the default classification network provided by the setup function template) 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. To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor.

In this example, the input to the setup function is a `struct` with fields from the hyperparameter table. The setup function returns three outputs that you use to train a network for image classification problems. The setup function has three sections.

- **Load Image 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 16-108.
- **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`.
- **Specify Training Options** defines a `trainingOptions` object for the experiment. The example trains the network for 10 epochs using the algorithm specified by the `Solver` entry in the hyperparameter table.

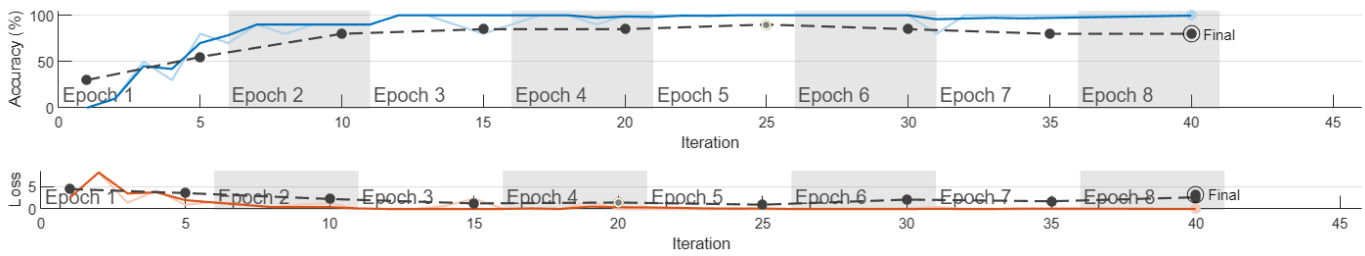
The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any metric functions.

Run Experiment

On the **Experiment Manager** tab, click **Run**. Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameters. 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 34 sec	default	sgdm	100.0000	1.3113e-7	80.0000	3.1913
2	Complete	100.0%	0 hr 0 min 29 sec	googlenet	sgdm	100.0000	0.0021	95.0000	0.0954
3	Complete	100.0%	0 hr 0 min 9 sec	default	rmsprop	100.0000	4.3908e-5	80.0000	2.0756
4	Complete	100.0%	0 hr 0 min 28 sec	googlenet	rmsprop	100.0000	0.0014	95.0000	0.1546
5	Complete	100.0%	0 hr 0 min 10 sec	default	adam	100.0000	0.0000	85.0000	2.3943
6	Complete	100.0%	0 hr 0 min 28 sec	googlenet	adam	100.0000	0.0002	95.0000	0.1333

While the experiment is running, click **Training Plot** to display the training plot and track the progress of each trial.



Evaluate Results

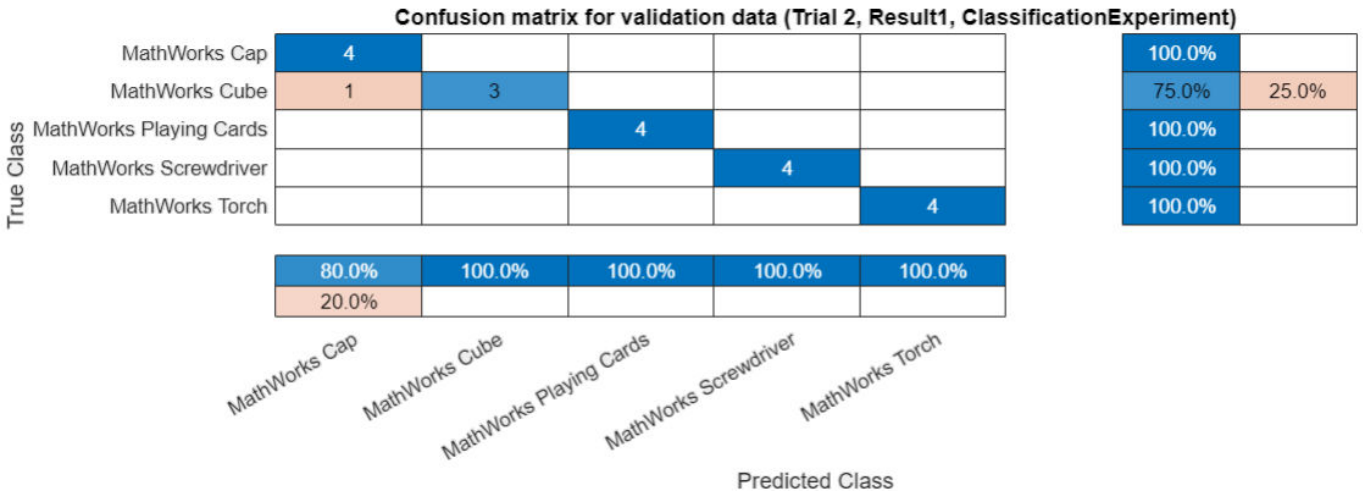
To find the best result for your experiment, sort the table of results by validation accuracy.

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 29 sec	googlenet	sgdm	100.0000	0.0021	95.0000	0.0954
4	Complete	100.0%	0 hr 0 min 28 sec	googlenet	rmsprop	100.0000	0.0014	95.0000	0.1546
6	Complete	100.0%	0 hr 0 min 28 sec	googlenet	adam	100.0000	0.0002	95.0000	0.1333
5	Complete	100.0%	0 hr 0 min 10 sec	default	adam	100.0000	0.0000	85.0000	2.3943
1	Complete	100.0%	0 hr 0 min 34 sec	default	sgdm	100.0000	1.3113e-7	80.0000	3.1913
3	Complete	100.0%	0 hr 0 min 9 sec	default	rmsprop	100.0000	4.3908e-5	80.0000	2.0756

To display the confusion matrix for this trial, select the top row in the results table and click **Confusion Matrix**.



Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager saves your results and closes all of the experiments contained in the project.

See Also

Experiment Manager | `googlenet` | `resnet18` | `trainNetwork` | `trainingOptions`

More About

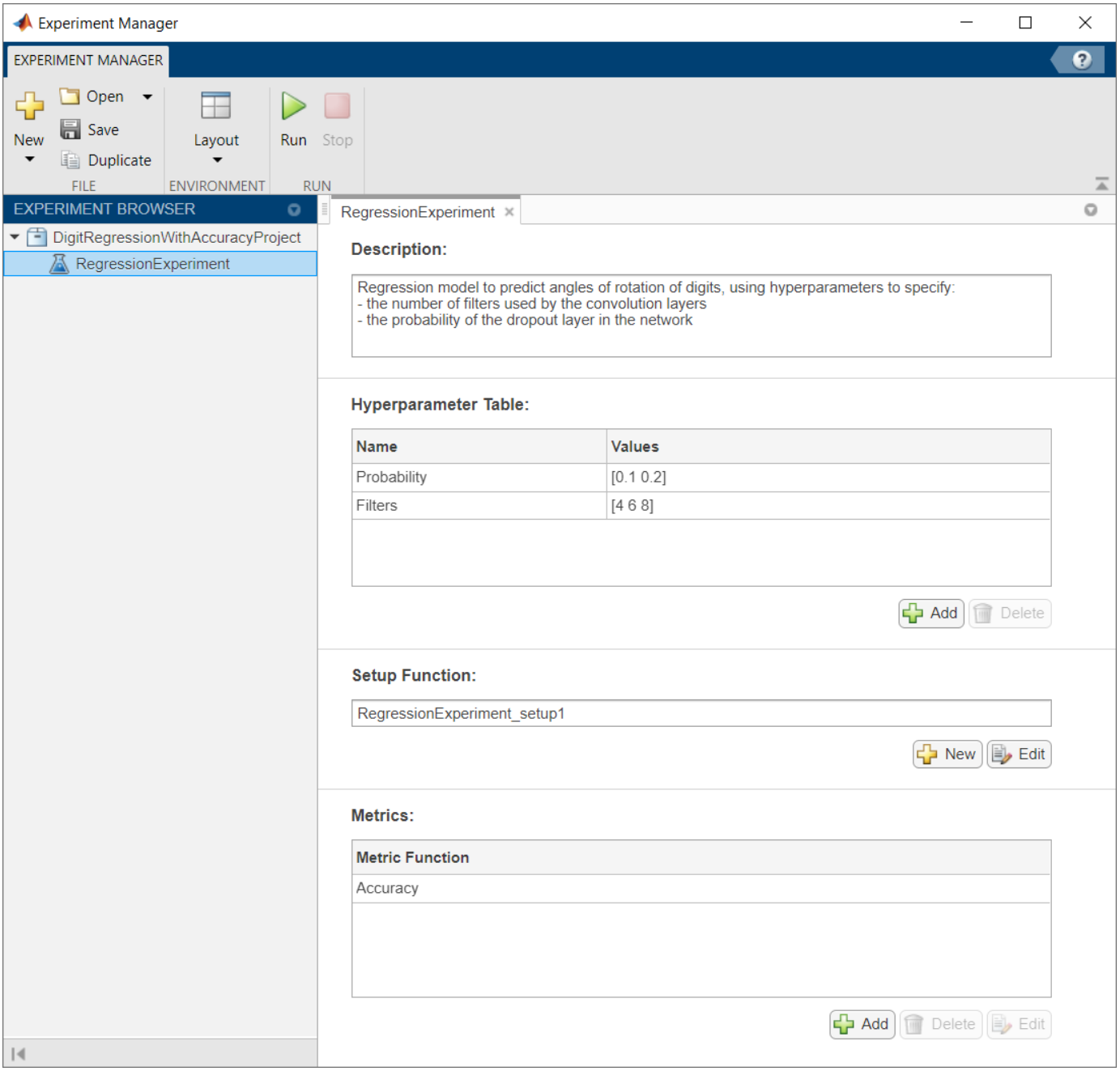
- “Get Started with Transfer Learning”
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Create a Deep Learning Experiment for Regression” on page 6-7
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-12

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 3-46.

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**, double-click the name of the experiment (`RegressionExperiment`).



An experiment definition consists of a description, a hyperparameter table, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Deep Learning Experiment”.

The **Description** box 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 **Hyperparameter Table** contains the names and values of the hyperparameters used in the experiment. When you run the experiment, Experiment Manager sweeps through the hyperparameter values and trains the network multiple times. Each trial uses a different combination of the hyperparameter values specified in the 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. To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor.

In this example, the input to the setup function is a `struct` 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 Image 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.
- **Define Network Architecture** defines the architecture for a convolutional neural network for regression.
- **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.

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.

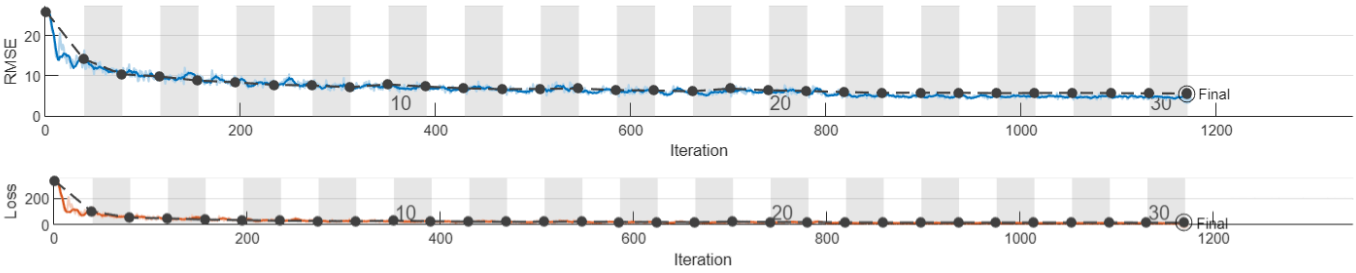
This example includes a metric function `Accuracy` that determines the percentage of angle predictions within an acceptable error margin from the true angles. By default, the function uses a threshold of 10 degrees.

Run Experiment

In the **Experiment Manager** tab, click **Run**. Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameters. 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 7 sec	0.1000	4.0000	4.4200	9.7682	5.5695	15.5097	93.5200
2	Complete	100.0%	0 hr 1 min 36 sec	0.2000	4.0000	5.5942	15.6475	5.9898	17.9387	90.8600
3	Complete	100.0%	0 hr 1 min 41 sec	0.1000	6.0000	3.9594	7.8382	4.7431	11.2487	96.5400
4	Complete	100.0%	0 hr 1 min 38 sec	0.2000	6.0000	4.2466	9.0170	4.8345	11.6863	95.4600
5	Complete	100.0%	0 hr 1 min 32 sec	0.1000	8.0000	3.2151	5.1684	4.4256	9.7931	97.3200
6	Complete	100.0%	0 hr 1 min 41 sec	0.2000	8.0000	3.6607	6.7003	4.6408	10.7685	96.3600

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.

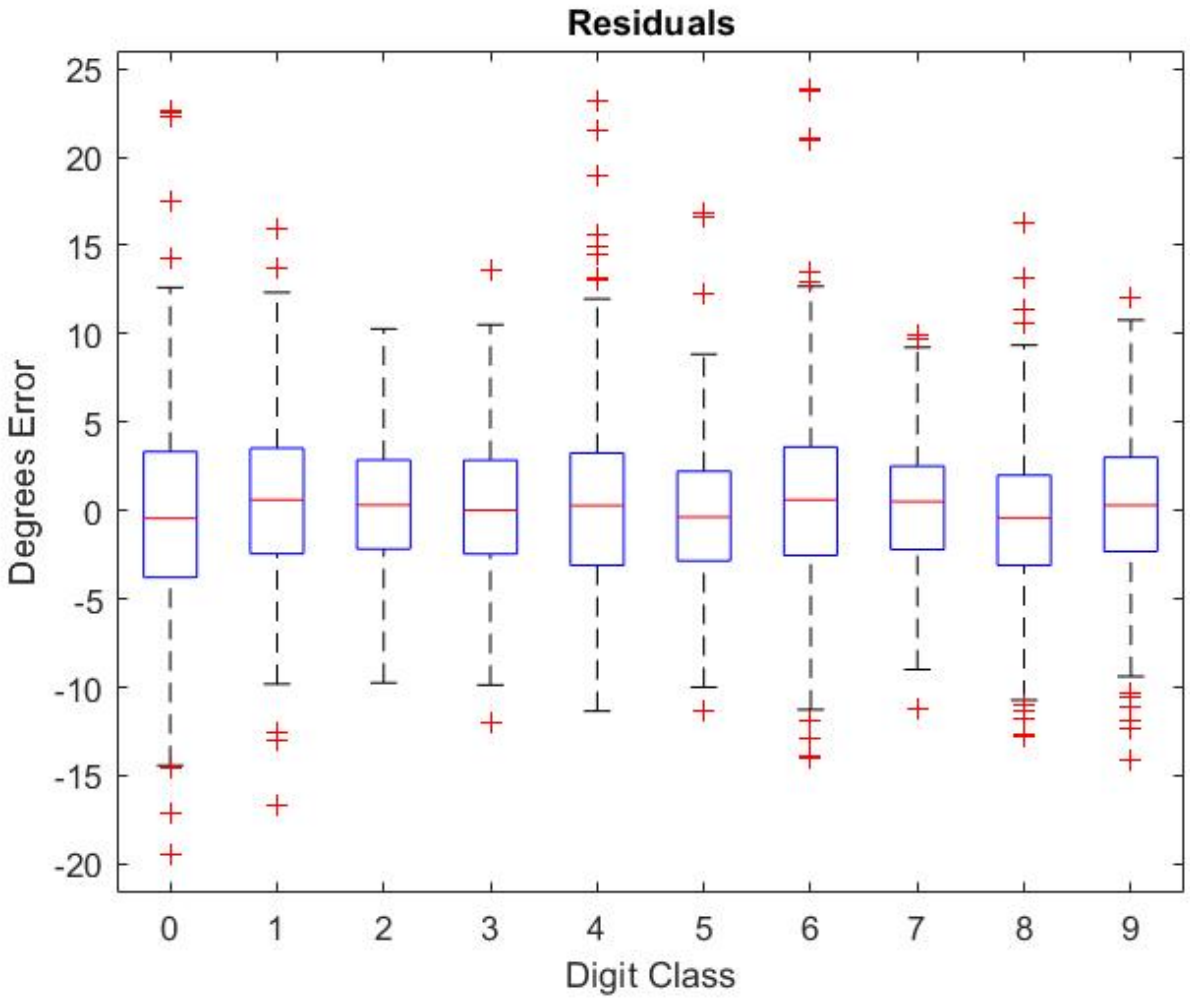
1. Point to the **Accuracy** column.
2. Click the triangle icon.
3. Select **Sort in Descending Order**.

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 32 sec	0.1000	8.0000	3.2151	5.1684	4.4256	9.7931	97.3200
3	Complete	100.0%	0 hr 1 min 41 sec	0.1000	6.0000	3.9594	7.8382	4.7431	11.2487	96.5400
6	Complete	100.0%	0 hr 1 min 41 sec	0.2000	8.0000	3.6607	6.7003	4.6408	10.7685	96.3600
4	Complete	100.0%	0 hr 1 min 38 sec	0.2000	6.0000	4.2466	9.0170	4.8345	11.6863	95.4600
1	Complete	100.0%	0 hr 1 min 7 sec	0.1000	4.0000	4.4200	9.7682	5.5695	15.5097	93.5200
2	Complete	100.0%	0 hr 1 min 36 sec	0.2000	4.0000	5.5942	15.6475	5.9898	17.9387	90.8600

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** tab, click **Export**.
3. In the dialog window, enter the name of a workspace variable for the exported network. The default name is `trainedNetwork`.
4. Create a residual box plot by calling the `plotResiduals` function. Use the exported network as the input to the function. For instance, in the MATLAB Command Window, enter `plotResiduals(trainedNetwork)`.



Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager saves your results and closes all of the experiments contained in the project.

See Also

Experiment Manager | `trainNetwork` | `trainingOptions`

More About

- “Train Convolutional Neural Network for Regression” on page 3-46
- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Evaluate Deep Learning Experiments by Using Metric Functions” on page 6-12

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 deep learning 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.

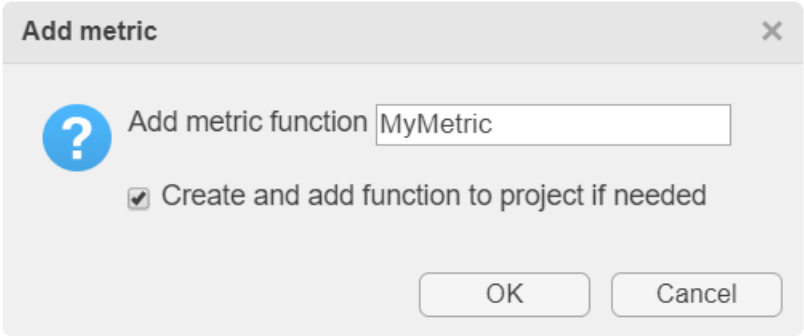
In this example, you train a network to classify images of handwritten digits. Two metric functions determine how well the trained network identifies the images of the numerals one and seven. For more information on using Experiment Manager to train a network for image classification, see “Sweep Hyperparameters to Train a Classification Network”.

Define Metric Functions

Add a metric function to an 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 `struct` with three fields:

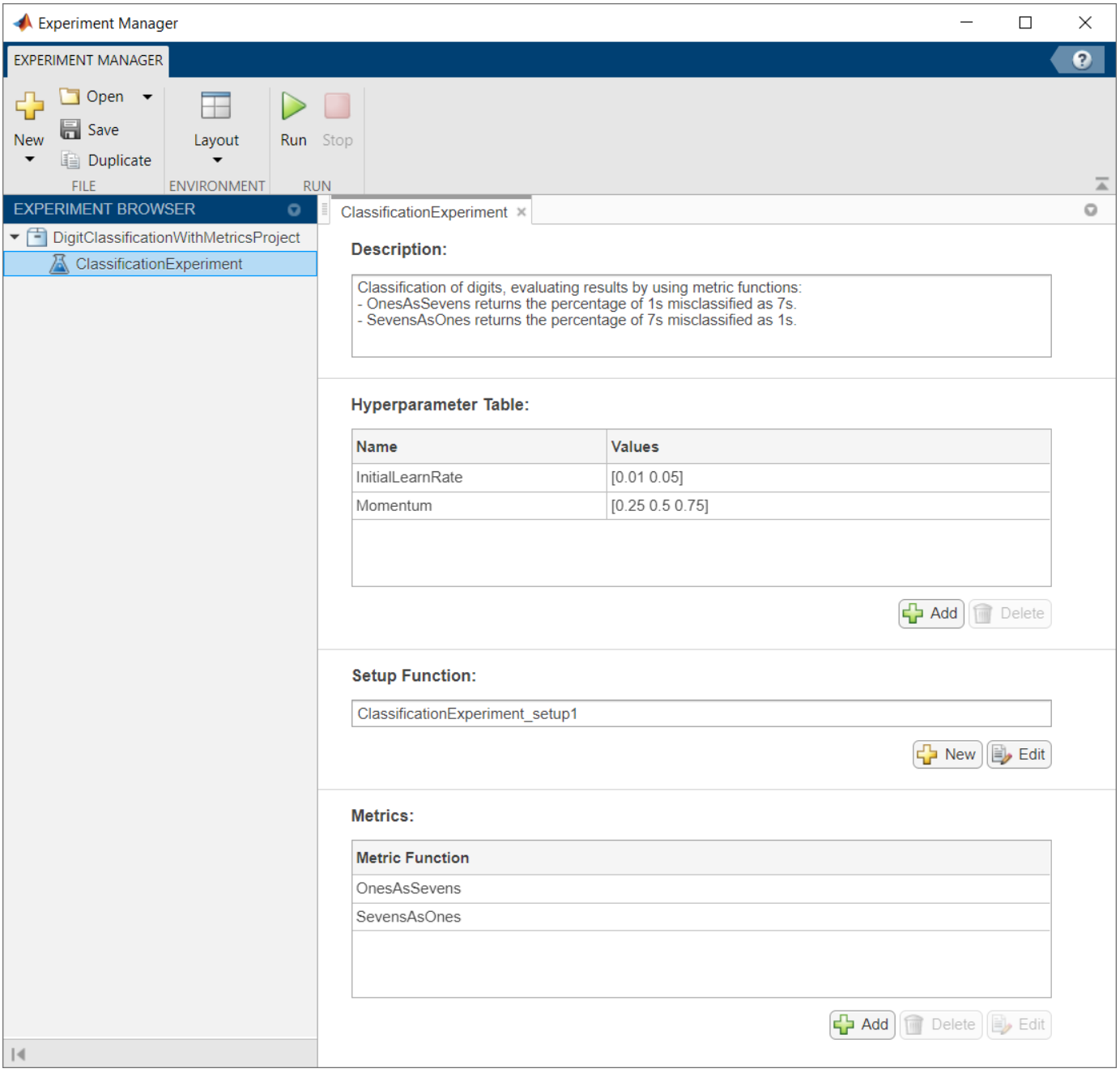
- `trainedNetwork` is the `SeriesNetwork` object or `DAGNetwork` object returned by the `trainNetwork` function. For more information, see “`net`”.
- `trainingInfo` is a `struct` containing the training information returned by the `trainNetwork` function. For more information, see “`info`”.

- `parameters` is a `struct` 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**, double-click the name of the experiment (`ClassificationExperiment`).



An experiment definition consists of a description, a hyperparameter table, a setup function, and a collection of metric functions to evaluate the results of the experiment. For more information, see "Configure Deep Learning Experiment".

The **Description** box 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 **Hyperparameter Table** contains the names and values of the hyperparameters used in the experiment. When you run the experiment, Experiment Manager sweeps through the hyperparameter values and trains the network multiple times. Each trial uses a different combination of the hyperparameter values specified in the table. This example uses the hyperparameters `InitialLearnRate` and `Momentum`.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor.

In this example, the setup function has three sections.

- **Load Image 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 16-108.
- **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.
- **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.

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 percentage of images of the numeral one that the trained network misclassifies as sevens.
- `SevensAsOnes` returns 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'`.

```
function metricOutput = SevensAsOnes(trialInfo)

actualValue = '7';
predValue = '1';
```

```
net = trialInfo.trainedNetwork;

digitDatasetPath = fullfile(matlabroot,'toolbox','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
```

Similarly, the function `SevensAsOnes` computes the number of images with an actual label of `'7'` and a predicted label of `'1'`.

Run Experiment

On the **Experiment Manager** tab, click **Run**. Experiment Manager trains the network defined by the setup function six times. Each trial uses a different combination of hyperparameters. 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 hr 1 min 25 sec	0.0100	0.2500	96.8750	0.1712	56.4400	1.2809	1.0000	2.9000
2	Complete	100%	0 hr 1 min 35 sec	0.0500	0.2500	99.2188	0.0778	57.6800	1.3819	1.1000	2.1000
3	Complete	100%	0 hr 1 min 26 sec	0.0100	0.5000	98.4375	0.1208	59.0400	1.2546	0.8000	2.8000
4	Complete	100%	0 hr 1 min 22 sec	0.0500	0.5000	100.0000	0.0580	57.8800	1.6945	1.4000	1.1000
5	Complete	100%	0 hr 1 min 27 sec	0.0100	0.7500	100.0000	0.0582	61.5600	1.2447	1.2000	1.7000
6	Complete	100%	0 hr 1 min 27 sec	0.0500	0.7500	100.0000	0.0327	56.8800	2.2024	1.0000	1.2000

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
3	Complete	100%	0 hr 1 min 26 sec	0.0100	0.5000	98.4375	0.1208	59.0400	1.2546	0.8000	2.8000
1	Complete	100%	0 hr 1 min 25 sec	0.0100	0.2500	96.8750	0.1712	56.4400	1.2809	1.0000	2.9000
6	Complete	100%	0 hr 1 min 27 sec	0.0500	0.7500	100.0000	0.0327	56.8800	2.2024	1.0000	1.2000
2	Complete	100%	0 hr 1 min 35 sec	0.0500	0.2500	99.2188	0.0778	57.6800	1.3819	1.1000	2.1000
5	Complete	100%	0 hr 1 min 27 sec	0.0100	0.7500	100.0000	0.0582	61.5600	1.2447	1.2000	1.7000
4	Complete	100%	0 hr 1 min 22 sec	0.0500	0.5000	100.0000	0.0580	57.8800	1.6945	1.4000	1.1000

Similiarly, 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
4	Complete	100%	0 hr 1 min 22 sec	0.0500	0.5000	100.0000	0.0580	57.8800	1.6945	1.4000	1.1000
6	Complete	100%	0 hr 1 min 27 sec	0.0500	0.7500	100.0000	0.0327	56.8800	2.2024	1.0000	1.2000
5	Complete	100%	0 hr 1 min 27 sec	0.0100	0.7500	100.0000	0.0582	61.5600	1.2447	1.2000	1.7000
2	Complete	100%	0 hr 1 min 35 sec	0.0500	0.2500	99.2188	0.0778	57.6800	1.3819	1.1000	2.1000
3	Complete	100%	0 hr 1 min 26 sec	0.0100	0.5000	98.4375	0.1208	59.0400	1.2546	0.8000	2.8000
1	Complete	100%	0 hr 1 min 25 sec	0.0100	0.2500	96.8750	0.1712	56.4400	1.2809	1.0000	2.9000

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 6 has the second smallest value for each metric function.

Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager saves your results and closes all of the experiments contained in the project.

See Also

Experiment Manager | `trainNetwork` | `trainingOptions`

More About

- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Create a Deep Learning Experiment for Regression” on page 6-7

Try Multiple Pretrained Networks for Transfer Learning

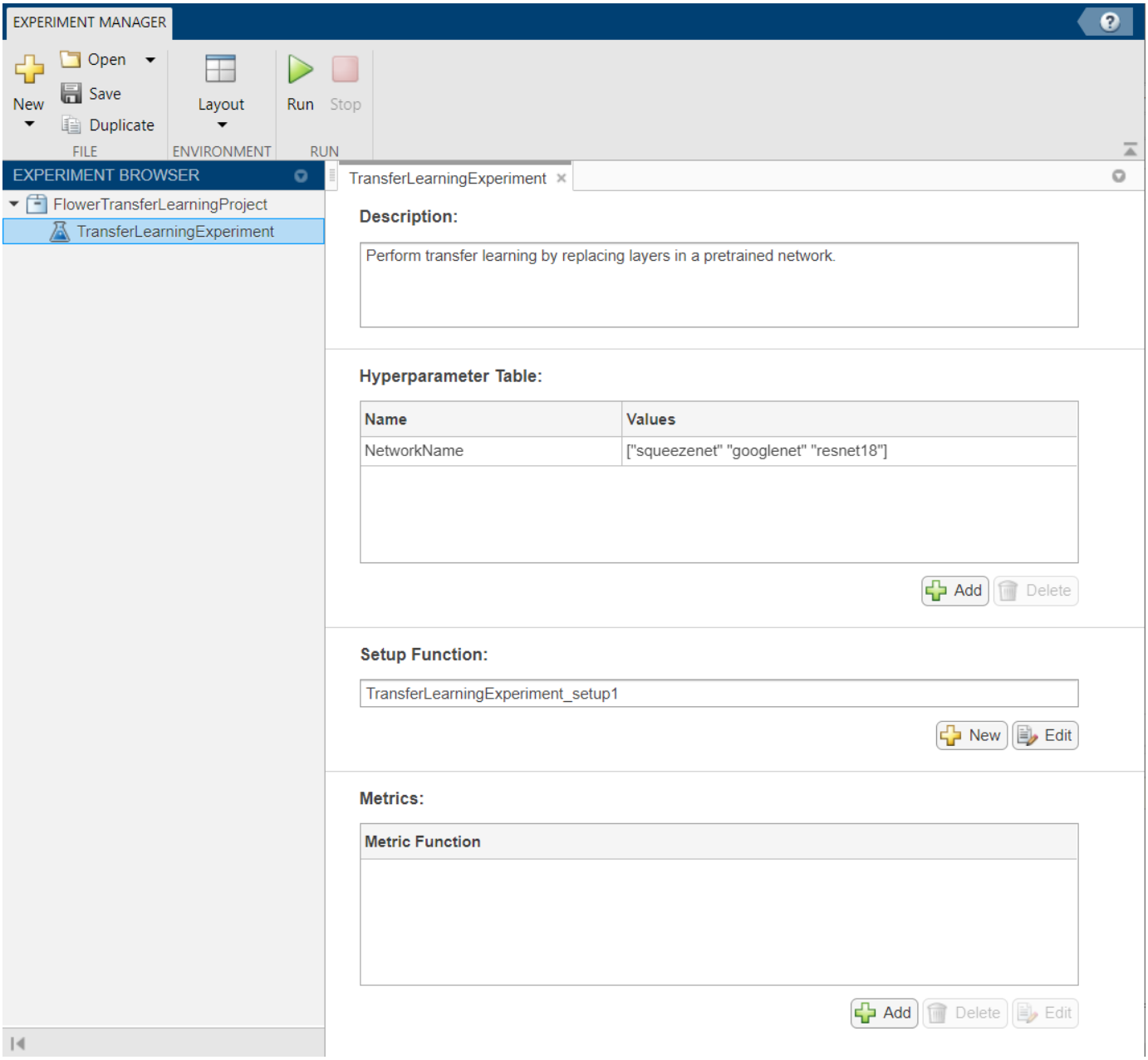
This example shows how to configure an experiment that replaces layers of different pretrained networks for transfer learning. To compare the performance of different pretrained networks for your task, edit this experiment and specify which pretrained networks to use. Before running the experiment, use functions such as `googlenet` to get links to download pretrained networks from the Add-On Explorer.

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. For more information, see “Pretrained Deep Neural Networks” on page 1-12.

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**, double-click the name of the experiment (`TransferLearningExperiment`).



An experiment definition consists of a description, a hyperparameter table, a setup function, and (optionally) a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Deep Learning Experiment”.

The **Description** box contains a textual description of the experiment. For this example, the description is:

```
Perform transfer learning by replacing layers in a pretrained network.
```

The **Hyperparameter Table** contains the names and values of the hyperparameters used in the experiment. When you run the experiment, Experiment Manager sweeps through the hyperparameter values and trains the network multiple times. Each trial uses a different combination of the

hyperparameter values specified in the 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. To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. In this example, the setup function:

- 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 16-108.
- Loads a pretrained network corresponding to the hyperparameter `NetworkName`. The auxiliary function `findLayersToReplace` 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 1-12.
- 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.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any metric functions.

Run Experiment

In the **Experiment Manager** tab, click **Run**. Experiment Manager trains the network defined by the setup function multiple times. Each trial uses a different combination of hyperparameters. 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.

When the experiment finishes, you can sort the results table by column or filter trials by using the **Filters** pane. For more information, see “Sort and Filter 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** tab, select **Export > Trained Network** or **Export > Training Information**, respectively. For more information, see “Output Arguments”.

Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager saves your results and closes all of the experiments contained in the project.

See Also

Experiment Manager | `googlenet` | `trainNetwork` | `trainingOptions`

More About

- “Pretrained Deep Neural Networks” on page 1-12
- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Experiment with Weight Initializers for Transfer Learning” on page 6-20

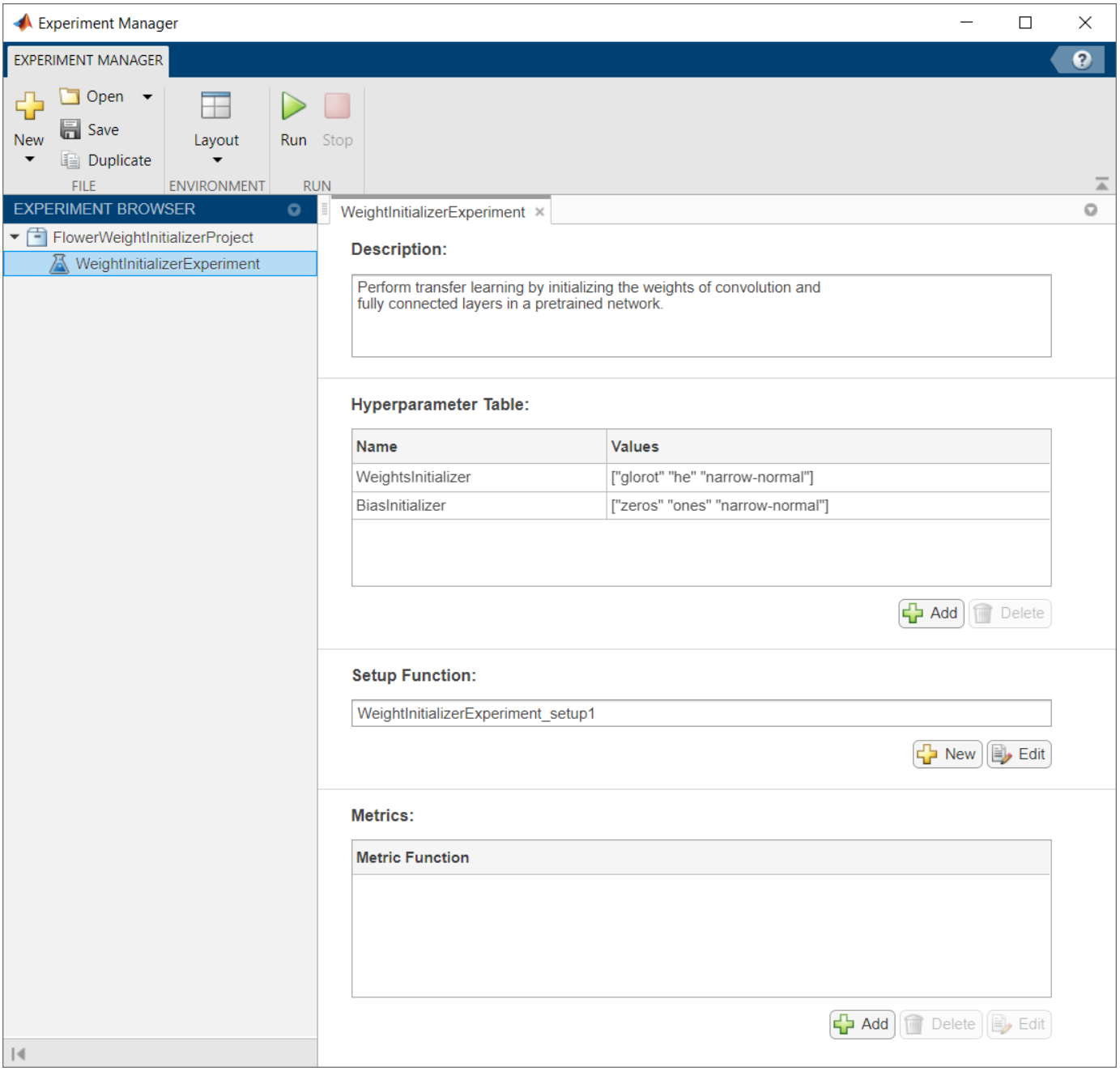
Experiment with Weight Initializers for Transfer Learning

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 15-95.

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**, double-click the name of the experiment (`WeightInitializerExperiment`).



An experiment definition consists of a description, a hyperparameter table, a setup function, and (optionally) a collection of metric functions to evaluate the results of the experiment. For more information, see “Configure Deep Learning Experiment”.

The **Description** box 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 **Hyperparameter Table** contains the names and values of the hyperparameters used in the experiment. When you run the experiment, Experiment Manager sweeps through the hyperparameter values and trains the network multiple times. Each trial uses a different combination of the hyperparameter values specified in the table. This example uses the hyperparameters `WeightsInitializer` and `BiasInitializer` to specify the weight initializers for the convolution and fully connected layers in a pretrained network.

The **Setup Function** configures the training data, network architecture, and training options for the experiment. To inspect the setup function, under **Setup Function**, click **Edit**. The setup function opens in MATLAB Editor. In this example, the setup function:

- 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 16-108.
- Loads a pretrained GoogLeNet network and initializes the input weight in the convolution and fully connected layers by using the initializers specified in the hyperparameter table. The auxiliary function `findLayersToReplace` determines which layers in the network architecture can be modified for transfer learning.
- 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.

The **Metrics** section specifies optional functions that evaluate the results of the experiment. This example does not include any metric functions.

Run Experiment

In the **Experiment Manager** tab, click **Run**. Experiment Manager trains the network defined by the setup function multiple times. Each trial uses a different combination of hyperparameters. 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.

When the experiment finishes, you can sort the results table by column or filter trials by using the **Filters** pane. For more information, see “Sort and Filter 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** tab, select **Export > Trained Network** or **Export > Training Information**, respectively. For more information, see “Output Arguments”.

Close Experiment

In the **Experiment Browser**, right-click the name of the project and select **Close Project**. Experiment Manager saves your results and closes all of the experiments contained in the project.

See Also

Experiment Manager | `trainNetwork` | `trainingOptions`

More About

- “Compare Layer Weight Initializers” on page 15-95

- “Create a Deep Learning Experiment for Classification” on page 6-2
- “Try Multiple Pretrained Networks for Transfer Learning” on page 6-17

Deep Learning in Parallel and the Cloud

Scale Up Deep Learning in Parallel and in the Cloud

In this section...

Deep Learning on Multiple GPUs

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, or rent GPUs in the cloud. 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 any parallel option requires Parallel Computing Toolbox.

Tip GPU support is automatic. By default, the `trainNetwork` function uses a GPU if available.

If you have access to a machine with multiple GPUs, simply specify the training option `'ExecutionEnvironment','multi-gpu'`.

If you want to use more resources, you can scale up deep learning training to the cloud.

Deep Learning Built-In Parallel Support

Training Resource	Settings	Learn More
Single GPU on local machine	Automatic. By default, the `trainNetwork` function uses a GPU if available.	“'ExecutionEnvironment'” “Create Simple Deep Learning Network for Classification” on page 3-40
Multiple GPUs on local machine	Specify `'ExecutionEnvironment','multi-gpu'` with the `trainingOptions` function.	“'ExecutionEnvironment'” “Select Particular GPUs to Use for Training” on page 7-5
Multiple CPU cores on local machine	Specify `'ExecutionEnvironment','parallel'`. With default settings, `'parallel'` uses the local cluster profile. Only use CPUs if you do not have a GPU, because CPUs are generally far slower than GPUs for training.	“'ExecutionEnvironment'”

Training Resource	Settings	Learn More
Cluster or in the cloud	After setting a default cluster, specify `'ExecutionEnvironment','parallel'` with the `trainingOptions` function. Training executes on the cluster and returns the built-in progress plot to your local MATLAB.	"Train Network in the Cloud Using Automatic Parallel Support" on page 7-10

Train Multiple Deep Networks in Parallel

Training Scenario	Recommendations	Learn More
Interactively on your local machine or in the cloud	Use a `parfor` loop to train multiple networks, and plot results using the `OutputFcn`. Runs locally by default, or choose a different cluster profile.	"Use parfor to Train Multiple Deep Learning Networks" on page 7-28
In the background on your local machine or in the cloud	Use `parfeval` to train without blocking your local MATLAB, and plot results using the `OutputFcn`. Runs locally by default, or choose a different cluster profile.	"Run Multiple Deep Learning Experiments in Parallel" on page 5-44 "Use parfeval to Train Multiple Deep Learning Networks" on page 7-14
On a cluster, and turn off your local machine	Use the `batch` function to send training code to the cluster. You can close MATLAB and fetch results later.	"Send Deep Learning Batch Job to Cluster" on page 7-21

Deep Learning in the Cloud

If your deep learning training takes hours or days, you can rent high performance GPUs in the cloud to accelerate training. Working in the cloud requires some initial setup, but after the initial setup using the cloud can reduce training time, or allow you to train more networks in the same time. To try deep learning in the cloud, you can follow example steps to set up your accounts, copy your data into the cloud, and create a cluster. After this initial setup, you can run your training code with minimal changes to run in the cloud. After setting up your default cluster, simply specify the training option `'ExecutionEnvironment','parallel'` to train networks on your cloud cluster on multiple GPUs.

Configure Deep Learning in the Cloud	Notes	Learn More
Set up MathWorks Cloud Center and Amazon accounts	One-time setup.	Getting Started with Cloud Center

Configure Deep Learning in the Cloud	Notes	Learn More
Create a cluster	Use Cloud Center to set up and run clusters in the Amazon cloud. For deep learning, choose a machine type with GPUs such as the P2 or G3 instances.	Create a Cloud Cluster
Upload data to the cloud	To work with data in the cloud, upload to Amazon S3. Use datastores to access the data in S3 from your desktop client MATLAB, or from your cluster workers, without changing your code.	“Upload Deep Learning Data to the Cloud” on page 7-35

Advanced Support for Fast Multi-Node GPU Communication

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 Knobs.

See Also

More About

- “Deep Learning with MATLAB on Multiple GPUs” on page 7-5
- “Run Multiple Deep Learning Experiments in Parallel” on page 5-44
- “Send Deep Learning Batch Job to Cluster” on page 7-21
- “Use parfeval to Train Multiple Deep Learning Networks” on page 7-14
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-28
- “Upload Deep Learning Data to the Cloud” on page 7-35

Deep Learning with MATLAB on Multiple GPUs

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

If you have access to a machine with multiple GPUs, you can simply specify the training option `'multi-gpu'`.

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 and in the Cloud” on page 7-2. To try an example, see “Train Network in the Cloud Using Automatic Parallel Support” on page 7-5.

Select Particular GPUs to Use for Training

To use all available GPUs on your machine, simply specify the training option `'ExecutionEnvironment','multi-gpu'`.

To select one of multiple GPUs to use to train a single model, use:

```
gpuDevice(index)
```

If you want to train a single model using multiple GPUs, and do not want to use all your GPUs, open the parallel pool in advance, and select the GPUs manually. To select particular GPUs, use the following code, where `gpuIndices` are the indices of the GPUs:

```
parpool('local', numel(gpuIndices));
spmd, gpuDevice(gpuIndices(labindex)); end
```

When you run `trainNetwork` with the `‘multi-gpu’` `ExecutionEnvironment` (or `‘parallel’` for the same result), the training function will use this pool and not open a new one.

Another option is to select workers using the `‘WorkerLoad’` option in `trainingOptions`. For example:

```
parpool('local', 5);
opts = trainingOptions('sgdm', 'WorkerLoad', [1 1 1 0 1], ...)
```

In this case, the 4th worker is part of the pool but idle, which is not an ideal use of the parallel resources. It is more efficient to specify GPUs with `gpuDevice`.

If you want to train multiple models with one GPU each, start a MATLAB session for each and select a device using `gpuDevice`.

Alternatively, use a `parfor` loop:

```
parfor i=1:gpuDeviceCount
 trainNetwork(…);
end
```

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 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 7-35.

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 7-35.

```
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
];
```

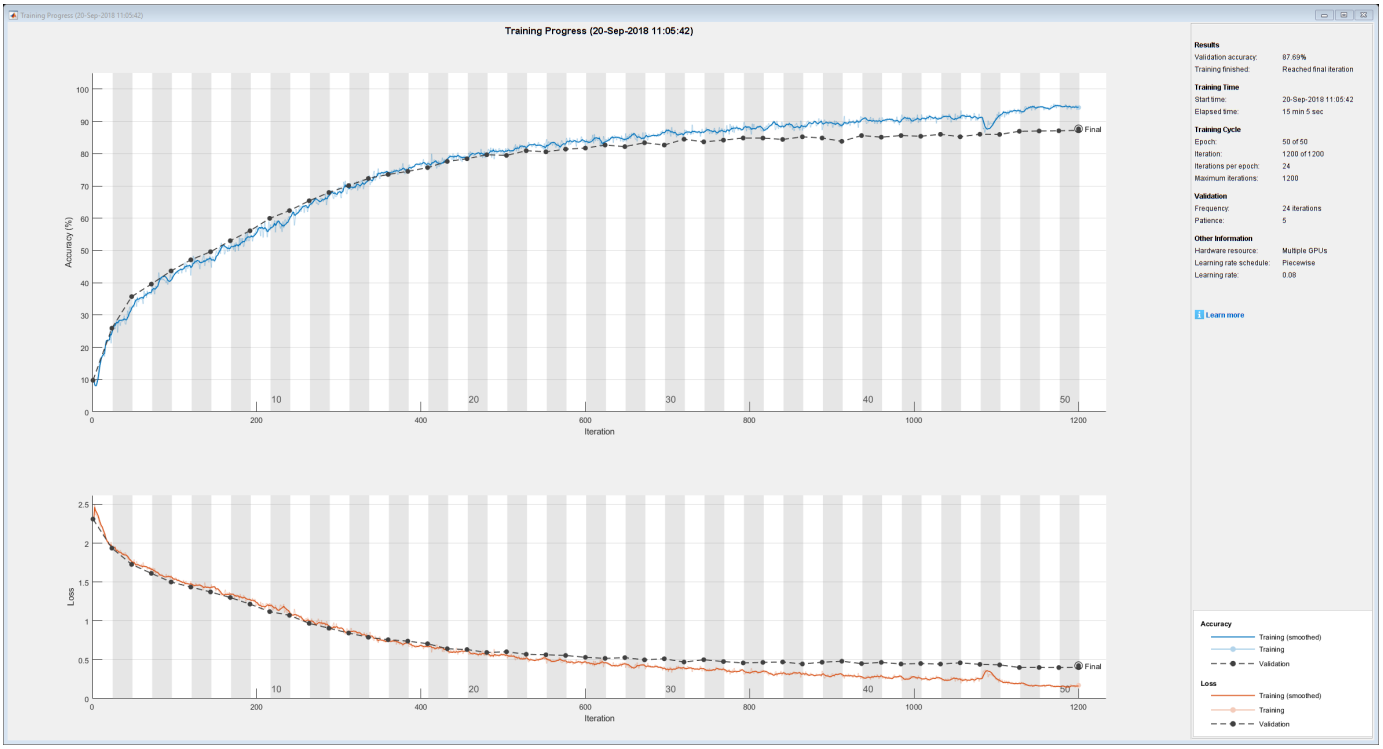
Define the training options. Train the network in parallel using the current cluster, by setting the execution environment to `parallel`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU 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

`gpuDevice` | `imageDatastore` | `spmd` | `trainNetwork` | `trainingOptions`

Related Examples

- “Run Multiple Deep Learning Experiments in Parallel” on page 5-44
- “Upload Deep Learning Data to the Cloud” on page 7-35
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-28
- “Scale Up Deep Learning in Parallel and in the Cloud” on page 7-2

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 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 7-35.

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 7-35.

```
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
];
```

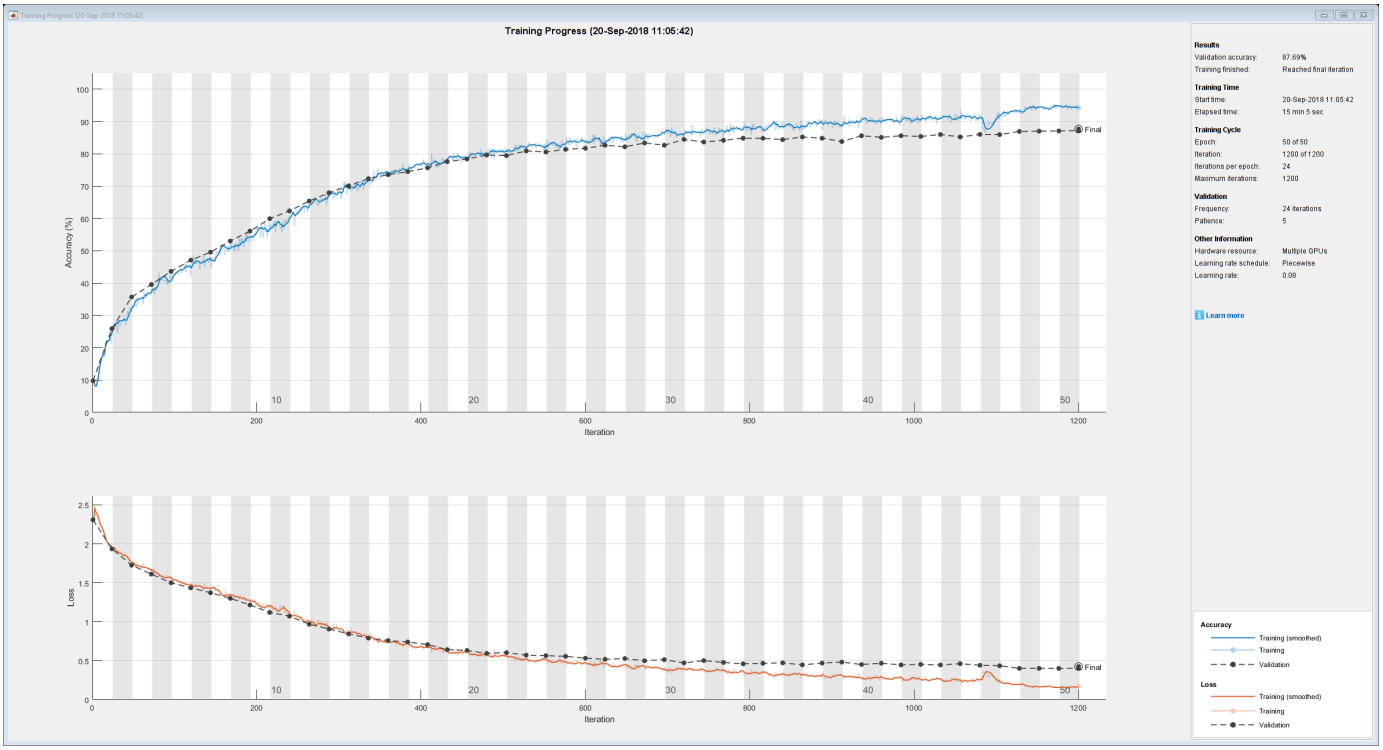
Define the training options. Train the network in parallel using the current cluster, by setting the execution environment to `parallel`. When you use multiple GPUs, you increase the available computational resources. Scale up the mini-batch size with the number of GPUs to keep the workload on each GPU 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

`imageDatastore` | `trainNetwork` | `trainingOptions`

Related Examples

- “Upload Deep Learning Data to the Cloud” on page 7-35
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-28

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 7-35.

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 7-35.

```
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
```

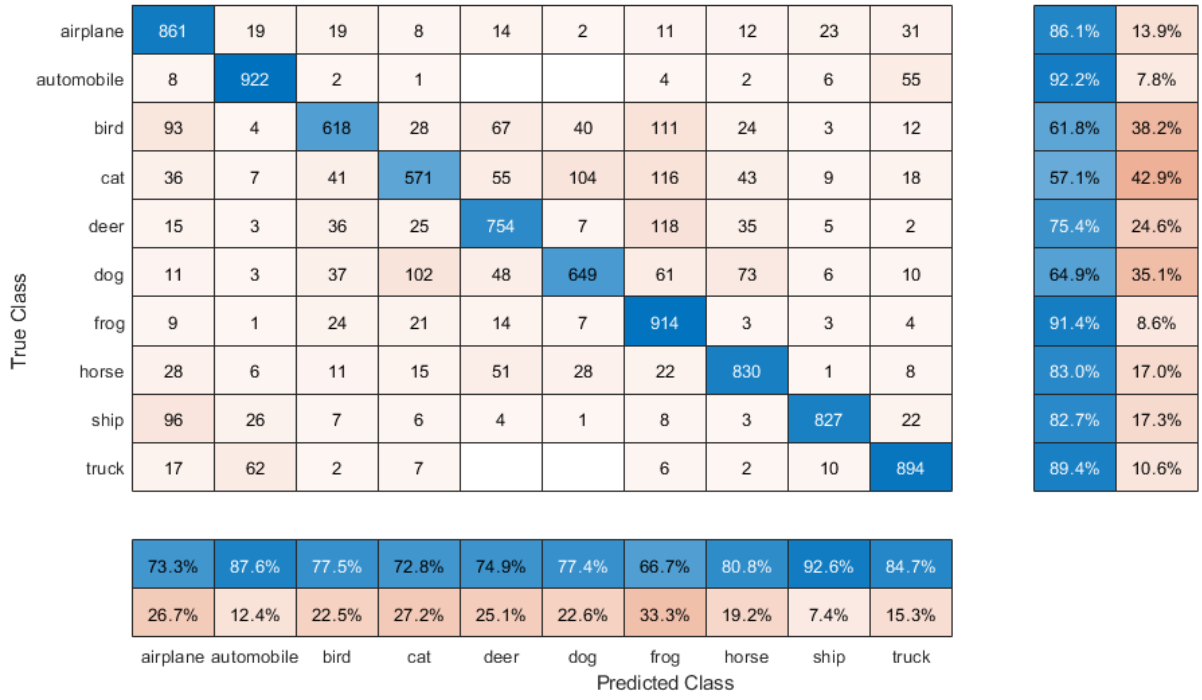
Select the best network in terms of accuracy. Test its performance against the test data set.

```
[~, 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.

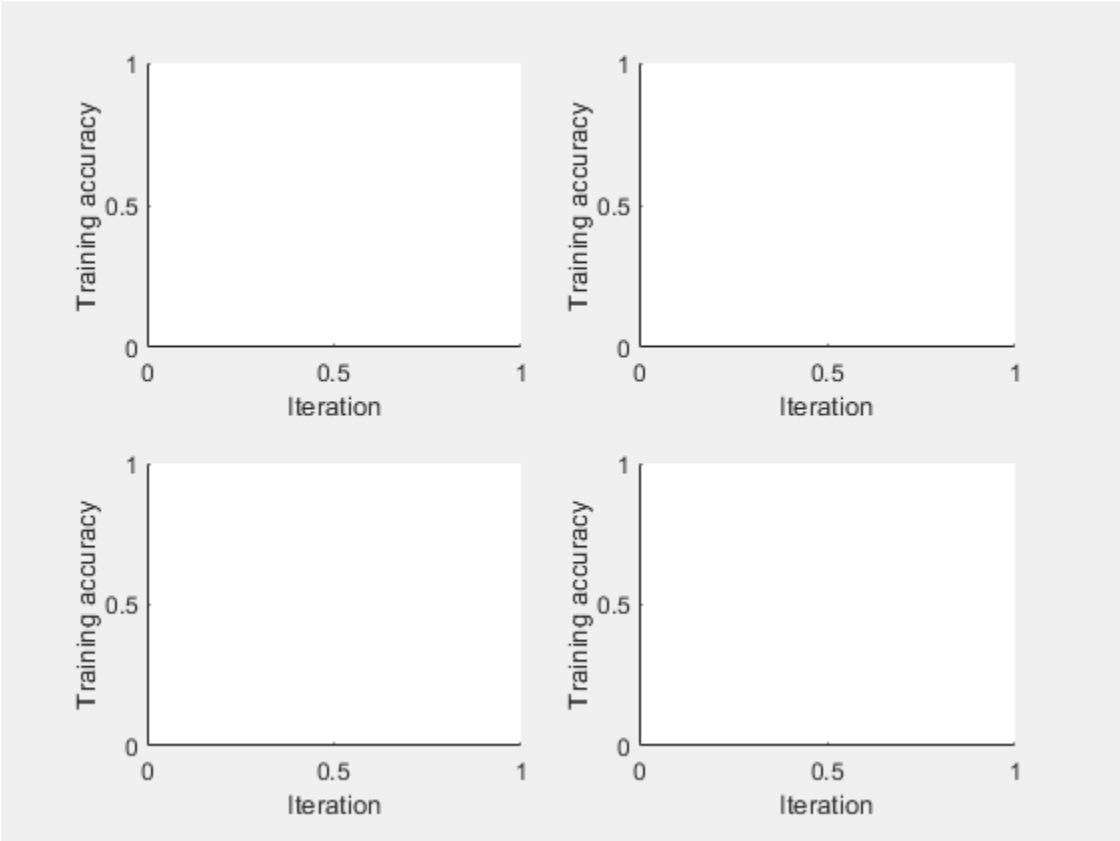
```
figure('Units','normalized','Position',[0.2 0.2 0.4 0.4]);
confusionchart(imdsTest.Labels,YPredicted,'RowSummary','row-normalized','ColumnSummary','column-n
```



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.

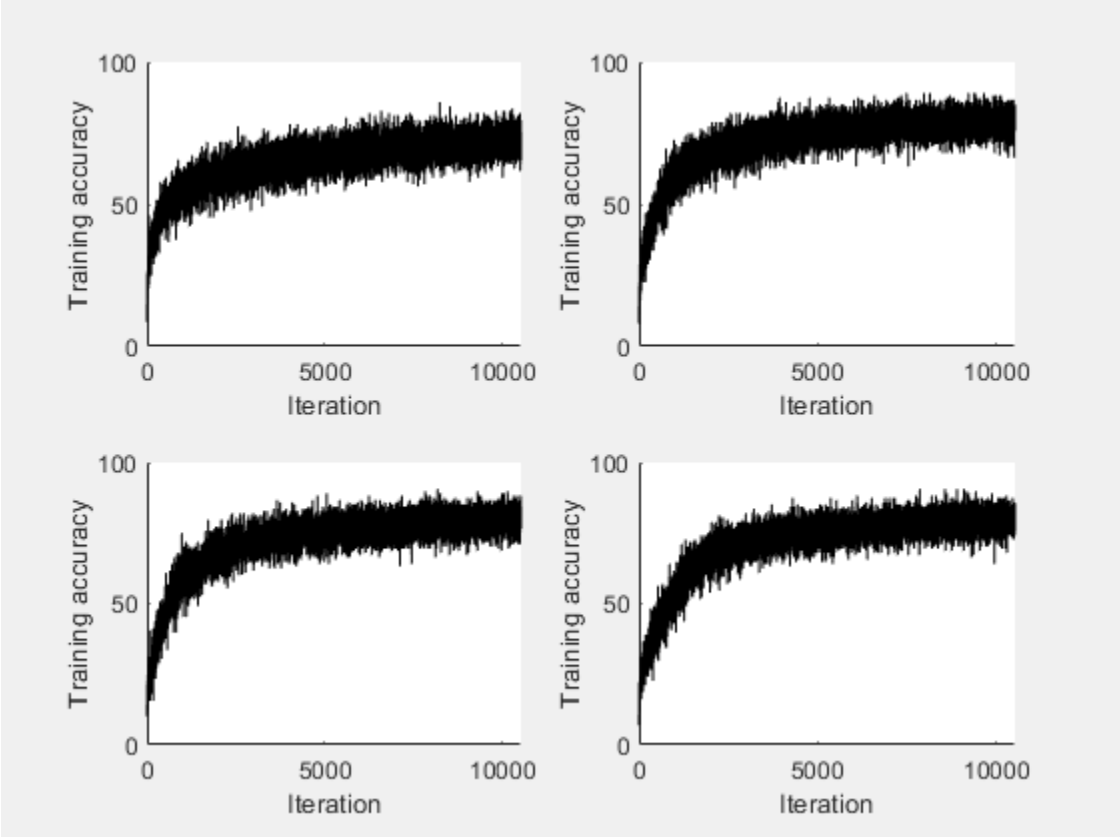
```
netDepths = 1:4;
addAttachedFiles(gcp,mfilename);
for idx = 1:numel(netDepths)

    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

`afterEach` | `imageDatastore` | `parfeval` | `trainNetwork` | `trainingOptions`

Related Examples

- “Train Network in the Cloud Using Automatic Parallel Support” on page 7-10
- “Use parfor to Train Multiple Deep Learning Networks” on page 7-28
- “Upload Deep Learning Data to the Cloud” on page 7-35

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 7-28 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 7-35.

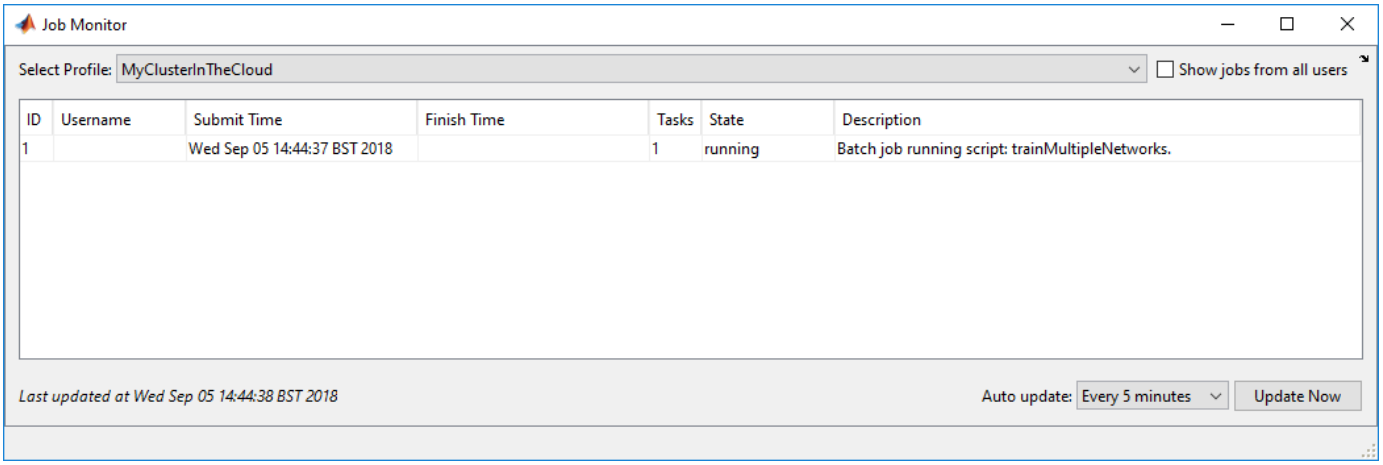
Submit Batch Job

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 7-28. 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.

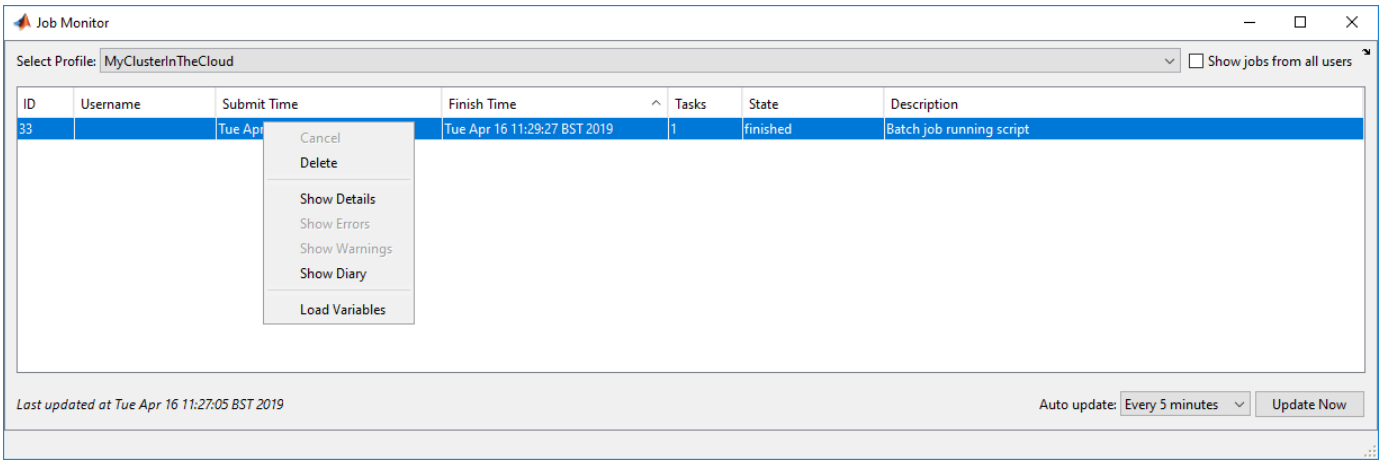
```
delete(job1);
```

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 7-28
- “Upload Deep Learning Data to the Cloud” on page 7-35

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 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.

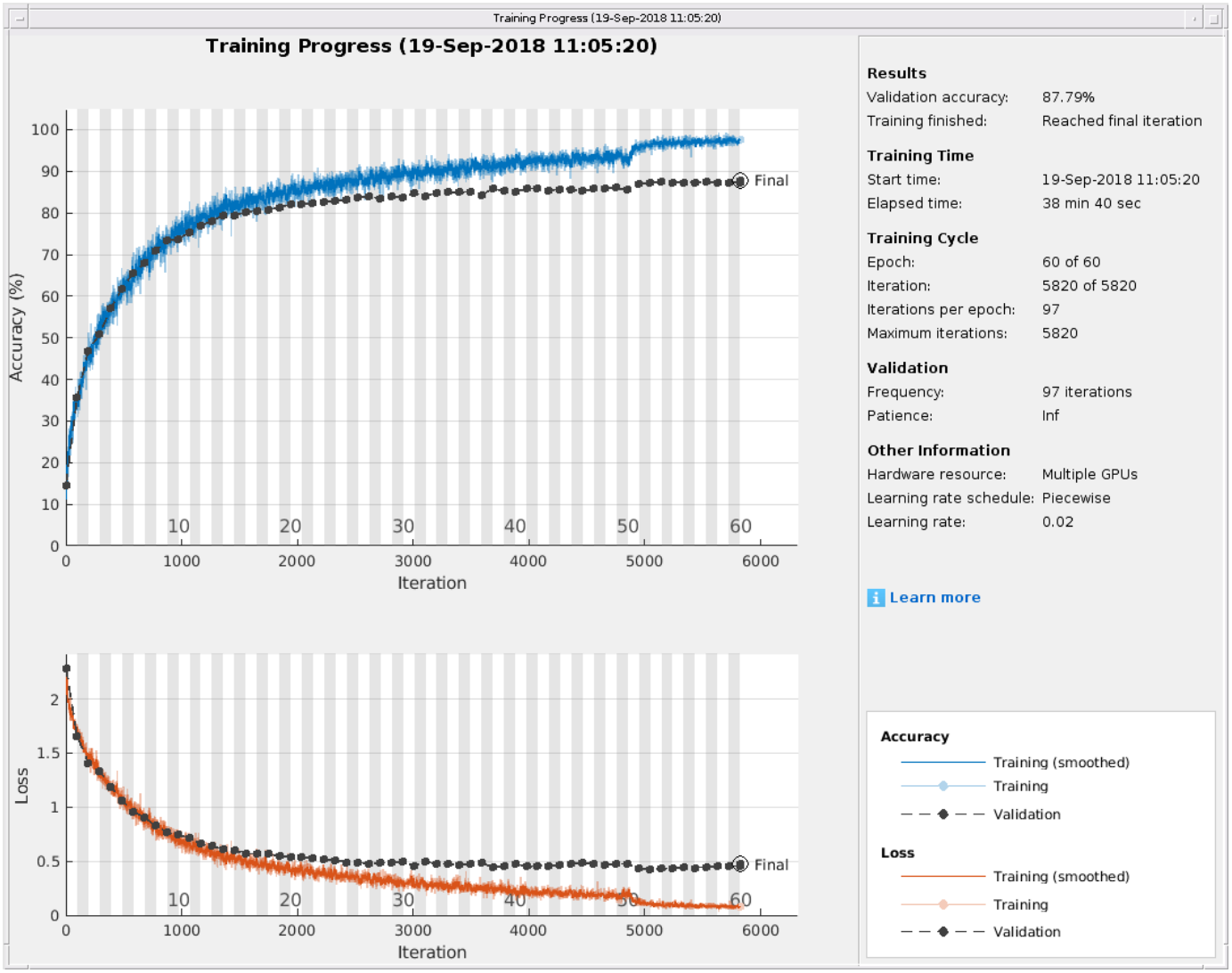
```
numGPUs = 2;
miniBatchSize = 256*numGPUs;
initialLearnRate = 1e-1*miniBatchSize/256;

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).
```



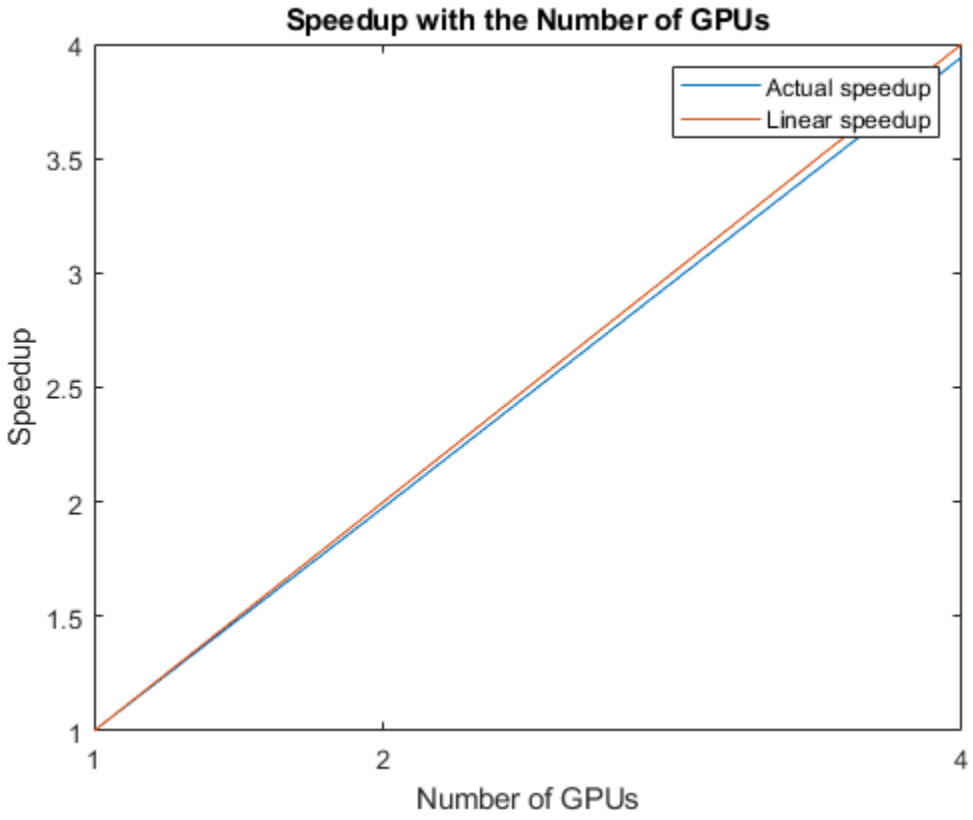

```
net =
  SeriesNetwork with properties:

    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

`imageDatastore` | `trainNetwork` | `trainingOptions`

Related Examples

- “Train Network in the Cloud Using Automatic Parallel Support” (Parallel Computing Toolbox)
- “Scale Up Deep Learning in Parallel 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 7-21.

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 7-35.

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 7-35.

```
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)
```

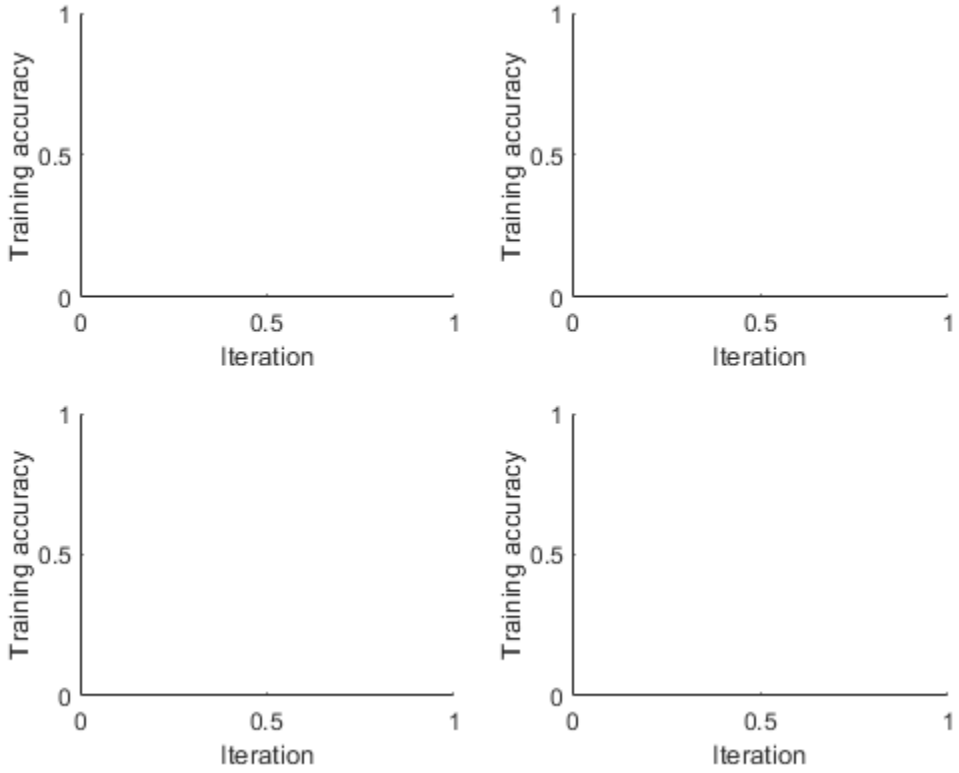
```
accuracy = 0.8173
```

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;
```

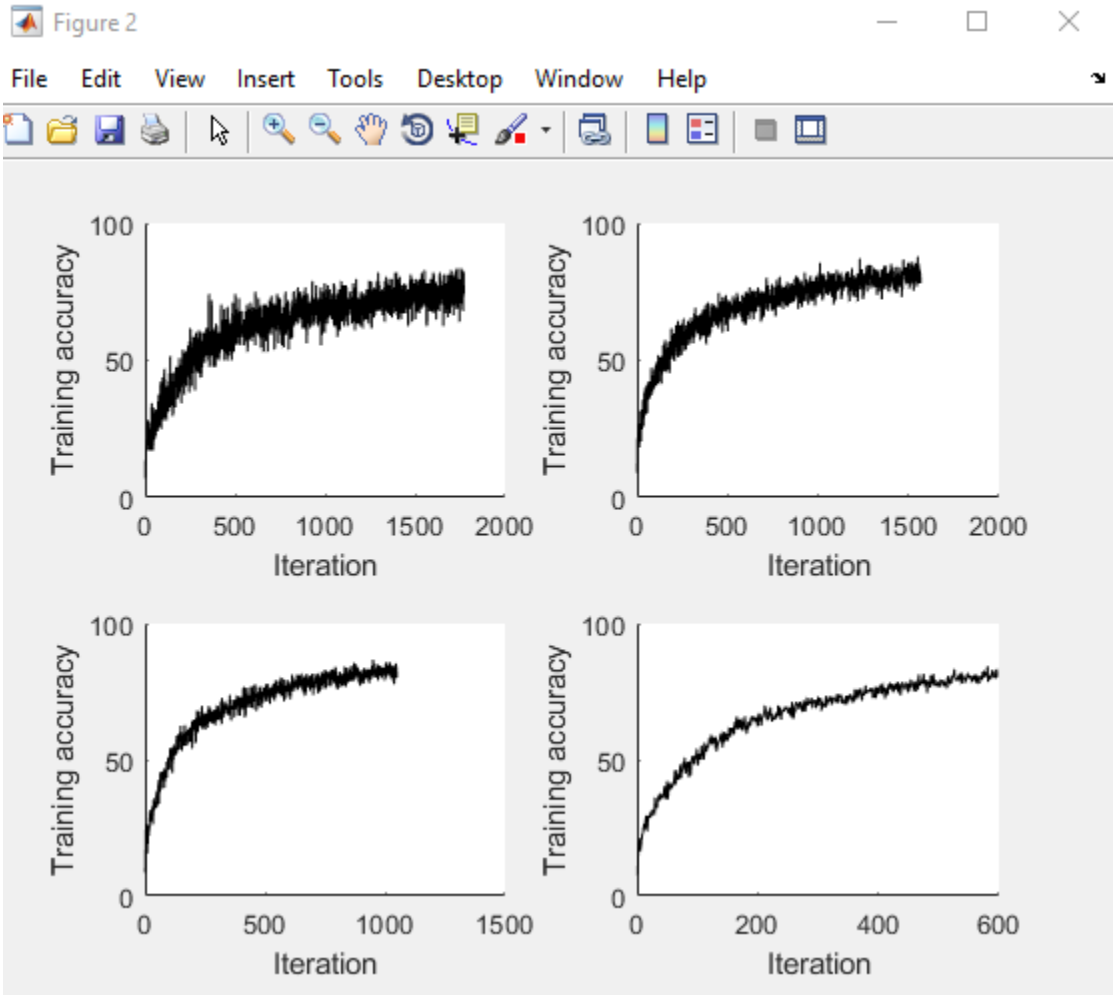
```
for i=1:4
    subplot(2,2,i)
    xlabel('Iteration');
    ylabel('Training accuracy');
    lines(i) = animatedline;
end
```



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

`imageDatastore` | `parallel.pool.DataQueue` | `trainNetwork`

Related Examples

- “Upload Deep Learning Data to the Cloud” on page 7-35
- “Send Deep Learning Batch Job to Cloud” on page 7-21

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 7-28

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.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');

[imdsTrain,imdsTest] = splitEachLabel(imds,0.9,"randomized");
```

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'}
```

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;
    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: 2).
```

Get the number of workers in the parallel pool. Later in this example, you divide the workload according to this number.

```
N = pool.NumWorkers;
```

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;
afterEach(Q,@displayTrainingProgress);
```

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 “Training with Multiple GPUs” on page 1-9.

```
if executionEnvironment == "gpu"
    miniBatchSize = miniBatchSize .* N
end
```

```
miniBatchSize = 256
```

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 = 1×2

   128   128
```

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, and set `ReadSize` to the mini-batch size of the worker.

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 datastore by using the `read` function, and concatenate the retrieved images into a four-dimensional array of images. Normalize the images so that the pixels take values between `0` and `1`.
- Convert the labels to a matrix of dummy variables that puts labels against observations. Dummy variables contain `1` for the label of the observation and `0` otherwise.
- Convert the mini-batch of data to a `dlarray` object with the underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch). For GPU training, convert the data to `gpuArray`.
- 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.
- To aggregate and update the network state, use the `dlupdate` function with the `aggregateState` function. `aggregateState` is a supporting function defined at the end of this example. The function uses `gplus` to add together and replicate the state across all 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. The batchnormalization 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.

```
spmd
    % Partition datastore.
    workerImds = partition(imdsTrain,N,labindex);
    workerImds.ReadSize = workerMiniBatchSize(labindex);

    workerVelocity = velocity;

    iteration = 0;

    for epoch = 1:numEpochs
        % Reset and shuffle the datastore.
        reset(workerImds);
        workerImds = shuffle(workerImds);

        % Loop over mini-batches.
        while gop(@and,hasdata(workerImds))
            iteration = iteration + 1;

            % Read a mini-batch of data.
            [workerXBatch,workerTBatch] = read(workerImds);
            workerXBatch = cat(4,workerXBatch{:});
            workerNumObservations = numel(workerTBatch.Label);

            % Normalize the images.
            workerXBatch =  single(workerXBatch) ./ 255;

            % Convert the labels to dummy variables.
            workerY = zeros(numClasses,workerNumObservations,'single');
            for c = 1:numClasses
                workerY(c,workerTBatch.Label==classes(c)) = 1;
            end

            % Convert the mini-batch of data to dlarray.
            dlworkerX = dlarray(workerXBatch,'SSCB');

            % If training on GPU, then convert data to gpuArray.
            if executionEnvironment == "gpu"
                dlworkerX = gpuArray(dlworkerX);
            end

            % 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
            workerState.Value = dlupdate(@aggregateState,workerState.Value,{workerNormalizationFa
            dlnet.State = workerState;

            % 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];
            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 = 0.9990
```

Define Helper 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
```

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)
disp("Epoch: " + data(1) + ", Loss: " + data(2));
end
```

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`. Passing the class of the contents as the second input argument to `gplus` ensures, if the data is a `gpuArray`, that fast communication is enabled where available.

```
function gradients = aggregateGradients(dlgradients,factor)
gradients = extractdata(dlgradients);
gradients = gplus(factor*gradients,class(gradients));
end
```

Define a function that aggregates the network state on all workers. The network state contains the batch normalization statistics of the data set that are calculated as a weighted average of the mean and variance across the training iterations. 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.

```
function state = aggregateState(state, factor)
state = gplus(factor*state);
end
```

See Also

`crossentropy` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `dlupdate` | `forward` | `predict` | `sgdmupdate` | `softmax`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173

- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

Computer Vision Examples

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.

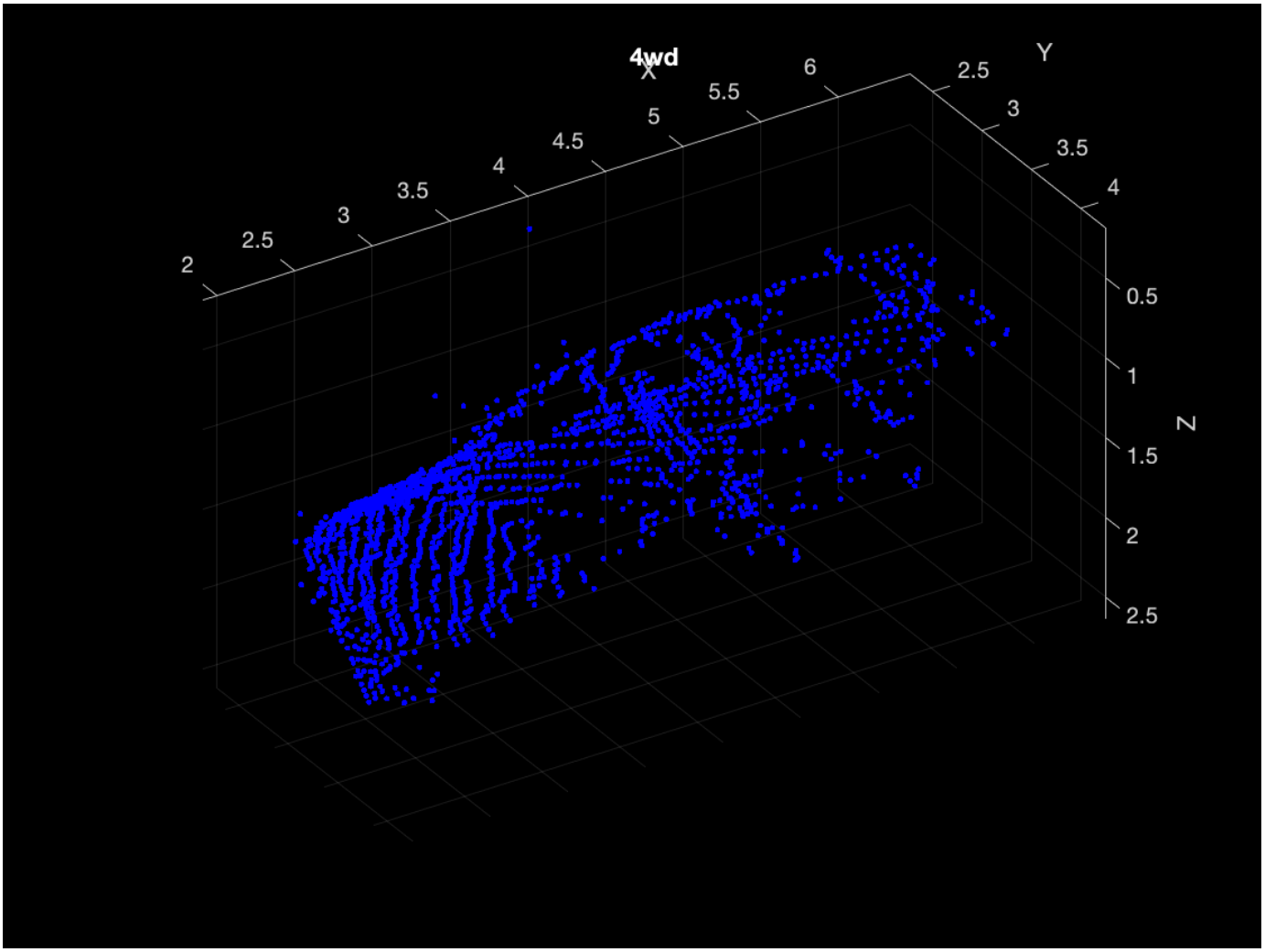
```
downloadDirectory = tempdir;
datapath = downloadSydneyUrbanObjects(downloadDirectory);
```

Load the downloaded training and validation data set using the `sydneyUrbanObjectsClassificationDatastore` 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 = sydneyUrbanObjectsClassificationDatastore(datapath,foldsTrain);
dsVal = sydneyUrbanObjectsClassificationDatastore(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 0 1],"MarkerSize",40,"VerticalAxisDir","down")
xlabel("X")
ylabel("Y")
zlabel("Z")
title(label)
```

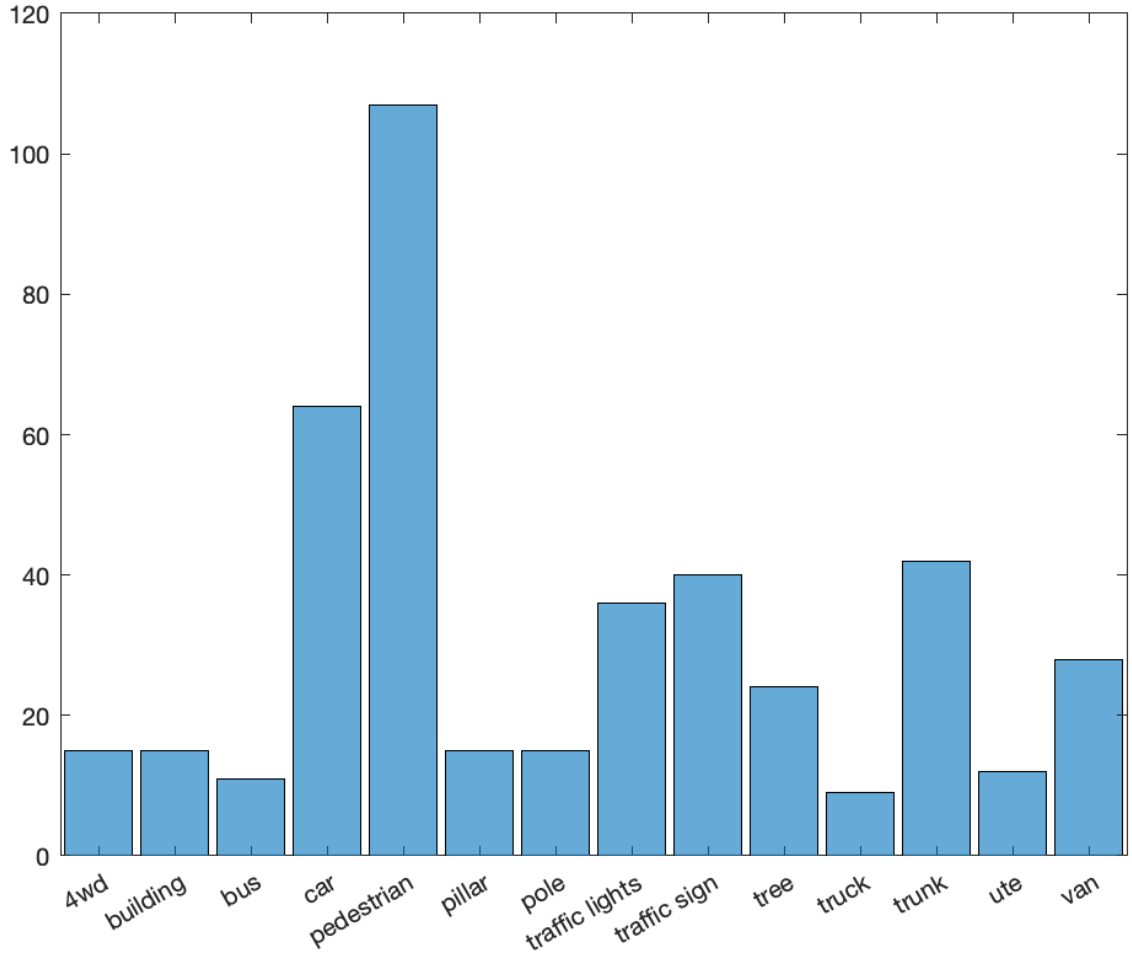


Read the labels and count the number of points assigned to each label to better understand the distribution of labels within the data set.

```
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;
```

Shuffle the replicated file list so that the same classes of observations are not in every training batch.

```
dsTrain.Files = dsTrain.Files(randperm(length(dsTrain.Files)));
```

Set the mini-batch size of the training and validation datastores to 128.

```
dsTrain.MiniBatchSize = 128;
dsVal.MiniBatchSize = 128;
```

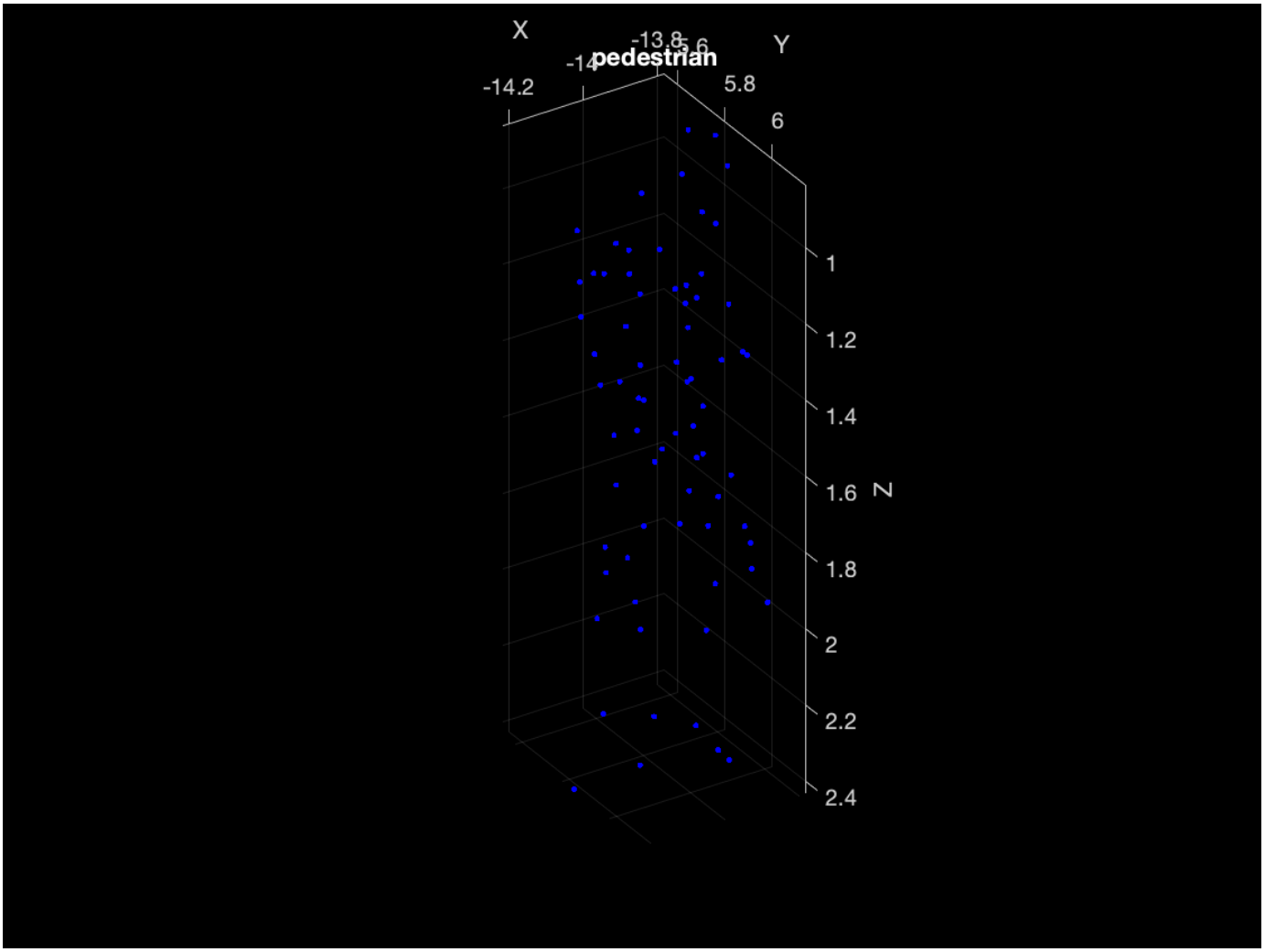
Data Augmentation

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 0 1],"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.

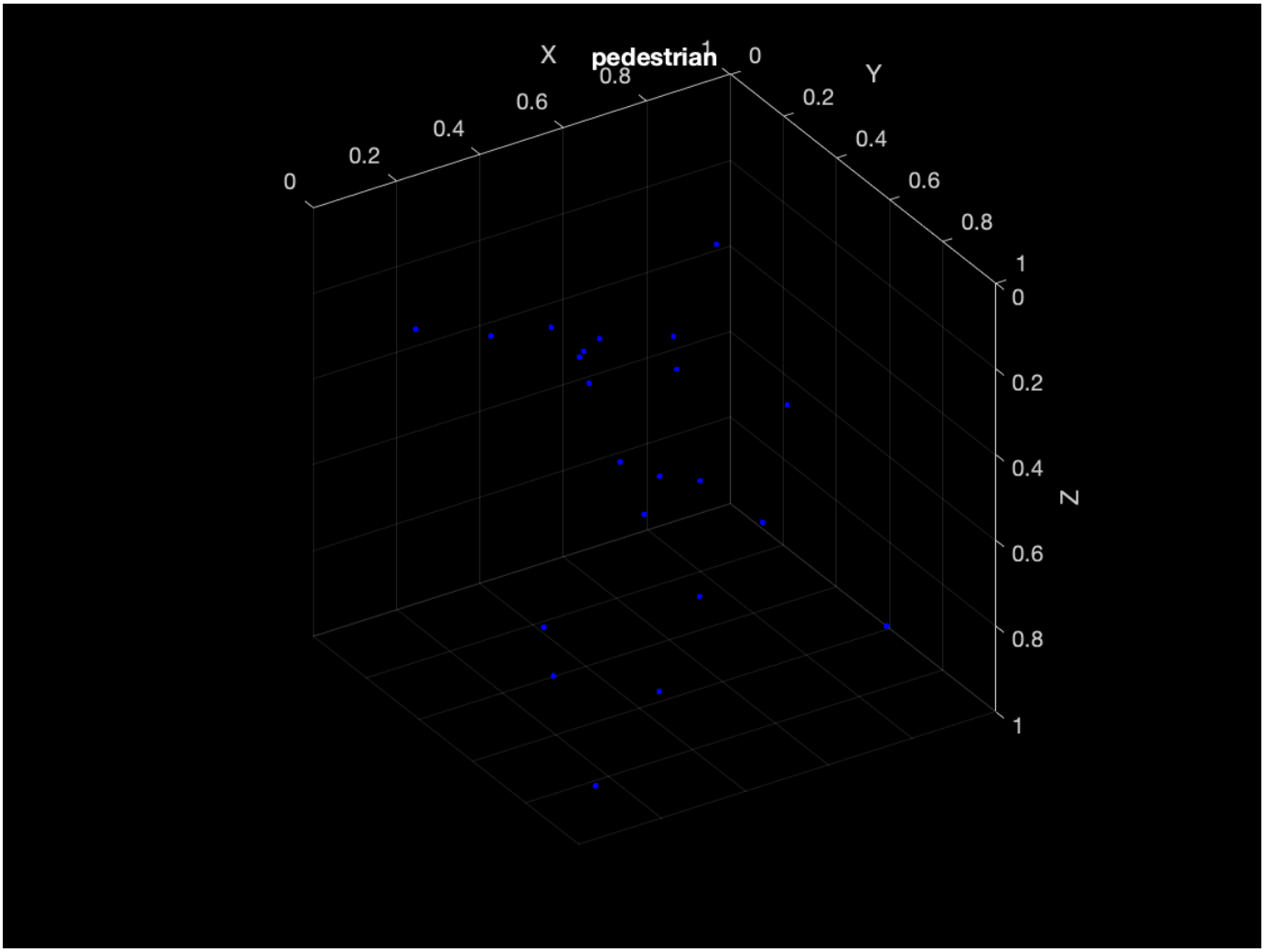
```
numPoints = 1024;
dsTrain = transform(dsTrain,@(data)selectPoints(data,numPoints));
dsVal = transform(dsVal,@(data)selectPoints(data,numPoints));
```

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 0 1],"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 40 epochs. Set the initial learning rate to 0.001 and the L2 regularization factor to 0.01.

```
numEpochs = 40;
learnRate = 0.001;
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.

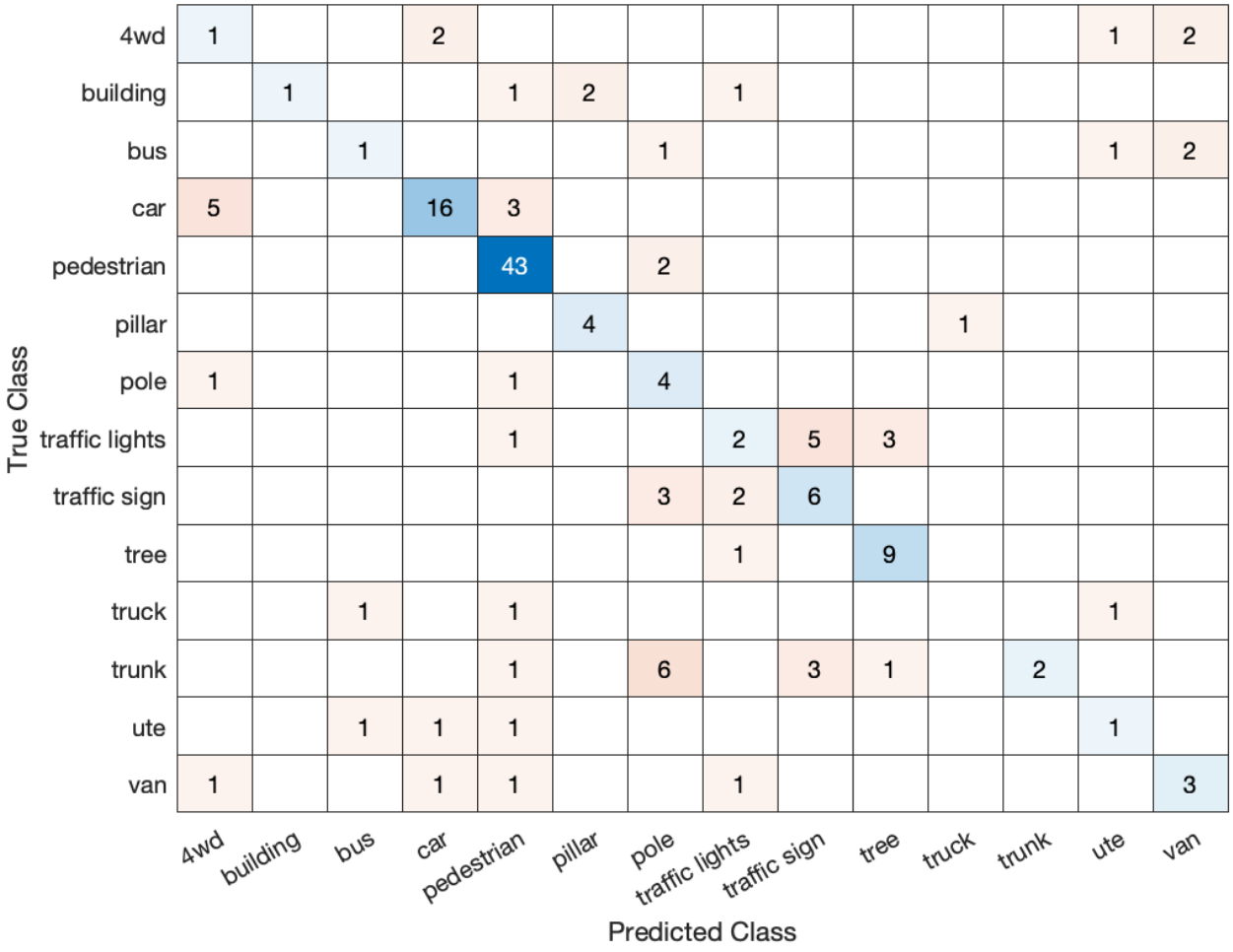
Note that training was verified on an NVIDIA Titan X with 12 GB of GPU memory. If your GPU has less memory, you may run out of memory during training. If this happens, lower the miniBatchSize. Training this network takes about 5 minutes. Depending on your GPU hardware, it can take longer.

```
doTraining = false;

if doTraining

    % 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

        % Reset training and validation datastores.
        reset(dsTrain);
        reset(dsVal);

        % Iterate through data set.
        while hasdata(dsTrain)
            iteration = iteration + 1;

            % Read data.
            data = read(dsTrain);

            % Create batch.
            [XTrain,YTrain] = batchData(data);

            % 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(dsVal)

            % Get the next batch of data.
            data = read(dsVal);
```

```
            % Create batch.
            [XVal,YVal] = batchData(data);

            % 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 datastores.
        reset(dsTrain);
        reset(dsVal);
    end

else
    % Download pretrained model parameters, model state, and validation
    % results.
    pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/pretrainedPointNet.mat';

    pretrainedNetwork = fullfile(pwd,'pretrainedPointNet.mat');
    if ~exist(pretrainedNetwork,'file')
        disp('Downloading pretrained network (5 MB)...');
        websave(pretrainedNetwork,pretrainedURL);
    end

    % Load pretrained model.
    pretrainedResults = load('pretrainedPointNet.mat');
    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 = 0.6000
```

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`.

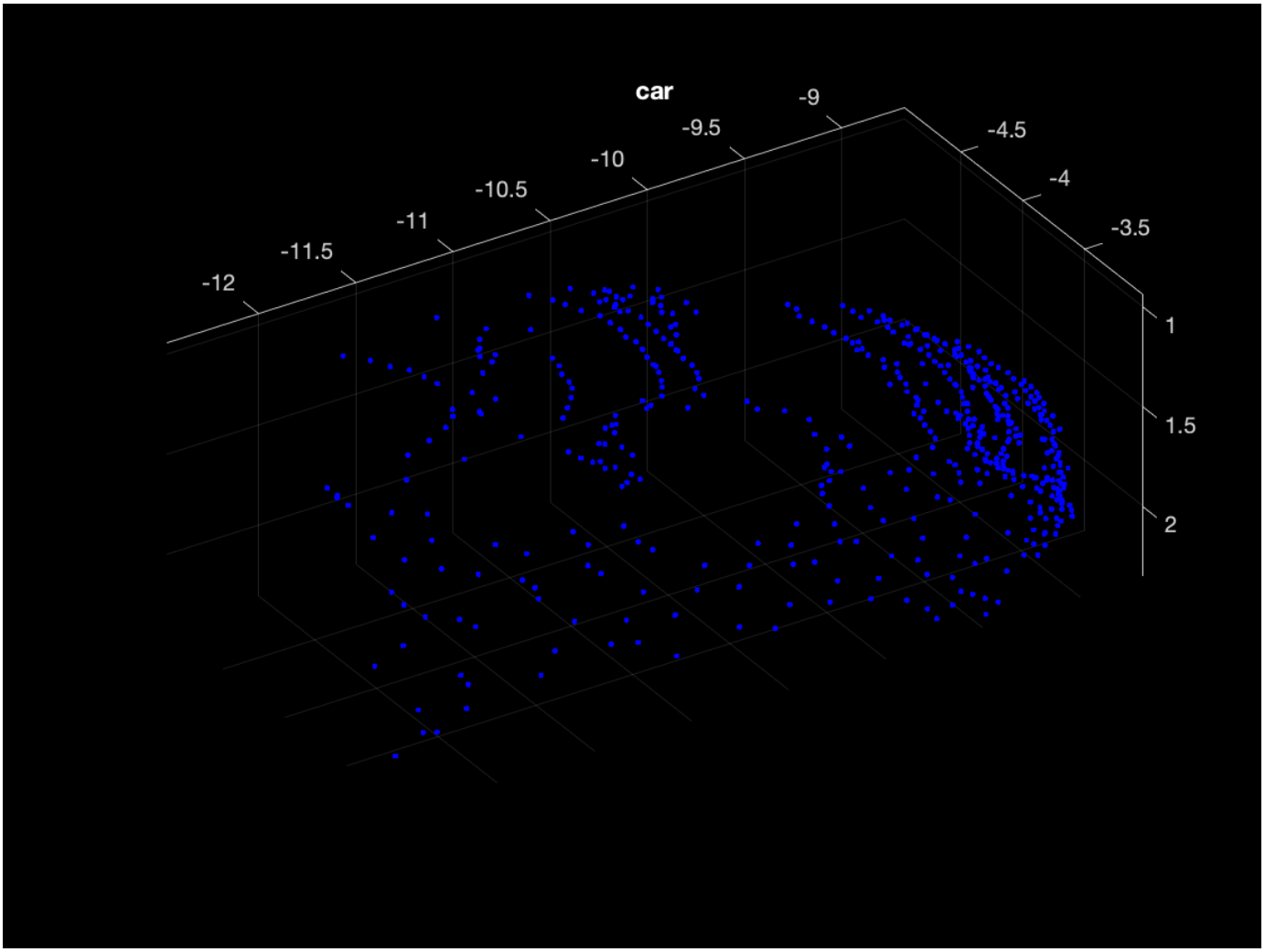
```
ptCloud = pcread("car.pcd");
X = preprocessPointCloud(ptCloud);
dlX = dlarray(X{1},"SCSB");
```

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 0 1],"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,dlmtimes(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.SharedM

% Feature transform.
[dlY,state.FeatureTransform,T] = dataTransform(dlY,parameters.FeatureTransform,state.FeatureTrans

% Shared MLP.
[dlY,state.SharedMLP2.Perceptron] = sharedMLP(dlY,parameters.SharedMLP2.Perceptron,state.SharedM

% 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 = extractdata(dlY);
dlY = squeeze(dlY); % N-by-B
T = parameters.Transform * 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.
X = extractdata(dlX); % [M 1 K B]
[C,B] = size(X,[3 4]);
X = reshape(X,[],C,1,B); % [M K 1 B]
Y = dlmtimes(X,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].

```
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(fullfile(url,name),dataLoc);
end

end
```

`batchData` Function

The `batchData` function collates input the data read from the `sydneyUrbanObjectsClassificationDatastore` into batches and moves data to the GPU for processing.

```
function [dlX,dlY] = batchData(data)
X = cat(4,data{:,1});
labels = cat(1,data{:,2});
Y = oneHotEncode(labels);

% Cast data to single for processing.
X = single(X);
Y = single(Y);

% Move data to the GPU if possible.
if canUseGPU
    X = gpuArray(X);
    Y = gpuArray(Y);
end

% Return X and Y as dlarray objects.
dlX = dlarray(X,'SCSB');
dlY = dlarray(Y,'CB');
end
```

`oneHotEncode` Function

The `oneHotEncode` function encodes categorical labels into a one-hot numeric vector.

```
function Y = oneHotEncode(labels)
numObservations = numel(labels);
numCategories = numel(categories(labels));
sz = [numCategories, numObservations];
Y = zeros(sz, 'single');
labels = labels';
idx = sub2ind(sz, int32(labels), 1:numObservations);
Y(idx) = 1;
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 15-121
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

- “Automatic Differentiation Background” on page 15-112

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 [2] on page 8-0 , [3] on page 8-0 .

```
pretrainedURL = 'https://onnxzoo.blob.core.windows.net/models/opset_8/tiny_yolov2/tiny_yolov2.ta
pretrainedNetZip = 'yolov2Tmp.tar.gz';
if ~exist(pretrainedNetZip,'file')
    disp('Downloading pretrained network (58 MB)...');
    websave(pretrainedNetZip,pretrainedURL);
end
```

Extract YOLO v2 Network

Unzip and untar the downloaded file to extract the Tiny YOLO v2 network. Load the `'model.onnx'` model, 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 .

```
pretrainedNetTar = gunzip(pretrainedNetZip);
onnxfiles = untar(pretrainedNetTar{1});
pretrainedNet = onnxfiles{1,2};
```

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.

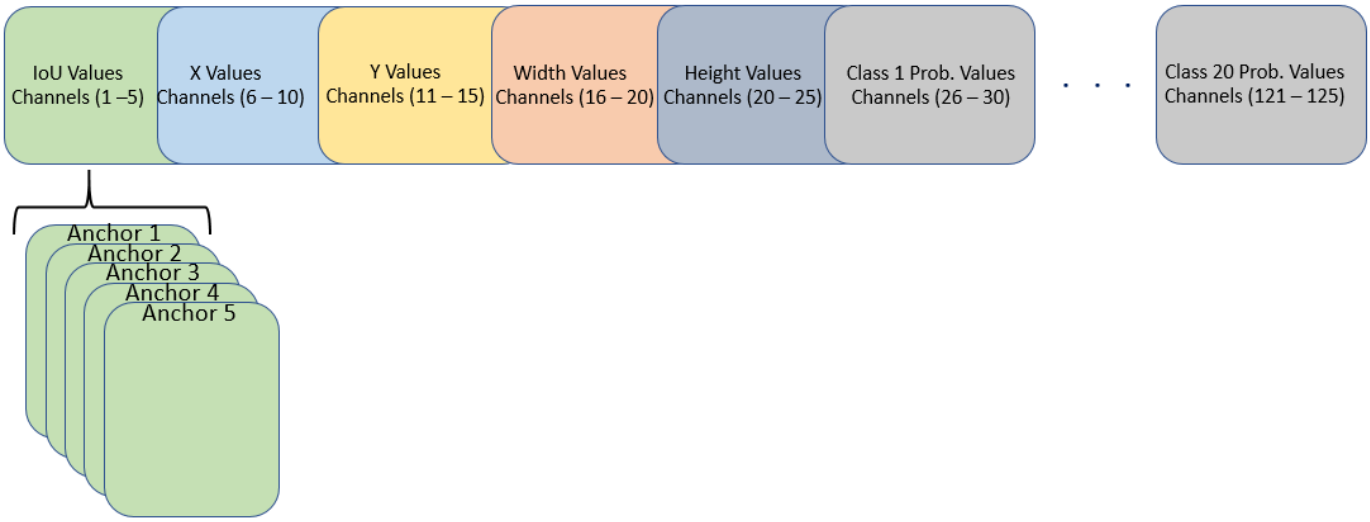
```
onnxAnchors = [1.08,1.19; 3.42,4.41; 6.63,11.38; 9.42,5.11; 16.62,10.52];

inputSize = lgraph.Layers(1,1).InputSize(1:2);
```

```
lastActivationSize = [13,13];
upScaleFactor = inputSize./lastActivationSize;
anchorBoxes = round(upScaleFactor.* onnxAnchors);
```

Reorder Detection Layer Weights

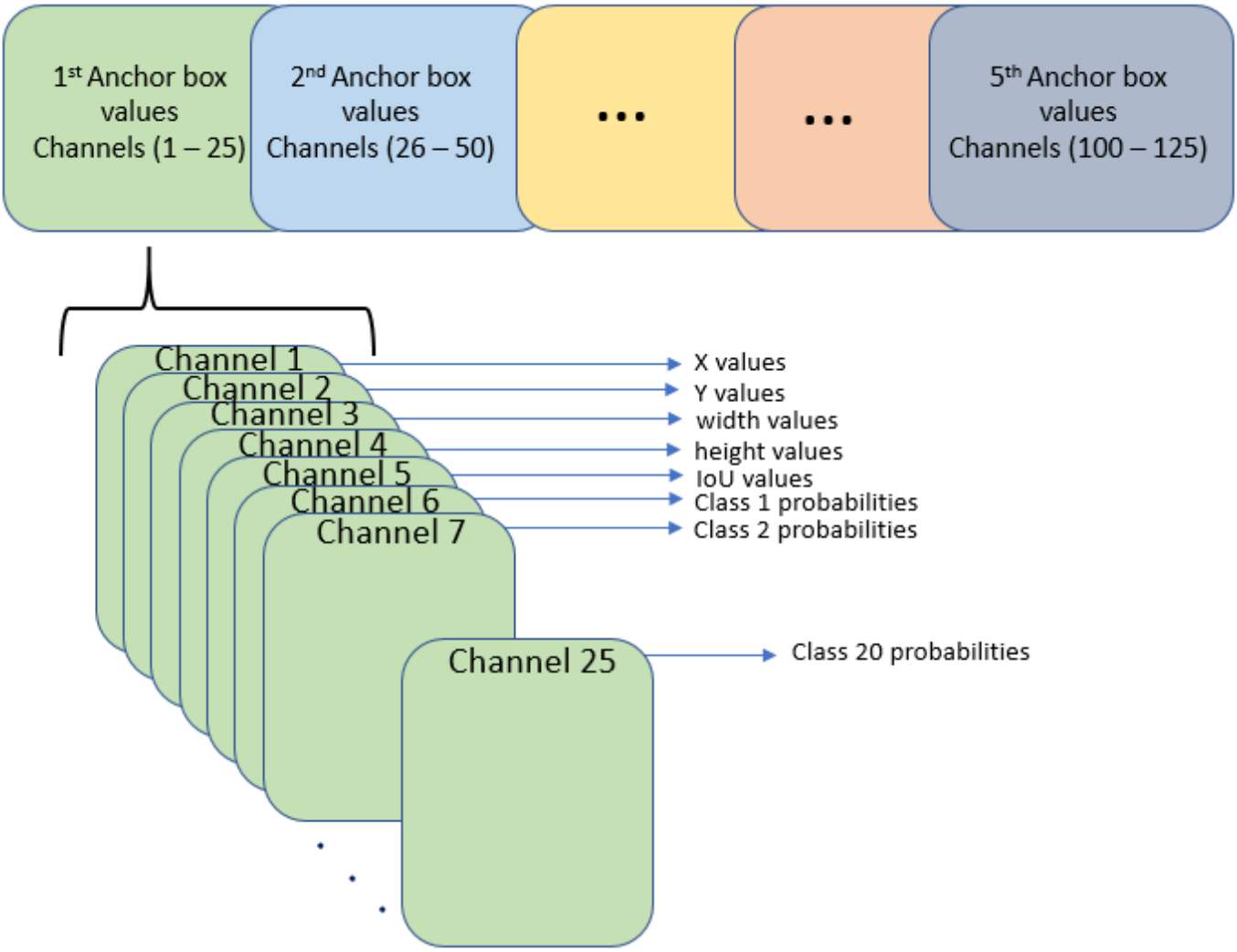
For efficient processing, you must reorder the weights and biases of the last convolution layer in the imported network to obtain the activations in the arrangement that `yolov2ObjectDetector` requires. `yolov2ObjectDetector` expects the 125 channels of the feature map of the last convolution layer in 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)
yolov2Detector = yolov2ObjectDetector(net)
```

Detect Objects Using Imported YOLO v2 Detector

Use the imported detector to detect objects in a test image. Display the results.

```
I = imread('car1.jpg');
% Convert image to BGR format.
I = cat(3,I(:,:,3),I(:,:,2),I(:,:,1));
[bboxes, scores, labels] = detect(yolov2Detector, I);
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 orignal 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 orignal 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

`addLayers` | `assembleNetwork` | `connectLayers` | `convolution2dLayer` | `detect` | `findPlaceholderLayers` | `importONNXNetwork` | `removeLayers` | `replaceLayer` | `trainYOLOv2ObjectDetector`

Objects

`yolov2ObjectDetector`

More About

- “Export to and Import from ONNX” on page 1-16

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.

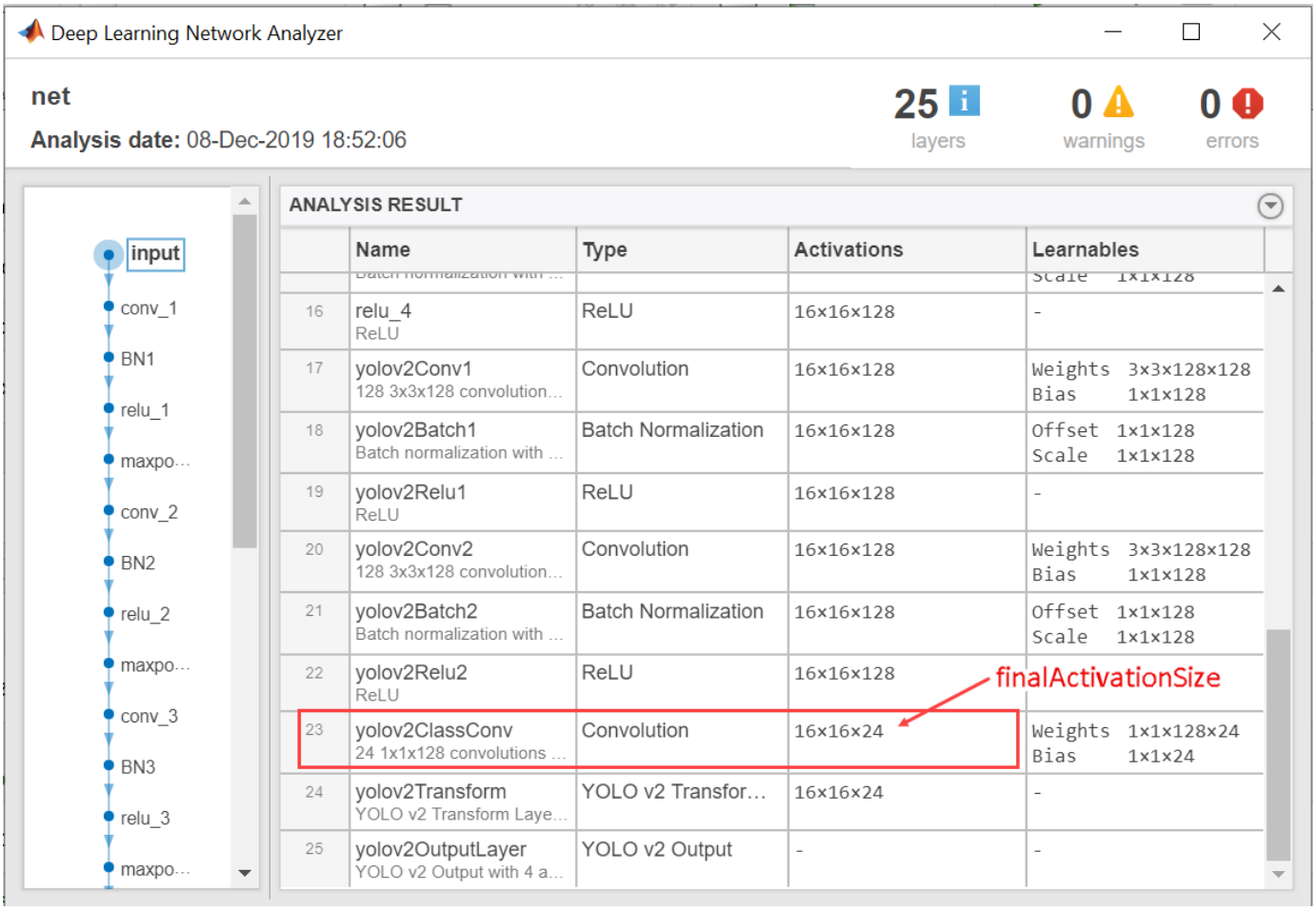
```
inputImageSize = net.Layers(1,1).InputSize;
```

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` and `yolov2OutputLayer` 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.

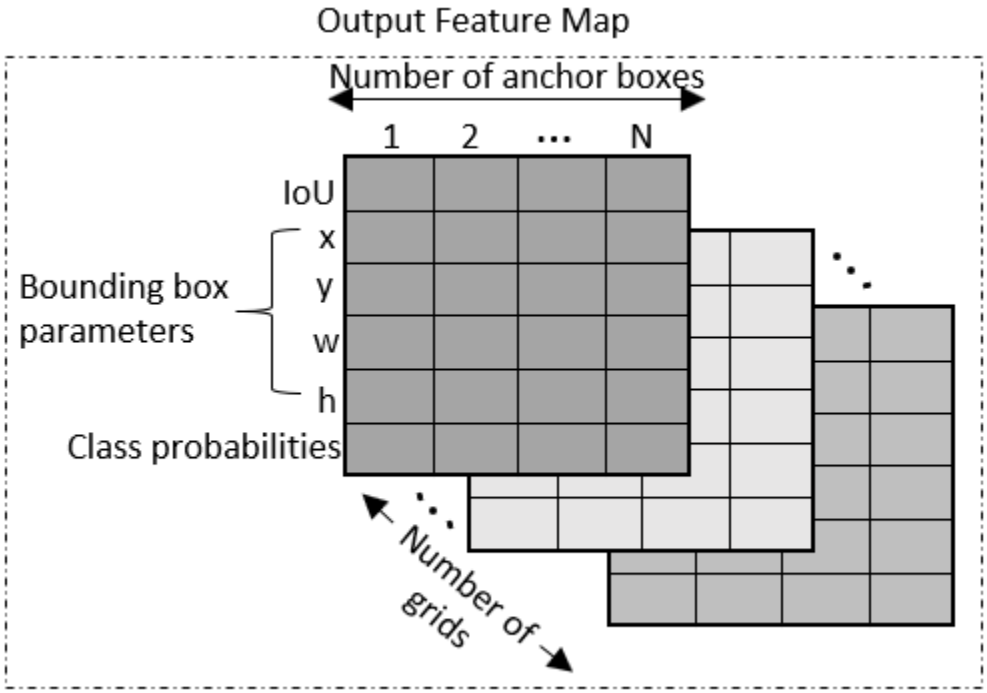
Extract Predictions from Output of ONNX Model

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,'RatioType

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) = w;
```

```
            bBoxes(4,anchorIdx,ind) = h;
        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

`analyzeNetwork` | `exportONNXNetwork`

More About

- “Export to and Import from ONNX” on page 1-16

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 using Deep Learning” (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
```

Load Dataset

This example uses a small vehicle data set that contains 295 images. 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'}    {1×4 double}
    {'vehicleImages/image_00002.jpg'}    {1×4 double}
    {'vehicleImages/image_00003.jpg'}    {1×4 double}
    {'vehicleImages/image_00004.jpg'}    {1×4 double}
```

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 1-12 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.

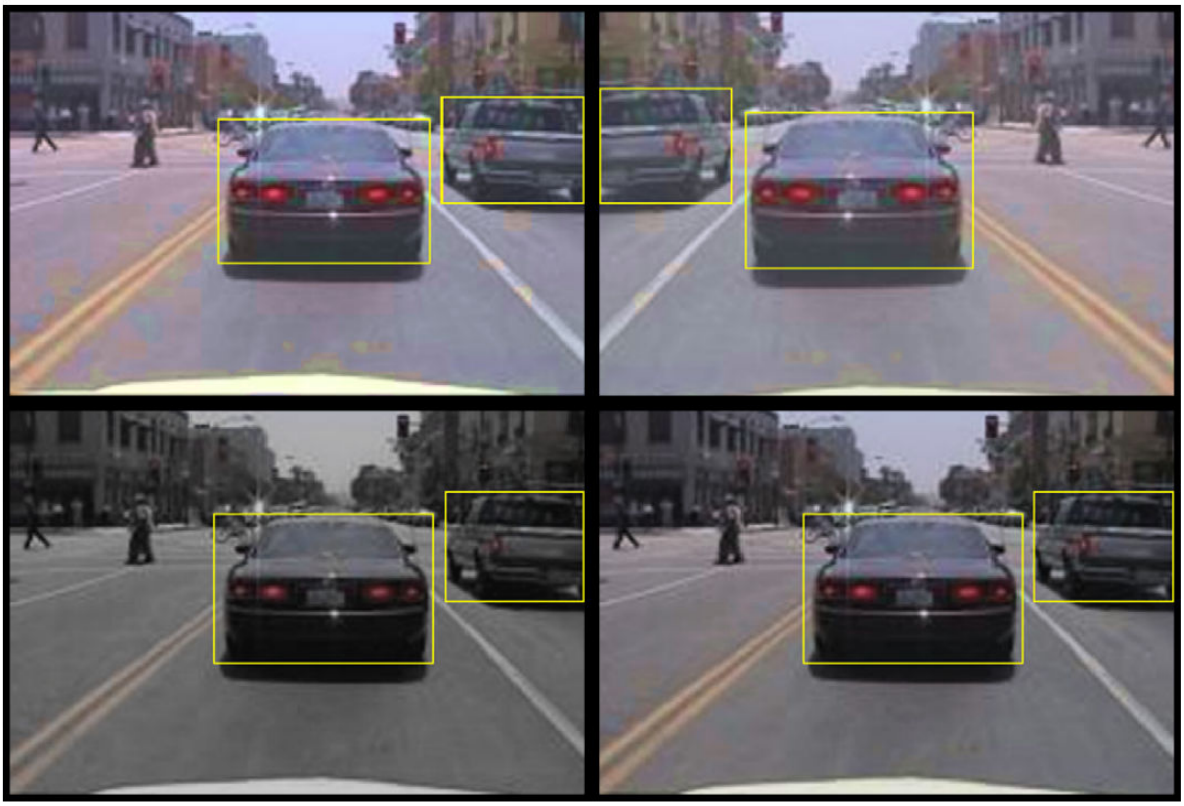
Note that data augmentation is not applied to the test data. Ideally, test data should be representative of the original data and is left unmodified for unbiased evaluation.

```
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.

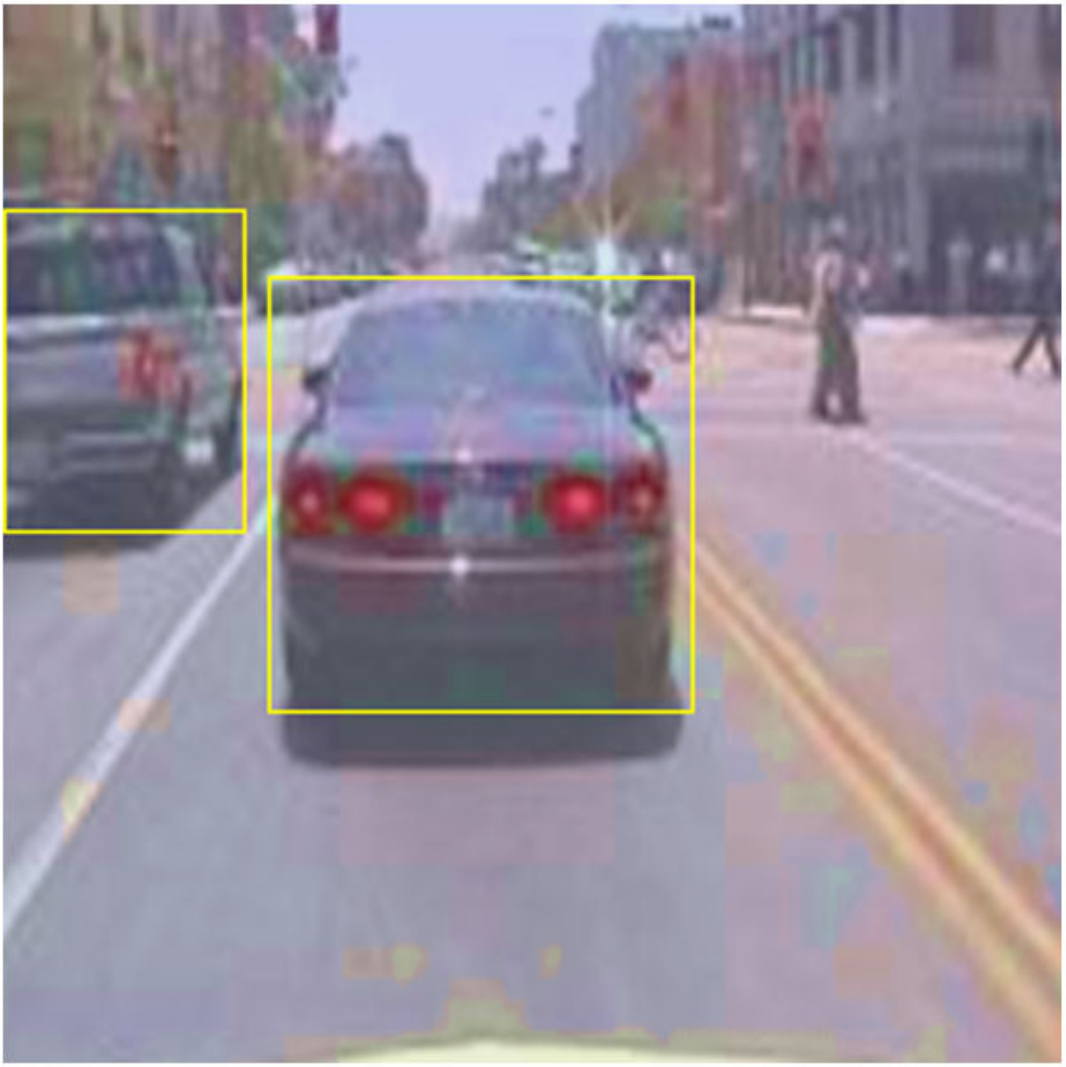
```
preprocessedTrainingData = transform(augmentedTrainingData,@(data)preprocessData(data,inputSize))
```

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 `trainYOLOv2ObjectDetector` 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);
```

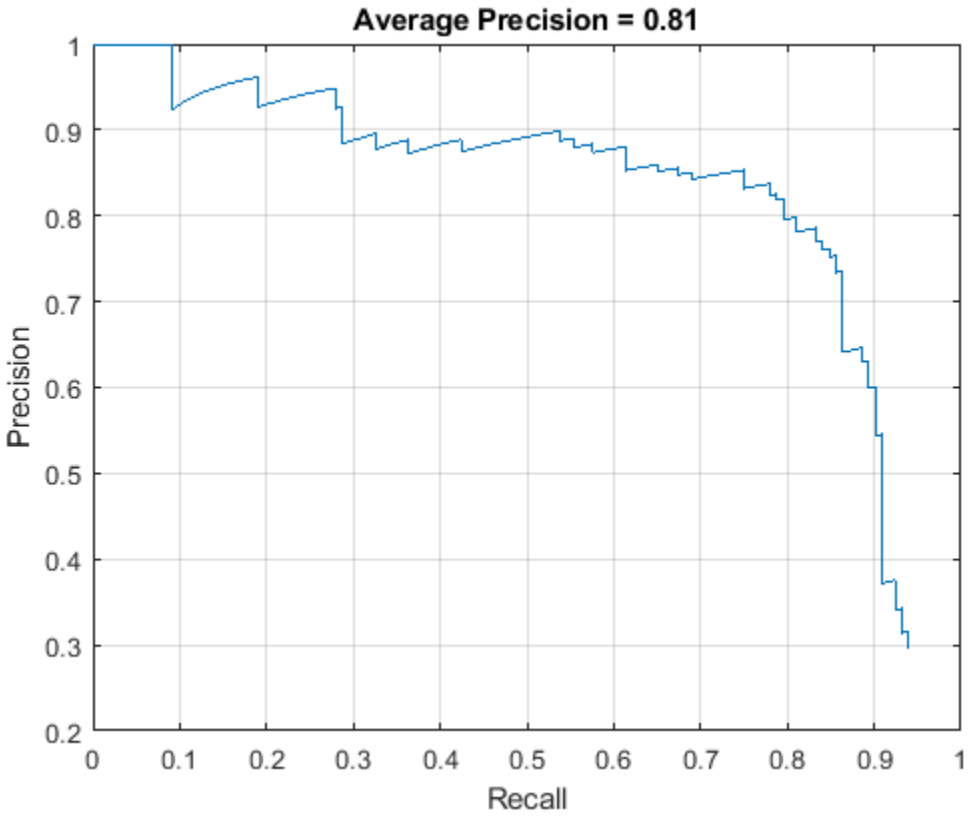
Evaluate the object detector using average precision metric.

```
[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);

% 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.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
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
`analyzeNetwork` | `combine` | `detect` | `evaluateDetectionPrecision` | `read` | `ssdLayers` | `trainSSDObjectDetector` | `trainingOptions` | `transform`

Objects
`boxLabelDatastore` | `imageDatastore` | `ssdObjectDetector`

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 Deep Learning” (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. 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.

Download Pretrained Network

Download a 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
    if ~exist('yolov3SqueezeNetVehicleExample_20a.mat','file')
        disp('Downloading pretrained detector (8.9 MB)...');
        pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehi
        websave('yolov3SqueezeNetVehicleExample_20a.mat', pretrainedURL);
    end
    pretrained = load("yolov3SqueezeNetVehicleExample_20a.mat");
    net = pretrained.net;
end
```

```
Downloading pretrained detector (8.9 MB)...
```

Load Data

This example uses a small labeled data set that contains 295 images. 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);
```

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));
```

Specify the mini-batch size. Set `ReadSize` of the training image datastore and box label datastore equal to the mini-batch size.

```
miniBatchSize = 8;
imdsTrain.ReadSize = miniBatchSize;
bldsTrain.ReadSize = miniBatchSize;
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain, bldsTrain);
testData = combine(imdsTest,bldsTest);
```

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.

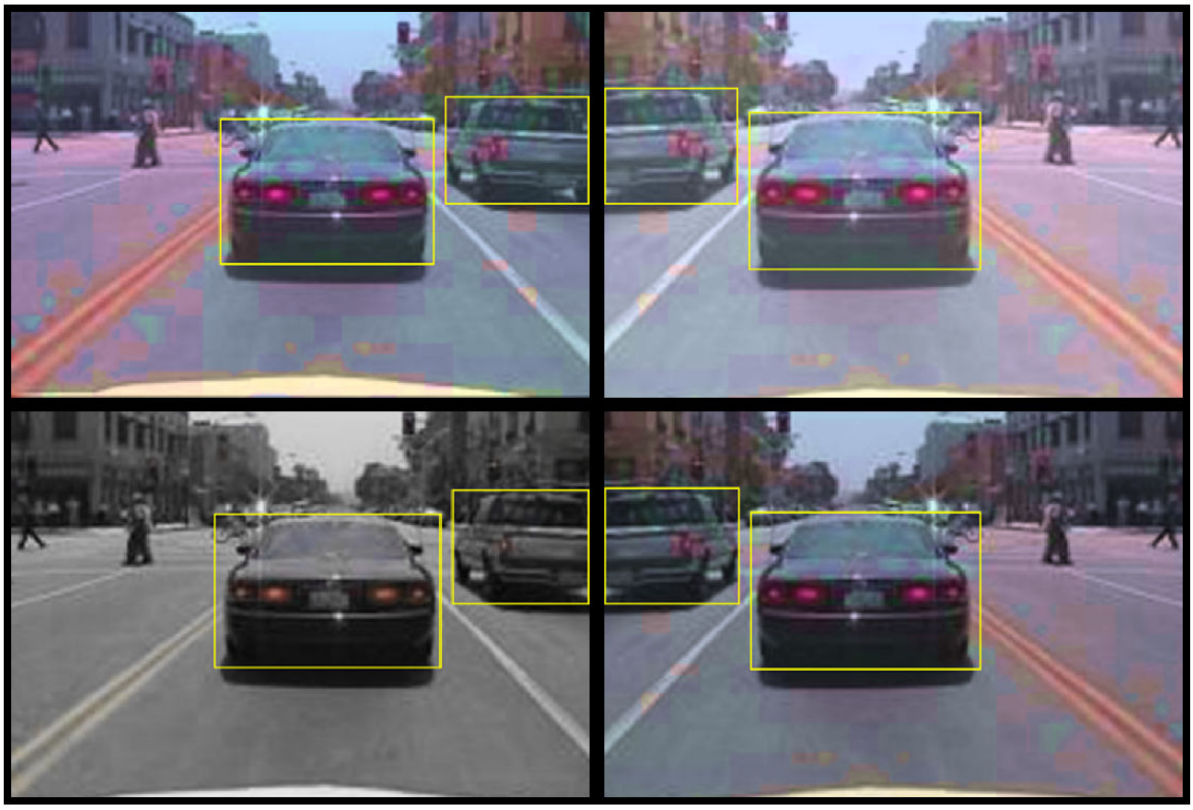
Use `transform` function to apply custom data augmentations to the training data. The `augmentData` helper function, listed at the end of the example, applies the following augmentations to the input data.

- 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)
```



Preprocess Training Data

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];
```

Preprocess the augmented training data to prepare for training. The `preprocessData` helper function, listed at the end of the example, applies the following preprocessing operations to the input data.

- Resize the images to the network input size, as the images are bigger than 227-by-227.
- Scale the image pixels in the range [0 1].

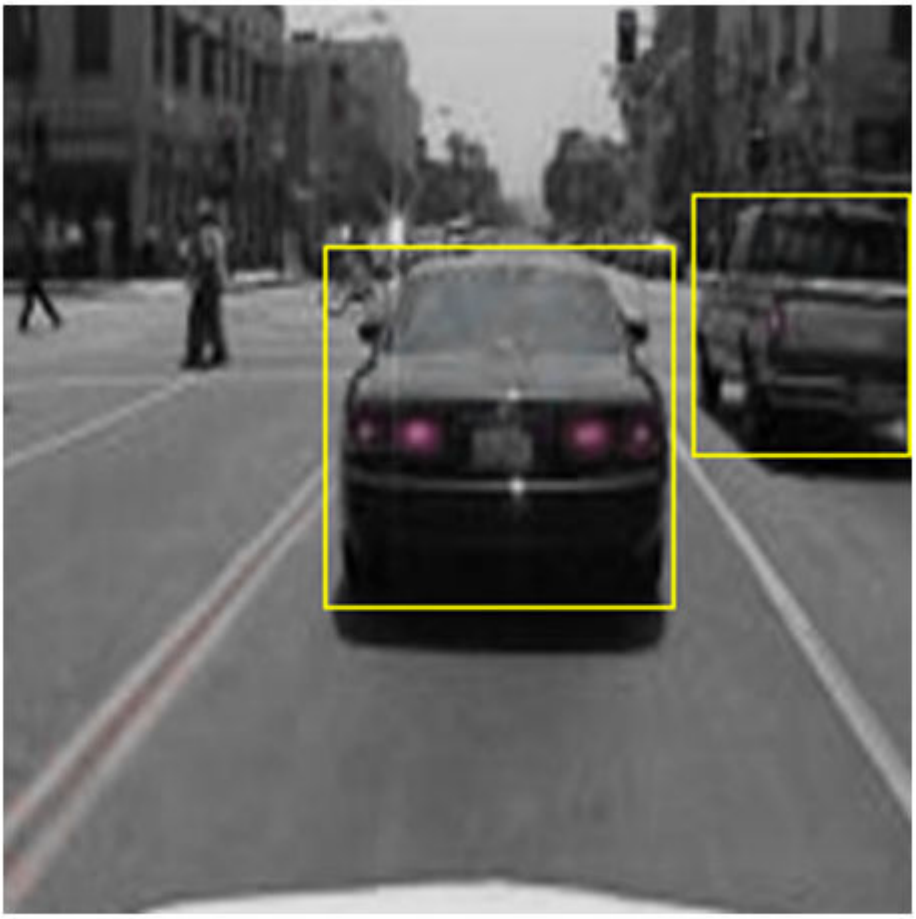
```
preprocessedTrainingData = transform(augmentedTrainingData, @(data)preprocessData(data, networkI
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image with the bounding boxes.

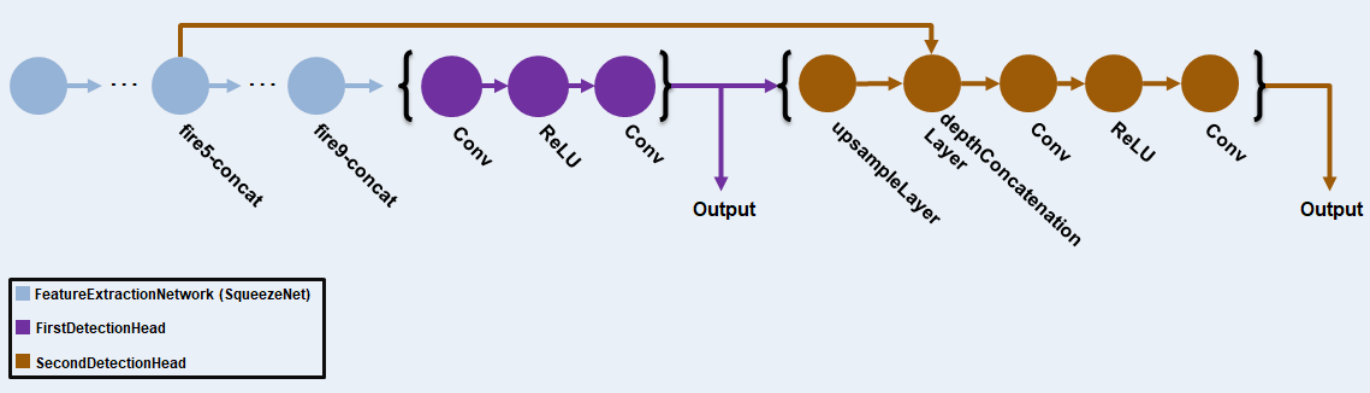
```
I = data{1,1};
bbox = data{1,2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Define 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 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 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.



You can use Deep Network Designer to create the network shown in the diagram.

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).

```
trainingDataForEstimation = transform(trainingData,@(data)preprocessData(data,networkInputSize))
numAnchors = 6;
[anchorBoxes, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchorBoxes = 6×2

    35    25
   165   138
    73    70
   151   125
   113   103
    42    38


meanIoU = 0.8369
```

Specify `anchorBoxMasks` to select anchor boxes to use in both the detection heads. `anchorBoxMasks` is a cell array of [Mx1], where M denotes the number of detection heads. Each detection head consists of a [1xN] 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 = anchorBoxes(:, 1).*anchorBoxes(:, 2);
[~, idx] = sort(area, 'descend');
anchorBoxes = anchorBoxes(idx, :);
anchorBoxMasks = {[1,2,3]
    [4,5,6]
    };
```

Load the SqueezeNet network pretrained on Imagenet data set. You can also choose to load a different pretrained network such as MobileNet-v2 or ResNet-18. YOLO v3 performs better and trains faster when you use a pretrained network.

Next, create the feature extraction network. Choosing the optimal feature extraction layer requires trial and error, and you can use `analyzeNetwork` to find the names of potential feature extraction layers within a network. For this example, use the `squeezenetFeatureExtractor` helper function, listed at the end of this example, to remove the layers after the feature extraction layer `'fire9-concat'`. The layers after this layer are specific to classification tasks and do not help with object detection.

```
baseNetwork = squeezenet;
lgraph = squeezenetFeatureExtractor(baseNetwork, networkInputSize);
```

Then specify the class names, the number of object classes to detect, and number of prediction elements per anchor box, and add the detection heads to the feature extraction network. 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. Use the supporting functions `addFirstDetectionHead` and `addSecondDetectionHead` to add the detection heads to the feature extraction network.

```
classNames = trainingDataTbl.Properties.VariableNames(2:end);
numClasses = size(classNames, 2);
numPredictorsPerAnchor = 5 + numClasses;
lgraph = addFirstDetectionHead(lgraph, anchorBoxMasks{1}, numPredictorsPerAnchor);
lgraph = addSecondDetectionHead(lgraph, anchorBoxMasks{2}, numPredictorsPerAnchor);
```

Finally, connect the detection heads by connecting the first detection head to the feature extraction layer and the second detection head to the output of the first detection head. In addition, merge the upsampled features in the second detection head with features from the `'fire5-concat'` layer to get more meaningful semantic information in the second detection head.

```
lgraph = connectLayers(lgraph, 'fire9-concat', 'conv1Detection1');
lgraph = connectLayers(lgraph,'relu1Detection1','upsample1Detection2');
lgraph = connectLayers(lgraph,'fire5-concat','depthConcat1Detection2/in2');
```

The detection heads comprise the output layer of the network. To extract output features, specify the names of detection heads using an array of form [Mx1]. M is the number of detection heads. Specify the names of detection heads in the order in which it occurs in the network.

```
networkOutputs = ["conv2Detection1"
    "conv2Detection2"
    ];
```

Specify Training Options

Specify these training options.

- Set the number of iterations to 2000.
- Set the learning rate to `0.001`.
- Set the warmup period to 1000. This parameter denotes the number of iterations to increase the learning rate exponentially based on the formula learningRate $\times \left(\frac{\text{iteration}}{\text{warmupPeriod}}\right)$. 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.

```
numIterations = 2000;
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 with compute capability 3.0 or higher. 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";
```

To train the network with a custom training loop and enable automatic differentiation, convert the layer graph to a `dlnetwork` object. Then create the training progress plotter using supporting function `configureTrainingProgressPlotter`.

Finally, specify the custom training loop. For each iteration:

- Read from `preprocessedTrainingData` and create a batch of images and ground truth boxes using the `createBatchData` supporting function.
- Convert the batch of images to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the data to `gpuArray` objects.
- 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 number of iterations using the `piecewiseLearningRateWithWarmup` supporting function.
- Update the network parameters using the `sgdmupdate` function.
- Update the `state` parameters of `net` with the moving average.
- Update the training progress plot.

```
if doTraining
    % Convert layer graph to dlnetwork.
    net = dlnetwork(lgraph);

    % Create subplots for the learning rate and mini-batch loss.
    fig = figure;
    [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(fig);
```

```
    % Custom training loop.
    for iteration = 1:numIterations

        % Reset datastore.
        if ~hasdata(preprocessedTrainingData)
            reset(preprocessedTrainingData);
        end

        % Read batch of data and create batch of images and
        % ground truths.
        data = read(preprocessedTrainingData);
        [XTrain,YTrain] = createBatchData(data, classNames);

        % Convert mini-batch of data to dlarray.
        XTrain = dlarray(single(XTrain),'SSCB');

        % If training on a GPU, then convert data to gpuArray.
        if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
            XTrain = gpuArray(XTrain);
        end

        % Evaluate the model gradients and loss using dlfeval and the
        % modelGradients function.
        [gradients,loss,state] = dlfeval(@modelGradients, net, XTrain, YTrain, anchorBoxes, ancho

        % Apply L2 regularization.
        gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients, net.Learnables);

        % Determine the current learning rate value.
        currentLR = piecewiseLearningRateWithWarmup(iteration, learningRate, warmupPeriod, numIte

        % Update the network learnable parameters using the SGDM optimizer.
        [net, velocity] = sgdmupdate(net, gradients, velocity, currentLR);

        % Update the state parameters of dlnetwork.
        net.State = state;

        % Update training plot with new points.
        addpoints(lossPlotter, iteration, double(gather(extractdata(loss))));
        addpoints(learningRatePlotter, iteration, currentLR);
        drawnow
    end
end
```

Evaluate Model

Computer Vision System 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).

Following these steps to evaluate the trained `dlnetwork` object `net` on test data.

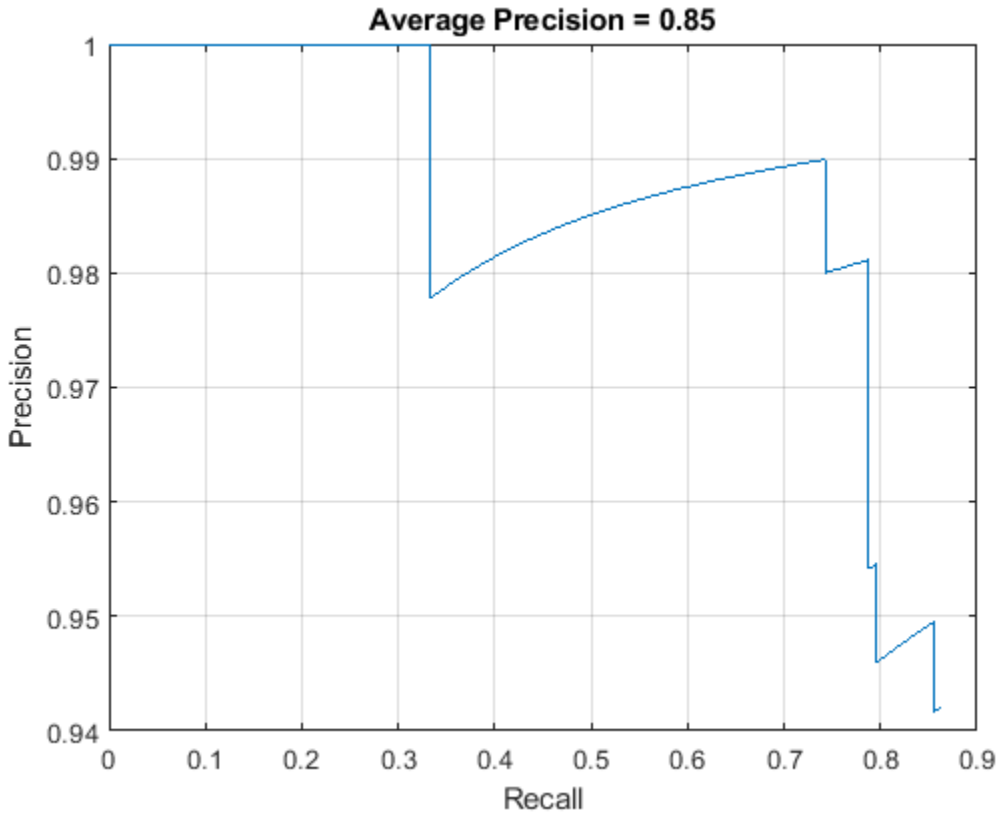
- Specify the confidence threshold as 0.5 to keep only detections with confidence scores above this value.
- Specify the overlap threshold as 0.5 to remove overlapping detections.

- 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 must be representative of the original data and be left unmodified for unbiased evaluation.
- Collect the detection results by running the detector on `preprocessedTestData`. Use the supporting function `yolov3Detect` to get the bounding boxes, object confidence scores, and class labels.
- Call `evaluateDetectionPrecision` with predicted `results` and `preprocessedTestData` as arguments.

```
confidenceThreshold = 0.5;
overlapThreshold = 0.5;

% Create the test datastore.
preprocessedTestData = transform(testData,@(data)preprocessData(data,networkInputSize));

% Create a table to hold the bounding boxes, scores, and labels returned by
% the detector.
numImages = size(testDataTbl,1);
results = table('Size',[numImages 3],...
    'VariableTypes',{'cell','cell','cell'},...
    'VariableNames',{'Boxes','Scores','Labels'});

% Run detector on each image in the test set and collect results.
for i = 1:numImages

    % Read the datastore and get the image.
    data = read(preprocessedTestData);
    I = data{1};

    % Convert to dlarray. If GPU is available, then convert data to gpuArray.
    XTest = dlarray(I,'SSCB');
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        XTest = gpuArray(XTest);
    end

    % Run the detector.
    [bboxes, scores, labels] = yolov3Detect(net, XTest, networkOutputs, anchorBoxes, anchorBoxMas

    % Collect the results.
    results.Boxes{i} = bboxes;
    results.Scores{i} = scores;
    results.Labels{i} = labels;
end

% Evaluate the object detector using Average Precision metric.
[ap, recall, precision] = evaluateDetectionPrecision(results, preprocessedTestData);
```

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 network for object detection.

- Read an image.
- Convert the image to a `dlarray` and use a GPU if one is available..
- Use the supporting function `yolov3Detect` to get the predicted bounding boxes, confidence scores, and class labels.
- Display the image with bounding boxes and confidence scores.

```
% Read the datastore.
reset(preprocessedTestData)
data = read(preprocessedTestData);

% Get the image.
I = data{1};

% Convert to dlarray.
XTest = dlarray(I,'SSCB');

% If GPU is available, then convert data to gpuArray.
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    XTest = gpuArray(XTest);
end

[bboxes, scores, labels] = yolov3Detect(net, XTest, networkOutputs, anchorBoxes, anchorBoxMasks,
```

```
% Display the detections on image.
if ~isempty(scores)
    I = insertObjectAnnotation(I, 'rectangle', bboxes, scores);
end
figure
imshow(I)
```

Supporting Functions

Model Gradients Function

The function `modelGradients` takes as input the `dlnetwork` object `net`, a mini-batch of input data `XTrain` with corresponding ground truth boxes `YTrain`, anchor boxes, anchor box mask, the specified penalty threshold, and the network output names as input arguments and 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.

The model gradients function computes the total loss and gradients by performing these operations.

- Generate predictions from the input batch of images using the supporting function `yolov3Forward`.
- 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 supporting functions `generateTiledAnchors` and `applyAnchorBoxOffsets`.
- 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, totalLoss, state] = modelGradients(net, XTrain, YTrain, anchors, mask, penal
inputImageSize = size(XTrain,1:2);

% Extract the predictions from the network.
[YPredCell, state] = yolov3Forward(net,XTrain,networkOutputs,mask);

% Gather the activations in the CPU for post processing and extract dlarray data.
gatheredPredictions = cellfun(@ gather, YPredCell(:,1:6),'UniformOutput',false);
gatheredPredictions = cellfun(@ extractdata, gatheredPredictions, 'UniformOutput', false);

% Convert predictions from grid cell coordinates to box coordinates.
tiledAnchors = generateTiledAnchors(gatheredPredictions(:,2:5),anchors,mask);
gatheredPredictions(:,2:5) = applyAnchorBoxOffsets(tiledAnchors, gatheredPredictions(:,2:5), inpu

% Generate target for predictions from the ground truth data.
[boxTarget, objectnessTarget, classTarget, objectMaskTarget, boxErrorScale] = generateTargets(gat

% 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;

% Compute gradients of learnables with regard to loss.
gradients = dlgradient(totalLoss, net.Learnables);
end

function [YPredCell, state] = yolov3Forward(net, XTrain, networkOutputs, anchorBoxMask)
% Predict the output of network and extract the confidence score, x, y,
% width, height, and class.
YPredictions = cell(size(networkOutputs));
[YPredictions{:}, state] = forward(net, XTrain, 'Outputs', networkOutputs);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);

% Append predicted width and height to the end as they are required
% for computing the loss.
YPredCell(:,7:8) = YPredCell(:,4:5);

% Apply sigmoid and exponential activation.
YPredCell(:,1:6) = applyActivations(YPredCell(:,1:6));
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 = 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) == 1
        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 [x,y] = createBatchData(data, classNames)
% The createBatchData function creates a batch of images and ground truths
% from input data, which is a [Nx3] cell array returned by the transformed
% datastore for YOLO v3. It creates two 4-D arrays by concatenating all the
% images and ground truth boxes along the batch dimension. The function
% performs these operations on the bounding boxes before concatenating
% along the fourth dimension:
% * Convert the class names to numeric class IDs based on their position in
%   the class names.
% * Combine the ground truth boxes, class IDs and network input size into
%   single cell array.
% * Pad with zeros to make the number of ground truths consistent across
%   a mini-batch.

% Concatenate images along the batch dimension.
x = cat(4,data{:,1});

% Get class IDs from the class names.
groundTruthClasses = data(:,3);
classNames = repmat({categorical(classNames')},size(groundTruthClasses));
[~,classIndices] = cellfun(@(a,b)ismember(a,b),groundTruthClasses,classNames,'UniformOutput',fals

% Append the label indexes and training image size to scaled bounding boxes
% and create a single cell array of responses.
groundTruthBoxes = data(:,2);
combinedResponses = cellfun(@(bbox,classid)[bbox,classid],groundTruthBoxes,classIndices,'Uniform
len = max( cellfun(@(x)size(x,1), combinedResponses ) );
paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'), combinedResponses, 'Uniform
y = cat(4,paddedBBoxes{:,1});
end
```

Network Creation Functions

```
function lgraph = squeezenetFeatureExtractor(net, imageInputSize)
% The squeezenetFeatureExtractor function removes the layers after 'fire9-concat'
% in SqueezeNet and also removes any data normalization used by the image input layer.

% Convert to layerGraph.
lgraph = layerGraph(net);

lgraph = removeLayers(lgraph, {'drop9' 'conv10' 'relu_conv10' 'pool10' 'prob' 'ClassificationLaye
inputLayer = imageInputLayer(imageInputSize,'Normalization','none','Name','data');
lgraph = replaceLayer(lgraph,'data',inputLayer);
end
```

```
function lgraph = addFirstDetectionHead(lgraph,anchorBoxMasks,numPredictorsPerAnchor)
% The addFirstDetectionHead function adds the first detection head.

numAnchorsScale1 = size(anchorBoxMasks, 2);
% Compute the number of filters for last convolution layer.
numFilters = numAnchorsScale1*numPredictorsPerAnchor;
firstDetectionSubNetwork = [
    convolution2dLayer(3,256,'Padding','same','Name','conv1Detection1','WeightsInitializer','he'
    reluLayer('Name','relu1Detection1')
    convolution2dLayer(1,numFilters,'Padding','same','Name','conv2Detection1','WeightsInitialize
    ];
lgraph = addLayers(lgraph,firstDetectionSubNetwork);
end

function lgraph = addSecondDetectionHead(lgraph,anchorBoxMasks,numPredictorsPerAnchor)
% The addSecondDetectionHead function adds the second detection head.

numAnchorsScale2 = size(anchorBoxMasks, 2);
% Compute the number of filters for the last convolution layer.
numFilters = numAnchorsScale2*numPredictorsPerAnchor;
secondDetectionSubNetwork = [
    upsampleLayer(2,'upsample1Detection2')
    depthConcatenationLayer(2,'Name','depthConcat1Detection2');
    convolution2dLayer(3,128,'Padding','same','Name','conv1Detection2','WeightsInitializer','he'
    reluLayer('Name','relu1Detection2')
    convolution2dLayer(1,numFilters,'Padding','same','Name','conv2Detection2','WeightsInitialize
    ];
lgraph = addLayers(lgraph,secondDetectionSubNetwork);
end
```

Learning Rate Schedule Function

```
function currentLR = piecewiseLearningRateWithWarmup(iteration, learningRate, warmupPeriod, numIt
% The piecewiseLearningRateWithWarmup function computes the current
% learning rate based on the iteration number.

if iteration <= warmupPeriod
    % Increase the learning rate for number of iterations in warmup period.
    currentLR = learningRate * ((iteration/warmupPeriod)^4);

elseif iteration >= warmupPeriod && iteration < warmupPeriod+floor(0.6*(numIterations-warmupPerio
    % After warm up period, keep the learning rate constant if the remaining number of iteration
    currentLR = learningRate;

elseif iteration >= warmupPeriod+floor(0.6*(numIterations-warmupPeriod)) && iteration < warmupPer
    % If the remaining number of iteration is more than 60 percent but less
    % than 90 percent multiply the learning rate by 0.1.
    currentLR = learningRate*0.1;

else
    % If remaining iteration is more than 90 percent multiply the learning
    % rate by 0.01.
    currentLR = learningRate*0.01;
end
end
```

Predict Functions

```
function [bboxes,scores,labels] = yolov3Detect(net, XTest, networkOutputs, anchors, anchorBoxMasl
% The yolov3Detect function detects the bounding boxes, scores, and labels in an image.

imageSize = size(XTest,[1,2]);

% Find the input image layer and get the network input size.
networkInputIdx = arrayfun( @(x)isa(x,'nnet.cnn.layer.ImageInputLayer'), net.Layers);
networkInputSize = net.Layers(networkInputIdx).InputSize;

% Predict and filter the detections based on confidence threshold.
predictions = yolov3Predict(net,XTest,networkOutputs,anchorBoxMask);
predictions = cellfun(@ gather, predictions,'UniformOutput',false);
predictions = cellfun(@ extractdata, predictions, 'UniformOutput', false);
tiledAnchors = generateTiledAnchors(predictions(:,2:5),anchors,anchorBoxMask);
predictions(:,2:5) = applyAnchorBoxOffsets(tiledAnchors, predictions(:,2:5), networkInputSize);
[bboxes,scores,labels] = generateYOLOv3Detections(predictions, confidenceThreshold, imageSize, c

% Apply suppression to the detections to filter out multiple overlapping
% detections.
if ~isempty(scores)
    [bboxes, scores, labels] = selectStrongestBboxMulticlass(bboxes, scores, labels ,...
        'RatioType', 'Union', 'OverlapThreshold', overlapThreshold);
end
end

function YPredCell = yolov3Predict(net,XTrain,networkOutputs,anchorBoxMask)
% Predict the output of network and extract the confidence, x, y,
% width, height, and class.
YPredictions = cell(size(networkOutputs));
[YPredictions{:}] = predict(net, XTrain);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);

% Apply activation to the predicted cell array.
YPredCell = applyActivations(YPredCell);
end
```

Utility Functions

```
function YPredCell = applyActivations(YPredCell)
YPredCell(:,1:3) = cellfun(@ sigmoid ,YPredCell(:,1:3),'UniformOutput',false);
YPredCell(:,4:5) = cellfun(@ exp,YPredCell(:,4:5),'UniformOutput',false);
YPredCell(:,6) = cellfun(@ sigmoid ,YPredCell(:,6),'UniformOutput',false);
end

function predictions = extractPredictions(YPredictions, anchorBoxMask)
predictions = cell(size(YPredictions, 1),6);
for ii = 1:size(YPredictions, 1)
    numAnchors = size(anchorBoxMask{ii},2);
    % Confidence scores.
    startIdx = 1;
    endIdx = numAnchors;
    predictions{ii,1} = YPredictions{ii}(:,:,startIdx:endIdx,:);

    % X positions.
    startIdx = startIdx + numAnchors;
    endIdx = endIdx+numAnchors;
```

```
        predictions{ii,2} = YPredictions{ii}(:,:,startIdx:endIdx,:);

        % Y positions.
        startIdx = startIdx + numAnchors;
        endIdx = endIdx+numAnchors;
        predictions{ii,3} = YPredictions{ii}(:,:,startIdx:endIdx,:);

        % Width.
        startIdx = startIdx + numAnchors;
        endIdx = endIdx+numAnchors;
        predictions{ii,4} = YPredictions{ii}(:,:,startIdx:endIdx,:);

        % Height.
        startIdx = startIdx + numAnchors;
        endIdx = endIdx+numAnchors;
        predictions{ii,5} = YPredictions{ii}(:,:,startIdx:endIdx,:);

        % Class probabilities.
        startIdx = startIdx + numAnchors;
        predictions{ii,6} = YPredictions{ii}(:,:,startIdx:end,:);
    end
    end

    function tiledAnchors = generateTiledAnchors(YPredCell,anchorBoxes,anchorBoxMask)
    % Generate tiled anchor offset.
    tiledAnchors = cell(size(YPredCell));
    for i=1:size(YPredCell,1)
        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

    function tiledAnchors = applyAnchorBoxOffsets(tiledAnchors,YPredCell,inputImageSize)
    % Convert grid cell coordinates to box coordinates.
    for i=1:size(YPredCell,1)
        [h,w,~,~] = size(YPredCell{i,1});
        tiledAnchors{i,1} = (tiledAnchors{i,1}+YPredCell{i,1})./w;
        tiledAnchors{i,2} = (tiledAnchors{i,2}+YPredCell{i,2})./h;
        tiledAnchors{i,3} = (tiledAnchors{i,3}.*YPredCell{i,3})./inputImageSize(2);
        tiledAnchors{i,4} = (tiledAnchors{i,4}.*YPredCell{i,4})./inputImageSize(1);
    end
    end

    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
```

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
`analyzeNetwork` | `combine` | `dlfeval` | `dlupdate` | `estimateAnchorBoxes` | `evaluateDetectionPrecision` | `read` | `sgdmupdate` | `transform`

Objects
`boxLabelDatastore` | `dlarray` | `dlnetwork` | `imageDatastore`

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 “Object Detection using Deep Learning” (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. 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 1-12). 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` 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` 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

   145   122
    81    76
   160   132
    41    34
    63    62
   103    97
    33    23


meanIoU = 0.8630
```

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

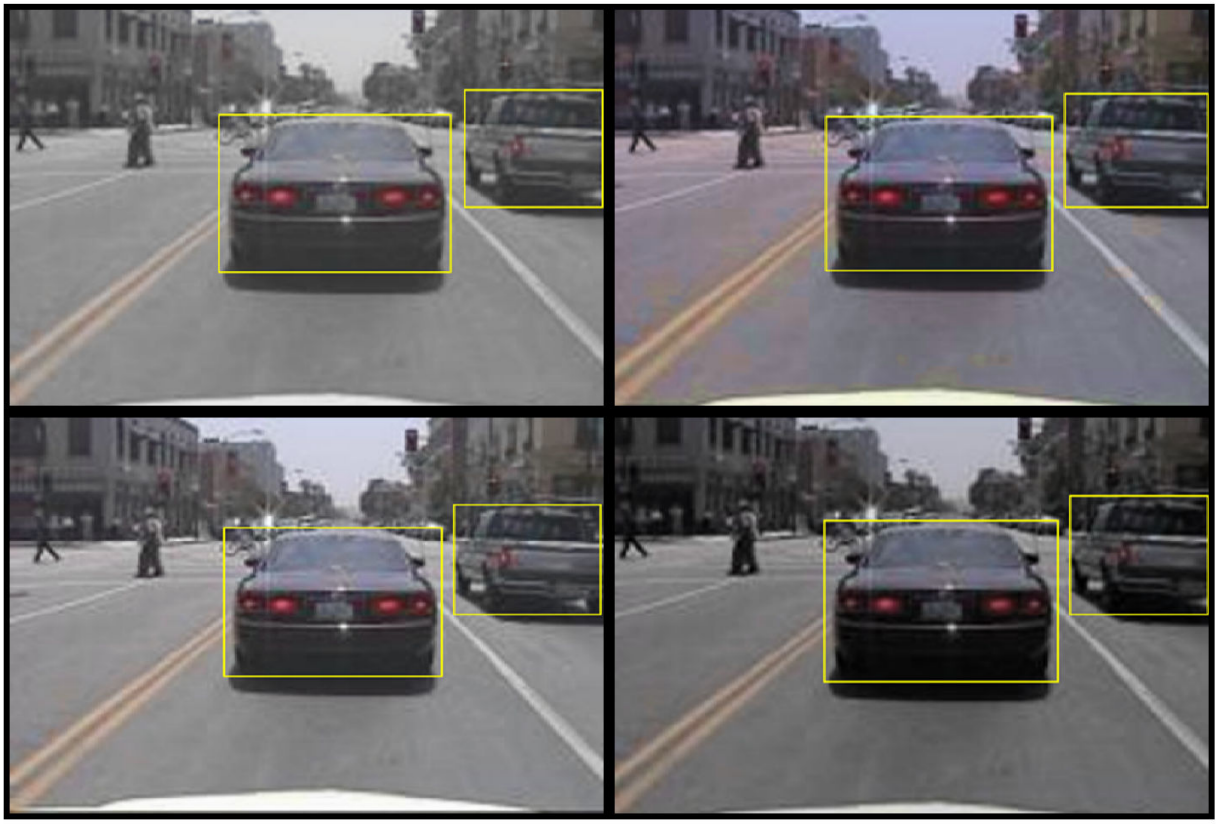
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation you can add more variety to the training data without actually having to increase the number of labeled training samples.

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` 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 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. 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.

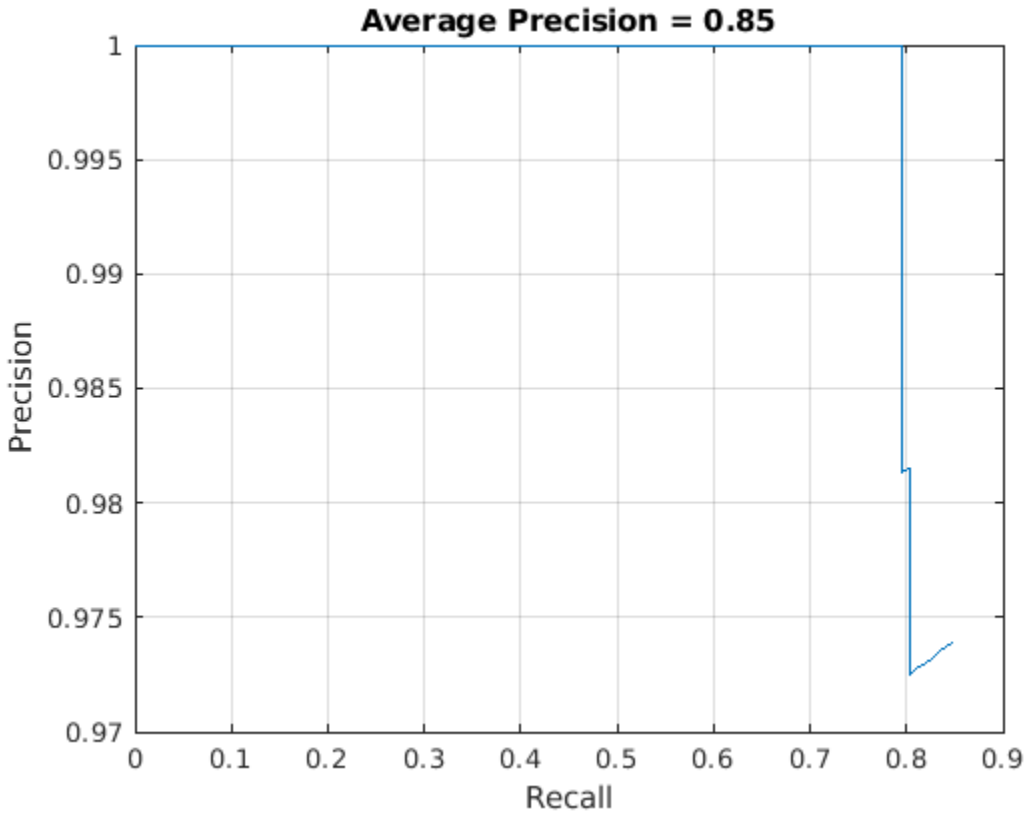
```
detectionResults = detect(detector, preprocessedTestData);
```

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);

% 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.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

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 1-12.

To get a pretrained Resnet-18, install Deep Learning Toolbox™ Model for Resnet-18 Network. 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 with compute capability 3.0 or higher is highly recommended for running this example. Use of a GPU requires 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,1);
I = histeq(I);
imshow(I)
```

Load CamVid Pixel-Labeled Images

Use `pixelLabelDatastore` 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,1);
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

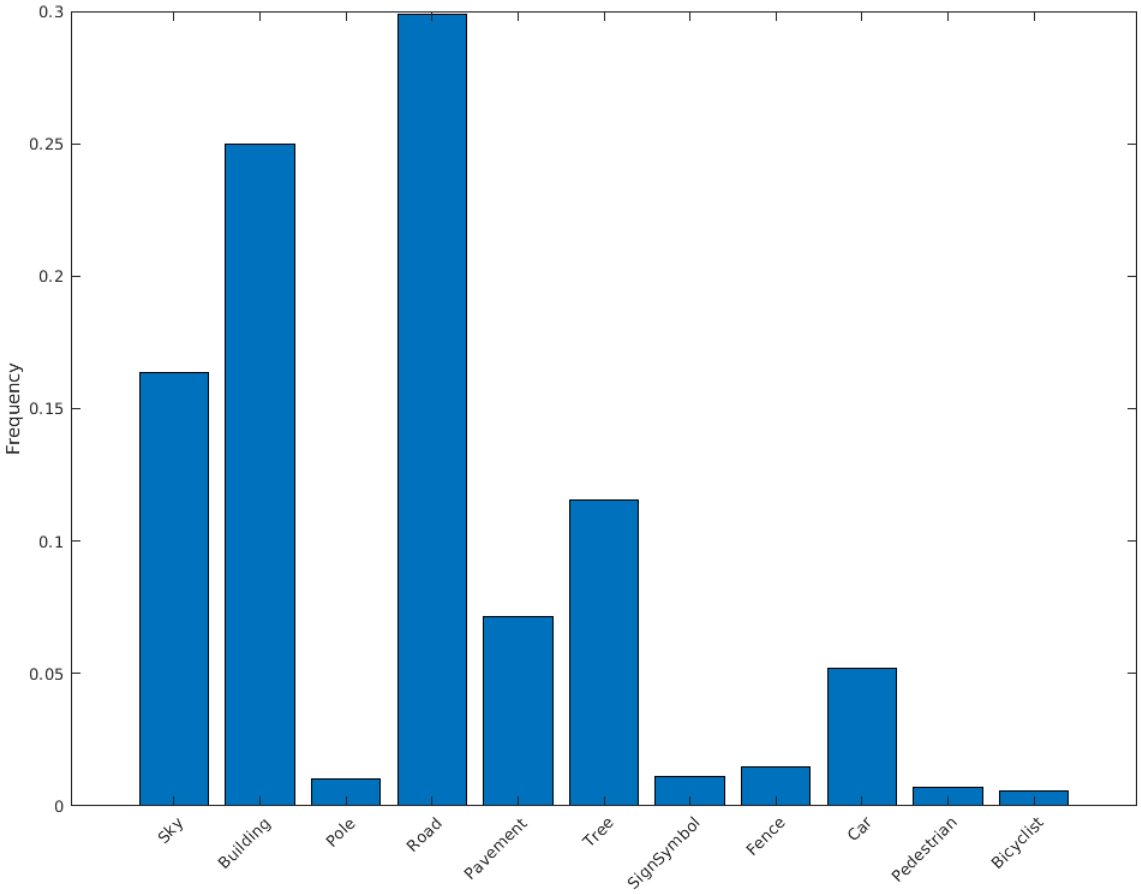
To see the distribution of class labels in the CamVid dataset, use `countEachLabel`. This function counts the number of pixels by class label.

```
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` 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`.

```
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.
pximdsVal = pixelLabelImageDatastore(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',pximdsVal,...
    '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 during training to provide more examples to the network because it helps improve the accuracy of the network. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation. Use the `imageDataAugmenter` to specify these data augmentation parameters.

```
augmenter = imageDataAugmenter('RandXReflection',true,...
    'RandXTranslation',[-10 10],'RandYTranslation',[-10 10]);
```

`imageDataAugmenter` supports several other types of data augmentation. Choosing among them requires empirical analysis and is another level of hyper-parameter tuning.

Start Training

Combine the training data and data augmentation selections using `pixelLabelImageDatastore`. The `pixelLabelImageDatastore` reads batches of training data, applies data augmentation, and sends the augmented data to the training algorithm.

```
pximds = pixelLabelImageDatastore(imdsTrain,pxdsTrain, ...
    'DataAugmentation',augmenter);
```

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 using the `'OutputSize'` parameter of `pixelLabelImageDatastore`. Training this network takes about 5 hours. Depending on your GPU hardware, it can take even longer.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(pximds,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.

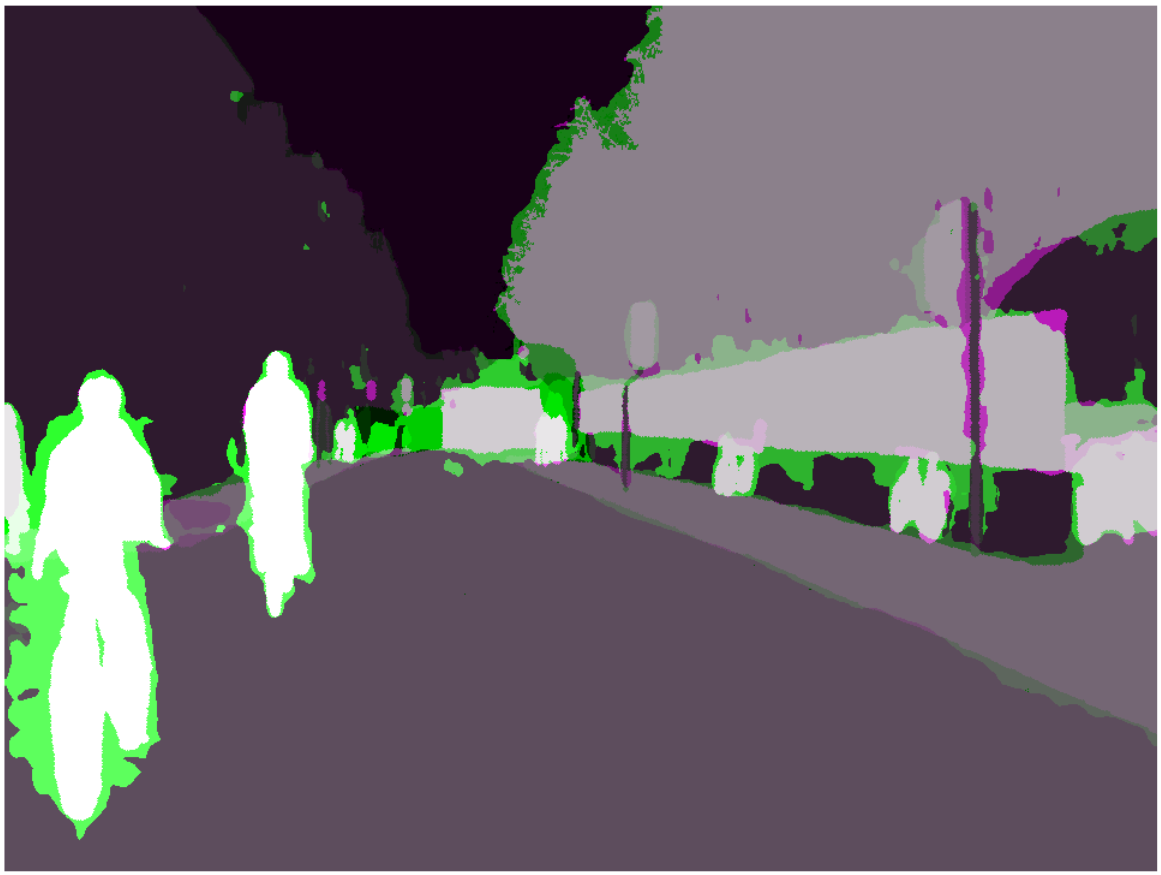
```
I = readimage(imdsTest,35);
C = semanticseg(I, net);
```

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` 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` and the `bfscore` contour matching score.

Evaluate Trained Network

To measure accuracy for multiple test images, run`semanticseg` 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` 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      0.48564
    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
```

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

`countEachLabel` | `evaluateSemanticSegmentation` | `imageDataAugmenter` | `labeloverlay` | `pixelClassificationLayer` | `pixelLabelDatastore` | `pixelLabelImageDatastore` | `segnetLayers` | `semanticseg` | `trainNetwork` | `trainingOptions`

More About

- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 8-95
- “Semantic Segmentation Using Dilated Convolutions” on page 8-90
- “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 1-12

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.

```
pximdsTrain = pixelLabelImageDatastore(imdsTrain,pxdsTrain);
tbl = countEachLabel(pximdsTrain)
```

```
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(pximdsTrain,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 |       00:00:42 |       88.56% |       0.2393 |          0.0010 |
|      34 |         100 |       00:01:25 |       92.08% |       0.1672 |          0.0010 |
|      50 |         150 |       00:02:05 |       93.17% |       0.1472 |          0.0010 |
|      67 |         200 |       00:02:48 |       94.15% |       0.1313 |          0.0010 |
|      84 |         250 |       00:03:35 |       94.47% |       0.1166 |          0.0010 |
|     100 |         300 |       00:04:16 |       95.04% |       0.1100 |          0.0010 |
|=========================================================================================|
```

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);
```

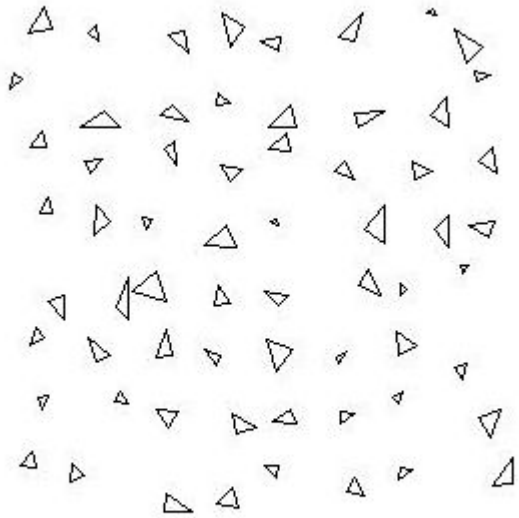
```
Evaluating semantic segmentation results
----------------------------------------
* Selected metrics: global accuracy, class accuracy, IoU, weighted IoU, BF score.
* Processed 100 images.
* Finalizing... Done.
* Data set metrics:

    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`.

Segment New Image

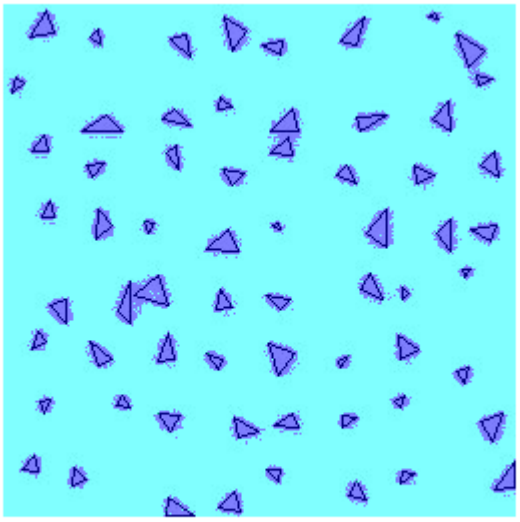
Read and display the test image `triangleTest.jpg`.

```
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

`convolution2dLayer` | `countEachLabel` | `evaluateSemanticSegmentation` | `labeloverlay` | `pixelClassificationLayer` | `pixelLabelDatastore` | `pixelLabelImageDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions`

More About

- “Semantic Segmentation Using Deep Learning” on page 8-74
- “Semantic Segmentation of Multispectral Images Using Deep Learning” on page 8-95
- “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 1-12

Semantic Segmentation of Multispectral Images Using Deep Learning

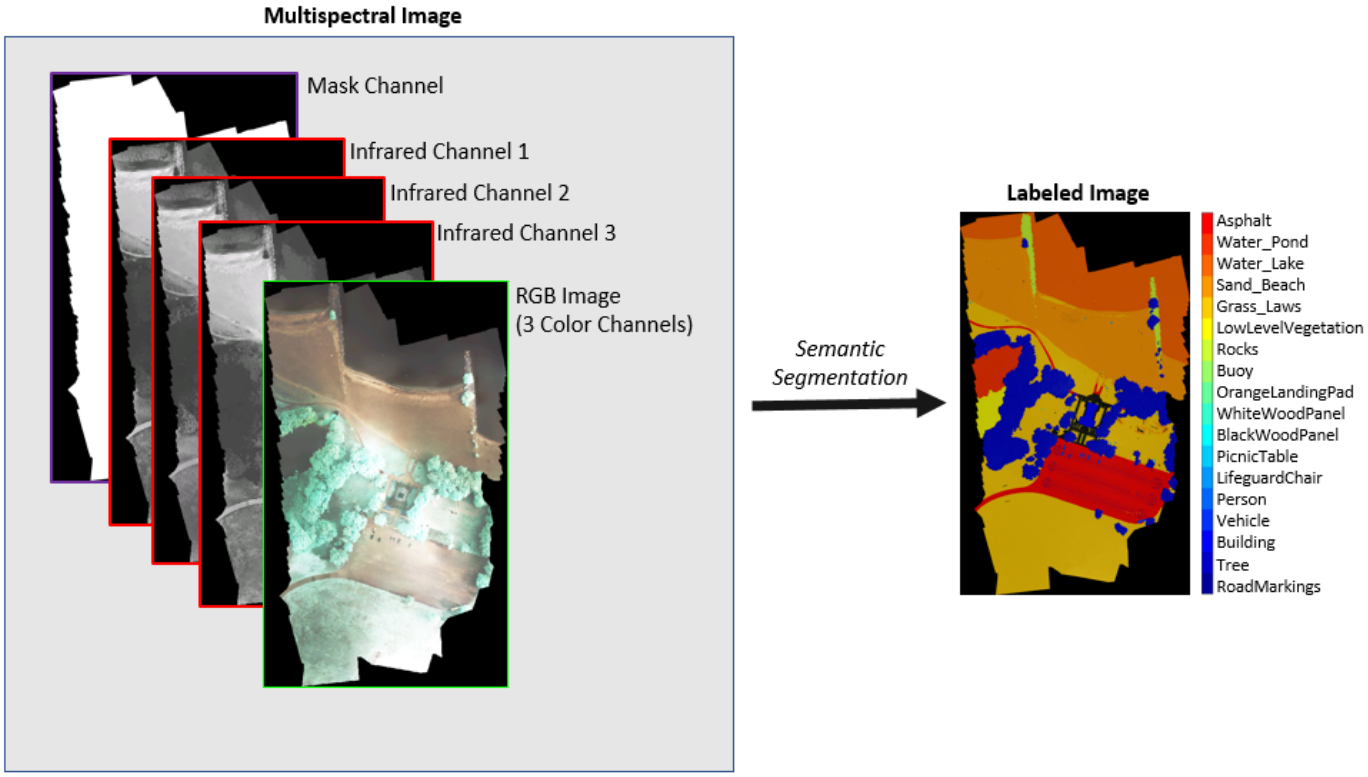
This example shows how to train a U-Net convolutional neural network to perform semantic segmentation of a multispectral image with seven channels: three color channels, three near-infrared channels, and a mask.

The example shows how to train a U-Net network and also provides a pretrained U-Net network. If you choose to train the U-Net network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended (requires Parallel Computing Toolbox™).

Introduction

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 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);
```

In addition, download 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 model enables you to run the entire example without having to wait for training to complete.

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_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);
```

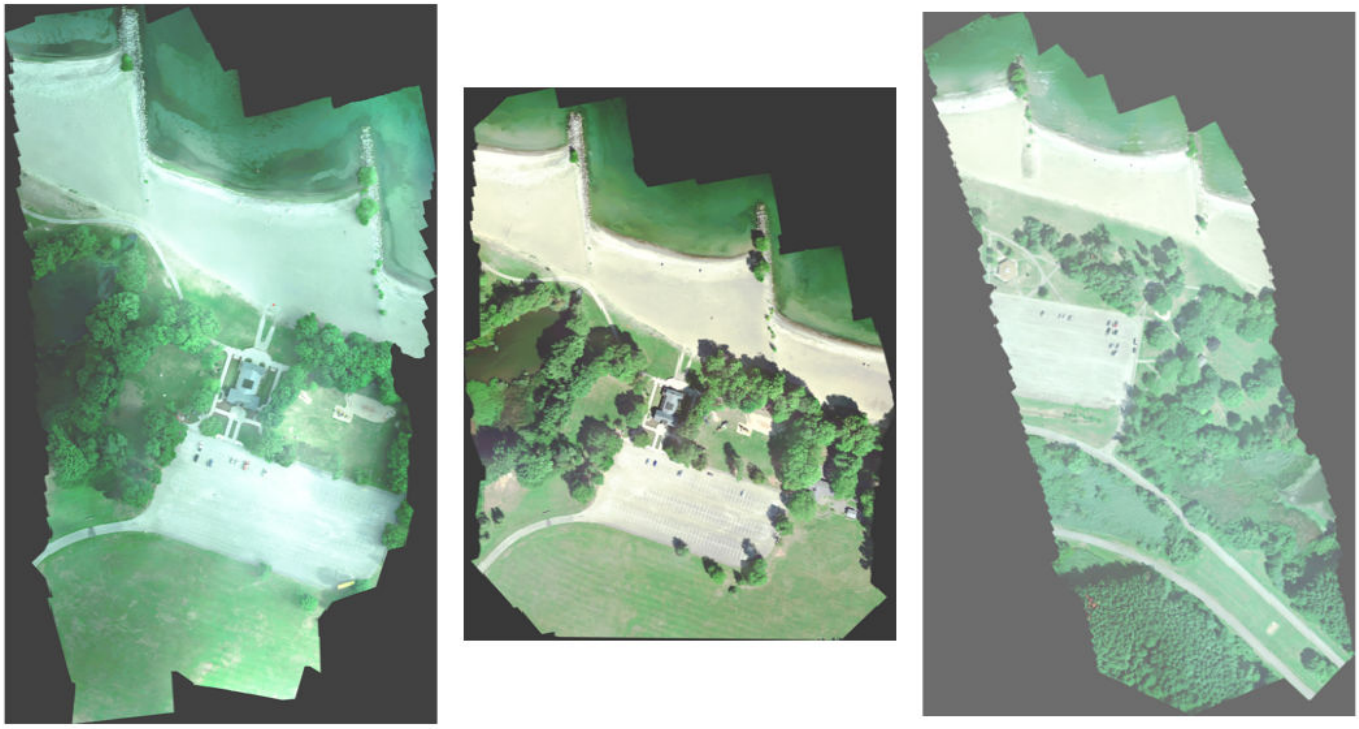
Confirm that the data has the correct structure.

```
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
```

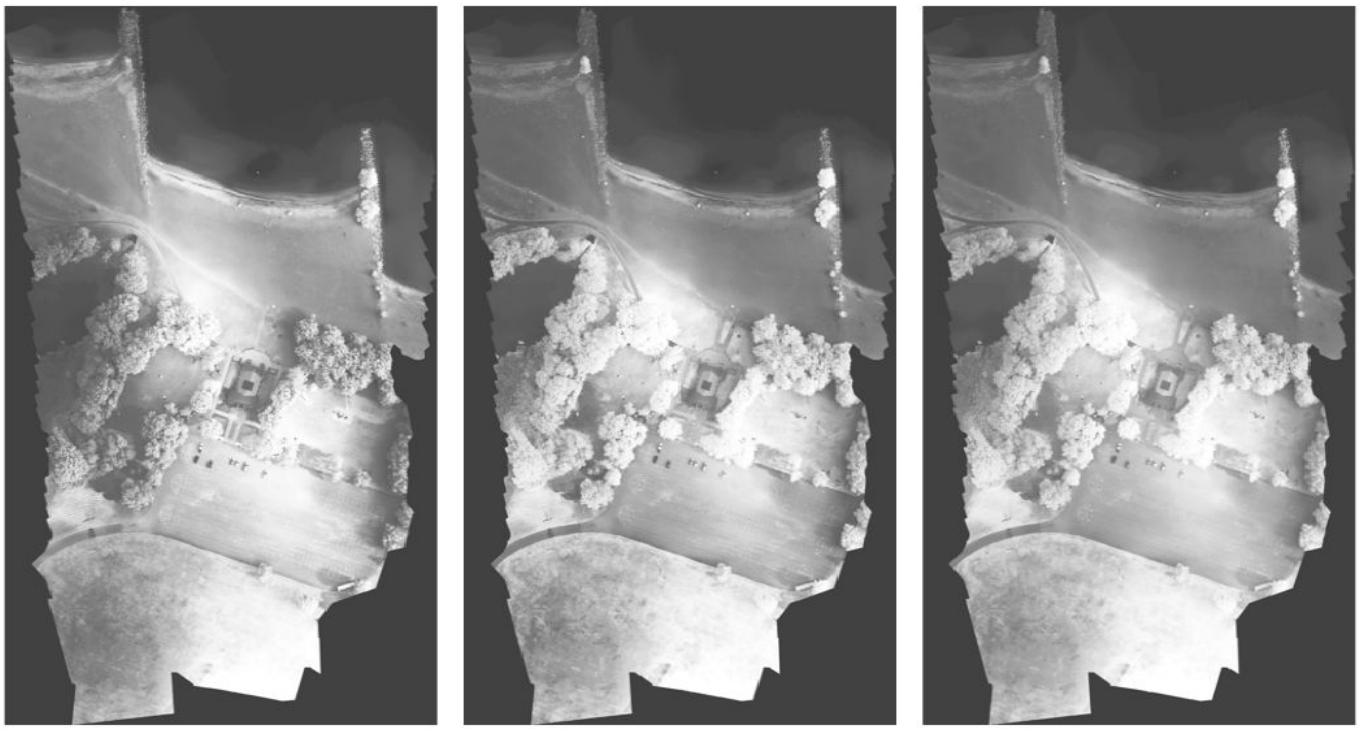
The RGB color channels are the 3rd, 2nd and 1st image channels. Display the color component of the training, validation, and test images as a montage. To make the images appear brighter on the screen, equalize their histograms by using the `histeq` function.

```
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)
```

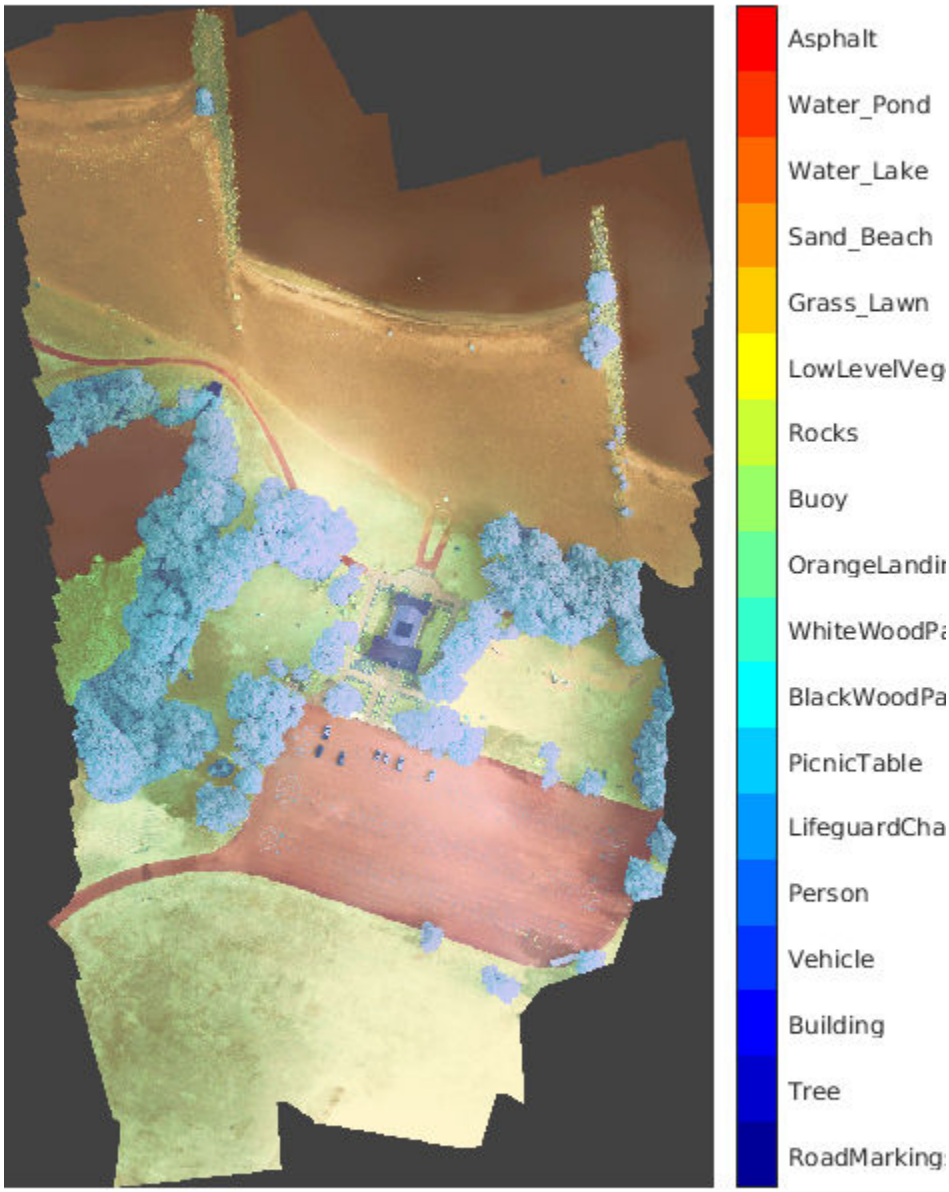
Create a vector of class names.

```
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 colorbar to the image.

```
cmap = jet(numel(classNames));
B = labeloverlay(histeq(train_data(:,:,4:6)),train_labels,'Transparency',0.8,'Colormap',cmap);

figure
title('Training Labels')
imshow(B)
N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
```



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` to store the label patches containing the 18 labeled regions.

```
pixelLabelIds = 1:18;
pxds = pixelLabelDatastore('train_labels.png',classNames,pixelLabelIds);
```

Create a `randomPatchExtractionDatastore` 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)
```

```
  58x1 Layer array with layers:

     1   'ImageInputLayer'                           Image Input                 256x256x6 images w
     2   'Encoder-Section-1-Conv-1'                  Convolution                 64 3x3x6 convoluti
     3   'Encoder-Section-1-ReLU-1'                  ReLU                        ReLU
     4   'Encoder-Section-1-Conv-2'                  Convolution                 64 3x3x64 convolut
     5   'Encoder-Section-1-ReLU-2'                  ReLU                        ReLU
     6   'Encoder-Section-1-MaxPool'                 Max Pooling                 2x2 max pooling wi
     7   'Encoder-Section-2-Conv-1'                  Convolution                 128 3x3x64 convolu
     8   'Encoder-Section-2-ReLU-1'                  ReLU                        ReLU
     9   'Encoder-Section-2-Conv-2'                  Convolution                 128 3x3x128 convol
    10   'Encoder-Section-2-ReLU-2'                  ReLU                        ReLU
    11   'Encoder-Section-2-MaxPool'                 Max Pooling                 2x2 max pooling wi
    12   'Encoder-Section-3-Conv-1'                  Convolution                 256 3x3x128 convol
    13   'Encoder-Section-3-ReLU-1'                  ReLU                        ReLU
    14   'Encoder-Section-3-Conv-2'                  Convolution                 256 3x3x256 convol
    15   'Encoder-Section-3-ReLU-2'                  ReLU                        ReLU
    16   'Encoder-Section-3-MaxPool'                 Max Pooling                 2x2 max pooling wi
    17   'Encoder-Section-4-Conv-1'                  Convolution                 512 3x3x256 convol
    18   'Encoder-Section-4-ReLU-1'                  ReLU                        ReLU
    19   'Encoder-Section-4-Conv-2'                  Convolution                 512 3x3x512 convol
    20   'Encoder-Section-4-ReLU-2'                  ReLU                        ReLU
    21   'Encoder-Section-4-DropOut'                 Dropout                     50% dropout
    22   'Encoder-Section-4-MaxPool'                 Max Pooling                 2x2 max pooling wi
    23   'Mid-Conv-1'                                Convolution                 1024 3x3x512 convo
    24   'Mid-ReLU-1'                                ReLU                        ReLU
    25   'Mid-Conv-2'                                Convolution                 1024 3x3x1024 conv
    26   'Mid-ReLU-2'                                ReLU                        ReLU
    27   'Mid-DropOut'                               Dropout                     50% dropout
    28   'Decoder-Section-1-UpConv'                  Transposed Convolution      512 2x2x1024 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 3x3x1024 convo
    32   'Decoder-Section-1-ReLU-1'                  ReLU                        ReLU
    33   'Decoder-Section-1-Conv-2'                  Convolution                 512 3x3x512 convol
    34   'Decoder-Section-1-ReLU-2'                  ReLU                        ReLU
    35   'Decoder-Section-2-UpConv'                  Transposed Convolution      256 2x2x512 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 3x3x512 convol
    39   'Decoder-Section-2-ReLU-1'                  ReLU                        ReLU
    40   'Decoder-Section-2-Conv-2'                  Convolution                 256 3x3x256 convol
    41   'Decoder-Section-2-ReLU-2'                  ReLU                        ReLU
    42   'Decoder-Section-3-UpConv'                  Transposed Convolution      128 2x2x256 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 3x3x256 convol
    46   'Decoder-Section-3-ReLU-1'                  ReLU                        ReLU
    47   'Decoder-Section-3-Conv-2'                  Convolution                 128 3x3x128 convol
    48   'Decoder-Section-3-ReLU-2'                  ReLU                        ReLU
    49   'Decoder-Section-4-UpConv'                  Transposed Convolution      64 2x2x128 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 3x3x128 convolu
    53   'Decoder-Section-4-ReLU-1'               ReLU                          ReLU
    54   'Decoder-Section-4-Conv-2'               Convolution                   64 3x3x64 convolut
    55   'Decoder-Section-4-ReLU-2'               ReLU                          ReLU
    56   'Final-ConvolutionLayer'                 Convolution                   18 1x1x64 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

After configuring the training options and the random patch extraction datastore, train the U-Net network by using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained U-Net network.

Note: Training takes about 20 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    save(['multispectralUnet-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net','options
```

```
else
    load(fullfile(imageDir,'trainedUnet','multispectralUnet.mat'));
end
```

You can now use the U-Net to semantically segment the multispectral image.

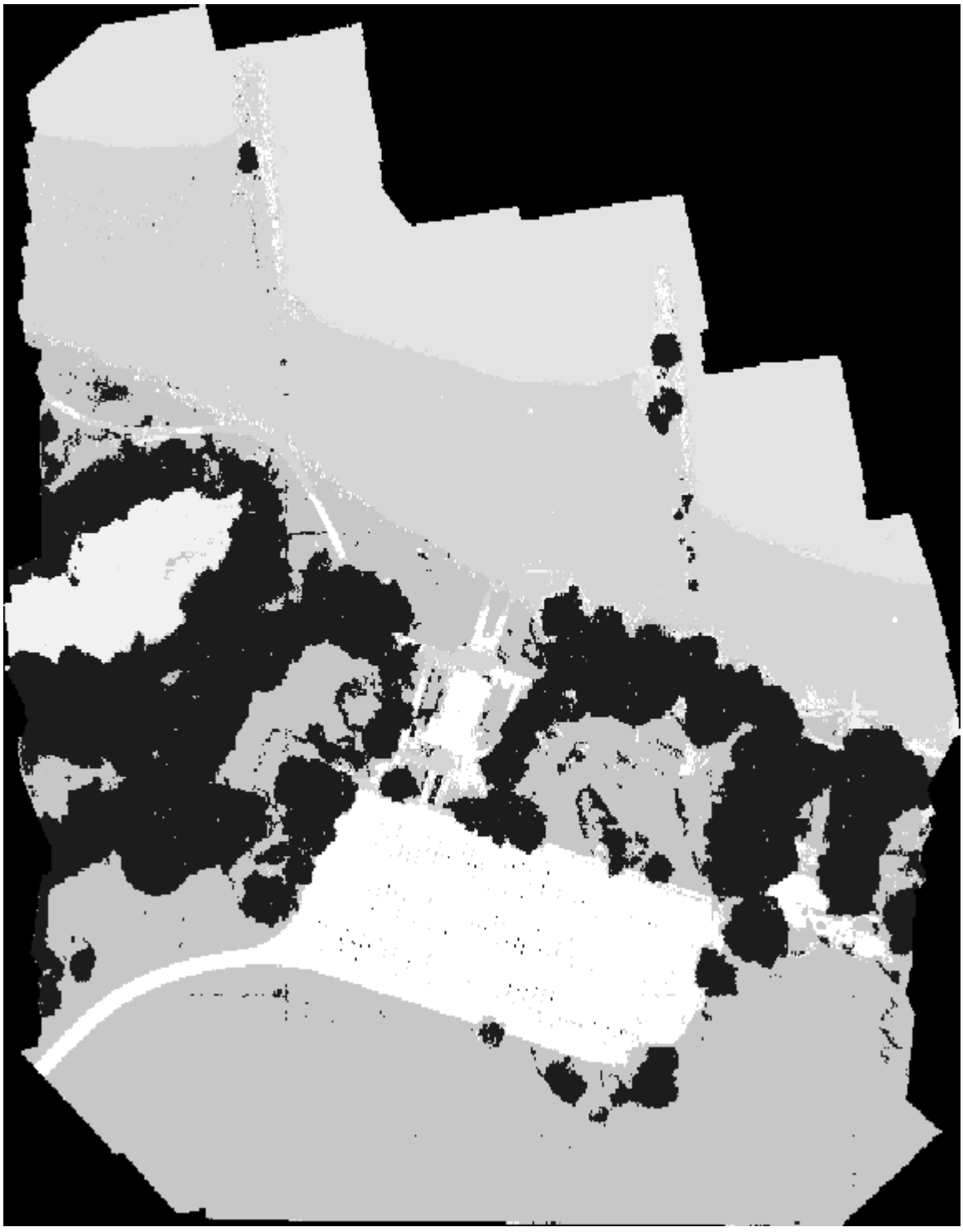
Predict Results on Test Data

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` 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')
```

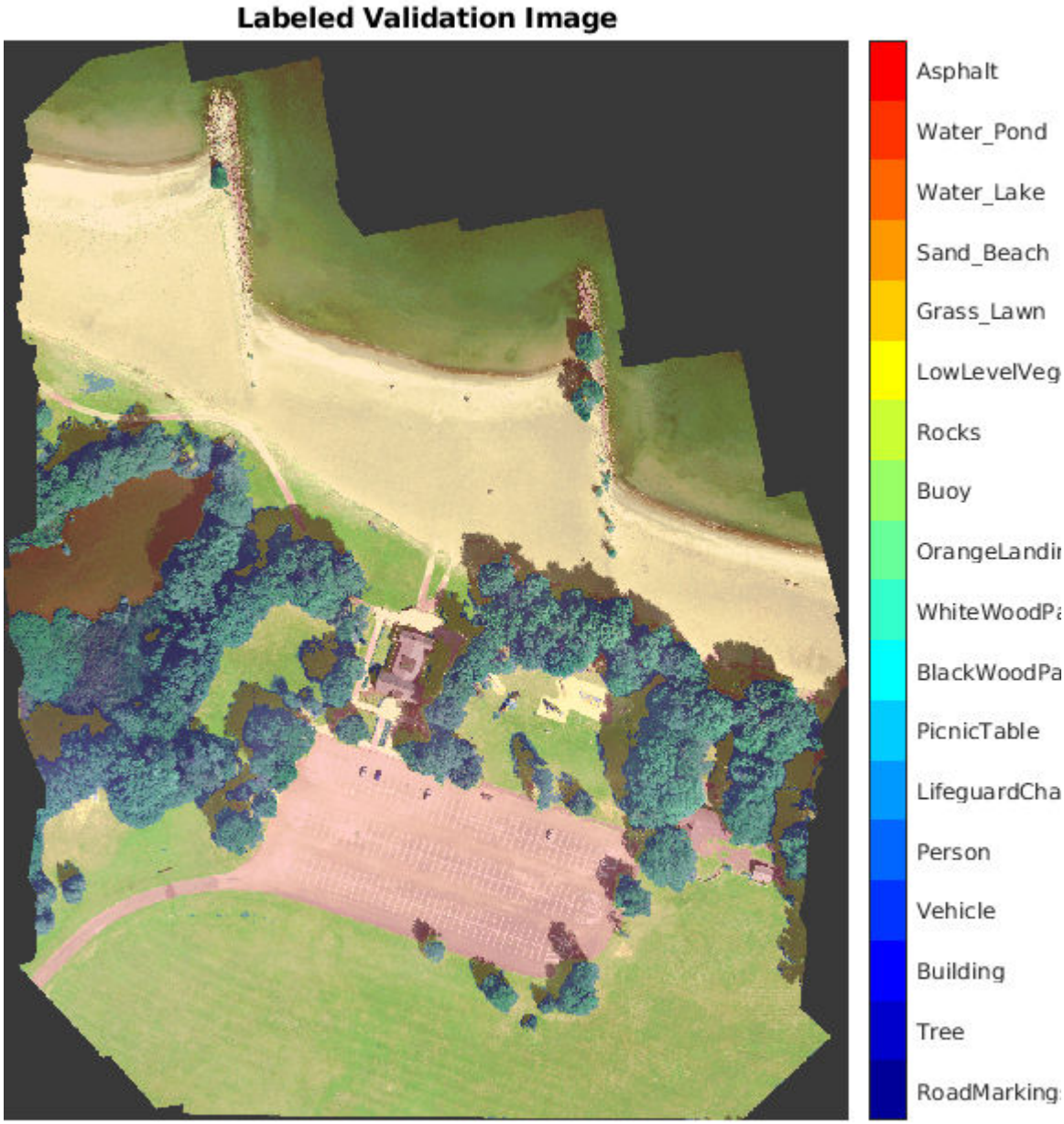


The output of semantic segmentation is noisy. Perform post image processing to remove noise and stray pixels. Use the `medfilt2` function to remove salt-and-pepper noise from the segmentation. Visualize the segmented image with the noise removed.

```
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` 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` 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

`evaluateSemanticSegmentation` | `histeq` | `imageDatastore` | `pixelLabelDatastore` | `randomPatchExtractionDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions` | `unetLayers`

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation Using Deep Learning” on page 8-74
- “Semantic Segmentation Using Dilated Convolutions” on page 8-90
- “Datastores for Deep Learning” on page 16-2

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. The example shows how to train a 3-D U-Net network and also provides a pretrained network. Use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for 3-D semantic segmentation (requires Parallel Computing Toolbox™).

Introduction

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);
```

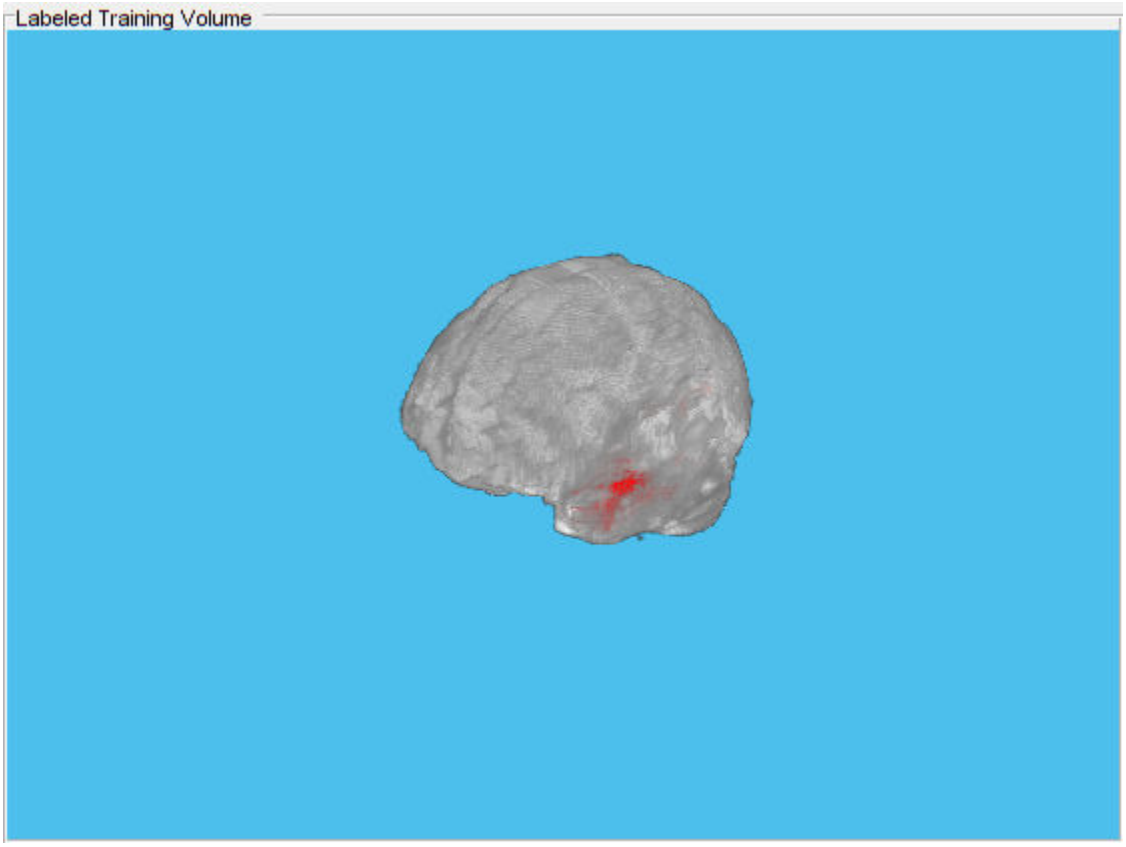
Create a `pixelLabelDatastore` to store the labels.

```
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` 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` 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;
```

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,dataSource));

dataSource = 'Validation';
dsVal = transform(dsVal,@(patchIn)augmentAndCrop3dPatch(patchIn,dataSource));
```

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` 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.

```
inputPatchSize = [132 132 132 4];
numClasses = 2;
[lgraph,outPatchSize] = unet3dLayers(inputPatchSize,numClasses,'ConvolutionPadding','valid');
```

To better segment smaller tumor regions and reduce the influence of larger background regions, this example uses a `dicePixelClassificationLayer`. 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

Optionally, 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);
```

```
Downloading pretrained 3-D U-Net for BraTS data set.
This will take several minutes to download...
Done.

Downloading sample BraTS test dataset.
This will take several minutes to download and unzip...
Done.
```

Train Network

After configuring the training options and the data source, train the 3-D U-Net network by using the `trainNetwork` function. To train the network, set the `doTraining` variable in the following code to `true`. A CUDA capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` variable in the following code as `false`, then the example returns a pretrained 3-D U-Net network.

Note: 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 = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,lgraph,options);
    save(['trained3DUNetValid-' 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
```

You can now use the U-Net to semantically segment brain tumors.

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` 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,outI
    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
```

```
Processing test volume 1
Processing test volume 2
Processing test volume 3
Processing test volume 4
Processing test volume 5
```

Compare Ground Truth Against Network Prediction

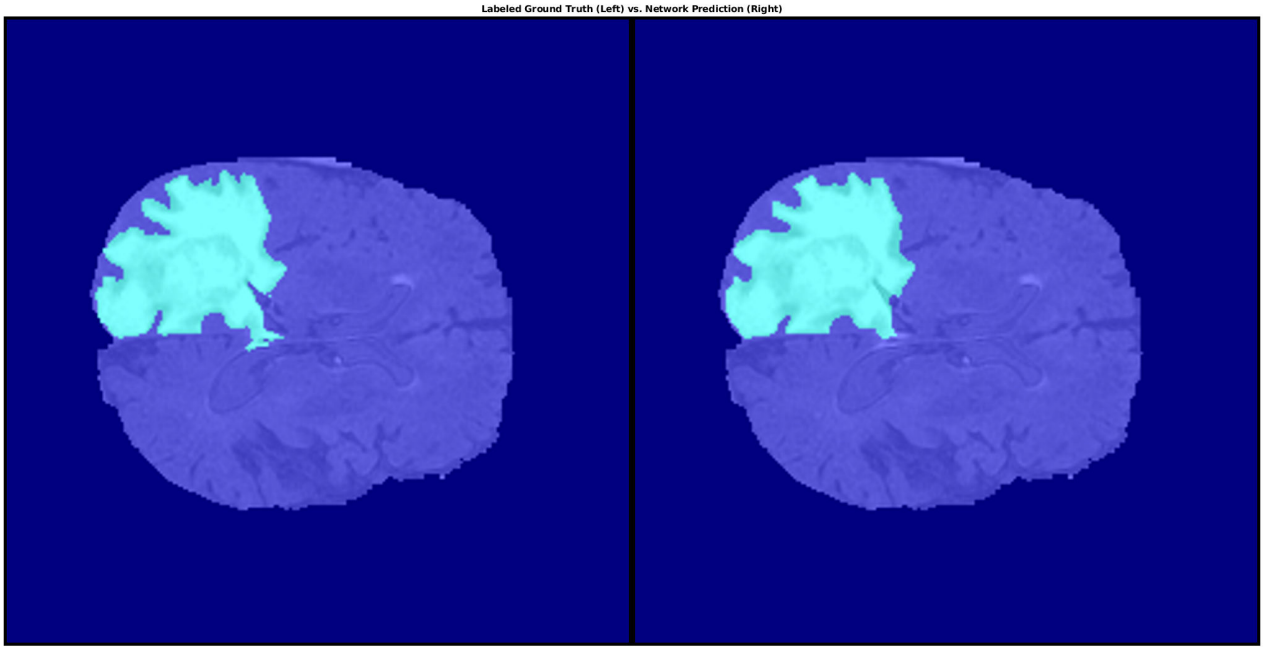
Select one of the test images to evaluate the accuracy of the semantic segmentation. Extract the first modality from the 4-D volumetric data and store this 3-D volume in the variable `vol3d`.

```
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)')
```

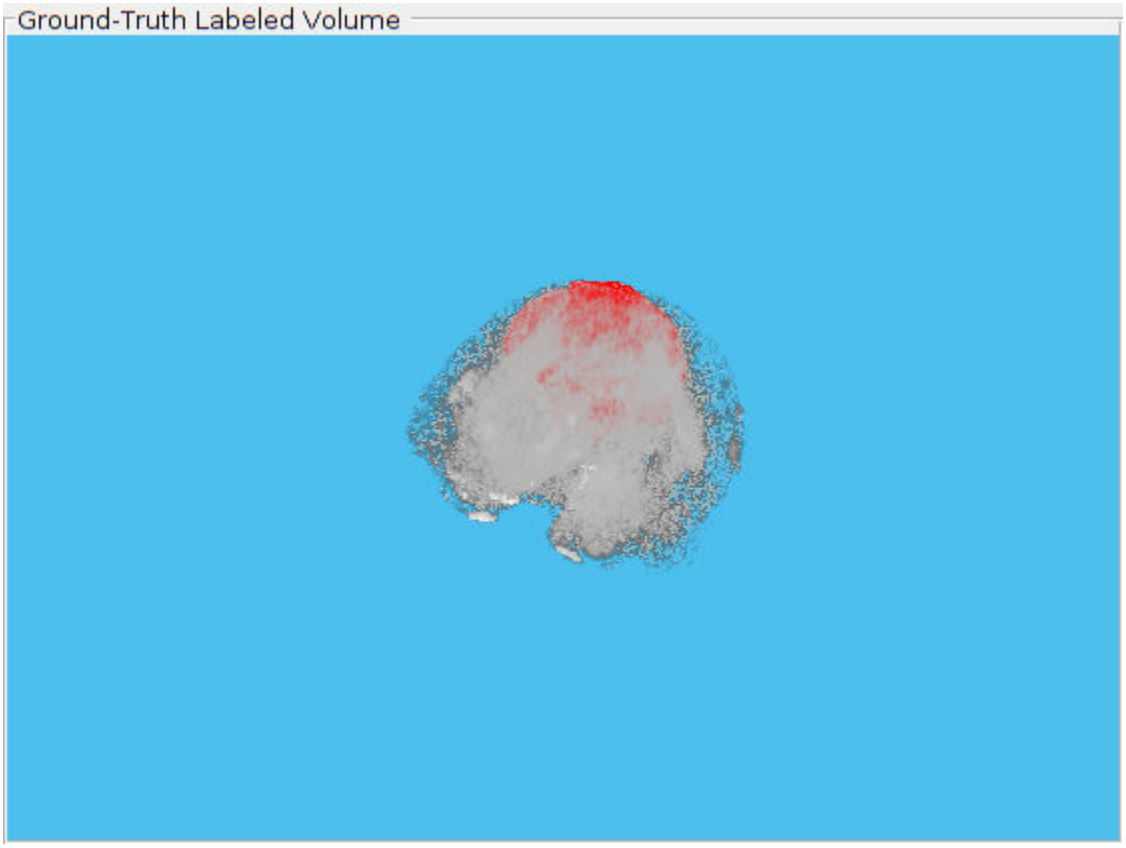


Display the ground-truth labeled volume using the `labelvolshow` function. Make the background fully transparent by setting the visibility of the background label (`1`) to `0`. Because the tumor is inside the brain tissue, make some of the brain voxels transparent, so that the tumor is visible. To make 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.

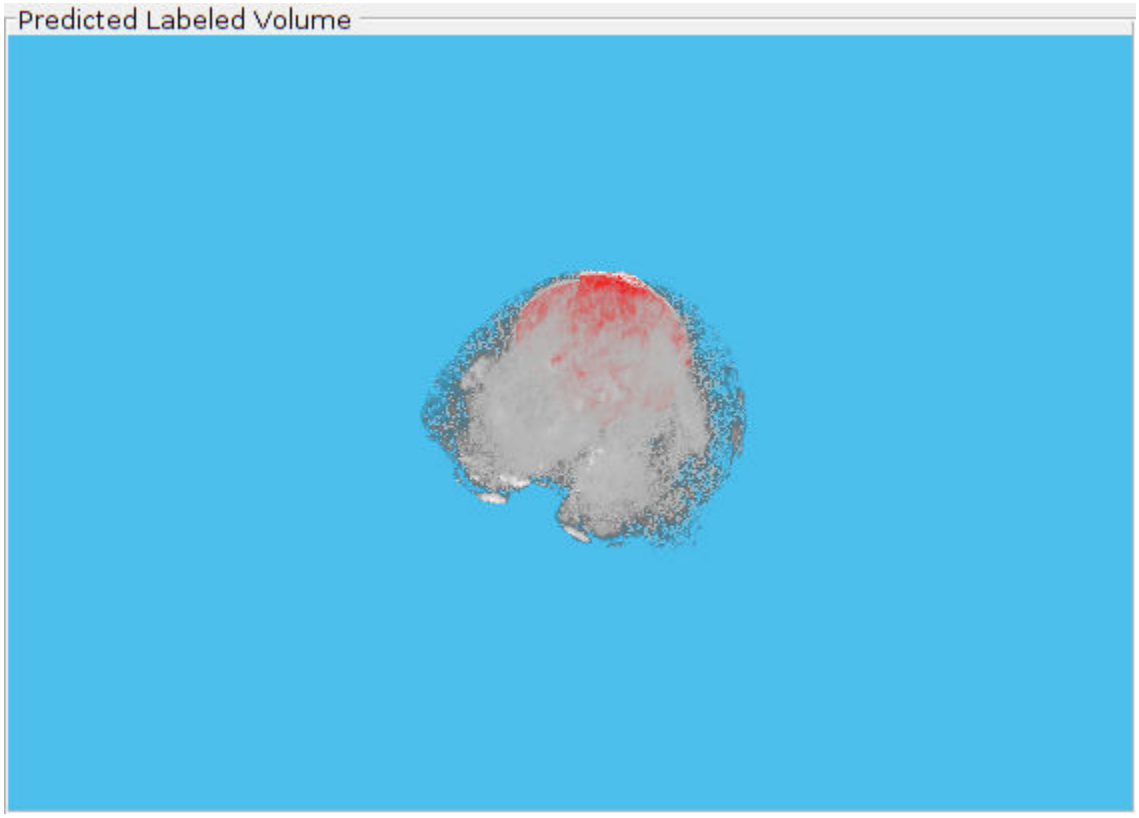
```
viewPnlTruth = uipanel(figure,'Title','Ground-Truth Labeled Volume');
hTruth = labelvolshow(groundTruthLabels{volId},vol3d,'Parent',viewPnlTruth, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
hTruth.LabelVisibility(1) = 0;
```

For the same volume, display the predicted labels.

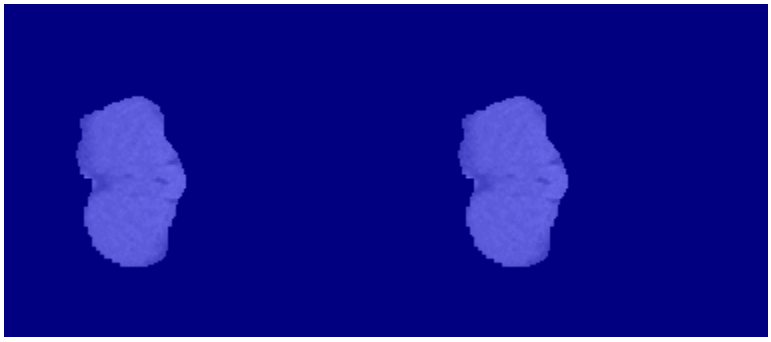
```
viewPnlPred = uipanel(figure,'Title','Predicted Labeled Volume');
hPred = labelvolshow(predictedLabels{volId},vol3d,'Parent',viewPnlPred, ...
    'LabelColor',[0 0 0;1 0 0],'VolumeThreshold',0.68);
```




```
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` 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.

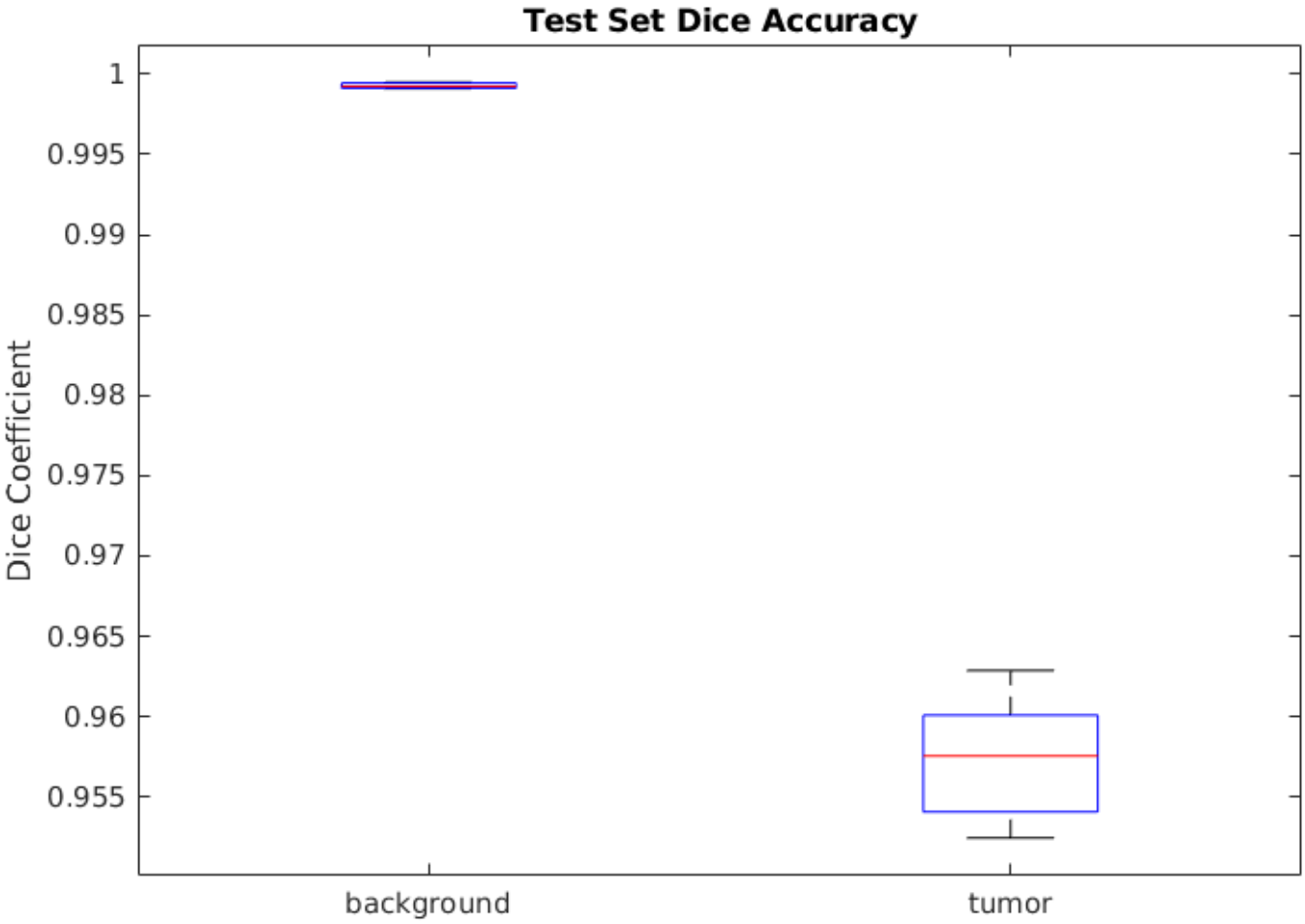
```
meanDiceBackground = mean(diceResult(:,1));
disp(['Average Dice score of background across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceBackground)])
```

```
Average Dice score of background across 5 test volumes = 0.9993
```

```
meanDiceTumor = mean(diceResult(:,2));
disp(['Average Dice score of tumor across ',num2str(j), ...
    ' test volumes = ',num2str(meanDiceTumor)])
```

```
Average Dice score of tumor across 5 test volumes = 0.9585
```

The figure shows a `boxplot` 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

`dicePixelClassificationLayer` | `imageDatastore` | `pixelLabelDatastore` | `randomPatchExtractionDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions` | `transform`

More About

- “Preprocess Volumes for Deep Learning” on page 16-12
- “Datastores for Deep Learning” on page 16-2
- “List of Deep Learning Layers” on page 1-23

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 15-2.

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

$$\mathrm{TI}_c = \frac{\sum_{m=1}^{M} Y_{\mathrm{cm}} T_{\mathrm{cm}}}{\sum_{m=1}^{M} Y_{\mathrm{cm}} T_{\mathrm{cm}} + \alpha \sum_{m=1}^{M} Y_{\mathrm{cm}} T_{\bar{\mathrm{c}}m} + \beta \sum_{m=1}^{M} Y_{\bar{\mathrm{c}}m} T_{\mathrm{cm}}}$$

- 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 15-194.

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.

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 2.2151 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)]
```

```
layers =
  11x1 Layer array with layers:
```

```
     1   ''          Image Input              32x32x1 images with 'zerocenter' normalization
     2   ''          Convolution              64 3x3 convolutions with stride [1  1] and padding
     3   ''          Batch Normalization      Batch normalization
     4   ''          ReLU                     ReLU
     5   ''          Max Pooling              2x2 max pooling with stride [2  2] and padding [0
     6   ''          Convolution              64 3x3 convolutions with stride [1  1] and padding
     7   ''          ReLU                     ReLU
     8   ''          Transposed Convolution   64 4x4 transposed convolutions with stride [2  2] a
     9   ''          Convolution              2 1x1 convolutions with stride [1  1] and padding
    10   ''          Softmax                  softmax
    11   'tversky'   Classification Output    Tversky loss
```

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 `pixelLabelImageDatastore`.

```
ds = pixelLabelImageDatastore(imds,pxds);
```

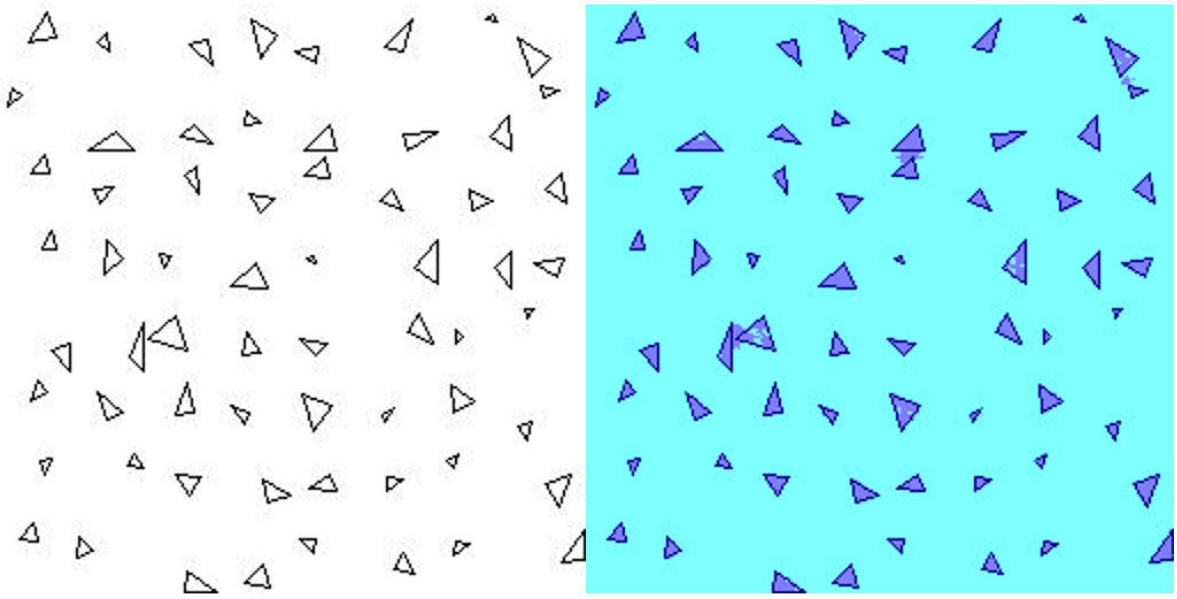
Set the training options and train the network.

```
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 |       00:00:02 |       50.32% |       1.2933 |          0.0010 |
|      13 |          50 |       00:00:47 |       98.82% |       0.0985 |          0.0010 |
|      25 |         100 |       00:01:31 |       99.32% |       0.0545 |          0.0005 |
|      38 |         150 |       00:02:13 |       99.37% |       0.0472 |          0.0005 |
|      50 |         200 |       00:02:51 |       99.48% |       0.0401 |          0.0003 |
|      63 |         250 |       00:03:33 |       99.48% |       0.0379 |          0.0001 |
|      75 |         300 |       00:04:10 |       99.54% |       0.0348 |          0.0001 |
|      88 |         350 |       00:04:46 |       99.51% |       0.0351 |      6.2500e-05 |
|     100 |         400 |       00:05:23 |       99.56% |       0.0330 |      6.2500e-05 |
|=========================================================================================|
```

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` | `pixelLabelDatastore` | `semanticseg` | `trainNetwork` | `trainingOptions`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation Using Deep Learning” on page 8-74

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 with compute capability 3.0 or higher is highly recommended for running this example. Use of a GPU requires the 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 with compute capability 3.0 or higher is highly recommended for training.

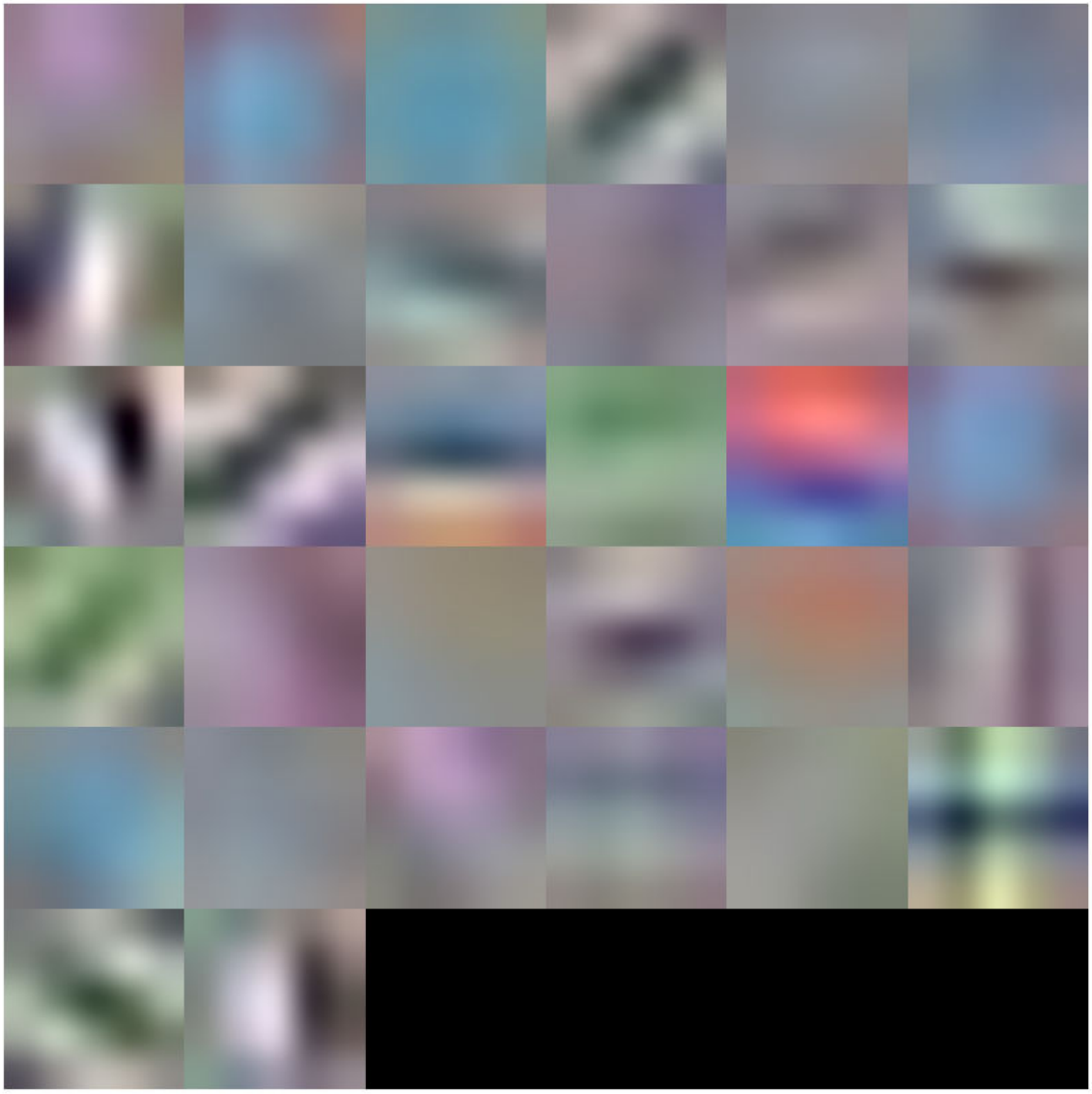
```
% A trained network is loaded from disk to save time when running the
% example. Set this flag to true to train the network.
```

```
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.

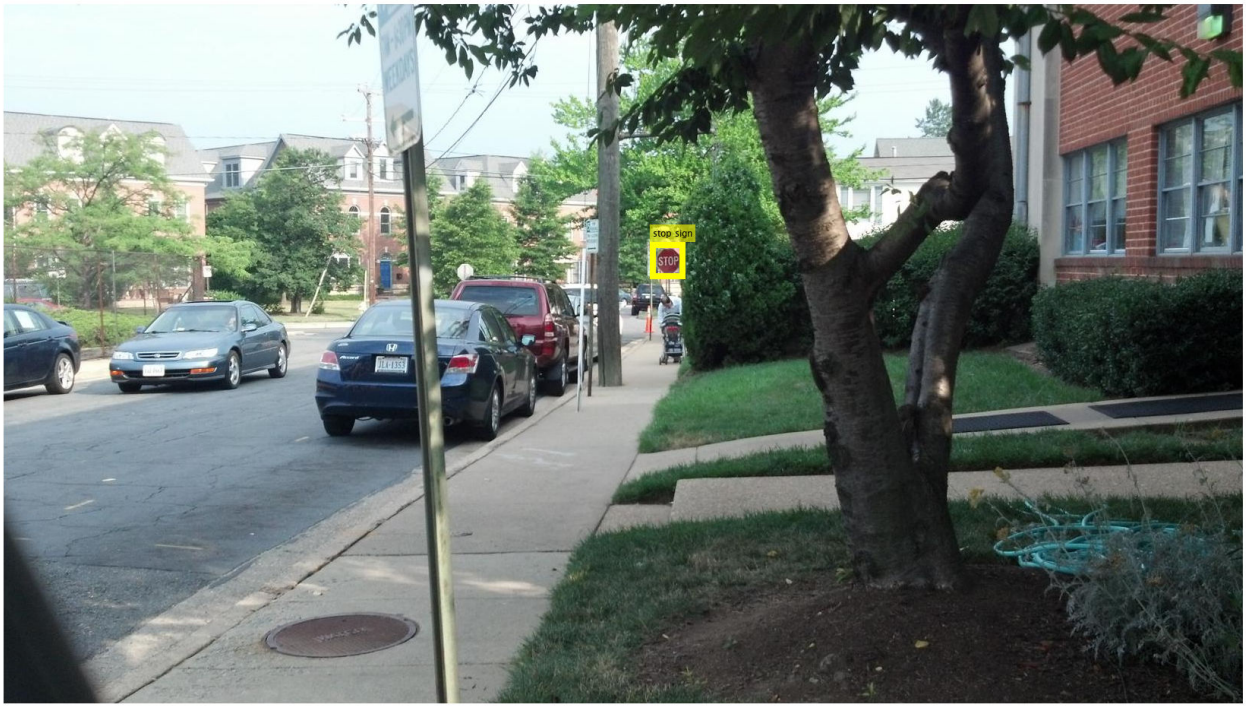
Start by loading the ground truth data for stop signs.

```
% 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`. 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 with compute capability 3.0 or higher 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` 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

`activations` | `classify` | `detect` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `rcnnObjectDetector` | `trainFastRCNNObjectDetector` |

`trainFasterRCNNObjectDetector` | `trainNetwork` | `trainRCNNObjectDetector` | `trainingOptions`

More About

- “Object Detection Using Faster R-CNN Deep Learning” on page 8-145
- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection using Deep Learning” (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 using Deep Learning” (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/fasterRCNNResNet50EndToEn
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Data Set

This example uses a small labeled dataset that contains 295 images. 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 1-12). 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

   136   119
    55    48
   157   128
```

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

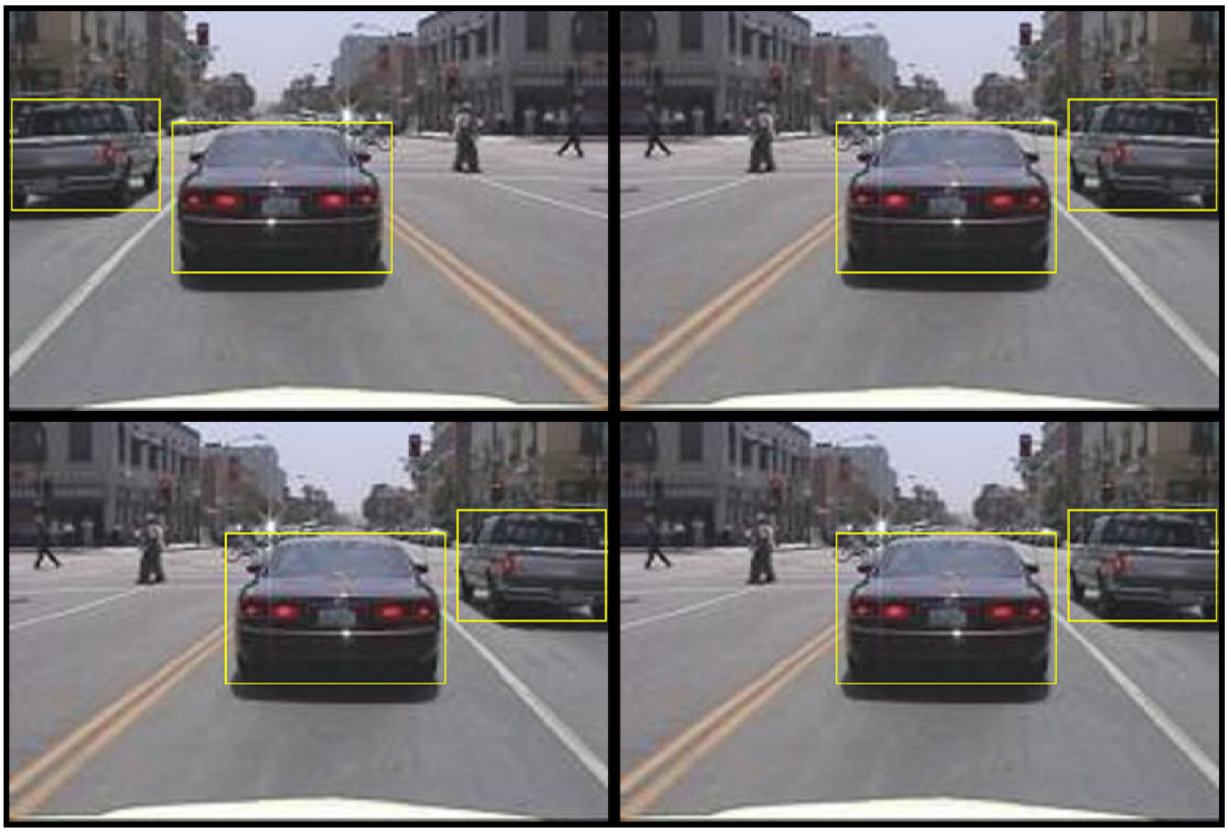
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

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 `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.

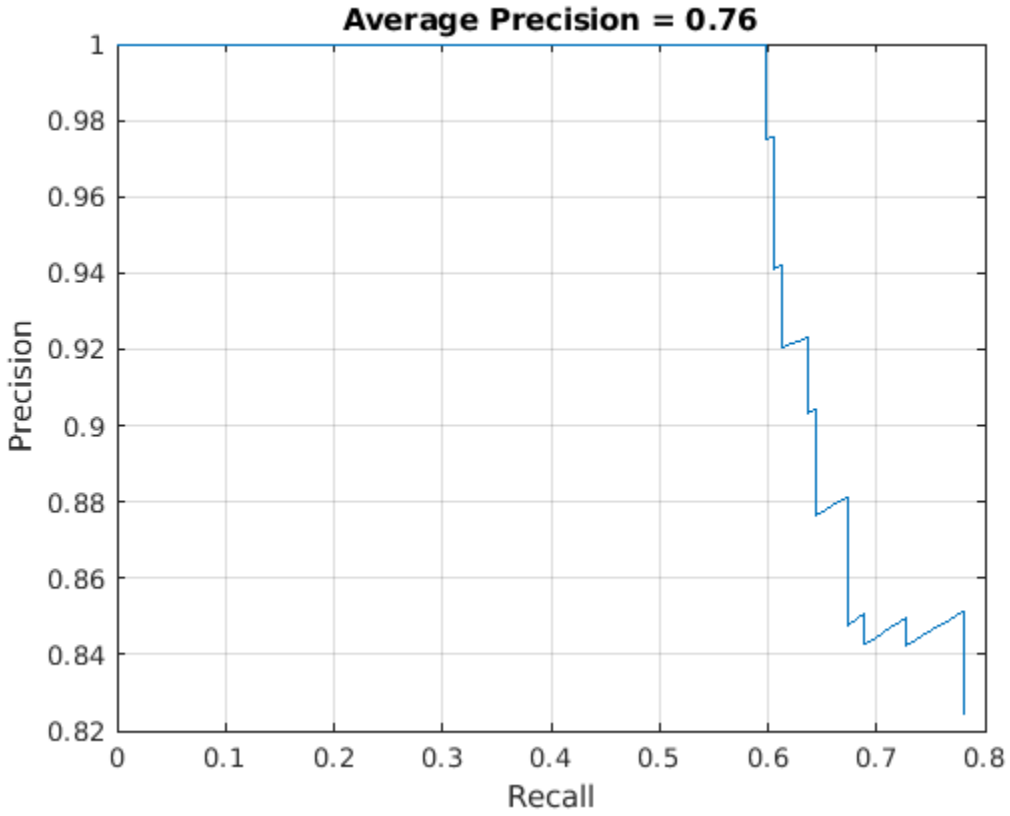
```
if doTrainingAndEval
    detectionResults = detect(detector,testData,'MinibatchSize',4);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detectionResults = pretrained.detectionResults;
end
```

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);
rout = affineOutputView(size(data{1}),tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
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

`detect` | `evaluateDetectionMissRate` | `evaluateDetectionPrecision` | `fastRCNNObjectDetector` | `fasterRCNNObjectDetector` | `insertObjectAnnotation` | `rcnnObjectDetector` | `trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `trainNetwork` | `trainRCNNObjectDetector` | `trainingOptions`

More About

- “Semantic Segmentation” (Computer Vision Toolbox)
- “Object Detection 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. Split the image into separate color channels, then denoise each channel using a pretrained denoising neural network, DnCNN.

Read a color image into the workspace and convert the data to `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. `imnoise` adds noise to each color channel independently. Display the noisy color image.

```
noisyRGB = imnoise(pristineRGB,'gaussian',0,0.01);
imshow(noisyRGB)
title('Noisy Image')
```

Split the noisy RGB image into its 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 0.7397.
```

```
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 0.9509.
```

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

`denoiseImage` | `denoisingNetwork` | `imnoise` | `lab2rgb` | `psnr` | `rgb2lab` | `ssim`

More About

- “Train and Apply Denoising Neural Networks” (Image Processing Toolbox)

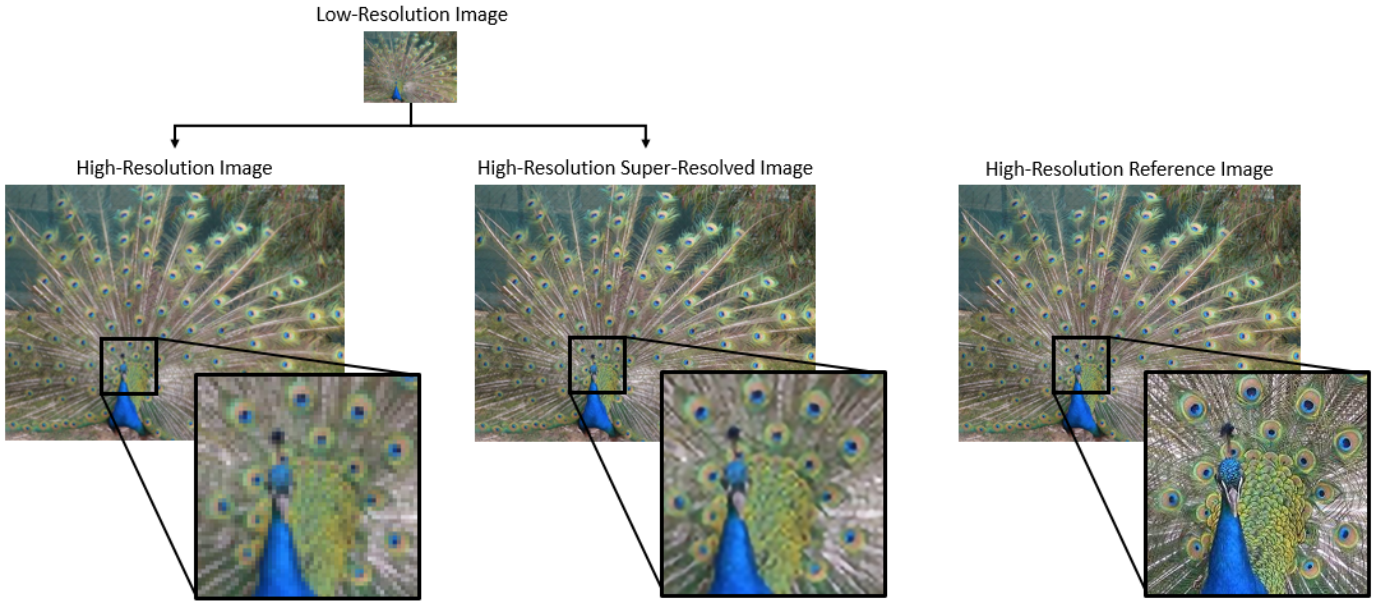
Single Image Super-Resolution Using Deep Learning

This example shows how to train a Very-Deep Super-Resolution (VDSR) neural network, then use a VDSR network to estimate a high-resolution image from a single low-resolution image.

The example shows how to train a VDSR network and also provides a pretrained VDSR network. If you choose to train the VDSR network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires the Parallel Computing Toolbox™.

Introduction

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].

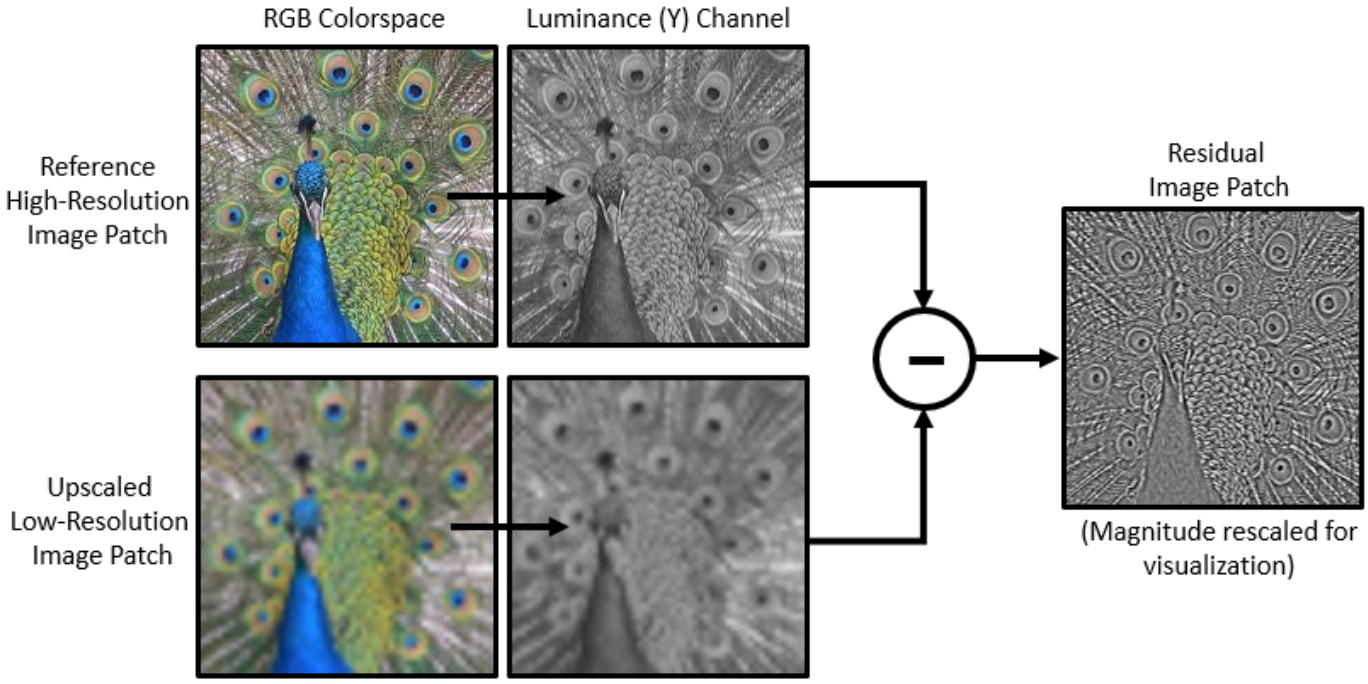
The VDSR Network

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` 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

After configuring the training options and the random patch extraction datastore, train the VDSR network using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained VDSR network that has been trained to super-resolve images for scale factors 2, 3 and 4.

Note: Training takes about 6 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    net = trainNetwork(dsTrain,layers,options);
    save(['trainedVDSR-' modelDateTime '-Epoch-' num2str(maxEpochs*epochIntervals) '-ScaleFactors
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.

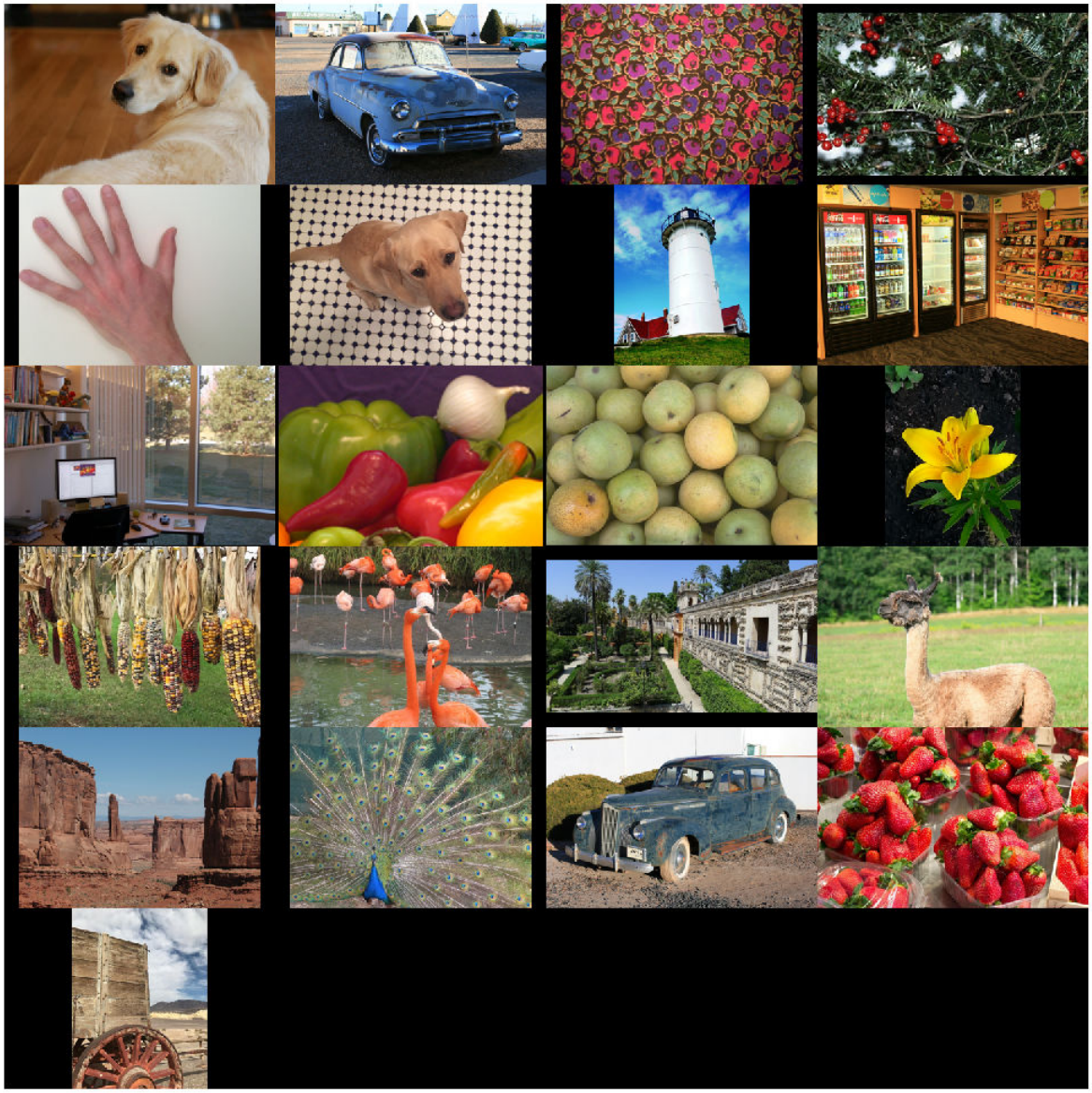
Create Sample Low-Resolution Image

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);
```

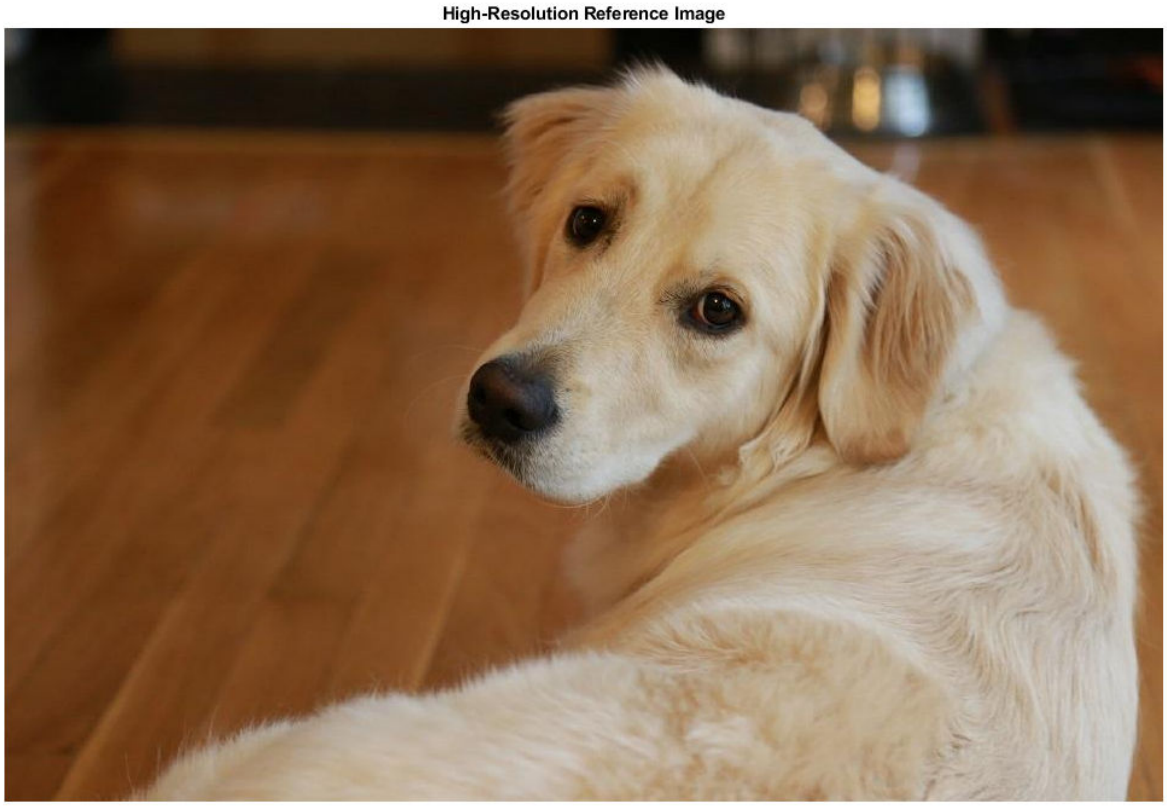
Display the testing images as a montage.

```
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` function.

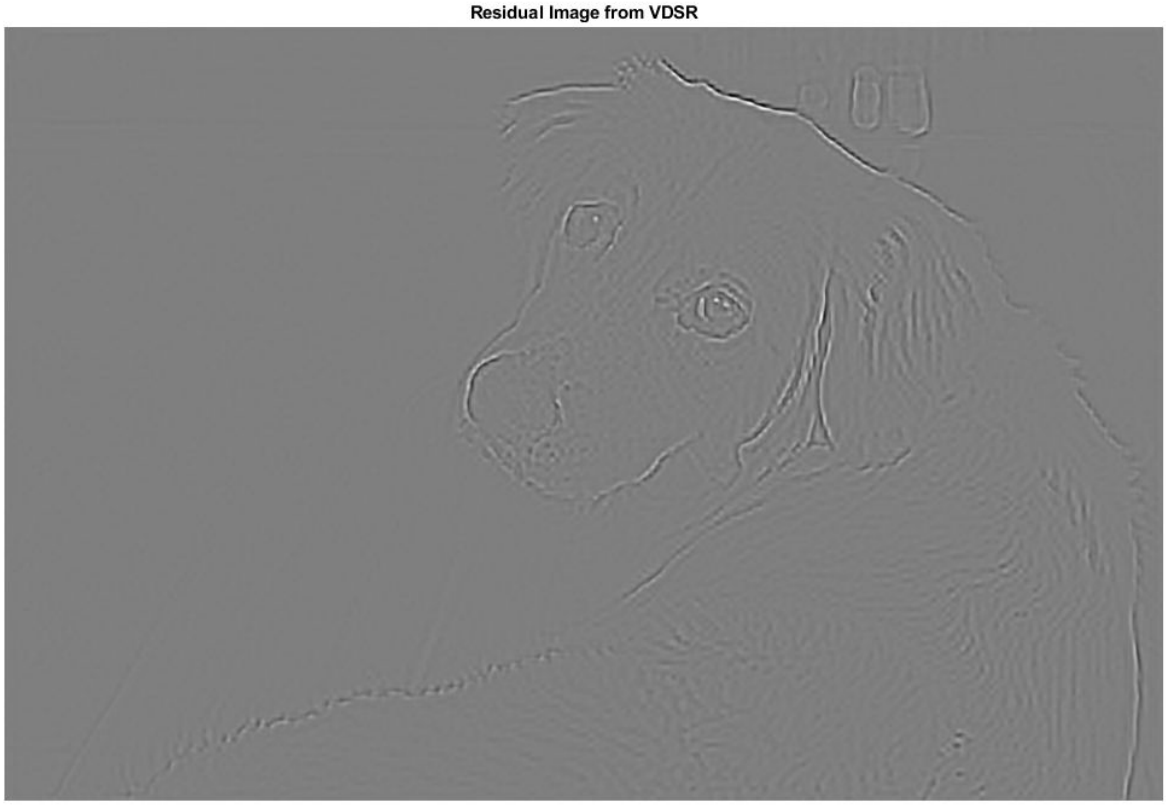
```
Iycbcr = rgb2ycbcr(Ilowres);
Iy = Iycbcr(:,:,1);
Icb = Iycbcr(:,:,2);
Icr = Iycbcr(:,:,3);
```

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` 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')
```

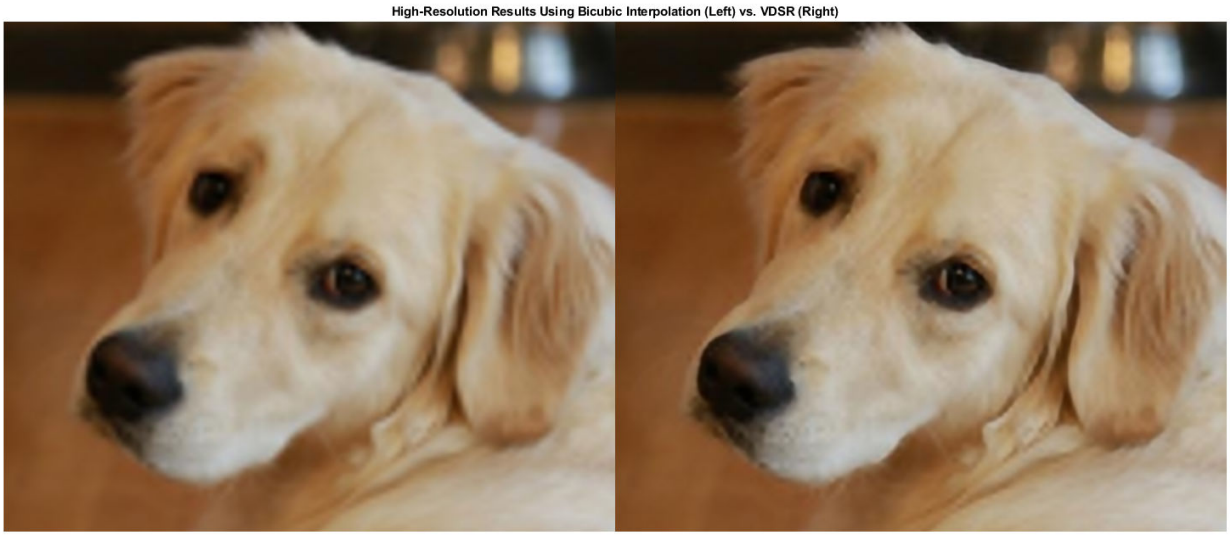
Visual and Quantitative Comparison

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` 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` 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` 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

`activations` | `combine` | `imageDataAugmenter` | `imageDatastore` | `trainNetwork` | `trainingOptions` | `transform`

More About

- “Datastores for Deep Learning” on page 16-2
- “Preprocess Images for Deep Learning” on page 16-8

- “List of Deep Learning Layers” on page 1-23

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.

The example shows how to train a DnCNN network and also provides a pretrained DnCNN network. If you choose to train the DnCNN network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended (requires Parallel Computing Toolbox™).

Introduction

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.

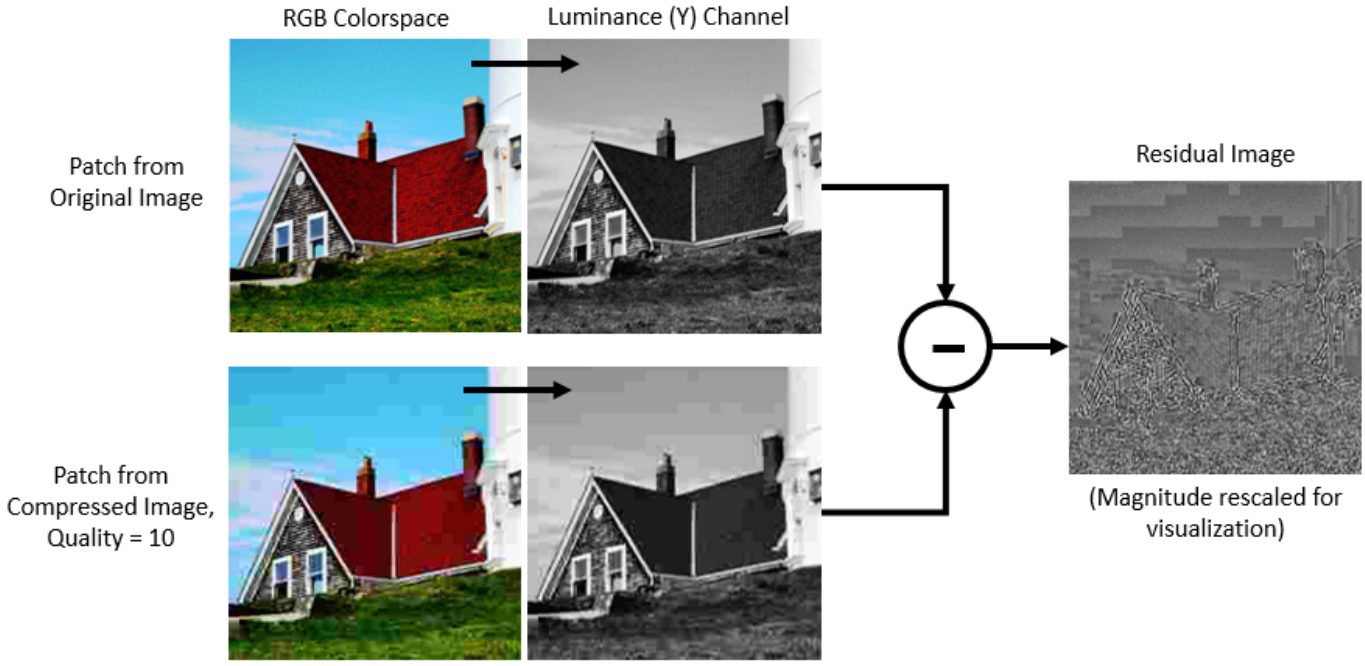
The DnCNN Network

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` 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` 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

After configuring the training options and the random patch extraction datastore, train the DnCNN network using the `trainNetwork` function. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained DnCNN network.

Note: Training takes about 40 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
% Training runs when doTraining is true
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    [net,info] = trainNetwork(dsTrain,layers,options);
    save(['trainedJPEGDnCNN-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net','options'
else
    load('pretrainedJPEGDnCNN.mat');
end
```

You can now use the DnCNN network to remove JPEG compression artifacts from new 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.

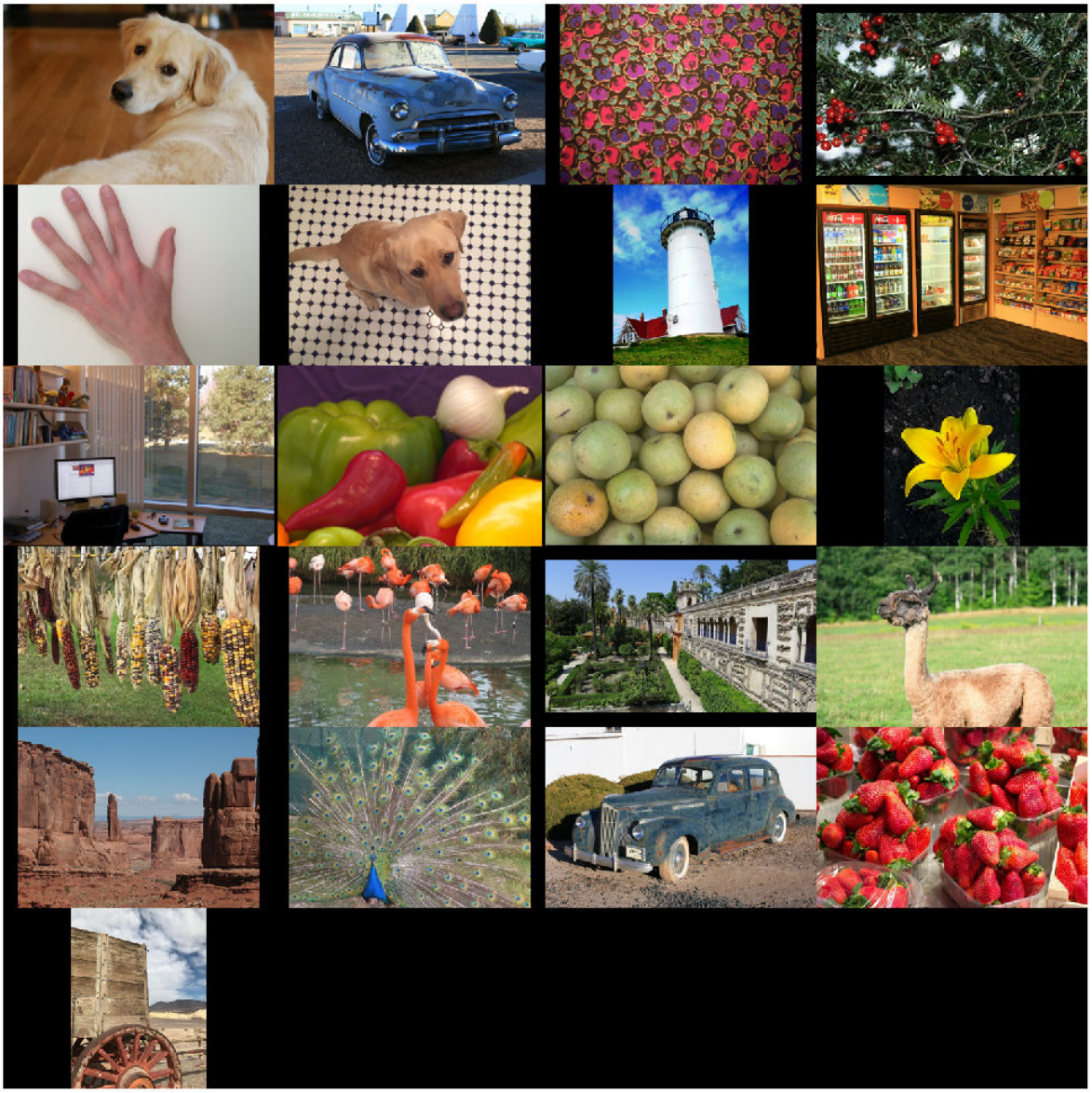
Create Sample Images with Blocking Artifacts

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` 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` 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` 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` 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` 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` 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` 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

`denoiseImage` | `dnCNNLayers` | `randomPatchExtractionDatastore` | `rgb2ycbcr` | `trainNetwork` | `trainingOptions` | `ycbcr2rgb`

More About

- “Preprocess Images for Deep Learning” on page 16-8
- “Datastores for Deep Learning” on page 16-2
- “List of Deep Learning Layers” on page 1-23

Image Processing Operator Approximation Using Deep Learning

This example shows how to train a multiscale context aggregation network (CAN) that is used to approximate an image filtering operation.

The example shows how to train a multiscale CAN network to approximate a bilateral filter, and also provides a pretrained network. If you choose to train the network, use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires the Parallel Computing Toolbox™.

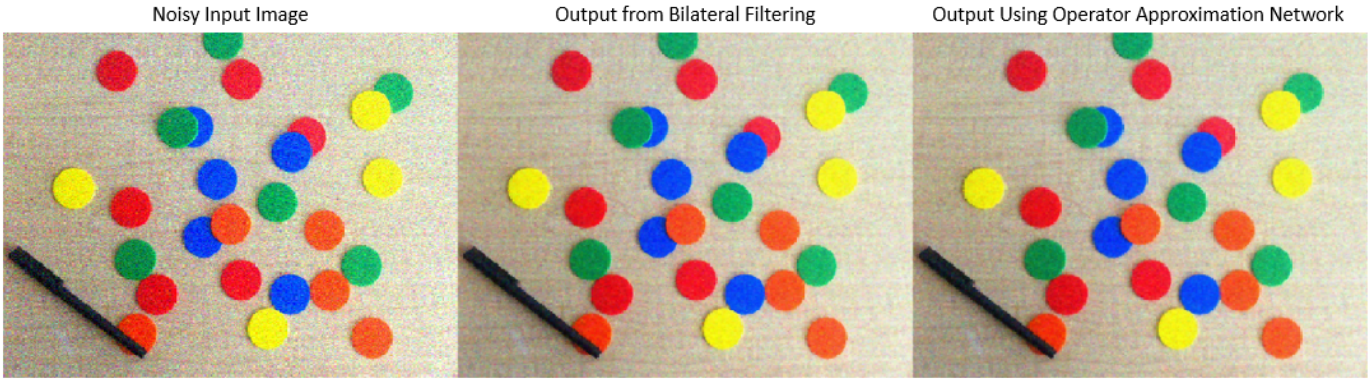
Introduction

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.

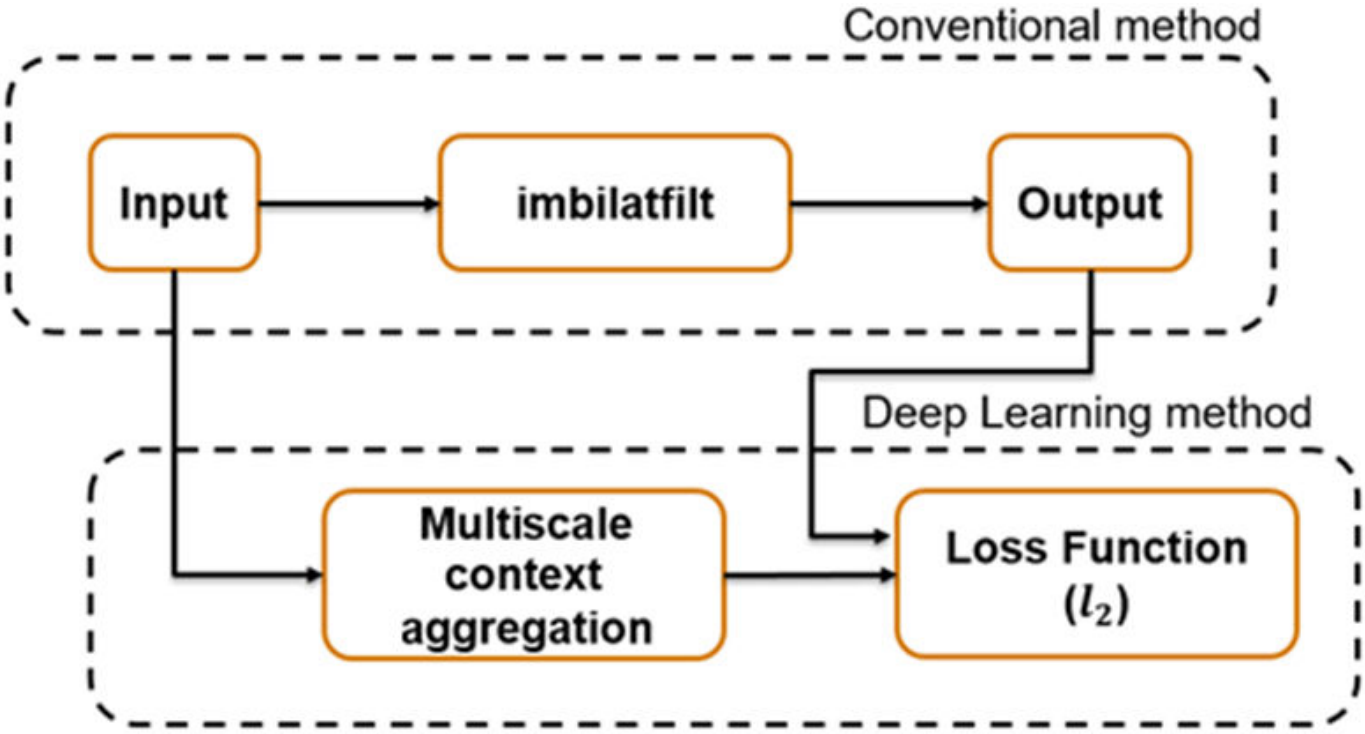
This example explores how to train a multiscale CAN to approximate a bilateral image filtering operation, which reduces image noise while preserving edge sharpness. The example presents the complete training and inference workflow, which includes the process of creating a training datastore, selecting 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`.
- 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` 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');
```

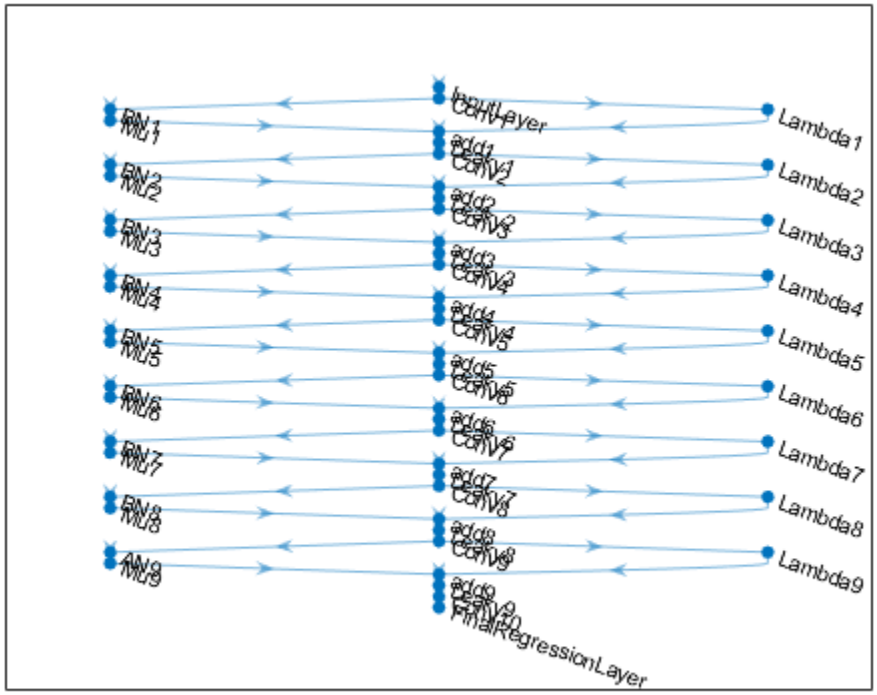
```
lgraph = addLayers(lgraph,skipConv1);
lgraph = connectLayers(lgraph,'Conv1','Lambda1');
lgraph = connectLayers(lgraph,'Lambda1','add1/in2');

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');
```

Alternatively, you can use this helper function to create the multiscale CAN layers.

```
layers = operatorApproximationLayers;
```

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

Now that the data source and training options are configured, train the multiscale CAN using the `trainNetwork function`. To train the network, set the `doTraining` parameter in the following code to `true`. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

If you keep the `doTraining` parameter in the following code as `false`, then the example returns a pretrained multiscale CAN that has been trained to approximate a bilateral filter.

Note: Training usually takes about 15 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    modelDateTime = datestr(now,'dd-mmm-yyyy-HH-MM-SS');
    net = trainNetwork(dsTrain,layers,options);
    save(['trainedOperatorLearning-' modelDateTime '-Epoch-' num2str(maxEpochs) '.mat'],'net');
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` 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.

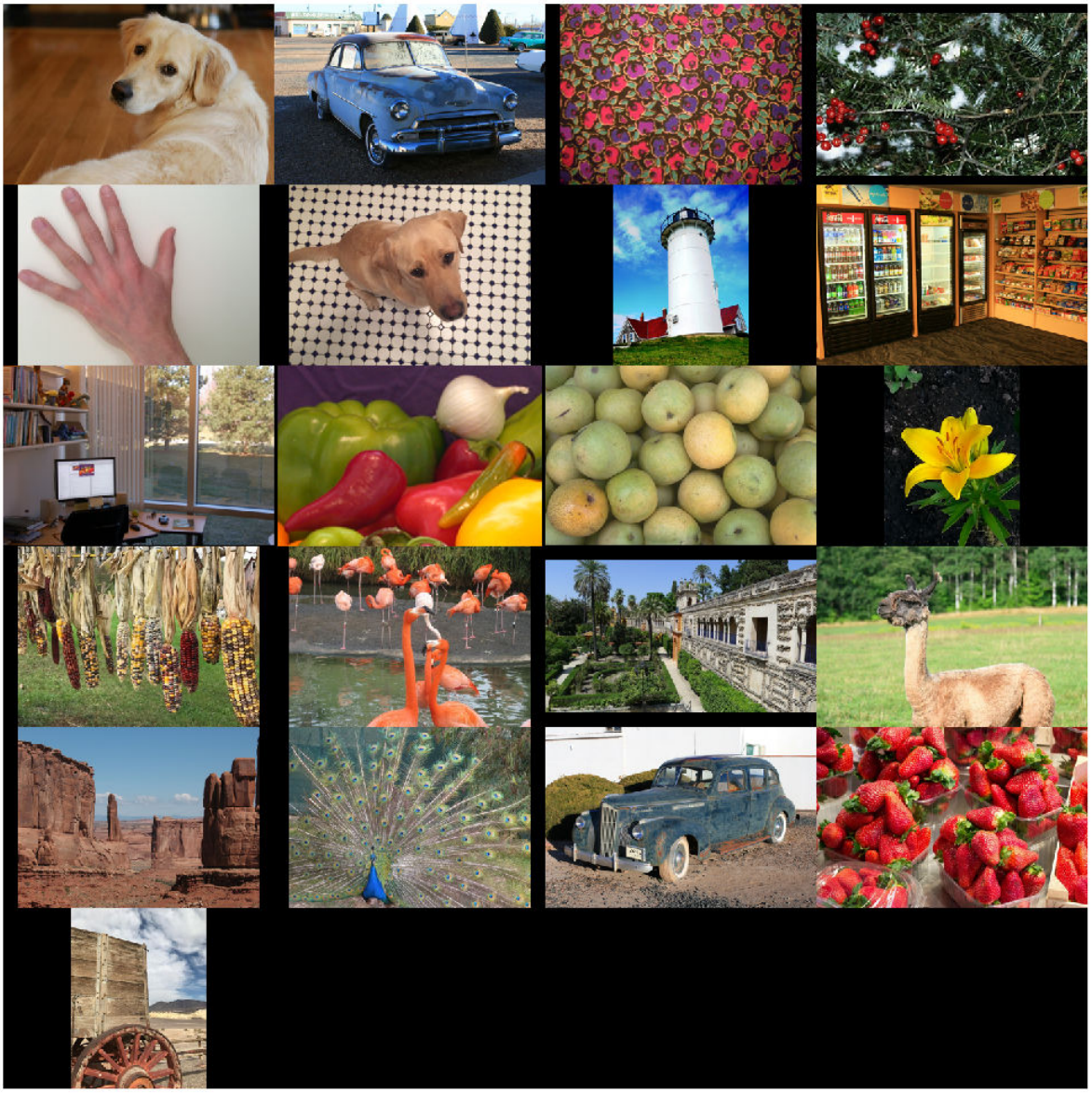
Create Sample Noisy Image

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);
```

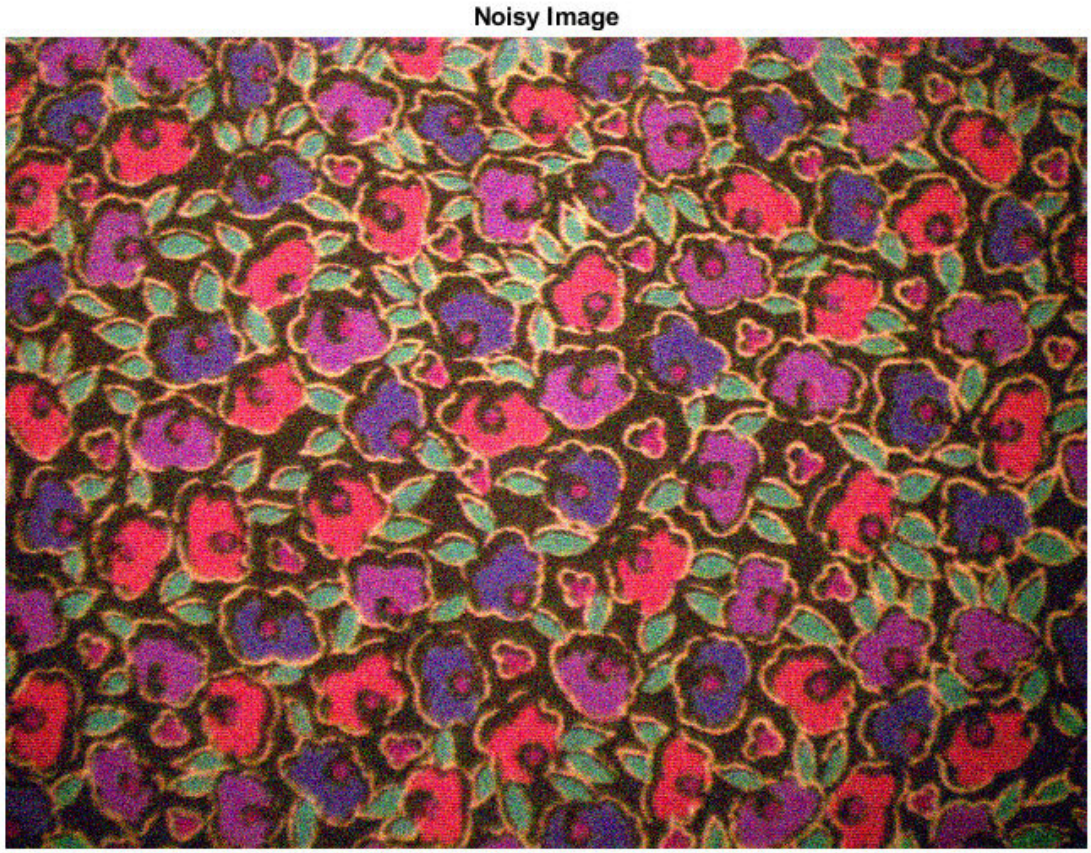
You can optionally use your own image as the reference image. Note that the size of the test image must be at least 256-by-256. If the test image is smaller than 256-by-256, then increase the image size by using the `imresize` function. The network also requires an RGB test image. If the test image is 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` 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` 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')
```

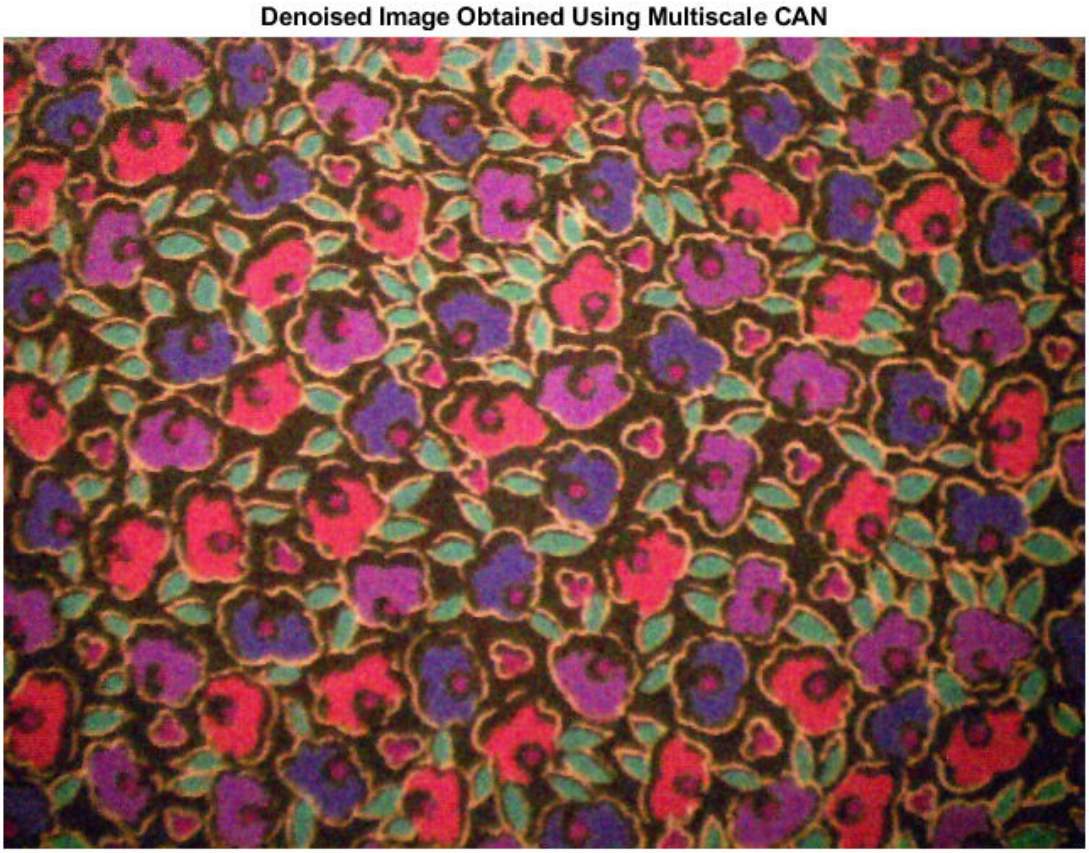
Process Image Using Trained Network

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')
```



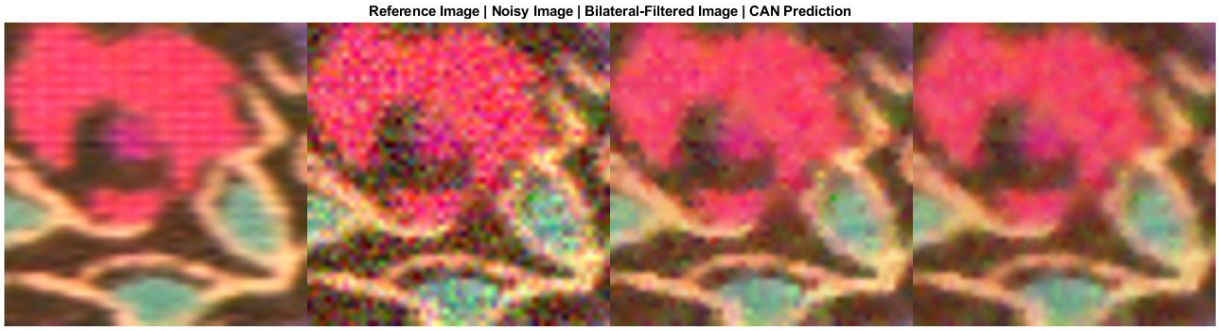
Visual and Quantitative Comparison

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` 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` 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` 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.

Summary

This example shows how to create and train a multiscale CAN to approximate a bilateral filter. These are the steps to train the network:

- Download the training data.
- Define a `randomPatchExtractionDatastore` that feeds training data to the network.
- Define the layers of the multiscale CAN.
- Specify training options.
- Train the network using the `trainNetwork` function.

After training the multiscale CAN or loading a pretrained network, the example approximates a bilateral filtering operation on a noisy image. The example compares the result against a conventional bilateral filter that does not use deep learning. Operator approximation outperforms bilateral filtering with respect to both perceptual image quality and quantitative quality measurements.

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

`activations` | `imageDatastore` | `imbilatfilt` | `layerGraph` | `randomPatchExtractionDatastore` | `trainNetwork` | `trainingOptions`

More About

- “Preprocess Images for Deep Learning” on page 16-8
- “Datastores for Deep Learning” on page 16-2
- “List of Deep Learning Layers” on page 1-23

Deep Learning Classification of Large Multiresolution Images

This example shows how to train an Inception-v3 deep neural network to classify multiresolution whole slide images (WSIs) that do not fit in memory.

The example shows how to train an Inception-v3 tumor classification network and also provides a pretrained Inception-v3 network. If you choose to train the Inception-v3 network, use of a CUDA capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended. Use of a GPU requires Parallel Computing Toolbox™.

Introduction

The only definitive way to diagnose breast cancer is by examining tissue samples collected from biopsy or surgery. The samples are commonly prepared with hematoxylin and eosin (H&E) staining to increase the contrast of structures in the tissue. Traditionally, pathologists examine the tissue on glass slides under a microscope to detect tumor tissue. Diagnosis takes time as pathologists must thoroughly inspect an entire slide at close magnification. Further, pathologists may not notice small tumors. Deep learning methods aim to automate the detection of tumor tissue, saving time and improving the detection rate of small tumors.

Deep learning methods for tumor classification rely on digital pathology, in which whole tissue slides are imaged and digitized. The resulting WSIs have extremely high resolution. WSIs are frequently stored in a multiresolution file to facilitate the display, navigation, and processing of the images.

Reading WSIs is a challenge because the images cannot be loaded as a whole into memory and therefore require out-of-core image processing techniques. Image Processing Toolbox™ `bigimage` objects can store and process this type of large multiresolution image. `bigimageDatastore` objects can prepare training patches from a `bigimage` to feed into a neural network.

This example shows how to train a deep learning network to classify tumors in very large multiresolution images using `bigimage` and `bigimageDatastore`. The example presents classification results as heatmaps that depict the probability that local tissue is tumorous. The localization of tumor regions enables medical pathologists to investigate specific regions and quickly identify tumors of any size in an image.

Download Training Data

This example uses WSIs from the Camelyon16 challenge [1 on page 9-0]. The data from this challenge contains a total of 400 WSIs of lymph nodes from two independent sources, separated into 270 training images and 130 test images. The WSIs are stored as TIF files in a stripped format with an 11-level pyramid structure.

`bigimage` objects warn when images contain stripped formats because this format is slow to process. To avoid displaying warnings in the command window, turn off the warning while the example runs and restore the warning state on completion.

```
warningState = warning('off','images:bigimage:singleStripTiff');
c = onCleanup(@()warning(warningState));
```

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.

The size of each training file is approximately 2 GB. If you do not want to download the training data set or train the network, then go directly to the Download and Preprocess Testing Data on page 9-0 section in this example.

Create a directory to store the training data set.

```
trainingImageDir = fullfile(tempdir,'Camelyon16','training');
if ~exist(trainingImageDir,'dir')
    mkdir(trainingImageDir);
    mkdir(fullfile(trainingImageDir,'normal'));
    mkdir(fullfile(trainingImageDir,'tumor'));
    mkdir(fullfile(trainingImageDir,'lesion_annotations'));
end

trainNormalDataDir = fullfile(trainingImageDir,'normal');
trainTumorDataDir = fullfile(trainingImageDir,'tumor');
trainTumorAnnotationDir = fullfile(trainingImageDir,'lesion_annotations');
```

To download the training data, go to the Camelyon17 website and click 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 `trainTumorAnnotationDir` variable.
- Open the "normal" directory. Download the images to the directory specified by the `trainNormalDataDir` variable.
- Open the "tumor" directory. Download the images to the directory specified by the `trainTumorDataDir` variable.

Specify the number of training images. Note that one of the training images of normal tissue, 'normal_144.tif', has metadata that cannot be read by the `bigimage` object. This example uses the remaining 158 training files.

```
numNormalFiles = 158;
numTumorFiles = 111;
```

Visualize Training Data

To get a better understanding of the training data, display one training image. Because loading the entire image into memory at the finest resolution is not possible, you cannot use traditional image display functions such as `imshow`. To display and process the image data, use a `bigimage` object.

Create a `bigimage` object from a tumor training image.

```
tumorFileName = fullfile(trainTumorDataDir,'tumor_001.tif');
tumorImage = bigimage(tumorFileName);
```

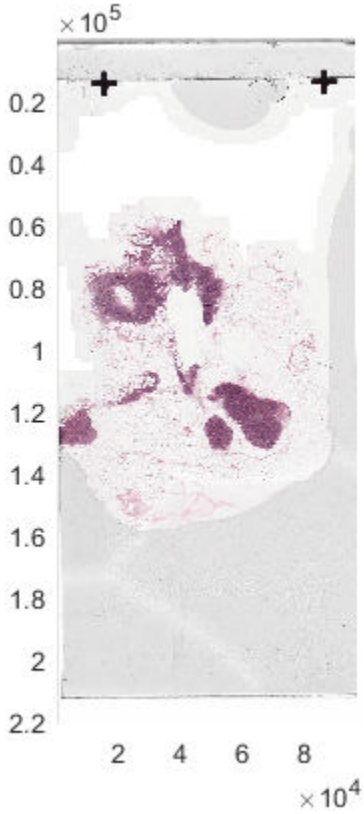
Inspect the dimensions of the `bigimage` at each resolution level. Level 1 has the most pixels and is the finest resolution level. Level 10 has the fewest pixels and is the coarsest resolution level. The aspect ratio is not consistent, which indicates that levels do not all span the same world area.

```
levelSizeInfo = table((1:length(tumorImage.LevelSizes))', ...
    tumorImage.LevelSizes(:,1), ...
    tumorImage.LevelSizes(:,2), ...
    tumorImage.LevelSizes(:,1)./tumorImage.LevelSizes(:,2), ...
    'VariableNames',["Resolution Level" "Image Width" "Image Height" "Aspect Ratio"])
```

```
levelSizeInfo=11×4 table
    Resolution Level    Image Width    Image Height    Aspect Ratio
    ________________    ___________    ____________    ____________

           1            2.2118e+05        97792           2.2618
           2            1.1059e+05        49152             2.25
           3                 55296        24576             2.25
           4                 27648        12288             2.25
           5                 13824         6144             2.25
           6                  7168         3072           2.3333
           7                  3584         1536           2.3333
           8                  2048         1024                2
           9                  1024          512                2
          10                   512          512                1
          11                  1577         3629          0.43455
```

Display the `bigimage` at a coarse resolution level by using the `bigimageshow` function. Return a handle to the `bigimageshow` object. You can use the handle to adjust the display. The image contains a lot of empty white space. The tissue occupies only a small portion of the image.

```
h = bigimageshow(tumorImage,'ResolutionLevel',7);
```

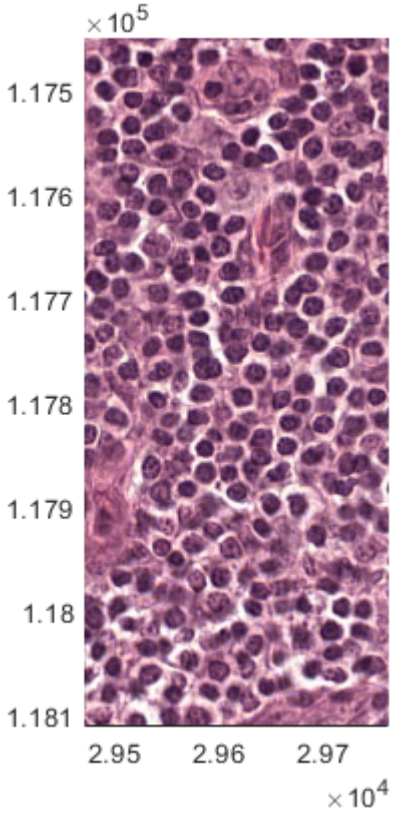


Zoom in on one part of the image by setting the horizontal and vertical spatial extents with respect to the finest resolution level. The image looks blurry because this resolution level is very coarse.

```
xlim([29471,29763]);
ylim([117450,118110]);
```

To see more details, change the resolution level to a finer level.

```
h.ResolutionLevel = 1;
```



Create Masks

You can reduce the amount of computation by processing only regions of interest (ROIs). Use a mask to define ROIs. A mask is a logical image in which `true` pixels represent the ROI.

To further reduce the amount of computation, create masks at a coarse resolution level that can be processed entirely in memory instead of on a block-by-block basis. If the spatial referencing of the coarse resolution level matches the spatial referencing of finer resolution levels, then locations at the coarse level correspond to locations in finer levels. In this case, you can use the coarse mask to select which blocks to process at the finer levels. For more information, see “Set Spatial Referencing for Big Images” (Image Processing Toolbox) and “Process Big Images Efficiently Using Mask” (Image Processing Toolbox).

Specify a resolution level to use for creating the mask. This example uses resolution level 7, which is coarse and fits in memory.

```
resolutionLevel = 7;
```

Create Masks for Normal Images

In normal images, the ROI consists of healthy tissue. The color of healthy tissue is distinct from the color of the background, so use color thresholding to segment the image and create an ROI. The L*a*b* color space provides the best color separation for segmentation. Convert the image to the L*a*b* color space, then threshold the a* channel to create the tissue mask.

You can use the helper function `createMaskForNormalTissue` to create masks using color thresholding. This helper function is attached to the example as a supporting file.

The helper function performs these operations on each training image of normal tissue:

- Create a `bigimage` object from the .TIF image file.
- Set the spatial referencing of all resolution levels from the image metadata.
- Get the image at the coarse resolution level.
- Convert the coarse image to the L*a*b* color space, then extract the a* channel.
- Create a binary image by thresholding the image using Otsu's method, which minimizes the intraclass variance between the black and white pixels.
- Create a single-resolution `bigimage` object from the mask and set the spatial referencing of the mask to match the spatial referencing of the input image.
- Write the mask `bigimage` to memory. Only the `bigimage` object is in memory. The individual image blocks corresponding to the logical mask image are in a temporary directory. Writing to a directory preserves the custom spatial referencing, which ensures that the normal images and their corresponding mask images have the same spatial referencing.

```
trainNormalMaskDir = fullfile(trainNormalDataDir,['normal_mask_level' num2str(resolutionLevel)]);
createMaskForNormalTissue(trainNormalDataDir,trainNormalMaskDir,resolutionLevel)
```

Now that both normal images and masks are on disk, create `bigimage` objects to manage the data by using the helper function `createBigImageAndMaskArrays`. This function creates an array of `bigimage` objects from the normal images and a corresponding array of `bigimage` objects from the normal mask images. The helper function is attached to the example as a supporting file.

```
[bigNormalImages,bigNormalMasks] = createBigImageAndMaskArrays(trainNormalDataDir,trainNormalMask
```

Select a sample normal image and mask. Confirm that the spatial world extents of the mask match the extents of the image at the finest resolution level. The spatial world extents are specified by the `XWorldLimits` and `YWorldLimits` properties.

```
idx = 2;
bigNormalImages(idx).SpatialReferencing(1)
```

```
ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 221184]
              ImageSize: [221184 97792]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 97792
    ImageExtentInWorldY: 221184
       XIntrinsicLimits: [0.5000 9.7793e+04]
       YIntrinsicLimits: [0.5000 2.2118e+05]
```

```
bigNormalMasks(idx).SpatialReferencing(1)
```

```
ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
```

```
          YWorldLimits: [0 221184]
             ImageSize: [3456 1527]
    PixelExtentInWorldX: 64.0419
    PixelExtentInWorldY: 64
    ImageExtentInWorldX: 97792
    ImageExtentInWorldY: 221184
       XIntrinsicLimits: [0.5000 1.5275e+03]
       YIntrinsicLimits: [0.5000 3.4565e+03]
```

Verify that the mask contains the correct ROIs and spatial referencing. Display the sample image by using the `bigimageshow` function. Get the axes containing the display.

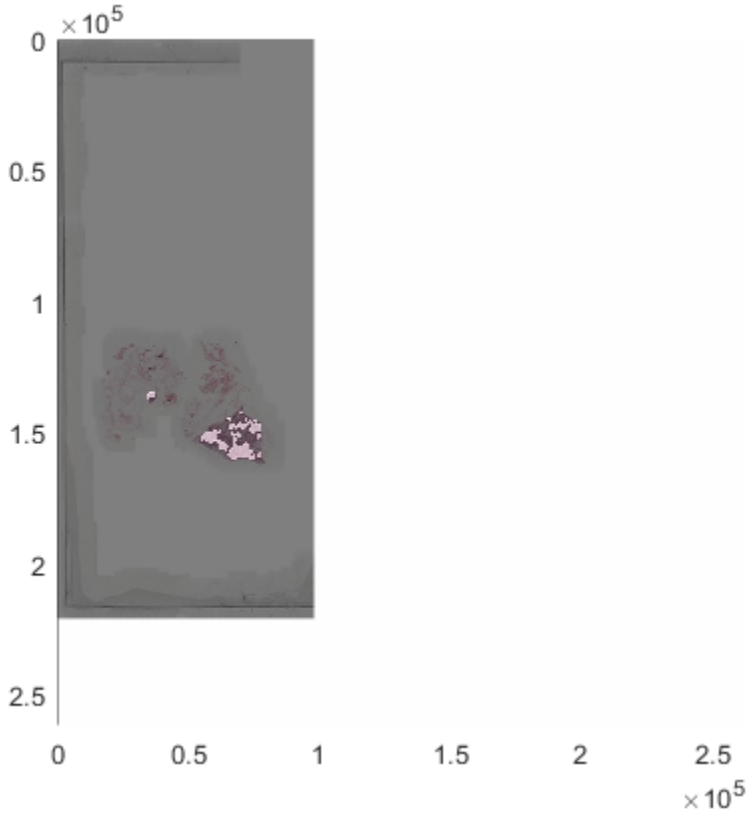
```
figure
hBigNormal = bigimageshow(bigNormalImages(idx));
hNormalAxes = hBigNormal.Parent;
```

Create a new axes on top of the displayed `bigimage`. In the new axes, display the corresponding mask image with partial transparency. The mask highlights the regions containing normal tissue.

```
hMaskAxes = axes;
hBigMask = bigimageshow(bigNormalMasks(idx),'Parent',hMaskAxes, ...
    'Interpolation','nearest','AlphaData',0.5);
hMaskAxes.Visible = 'off';
```

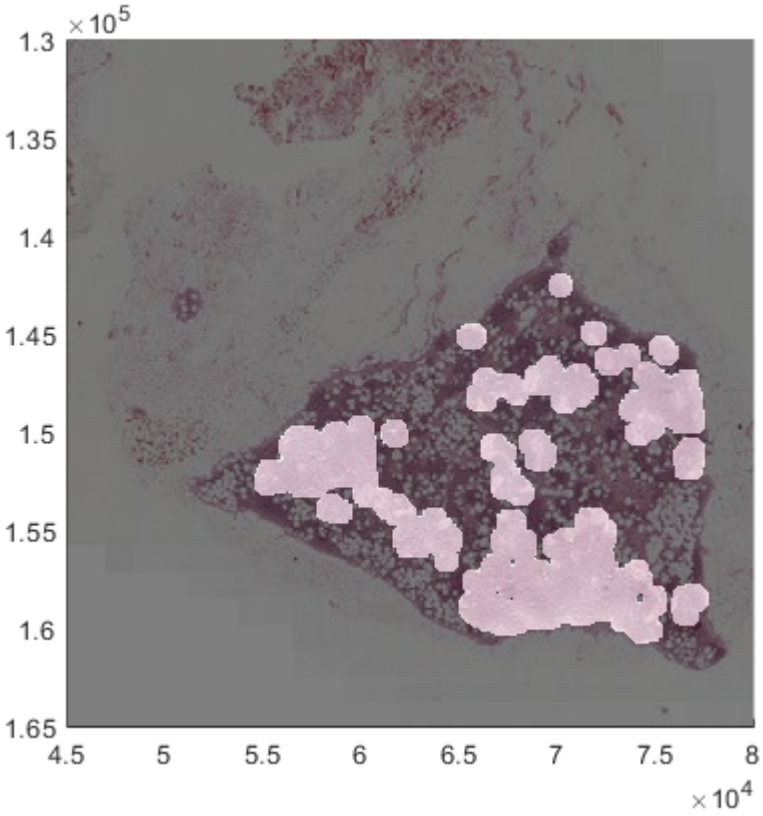
Link the axes of the image with the axes of the mask. When you zoom and pan, both axes are updated identically.

```
linkaxes([hNormalAxes,hMaskAxes]);
```



Zoom in on one part of the image by setting the horizontal and vertical spatial extents. The mask overlaps the normal tissue correctly.

```
xlim([45000 80000]);
ylim([130000 165000]);
```



Create Masks for Tumor Images

In tumor images, the ROI consists of tumor tissue. The color of tumor tissue is similar to the color of healthy tissue, so you cannot use color segmentation techniques. Instead, create ROIs by using the ground truth coordinates of the lesion boundaries that accompany the tumor images.

You can use the helper function `createMaskForTumorTissue` to create a mask using ROIs. This helper function is attached to the example as a supporting file.

The helper function performs these operations on each training image of tumor tissue:

- Create a `bigimage` object from the .TIF image file.
- Set the spatial referencing from the image metadata.
- Read the corresponding lesion annotations in XML files and convert the annotations to polygons (`Polygon` objects).
- For each image block, use the polygon data to create a mask for the corresponding block. Images with tumor regions can contain some normal regions within them. Use the annotations of normal tissue to exclude those regions.
- Create an output logical mask `bigimage` object at a coarser resolution level. Write the mask image on a block-by-block basis by using the `setBlock` function.

- Write the mask `bigimage` object to a directory in memory. Only the `bigimage` object is in memory. The individual image blocks corresponding to the logical mask image are in a temporary directory. Writing to a directory preserves the custom spatial referencing, which ensures that the tumor images and their corresponding mask images have the same spatial referencing.

```
trainTumorMaskDir = fullfile(trainTumorDataDir,['tumor_mask_level' num2str(resolutionLevel)]);
createMaskForTumorTissue(trainTumorDataDir,trainTumorAnnotationDir, ...
    trainTumorMaskDir,resolutionLevel);
```

Now that both tumor images and masks are on disk, create `bigimage` objects to manage the data by using the helper function `createBigImageAndMaskArrays`. This function creates an array of `bigimage` objects from the tumor images and a corresponding array of `bigimage` objects from the tumor mask images. The helper function is attached to the example as a supporting file.

```
[bigTumorImages,bigTumorMasks] = createBigImageAndMaskArrays(trainTumorDataDir,trainTumorMaskDir
```

Select a sample tumor image and mask. Confirm that the spatial world extents of the mask match the extents of the image at the finest resolution level. The spatial world extents are specified by the `XWorldLimits` and `YWorldLimits` properties.

```
idx = 5;
bigTumorImages(idx).SpatialReferencing(1)
```

```
ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 219648]
              ImageSize: [219648 97792]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 97792
    ImageExtentInWorldY: 219648
       XIntrinsicLimits: [0.5000 9.7793e+04]
       YIntrinsicLimits: [0.5000 2.1965e+05]
```

```
bigTumorMasks(idx).SpatialReferencing(1)
```

```
ans =
  imref2d with properties:

           XWorldLimits: [0 97792]
           YWorldLimits: [0 219648]
              ImageSize: [3432 1527]
    PixelExtentInWorldX: 64.0419
    PixelExtentInWorldY: 64
    ImageExtentInWorldX: 97792
    ImageExtentInWorldY: 219648
       XIntrinsicLimits: [0.5000 1.5275e+03]
       YIntrinsicLimits: [0.5000 3.4325e+03]
```

Verify that the mask contains the correct ROIs and spatial referencing. Display the sample image by using the `bigimageshow` function. Get the axes containing the display.

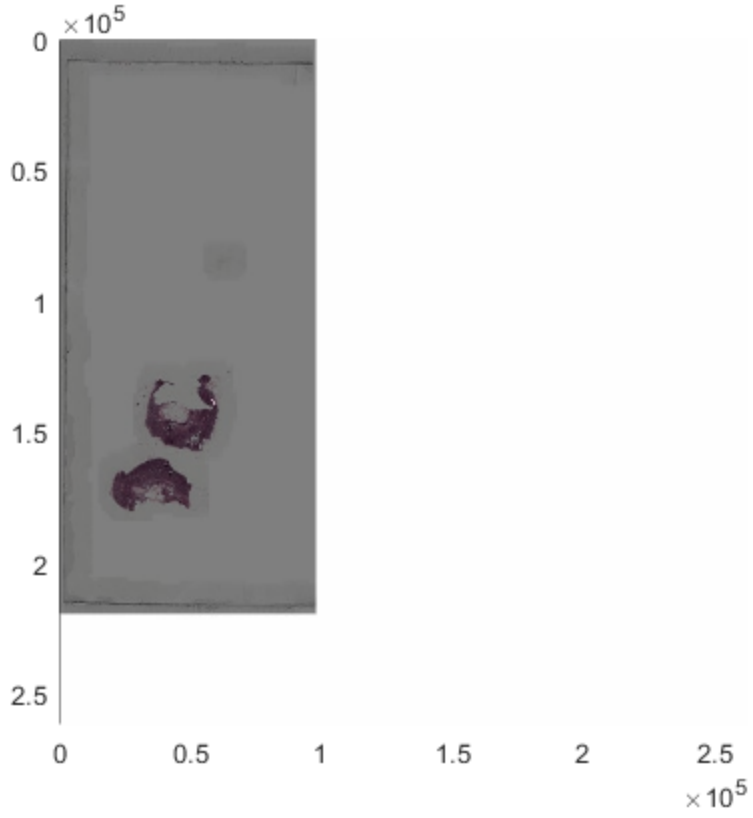
```
figure
hBigTumor = bigimageshow(bigTumorImages(idx));
hTumorAxes = hBigTumor.Parent;
```

Create a new axes on top of the displayed big image. In the new axes, display the corresponding mask image with partial transparency. The mask highlights the regions containing normal tissue.

```
hMaskAxes = axes;
hBigMask = bigimageshow(bigTumorMasks(idx),'Parent',hMaskAxes, ...
    'Interpolation','nearest','AlphaData',0.5);
hMaskAxes.Visible = 'off';
```

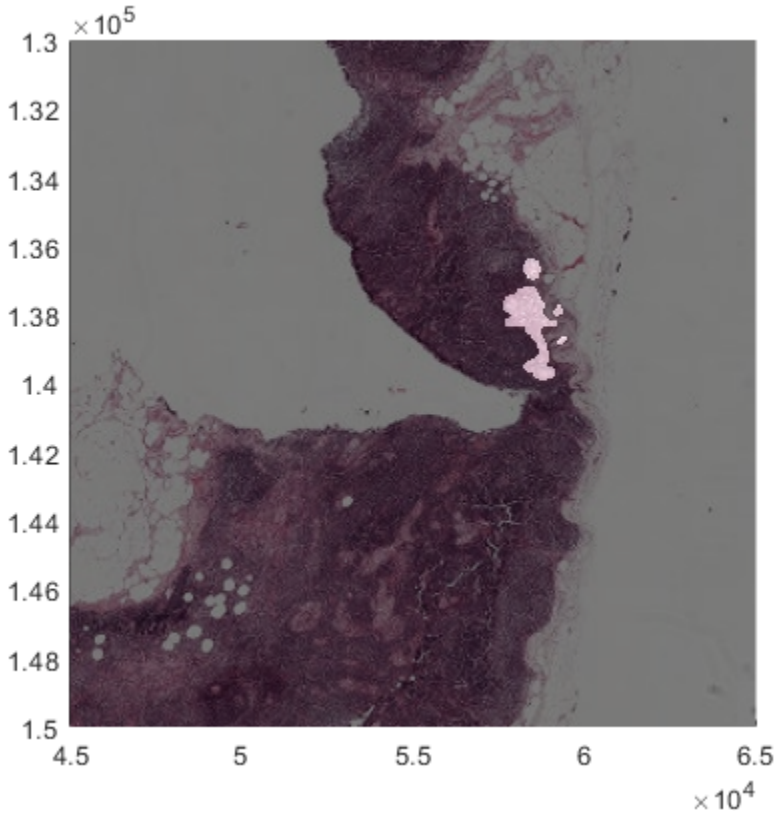
Link the axes of the image with the axes of the mask. When you zoom and pan, both axes are updated identically.

```
linkaxes([hTumorAxes,hMaskAxes]);
```



Zoom in on one part of the image by setting the horizontal and vertical spatial extents. The mask overlaps the tumor tissue correctly.

```
xlim([45000 65000]);
ylim([130000 150000]);
```



Create Big Image Datastore for Training and Validation

To extract patches of training data from the `bigimage` objects, use a `bigimageDatastore`. This datastore reads patches of `bigimage` data at a single resolution level. If you specify a mask, then the datastore reads patches only from within the ROI blocks. Note that each mask block can contain a mixture of ROI and background pixels, which can lead to ambiguity about whether the entire block is ROI or background. To remove the ambiguity, the datastore has an `InclusionThreshold` property. This property enables you to specify the minimum fraction of `true` pixels within a mask block necessary for the block to be considered ROI.

Two main issues with the raw training patches can potentially bias the network. The first issue is color imbalance resulting from nonuniform color staining of the tissue. The second issue is class imbalance resulting from an unequal amount of tumor and normal tissue in the data. To correct these issues, you can preprocess and augment the datastore.

This example shows how to create a `bigimageDatastore` to extract tumor and normal patches for training the network and how to preprocess and augment the datastores to avoid biasing the network.

Create Datastore for Normal Images

Randomly split the normal images and corresponding masks into two sets. The validation set contains two randomly selected images and corresponding masks. The training set contains the remaining images and masks.

```
normalValidationIdx = randi(numNormalFiles,[1 2]);
normalTrainIdx = setdiff(1:numNormalFiles,normalValidationIdx);
```

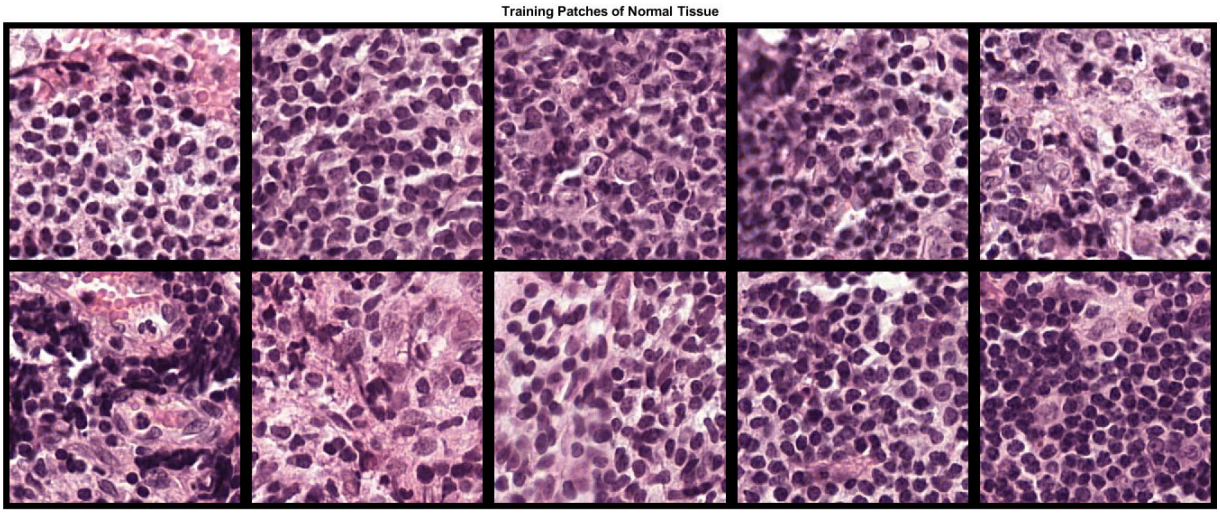
The patch size is small compared to the size of features in the image. By default, a `bigimageDatastore` extracts patches with no overlap and no gap, which generates a huge quantity of training patches. You can reduce the amount of training data by using the `BlockOffsets` property to add a gap between sampled patches. Specify a block offset that is larger than the patch size. Increase the inclusion threshold from the default value of `0.5` so that the network trains on relatively homogenous patches.

The datastores `dsNormalData` and `dsNormalDataValidation` read image patches from normal images at the finest resolution level for training and validation, respectively.

```
patchSize = [299,299,3];

normalStrideFactor = 10;
dsNormalData = bigimageDatastore(bigNormalImages(normalTrainIdx),1, ...
    'InclusionThreshold',0.75,'Mask',bigNormalMasks(normalTrainIdx), ...
    'BlockSize',patchSize(1:2),'BlockOffsets',patchSize(1:2)*normalStrideFactor, ...
    'IncompleteBlocks','exclude');

dsNormalDataValidation = bigimageDatastore(bigNormalImages(normalValidationIdx),1, ...
    'InclusionThreshold',0.75,'Mask',bigNormalMasks(normalValidationIdx), ...
    'BlockSize',patchSize(1:2),'IncompleteBlocks','exclude');
```

Preview patches from the datastore containing normal training images.

```
imagesToPreview = zeros([patchSize 10],'uint8');
for n = 1:10
    im = read(dsNormalData);
    imagesToPreview(:,:,:,n) = im{1};
end
figure
montage(imagesToPreview,'Size',[2 5],'BorderSize',10,'BackgroundColor','k');
title("Training Patches of Normal Tissue")
```



Create Datastore for Tumor Images

Randomly split the tumor images and corresponding masks into two sets. The validation set contains two images and masks. The training set contains the remaining images and masks.

```
tumorValidationIdx = randi(numTumorFiles,[1 2]);
tumorTrainIdx = setdiff(1:numTumorFiles,tumorValidationIdx);
```

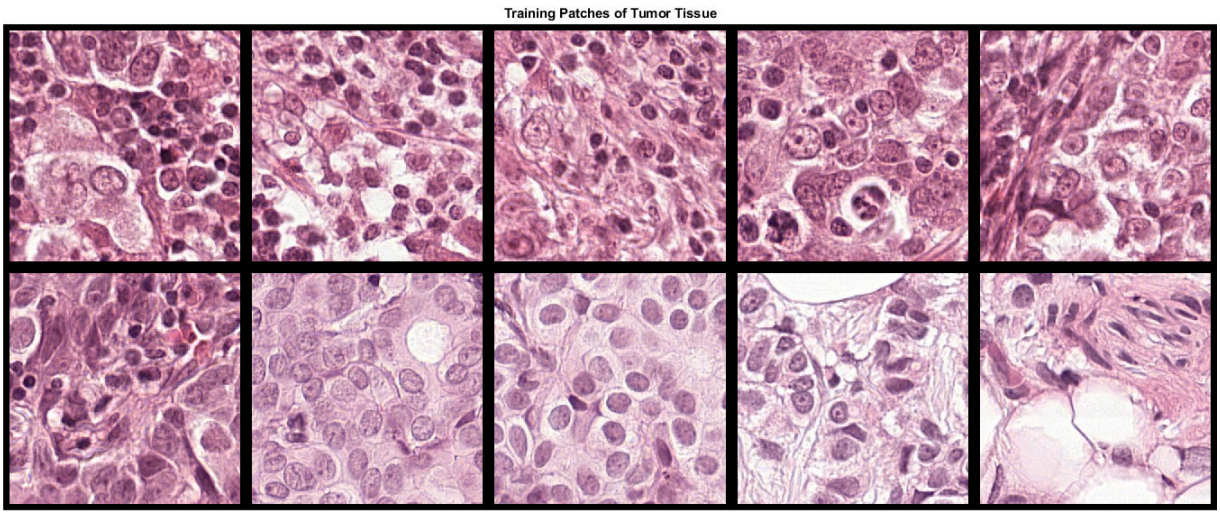
Create a `bigimageDatastore` from the training tumor images and masks. Tumor tissue is more sparse than normal tissue. To increase the sampling density and generate more training patches, specify a smaller block offset. Note that if you want to train using fewer training images, then you might need to increase the size of the training set by decreasing the block offset even further.

The datastores `dsTumorData` and `dsTumorDataValidation` read image patches from tumor images at the finest resolution level for training and validation, respectively.

```
tumorStrideFactor = 4;
dsTumorData = bigimageDatastore(bigTumorImages(tumorTrainIdx),1, ...
    'InclusionThreshold',0.75,'Mask',bigTumorMasks(tumorTrainIdx), ...
    'BlockSize',patchSize(1:2),'BlockOffsets',patchSize(1:2)*tumorStrideFactor, ...
    'IncompleteBlocks','exclude');

dsTumorDataValidation = bigimageDatastore(bigTumorImages(tumorValidationIdx),1, ...
    'InclusionThreshold',0.75,'Mask',bigTumorMasks(tumorValidationIdx), ...
    'BlockSize',patchSize(1:2),'IncompleteBlocks','exclude');
```

Preview patches from the datastore containing tumor training images.

```
imagesToPreview = zeros([patchSize 10],'uint8');
for n = 1:10
    im = read(dsTumorData);
    imagesToPreview(:,:,:,n) = im{1};
end
montage(imagesToPreview,'Size',[2 5],'BorderSize',10,'BackgroundColor','k');
title("Training Patches of Tumor Tissue")
```



Normalize Colors and Augment Training Data

The training images have different color distributions because the data set came from different sources and color staining the tissue does not result in identically stained images. Additional preprocessing is necessary to avoid biasing the network.

To prevent color variability, this example preprocesses the data with standard stain normalization techniques. Apply stain normalization and augmentation by using the `transform` function with custom preprocessing operations specified by the helper function `augmentAndLabel`. This function is attached to the example as a supporting file.

The `augmentAndLabel` function performs these operations:

- Normalize staining by using the `normalizeStaining.m` function [4 on page 9-0]. Stain normalization is performed using Macenko's method, which separates H&E color channels by color deconvolution using a fixed matrix and then recreates the normalized images with individual corrected mixing. The function returns the normalized image as well as the H&E images.
- Add color jitter by using the `jitterColorHSV` function. Color jitter varies the color of each patch by perturbing the image contrast, hue, saturation, and brightness. Color jitter is performed in the HSV color space to avoid unwanted color artifacts in the RGB image.
- Apply random combinations of 90 degree rotations and vertical and horizontal reflection. Randomized affine transformations make the network agnostic to the orientation of the input image data.
- Label the patch as `'normal'` or `'tumor'`.

Each image patch generates five augmented and labeled patches: the stain-normalized patch, the stain-normalized patch with color jitter, the stain-normalized patch with color jitter and random affine transformation, the hematoxylin image with random affine transformation, and the eosin image with random affine transformation.

Create datastores that transform the normal training and validation images and label the generated patches as `'normal'`.

```
dsLabelledNormalData = transform(dsNormalData, ...
    @(x,info)augmentAndLabel(x,info,'normal'),'IncludeInfo',true);
dsLabelledNormalDataValidation = transform(dsNormalDataValidation, ...
    @(x,info)augmentAndLabel(x,info,'normal'),'IncludeInfo',true);
```

Create datastores that transform the tumor training and validation images and label the generated patches as `'tumor'`.

```
dsLabelledTumorData = transform(dsTumorData, ...
    @(x,info)augmentAndLabel(x,info,'tumor'),'IncludeInfo',true);
dsLabelledTumorDataValidation = transform(dsTumorDataValidation, ...
    @(x,info)augmentAndLabel(x,info,'tumor'),'IncludeInfo',true);
```

Balance Tumor and Normal Classes

The amount of cancer tissue in the tumor images is very small compared to the amount of normal tissue. Additional preprocessing is necessary to avoid training the network on class-imbalanced data containing a large amount of normal tissue and a very small amount of tumor tissue.

To prevent class imbalance, this example defines a custom datastore called a `randomSamplingDatastore` that randomly selects normal and tumor training patches in a balanced way. The script to define this custom datastore is attached to the example as a supporting file. For more information, see “Develop Custom Datastore” (MATLAB).

Create the custom `randomSamplingDatastore` from the normal and tumor training datastores. The random sampling datastore `dsTrain` provides mini-batches of training data to the network at each iteration of the epoch.

```
dsTrain = randomSamplingDatastore(dsLabelledTumorData,dsLabelledNormalData);
```

To limit the number of patches used during validation, this example defines a custom datastore called a `validationDatastore` that returns five validation patches from each class. The script to define this custom datastore is attached to the example as a supporting file.

Create the custom `validationDatastore` from the normal and tumor validation datastores.

```
numValidationPatchesPerClass = 5;
dsValidation = validationDatastore(dsLabelledTumorDataValidation, ...
    dsLabelledNormalDataValidation,numValidationPatchesPerClass);
```

Set Up Inception-v3 Network Layers

This example uses the Inception-v3 network, a convolutional neural network that is trained on more than a million images from the ImageNet database [3 on page 9-0]. The network is 48 layers deep and can classify images into 1,000 object categories, such as keyboard, mouse, pencil, and many animals. The network expects an image input size of 299-by-299 with 3 channels.

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;
```

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 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. For more information, see “Train Deep Learning Network to Classify New Images” on page 3-6.

Extract the layer graph from the trained network.

```
lgraph = layerGraph(net);
```

Find the names of the two layers to replace by using the supporting 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)
```

```
learnableLayer =
  FullyConnectedLayer with properties:

          Name: 'predictions'

   Hyperparameters
     InputSize: 2048
    OutputSize: 1000

   Learnable Parameters
       Weights: [1000×2048 single]
          Bias: [1000×1 single]
```

```
  Show all properties


classLayer =
  ClassificationOutputLayer with properties:

           Name: 'ClassificationLayer_predictions'
        Classes: [1000×1 categorical]
     OutputSize: 1000

   Hyperparameters
    LossFunction: 'crossentropyex'
```

The goal of this example is to perform binary segmentation between two classes, tumor and nontumor regions. Create a new fully connected layer for two classes. Replace the original 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 original 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 the `rmsprop` optimization solver. This solver automatically adjusts the learning rate and momentum for faster convergence. Specify other hyperparameter settings 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. If you have Parallel Computing Toolbox™ and the hardware resources available for multi-GPU or parallel training, then accelerate training by specifying the `ExecutionEnvironment` name-value pair argument as `'multi-gpu'` or `'parallel'`, respectively.

```
checkpointsDir = fullfile(trainingImageDir,'checkpoints');
if ~exist(checkpointsDir,'dir')
    mkdir(checkpointsDir);
end

options = trainingOptions('rmsprop', ...
    'InitialLearnRate',1e-5, ...
    'SquaredGradientDecayFactor',0.99, ...
    'MaxEpochs',3, ...
    'MiniBatchSize',32, ...
    'Plots','training-progress', ...
    'CheckpointPath',checkpointsDir, ...
    'ValidationData',dsValidation, ...
    'ExecutionEnvironment','auto', ...
    'Shuffle','every-epoch');
```

Download and Preprocess Test Data

The Camelyon16 test data set consists of 130 WSIs. These images have both normal and tumor tissue. This example uses two test images from the Camelyon16 test data. The size of each file is approximately 2 GB.

Create a directory to store the test data.

```
testingImageDir = fullfile(tempdir,'Camelyon16','testing');
if ~exist(testingImageDir,'dir')
    mkdir(testingImageDir);
    mkdir(fullfile(testingImageDir,'images'));
    mkdir(fullfile(testingImageDir,'lesion_annotations'));
end

testDataDir = fullfile(testingImageDir,'images');
testTumorAnnotationDir = fullfile(testingImageDir,'lesion_annotations');
```

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 `testTumorAnnotationDir` variable.
- Open the "images" directory. Download the first two files, "test_001.tif" and "test_002.tif". Move the files to the directory specified by the `testDataDir` variable.

Specify the number of test images.

```
numTestFiles = 2;
```

Create Masks for Test Images

The test images contain a mix of normal and tumor images. To reduce the amount of computation during classification, define the ROIs by creating masks.

Specify a resolution level to use for creating the mask. This example uses resolution level 7, which is coarse and fits in memory.

```
resolutionLevel = 7;
```

Create masks for regions containing tissue. You can use the helper function `createMaskForNormalTissue` to create masks using color thresholding. This helper function is attached to the example as a supporting file. For more information about this helper function, see Create Masks for Normal Images on page 9-0 .

```
testTissueMaskDir = fullfile(testDataDir,['test_tissuemask_level' num2str(resolutionLevel)]);
createMaskForNormalTissue(testDataDir,testTissueMaskDir,resolutionLevel);
```

Create masks for images that contain tumor tissue. Skip images that do not contain tumor tissue. You can use the helper function `createMaskForTumorTissue` to create masks using ROI objects. This helper function is attached to the example as a supporting file. For more information about this helper function, see Create Masks for Tumor Images on page 9-0 .

```
testTumorMaskDir = fullfile(testDataDir,['test_tumormask_level' num2str(resolutionLevel)]);
createMaskForTumorTissue(testDataDir,testTumorAnnotationDir,testTumorMaskDir,resolutionLevel);
```

Train Network

After configuring the training options and the data source, train the Inception-v3 network by using the `trainNetwork` function. By default, this example downloads a pretrained version of the Inception-v3 network for this data set by using the helper function `downloadTrainedCamelyonNet`. 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.

Note: Training takes 20 hours on an NVIDIA™ Titan X and can take even longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    trainedNet = trainNetwork(dsTrain,lgraph,options);
    save(['trainedCamelyonNet-' datestr(now, 'dd-mmm-yyyy-HH-MM-SS') '.mat'],'trainedNet');
else
    trainedCamelyonNet_url = 'https://www.mathworks.com/supportfiles/vision/data/trainedCamelyonN
    netDir = fullfile(tempdir,'Camelyon16');
    downloadTrainedCamelyonNet(trainedCamelyonNet_url,netDir);
    load(fullfile(netDir,'trainedCamelyonNet.mat'));
end
```

```
Downloading pretrained Inception-v3 network for Cameylon16 data.
This can take several minutes to download...
Done.
```

Classify Test Data and Create Heatmaps

A GPU is highly recommended for classification of the test images (requires Parallel Computing Toolbox™).

Each test image has two masks, one indicating normal tissue and one indicating tumor tissue. Create `bigimage` objects to manage the test data and masks by using the helper function `createBigImageAndMaskArrays`. The helper function is attached to the example as a supporting file.

```
[bigTestImages,bigTestTissueMasks] = createBigImageAndMaskArrays(testDataDir,testTissueMaskDir);
[~,bigTestTumorMasks] = createBigImageAndMaskArrays(testDataDir,testTumorMaskDir);
```

Use the trained Inception-v3 network to identify tumor patches in the test images, `bigTestImages`. Classify the test images on a block-by-block basis by using the `apply` function with a custom processing pipeline specified by the helper function `createHeatMap`. This helper function is attached to the example as a supporting file. To reduce the amount of computation required, specify the tissue mask `bigTestTissueMask` so that the `apply` function processes only patches that contain tissue. If you have Parallel Computing Toolbox™ and a GPU, then you can evaluate blocks in parallel by specifying the `'UseParallel'` name-value pair argument as `true`.

The `createHeatMap` function performs these operations on each tissue block:

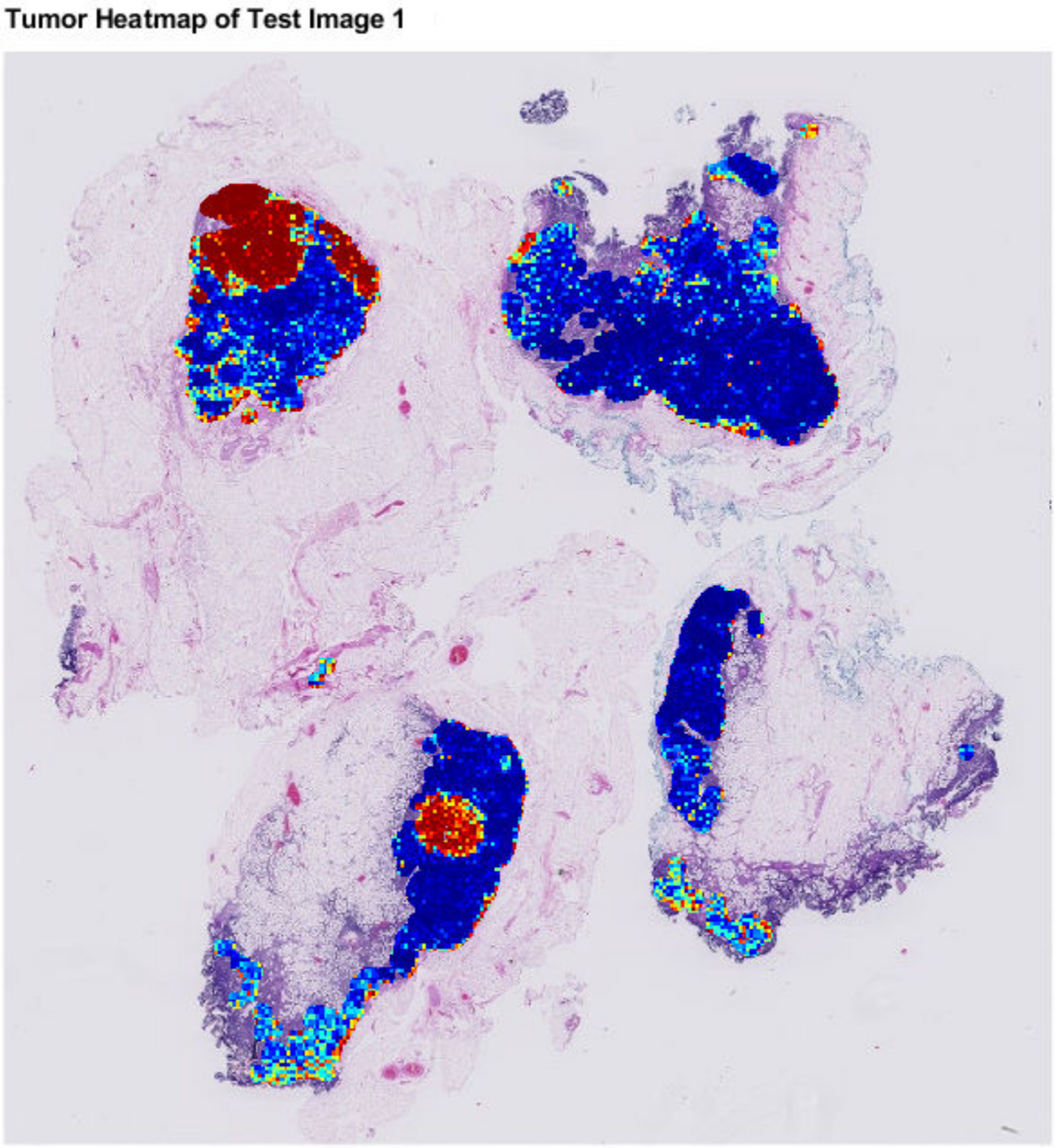
- Calculate the tumor probability score by using the `predict` function.
- Create a heatmap image patch with pixel values equal to the tumor probability score.

The `apply` function stitches the heatmap of each block into a single heatmap for the test image. The heatmap shows where the network detects regions containing tumors.

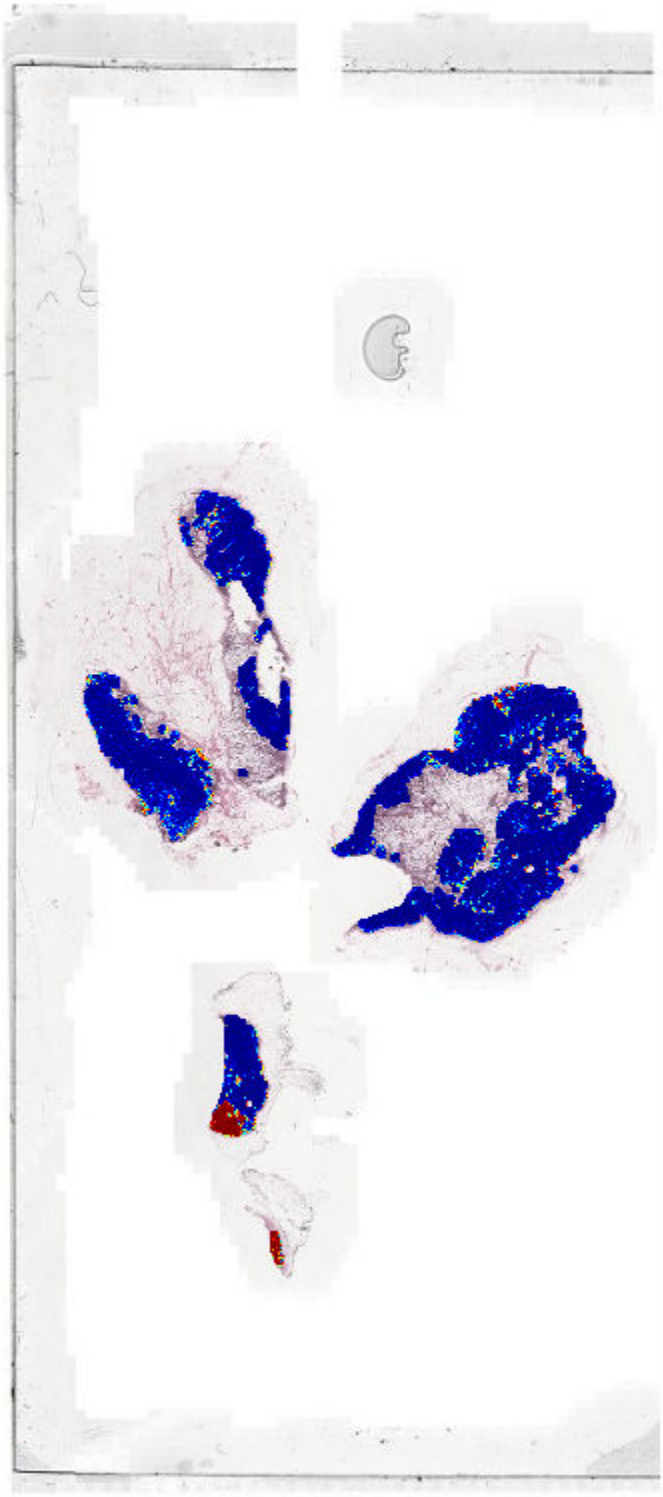
To visualize the heatmap, overlay the heatmap on the original image and set the transparency `'AlphaData'` property as the tissue mask. The overlay shows how well the tumor is localized in the

image. Regions with a high probability of being tumors are displayed with red pixels. Regions with a low probability of being tumors are displayed as blue pixels.

```
patchSize = [299,299,3];

for idx = 1:numTestFiles
    bigTestHeatMaps(idx) = apply(bigTestImages(idx),1,@(x)createHeatMap(x,trainedNet), ...
        'Mask',bigTestTissueMasks(idx),'InclusionThreshold',0, ...
        'BlockSize',patchSize(1:2),'UseParallel',false);

    figure
    hBigTest = bigimageshow(bigTestImages(idx));
    hTestAxes = hBigTest.Parent;
    hTestAxes.Visible = 'off';

    hMaskAxes = axes;
    hBigMask = bigimageshow(bigTestHeatMaps(idx),'Parent',hMaskAxes, ...
        "Interpolation","nearest","AlphaData",bigTestTissueMasks(idx));
    colormap(jet(255));
    hMaskAxes.Visible = 'off';

    linkaxes([hTestAxes,hMaskAxes]);
    title(['Tumor Heatmap of Test Image ',num2str(idx)])
end
```


Tumor Heatmap of Test Image 1

Tumor Heatmap of Test Image 2



References

[1] Ehteshami B. B., et al. "Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer." *Journal of the American Medical Association*. Vol. 318, No. 22, 2017, pp. 2199–2210. doi:10.1001/jama.2017.14585

[2] Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. "Rethinking the Inception Architecture for Computer Vision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–2826. Las Vegas, NV: IEEE, 2016.

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

[4] Macenko, M., et al. "A Method for Normalizing Histology Slides for Quantitative Analysis." In *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 1107–1110. Boston, MA: IEEE, 2009.

[5] xml2struct https://www.mathworks.com/matlabcentral/fileexchange/28518-xml2struct.

See Also

`bigimage` | `bigimageDatastore` | `bigimageshow` | `trainNetwork` | `trainingOptions` | `transform`

More About

- “Datastores for Deep Learning” on page 16-2
- “List of Deep Learning Layers” on page 1-23

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 9-0] 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 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. 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`. This function is attached to the example as a supporting file.

```
numClasses = 32;
[classes,labelIDs] = defineCamVid32ClassesAndPixelLabelIDs;
```

Create a `pixelLabelDatastore` 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` function, then display the pixel label image and ground truth image in a montage.

```
im = preview(imds);
px = preview(pxds);
px = label2rgb(px);
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 and validation 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.

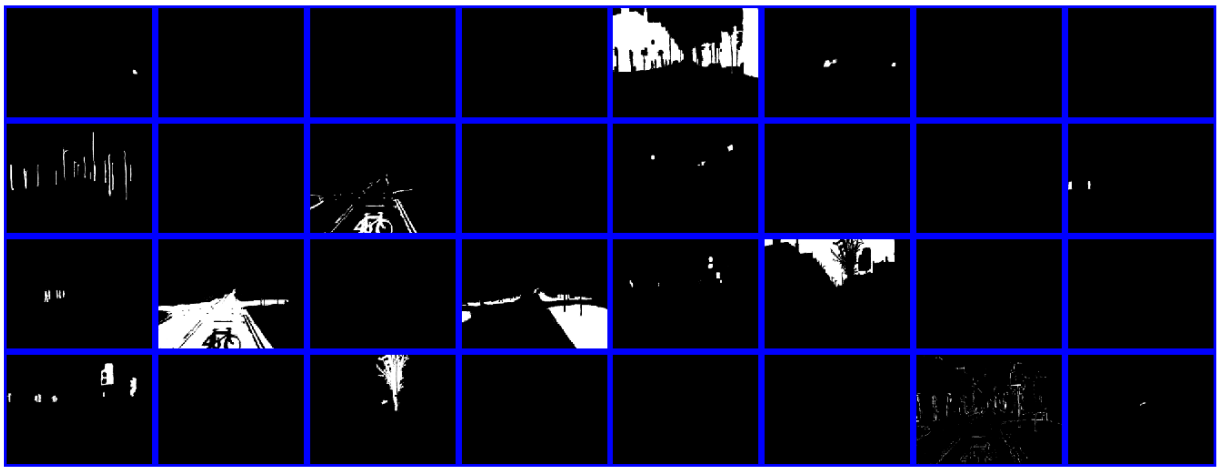
The `preprocessCamVidForPix2PixHD` function performs these operations:

1. Scale the ground truth data to the range [-1,1]. This range matches the range of the `tanhLayer` in the generator network.
2. Resize the image and labels to the output size of the network, 576-by-768 pixels, using nearest neighbor downsampling.
3. Convert the single channel segmentation map to a 32-channel one-hot encoded segmentation map. This map represents each class using a unique binary mask, in which pixels of the class are `1` and all other pixels are `0`. The map concatenates the masks of each class along the depth direction.
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.

```
map = preview(dsTrain);
montage(map{1},'Size',[4 8],'Bordersize',5,'BackgroundColor','b')
```



Create Generator Network

Define a 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.

```
inputSize = [imageSize numClasses];
```

Create the initial subnetwork layers. Specify the filter size and number of filters in first convolutional layer of the generator. `reflectionPad2d` is a custom layer implemented specifically for this example. This layer is attached to the example as a supporting file.

```
filterSize = [7,7];
numFiltersGenerator = 64;

layers = [ ...
    imageInputLayer(inputSize,'Normalization','none','Name','inputLayer') ...
    reflectionPad2dLayer(3,'iPad') ...
    convolution2dLayer(filterSize,numFiltersGenerator,'Name','iConv', ...
        "BiasLearnRateFactor",0,"WeightsInitializer",'he',"BiasInitializer",'zeros') ...
    batchNormalizationLayer('Name','iBn', ...
        'Scale',ones([1 1 numFiltersGenerator]),'ScaleLearnRateFactor',0, ...
        'Offset',zeros([1 1 numFiltersGenerator]),'OffsetLearnRateFactor',0)...
    reluLayer('Name','iRelu')
    ];
```

Create the layer graph.

```
generator = layerGraph(layers);
```

Create layers of the downsampling subnetwork. Specify the filter size for the downsampling layers. Use four downsampling layers.

```
filterSize = [3 3];
numDownSamplingLayers = 4;

for idx = 1:numDownSamplingLayers
    % Compute the number of filters in the next convolutional layer
    multiplier = 2^(idx - 1);
    numFilters = numFiltersGenerator*multiplier*2;

    layers = [ ...
        reflectionPad2dLayer(1,strcat("dPad",int2str(idx))) ...
        convolution2dLayer(filterSize,numFilters,"Name",strcat("dConv",int2str(idx)), ...
            "BiasLearnRateFactor",0,"WeightsInitializer",'he',"Stride",2)
        batchNormalizationLayer('Name',strcat("dBn",int2str(idx)), ...
            'OffsetLearnRateFactor',0,'ScaleLearnRateFactor',0) ...
        reluLayer('Name',strcat('dRelu',int2str(idx)))
    ];

    % Add the layers to the layer graph
    lg = addLayers(generator,layers);
    generator = connectLayers(lg,generator.Layers(end).Name,layers(1).Name);
end
```

Create layers of the residual subnetwork. Specify nine residual blocks in the generator.

```
numFiltersResidual = numFilters;
numResidualBlocks = 9;
for idx = 1:numResidualBlocks
    % Name of the layer which will act as the source of the residual connection
    res = generator.Layers(end).Name;

    % Specify the layer names
    s = int2str(idx);
    convLayer1Name = strcat("rConv",s,"_1");
    convLayer2Name = strcat("rConv",s,"_2");
    bnLayer1Name = strcat("rBn",s,"_1");
    bnLayer2Name = strcat("rBn",s,"_2");
    pad1Name = strcat("rPad",s,"_1");
    pad2Name = strcat("rPad",s,"_2");

    layers = [...
        reflectionPad2dLayer(1,pad1Name) ...
        convolution2dLayer(filterSize,numFiltersResidual,"Name",convLayer1Name, ...
            "BiasLearnRateFactor",0,"WeightsInitializer",'he') ...
        batchNormalizationLayer('Name',bnLayer1Name, ...
            'OffsetLearnRateFactor',0,'ScaleLearnRateFactor',0) ...
        reflectionPad2dLayer(1,pad2Name) ...
        convolution2dLayer(filterSize,numFiltersResidual,"Name",convLayer2Name, ...
            "BiasLearnRateFactor",0,"WeightsInitializer",'he') ...
        batchNormalizationLayer('Name',bnLayer2Name, ...
            'OffsetLearnRateFactor',0,'ScaleLearnRateFactor',0) ...
        reluLayer("Name",strcat("rRelu",s))...
        additionLayer(2,'Name',strcat("rAdd",s))
        ];

    % Add the layers to the layer graph
```

```
    lg = addLayers(generator,layers);
    generator = connectLayers(lg,generator.Layers(end).Name,layers(1).Name);

    % Link the residual connection.
    generator = connectLayers(generator,res,strcat("rAdd",s,"/in2"));
end
```

Create layers of the upsampling subnetwork.

```
for idx=1:numDownSamplingLayers
    s = int2str(idx);
    % Compute the number of filters in the next convolutional layer
    multiplier = 2^(numDownSamplingLayers + 1 - idx);
    numFilters = (numFiltersGenerator * multiplier) / 2;

    layers = [ ...
        transposedConv2dLayer(filterSize,numFilters,"Name",strcat("uConv",s), ...
            "Stride",2,"BiasLearnRateFactor",0,"WeightsInitializer",'he',"Cropping","Same") ...
        batchNormalizationLayer('Name',strcat("uBn",s), ...
            'OffsetLearnRateFactor',0,'ScaleLearnRateFactor',0) ...
        reluLayer('Name',strcat('uRelu',s));
    ];

    % Add the layers to the layer graph
    lg = addLayers(generator,layers);
    generator = connectLayers(lg,generator.Layers(end).Name,layers(1).Name);
end
```

Create the final subnetwork layers. Specify the filter size and number of filters of the final convolutional layer of the generator.

```
filterSize = [7,7];
numFilters = 3;

layers = [ ...
    reflectionPad2dLayer(3,'fPad') ...
    convolution2dLayer(filterSize,numFilters,'Name','fConv', ...
        "WeightsInitializer",'he',"BiasInitializer",'narrow-normal') ...
    tanhLayer('Name','ftanh')];
```

Add the subnetwork layers to the layer graph.

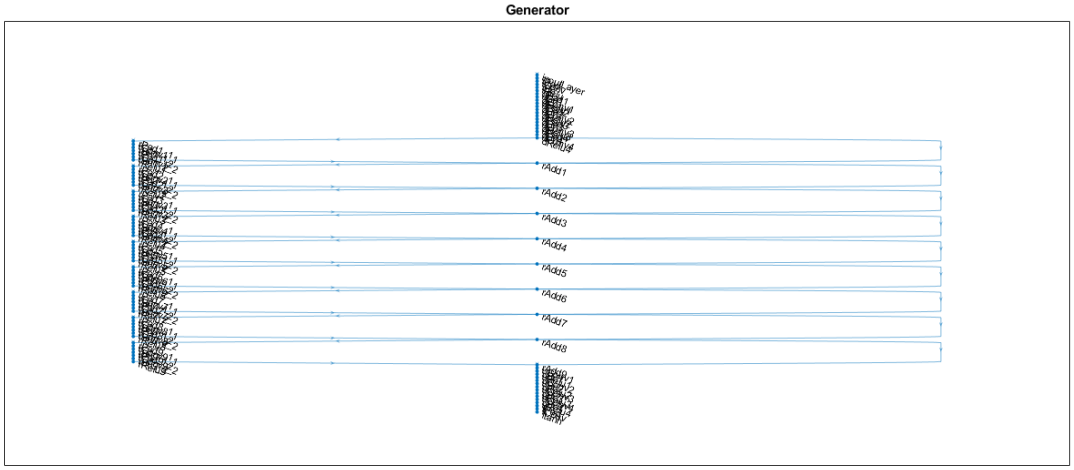
```
lg = addLayers(generator,layers);
lgraphGenerator = connectLayers(lg,generator.Layers(end).Name,layers(1).Name);
```

To train the network with a custom training loop and to enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetGenerator = dlnetwork(lgraphGenerator);
```

Visualize the generator network in a plot.

```
plot(lgraphGenerator)
title("Generator")
```



Create Discriminator Network

Define a discriminator network that classifies an input image as either real (1) or fake (0).

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;
inputSize = [imageSize numChannelsDiscriminator];
```

Specify the number of filters in the first convolutional layer of the discriminator.

```
numFilters = 64;
```

Specify the filter size of the first convolutional layer of the discriminator.

```
filterSize  = [4 4];
```

Define the discriminator layers.

```
discriminator = [
    imageInputLayer(inputSize,"Name","inputLayer","Normalization","none")
    convolution2dLayer(filterSize,numFilters, ...
        "Name","iConv","BiasInitializer","narrow-normal", ...
        "Padding",2,"Stride",2,"WeightsInitializer","he")
    leakyReluLayer(0.2,"Name","lrelu1")
    convolution2dLayer(filterSize,numFilters*2,"Name","dConv1", ...
        "BiasInitializer","narrow-normal", ...
        "Padding",2,"Stride",2,"WeightsInitializer","he")
    batchNormalizationLayer("Name","dBn1", ...
        "OffsetLearnRateFactor",0,"ScaleLearnRateFactor",0)
    leakyReluLayer(0.2,"Name","lrelu2")
    convolution2dLayer(filterSize,numFilters*4,"Name","dConv2", ...
        "BiasInitializer","narrow-normal", ...
        "Padding",2,"Stride",2,"WeightsInitializer","he");
    batchNormalizationLayer("Name","dBn2", ...
```

```
    "OffsetLearnRateFactor",0,"ScaleLearnRateFactor",0)
leakyReluLayer(0.2,"Name","lrelu3")
convolution2dLayer(filterSize,numFilters*8,"Name","dConv3", ...
    "BiasInitializer","narrow-normal", ...
    "Padding",2,"WeightsInitializer","he")
batchNormalizationLayer("Name","dBn3", ...
    "OffsetLearnRateFactor",0,"ScaleLearnRateFactor",0)
leakyReluLayer(0.2,"Name","lrelu4")
convolution2dLayer(filterSize,1,"Name","fConv", ...
    "BiasInitializer","narrow-normal", ...
    "Padding",2,"WeightsInitializer","he")
];
```

Create the layer graph.

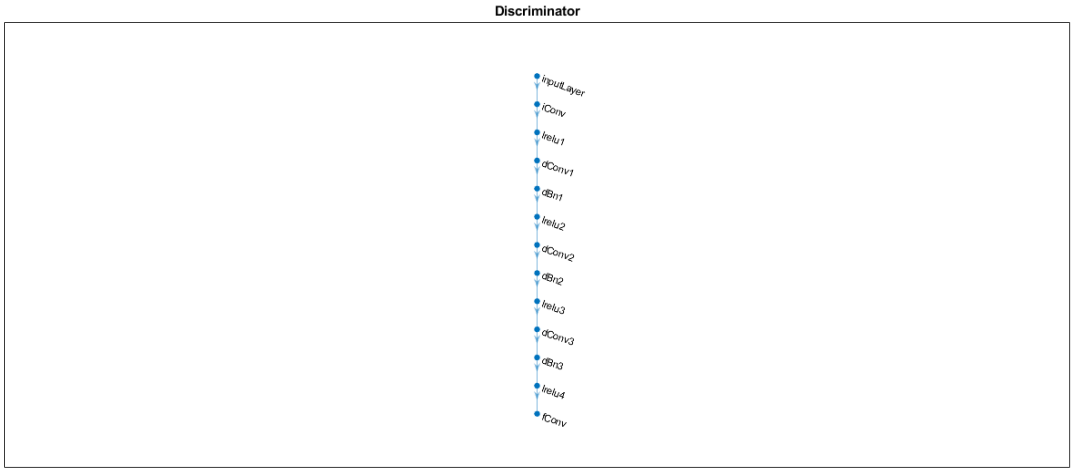
```
lgraphDiscriminator = layerGraph(discriminator);
```

To train the network with a custom training loop and to enable automatic differentiation, convert the layer graph to a `dlnetwork` object.

```
dlnetDiscriminator = dlnetwork(lgraphDiscriminator);
```

Visualize the discriminator network in a plot.

```
plot(lgraphDiscriminator)
title("Discriminator")
```



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 9-0 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 9-0 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 9-0 section of this example

$$lossFeatureMatching = \sum_{i=1}^{T} \left|\left| Y_{real} - \widehat{Y}_{generated} \right|\right|_1$$

- 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 9-0 section of this example. The feature extraction network is created in Load Feature Extraction Network on page 9-0 .

$$lossVgg = \sum_{i=1}^{T} \left|\left| Y_{VggReal} - \widehat{Y}_{VggGenerated} \right|\right|_1$$

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

$$lossDiscriminator = (1 - Y_{real})^2 + (0 - \widehat{Y}_{generated})^2$$

The discriminator loss is implemented using part of the `pix2pixhdAdversarialLoss` helper function defined in the Supporting Functions on page 9-0 section of this example. Note that adversarial loss for the discriminator is computed for two different scales.

Load Feature Extraction Network

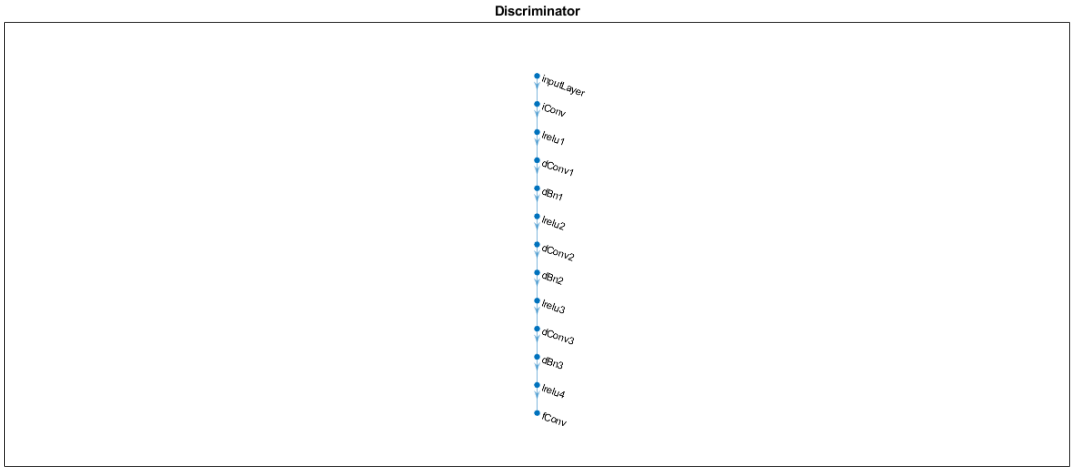
This example modifies a pretrained VGG-19 deep neural network to extract the features of the real and generated images at various layers. These multilayer features are used to compute the perceptual loss of the generator.

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.

```
net = vgg19;
```

Use the `analyzeNetwork` function to display an interactive visualization of the network architecture.

```
analyzeNetwork(net)
```



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.

```
netLg = layerGraph(net.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");
netLg = replaceLayer(netLg,"input",inp);
netVGG = dlnetwork(netLg);
```

Specify Training Options

Specify the options for Adam optimization. Train for 60 epochs. Specify identical options for the generator and discriminator networks.

- Start with 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.

```
numEpochs = 60;
learningRate = 0.0002;
trailingAvgGenerator = [];
trailingAvgSqGenerator = [];
trailingAvgDiscriminator = [];
trailingAvgSqDiscriminator = [];
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;
```

Train Network

Train the model by using the `trainNetwork` function in a custom training loop. Loop over the training data and update the network parameters each iteration. This example uses a learning rate scheduling policy where the learning rate is constant for the first half of the number of epochs and then decays linearly to zero for remaining half of the number of epochs.

For each iteration:

- Convert both segmentation map and the ground truth image data to `dlarray` objects with the underlying type `single` and spatial dimension labels `SSCB` (spatial, spatial, channel, batch).
- For GPU training, convert the data to `gpuArray` objects.
- 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.

By default, this example downloads a pretrained version of the Pix2PixHD 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.

Train on a GPU if possible. Use of a CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training Pix2PixHD network (requires Parallel Computing Toolbox™). Training takes about five and a half days on a 12 GB NVIDIA™ Titan X and can take even longer depending on your GPU hardware. Training the network requires at least 14 MB of memory. If you want to train the network with less memory, try reducing the size of the input images to 480-by-640 pixels by specifying the `imageSize` variable in the Preprocess Training Data on page 9-0 section as [480 640] and rerunning the example.

```
doTraining = false;
if doTraining
    fig = figure;
    lossPlotter = configureTrainingProgressPlotter(fig);
    iteration = 0;

    % Loop over epochs.
    for epoch = 1:numEpochs

        % Reset and shuffle the datastore.
        reset(dsTrain);
        dsTrain = shuffle(dsTrain);

        % Loop over each image.
        while hasdata(dsTrain)
            iteration = iteration + 1;
```

```
            % Read data.
            data = read(dsTrain);
            segMap = data{1,1};
            realImage = data{1,2};

            % Convert data to dlarray specify the dimension labels 'SSCB'
            % (spatial, spatial, channel, batch).
            dlRealImage = dlarray(realImage,'SSCB');
            dlInputSegMap = dlarray(segMap,'SSCB');

            % If training on a GPU, then convert data to gpuArray.
            if canUseGPU
                dlRealImage = gpuArray(dlRealImage);
                dlInputSegMap = gpuArray(dlInputSegMap);
            end

            % Evaluate the model gradients and the generator state using
            % dlfeval and the GANLoss function listed at the end of the
            % example.
            [gradParamsG,gradParamsD,losses] = ...
                dlfeval(@modelGradients,dlInputSegMap,dlRealImage,dlnetGenerator,dlnetDiscrimina

            % Update the generator parameters.
            [dlnetGenerator,trailingAvgGenerator,trailingAvgSqGenerator] = ...
                adamupdate(dlnetGenerator,gradParamsG, ...
                trailingAvgGenerator,trailingAvgSqGenerator,iteration, ...
                learningRate,gradientDecayFactor,squaredGradientDecayFactor);

            % Update the discriminator parameters.
            [dlnetDiscriminator,trailingAvgDiscriminator,trailingAvgSqDiscriminator] = ...
                adamupdate(dlnetDiscriminator,gradParamsD, ...
                trailingAvgDiscriminator,trailingAvgSqDiscriminator, iteration, ...
                learningRate,gradientDecayFactor,squaredGradientDecayFactor);

            % Plot and display various losses.
            lossPlotter = updateTrainingProgressPlotter(lossPlotter,iteration,losses);
        end
    end
    save('trainedPix2PixHDNet.mat','dlnetGenerator');
else
    trainedPix2PixHDNet_url = 'https://www.mathworks.com/supportfiles/vision/data/trainedPix2Pix
    netDir = fullfile(tempdir,'CamVid');
    downloadTrainedPix2PixHDNet(trainedPix2PixHDNet_url,netDir);
    load(fullfile(netDir,'trainedPix2PixHDNet.mat'));
end
```

Generate Images from Test Data

Get a ground truth scene image from the test data. Resize the image.

```
gtImage = read(imdsTest);
gtImage = imresize(gtImage,imageSize,"nearest");
```

Get the corresponding pixel label image from the test data. Resize the pixel label image.

```
segMap = read(pxdsTest);
segMap = segMap{1};
segMap = imresize(single(segMap),imageSize,"nearest");
```

For display, convert the labels from categorical labels to RGB colors by using the `label2rgb` function.

```
coloredSegMap = label2rgb(segMap);
```

Convert the pixel label image to a multichannel one-hot segmentation map by using the helper function `oneHotEncodeSegMap`. The helper function is attached to the example as a supporting file.

```
segMap1Hot = oneHotEncodeSegMap(segMap,numClasses);
```

Create a `dlarray` object to input data to the generator. For GPU inference, convert data to a `gpuArray` object.

```
dlSegMap = dlarray(segMap1Hot,'SSCB');

if canUseGPU
    dlSegMap = gpuArray(dlSegMap);
end
```

Generate a scene image from the generator and one-hot segmentation map. Use the `forward` function because the generator network has non-learnable batch normalization layers. Using the `predict` function yields incorrect results.

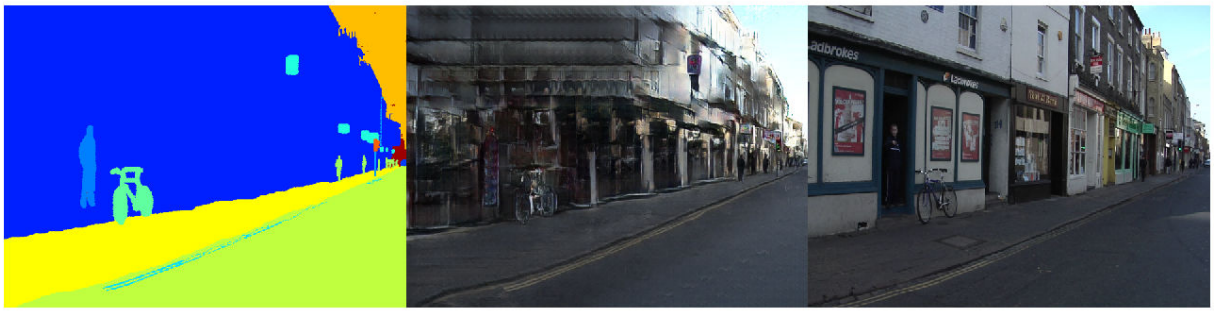
```
[dlGeneratedImage,~] = forward(dlnetGenerator,dlSegMap);
generatedImage = extractdata(gather(dlGeneratedImage));
```

The final layer of the generator network is a tanh layer, which produces activations in the range [-1, 1]. For display, rescale the activations to the range [0, 1].

```
generatedImage = rescale(generatedImage);
```

Display the RGB pixel label image, generated scene image, and ground truth scene image in a montage.

```
montage({coloredSegMap generatedImage gtImage},'Size',[1 3])
```



Generate Images from Custom Pixel Label Images

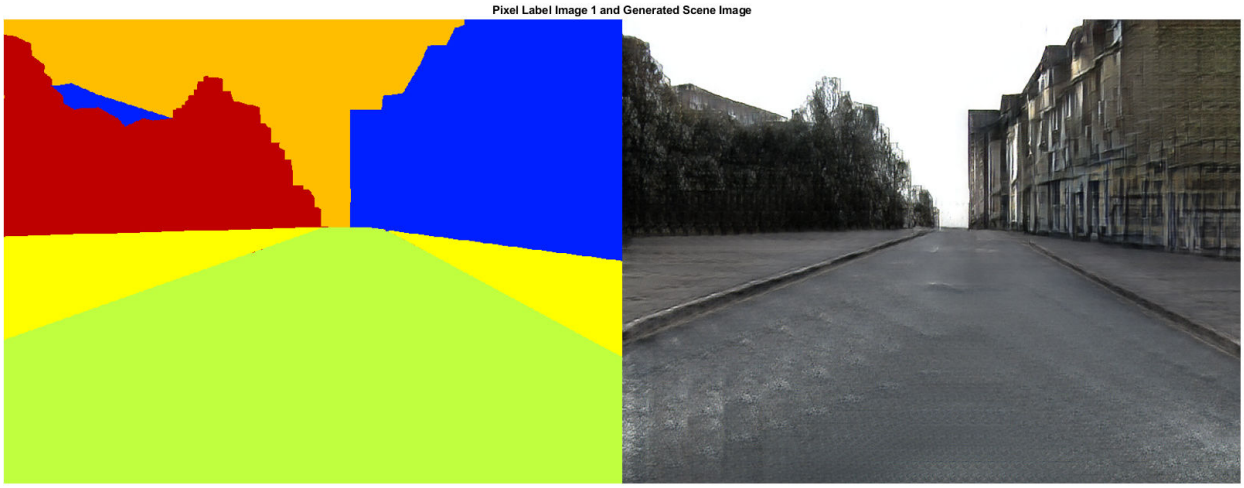
To evaluate how well the network generalizes to pixel label images outside the CamVid data set, generate scene images from custom pixel label images. This example uses five pixel label images that were created using the Image Labeler app. The pixel label images are attached to the example as supporting files.

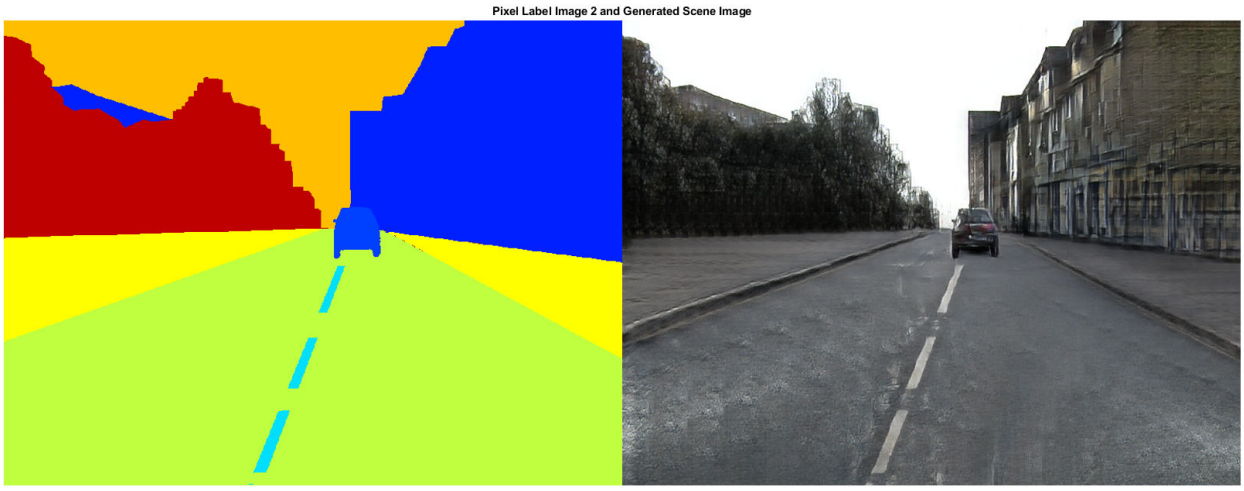
Create a pixel label datastore to read and process the pixel label images in the current example directory.

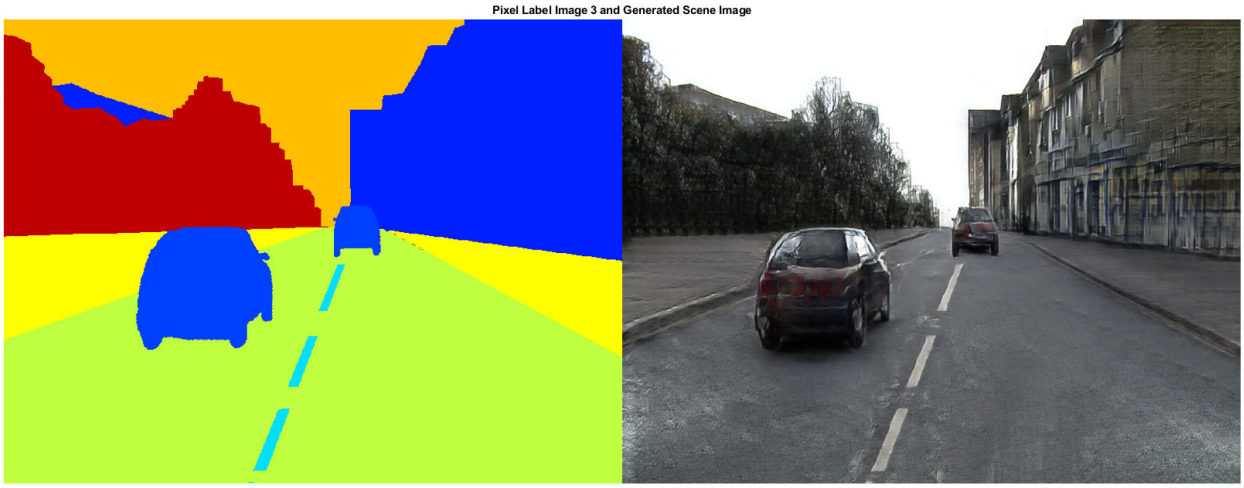
```
cpxds = pixelLabelDatastore(pwd,classes,labelIDs);
```

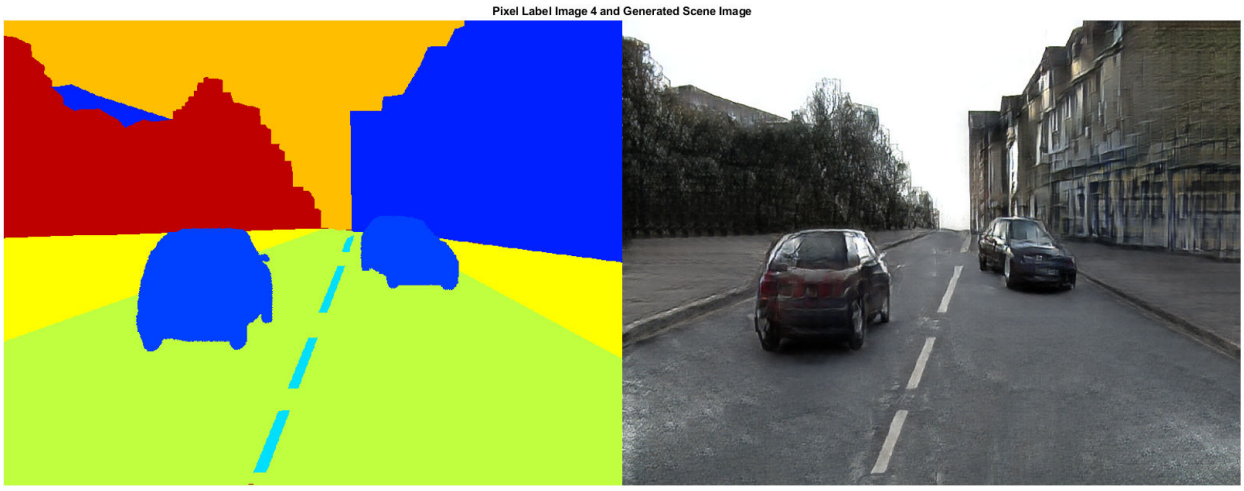
For each test pixel label image, generate and display a scene image.

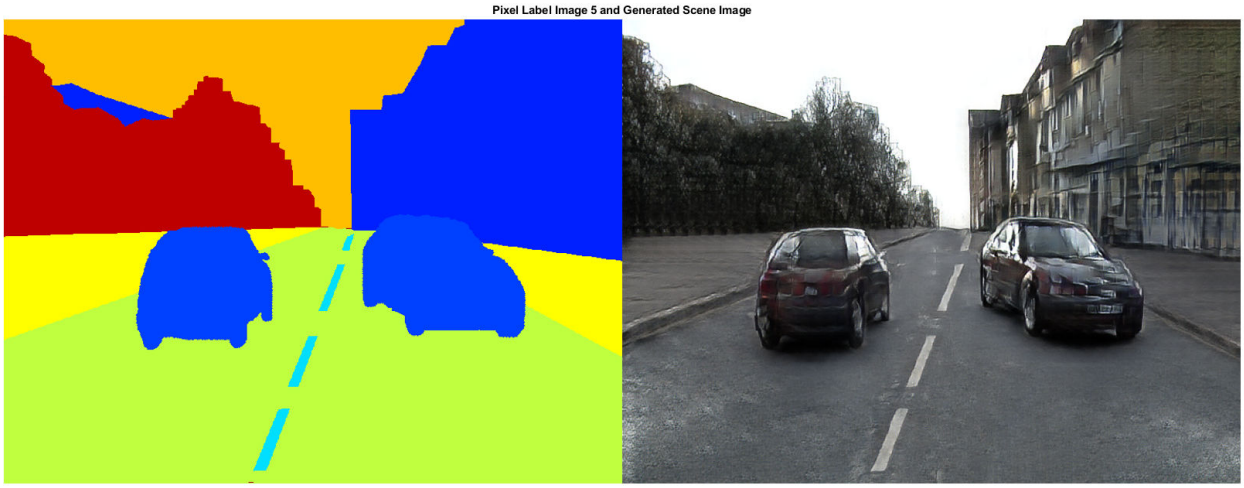
```
for idx = 1:length(cpxds.Files)

    segMap = readimage(cpxds,idx);
    segMap = imresize(single(segMap),imageSize,"nearest");

    % Convert the pixel label image to a multichannel one-hot image by using
    % the helper function oneHotEncodeSegMap. The helper function is attached
    % to the example as a supporting file.
    segMap1Hot = oneHotEncodeSegMap(segMap,numClasses);

    % Convert data to dlarray specify the dimension labels 'SSCB'
    % (spatial, spatial, channel, batch).
    dlSegMap = dlarray(segMap1Hot,'SSCB');

    % If training on a GPU, then convert data to gpuArray.
    if canUseGPU
        dlSegMap = gpuArray(dlSegMap);
    end

    % Generate the image.
    [generated,~] = forward(dlnetGenerator,dlSegMap);
    generated = extractdata(gather(generated));

    % Since the final layer in the generator is a tanh layer, rescale
    % the image to the range [0, 1].
    generated = rescale(generated);

    % Display the pixel label image and generated scene image in a montage.
    figure
    montage({label2rgb(segMap) generated})
    title(['Pixel Label Image ',num2str(idx),' and Generated Scene Image'])

end
```


Pixel Label Image 1 and Generated Scene Image


Pixel Label Image 2 and Generated Scene Image


Pixel Label Image 3 and Generated Scene Image


Pixel Label Image 4 and Generated Scene Image



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,gradParamsD,lossGGAN,lossGFM,lossGVGG,lossD] = modelGradients(inputSegMap,

    % 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.
    [DLossScale1,GLossScale1,realPredScale1D,fakePredScale1G] = pix2pixHDAdverserialLoss(inpDisc

    % Scale the generated image, the real image, and the input semantic map to
    % half size using average pooling.
    resizedRealImage = avgpool(realImage,[3 3],"Stride",[2 2],"Padding",[1 1]);
    resizedGeneratedImage = avgpool(generatedImage,[3 3],"Stride",[2 2],"Padding",[1 1]);
    resizedinputSegMap = avgpool(inputSegMap,[3 3],"Stride",[2 2],"Padding",[1 1]);

    % 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
    [DLossScale2,GLossScale2,realPredScale2D,fakePredScale2G] = pix2pixHDAdverserialLoss(inpDisc
```

```
    % Compute the feature matching loss for scale 1
    FMLossScale1 = pix2pixHDFeatureMatchingLoss(realPredScale1D,fakePredScale1G);
    FMLossScale1 = FMLossScale1 * lambdaFeatureMatching;

    % Compute the feature matching loss for scale 2
    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);

    % Compute the combined discriminator loss
    lossDCombined = (DLossScale1 + DLossScale2)/2 * lambdaDiscriminator;

    % Compute gradients for the discriminator
    gradParamsD = dlgradient(lossDCombined,discriminator.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));

    % Plot the losses
    addpoints(lossPlot.dline,iterNum,double(lossD));
    addpoints(lossPlot.gline,iterNum,double(lossGGAN));
    addpoints(lossPlot.fmline,iterNum,double(lossGFM));
    addpoints(lossPlot.vggline,iterNum,double(lossGVGG));
    drawnow

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,genPredFtrsG] = pix2pixHDAdverserialLoss(inpReal,inpGenerated

    % Discriminator layer names containing feature maps
    featureNames = {'lrelu1','lrelu2','lrelu3','lrelu4','fConv'};

    % 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
    genPredFtrsG = cell(size(featureNames));
    [genPredFtrsG{:}] = forward(discriminator,inpGenerated,"Outputs",featureNames);
    genPredG = genPredFtrsG{end};
    GLoss = (1 - genPredG).^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.

[3] Isola, Phillip, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. "Image-to-Image Translation with Conditional Adversarial Networks." In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 5967–76, 2017. https://doi.org/10.1109/CVPR.2017.632.

[4] Johnson, Justin, Alexandre Alahi, and Fei-Fei Li. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution." In *2016 European Conference on Computer Vision (ECCV)*, 694–711, 2016.

See Also

`combine` | `dlarray` | `pixelLabelDatastore` | `trainNetwork` | `trainingOptions` | `transform` | `vgg19`

More About

- “Datastores for Deep Learning” on page 16-2
- “List of Deep Learning Layers” on page 1-23
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194
- “Train Generative Adversarial Network (GAN)” on page 3-72

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 . This example uses the distinctive Van Gogh painting "Starry Night" as the style image and a photograph of a lighthouse as the content image.

Load Data

Load the style image and 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 .

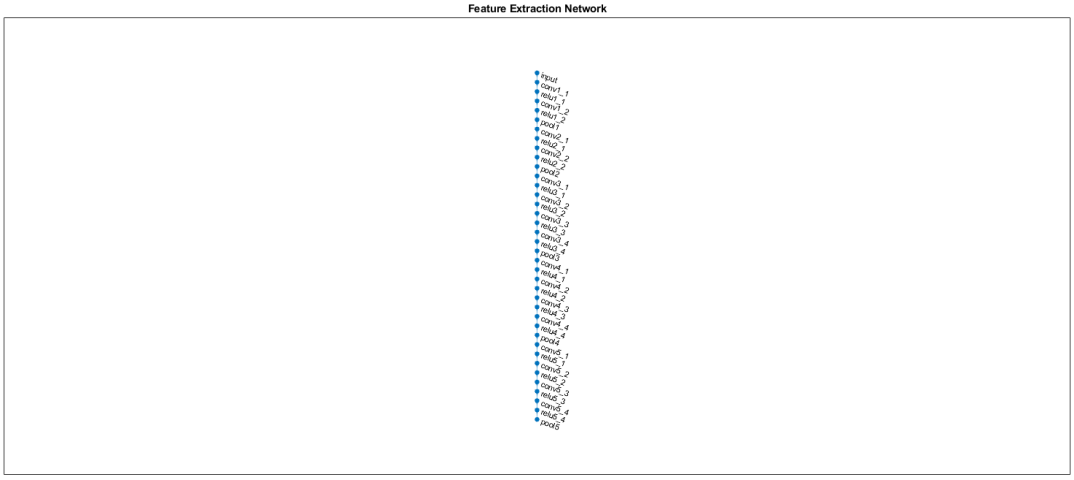
```
for l = 1:lastFeatureLayerIdx
    layer = layers(l);
    if isa(layer,'nnet.cnn.layer.MaxPooling2DLayer')
        layers(l) = averagePooling2dLayer(layer.PoolSize,'Stride',layer.Stride,'Name',layer.Name
    end
end
```

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 intitalize 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 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 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. Use of a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. 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.contentFeatureLayer
```

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,style
    [dlTransfer,trailingAvg,trailingAvgSq] = adamupdate(dlTransfer,grad,trailingAvg,trailingAvgS

    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

`dlarray` | `trainNetwork` | `trainingOptions` | `vgg19`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “List of Functions with dlarray Support” on page 15-194
- “List of Deep Learning Layers” on page 1-23

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` and `vehicleDetectorACF`) 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 using Deep Learning” (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/fasterRCNNResNet50EndToEn
    websave('fasterRCNNResNet50VehicleExample.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small labeled dataset that contains 295 images. 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 1-12). 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

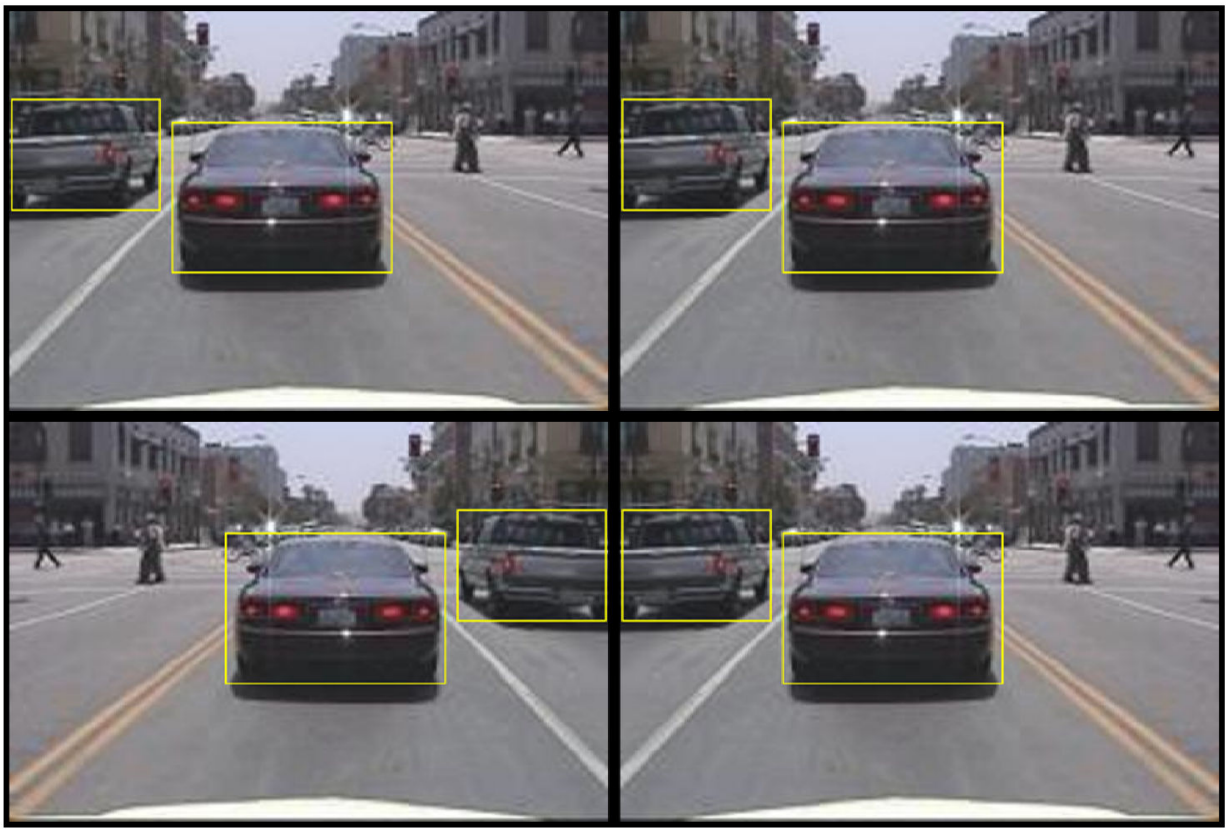
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

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.

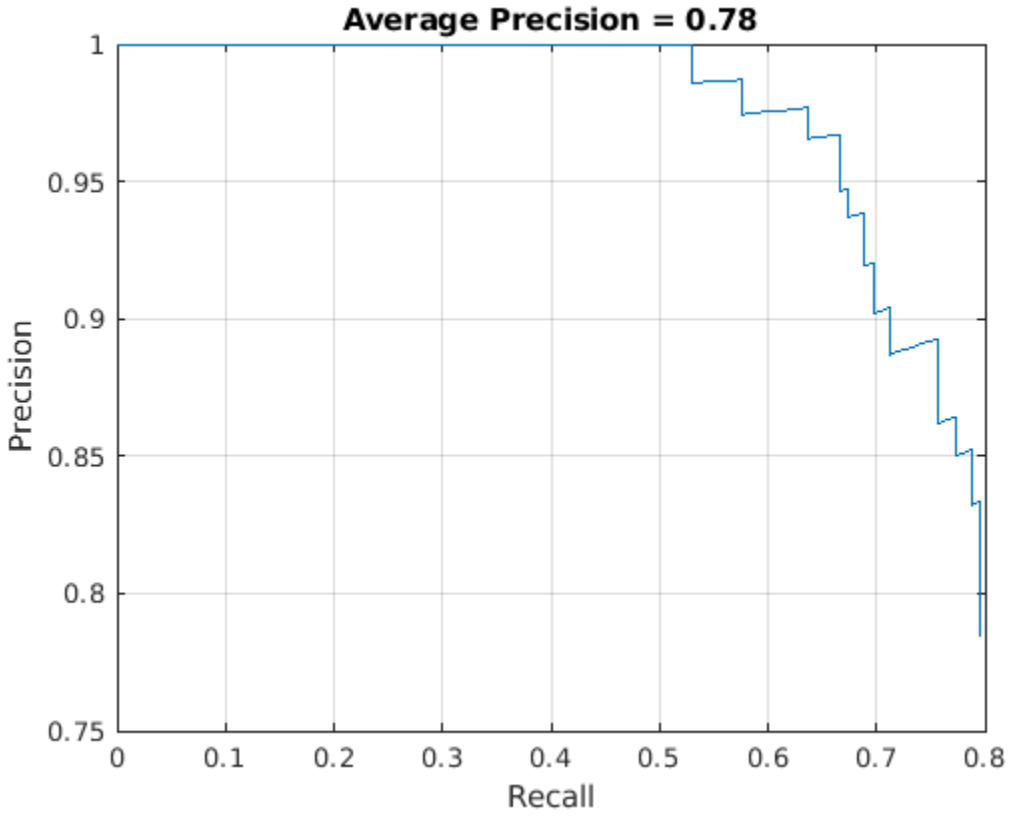
```
if doTrainingAndEval
    detectionResults = detect(detector,testData,'MinibatchSize',4);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detectionResults = pretrained.detectionResults;
end
```

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);
rout = affineOutputView(size(data{1}),tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
scale = targetSize(1:2)./size(data{1},[1 2]);
data{1} = imresize(data{1},targetSize(1:2));
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

`trainFastRCNNObjectDetector` | `trainFasterRCNNObjectDetector` | `trainRCNNObjectDetector`

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 8-145
- “Train Object Detector Using R-CNN Deep Learning” on page 8-131

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

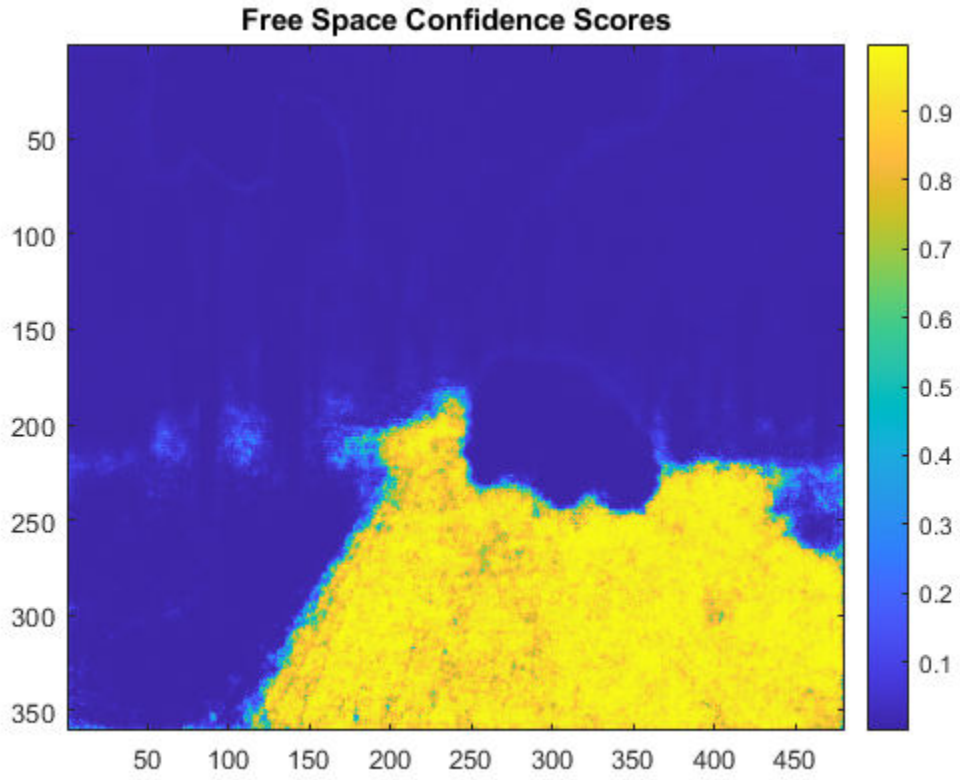
Estimate free space by processing the image using downloaded semantic segmentation network. The network returns classifications for each image pixel in the image. The free space is identified as image pixels that have been classified as `Road`.

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` object representing the monocular camera used to collect the CamVid[1] data. Configuring the `monoCamera` 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 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` 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` 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.

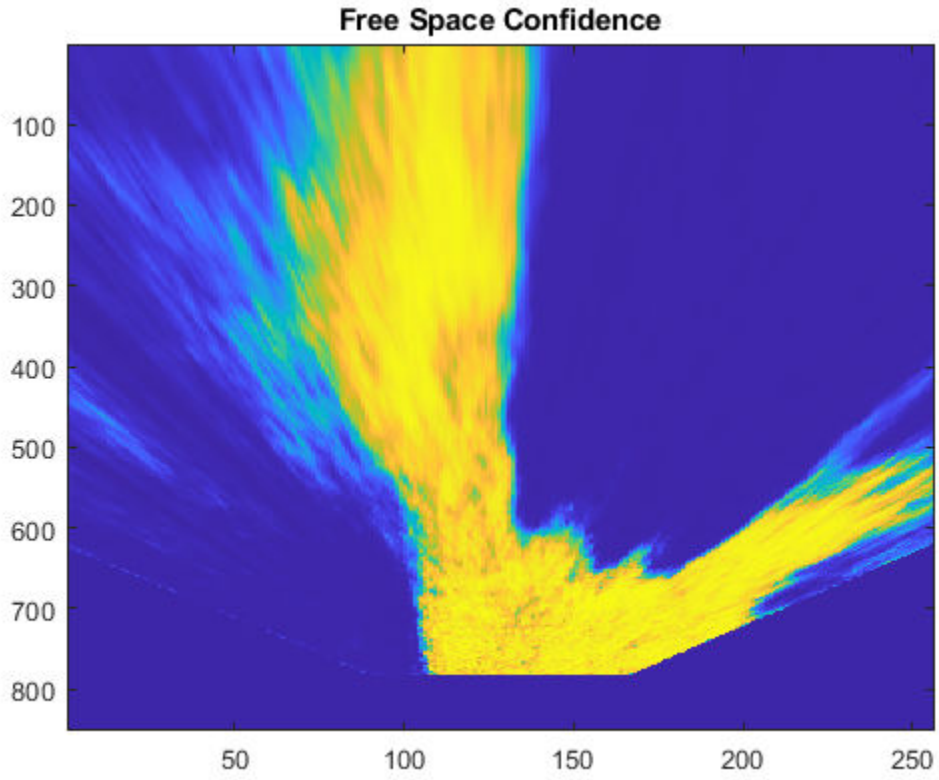
```
% Define bird's-eye-view transformation parameters.
distAheadOfSensor = 20; % in meters, as previously specified in monoCamera height input
spaceToOneSide    = 3;  % look 3 meters to the right and left
bottomOffset      = 0;
outView = [bottomOffset, distAheadOfSensor, -spaceToOneSide, spaceToOneSide];

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 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` 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.

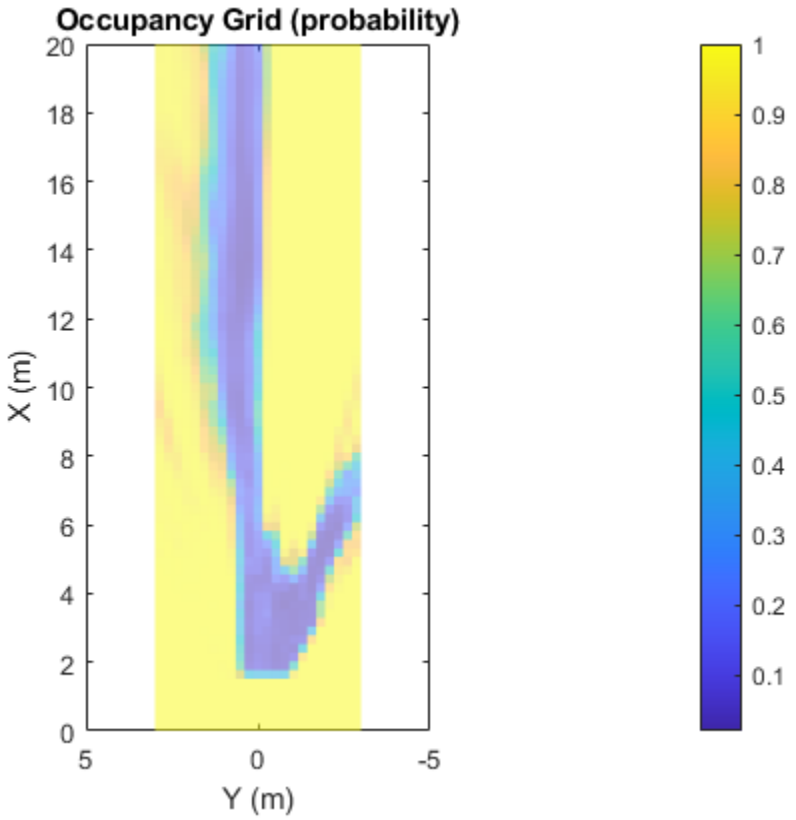
For brevity, the procedure shown above is implemented in the supporting function, `createOccupancyGridFromFreeSpaceEstimate`, which is listed at the end of this example.

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);
```

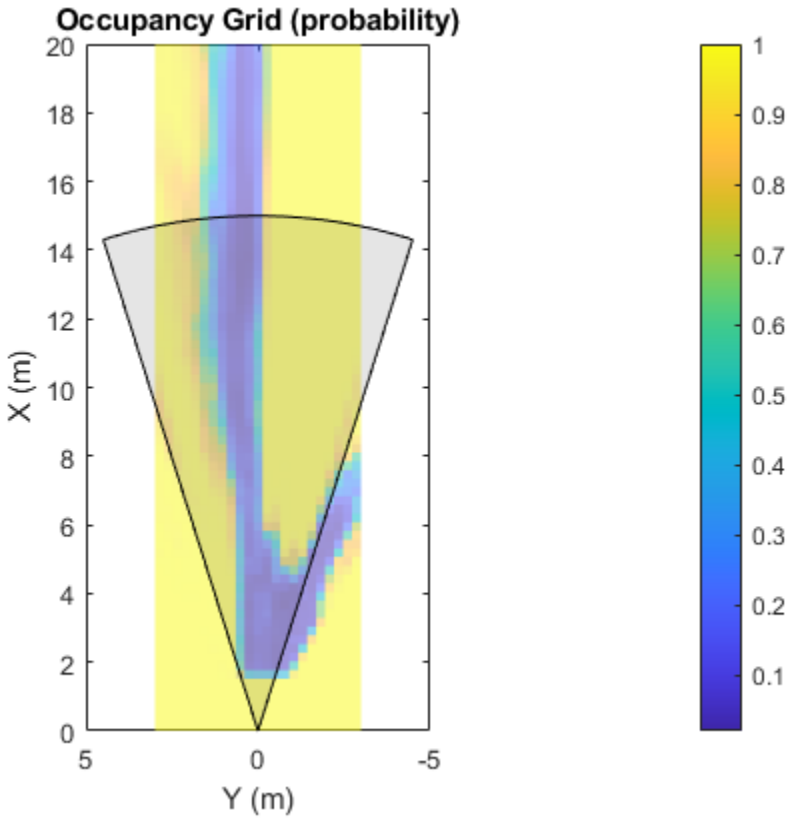
Visualize the occupancy grid using `birdsEyePlot`. Create a `birdsEyePlot` and add the occupancy grid on top using `pcolor`.

```
% 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`.

```
% 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
```

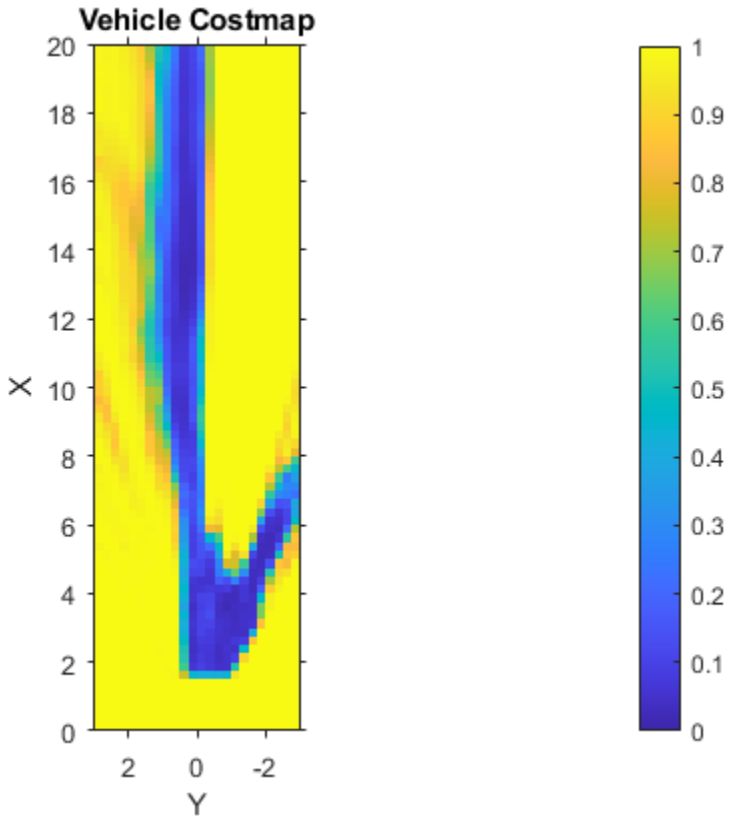


Displaying data from multiple sensors is useful for diagnosing and debugging decisions made by autonomous vehicles.

Create Vehicle Costmap Using the Occupancy Grid

The `vehicleCostmap` 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` 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)
```



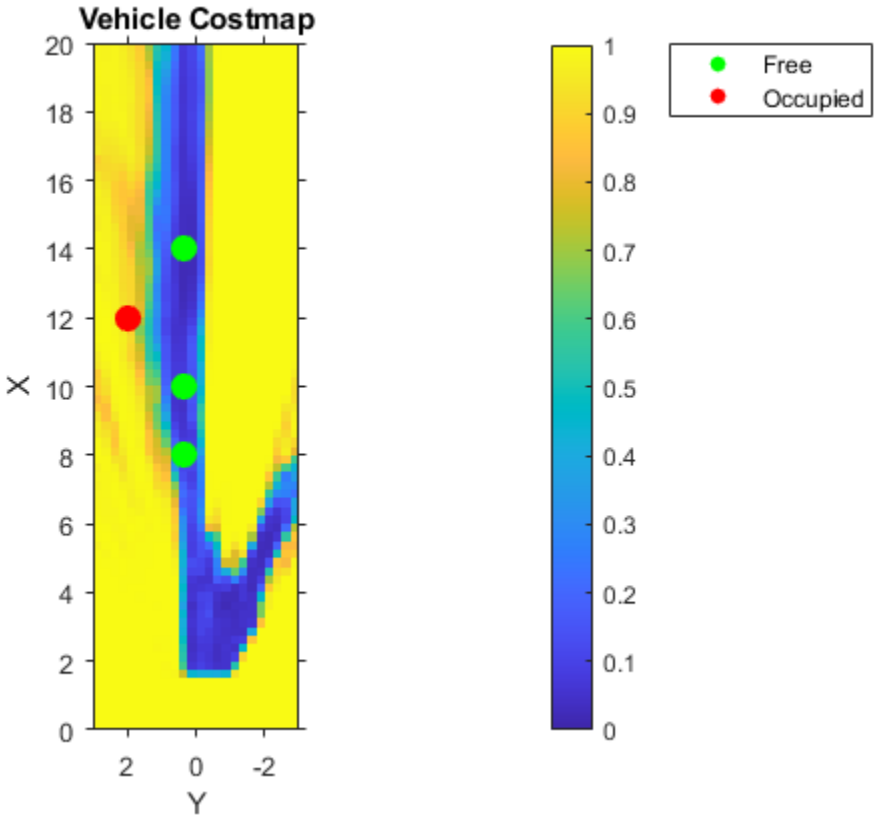
To illustrate how to use the `vehicleCostmap`, create a set of locations in world coordinates. These locations represent a path the vehicle could traverse.

```
% Create a set of locations in vehicle coordinates.
candidateLocations = [
    8 0.375
    10 0.375
    12 2
    14 0.375
    ];
```

Use `checkOccupied` 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`, and `checkOccupied` shown above illustrate the basic operations used by path planners such as `pathPlannerRRT`. 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]:
% http://mi.eng.cam.ac.uk/~gjb47/tmp/prl08.pdf
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
```

Signal Processing Examples

Radar Waveform Classification Using Deep Learning

This example shows how to classify radar waveform types of generated synthetic data using the Wigner-Ville distribution (WVD) and a deep convolutional neural network (CNN).

Modulation classification is an important function for an intelligent receiver. Modulation classification has numerous applications, such as cognitive radar and software-defined radio. Typically, to identify these waveforms and classify them by modulation type it is necessary to define meaningful features and input them into a classifier. While effective, this procedure can require extensive effort and domain knowledge to yield an accurate classification. This example explores a framework to automatically extract time-frequency features from signals and perform signal classification using a deep learning network.

The first part of this example simulates a radar classification system that synthesizes three pulsed radar waveforms and classifies them. The radar waveforms are:

- Rectangular
- Linear frequency modulation (LFM)
- Barker Code

A radar classification system does not exist in isolation. Rather, it resides in an increasingly occupied frequency spectrum, competing with other transmitted sources such as communications systems, radio, and navigation systems. The second part of this example extends the network to include additional communication modulation types. In addition to the first set of radar waveforms, the extended network synthesizes and identifies these communication waveforms:

- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast frequency modulation (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

This example primarily focuses on radar waveforms, with the classification being extended to include a small set of amplitude and frequency modulation communications signals. See “Modulation Classification with Deep Learning” (Communications Toolbox) for a full workflow of modulation classification with a wide array of communication signals.

Generate Radar Waveforms

Generate 3000 signals with a sample rate of `100` MHz for each modulation type. Use `phased.RectangularWaveform` for rectangular pulses, `phased.LinearFMWaveform` for LFM, and `phased.PhaseCodedWaveform` for phase coded pulses with Barker code.

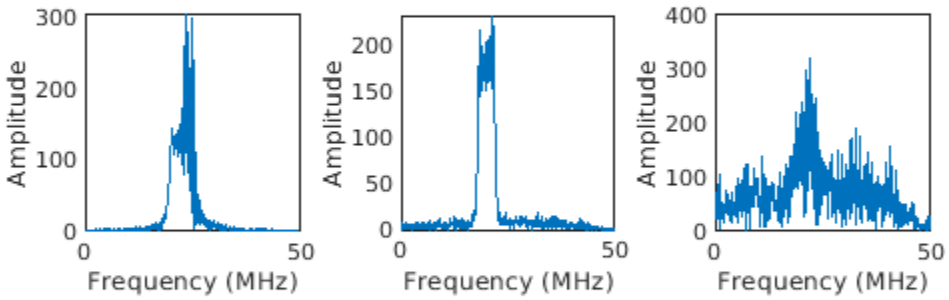
Each signal has unique parameters and is augmented with various impairments to make it more realistic. For each waveform, the pulse width and repetition frequency will be randomly generated. For LFM waveforms, the sweep bandwidth and direction are randomly generated. For Barker waveforms, the chip width and number are generated randomly. All signals are impaired with white Gaussian noise using the `awgn` function with a random signal-to-noise ratio in the range of [–6, 30] dB. A frequency offset with a random carrier frequency in the range of `[Fs/6, Fs/5]` is applied to each signal using the `comm.PhaseFrequencyOffset` object. Lastly, each signal is passed through a multipath Rician fading channel, `comm.RicianChannel`.

The provided helper function `helperGenerateRadarWaveforms` creates and augments each modulation type.

```
rng default
[wav, modType] = helperGenerateRadarWaveforms();
```

Plot the Fourier transform for a few of the LFM waveforms to show the variances in the generated set.

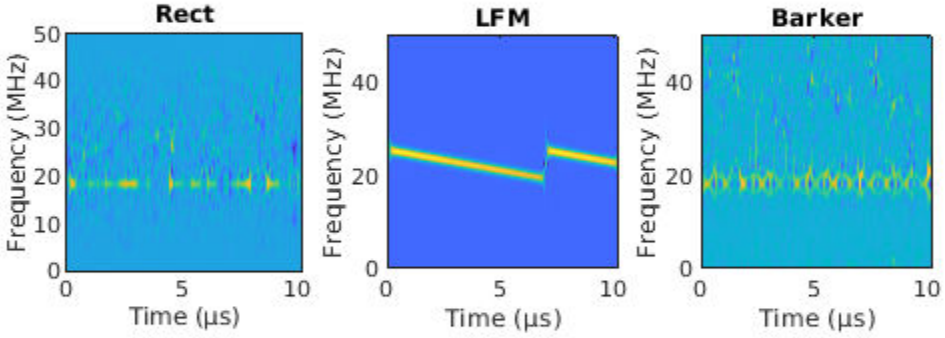
```
idLFM = find(modType == "LFM",3);
nfft = 2^nextpow2(length(wav{1}));
f = (0:(nfft/2-1))/nfft*100e6;

figure
subplot(1,3,1)
Z = fft(wav{idLFM(1)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,2)
Z = fft(wav{idLFM(2)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,3)
Z = fft(wav{idLFM(3)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
```



Feature Extraction Using Wigner-Ville Distribution

To improve the classification performance of machine learning algorithms, a common approach is to input extracted features in place of the original signal data. The features provide a representation of the input data that makes it easier for a classification algorithm to discriminate across the classes. The Wigner-Ville distribution represents a time-frequency view of the original data that is useful for time varying signals. The high resolution and locality in both time and frequency provide good features for the identification of similar modulation types. Use the `wvd` function to compute the smoothed pseudo WVD for each of the modulation types.

```
figure
subplot(1,3,1)
wvd(wav{find(modType == "Rect",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Rect')
subplot(1,3,2)
wvd(wav{find(modType == "LFM",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('LFM')
subplot(1,3,3)
wvd(wav{find(modType == "Barker",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Barker')
```



To store the smoothed-pseudo Wigner-Ville distribution of the signals, first create the directory `TFDDatabase` inside your temporary directory `tempdir`. Then create subdirectories in `TFDDatabase` for each modulation type. For each signal, compute the smoothed-pseudo Wigner-Ville distribution, and downsample the result to a 227-by-227 matrix. Save the matrix is a `.png` image file in the subdirectory corresponding to the modulation type of the signal. The helper function

`helperGenerateTFDfiles` performs all these steps. This process will take several minutes due to the large database size and the complexity of the `wvd` algorithm. You can replace `tempdir` with another directory where you have write permission.

```
parentDir = tempdir;
dataDir = 'TFDDatabase';
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,100e6)
```

Create an image datastore object for the created folder to manage the image files used for training the deep learning network. This step avoids having to load all images into memory. Specify the label source to be folder names. This assigns each signal's modulation type according to the folder name.

```
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker'});
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

The network is trained with 80% of the data and tested on with 10%. The remaining 10% is used for validation. Use the `splitEachLabel` function to divide the `imageDatastore` into training, validation, and testing sets.

```
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

Set Up Deep Learning Network

Before the deep learning network can be trained, define the network architecture. This example utilizes transfer learning SqueezeNet, a deep CNN created for image classification. Transfer learning is the process of retraining an existing neural network to classify new targets. This network accepts image input of size 227-by-227-by-3. Prior to input to the network, the custom read function `readTFDForSqueezeNet` will transform the two-dimensional time-frequency distribution to an RGB image of the correct size. SqueezeNet performs classification of 1000 categories in its default configuration.

Load SqueezeNet.

```
net = squeezenet;
```

Extract the layer graph from the network. Confirm that SqueezeNet is configured for images of size 227-by-227-by-3.

```
lgraphSqz = layerGraph(net);
lgraphSqz.Layers(1)
```

```
ans =
  ImageInputLayer with properties:

                      Name: 'data'
                 InputSize: [227 227 3]

   Hyperparameters
          DataAugmentation: 'none'
             Normalization: 'zerocenter'
    NormalizationDimension: 'auto'
                      Mean: [1×1×3 single]
```

To tune SqueezeNet for our needs, three of the last six layers need to be modified to classify the three radar modulation types of interest. 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 w
     3   'relu_conv10'                        ReLU                     ReLU
     4   'pool10'                             Global Average Pooling   Global average pooling
     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);
```

The last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, 'conv10'. Replace the layer with a new convolutional layer with the number of filters equal to the number of modulation types. Also increase the learning rate factors of the new layer.

```
numClasses = 3;
tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
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(end-5:end)

ans =
  6x1 Layer array with layers:

     1   'new_dropout'       Dropout                  60% dropout
     2   'new_conv'          Convolution              3 1x1 convolutions with stride [1  1] and p
     3   'relu_conv10'       ReLU                     ReLU
     4   'pool10'            Global Average Pooling   Global average pooling
     5   'prob'              Softmax                  softmax
     6   'new_classoutput'   Classification Output    crossentropyex
```

Choose options for the training process that ensures good network performance. Refer to the `trainingOptions` documentation for a description of each option.

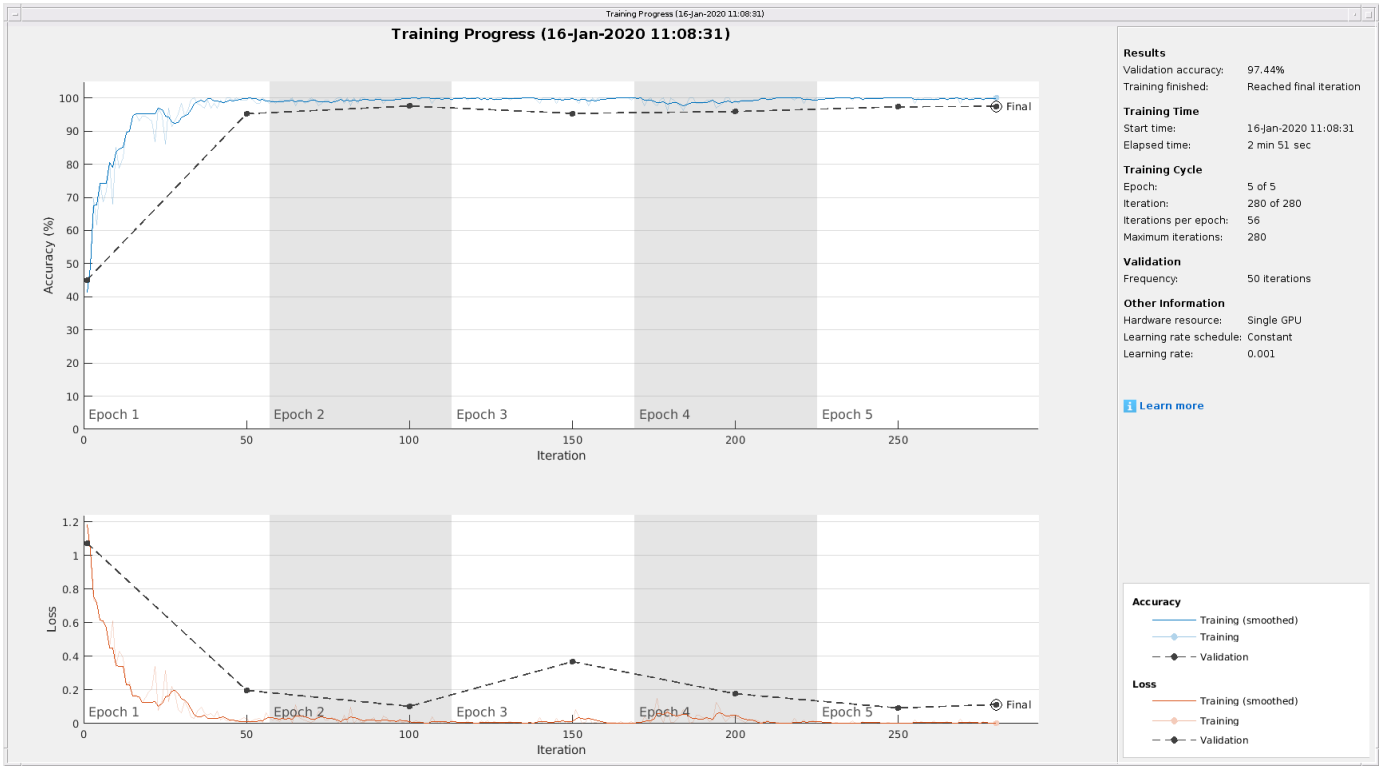
```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',128, ...
    'MaxEpochs',5, ...
```

```
'InitialLearnRate',1e-3, ...
'Shuffle','every-epoch', ...
'Verbose',false, ...
'Plots','training-progress',...
'ExecutionEnvironment','auto',...
'ValidationData',imdsValidation);
```

Train the Network

Use the `trainNetwork` command to train the created CNN. Because of the dataset's large size, the process may take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB automatically uses the GPU for training. Otherwise, it uses the CPU. The training accuracy plots in the figure show the progress of the network's learning across all iterations. On the three radar modulation types, the network classifies almost 100% of the training signals correctly.

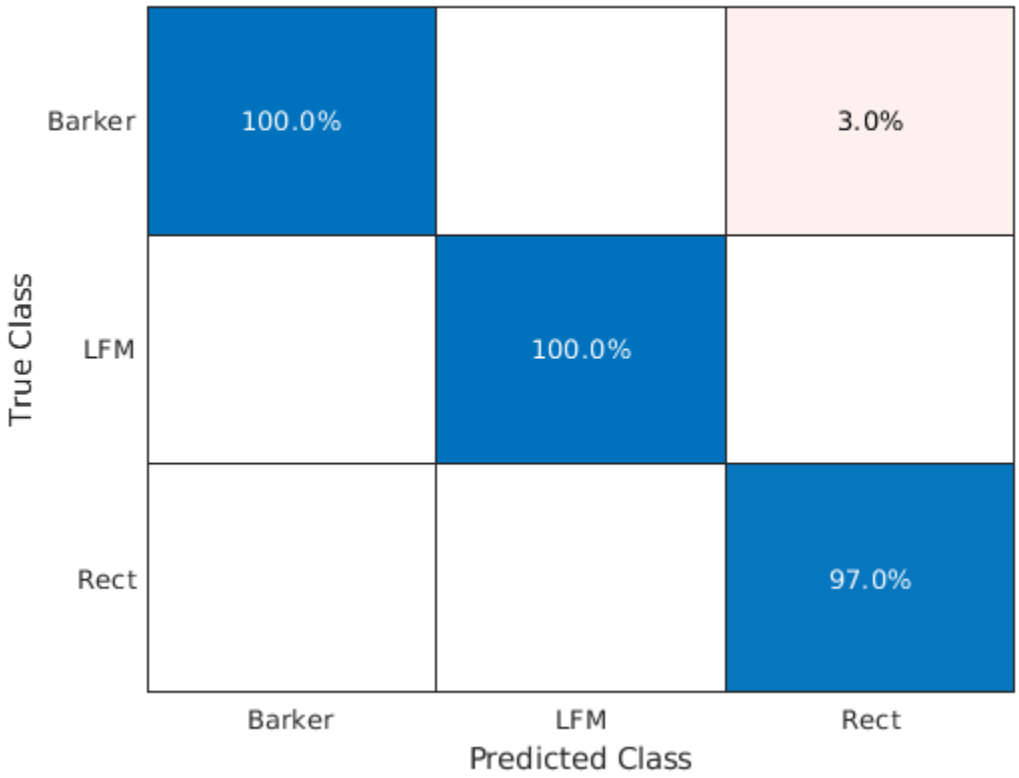
```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```



Evaluate Performance on Radar Waveforms

Use the trained network to classify the testing data using the `classify` command. A confusion matrix is one method to visualize classification performance. Use the `confusionchart` command to calculate and visualize the classification accuracy. For the three modulation types input to the network, almost all of the phase coded, LFM, and rectangular waveforms are correctly identified by the network.

```
predicted = classify(trainedNet,imdsTest);
figure
confusionchart(predicted,imdsTest.Labels,'Normalization','column-normalized')
```



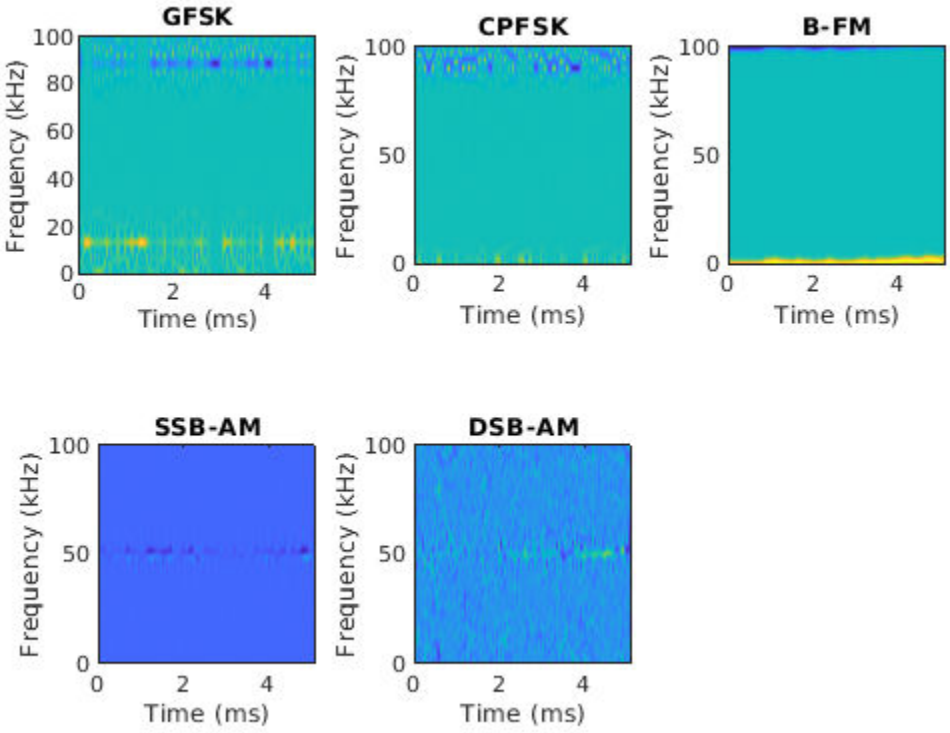
Generate Communications Waveforms and Extract Features

The frequency spectrum of a radar classification system must compete with other transmitted sources. Let's see how the created network extends to incorporate other simulated modulation types. Another MathWorks example, “Modulation Classification with Deep Learning” (Communications Toolbox), performs modulation classification of several different modulation types using Communications Toolbox™. The helper function `helperGenerateCommsWaveforms` generates and augments a subset of the modulation types used in that example. Since the WVD loses phase information, a subset of only the amplitude and frequency modulation types are used.

See the example link for an in-depth description of the workflow necessary for digital and analog modulation classification and the techniques used to create these waveforms. For each modulation type, use `wvd` to extract time-frequency features and visualize.

```
[wav, modType] = helperGenerateCommsWaveforms();

figure
subplot(2,3,1)
wvd(wav{find(modType == "GFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('GFSK')
subplot(2,3,2)
wvd(wav{find(modType == "CPFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('CPFSK')
subplot(2,3,3)
wvd(wav{find(modType == "B-FM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('B-FM')
subplot(2,3,4)
```

```
wvd(wav{find(modType == "SSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('SSB-AM')
subplot(2,3,5)
wvd(wav{find(modType == "DSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('DSB-AM')
```



Use the helper function `helperGenerateTFDfiles` again to compute the smoothed pseudo WVD for each input signal. Create an image datastore object to manage the image files of all modulation types.

```
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,200e3)
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker','GFSK','CPFSK','B-FM','SSB-AM','DSB-A
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

Again, divide the data into a training set, a validation set, and a testing set using the `splitEachLabel` function.

```
rng default
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

Adjust Deep Learning Network Architecture

Previously, the network architecture was set up to classify three modulation types. This must be updated to allow classification of all eight modulation types of both radar and communication signals. This is a similar process as before, with the exception that the `fullyConnectedLayer` now requires an output size of eight.

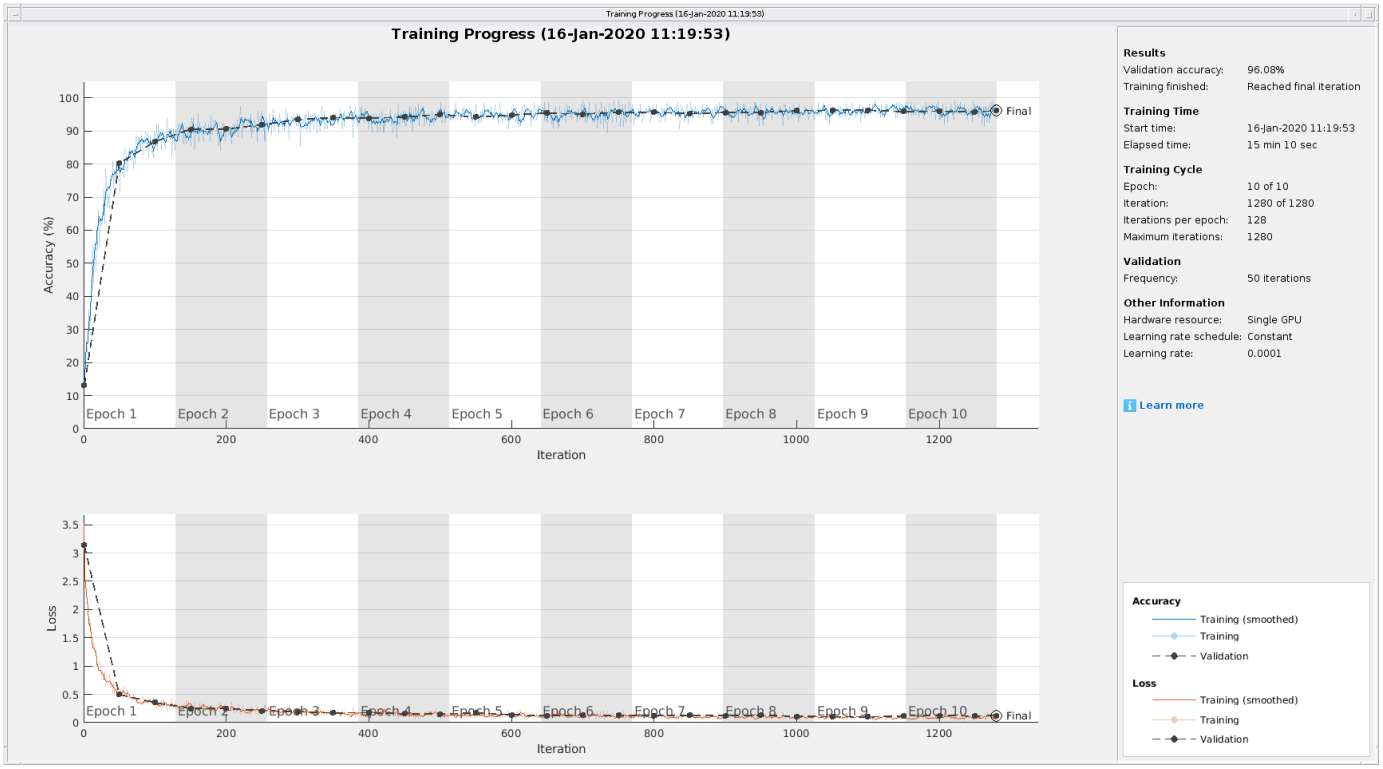
```
numClasses = 8;
net = squeezenet;
lgraphSqz = layerGraph(net);

tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);

tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);

tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Create a new set of training options.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',150, ...
    'MaxEpochs',10, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ExecutionEnvironment','auto',...
    'ValidationData',imdsValidation);
```

Use the `trainNetwork` command to train the created CNN. For all modulation types, the training converges with an accuracy of about 95% correct classification.

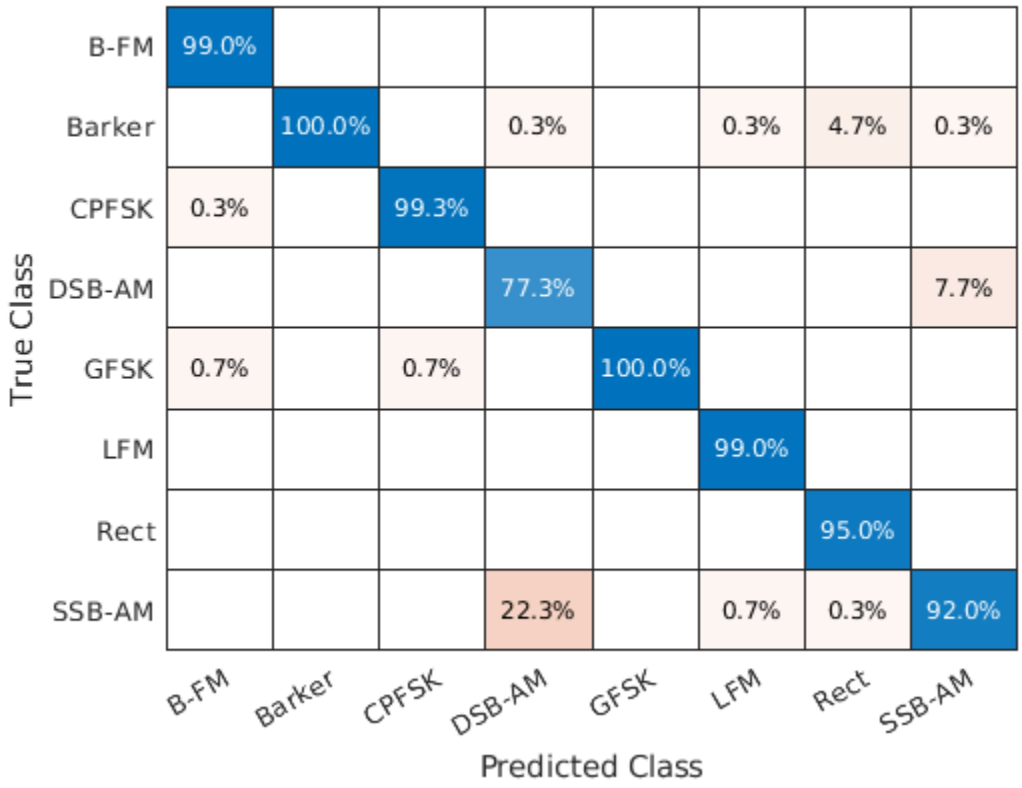
```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```



Evaluate Performance on All Signals

Use the `classify` command to classify the signals held aside for testing. Again, visualize the performance using `confusionchart`.

```
predicted = classify(trainedNet,imdsTest);
figure;
confusionchart(predicted,imdsTest.Labels,'Normalization','column-normalized')
```

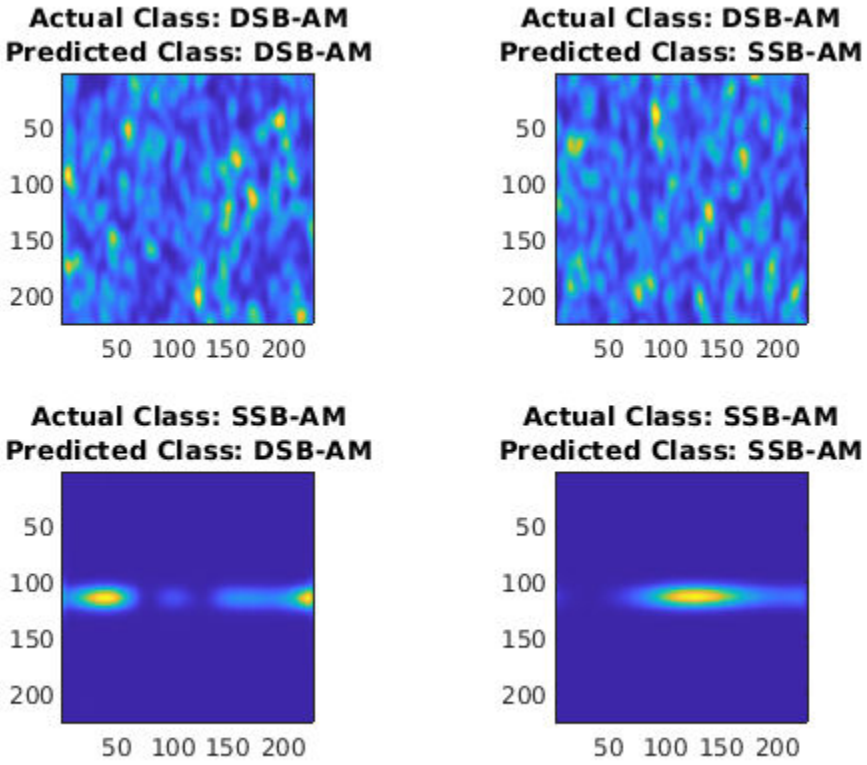


For the eight modulation types input to the network, over 99% of B-FM, CPFSK, GFSK, Barker, and LFM modulation types were correctly classified. On average, over 85% of AM signals were correctly identified. From the confusion matrix, a high percentage of SSB-AM signals were misclassified as DSB-AM, and DSB-AM signals as SSB-AM.

Let us investigate a few of these misclassifications to gain insight into the network's learning process. Use the `readimage` function on the image datastore to extract from the test dataset a single image from each class. The displayed WVD visually looks very similar. Since DSB-AM and SSB-AM signals have a very similar signature, this explains in part the network's difficulty in correctly classifying these two types. Further signal processing could make the differences between these two modulation types clearer to the network and result in improved classification.

```
DSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'DSB-AM'),1));
DSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'SSB-AM'),1));
SSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'DSB-AM'),1));
SSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'SSB-AM'),1));

figure
subplot(2,2,1)
imagesc(DSB_DSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: DSB-AM'})
subplot(2,2,2)
imagesc(DSB_SSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: SSB-AM'})
subplot(2,2,3)
imagesc(SSB_DSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: DSB-AM'})
```

```
subplot(2,2,4)
imagesc(SSB_SSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: SSB-AM'})
```



Summary

This example showed how radar and communications modulation types can be classified by using time-frequency techniques and a deep learning network. Further efforts for additional improvement could be investigated by utilizing time-frequency analysis available in Wavelet Toolbox™ and additional Fourier analysis available in Signal Processing Toolbox™.

References

[1] Brynolfsson, Johan, and Maria Sandsten. "Classification of one-dimensional non-stationary signals using the Wigner-Ville distribution in convolutional neural networks." *25th European Signal Processing Conference (EUSIPCO)*. IEEE, 2017.

[2] Liu, Xiaoyu, Diyu Yang, and Aly El Gamal. "Deep neural network architectures for modulation classification." *51st Asilomar Conference on Signals, Systems and Computers*. 2017.

[3] `Wang, Chao, Jian Wang, and Xudong Zhang. "Automatic radar waveform recognition based on time-frequency analysis and convolutional neural network." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017.

See Also

`classify` | `convolution2dLayer` | `layerGraph` | `trainNetwork` | `trainingOptions`

Related Examples

- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Deep Learning in MATLAB” on page 1-2

Pedestrian and Bicyclist Classification Using Deep Learning

This example shows how to classify pedestrians and bicyclists based on their micro-Doppler characteristics using a deep learning network and time-frequency analysis.

The movements of different parts of an object placed in front of a radar produce micro-Doppler signatures that can be used to identify the object. This example uses a convolutional neural network (CNN) to identify pedestrians and bicyclists based on their signatures.

This example trains the deep learning network using simulated data and then examines how the network performs at classifying two cases of overlapping signatures.

Synthetic Data Generation by Simulation

The data used to train the network is generated using `backscatterPedestrian` and `backscatterBicyclist` from Phased Array System Toolbox™. These functions simulate the radar backscattering of signals reflected from pedestrians and bicyclists, respectively.

The helper function `helperBackScatterSignals` generates a specified number of pedestrian, bicyclist, and car radar returns. Because the purpose of the example is to classify pedestrians and bicyclists, this example considers car signatures as noise sources only. To get an idea of the classification problem to solve, examine one realization of a micro-Doppler signature from a pedestrian, a bicyclist, and a car. (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. See “Radar Data Cube” (Phased Array System Toolbox) for more information.)

```
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);
```

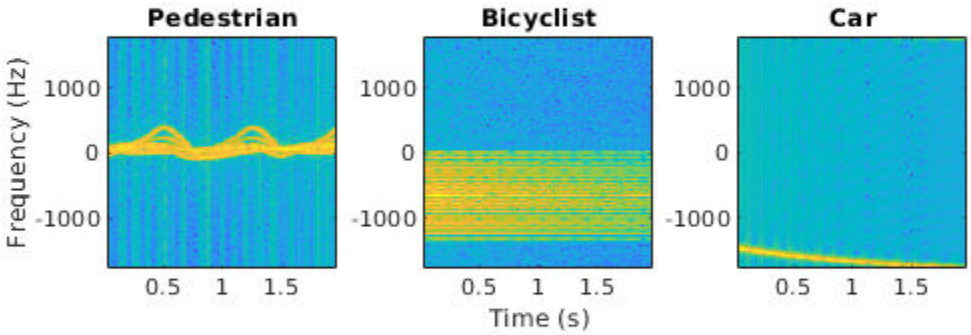
The helper function `helperDopplerSignatures` computes the short-time Fourier transform (STFT) of a radar return to generate the micro-Doppler signature. To obtain the micro-Doppler signatures, use the helper functions to apply the STFT and a preprocessing method to each signal.

```
[SPed,T,F] = helperDopplerSignatures(xPedRec,Tsamp);
[SBic,~,~] = helperDopplerSignatures(xBicRec,Tsamp);
[SCar,~,~] = helperDopplerSignatures(xCarRec,Tsamp);
```

Plot the time-frequency maps for the pedestrian, bicyclist, and car realizations.

```
% Plot the first realization of objects
figure
subplot(1,3,1)
imagesc(T,F,SPed(:,:,1))
ylabel('Frequency (Hz)')
title('Pedestrian')
axis square xy

subplot(1,3,2)
imagesc(T,F,SBic(:,:,1))
xlabel('Time (s)')
title('Bicyclist')
axis square xy

subplot(1,3,3)
```

```
imagesc(T,F,SCar(:,:,1))
title('Car')
axis square xy
```



The normalized spectrograms (STFT absolute values) show that the three objects have quite distinct signatures. Specifically, the spectrograms of the pedestrian and the bicyclist have rich micro-Doppler signatures caused by the swing of arms and legs and the rotation of wheels, respectively. By contrast, in this example, the car is modeled as a point target with rigid body, so the spectrogram of the car shows that the short-term Doppler frequency shift varies little, indicating little micro-Doppler effect.

Combining Objects

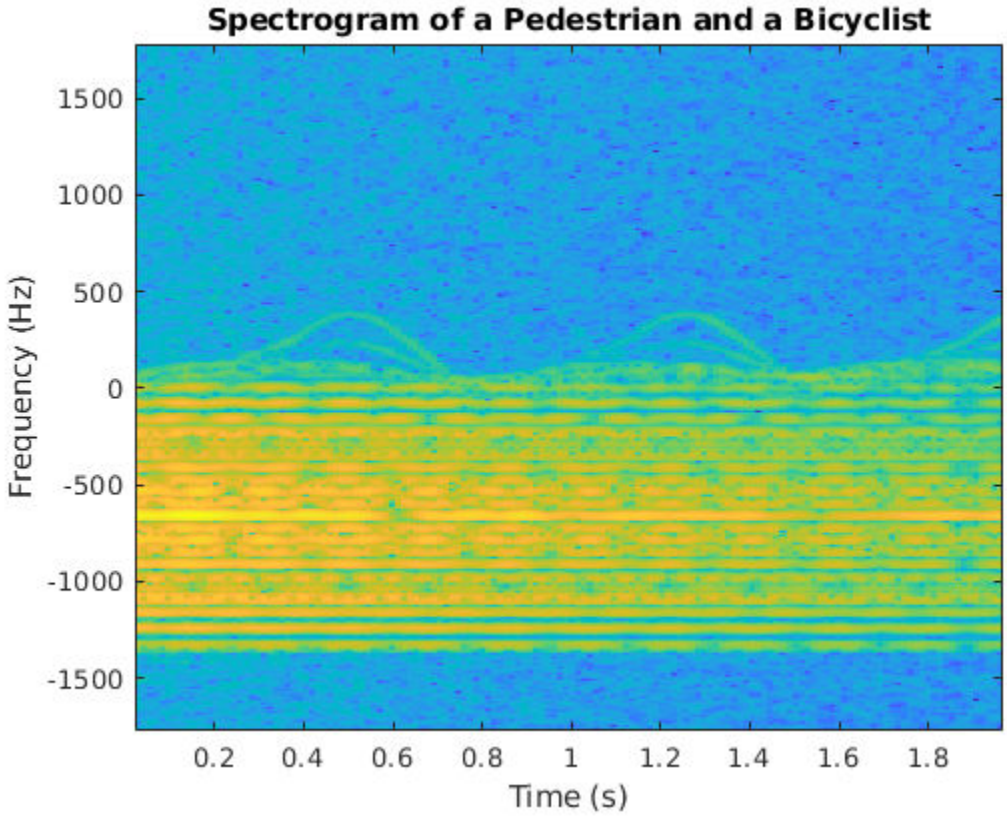
Classifying a single realization as a pedestrian or bicyclist is relatively simple because the pedestrian and bicyclist micro-Doppler signatures are dissimilar. However, classifying multiple overlapping pedestrians or bicyclists, with the addition of Gaussian noise or car noise, is much more difficult.

If multiple objects exist in the detection region of the radar at the same time, the received radar signal is a summation of the detection signals from all the objects. As an example, generate the received radar signal for a pedestrian and bicyclist with Gaussian background noise.

```
% Configure Gaussian noise level at the receiver
rx = phased.ReceiverPreamp('Gain',25,'NoiseFigure',10);

xRadarRec = complex(zeros(size(xPedRec)));
for ii = 1:size(xPedRec,3)
    xRadarRec(:,:,ii) = rx(xPedRec(:,:,ii) + xBicRec(:,:,ii));
end
```

Then obtain micro-Doppler signatures of the received signal by using the STFT.

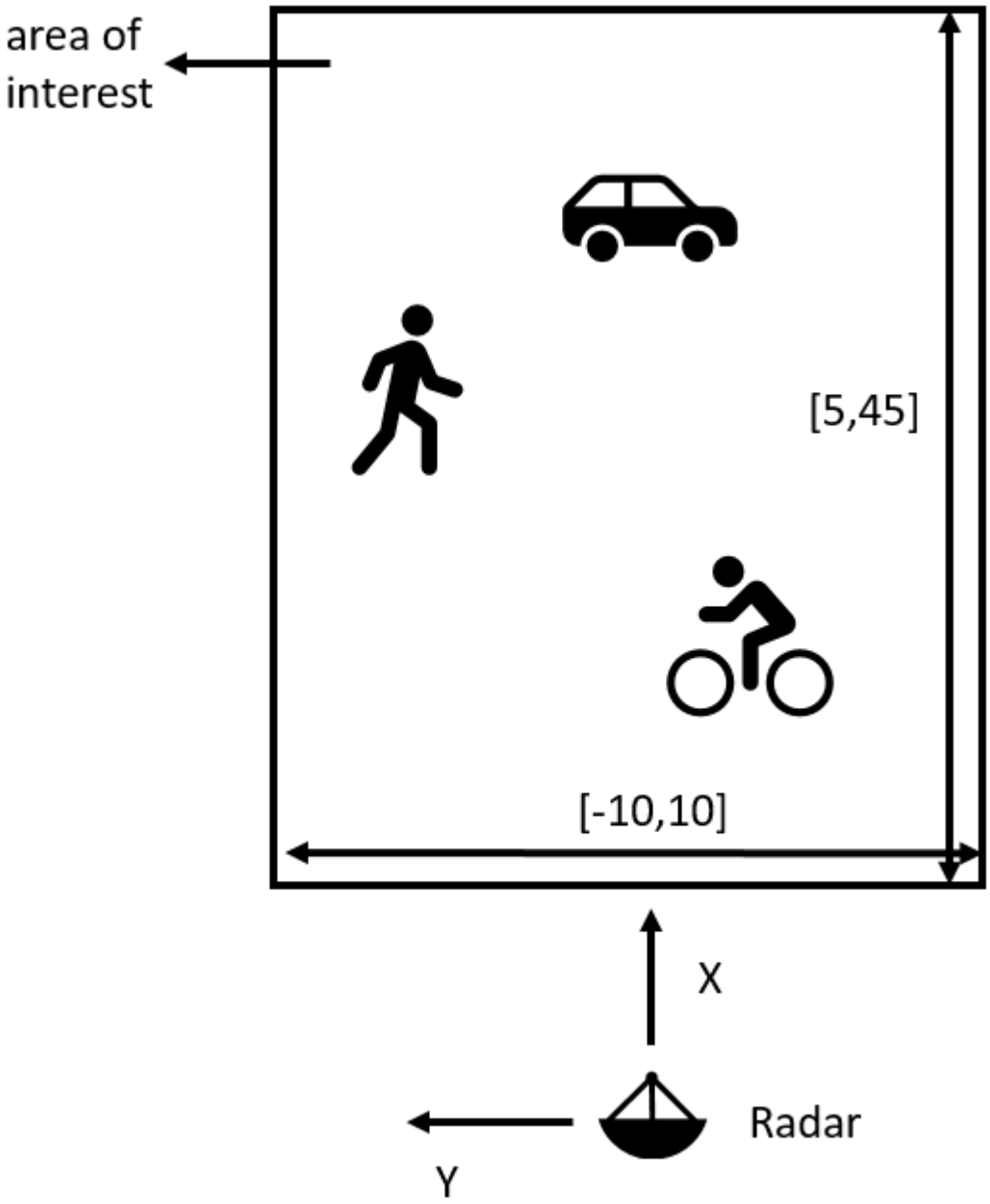
```
[S,~,~] = helperDopplerSignatures(xRadarRec,Tsamp);

figure
imagesc(T,F,S(:,:,1)) % Plot the first realization
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Spectrogram of a Pedestrian and a Bicyclist')
```



Because the pedestrian and bicyclist signatures overlap in time and frequency, differentiating between the two objects is difficult.

Generate Training Data

In this example, you train a CNN by using data consisting of simulated realizations of objects with varying properties—for example, bicyclists pedaling at different speeds and pedestrians with different heights walking at different speeds. Assuming the radar is fixed at the origin, in one realization, one object or multiple objects are uniformly distributed in a rectangular area of [5, 45] and [–10, 10] meters along the X and Y axes, respectively.



The other properties of the three objects that are randomly tuned are as follows:

1) Pedestrians

- Height — Uniformly distributed in the interval of [1.5, 2] meters
- Heading — Uniformly distributed in the interval of [–180, 180] degrees
- Speed — Uniformly distributed in the interval of [0, 1.4h] meters/second, where h is the height value

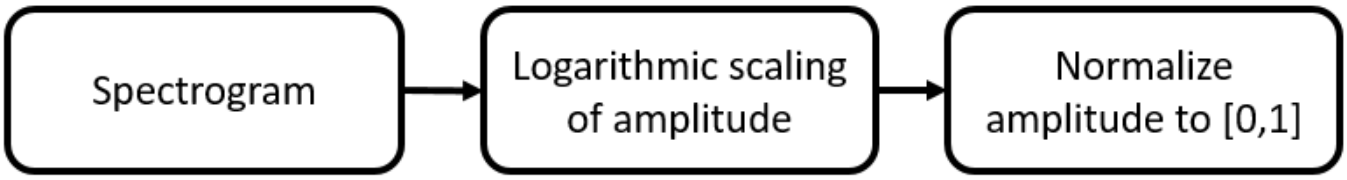
2) Bicyclists

- Heading — Uniformly distributed in the interval of [–180, 180] degrees
- Speed — Uniformly distributed in the interval of [1, 10] meters/second
- Gear transmission ratio — Uniformly distributed in the interval of [0.5, 6]
- Pedaling or coasting — 50% probability of pedaling (coasting means that the cyclist is moving without pedaling)

3) Cars

- Velocity — Uniformly distributed in the interval of [0, 10] meters/second along the X and Y directions

The input to the convolutional network is micro-Doppler signatures consisting of spectrograms expressed in decibels and normalized to [0, 1], as shown in this figure:



Radar returns originate from different objects and different parts of objects. Depending on the configuration, some returns are much stronger than others. Stronger returns tend to obscure weaker ones. Logarithmic scaling augments the features by making return strengths comparable. Amplitude normalization helps the CNN converge faster.

The data set contains realizations of the following scenes:

- One pedestrian present in the scene
- One bicyclist present in the scene
- One pedestrian and one bicyclist present in the scene
- Two pedestrians present in the scene
- Two bicyclists present in the scene

Download Data

The data for this example consists of 20,000 pedestrian, 20,000 bicyclist, and 12,500 car signals generated by using the helper functions `helperBackScatterSignals` and `helperDopplerSignatures`. The signals are divided into two data sets: one without car noise samples and one with car noise samples.

For the first data set (without car noise), the pedestrian and bicyclist signals were combined, Gaussian noise was added, and micro-Doppler signatures were computed to generate 5000 signatures for each of the five scenes to be classified.

In each category, 80% of the signatures (that is, 4000 signatures) are reserved for the training data set while 20% of the signatures (that is, 1000 signatures) are reserved for the test data set.

To generate the second data set (with car noise), the procedure for the first data set was followed, except that car noise was added to 50% of the signatures. The proportion of signatures with and without car noise is the same in the training and test data sets.

Download and unzip the data in your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. Due to the large size of the dataset, this process may take several minutes. If you have the data 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/PedBicCarData.zip';
saveFolder = fullfile(tempdir,'PedBicCarData');
zipFile = fullfile(tempdir,'PedBicCarData.zip');
if ~exist(saveFolder,'dir')
    websave(zipFile,dataURL);
end

% Unzip the data
unzip(zipFile,tempdir)
```

The data files are as follows:

- `trainDataNoCar.mat` contains the training data set `trainDataNoCar` and its label set `trainLabelNoCar`.
- `testDataNoCar.mat` contains the test data set `testDataNoCar` and its label set `testLabelNoCar`.
- `trainDataCarNoise.mat` contains the training data set `trainDataCarNoise` and its label set `trainLabelCarNoise`.
- `testDataCarNoise.mat` contains the test data set `testDataCarNoise` and its label set `testLabelCarNoise`.
- `TF.mat` contains the time and frequency information for the micro-Doppler signatures.

Network Architecture

Create a CNN with five convolution layers and one fully connected layer. The first four convolution layers are followed by a batch normalization layer, a rectified linear unit (ReLU) activation layer, and a max pooling layer. In the last convolution layer, the max pooling layer is replaced by an average pooling layer. The output layer is a classification layer after softmax activation. For network design guidance, see “Deep Learning Tips and Tricks” on page 1-45.

```
layers = [
    imageInputLayer([size(S,1),size(S,2),1],'Normalization','none')

    convolution2dLayer(10,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
```

```
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(5,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,'Stride',2)

    fullyConnectedLayer(5)
    softmaxLayer

    classificationLayer]
```

```
layers =
  24x1 Layer array with layers:

     1   ''   Image Input             400x144x1 images
     2   ''   Convolution             16 10x10 convolutions with stride [1  1] and padding 'same
     3   ''   Batch Normalization     Batch normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             10x10 max pooling with stride [2  2] and padding [0  0  0
     6   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding 'same'
     7   ''   Batch Normalization     Batch normalization
     8   ''   ReLU                    ReLU
     9   ''   Max Pooling             10x10 max pooling with stride [2  2] and padding [0  0  0
    10   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding 'same'
    11   ''   Batch Normalization     Batch normalization
    12   ''   ReLU                    ReLU
    13   ''   Max Pooling             10x10 max pooling with stride [2  2] and padding [0  0  0
    14   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding 'same'
    15   ''   Batch Normalization     Batch normalization
    16   ''   ReLU                    ReLU
    17   ''   Max Pooling             5x5 max pooling with stride [2  2] and padding [0  0  0  0
    18   ''   Convolution             32 5x5 convolutions with stride [1  1] and padding 'same'
    19   ''   Batch Normalization     Batch normalization
    20   ''   ReLU                    ReLU
    21   ''   Average Pooling         2x2 average pooling with stride [2  2] and padding [0  0  0
    22   ''   Fully Connected         5 fully connected layer
    23   ''   Softmax                 softmax
    24   ''   Classification Output   crossentropyex
```

Specify the optimization solver and the hyperparameters to train the CNN using `trainingOptions`. This example uses the ADAM optimizer and a mini-batch size of 128. Train the network using either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher. For information on other parameters, see `trainingOptions`. This example uses a GPU for training.

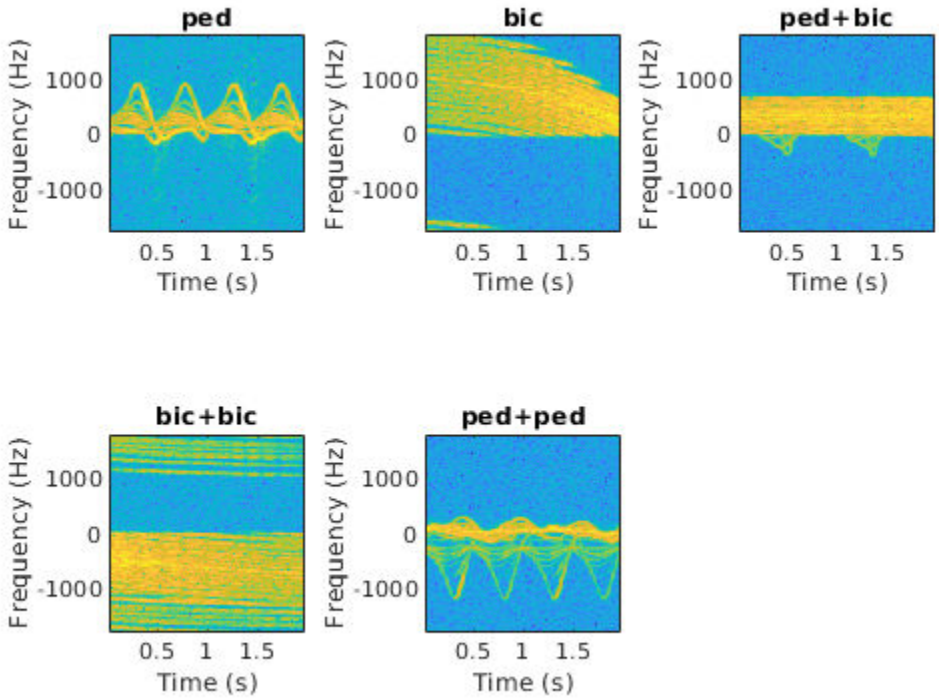
```
options = trainingOptions('adam', ...
    'ExecutionEnvironment','gpu',...
    'MiniBatchSize',128, ...
    'MaxEpochs',30, ...
    'InitialLearnRate',1e-2, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',10, ...
    'Shuffle','every-epoch', ...
```

```
    'Verbose',false, ...
    'Plots','training-progress');
```

Classify Signatures Without Car Noise

Load the data set without car noise and use the helper function `helperPlotTrainData` to plot one example of each of the five categories in the training data set,

```
load(fullfile(tempdir,'PedBicCarData','trainDataNoCar.mat')) % load training data set
load(fullfile(tempdir,'PedBicCarData','testDataNoCar.mat')) % load test data set
load(fullfile(tempdir,'PedBicCarData','TF.mat')) % load time and frequency information

helperPlotTrainData(trainDataNoCar,trainLabelNoCar,T,F)
```



Train the CNN that you created. You can view the accuracy and loss during the training process. In 30 epochs, the training process achieves almost 95% accuracy.

```
trainedNetNoCar = trainNetwork(trainDataNoCar,trainLabelNoCar,layers,options);
```

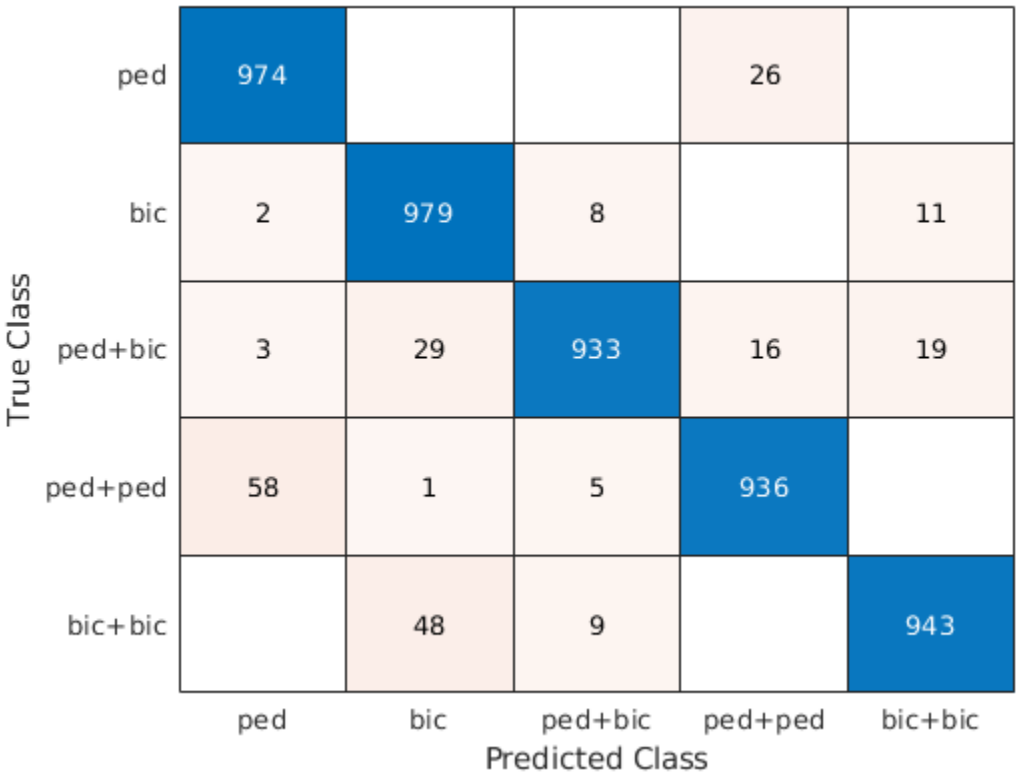
Use the trained network and the `classify` function to obtain the predicted labels for the test data set `testDataNoCar`. The variable `predTestLabel` contains the network predictions. The network achieves about 95% accuracy for the test data set without the car noise.

```
predTestLabel = classify(trainedNetNoCar,testDataNoCar);
testAccuracy = mean(predTestLabel == testLabelNoCar)
```

```
testAccuracy = 0.9530
```

Use a confusion matrix to view detailed information about prediction performance for each category. The confusion matrix for the trained network shows that, in each category, the network predicts the labels of the signals in the test data set with a high degree of accuracy.

```
figure
confusionchart(testLabelNoCar,predTestLabel);
```



Classify Signatures with Car Noise

To analyze the effects of car noise, classify data containing car noise with the `trainedNetNoCar` network, which was trained without car noise.

Load the car-noise-corrupted test data set `testDataCarNoise.mat`.

```
load(fullfile(tempdir,'PedBicCarData','testDataCarNoise.mat'))
```

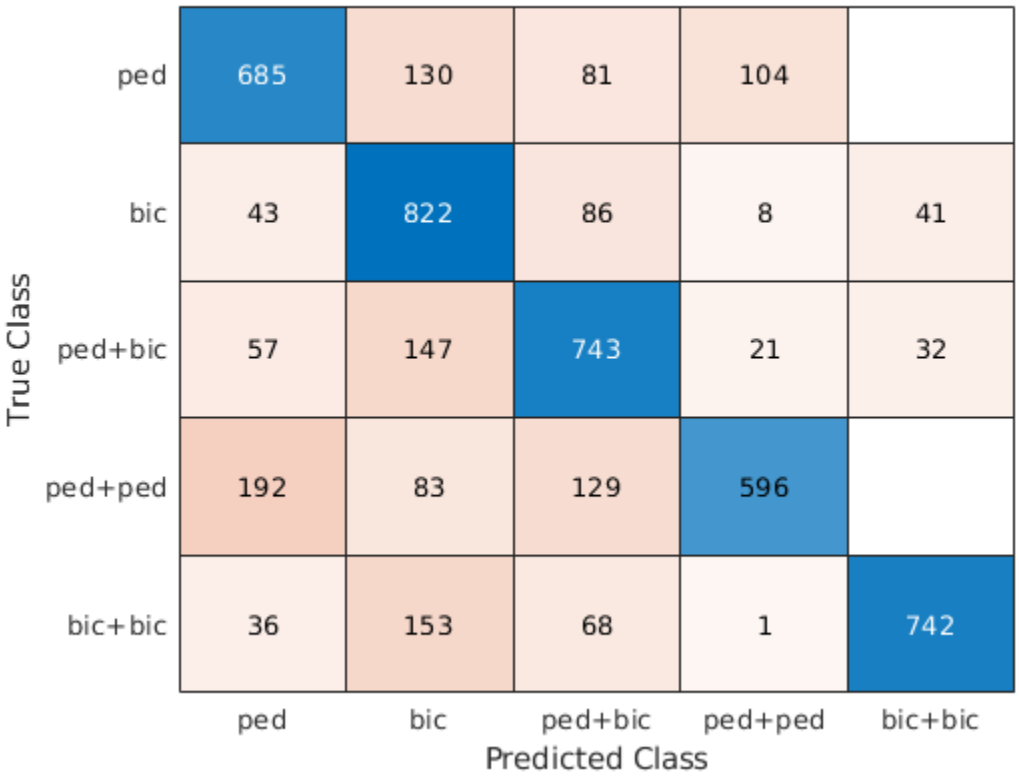
Input the car-noise-corrupted test data set to the network. The prediction accuracy for the test data set with the car noise drops significantly, to around 70%, because the network never saw training samples containing car noise.

```
predTestLabel = classify(trainedNetNoCar,testDataCarNoise);
testAccuracy = mean(predTestLabel == testLabelCarNoise)
```

```
testAccuracy = 0.7176
```

The confusion matrix shows that most prediction errors occur when the network takes in scenes from the "pedestrian," "pedestrian+pedestrian," or "pedestrian+bicyclist" classes and classifies them as "bicyclist."

```
confusionchart(testLabelCarNoise,predTestLabel);
```



Car noise significantly impedes the performance of the classifier. To solve this problem, train the CNN using data that contains car noise.

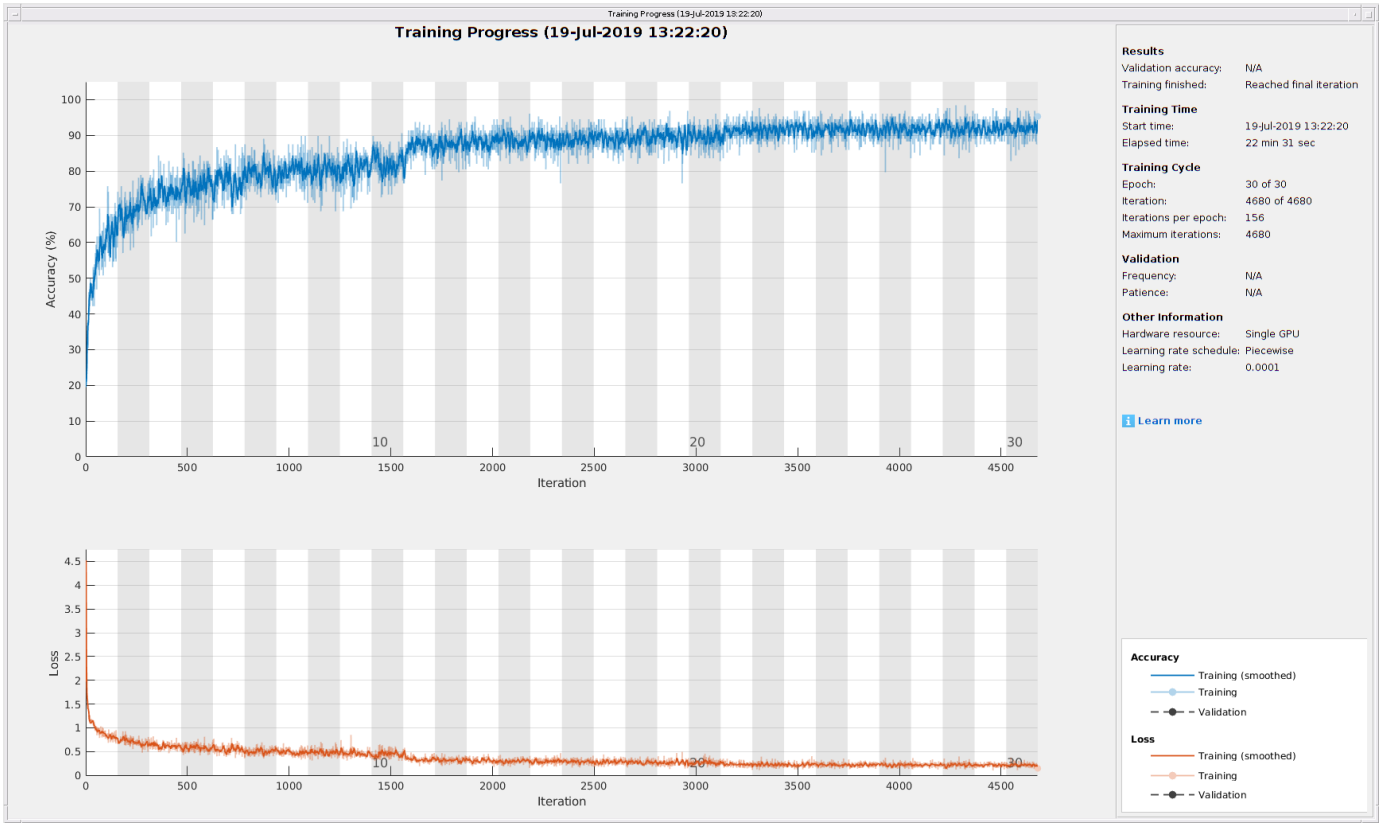
Retrain CNN by Adding Car Noise to Training Data Set

Load the car-noise-corrupted training data set `trainDataCarNoise.mat`.

```
load(fullfile(tempdir,'PedBicCarData','trainDataCarNoise.mat'))
```

Retrain the network by using the car-noise-corrupted training data set. In 30 epochs, the training process achieves almost 90% accuracy.

```
trainedNetCarNoise = trainNetwork(trainDataCarNoise,trainLabelCarNoise,layers,options);
```



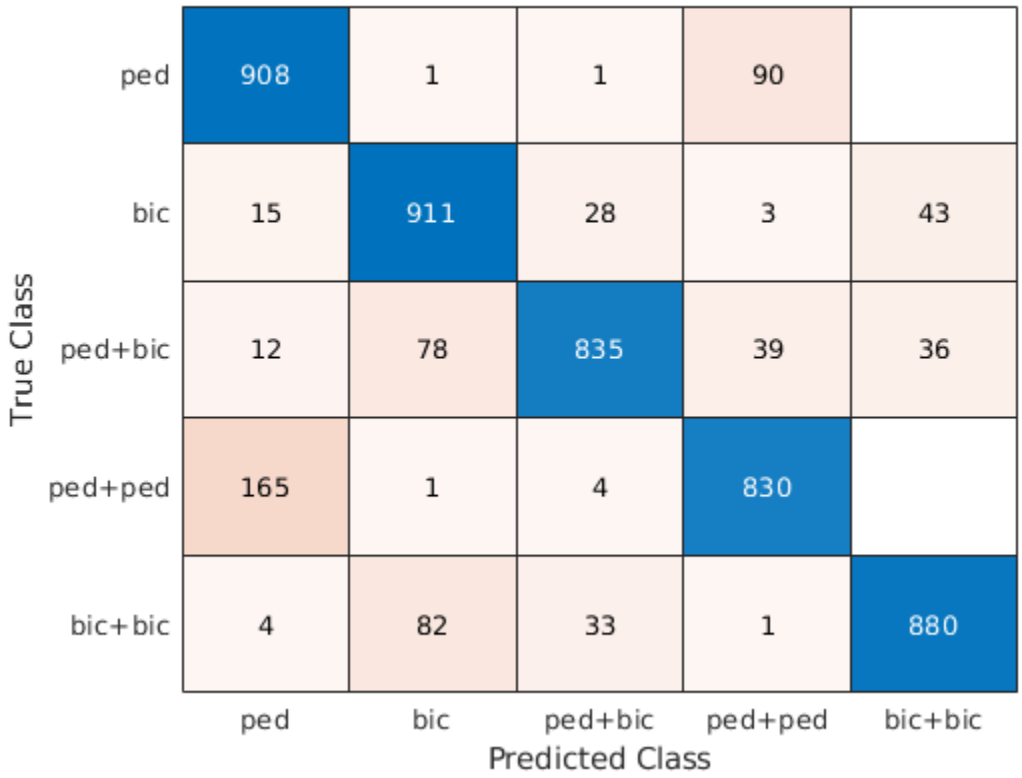
Input the car-noise-corrupted test data set to the network `trainedNetCarNoise`. The prediction accuracy is about 87%, which is approximately 15% higher than the performance of the network trained without car noise samples.

```
predTestLabel = classify(trainedNetCarNoise,testDataCarNoise);
testAccuracy = mean(predTestLabel == testLabelCarNoise)
```

```
testAccuracy = 0.8728
```

The confusion matrix shows that the network `trainedNetCarNoise` performs much better at predicting scenes with one pedestrian and scenes with two pedestrians.

```
confusionchart(testLabelCarNoise,predTestLabel);
```

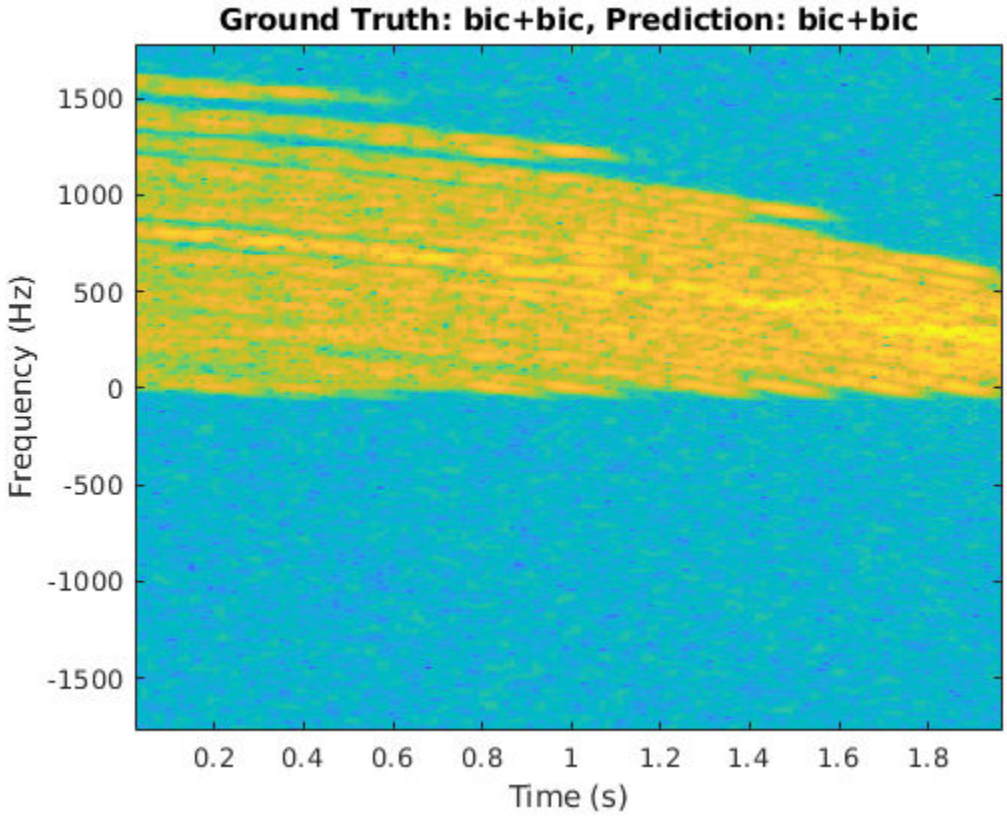


Case Study

To better understand the performance of the network, examine its performance in classifying overlapping signatures. This section is just for illustration. Due to the non-deterministic behavior of GPU training, you may not get the same classification results in this section when you rerun this example.

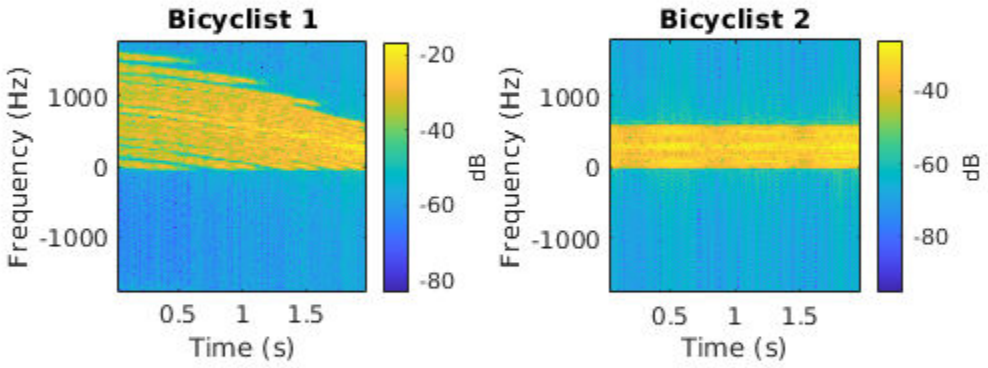
For example, signature #4 of the car-noise-corrupted test data, which does not have car noise, has two bicyclists with overlapping micro-Doppler signatures. The network correctly predicts that the scene has two bicyclists.

```
k = 4;
imagesc(T,F,testDataCarNoise(:,:,:,k))
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Ground Truth: '+string(testLabelCarNoise(k))+', Prediction: '+string(predTestLabel(k)))
```



From the plot, the signature appears to be from only one bicyclist. Load the data `CaseStudyData.mat` of the two objects in the scene. The data contains return signals summed along the fast time. Apply the STFT to each signal.

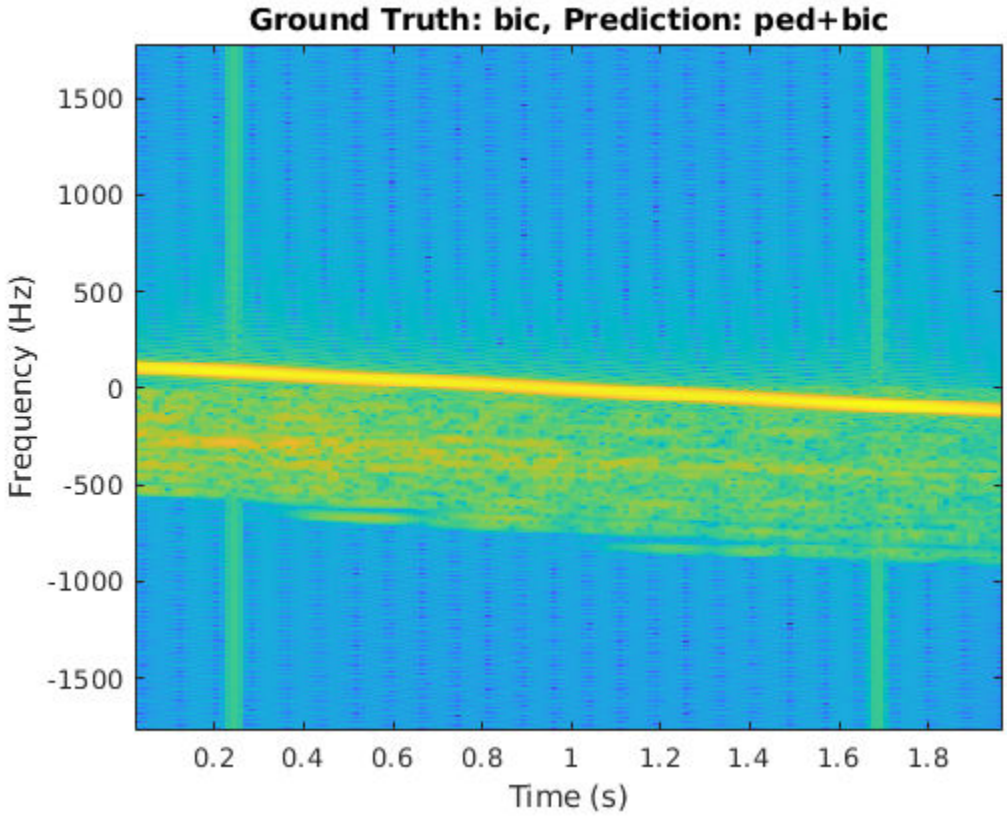
```
load CaseStudyData.mat
M = 200; % FFT window length
beta = 6; % window parameter
w = kaiser(M,beta); % kaiser window
R = floor(1.7*(M-1)/(beta+1)); % ROUGH estimate
noverlap = M-R; % overlap length

[Sc,F,T] = stft(x,1/Tsamp,'Window',w,'FFTLength',M*2,'OverlapLength',noverlap);

for ii = 1:2
    subplot(1,2,ii)
    imagesc(T,F,10*log10(abs(Sc(:,:,ii))))
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    title('Bicyclist')
    axis square xy
    title(['Bicyclist ' num2str(ii)])
    c = colorbar;
    c.Label.String = 'dB';
end
```



The amplitudes of the Bicyclist 2 signature are much weaker than those of Bicyclist 1, and the signatures of the two bicyclists overlap. When they overlap, the two signatures cannot be visually distinguished. However, the neural network classifies the scene correctly.

Another case of interest is when the network confuses car noise with a bicyclist, as in signature #267 of the car-noise-corrupted test data:

```
figure
k = 267;
imagesc(T,F,testDataCarNoise(:,:,:,k))
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Ground Truth: '+string(testLabelCarNoise(k))+', Prediction: '+string(predTestLabel(k)))
```



The signature of the bicyclist is weak compared to that of the car, and the signature has spikes from the car noise. Because the signature of the car closely resembles that of a bicyclist pedaling or a pedestrian walking at a low speed, and has little micro-Doppler effect, there is a high possibility that the network will classify the scene incorrectly.

References

[1] Chen, V. C. *The Micro-Doppler Effect in Radar*. London: Artech House, 2011.

[2] Gurbuz, S. Z., and Amin, M. G. "Radar-Based Human-Motion Recognition with Deep Learning: Promising Applications for Indoor Monitoring." *IEEE Signal Processing Magazine*. Vol. 36, Issue 4, 2019, pp. 16–28.

[3] Belgiovane, D., and C. C. Chen. "Micro-Doppler Characteristics of Pedestrians and Bicycles for Automotive Radar Sensors at 77 GHz." In *11th European Conference on Antennas and Propagation (EuCAP),* 2912–2916. Paris: European Association on Antennas and Propagation, 2017.

[4] Angelov, A., A. Robertson, R. Murray-Smith, and F. Fioranelli. "Practical Classification of Different Moving Targets Using Automotive Radar and Deep Neural Networks." *IET Radar, Sonar & Navigation*. Vol. 12, Number 10, 2017, pp. 1082–1089.

[5] Parashar, K. N., M. C. Oveneke, M. Rykunov, H. Sahli, and A. Bourdoux. "Micro-Doppler Feature Extraction Using Convolutional Auto-Encoders for Low Latency Target Classification." In *2017 IEEE Radar Conference (RadarConf)*, 1739–1744. Seattle: IEEE, 2017.

See Also

`batchNormalizationLayer` | `classify` | `convolution2dLayer` | `layerGraph` | `trainNetwork` | `trainingOptions`

Related Examples

- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Deep Learning in MATLAB” on page 1-2

Label QRS Complexes and R Peaks of ECG Signals Using Deep Network

This example shows how to use custom autolabeling functions in **Signal Labeler** to label QRS complexes and R peaks of electrocardiogram (ECG) signals. One custom function uses a previously trained recurrent deep learning network to identify and locate the QRS complexes. Another custom function uses a simple peak finder to locate the R peaks. In this example, the network labels the QRS complexes of two signals that are completely independent of the network training and testing process.

The QRS complex, which consists of three deflections in the ECG waveform, reflects the depolarization of the right and left ventricles of the heart. The QRS is also the highest-amplitude segment of the human heartbeat. Study of the QRS complex can help assess the overall health of a person's heart and the presence of abnormalities [1 on page 11-0]. In particular, by locating R peaks within the QRS complexes and looking at the time intervals between consecutive peaks, a diagnostician can compute the heart-rate variability of a patient and detect cardiac arrhythmia.

The deep learning network in this example was introduced in “Waveform Segmentation Using Deep Learning” (Signal Processing Toolbox), where it was trained using ECG signals from the publicly available QT Database [2 on page 11-0] [3 on page 11-0]. The data consists of roughly 15 minutes of ECG recordings from a total of 105 patients, sampled at 250 Hz. To obtain each recording, the examiners placed two electrodes on different locations on a patient's chest, which resulted in a two-channel signal. The database provides signal region labels generated by an automated expert system [1 on page 11-0]. The added labels make it possible to use the data to train a deep learning network.

Load, Resample, and Import Data into Signal Labeler

The signals labeled in this example are from the MIT-BIH Arrhythmia Database [4 on page 11-0]. Each signal in the database was irregularly sampled at a mean rate of 360 Hz and was annotated by two cardiologists, allowing for verification of the results.

Load two MIT database signals, corresponding to records 200 and 203. Resample the signals to a uniform grid with a sample time of 1/250 second, which corresponds to the nominal sample rate of the QT Database data.

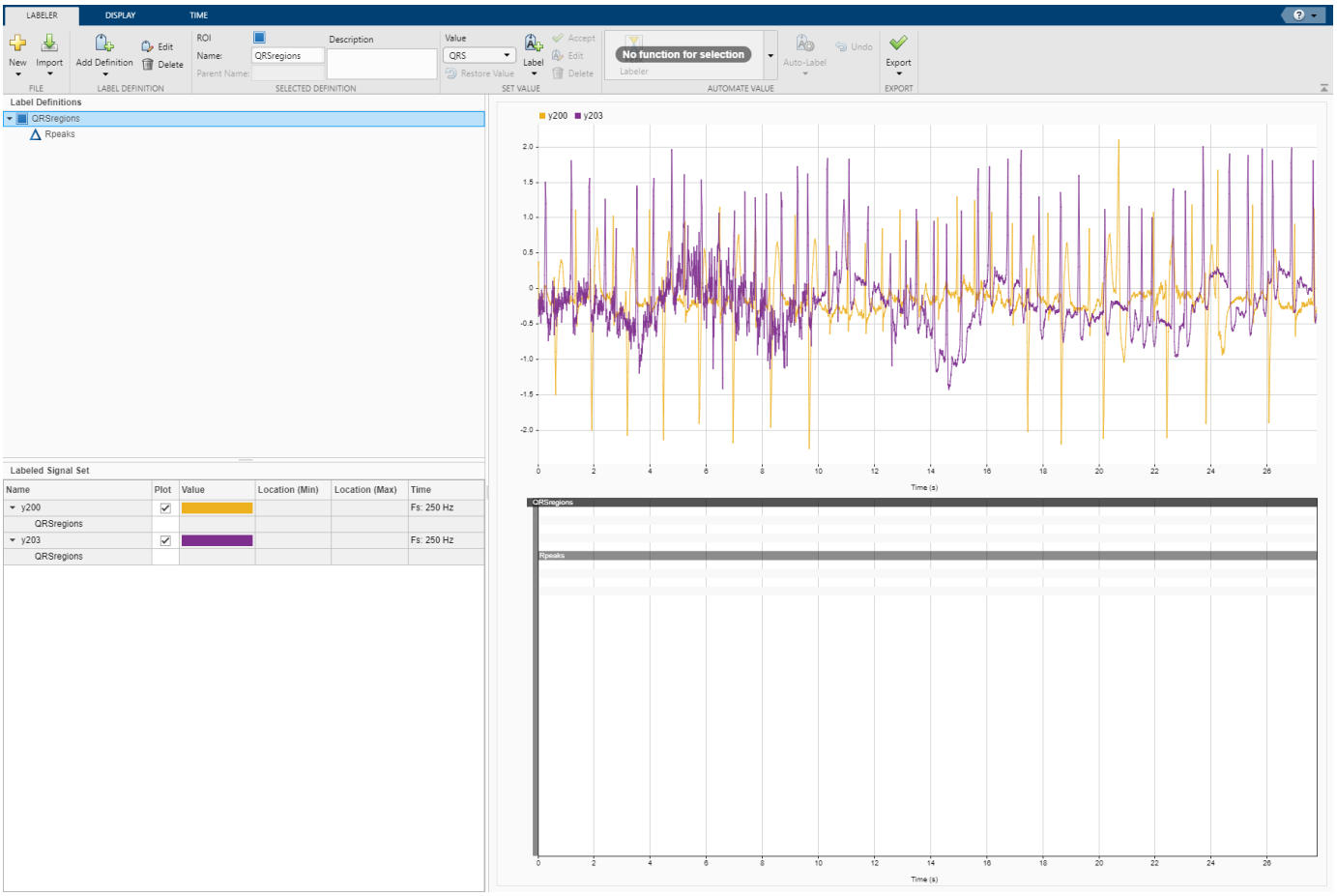
```
load mit200
y200 = resample(ecgsig,tm,250);

load mit203
y203 = resample(ecgsig,tm,250);
```

Open Signal Labeler. On the **Labeler** tab, click **Import ▼** and select **Members From Workspace**. In the dialog box, select the signals `y200` and `y203`. Add time information: Select `Time` from the drop-down list and specify a sample rate of 250 Hz. Click **Import** and close the dialog box. The signals appear in the **Labeled Signal Set** browser. Plot the signals by selecting the check boxes next to their names.

Define Labels

Define labels to attach to the signals.

1. Define a categorical region-of-interest (ROI) label for the QRS complexes. Click **Add Definition** on the **Labeler** tab. Specify the **Label Name** as `QRSregions`, select a **Label Type** of `ROI`, enter the **Data Type** as `categorical`, and add two **Categories,** `QRS` and `n/a`, each on its own line.
2. Define a sublabel of `QRSregions` as a numerical point label for the R peaks. Click **QRSregions** in the **Label Definitions** browser to select it. Click **Add Definition ▼** and select `Add sublabel definition`. Specify the **Label Name** as `Rpeaks`, select a **LabelType** of `Point`, and enter the **Data Type** as `numeric`.

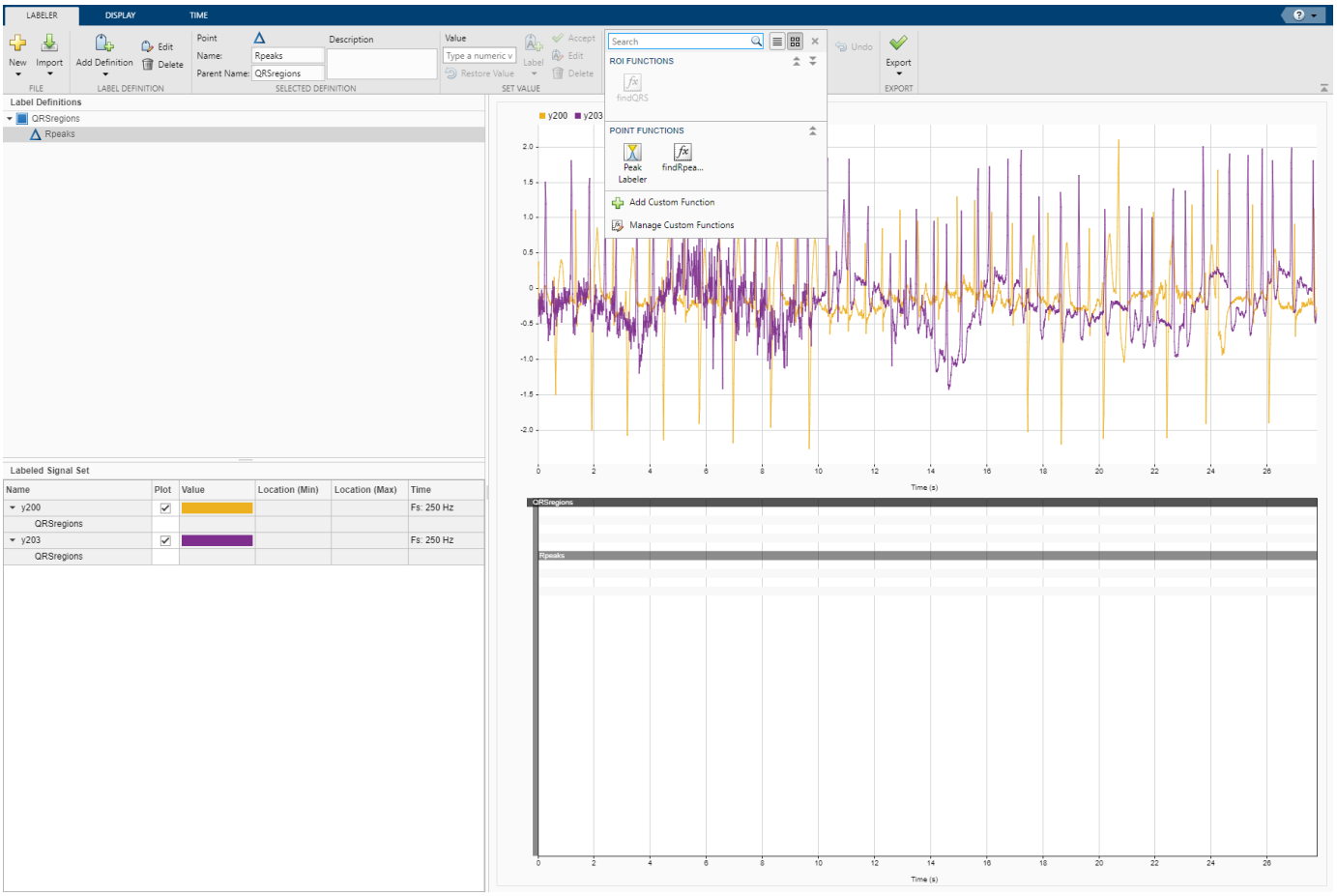


Create Custom Autolabeling Functions

Create two “Custom Labeling Functions” (Signal Processing Toolbox), one to locate and label the QRS complexes and another to locate and label the R peak within each QRS complex. (Code for the `findQRS` on page 11-0 , `computeFSST` on page 11-0 , `p2qrs` on page 11-0 , and `findRpeaks` on page 11-0 functions appears later in the example.) To create each function, in the **Labeler** tab, click **Automate Value ▼** and select **Add Custom Function**. **Signal Labeler** shows a dialog box asking for the name, description, and label type of the function.

1. For the function that locates the QRS complexes, enter `findQRS` in the **Name** field and select `ROI` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.
2. For the function that locates the R peaks, enter `findRpeaks` in the **Name** field and select `Point` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

If you already have written the functions, and the functions are in the current folder or in the MATLAB® path, **Signal Labeler** adds the functions to the gallery. If you have not written the functions, **Signal Labeler** opens blank templates in the Editor for you to type or paste the code. Save the files. Once you save the files, the functions appear in the gallery.



Label QRS Complexes and R Peaks

Find and label the QRS complexes of the input signals.

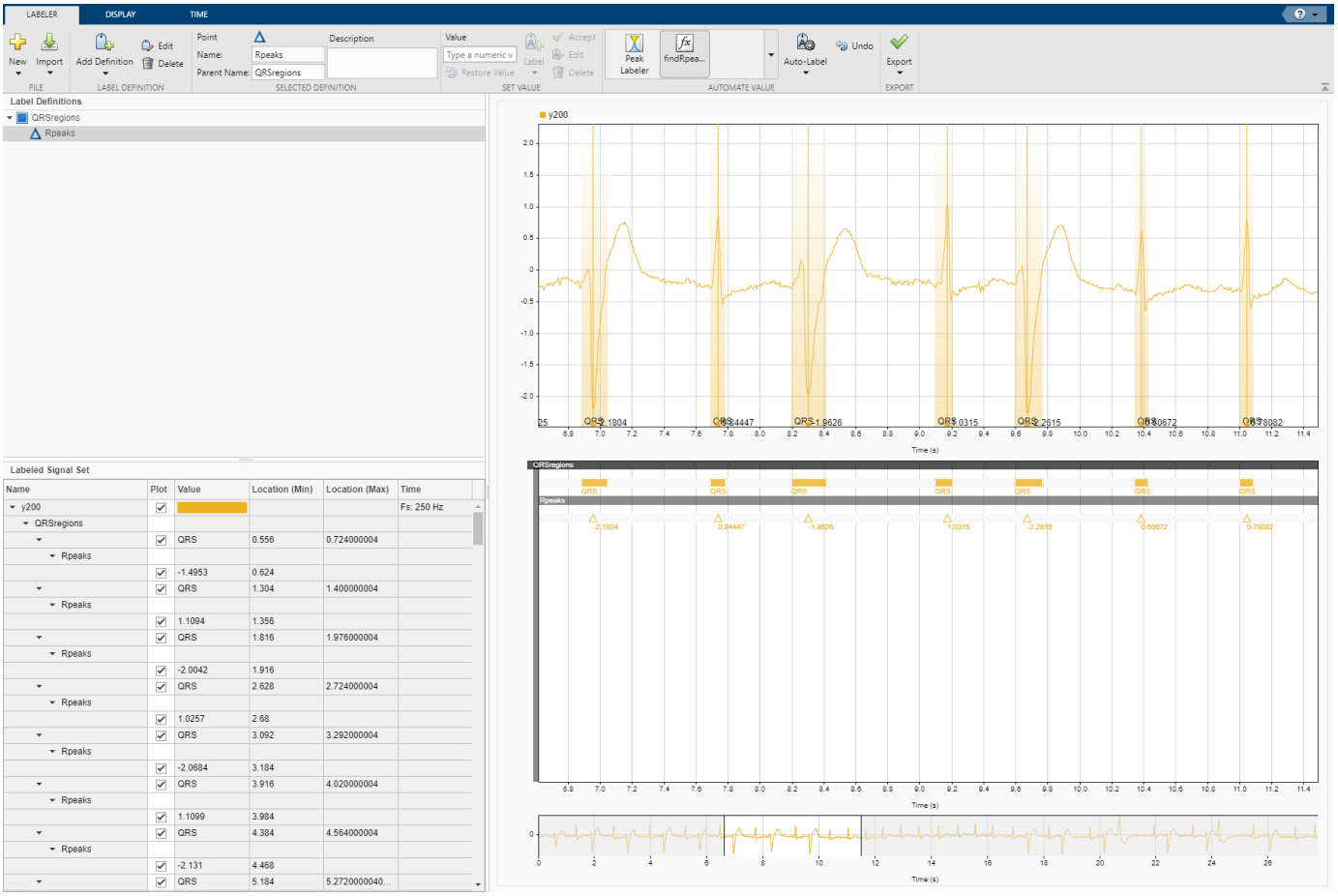
1. In the **Labeled Signal Set** browser, select the check box next to `y200`.
2. Select `QRSregions` in the **Label Definitions** browser.
3. In the **Automate Value** gallery, select `findQRS`.
4. Click **Auto-Label** and select `Auto-Label Signals`. Click **OK** in the dialog box that appears.

Signal Labeler locates and labels the QRS complexes for all signals, but displays labels only for the signals whose check boxes are selected. The QRS complexes appear as shaded regions in the plot and in the label viewer axes. Activate the panner by clicking **Panner** on the **Display** tab and zoom in on a region of the labeled signal.

Find and label the R peaks corresponding to the QRS complexes.

1. Select `Rpeaks` in the **Label Definitions** browser.
2. Go back to the **Labeler** tab. In the **Automate Value** gallery, select `findRpeaks`.
3. Click **Auto-Label** and select `Auto-Label Signals`. Click **OK** in the dialog box that appears.

The labels and their numeric values appear in the plot and in the label viewer axes.



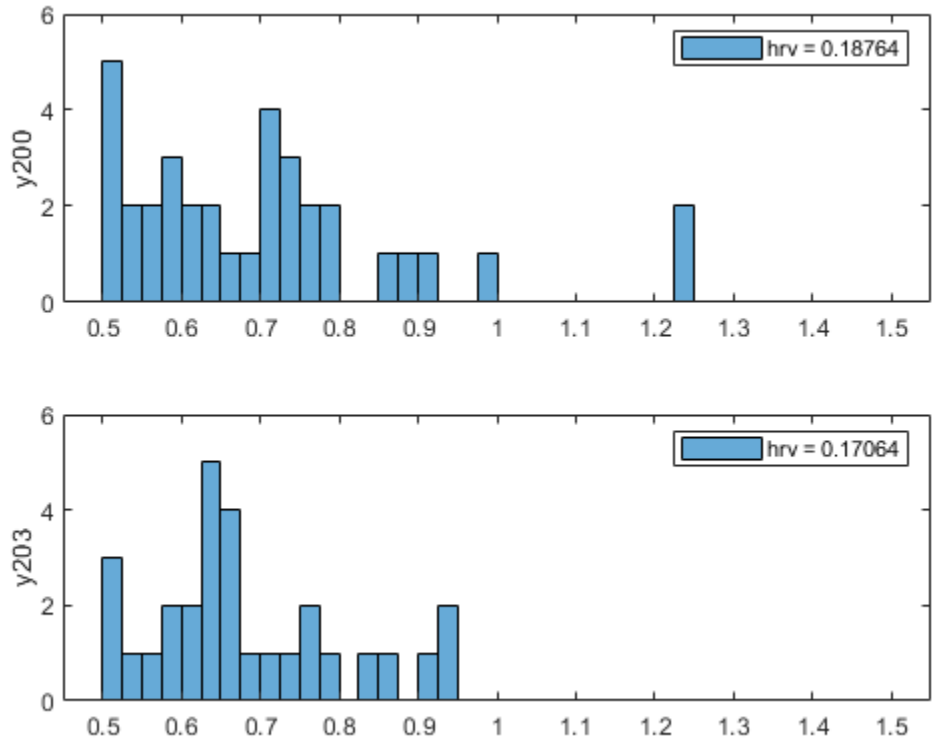
Export Labeled Signals and Compute Heart-Rate Variability

Export the labeled signals to compare the heart-rate variability for each patient. On the **Labeler** tab, click **Export** ▼ and select `Labeled Signal Set To File`. In the dialog box that appears, give the name `HeartRates.mat` to the labeled signal set and add an optional short description. Click **Export**.

Go back to the MATLAB® Command Window. Load the labeled signal set. For each signal in the set, compute the heart-rate variability as the standard deviation of the time differences between consecutive heartbeats. Plot a histogram of the differences and display the heart-rate variability.

```
load HeartRates

nms = getMemberNames(heartrates);

for k = 1:heartrates.NumMembers

    v = getLabelValues(heartrates,k,{'QRSregions','Rpeaks'});

    hr = diff(cellfun(@(x)x.Location,v));

    subplot(2,1,k)
    histogram(hr,0.5:.025:1.5)
    legend(['hrv = ' num2str(std(hr))])
    ylabel(nms(k))
```

```
    ylim([0 6])
end
```



`findQRS` Function: Find QRS Complexes

The `findQRS` function finds and labels the QRS complexes of the input signals.

The function uses two auxiliary functions, `computeFSST` and `p2qrs`. (Code for both auxiliary functions appears later in the example.) You can either store the functions in separate files in the same directory or nest them inside the `findQRS` function by inserting them before the final `end` statement.

Between calls to `computeFSST` and `p2qrs`, `findQRS` uses the `classify` function and the trained deep learning network `net` to identify the QRS regions. Before calling `classify`, `findQRS` converts the data into the format expected by `net`, as explained in "Waveform Segmentation Using Deep Learning" (Signal Processing Toolbox):

- Each signal must be sampled at 250 Hz and partitioned into a stack of 2-by-*N* cell arrays, where each row corresponds to a channel and *N* is a multiple of 5000. The actual partitioning and stacking is done in the `computeFSST` function.
- Each of the resampled MIT signals has 6945 samples, a number that is not a multiple of 5000. To keep all the data in each signal, pad the signal with random numbers. Later in the process, the `p2qrs` function labels the random numbers as not belonging to the QRS complexes and discards them.

```
function [labelVals,labelLocs] = findQRS(x,t,parentLabelVal,parentLabelLoc,varargin)
% This is a template for creating a custom function for automated labeling
%
%  x is a matrix where each column contains data corresponding to a
%  channel. If the channels have different lengths, then x is a cell array
%  of column vectors.
%
%  t is a matrix where each column contains time corresponding to a
%  channel. If the channels have different lengths, then t is a cell array
%  of column vectors.
%
%  parentLabelVal is the parent label value associated with the output
%  sublabel or empty when output is not a sublabel.
%  parentLabelLoc contains an empty vector when the parent label is an
%  attribute, a vector of ROI limits when parent label is an ROI or a point
%  location when parent label is a point.
%
%  labelVals must be a column vector with numeric, logical or string output
%  values.
%  labelLocs must be an empty vector when output labels are attributes, a
%  two column matrix of ROI limits when output labels are ROIs, or a column
%  vector of point locations when output labels are points.

labelVals = [];
labelLocs = [];

Fs = 250;

load('trainedQTSegmentationNetwork','net')

for kj = 1:size(x,2)

    sig = x(:,kj);

    % Create 10000-sample signal expected by the deep network

    sig = [sig;randn(10000-length(sig),1)/100]';

    % Resize input and compute synchrosqueezed Fourier transforms

    mitFSST = computeFSST(sig,Fs);

    % Use trained network to predict which points belong to QRS regions

    netPreds = classify(net,mitFSST,'MiniBatchSize',50);

    % Convert stack of cell arrays into a single vector

    Location = [1:length(netPreds{1}) length(netPreds{1})+(1:length(netPreds{2}))]';
    Value = [netPreds{1} netPreds{2}]';

    % Label QRS complexes as regions of interest and discard non-QRS data

    [Locs,Vals] = p2qrs(table(Location,Value));

    labelVals = [labelVals;Vals];
    labelLocs = [labelLocs;Locs/Fs];
```

```
end

% Insert computeFSST and p2qrs here if you want to nest them inside
% queryQRS instead of including them as separate functions in the folder.

end
```

`computeFSST` Function: Resize Input and Compute Synchrosqueezed Fourier Transforms

This function reshapes the input data into the form expected by `net` and then uses the `fsst` function to compute the Fourier synchrosqueezed transform (FSST) of the input. In "Waveform Segmentation Using Deep Learning" (Signal Processing Toolbox), the network performs best when you use a time-frequency map of each training or testing signal as the input. The FSST results in a set of features particularly useful for recurrent networks because the transform has the same time resolution as the original input. The function:

- Specifies a Kaiser window of length 128 to provide adequate frequency resolution.
- Extracts data over the frequency range from 0.5 Hz to 40 Hz.
- Subtracts the mean of each signal and divides by the standard deviation.
- Treats the real and imaginary parts of the FSST as separate features.

```
function signalsFsst = computeFSST(xd,Fs)

targetLength = 5000;
signalsOut = {};

for sig_idx = 1:size(xd,1)

    current_sig = xd(sig_idx,:)';

    % Compute the number of targetLength-sample chunks in the signal

    numSigs = floor(length(current_sig)/targetLength);

    % Truncate to a multiple of targetLength

    current_sig = current_sig(1:numSigs*targetLength);

    % Create a matrix with as many columns as targetLength signals

    xM = reshape(current_sig,targetLength,numSigs);

    % Vertically concatenate into cell arrays

    signalsOut = [signalsOut; mat2cell(xM.',ones(numSigs,1))];

end

signalsFsst = cell(size(signalsOut));

for idx = 1:length(signalsOut)

   [s,f] = fsst(signalsOut{idx},Fs,kaiser(128));

   % Extract data over the frequency range from 0.5 Hz to 40 Hz
```

```
    f_indices = (f > 0.5) & (f < 40);
    signalsFsst{idx}= [real(s(f_indices,:)); imag(s(f_indices,:))];

    % Subtract the mean and divide by the standard deviation

    signalsFsst{idx} = (signalsFsst{idx}-mean(signalsFsst{idx},2)) ...
        ./std(signalsFsst{idx},[],2);

end

end
```

`p2qrs` Function: Label QRS Complexes as Regions of Interest

The deep network outputs a categorical array that labels every point of the input signal as belonging to a P region, a QRS complex, a T region, or to none of those. This function converts those point labels to QRS region-of-interest labels.

- To perform the conversion, the function assigns integer numerical values to the categories and uses the `findchangepts` function to find the points where the numerical array changes value.
- Each of those changepoints is the left endpoint of a categorical region, and the point that precedes it in the array is the right endpoint of the preceding region.
- The algorithm adds `1e-6` to the right endpoints to prevent one-sample regions from having zero duration.
- The `df` parameter selects as regions of interest only those QRS complexes whose duration is greater than `df` samples.

```
function [locs,vals] = p2qrs(k)

fc = 1e-6;
df = 20;

ctgs = categories(k.Value);
levs = 1:length(ctgs);
for jk = levs
   cat2num(k.Value == ctgs{jk}) = levs(jk);
end
chpt = findchangepts(cat2num,'MaxNumChanges',length(cat2num));
locs = [[1;chpt'] [chpt'-1;length(cat2num)]+fc];

vals = categorical(cat2num(locs(:,1))',levs,ctgs);
locs = locs+round(k.Location(1))-1;

qrs = find(vals=='QRS' & diff(locs,[],2)>df);

vals = categorical(string(vals(qrs)),["QRS" "n/a"]);

locs = locs(qrs,:);

end
```

`findRpeaks` Function: Find R Peaks

This function locates the most prominent peak of the QRS regions of interest found by `findQRS`. The function applies the MATLAB® `islocalmax` function to the absolute value of the signal in the intervals located by `findQRS`.

```
function [labelVals,labelLocs] = findRpeaks(x,t,parentLabelVal,parentLabelLoc,varargin)

labelVals = zeros(size(parentLabelLoc,1),1);
labelLocs = zeros(size(parentLabelLoc,1),1);

for kj = 1:size(parentLabelLoc,1)
    tvals = t>=parentLabelLoc(kj,1) & t<=parentLabelLoc(kj,2);
    ti = t(tvals);
    xi = x(tvals);
    lc = islocalmax(abs(xi),'MaxNumExtrema',1);
    labelVals(kj) = xi(lc);
    labelLocs(kj) = ti(lc);
end

end
```

References

[1] 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.

[2] 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].

[3] 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.

[4] Moody, George B., and Roger G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20, No. 3, May–June 2001, pp. 45–50.

See Also

`trainNetwork` | `trainingOptions`

More About

- “Long Short-Term Memory Networks” on page 1-53
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34

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 11-0] :

- 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 11-0]. 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 11-0] [4 on page 11-0]. 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 11-0]. 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/QTDatabaseECGData.zip';
datasetFolder = fullfile(tempdir,'QTDataset');
zipFile = fullfile(tempdir,'QTDatabaseECGData.zip');
if ~exist(datasetFolder,'dir')
     websave(zipFile,dataURL);
end
```

```
% Unzip the data
unzip(zipFile,tempdir)
```

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 }
```

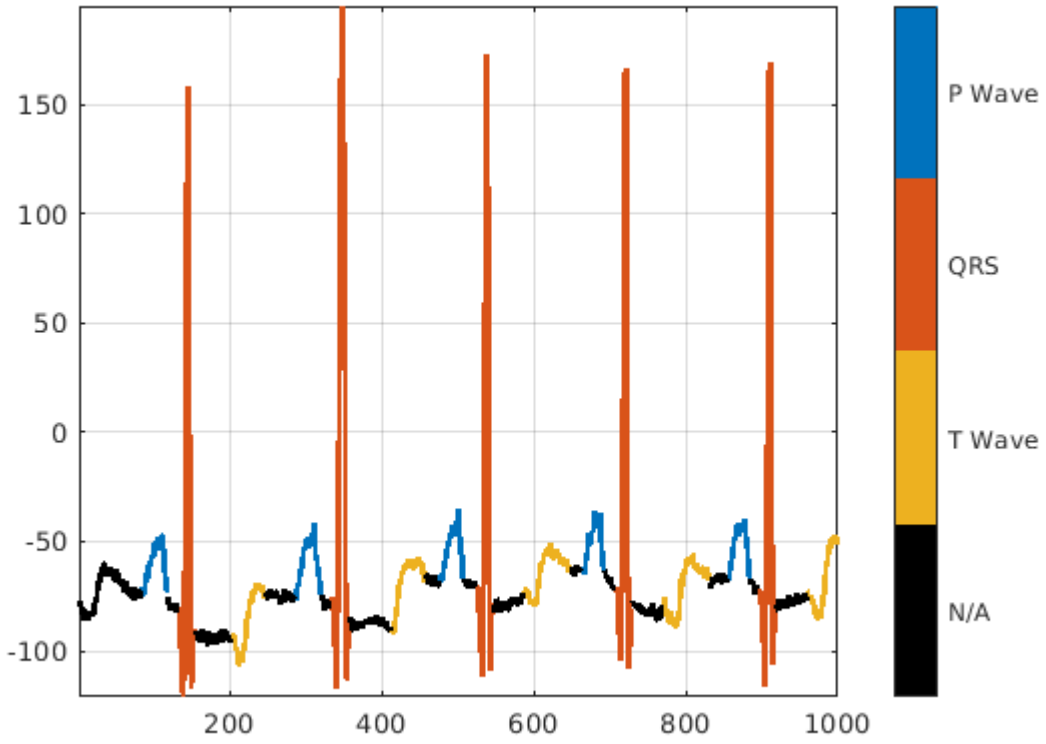
Look at the first few rows of the region labels table and observe that each row contains the region limit indices and the region class value (P, T, or QRS).

```
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 the `displayWaveformLabels` helper function.

```
displayWaveformLabels(data,true,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);
```

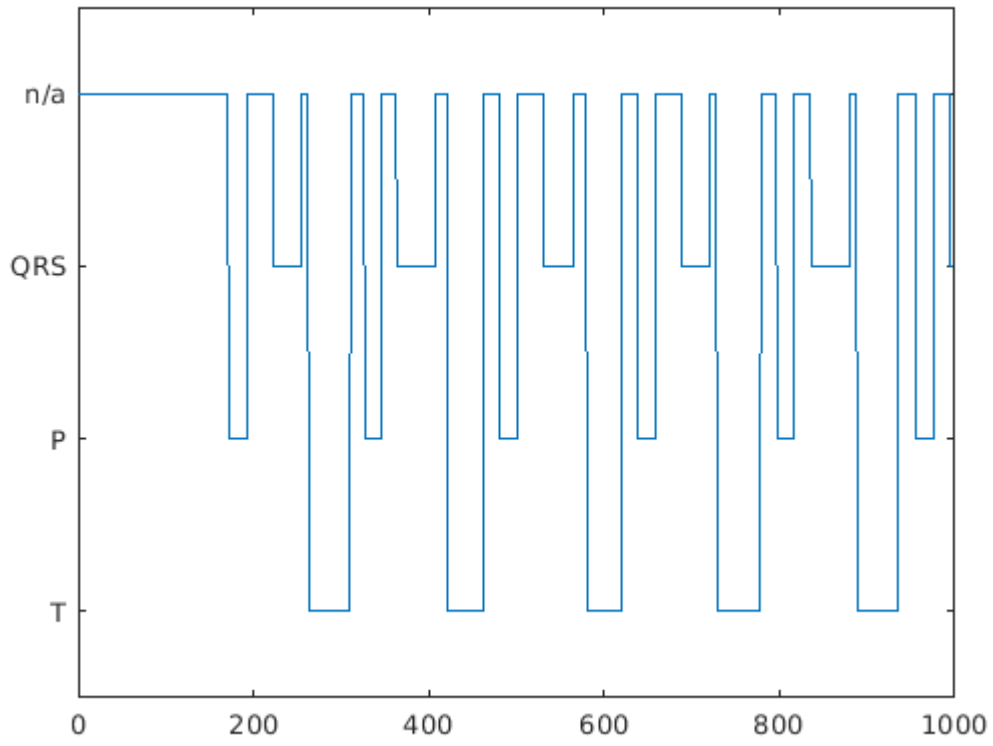
In this segmentation problem, the input to the LSTM network is an ECG signal and the output is a sequence or mask of labels with the same length as the input signal. The network task is to label each signal sample with the name of the region it belongs to. For this reason, it is necessary to transform the region labels on the dataset to sequences containing one label per signal sample. Use a transformed datastore and the `roi2mask` helper function to transform the labels. 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. The `roi2mask` function adds a label category, `n/a`, to label samples that do not belong to any region of interest.

```
trainDs = transform(trainDs, @roi2mask);
testDs = transform(testDs, @roi2mask);

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}
```

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: 0% complete
Evaluation 0% complete

- Pass 1 of 1: Completed in 28 sec
Evaluation completed in 28 sec
```

```
 trainData(1,:)
```

```
ans=1×2 cell array
    {1×5000 double}    {1×5000 categorical}
```

```
 testData = gather(tallTestSet);
```

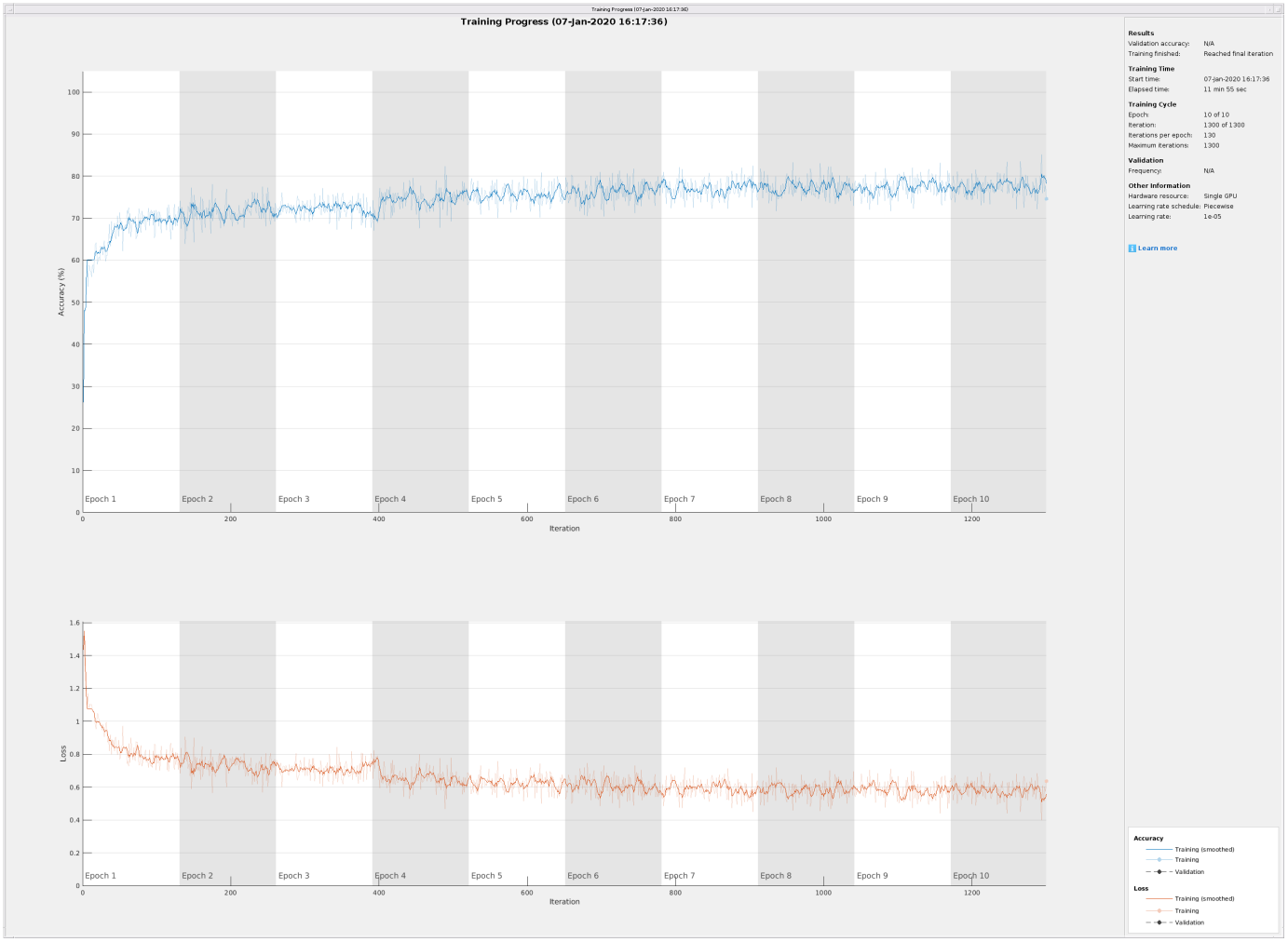
```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 9.8 sec
Evaluation completed in 10 sec
```

Train Network

Use the `trainNetwork` command to train the LSTM network. Due to the large size of the dataset, this process may take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB automatically uses the GPU for training. Otherwise, it uses the CPU.

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 N/A.

```
net = trainNetwork(trainData(:,1),trainData(:,2),layers,options);
```



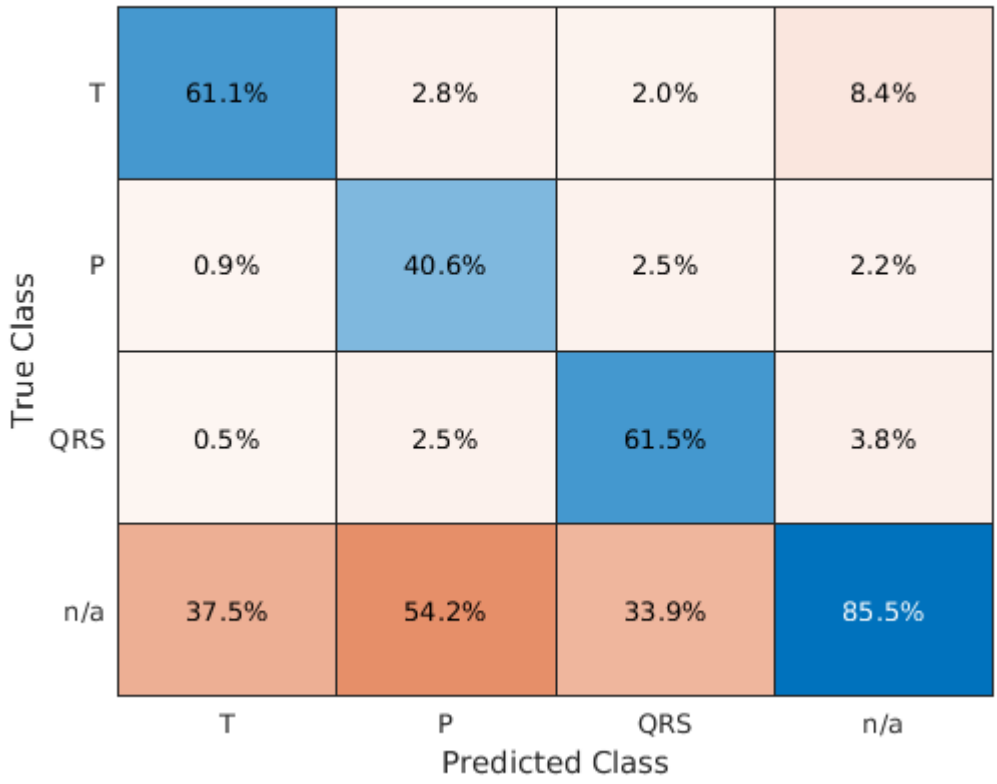
Classify Testing Data

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(net,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 11-0].

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 11-0].

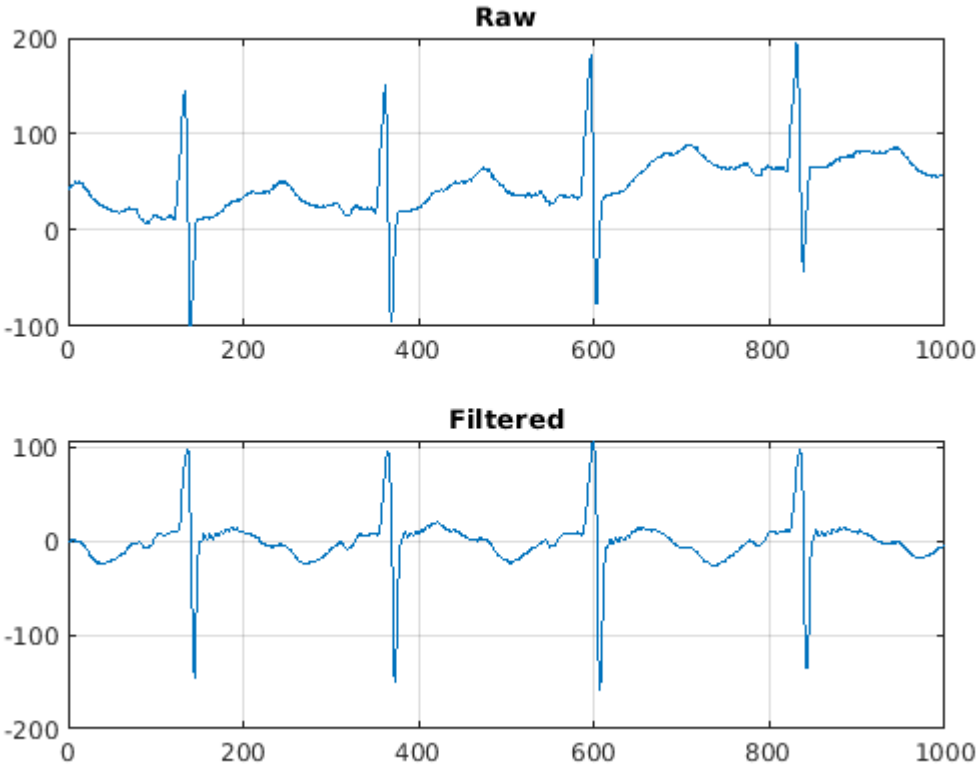
Design a bandpass filter with passband frequency range of [0.5, 40] Hz to remove the wander and any high frequency noise. Removing these components improves the LSTM training because the network does not learn irrelevant features. Use `cellfun` on the tall data cell arrays to filter the dataset in parallel.

```
% 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 19 sec
Evaluation completed in 19 sec

trainLabels = gather(tallTrainSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 19 sec
Evaluation completed in 19 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 8.5 sec
Evaluation completed in 8.6 sec

testLabels = gather(tallTestSet(:,2));

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 8.3 sec
Evaluation completed in 8.5 sec
```

Plot the raw and filtered signals for a typical case.

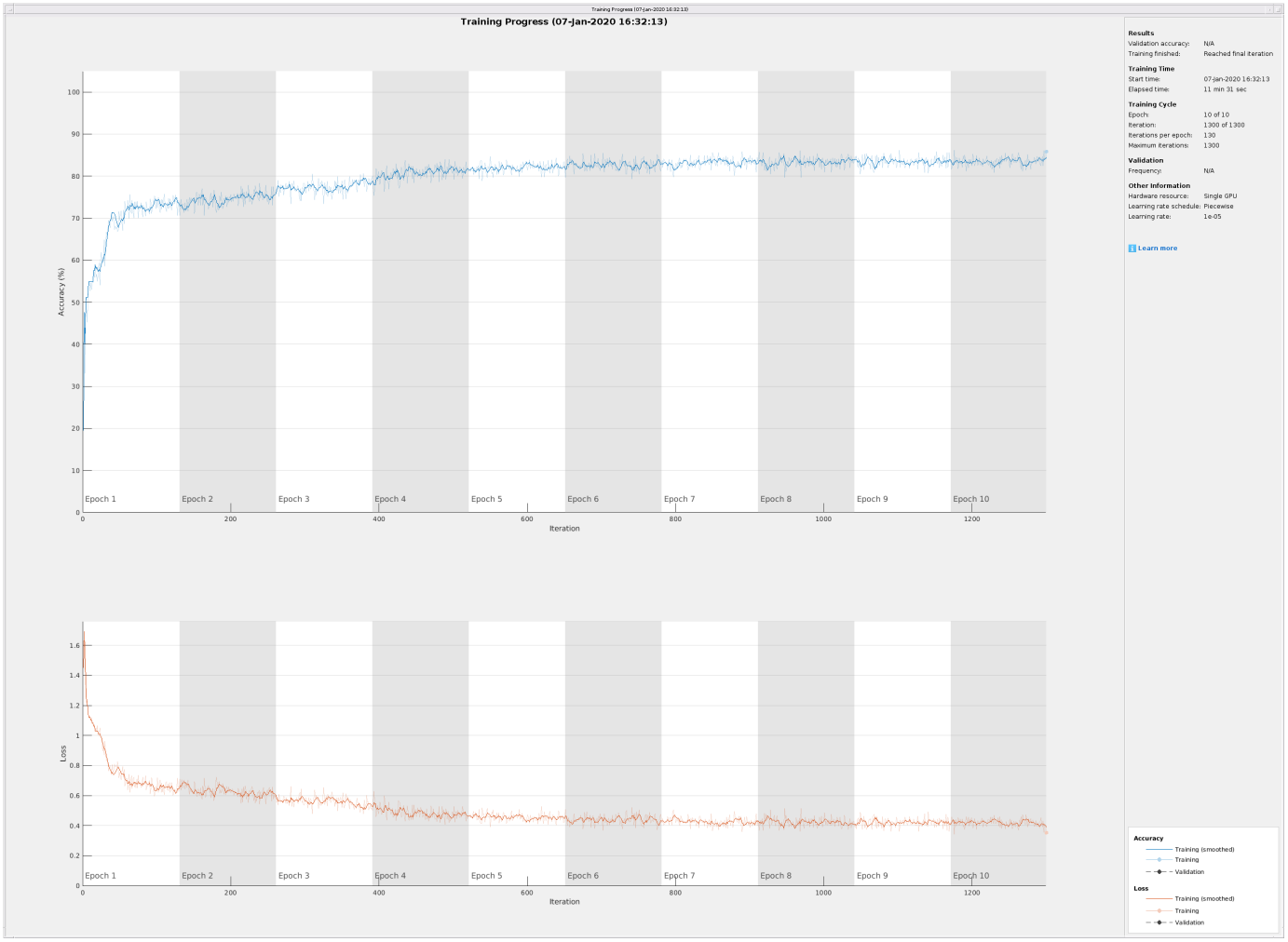
```
trainData = gather(tallTrainSet);

Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 19 sec
Evaluation completed in 19 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
```



Train Network with Filtered ECG Signals

Train the LSTM network on the filtered ECG signals using the same network architecture as before.

```
filteredNet = trainNetwork(filteredTrainSignals,trainLabels,layers,options);
```



Preprocessing the signals improves the training accuracy to better than 80%.

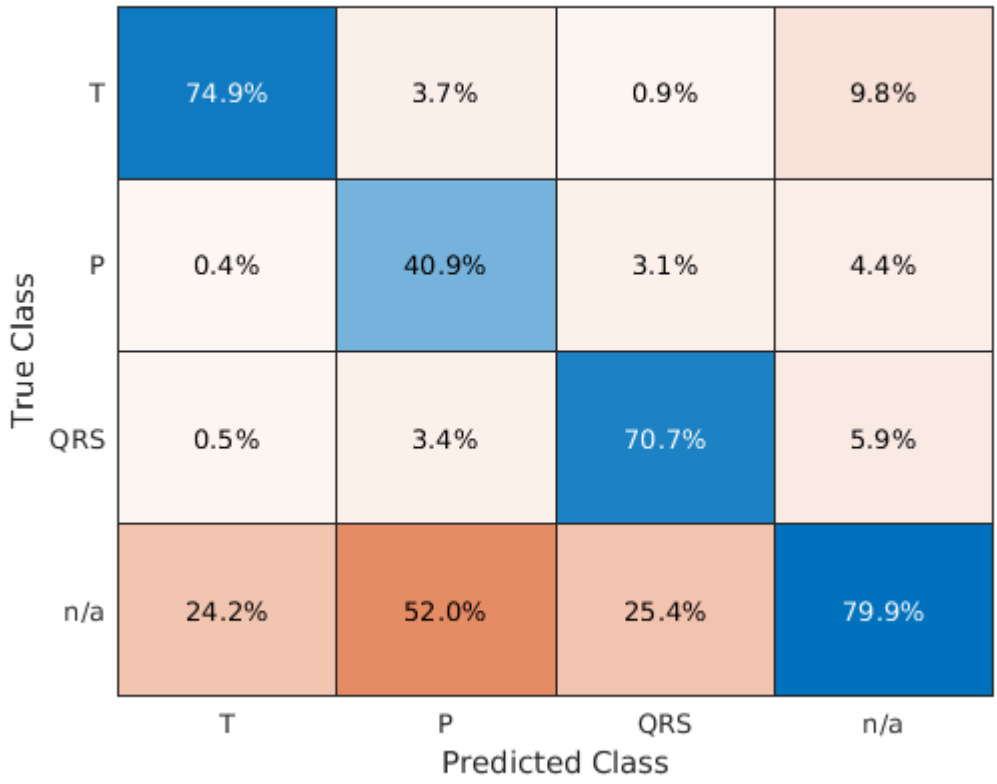
Classify Filtered ECG Signals

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');
```



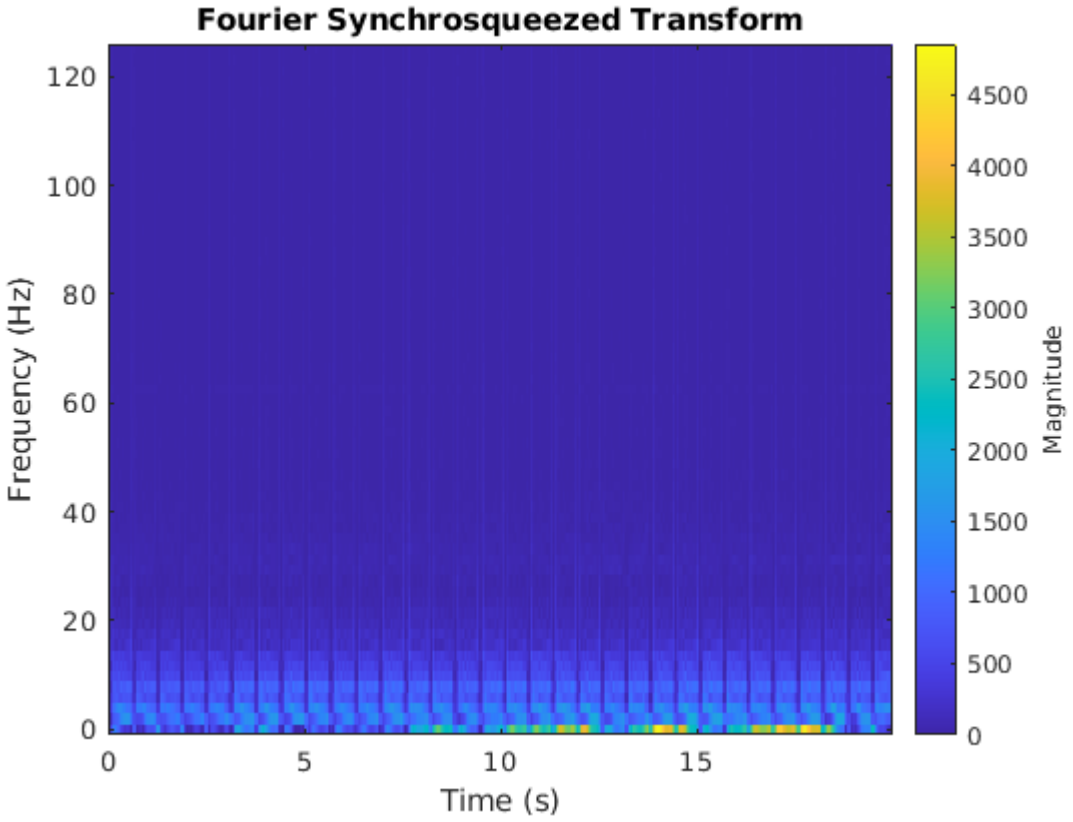
Simple preprocessing improves T-wave classification by about 15%, and QRS-complex classification by 10%. P-wave classification accuracy did not change.

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 11-0].

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 23 sec
Evaluation completed in 1 min 23 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
```

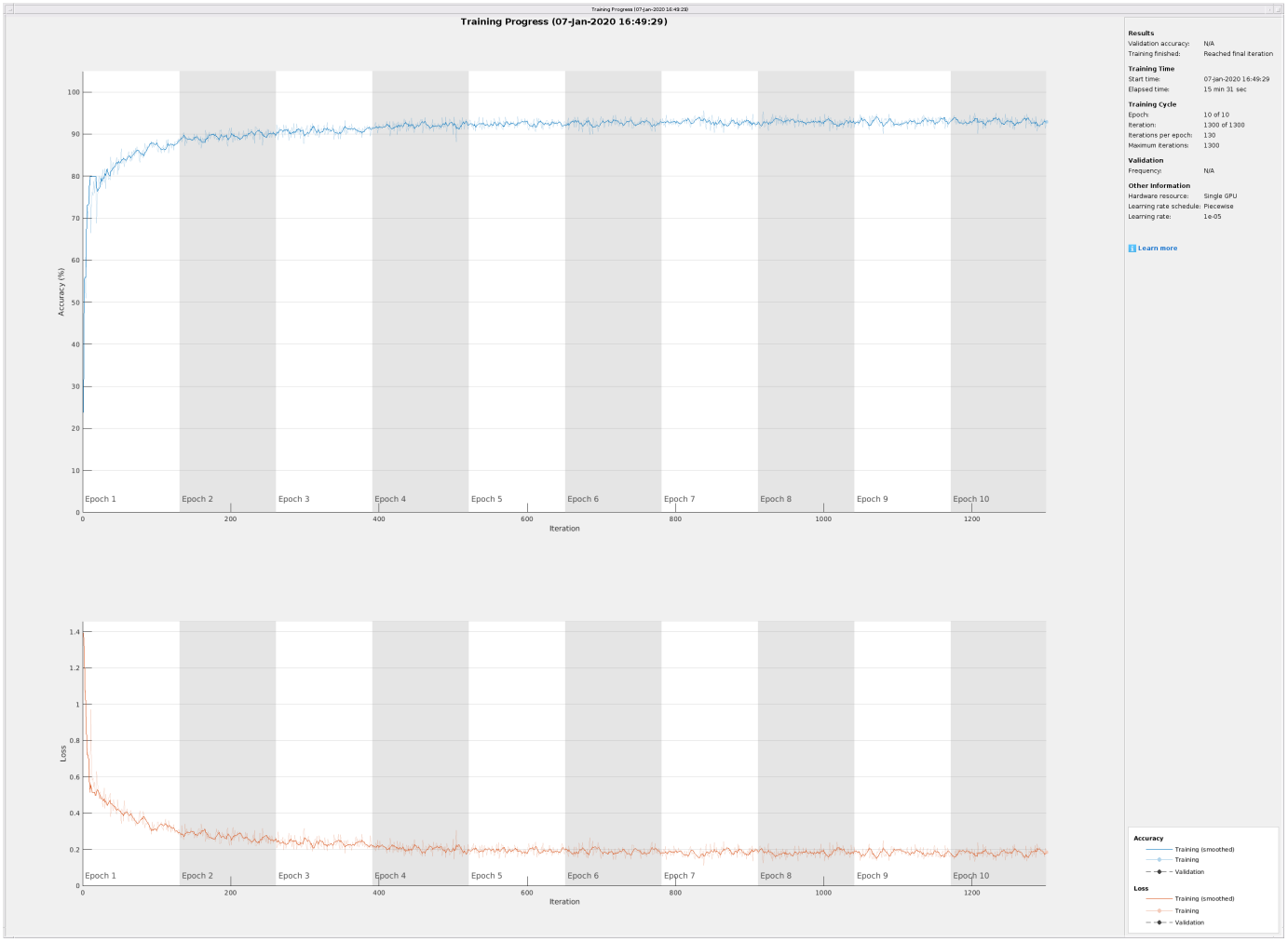
Specify a `sequenceInputLayer` of 40 input features. Keep the rest of the network parameters unchanged.

```
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.

```
fsstNet = trainNetwork(fsstTrainData(:,1),fsstTrainData(:,2),layers,options);
```



Using time-frequency features improves the training accuracy, which now exceeds 90%.

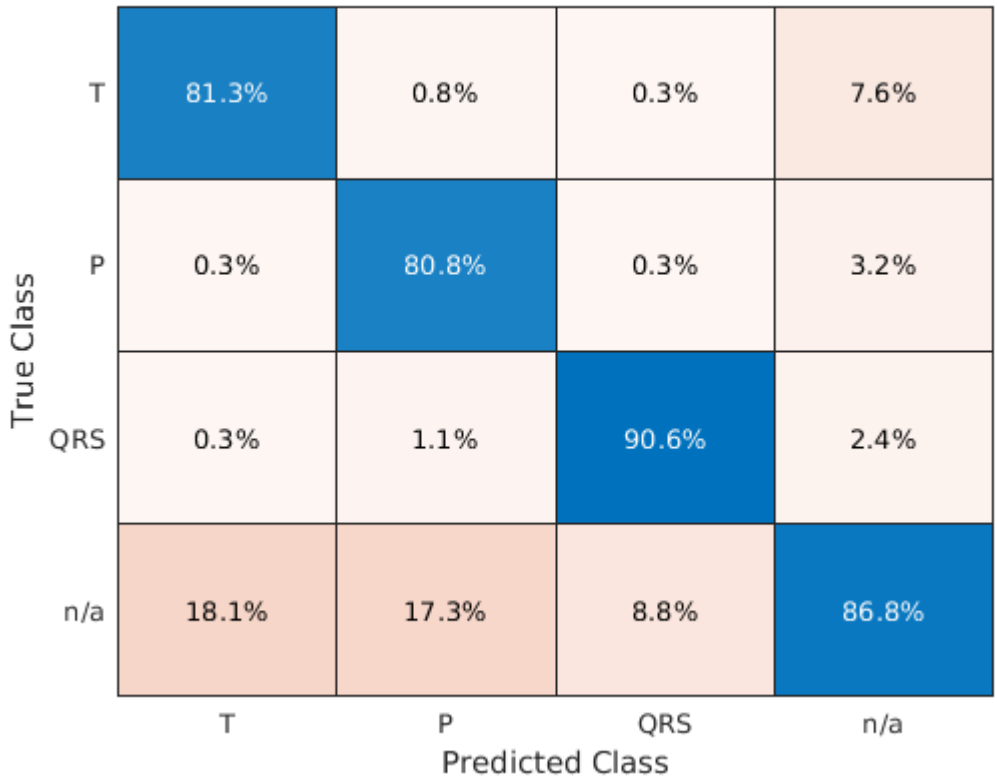
Classify Test Data with FSST

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');
```



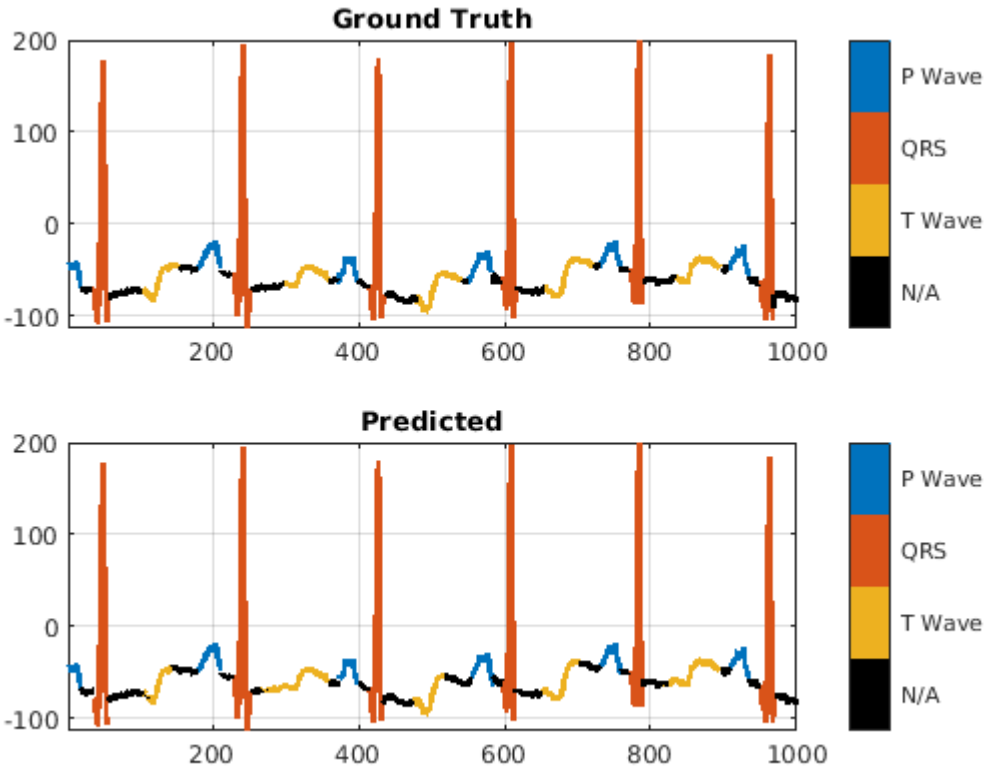
Using a time-frequency representation improves T-wave classification by about 20%, P-wave classification by about 40%, and QRS-complex classification by 30%, when compared to the raw data results.

Use the `displayWaveformLabels` helper function to compare the network prediction to the ground truth labels for a single ECG signal.

```
testData = gather(tall(testDs));
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 8.3 sec
Evaluation completed in 8.5 sec
```

```
figure
subplot(2,1,1)
displayWaveformLabels({testData{1,1}(3000:4000),testData{1,2}(3000:4000)},false)
title('Ground Truth')

subplot(2,1,2)
displayWaveformLabels({testData{1,1}(3000:4000),predFsstTest{1}(3000:4000)},false)
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` | `trainNetwork` | `trainingOptions`

More About

- “Long Short-Term Memory Networks” on page 1-53
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34

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 11-0 section.

```
load trainedModulationClassificationNetwork
trainedNet
```

```
trainedNet =
  SeriesNetwork with properties:

         Layers: [28x1 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` PAM4-modulate the bits
- `rcosdesign`: Design a square-root raised cosine pulse shaping filter
- `filter`: Pulse shape the symbols

- `comm.RicianChannel`: Apply Rician multipath channel
- `comm.PhaseFrequencyOffset`: Apply phase and/or frequency shift due to clock offset
- `interp1`: Apply timing drift due to clock offset
- `awgn`: 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);
```

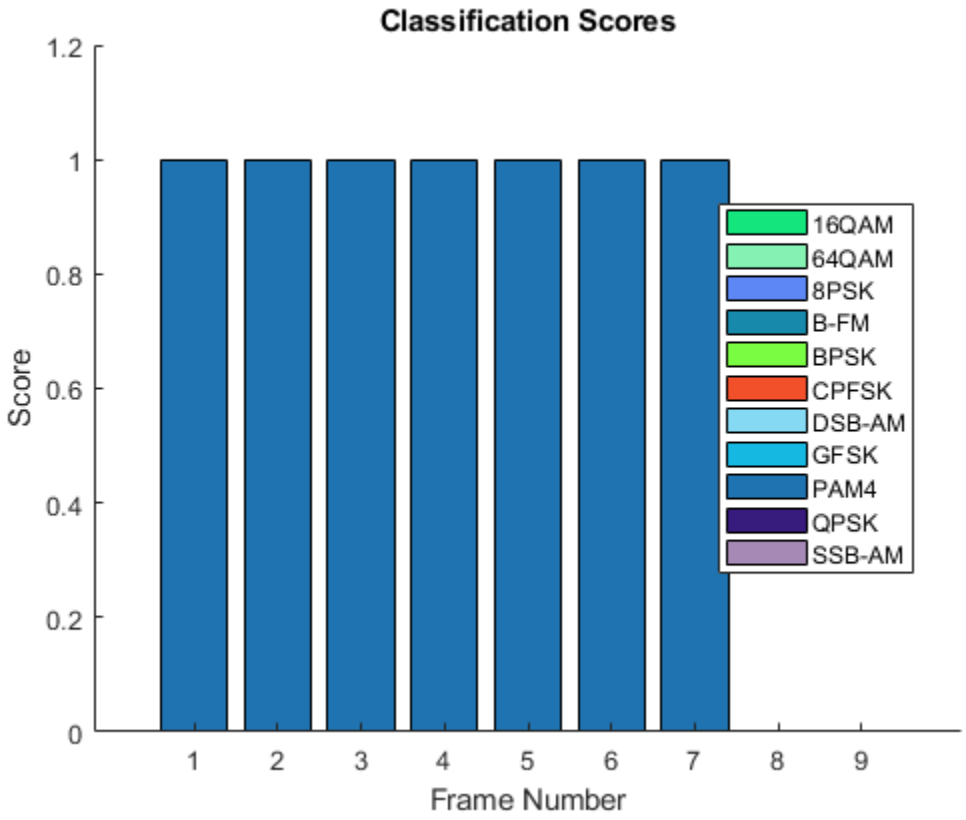
Return the classifier predictions, which are analogous to hard decisions. The network correctly identifies the frames as PAM4 frames. For details on the generation of the modulated signals, see helperModClassGetModulator function.

```
prediction1
```

```
prediction1 = 7x1 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 = ;
if trainNow == true
  numFramesPerModType = 10000;
else
  numFramesPerModType = 500;
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` function.

Rician Multipath

The channel passes the signals through a Rician multipath fading channel using the `comm.RicianChannel` 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. Implemented the channel using `comm.PhaseFrequencyOffset`.

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)
```

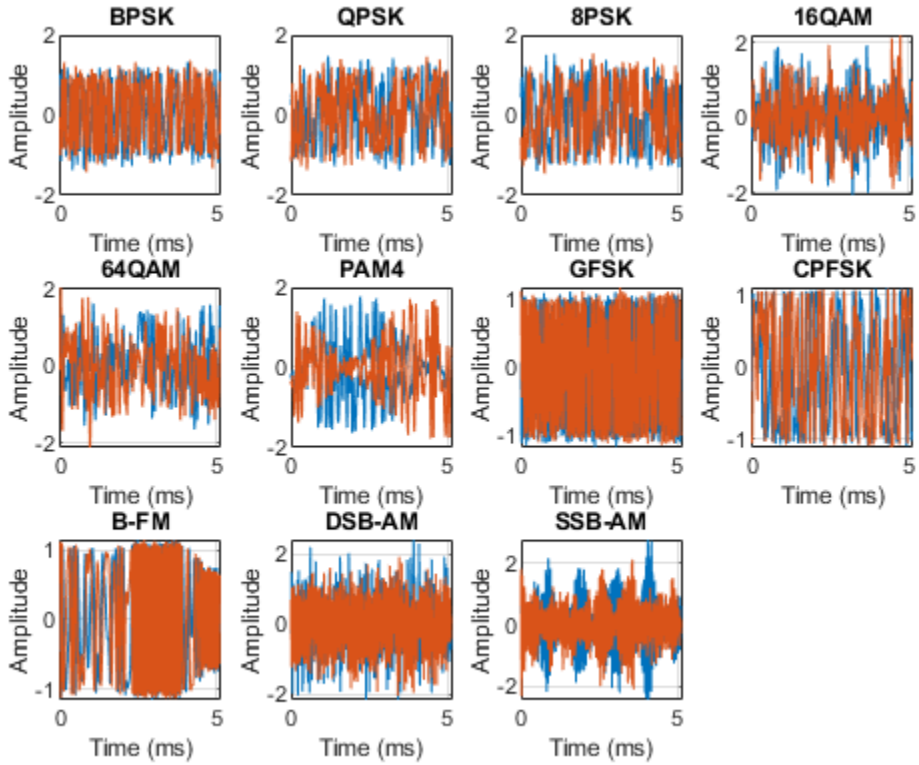
```
Data file directory is C:\TEMP\Bdoc20a_1326390_8984\ib9D0363\19\ModClassDataFiles
```

```
fileNameRoot = "frame";

% Check if data files exist
dataFilesExist = false;
if exist(dataDirectory,'dir')
  files = ls(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
    fprintf('%s - Generating %s frames\n', ...
      datestr(toc/86400,'HH:MM:SS'), 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
```

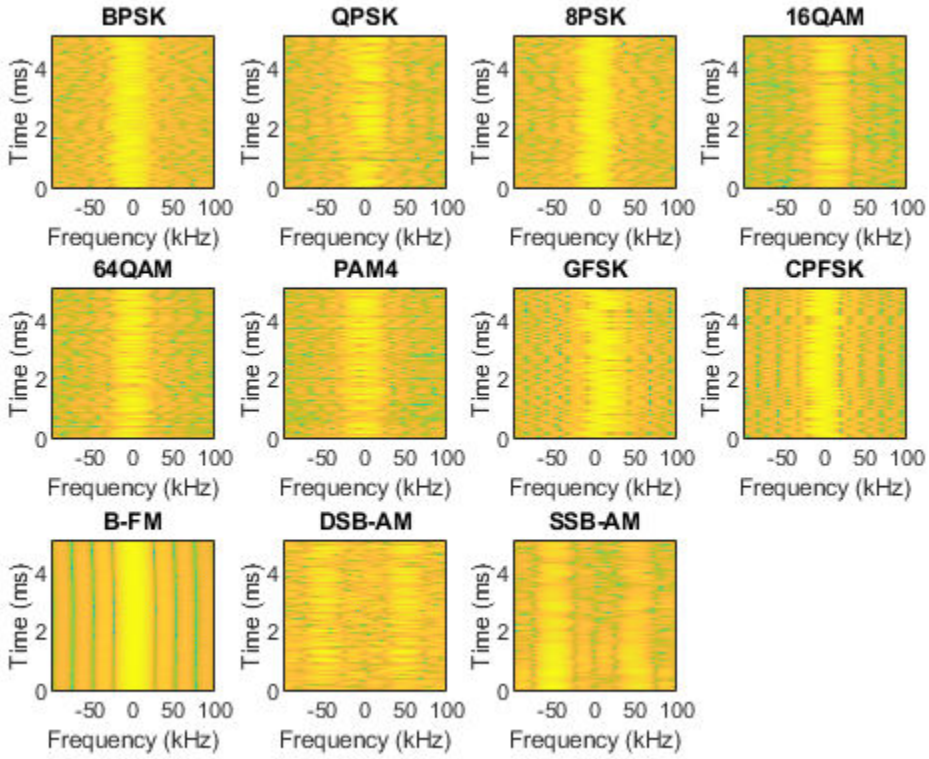
```
      % Digital modulation types use a center frequency of 902 MHz
      channel.CenterFrequency = 902e6;
    end

    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
00:00:08 - Generating QPSK frames
00:00:17 - Generating 8PSK frames
00:00:25 - Generating 16QAM frames
00:00:34 - Generating 64QAM frames
00:00:41 - Generating PAM4 frames
00:00:49 - Generating GFSK frames
00:00:57 - Generating CPFSK frames
00:01:04 - Generating B-FM frames
00:01:33 - Generating DSB-AM frames
00:01:41 - 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 the size [1xspfx2xN], where the first page (3rd dimension) is in-phase samples and the second page is quadrature samples. When the convolutional filters are of size [1xspf], 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: 12).
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 14 sec
- Pass 2 of 2: Completed in 13 sec
Evaluation completed in 31 sec
```

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 `tall` arrays to enable parallel processing of the transform functions, in case you have Parallel Computing Toolbox license. Since `gather` 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.

```
% Gather the training and validation frames into the memory
trainFramesTall = tall(transform(trainDSTrans, @helperModClassReadFrame));
rxTrainFrames = gather(trainFramesTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 11 sec
Evaluation completed in 11 sec
```

```
rxTrainFrames = cat(4, rxTrainFrames{:});
validFramesTall = tall(transform(validDSTrans, @helperModClassReadFrame));
rxValidFrames = gather(validFramesTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4.3 sec
Evaluation completed in 4.4 sec
```

```
rxValidFrames = cat(4, rxValidFrames{:});
```

```
% Gather the training and validation labels into the memory
trainLabelsTall = tall(transform(trainDSTrans, @helperModClassReadLabel));
rxTrainLabels = gather(trainLabelsTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 7.1 sec
- Pass 2 of 2: Completed in 9.6 sec
Evaluation completed in 18 sec
```

```
validLabelsTall = tall(transform(validDSTrans, @helperModClassReadLabel));
rxValidLabels = gather(validLabelsTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 4.1 sec
- Pass 2 of 2: Completed in 4.9 sec
Evaluation completed in 9.8 sec
```

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 1-45.

```
modClassNet = helperModClassCNN(modulationTypes,sps,spf);
```

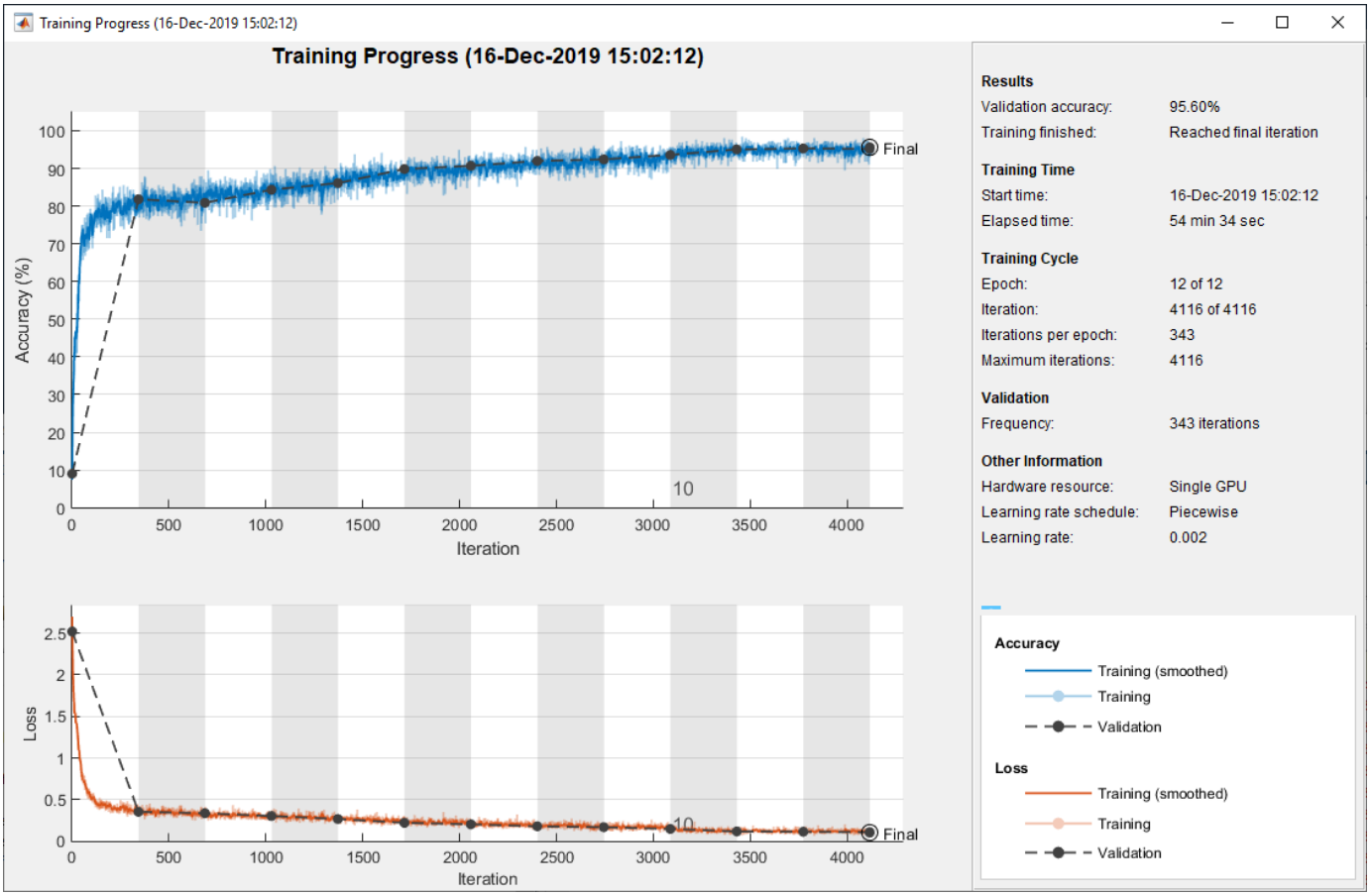
Next configure `TrainingOptionsSGDM` to use an SGDM solver with a mini-batch size of 256. Set the maximum number of epochs to 12, since a larger number of epochs provides no further training advantage. By default, the `'ExecutionEnvironment'` property is set to `'auto'`, where the `trainNetwork` function 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
  fprintf('%s - Training the network\n', datestr(toc/86400,'HH:MM:SS'))
  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.

```
fprintf('%s - Classifying test frames\n', datestr(toc/86400,'HH:MM:SS'))
```

```
00:06:19 - Classifying test frames
```

```
% Gather the test frames into the memory
testFramesTall = tall(transform(testDSTrans, @helperModClassReadFrame));
rxTestFrames = gather(testFramesTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4.4 sec
Evaluation completed in 4.5 sec
```

```
rxTestFrames = cat(4, rxTestFrames{:});
```

```
% Gather the test labels into the memory
testLabelsTall = tall(transform(testDSTrans, @helperModClassReadLabel));
rxTestLabels = gather(testLabelsTall);
```

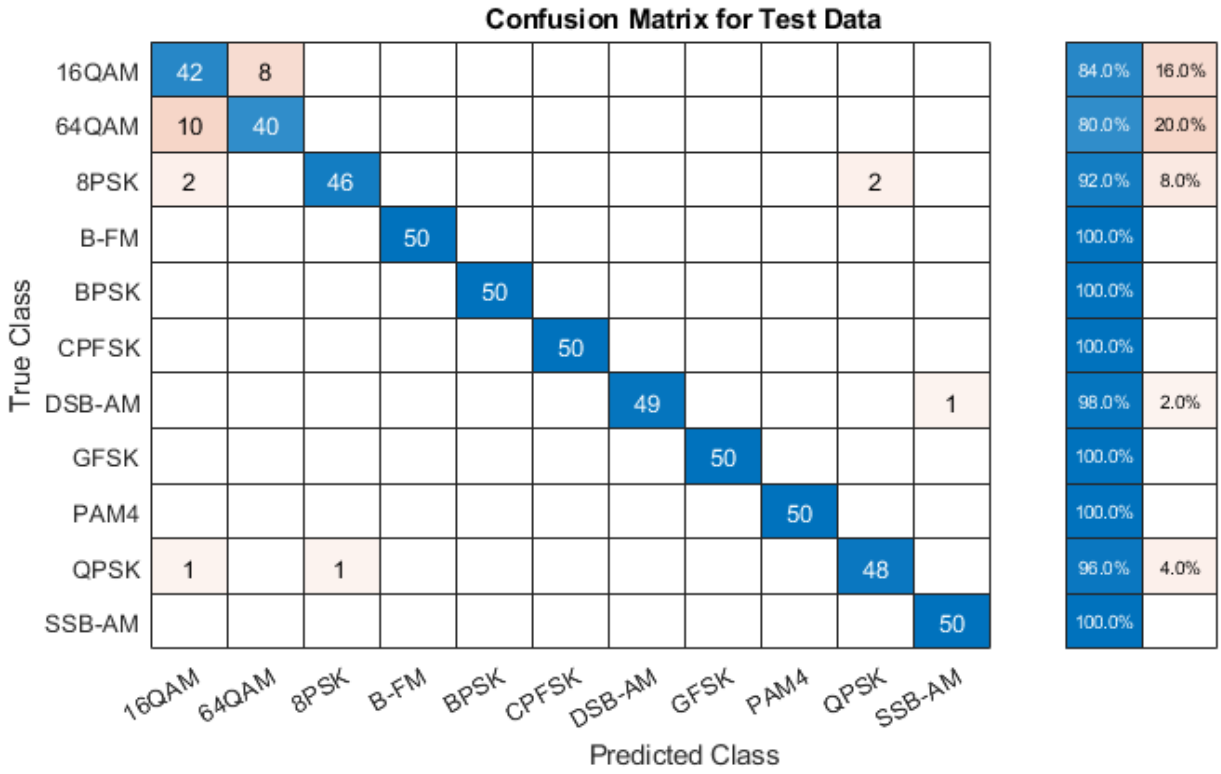
```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 4.2 sec
- Pass 2 of 2: Completed in 4.6 sec
Evaluation completed in 9.6 sec
```

```
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];
```



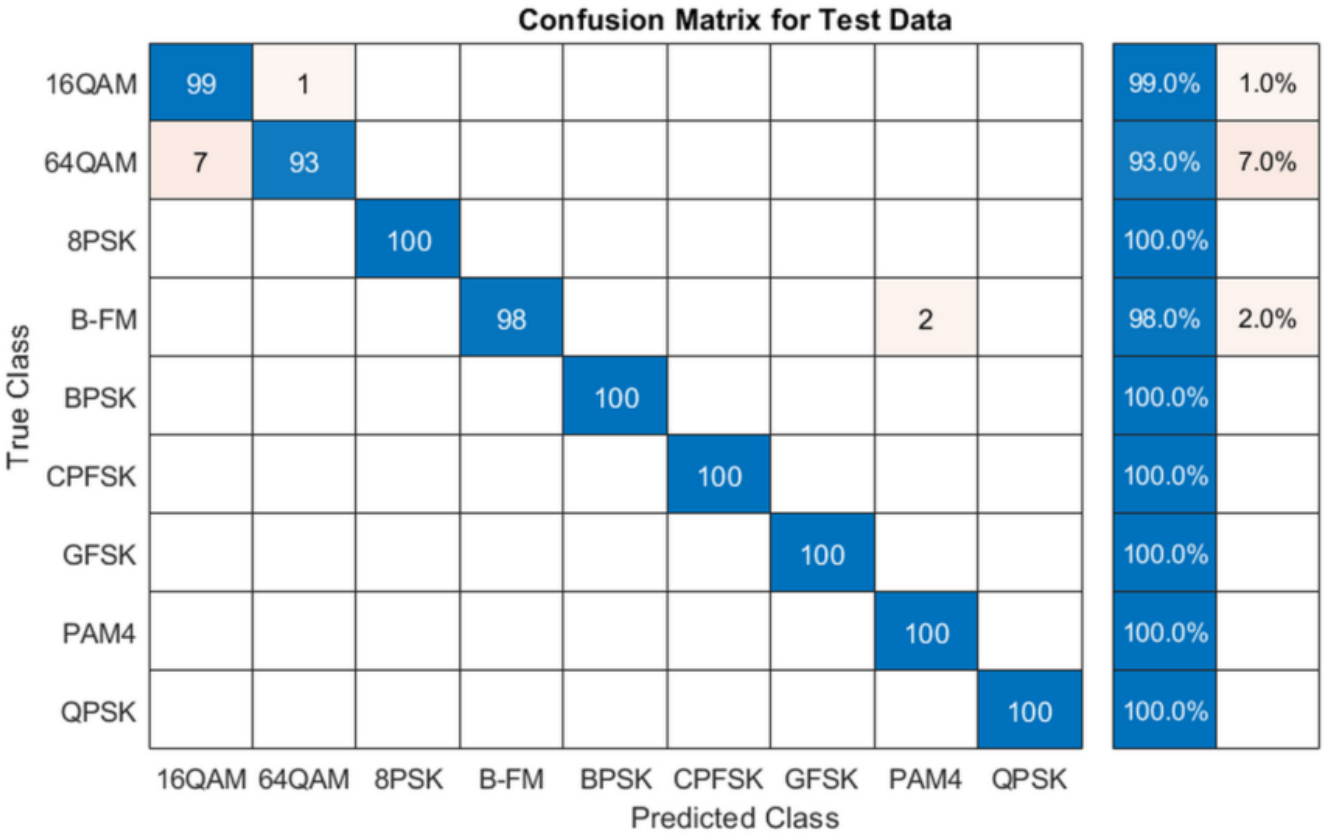
Test with SDR

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 = ;

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 “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

`trainNetwork` | `trainingOptions`

More About

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

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 11-0], [2 on page 11-0], [3 on page 11-0], 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.

```
ReadPhysionetData
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
```

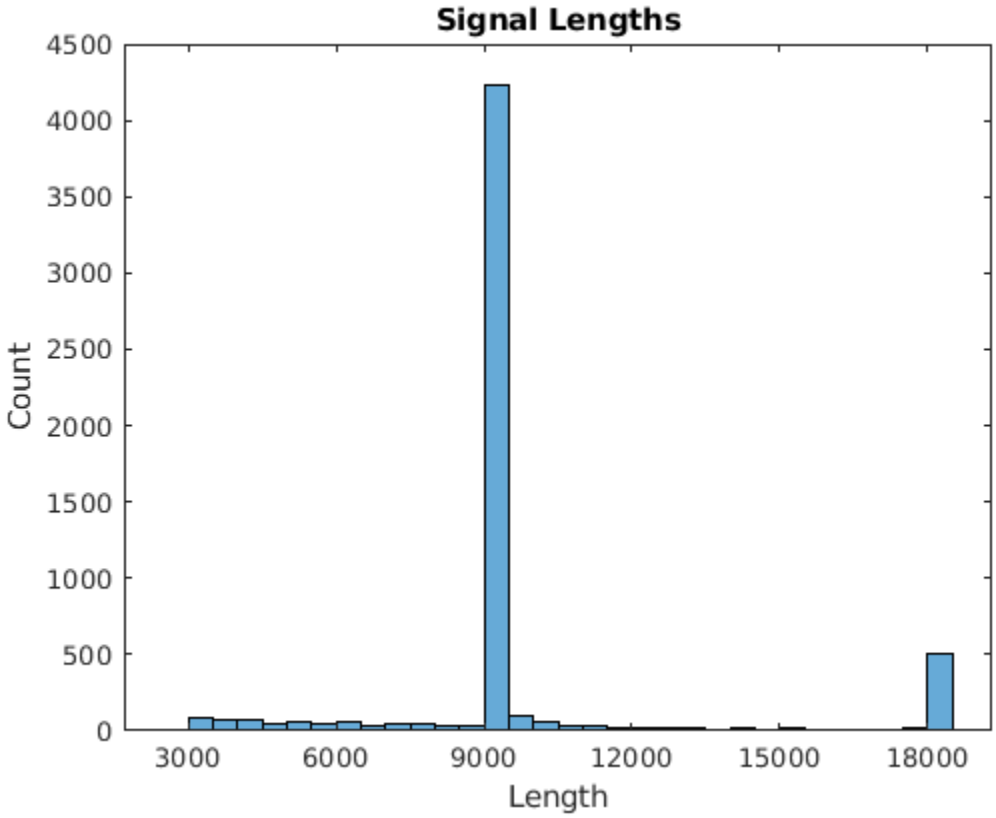
Use the `summary` function to see that there are 738 AFib signals and 5050 Normal signals.

```
summary(Labels)
```

```
     A       738
     N      5050
```

Generate a histogram of signal lengths. Notice that most of the signals are 9000 samples long.

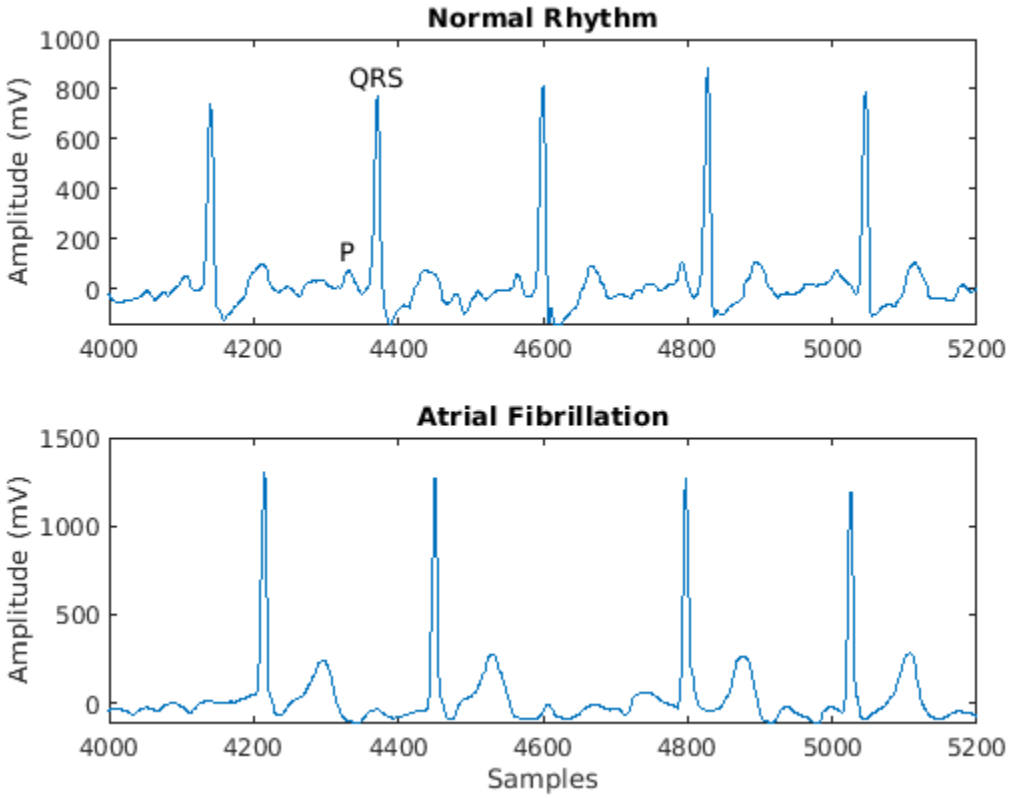
```
L = cellfun(@length,Signals);
h = histogram(L);
xticks(0:3000:18000);
xticklabels(0:3000:18000);
title('Signal Lengths')
xlabel('Length')
ylabel('Count')
```



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 there 718 AFib signals and 4937 Normal signals, a ratio of 1:7.

```
summary(Labels)
```

```
     A       718
     N      4937
```

Because 87.3% 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`.

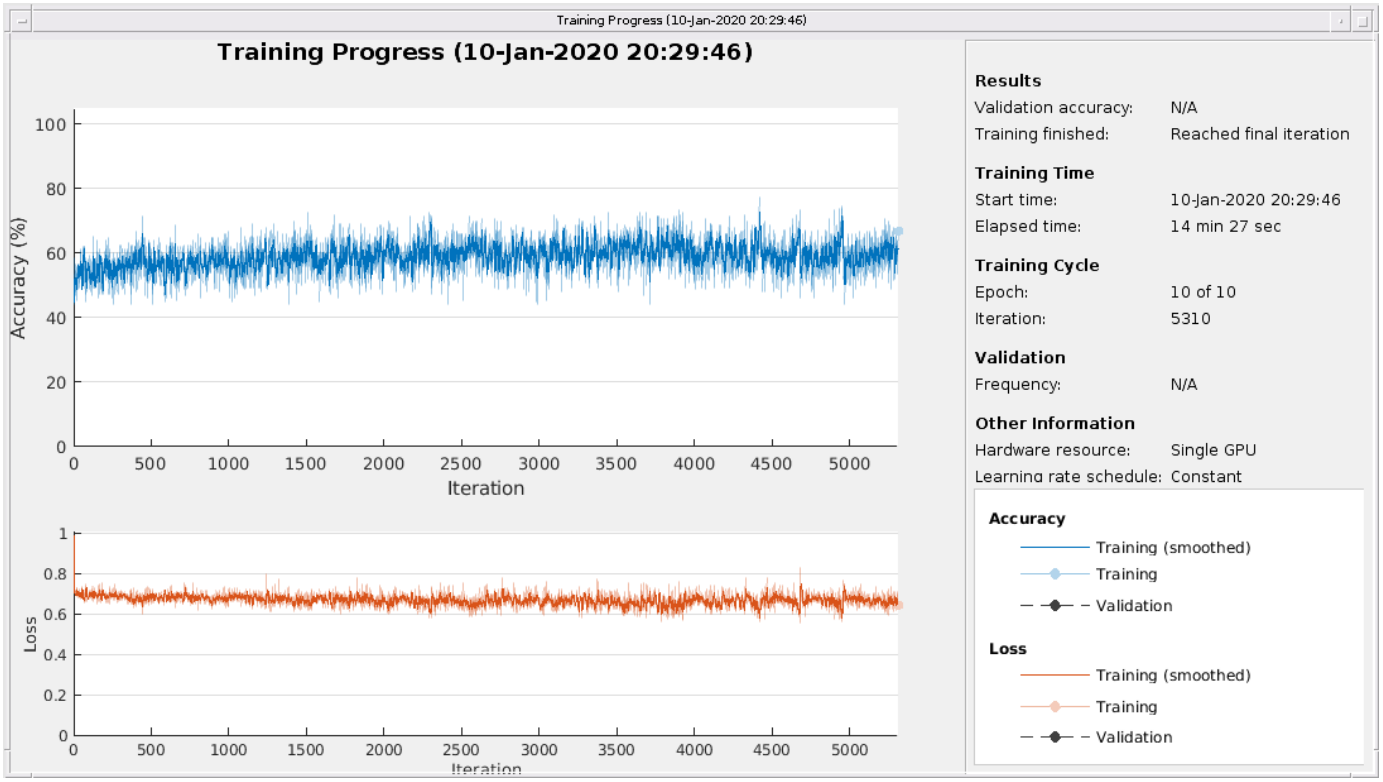
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with RNNs like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
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 50% and 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);
```

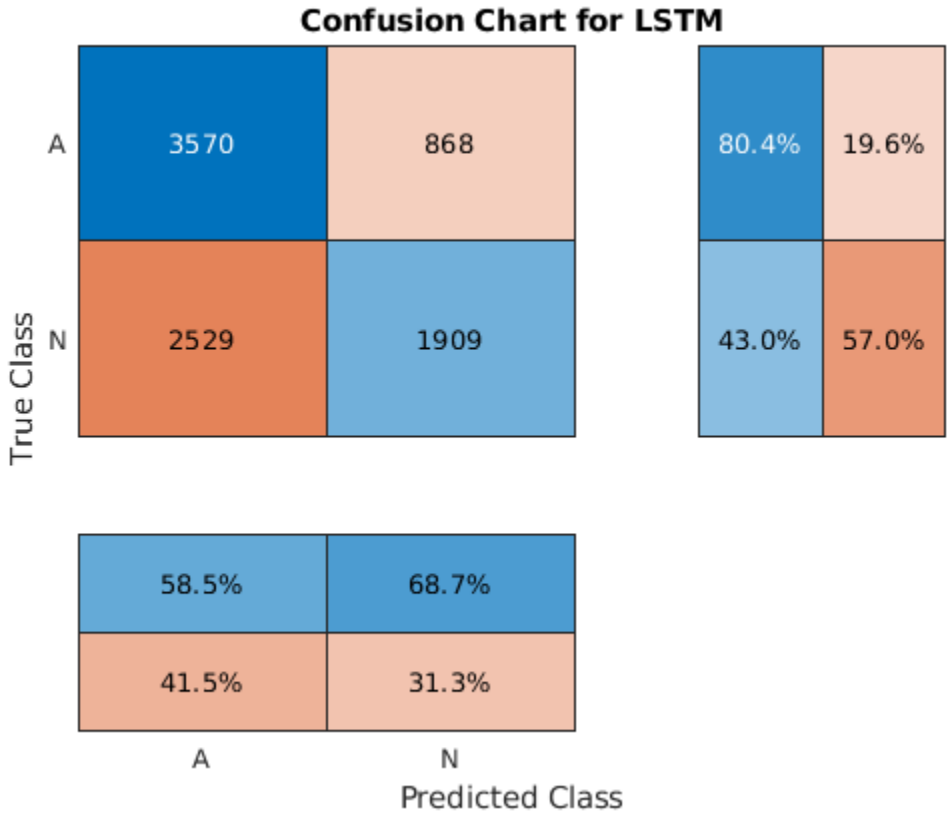
In classification problems, confusion matrices are used to visualize the performance of a classifier on a set of data for which the true values are known. The Target Class is the ground-truth label of the signal, and the Output Class is the label assigned to the signal by the network. The axes labels 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');
```



The confusion matrix shows that 81.7% of the ground-truth AFib signals are correctly classified as AFib, while 31.1% of ground-truth Normal signals are correctly classified as Normal. Furthermore, 54.2% of the signals classified as AFib are actually AFib, and 63.0% of the signals classified as Normal are actually Normal. The overall training accuracy is 56.4%.

Now classify the testing data with the same network.

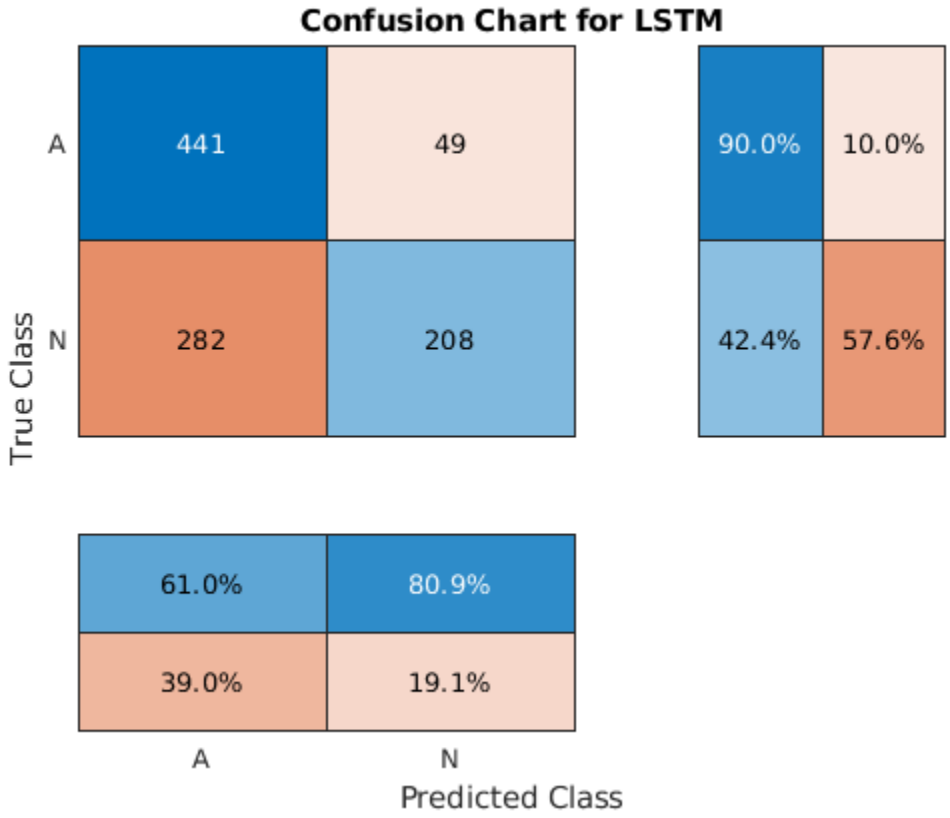
```
testPred = classify(net,XTest,'SequenceLength',1000);
```

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');
```



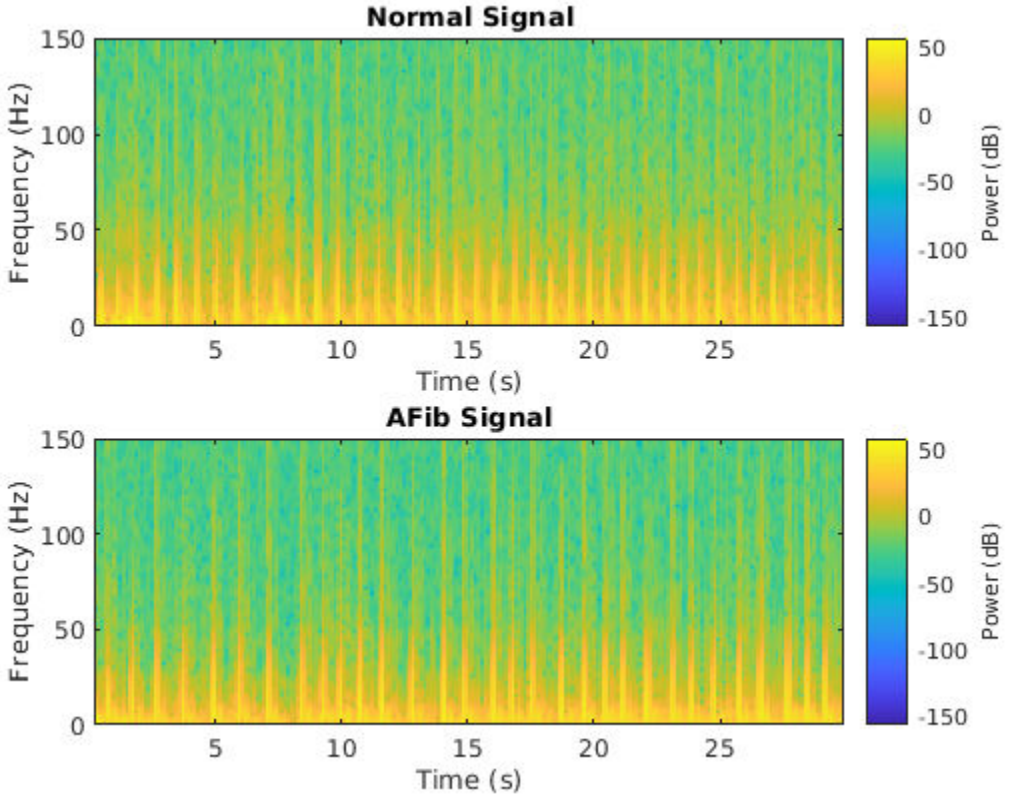
This confusion matrix is similar to the training confusion matrix. The overall testing accuracy is between 50% to 60%.

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 11-0], [5 on page 11-0].

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')
```



Since 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:

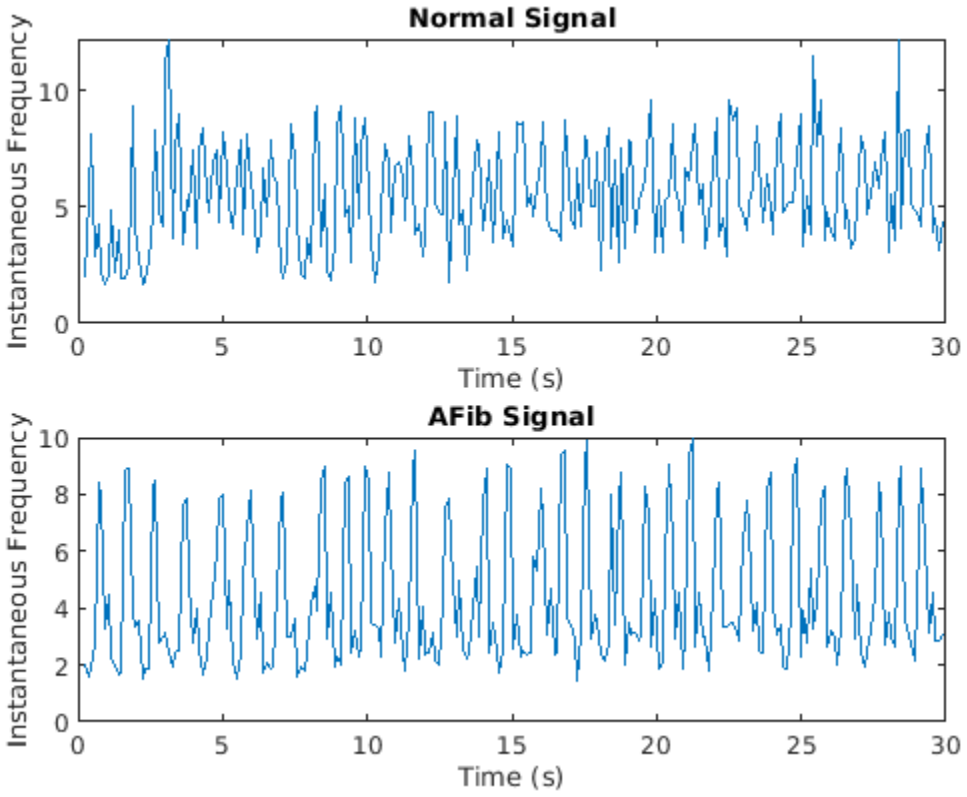
- Instantaneous frequency (`instfreq`)
- Spectral entropy (`pentropy`)

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.

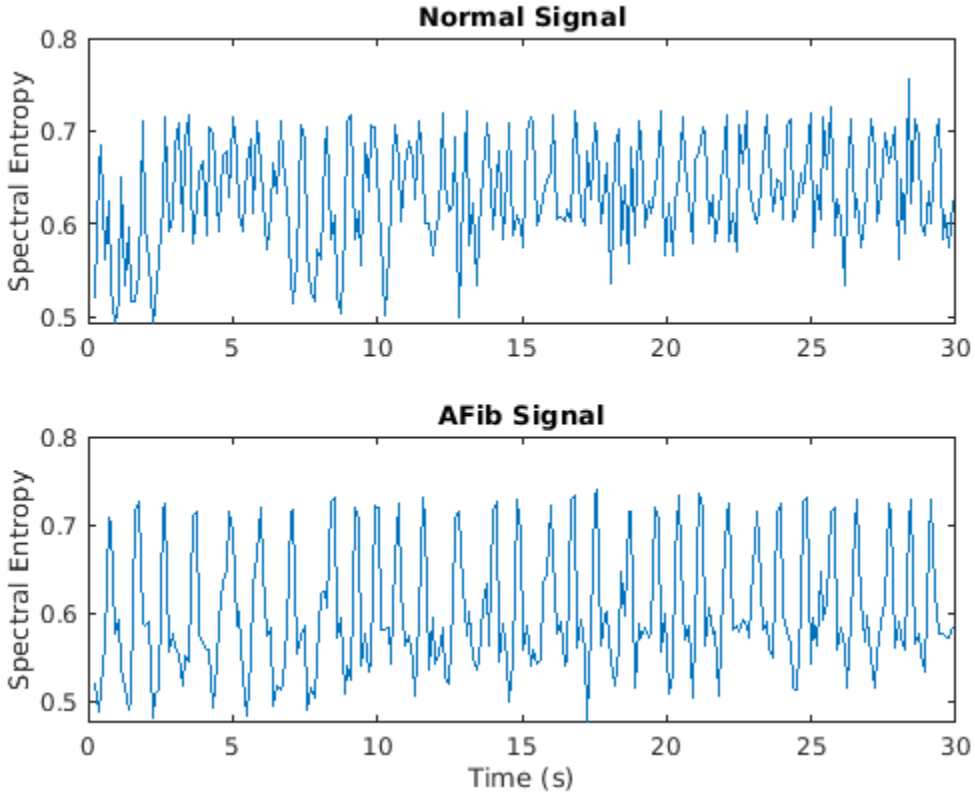
```
instfreqTrain = cellfun(@(x)instfreq(x,fs)',XTrain,'UniformOutput',false);
instfreqTest = cellfun(@(x)instfreq(x,fs)',XTest,'UniformOutput',false);
```

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 11-0].

```
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
```

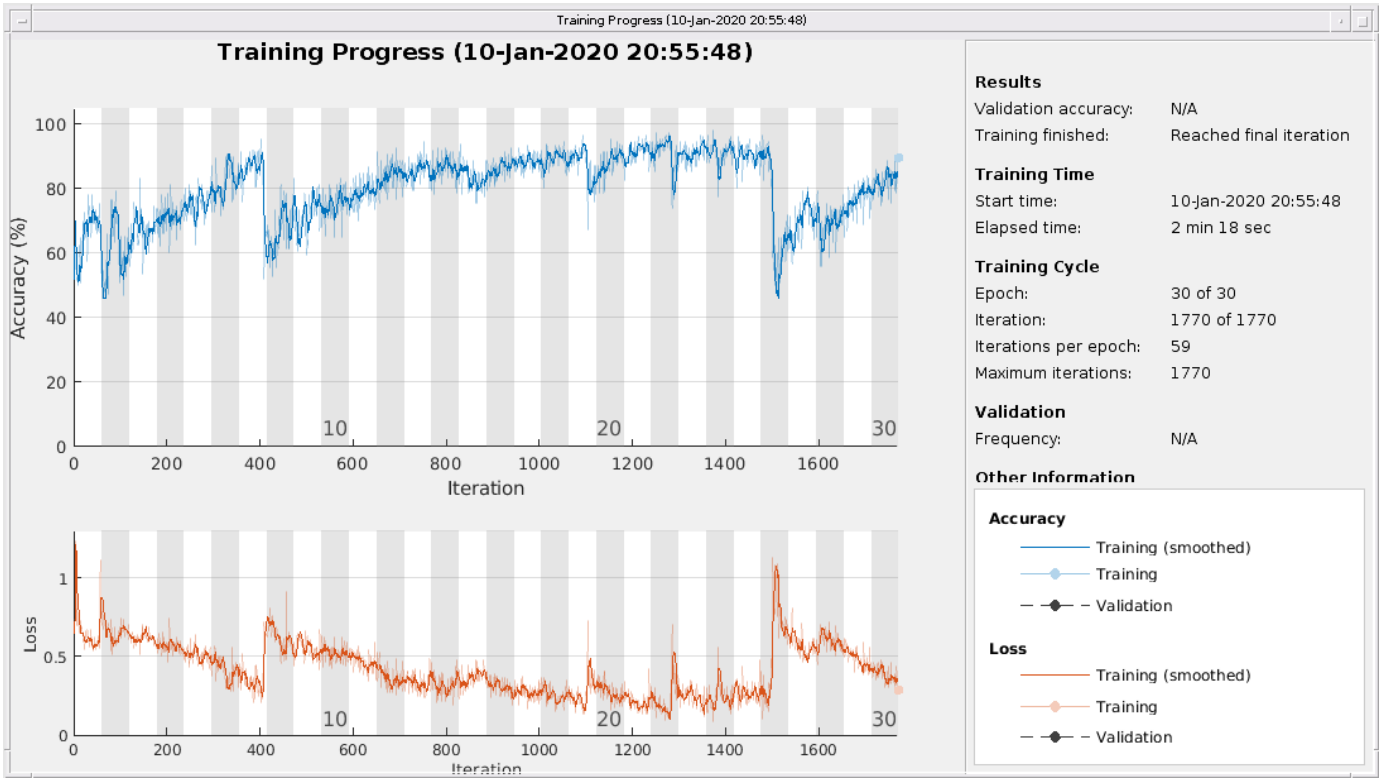
Specify the training options. Set the maximum number of epochs to 30 to allow the network to make 30 passes through the training data.

```
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, which is now greater than 90%. The cross-entropy loss trends towards 0. Furthermore, the time required for training decreases because the TF moments are shorter than the raw sequences.

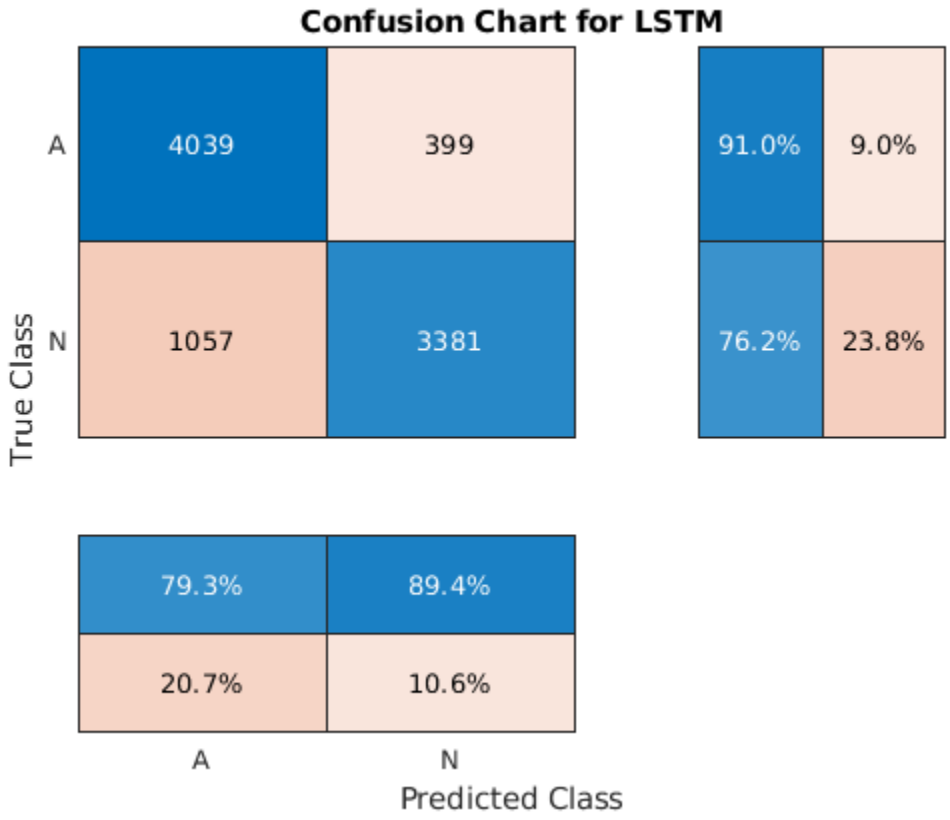
Visualize the Training and Testing Accuracy

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');
```

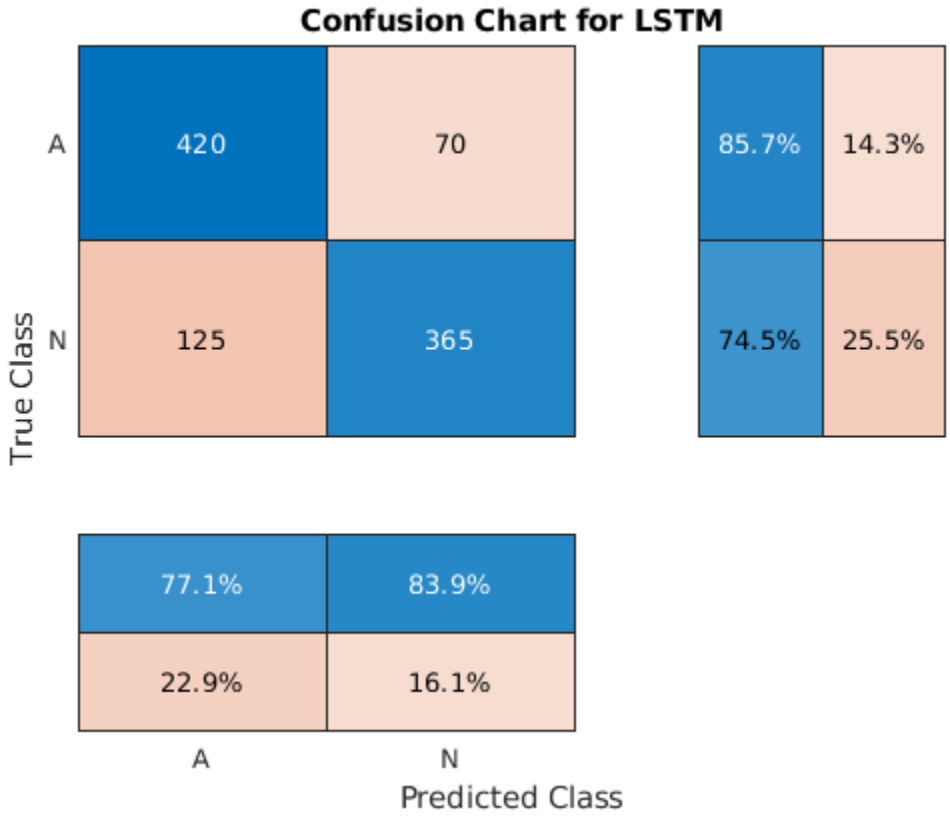


Classify the testing data with the updated network. Plot the confusion matrix to examine the testing accuracy.

```
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

`bilstmLayer` | `instfreq` | `lstmLayer` | `pentropy` | `trainNetwork` | `trainingOptions`

More About

- “Long Short-Term Memory Networks” on page 1-53
- “Deep Learning in MATLAB” on page 1-2

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 `Clone or download` and select `Download ZIP`. Save the file `physionet_ECG_data-master.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`. If you are familiar with Git, you can download the latest version of the tools (git) and obtain the data from a system command prompt using `git clone https://github.com/mathworks/physionet_ECG_data/`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir,'physionet_ECG_data-master.zip'),tempdir)
```

Unzipping creates the folder `physionet-ECG_data-master` 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-master`. Load the data file into your MATLAB workspace.

```
unzip(fullfile(tempdir,'physionet_ECG_data-master','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-master'))
load(fullfile(tempdir,'physionet_ECG_data-master','ECGData.mat'))
```

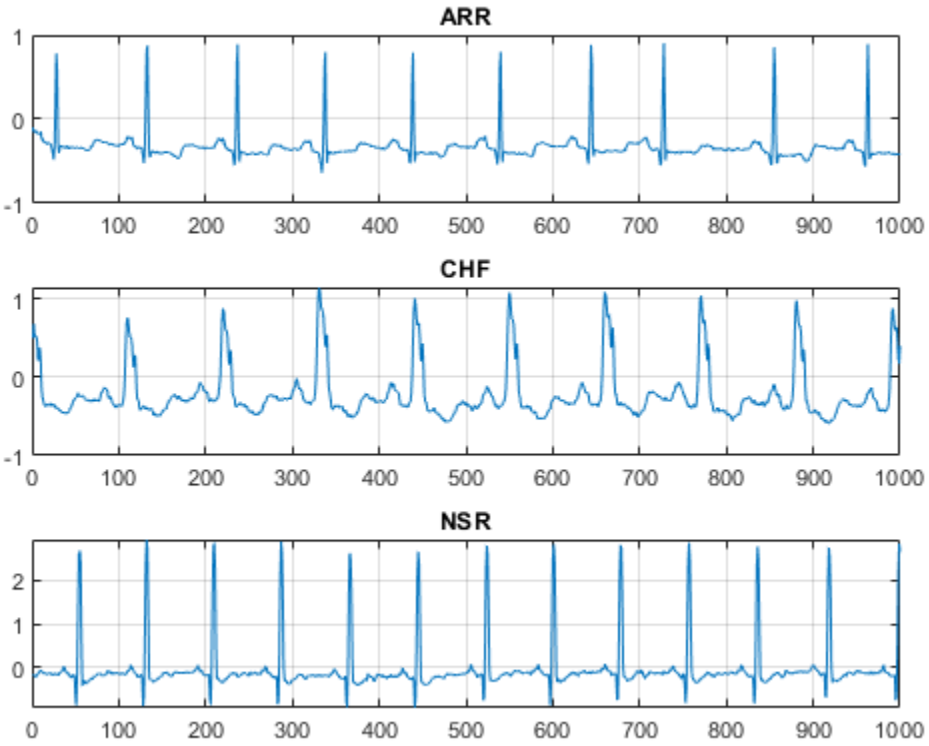
`ECGData` is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one for each row of `Data`. The three diagnostic categories are: `'ARR'`, `'CHF'`, 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)
```



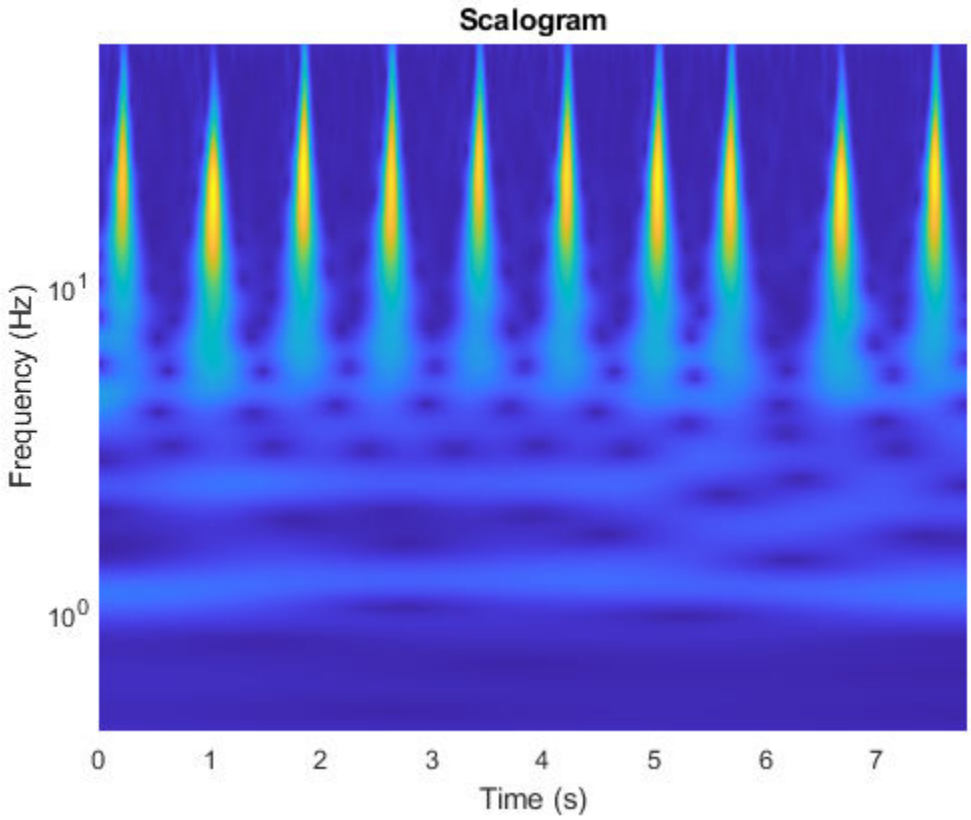
Create Time-Frequency Representations

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` 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

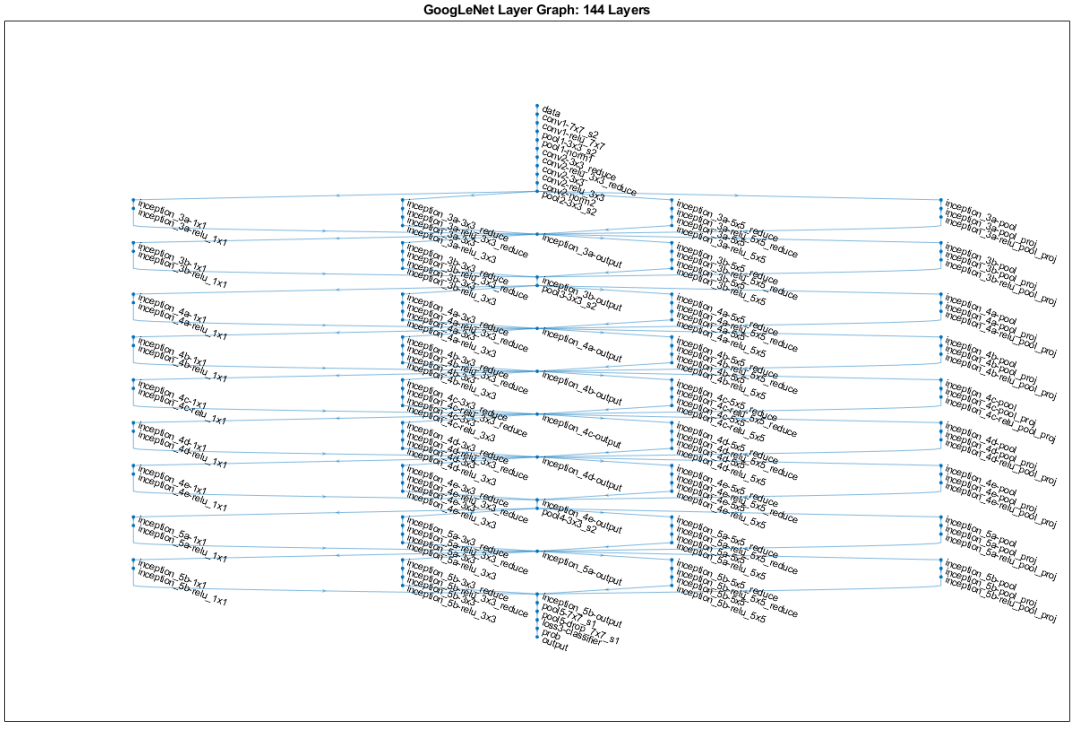
Load

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'
    NormalizationDimension: 'auto'
                     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.

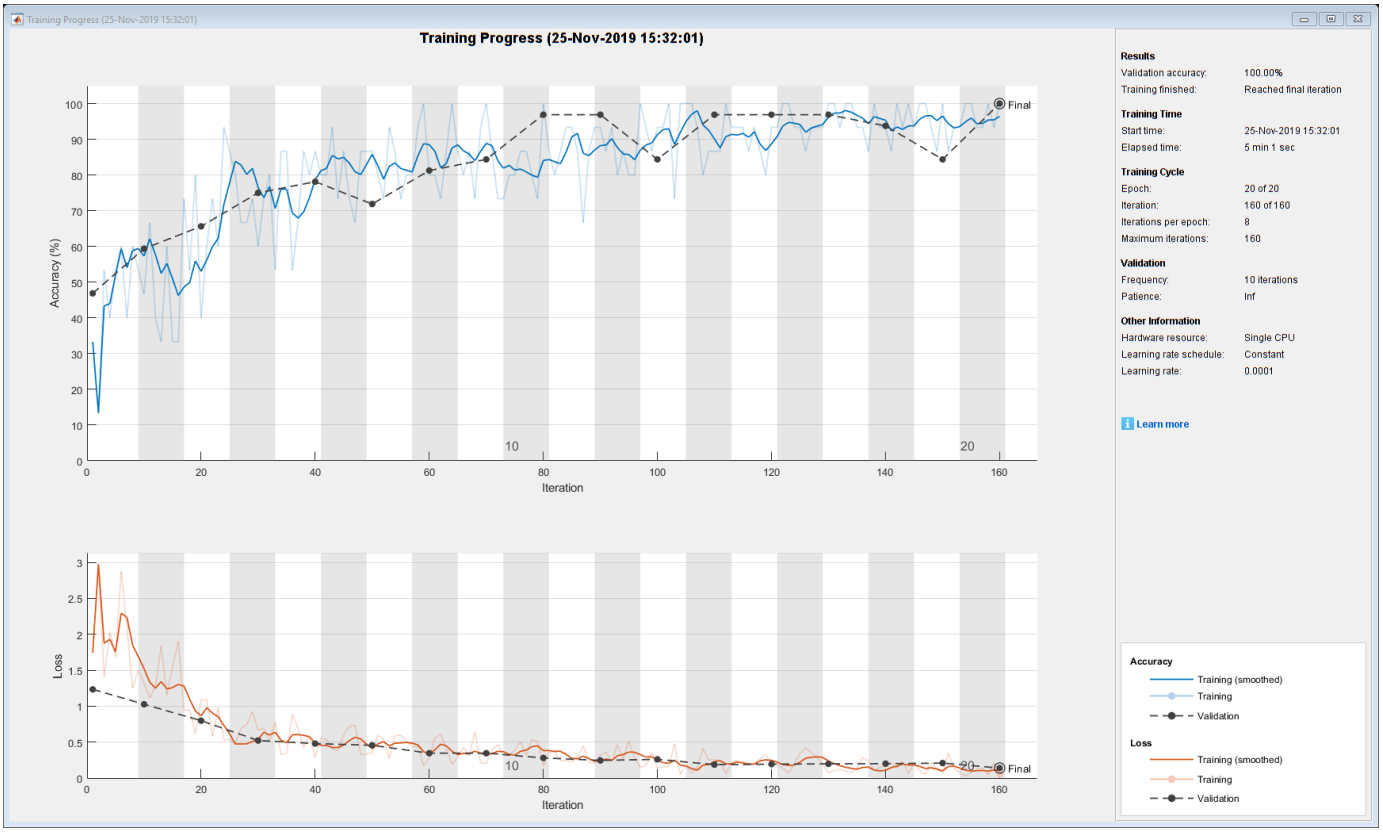
Training can be tuned by setting various options. `InitialLearnRate` specifies the initial step size in the direction of the negative gradient of the loss function. `MiniBatchSize` specifies how large of a subset of the training set to use in each iteration. One epoch is a full pass of the training algorithm 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 (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). 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 |       00:00:05 |       33.33% |       46.88% |       1.7424 |       1.2       |
|       2 |          10 |       00:00:24 |       46.67% |       59.38% |       1.3215 |       1.0       |
|       3 |          20 |       00:00:43 |       40.00% |       65.63% |       1.0882 |       0.7       |
|       4 |          30 |       00:01:02 |       60.00% |       75.00% |       0.6736 |       0.5       |
|       5 |          40 |       00:01:21 |       86.67% |       78.13% |       0.2956 |       0.4       |
|       7 |          50 |       00:01:41 |       86.67% |       71.88% |       0.3519 |       0.4       |
|       8 |          60 |       00:02:00 |       80.00% |       81.25% |       0.2808 |       0.3       |
|       9 |          70 |       00:02:18 |      100.00% |       84.38% |       0.2027 |       0.3       |
|      10 |          80 |       00:02:36 |      100.00% |       96.88% |       0.1735 |       0.2       |
|      12 |          90 |       00:02:55 |       93.33% |       96.88% |       0.3199 |       0.2       |
|      13 |         100 |       00:03:13 |       93.33% |       84.38% |       0.1391 |       0.2       |
|      14 |         110 |       00:03:31 |       86.67% |       96.88% |       0.2537 |       0.1       |
|      15 |         120 |       00:03:49 |       93.33% |       96.88% |       0.2141 |       0.1       |
|      17 |         130 |       00:04:07 |      100.00% |       96.88% |       0.0714 |       0.1       |
|      18 |         140 |       00:04:25 |       93.33% |       93.75% |       0.1607 |       0.2       |
|      19 |         150 |       00:04:42 |      100.00% |       84.38% |       0.0683 |       0.2       |
|      20 |         160 |       00:05:00 |      100.00% |      100.00% |       0.0540 |       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),'%'])
```

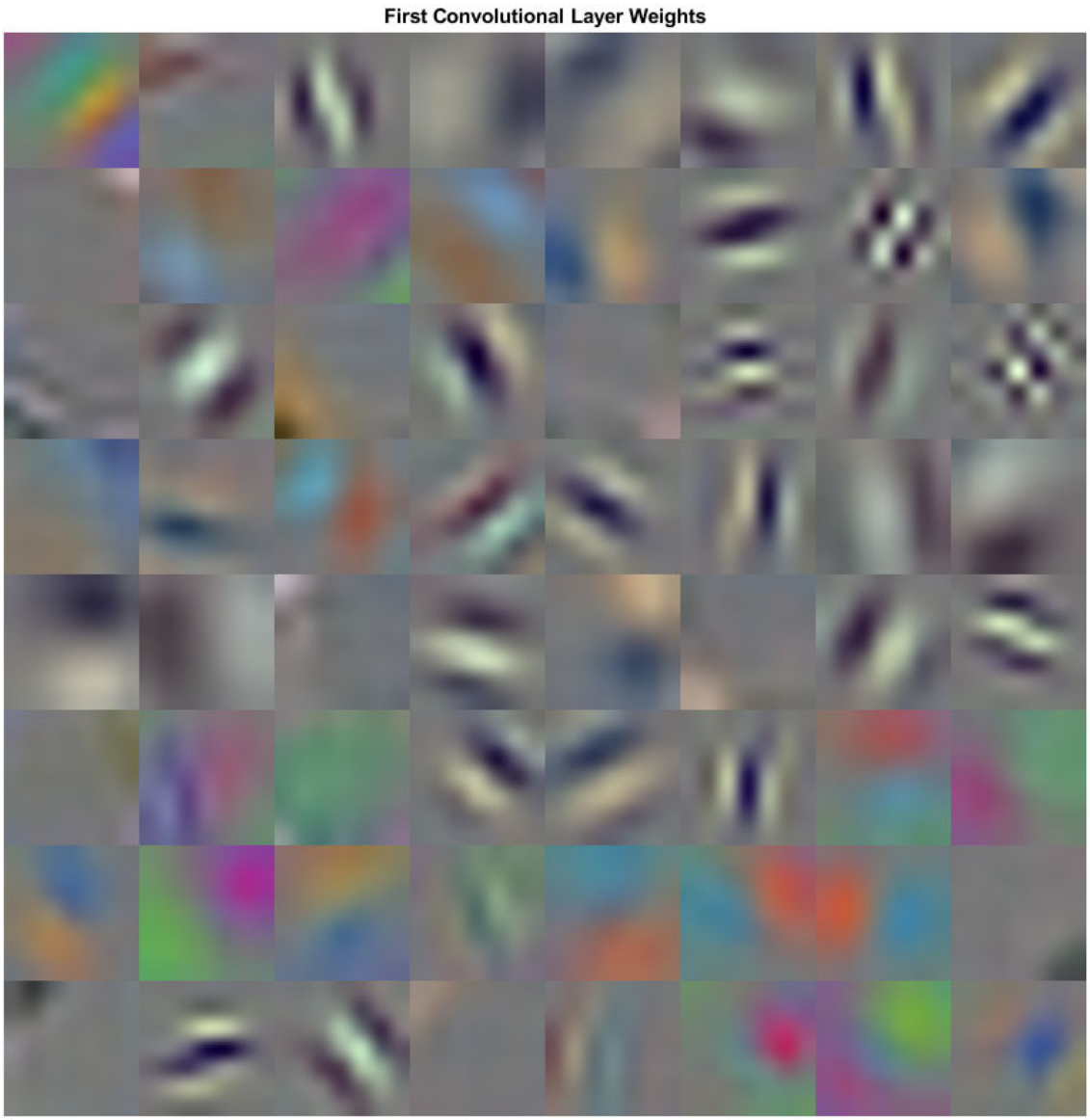
```
GoogLeNet Accuracy: 100%
```

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')
```

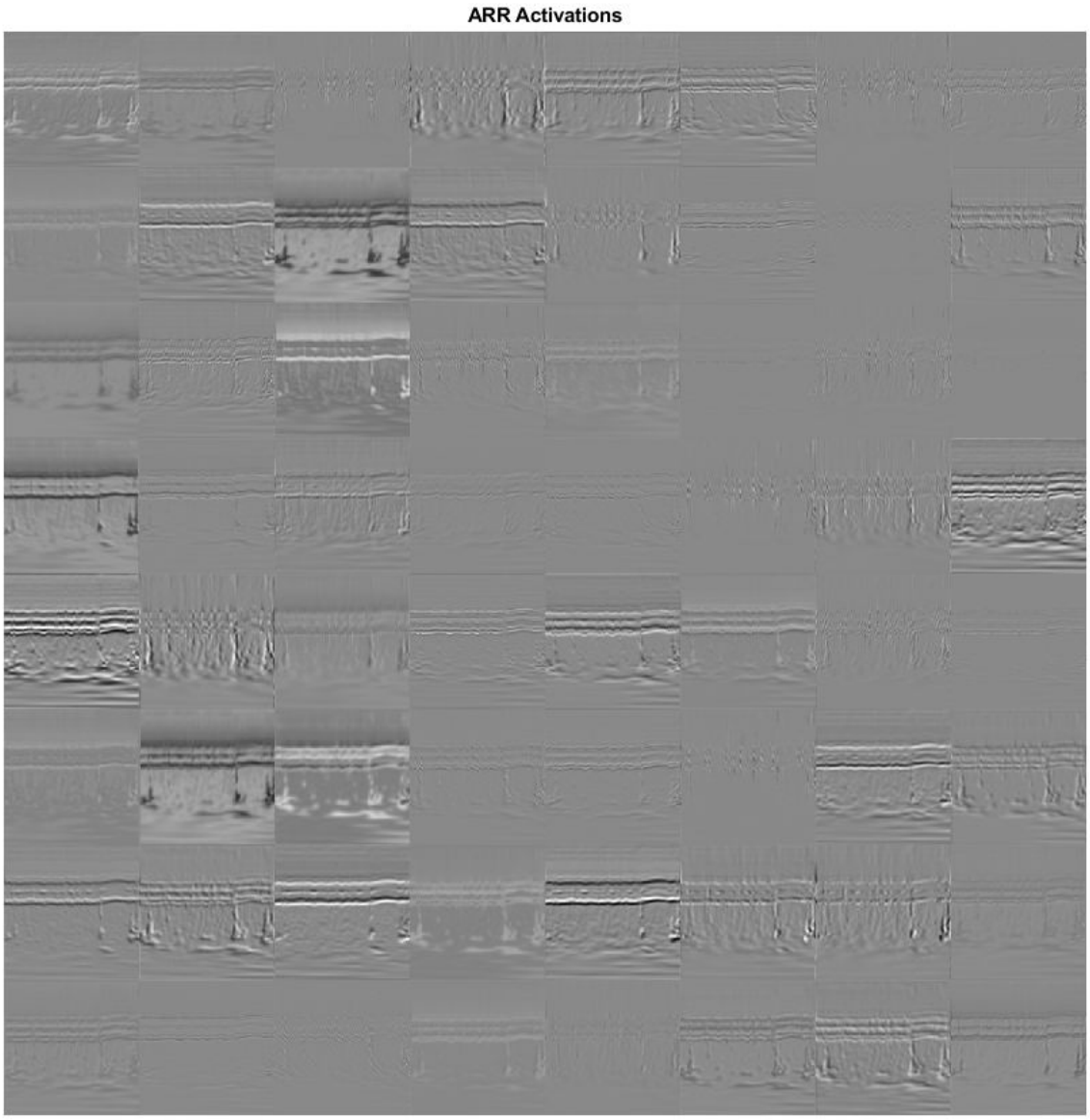


You can examine the activations and discover which features GoogLeNet learns by comparing areas of activation with the original image. For more information, see “Visualize Activations of a Convolutional Neural Network” on page 5-75 and “Visualize Features of a Convolutional Neural Network” on page 5-90.

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'`.

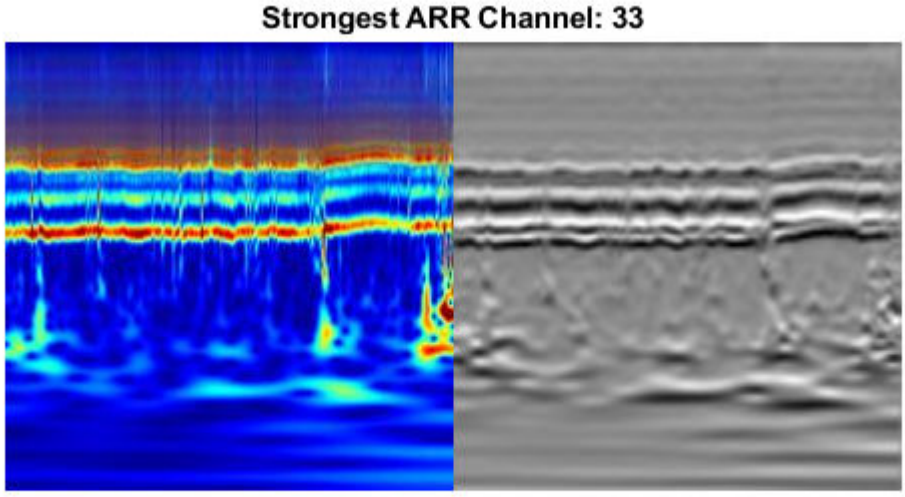
```
convLayer = 'conv1-7x7_s2';
```

```
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.

```
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 w
     3   'relu_conv10'                          ReLU                     ReLU
     4   'pool10'                               Global Average Pooling   Global average pooling
     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 p
     3   'relu_conv10'       ReLU                     ReLU
     4   'pool10'            Global Average Pooling   Global average pooling
     5   'prob'              Softmax                  softmax
     6   'new_classoutput'   Classification Output    crossentropyex
```

Prepare RGB Data for SqueezeNet

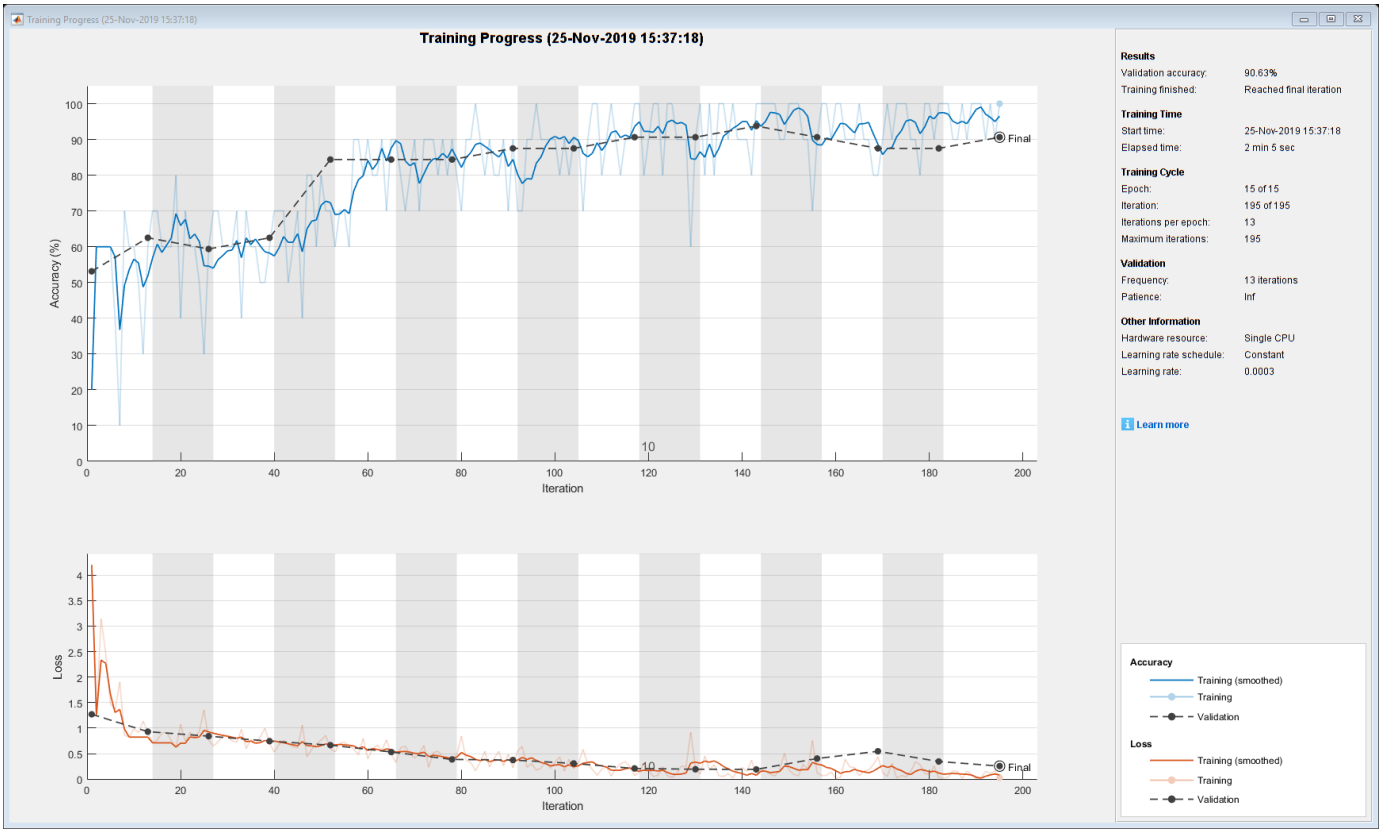
The RGB images have dimensions appropriate for the GoogLeNet architecture. Create augmented image datastores that automatically resize the existing RGB images for the SqueezeNet architecture. For more information, see `augmentedImageDatastore`.

```
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 |       00:00:02 |       20.00% |       53.13% |       4.2000 |       1.2
|       1 |          13 |       00:00:09 |       60.00% |       62.50% |       0.9177 |       0.9
|       2 |          26 |       00:00:18 |       60.00% |       59.38% |       0.7669 |       0.8
|       3 |          39 |       00:00:26 |       60.00% |       62.50% |       0.7080 |       0.7
|       4 |          50 |       00:00:33 |       80.00% |              |       0.7647 |
|       4 |          52 |       00:00:34 |       70.00% |       84.38% |       0.5962 |       0.6
|       5 |          65 |       00:00:43 |       90.00% |       84.38% |       0.4940 |       0.5
|       6 |          78 |       00:00:51 |       90.00% |       84.38% |       0.3721 |       0.3
|       7 |          91 |       00:00:59 |       90.00% |       87.50% |       0.2226 |       0.3
|       8 |         100 |       00:01:05 |       90.00% |              |       0.3315 |
|       8 |         104 |       00:01:08 |       90.00% |       87.50% |       0.2086 |       0.3
|       9 |         117 |       00:01:16 |      100.00% |       90.63% |       0.0727 |       0.2
|      10 |         130 |       00:01:24 |       90.00% |       90.63% |       0.2222 |       0.1
|      11 |         143 |       00:01:32 |      100.00% |       93.75% |       0.0767 |       0.1
|      12 |         150 |       00:01:36 |      100.00% |              |       0.1799 |
|      12 |         156 |       00:01:40 |       90.00% |       90.63% |       0.1234 |       0.4
|      13 |         169 |       00:01:48 |       80.00% |       87.50% |       0.4372 |       0.5
|      14 |         182 |       00:01:56 |      100.00% |       87.50% |       0.0245 |       0.3
|      15 |         195 |       00:02:04 |      100.00% |       90.63% |       0.0278 |       0.2
|=========================================================================================================
```

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: 90.625%
```

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. 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` 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

`augmentedImageDatastore` | `cwtfilterbank` | `googlenet` | `imageDatastore` | `squeezenet` | `trainNetwork` | `trainingOptions`

Related Examples

- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Pretrained Deep Neural Networks” on page 1-12
- “Deep Learning in MATLAB” on page 1-2

Audio Examples

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 signal.

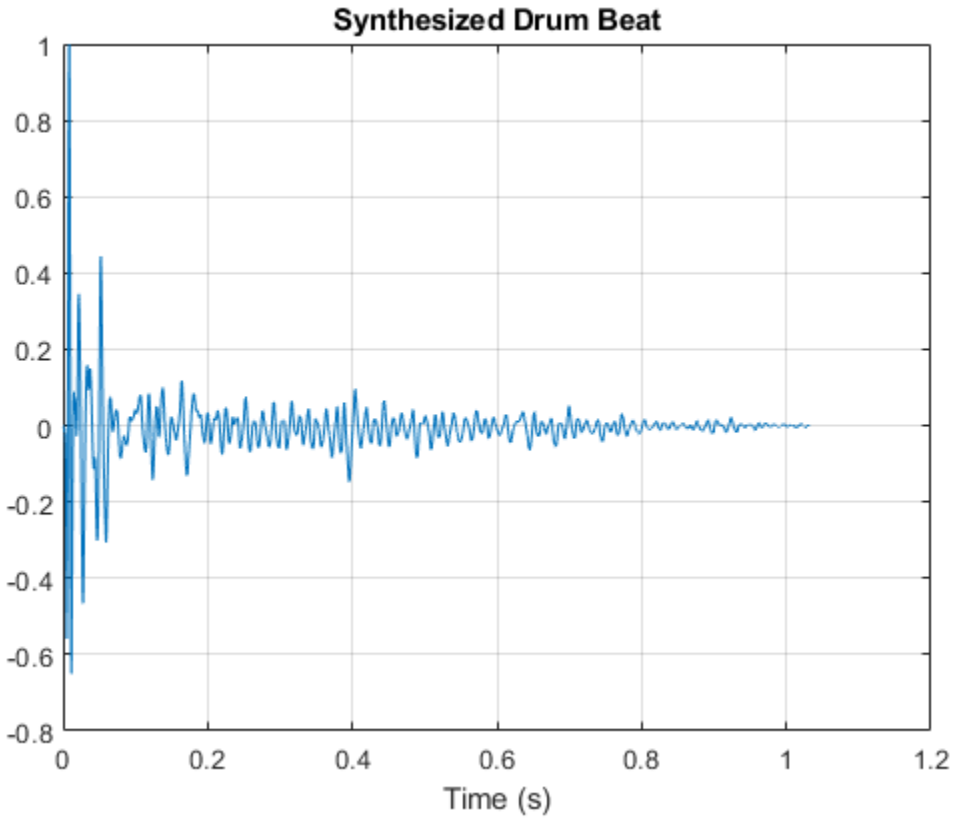
```
drum = synthesizeDrumBeat;
```

Listen to the synthesized drumbeat.

```
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` object and launch its parameter tuner UI. This UI enables you to tune the `reverberator` parameters as the simulation runs.

```
reverb = reverberator('SampleRate',fs);
parameterTuner(reverb);
```

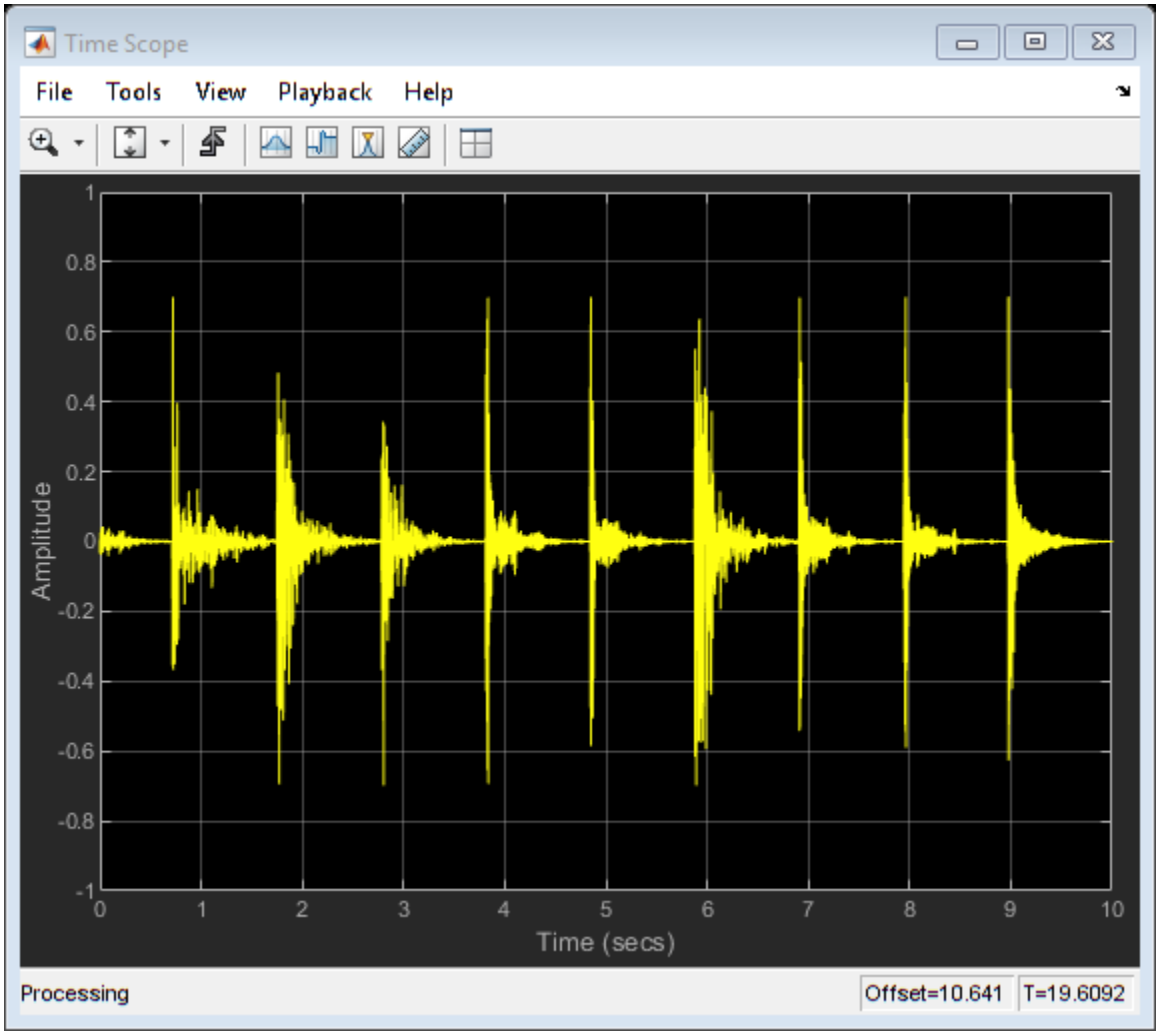
Create a `dsp.TimeScope` object to visualize the drum beats.

```
ts = dsp.TimeScope('SampleRate',fs, ...
                   '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.

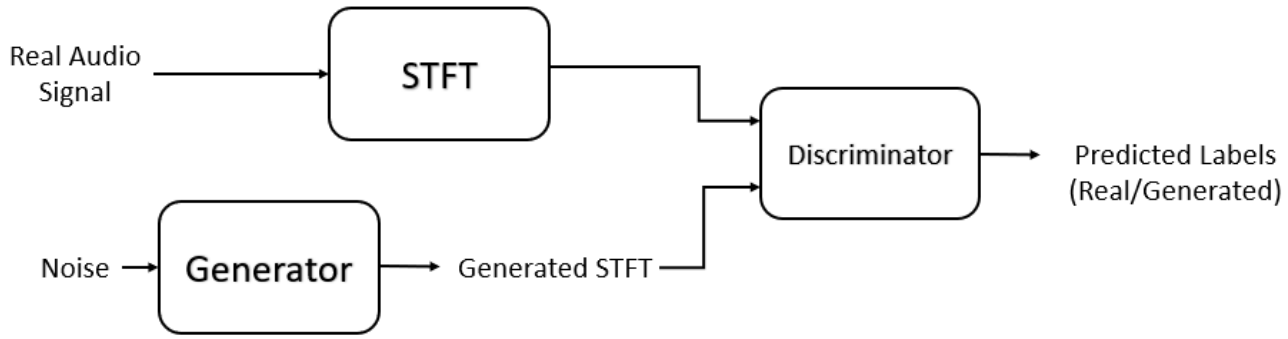
A GAN consists of two networks that train together, a *generator* and a *discriminator*:

- 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
```

```
Downloading Drum Sound Effects Dataset (218 MB)...
```

Create an `audioDatastore` 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
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

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™ and a CUDA-enabled NVIDIA GPU with compute capability of 3.0 or higher.

```
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` 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.

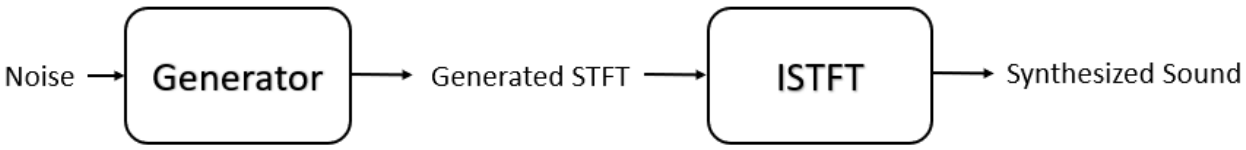
```
iteration = 0;

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 the iterative Griffin-Lim algorithm with 18 iterations to estimate the signal phase and produce 16,384 audio samples.

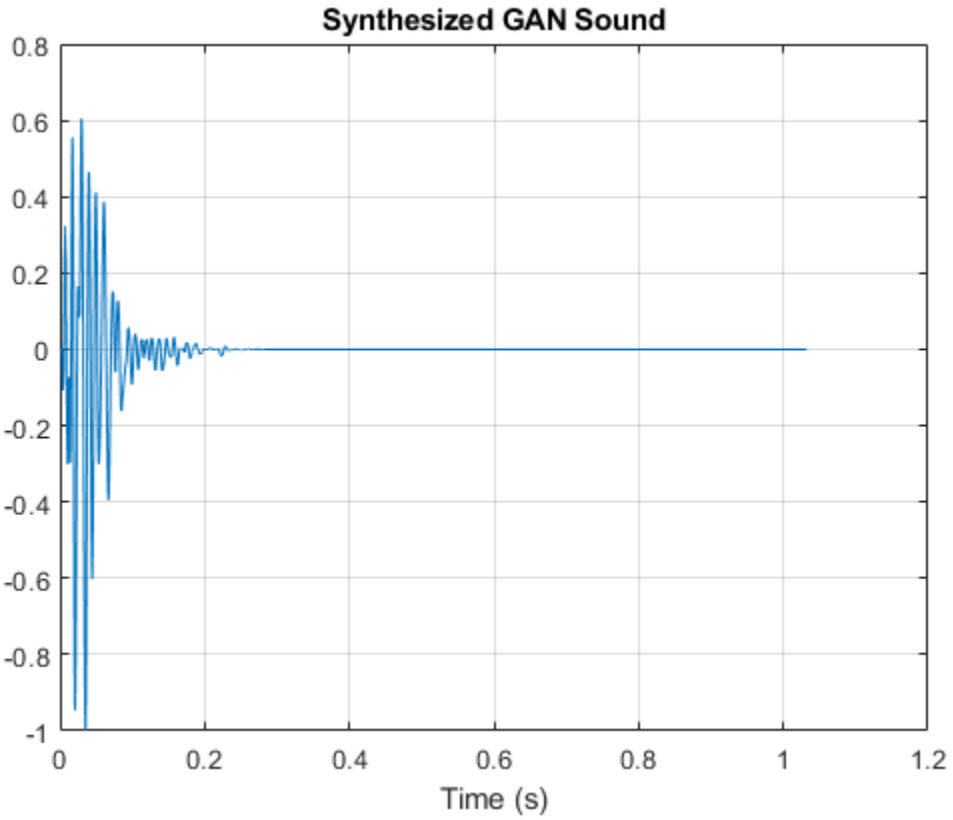
```
myAudio = griffinLim(S,18);
myAudio = myAudio(128*100:end-128*100);
```

Listen to the synthesized drumbeat.

```
sound(myAudio,fs)
```

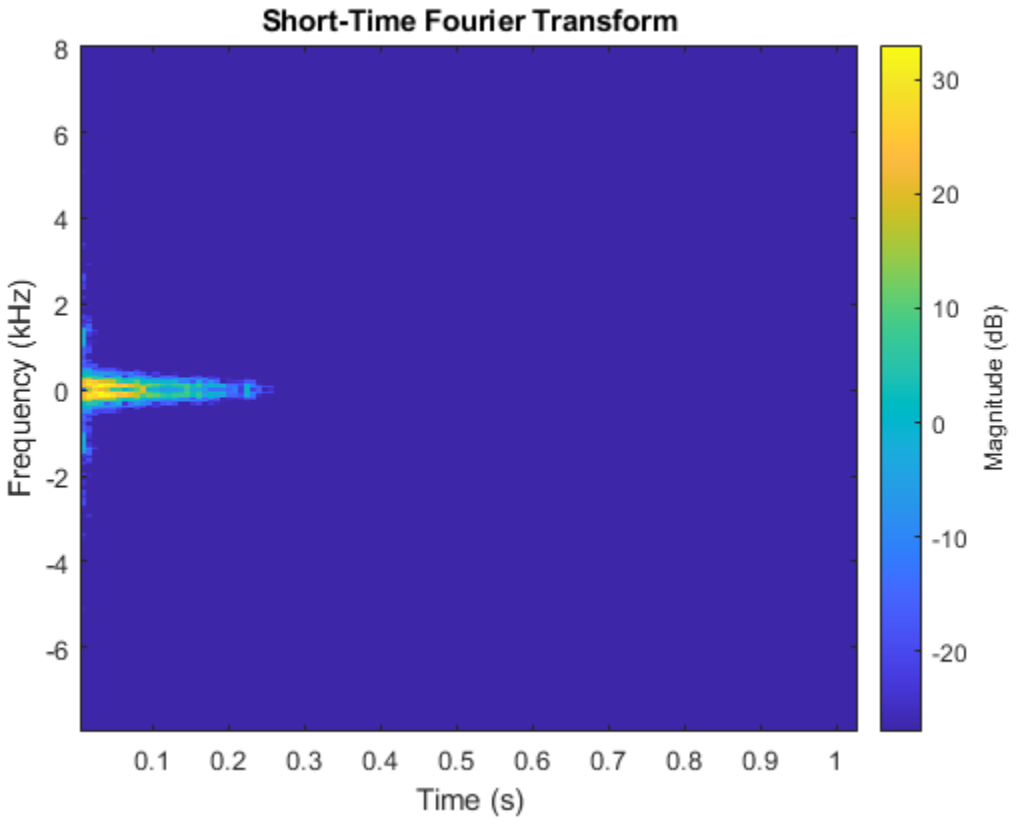
Plot the synthesized drumbeat.

```
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, generatorParameter

% 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 adversarial 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
```

Griffin-Lim algorithm

`griffinLim` performs inverse STFT by estimating the phase from a magnitude STFT.

```
function x = griffinLim(S,maxIter)
S0 = complex(S);
nfft = 256;
OverlapLength = 128;

for index = 1:maxIter
   x = istft(S,'Window',hann(nfft,'periodic'),'OverlapLength',OverlapLength,'Centered',false);
   S_est = stft(x,'Window',hann(nfft,'periodic'),'OverlapLength',OverlapLength,'Centered',false)
   phase = S_est ./ max(1e-8, abs(S_est));
   S = S0 .* phase;
end

x = istft(S,'Window',hann(nfft,'periodic'),'OverlapLength',OverlapLength,'Centered',false);

x = real(x);
x = x ./ max(abs(x));
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)];

myAudio = griffinLim(gather(S),18);
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 12-0 . In this example, you apply sequential forward selection to the task of spoken digit recognition using the Free Spoken Digit Dataset [2] on page 12-0 .

Streaming Spoken Digit Recognition

To motivate the example, begin by loading a pretrained network, the `audioFeatureExtractor` object used to train the network, and normalization factors for the features.

```
load('network_Audio_SequentialFeatureSelection.mat','bestNet','afe','normalizers');
```

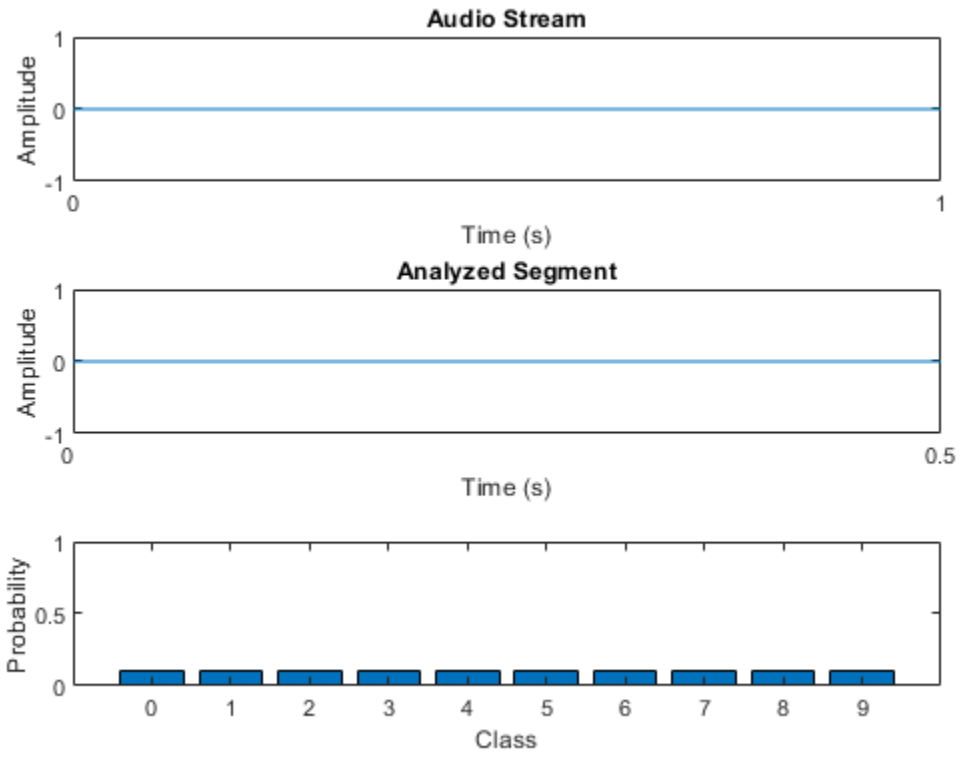
Create an `audioDeviceReader` to read audio from a microphone. Create three `dsp.AsyncBuffer` objects: one to buffer audio read from your microphone, one to buffer short-term energy of the input audio for speech detection, and one to buffer predictions.

```
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')
```



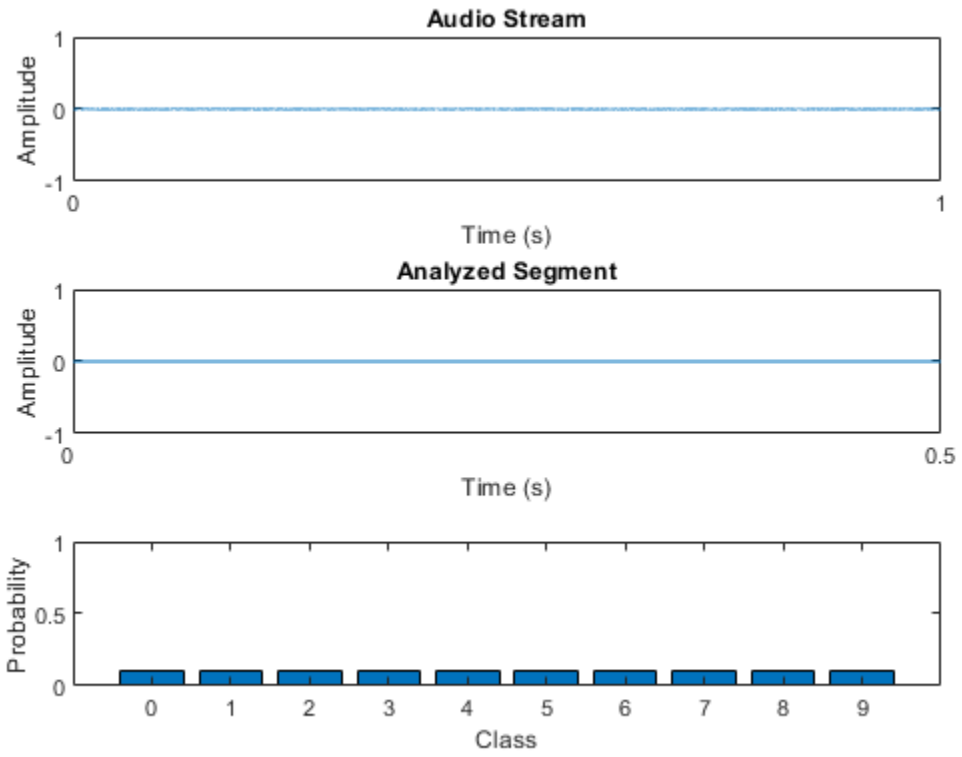
Perform streaming digit recognition (digits 0 through 9) for 20 seconds. While the loop runs, speak one of the digits and test its accuracy.

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 12-0 . 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` 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`. 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` 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 1-23 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 = ;
layers = [ ...
    sequenceInputLayer(1)
```

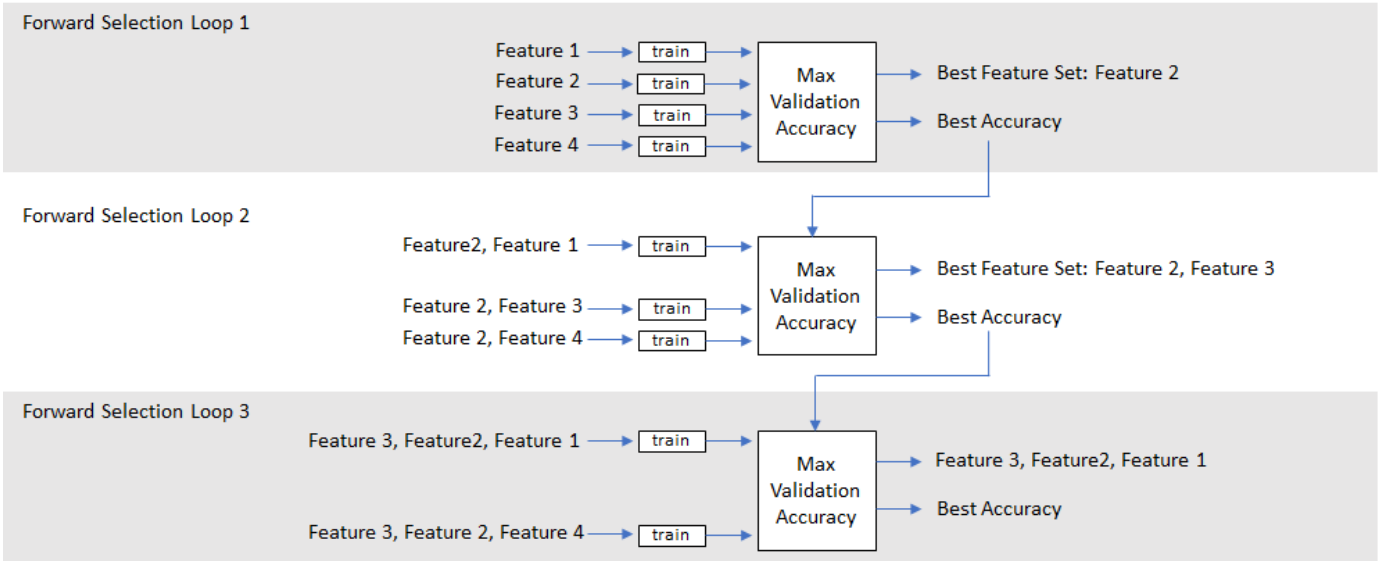
```
    bilstmLayer(numUnits,"OutputMode","last")
    fullyConnectedLayer(numel(categories(adsTrain.Labels)))
    softmaxLayer
    classificationLayer];

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 12-0 .

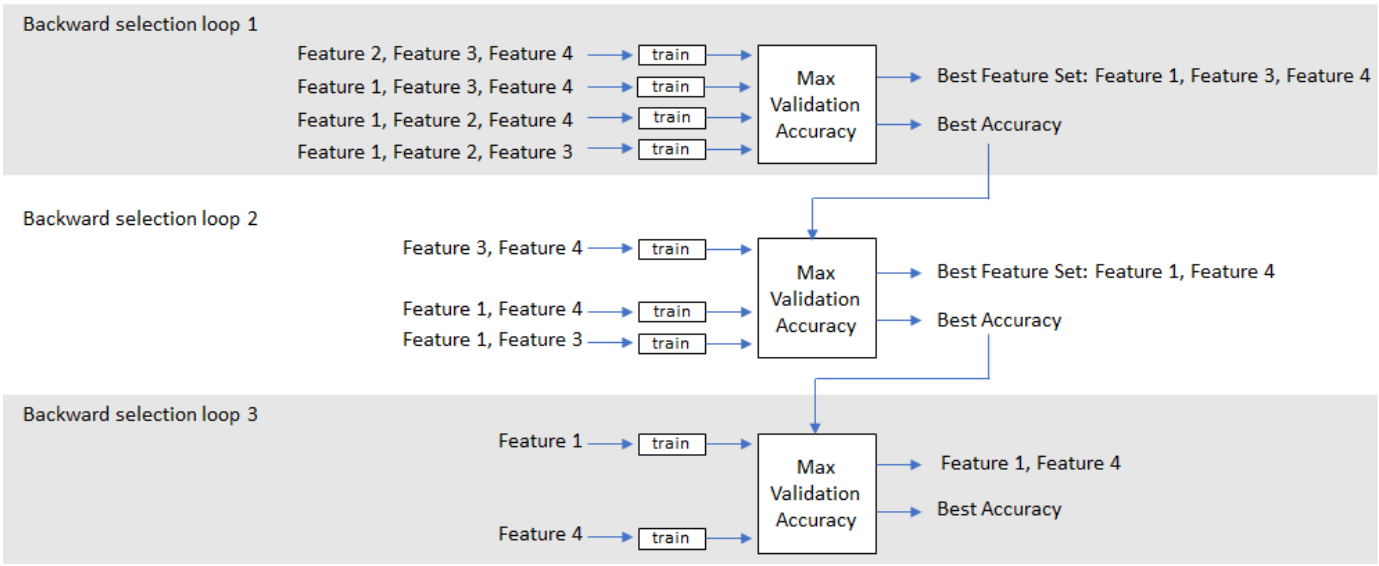
Forward Selection

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 12-0 , HelperTrainAndValidateNetwork on page 12-0 , and HelperTrimOrPad on page 12-0) 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 = ;
[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       - traingOptions 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     - traingOptions 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`) and Gaussian mixture models (GMMs) to describe their statistical distribution. Other popular features used for ASC include zero crossing rate, spectral centroid (`spectralCentroid`), spectral rolloff (`spectralRolloffPoint`), spectral flux (`spectralFlux`), and linear prediction coefficients (`lpc`) [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`). 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 6.5 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` to the acoustic scene. Call `countEachLabel` 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` 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` 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` 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, ...
    'WindowLength',windowLength, ...
    'OverlapLength',samplesOverlap, ...
    'FFTLength',fftLength, ...
    'NumBands',numBands);
```

Convert the mel spectrogram into the logarithmic scale.

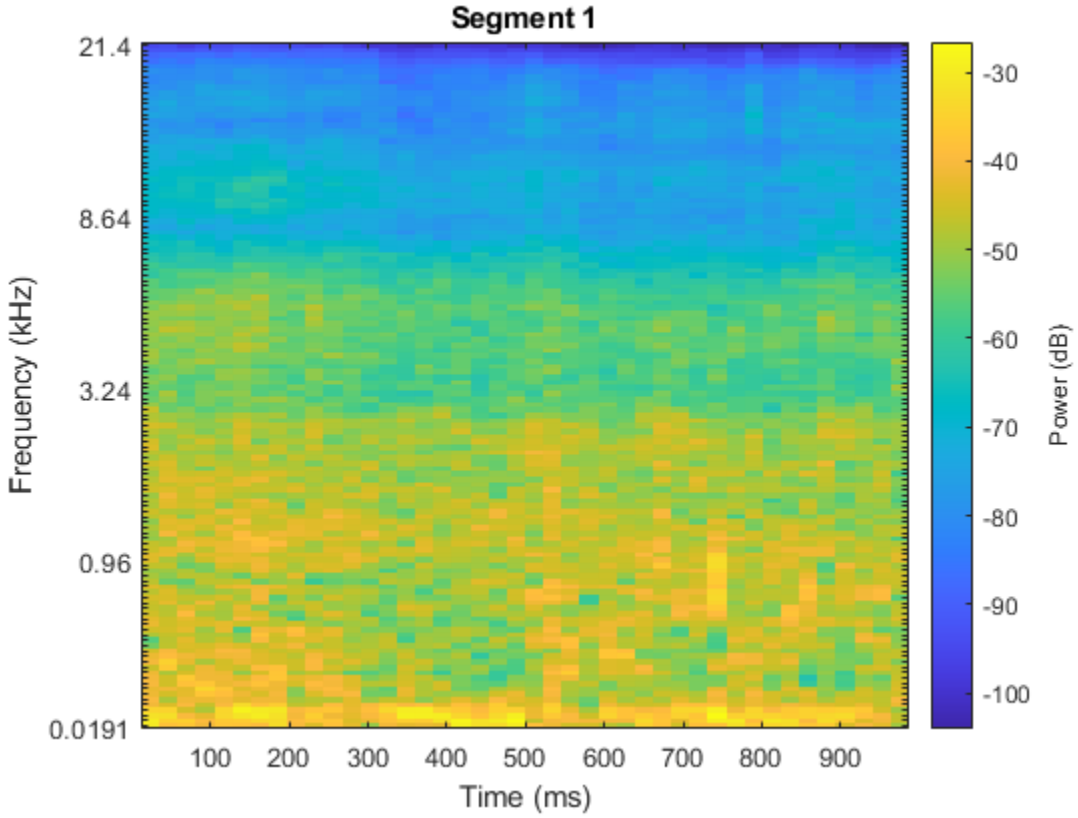
```
spec = log10(spec+eps);
```

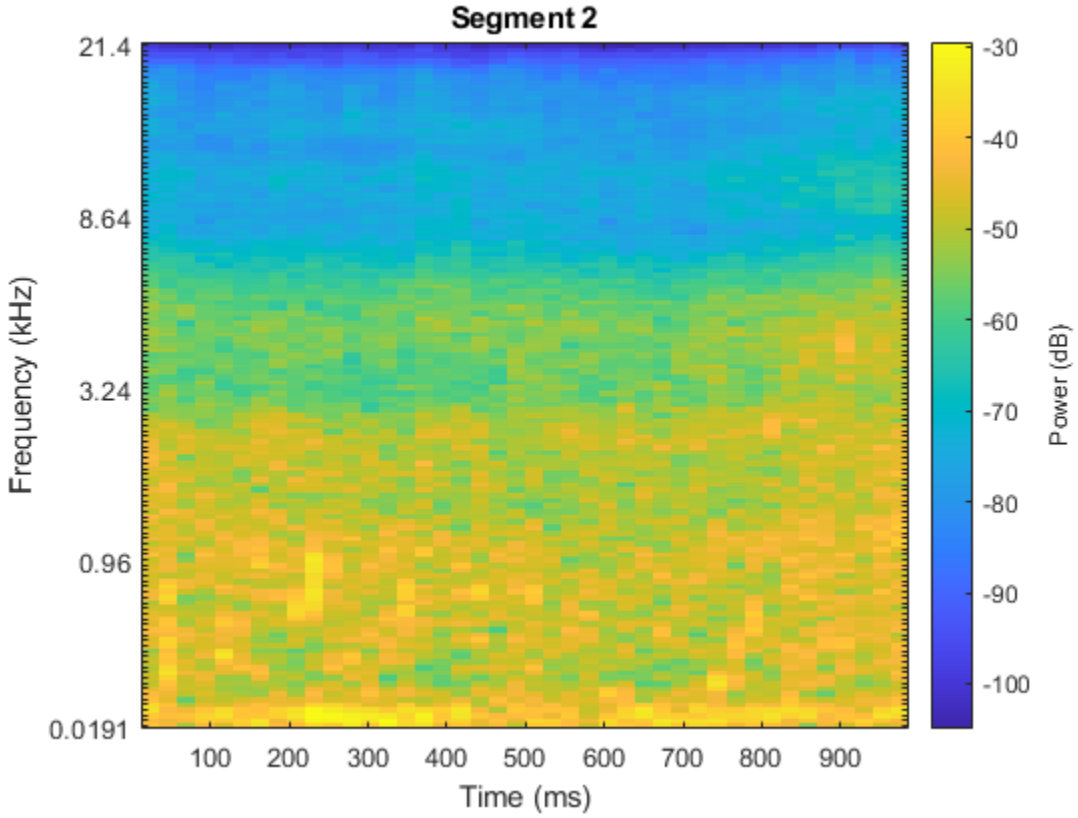
Reshape the array to dimensions (Number of bands)-by-(Number of hops)-by-(Number of channels)-by-(Number of segments). When you feed an image into a neural network, the first two dimensions are the height and width of the image, the third dimension is the channels, and the fourth dimension separates the individual images.

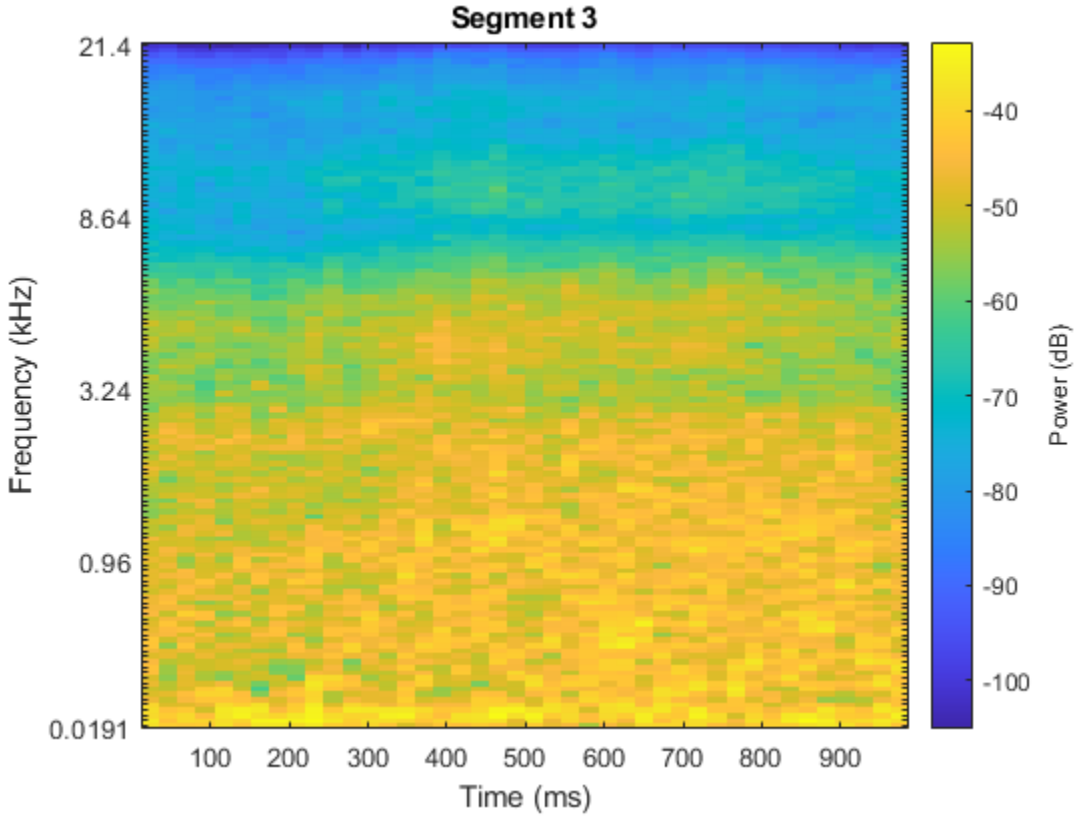
```
X = reshape(spec,size(spec,1),size(spec,2),size(data,2),[]);
```

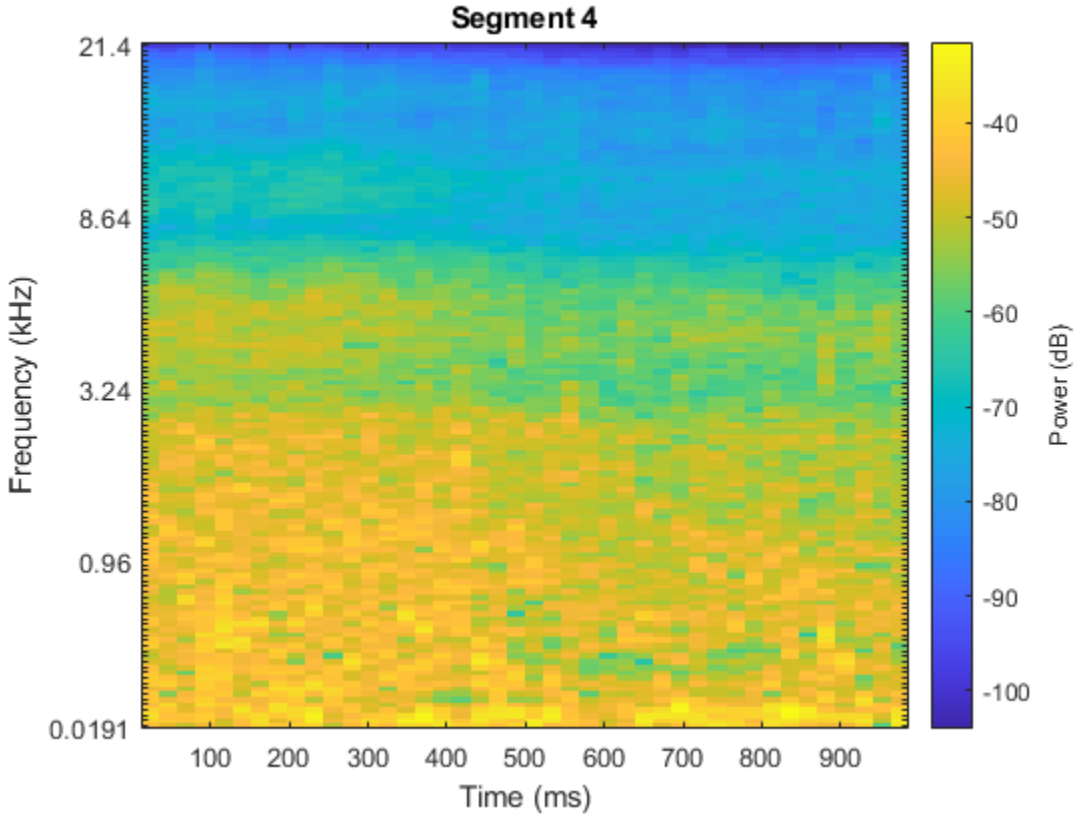
Call `melSpectrogram` without output arguments to plot the mel spectrogram of the mid channel for the first six of the one-second increments.

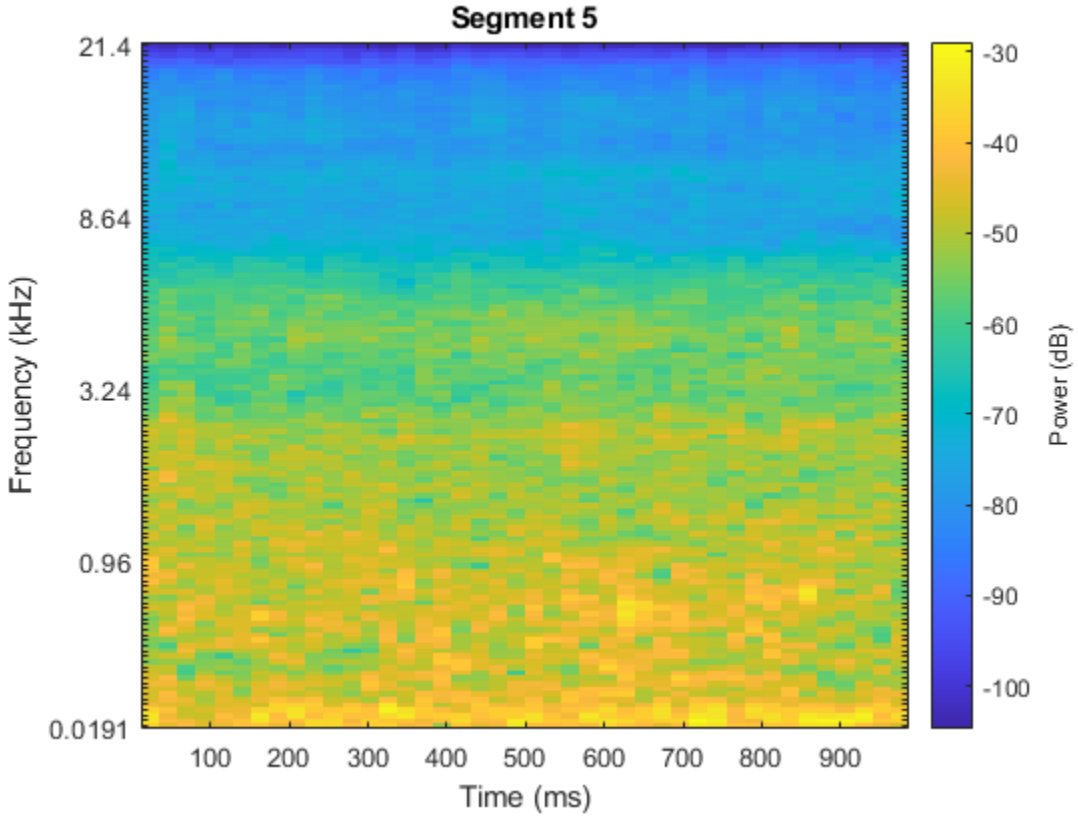
```
for channel = 1:2:11
    figure
    melSpectrogram(dataBuffered(:,channel),fs, ...
        'WindowLength',windowLength, ...
        'OverlapLength',samplesOverlap, ...
        'FFTLength',fftLength, ...
        'NumBands',numBands);
    title(sprintf('Segment %d',ceil(channel/2)))
end
```

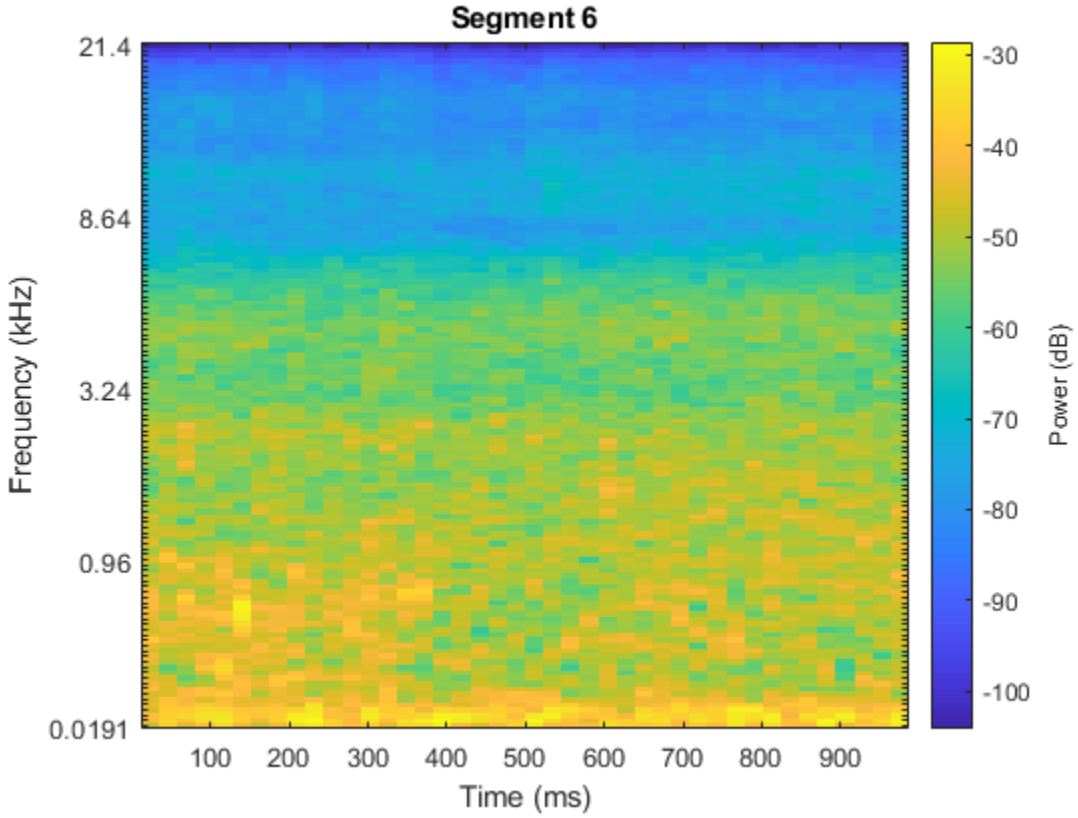

Segment 1
21.4
8.64
3.24
0.96
0.0191
Frequency (kHz)
100
200
300
400
500
600
700
800
900
Time (ms)
-30
-40
-50
-60
-70
-80
-90
-100
Power (dB)


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 4 min 43 sec
Evaluation completed in 4 min 43 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 40 sec
Evaluation completed in 1 min 40 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                11802
     bus                  11901
     cafe/restaurant      12026
     car                  11984
     city_center          11956
     forest_path          11712
     grocery_store        11873
     home                 11738
     library              11655
     metro_station        11867
     office               11876
     park                 11859
     residential_area     11831
     train                11926
     tram                 11834
```

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 1-23 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
```

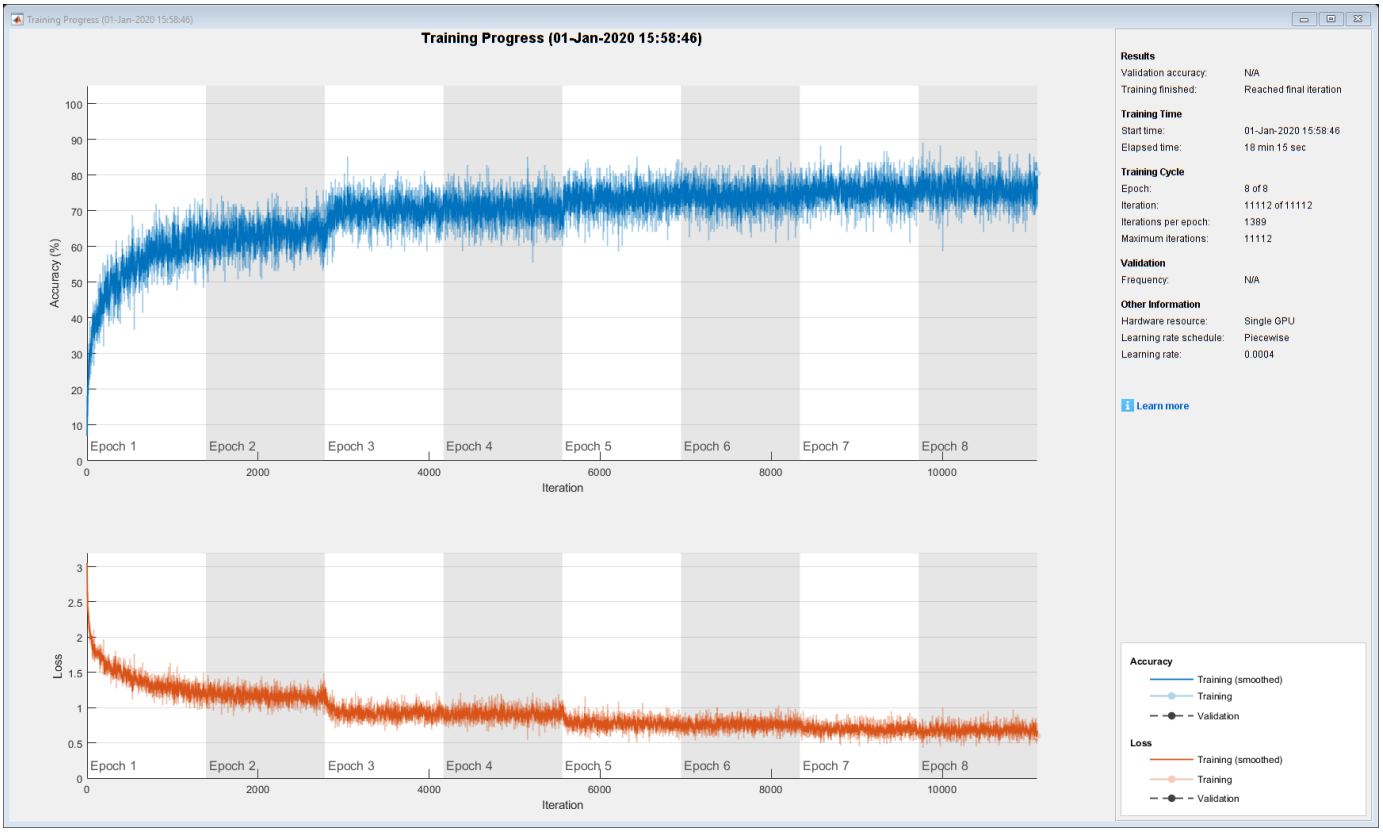
```
    convolution2dLayer(3,2*numF,'Padding','same')
    batchNormalizationLayer
    reluLayer

    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

    averagePooling2dLayer(ceil(imgSize(1:2)/8))

    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.

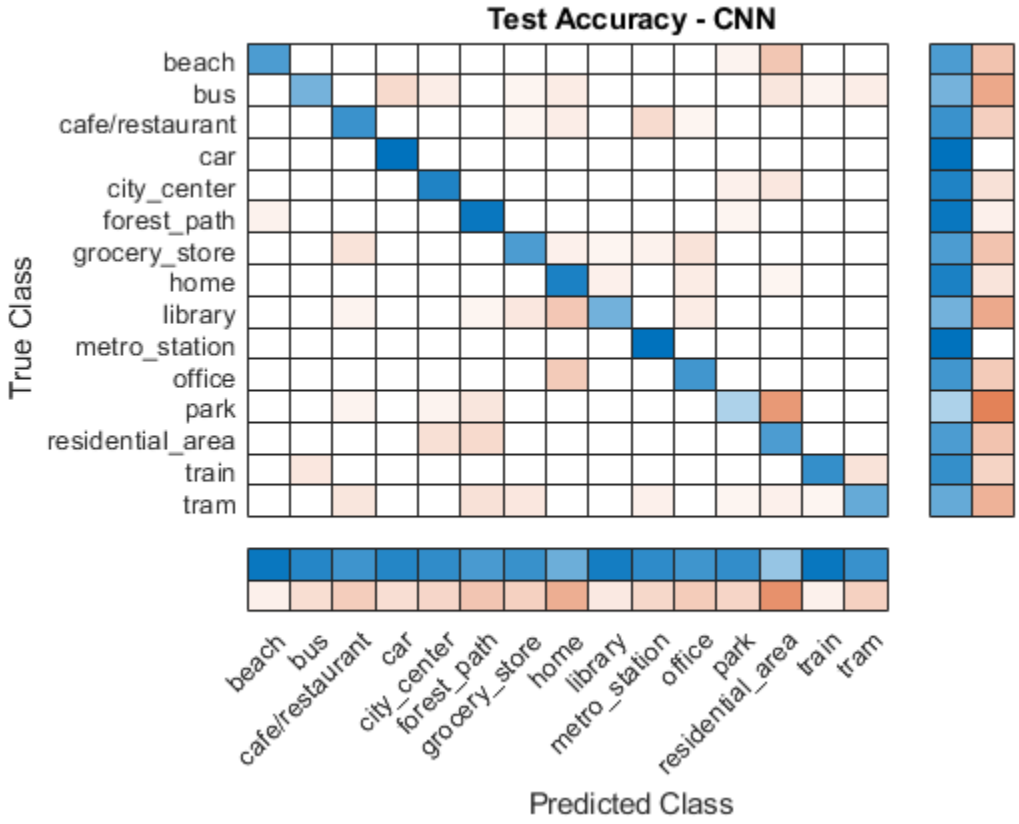
```
[~,classIdx] = max(cnnResponses,[],2);
cnnPredictedLabels = classes(classIdx);
```

Call `confusionchart` to visualize the accuracy on the test set. Return the average accuracy to the Command Window.

```
figure
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 = 73.15
```



Feature Extraction for Ensemble Classifier

Wavelet scattering has been shown in [4] to provide a good representation of acoustic scenes. Define a `waveletScattering` 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` 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 30 min 10 sec
Evaluation completed in 30 min 11 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 10 min 41 sec
Evaluation completed in 10 min 41 sec
```

Define and Train Ensemble Classifier

Use `fitcensemble` to create a trained classification ensemble model (`ClassificationEnsemble`).

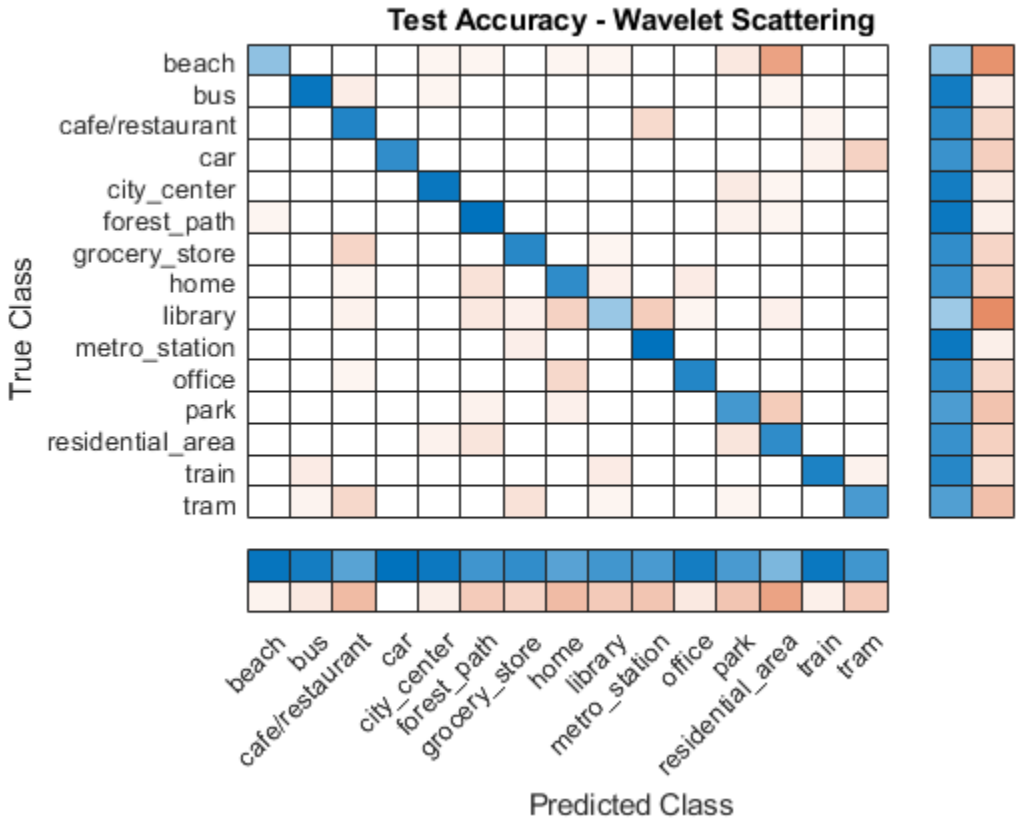
```
subspaceDimension = min(150,size(xTrain,2) - 1);
numLearningCycles = 30;
classificationEnsemble = fitcensemble(xTrain,adsTrain.Labels, ...
    'Method','Subspace', ...
    'NumLearningCycles',numLearningCycles, ...
    'Learners','discriminant', ...
    'NPredToSample',subspaceDimension, ...
    '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
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 = 76.11
```



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.

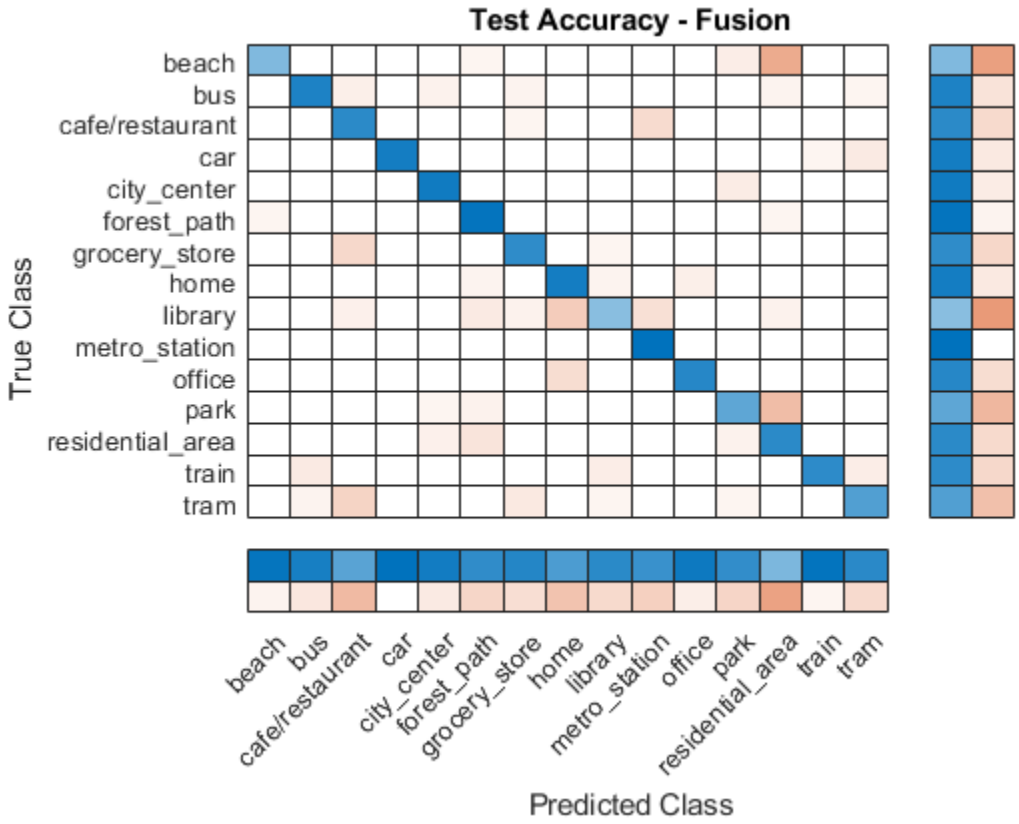
```
fused = waveletResponses .* cnnResponses;
[~,classIdx] = max(fused,[],2);

predictedLabels = classes(classIdx);
```

Evaluate Late Fusion

Call `confusionchart` to visualize the fused classification accuracy. Print the average accuracy to the Command Window.

```
figure
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 = 79.51
```



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, ...
    'WindowLength',params.WindowLength, ...
    '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

`batchNormalizationLayer` | `classify` | `convolution2dLayer` | `layerGraph` | `trainNetwork` | `trainingOptions`

Related Examples

- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

- “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 load a pretrained network by opening this example in MATLAB® and typing `load("KWSNet.mat")` at the command line.

Example Summary

The example 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`.
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

In this example, the keyword to spot is **YES**.

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.

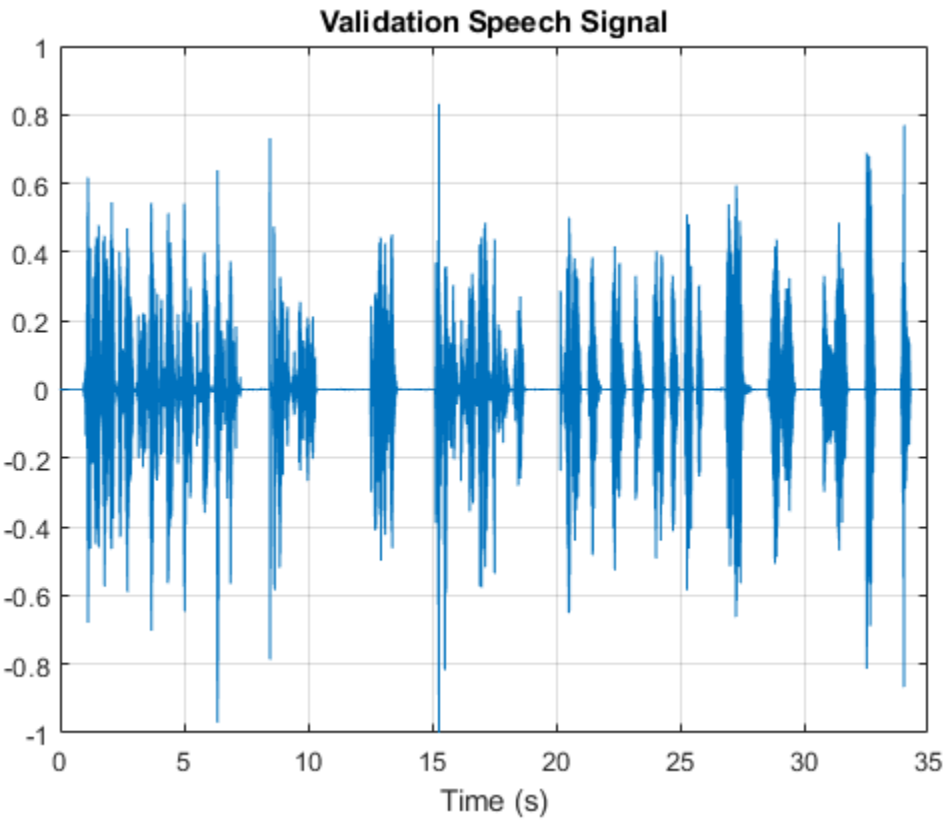
```
[audioIn,fs] = audioread('KeywordSpeech-16-16-mono-34secs.flac');
```

Listen to the signal.

```
sound(audioIn,fs)
```

Visualize the signal.

```
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).

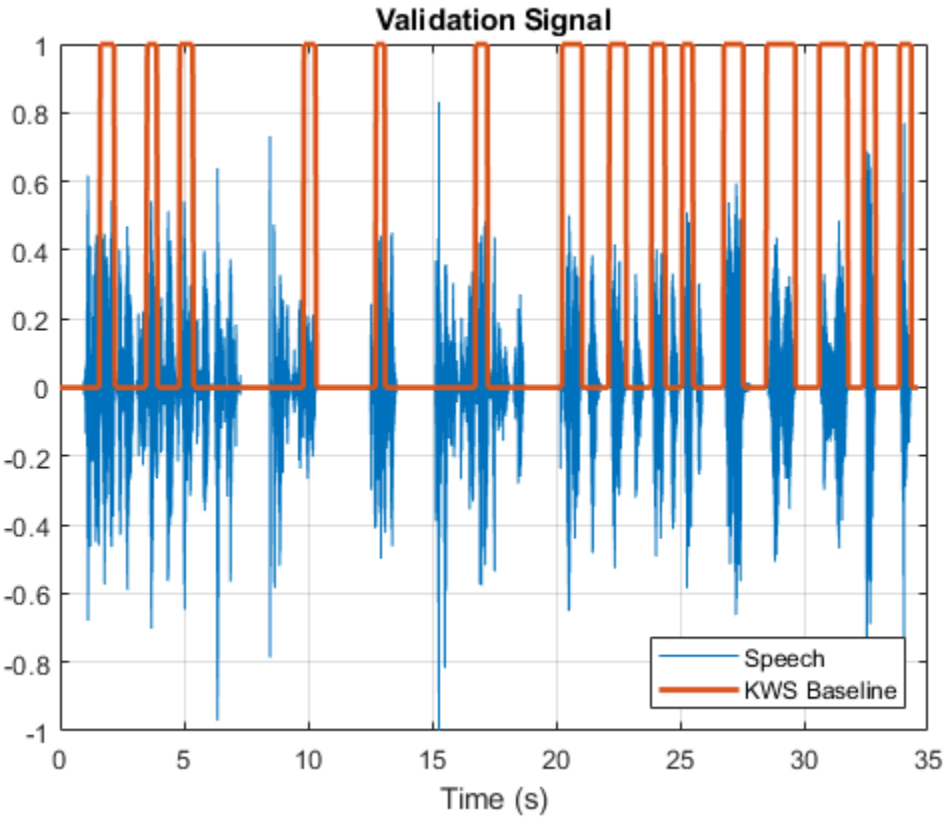
```
load('KWSBaseline.mat','KWSBaseline')
```

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 12-0 .

```
url = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz'

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading Google speech commands data set (1.5 GB)...')
    untar(url,datasetFolder)
end
```

Create an `audioDatastore` that points to the data set.

```
ads = audioDatastore(datasetFolder,'LabelSource','foldername','Includesubfolders',true);
```

The dataset contains background noise files that are not used in this example. Use `subset` to create a new datastore that does not have the background noise files.

```
isBackNoise = ismember(ads.Labels,"_background_noise_");
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`.

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.

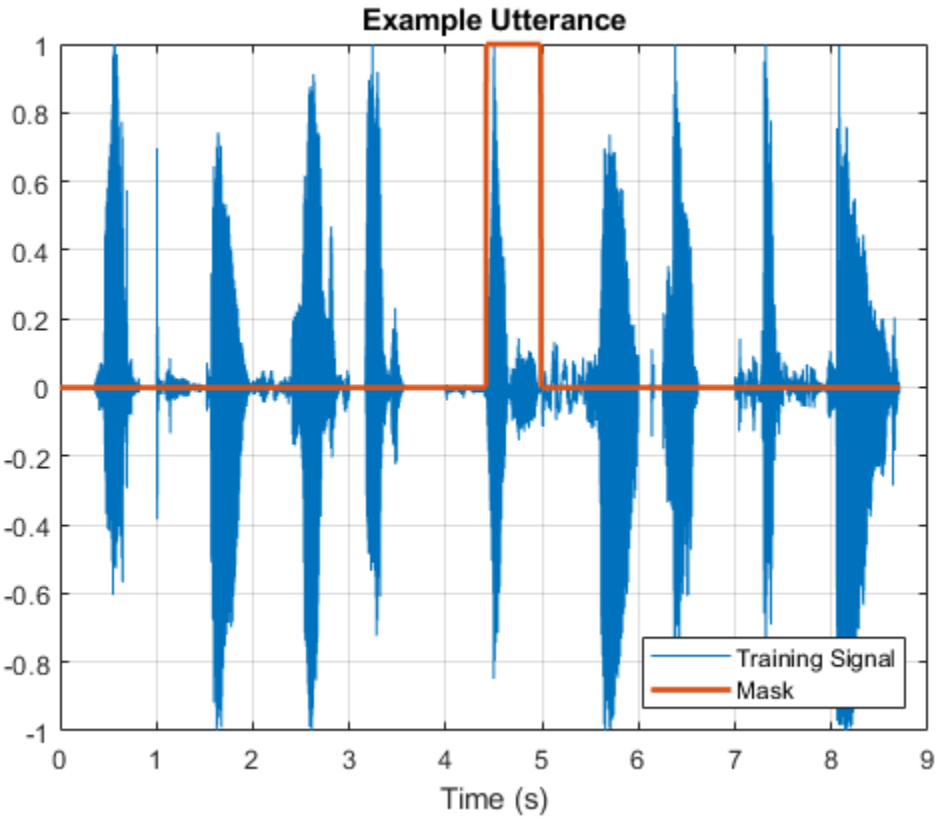
```
keywordLocation = randi([1 numWords+1]);
```

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 42 MFCC coefficients (14 MFCC, 14 delta and 14 delta-delta coefficients).

Define parameters required for MFCC extraction.

```
WindowLength = 512;
OverlapLength = 384;
```

Extract the MFCC features.

```
[coeffs,delta,deltaDelta] = mfcc(sentence,fs,'WindowLength',WindowLength,'OverlapL
```

Concatenate the coefficients into one feature matrix.

```
featureMatrix = [coeffs, delta, deltaDelta];
size(featureMatrix)
```

```
ans = 1×2

        1085          42
```

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(coeffs,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`.

```
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
        [coeffs,delta,deltaDelta] = mfcc(sentence,fs,'WindowLength',WindowLength,'OverlapLength'
        featureMatrix = [coeffs, delta, deltaDelta];
        featureMatrix(~isfinite(featureMatrix)) = 0;

        % Create mask
        hopLength = WindowLength - OverlapLength;
        range = (hopLength) * (1:size(coeffs,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;
        catVect = categorical(featureMask);
        catVect = addcats(catVect,{'1'});
        localMasks{count} = catVect;

        count = count + 1;

    end

    TrainingFeatures = [TrainingFeatures;localFeatures];
    TrainingMasks = [TrainingMasks;localMasks];

end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
fprintf('Training feature extraction took %f seconds.\n',toc)
```

```
Training feature extraction took 101.747986 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{:});
M = mean(featuresMatrix);
S = std(featuresMatrix);
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.

```
[coeffs,delta,deltaDelta] = mfcc(audioIn,fs,'WindowLength',WindowLength,'OverlapLength',OverlapLe
featureMatrix = [coeffs, delta, deltaDelta];
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+WindowLengtl
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.

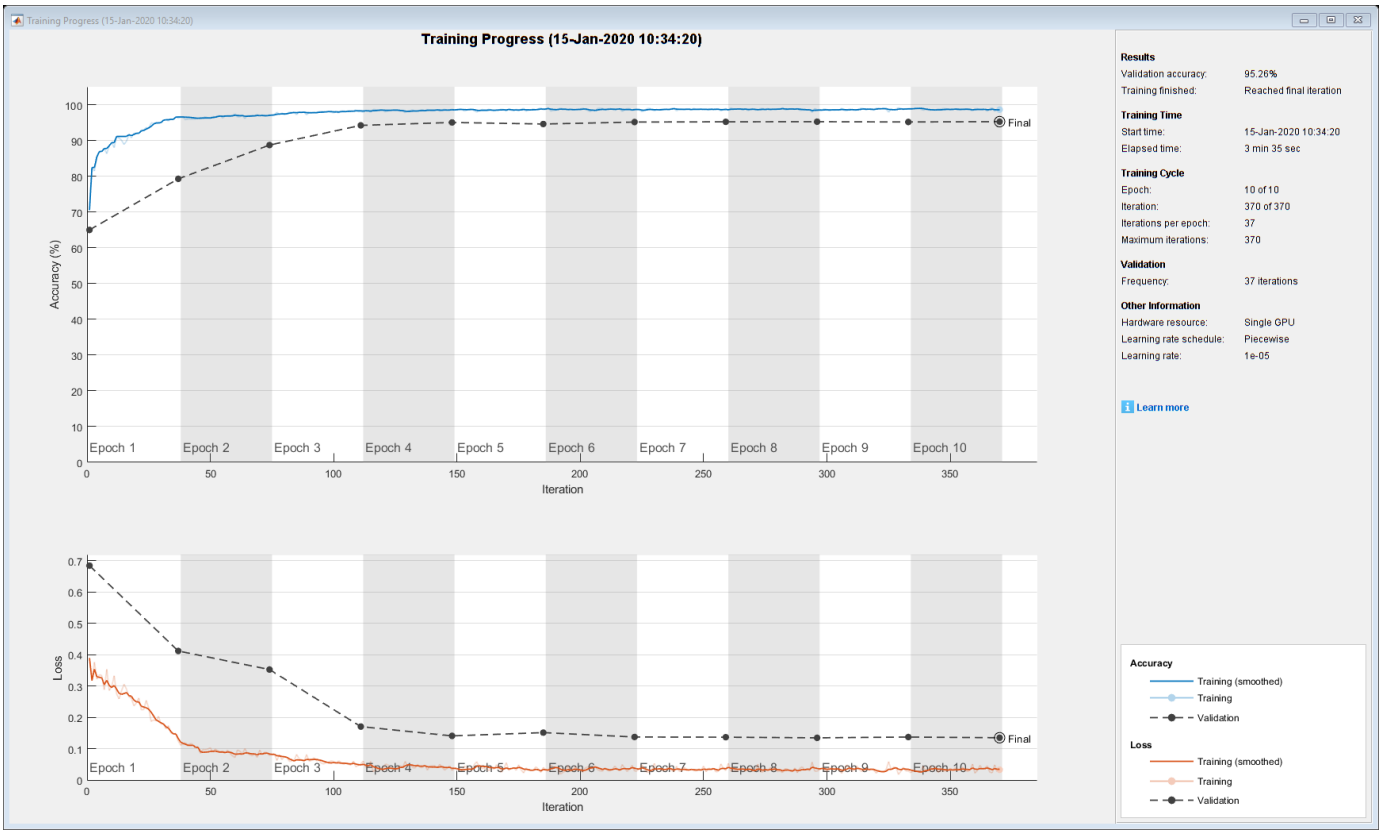
```
maxEpochs = 10;
miniBatchSize = 64;
options = trainingOptions("adam", ...
    "InitialLearnRate",1e-4, ...
    "MaxEpochs",maxEpochs, ...
    "MiniBatchSize",miniBatchSize, ...
    "Shuffle","every-epoch", ...
    "Verbose",false, ...
    "ValidationFrequency",floor(numel(TrainingFeatures)/miniBatchSize), ...
    "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.

```
if ~reduceDataset
    [keywordNetNoAugmentation,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,optio
```

```
    fprintf("Validation accuracy: %f percent.\n",netInfo.FinalValidationAccuracy);
else
    load('keywordNetNoAugmentation.mat','keywordNetNoAugmentation','M','S');%#ok
end
```




```
Validation accuracy: 95.255728 percent.
```

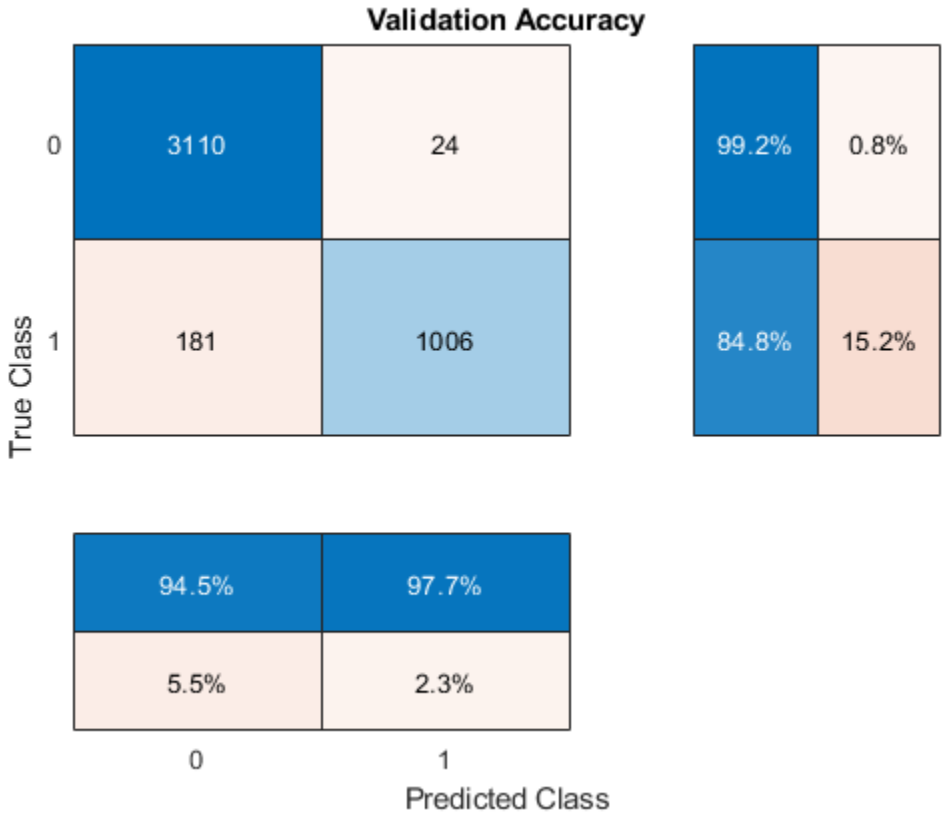
Check Network Accuracy for a Noise-Free Validation Signal

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.

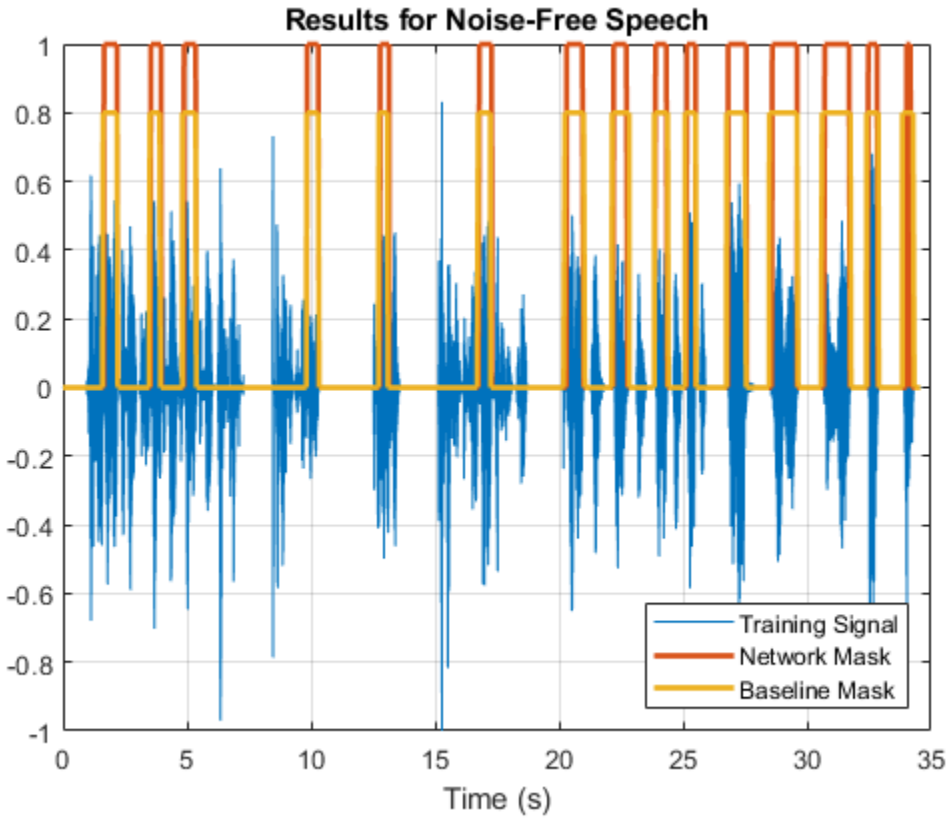
```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

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

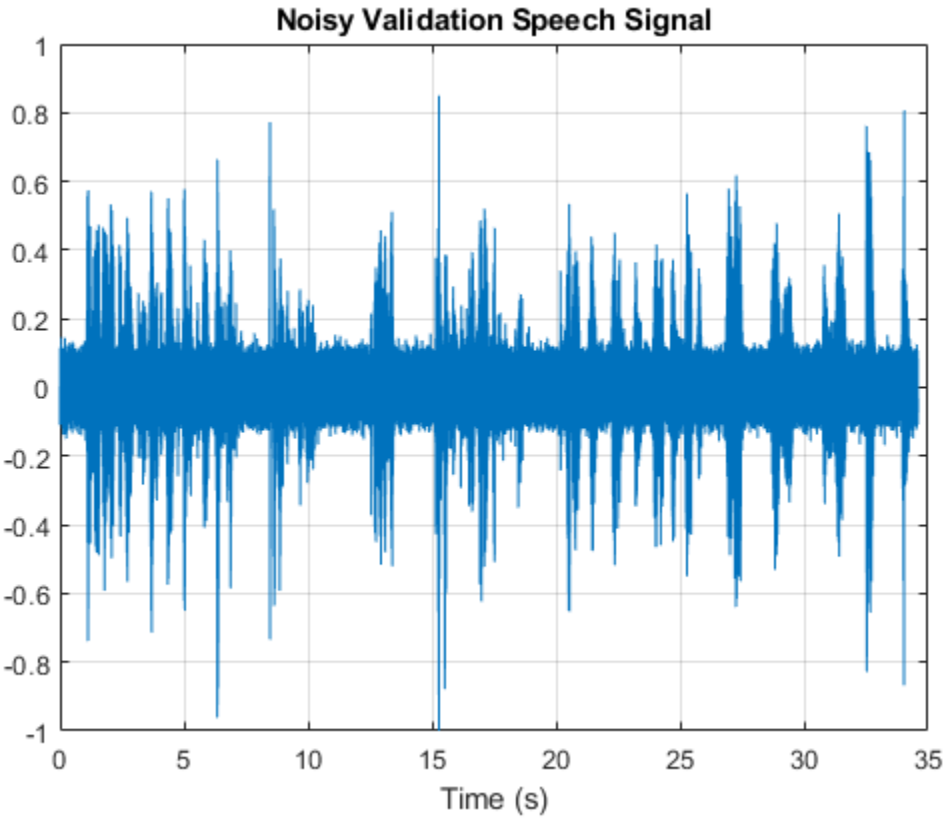
You will now check the network accuracy for a noisy speech signal. The noisy signal was obtained by corrupting the clean validation signal by additive white Gaussian noise.

Load the noisy signal.

```
[audioInNoisy,fs] = audioread('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.

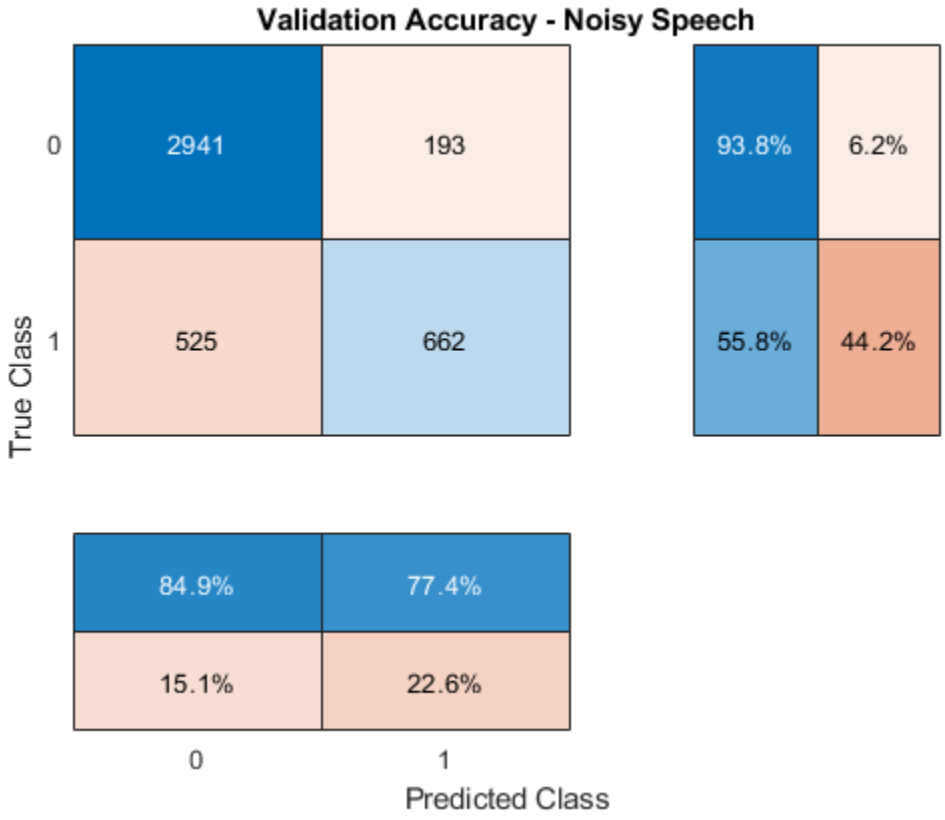
```
[coeffs,delta,deltaDelta] = mfcc(audioInNoisy,fs,'WindowLength',WindowLength,'OverlapLength',Ove
featureMatrixV = [coeffs, delta, deltaDelta];
featureMatrixV(~isfinite(featureMatrixV)) = 0;
FeaturesValidationNoisy = (featureMatrixV - M)./S;
```

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.

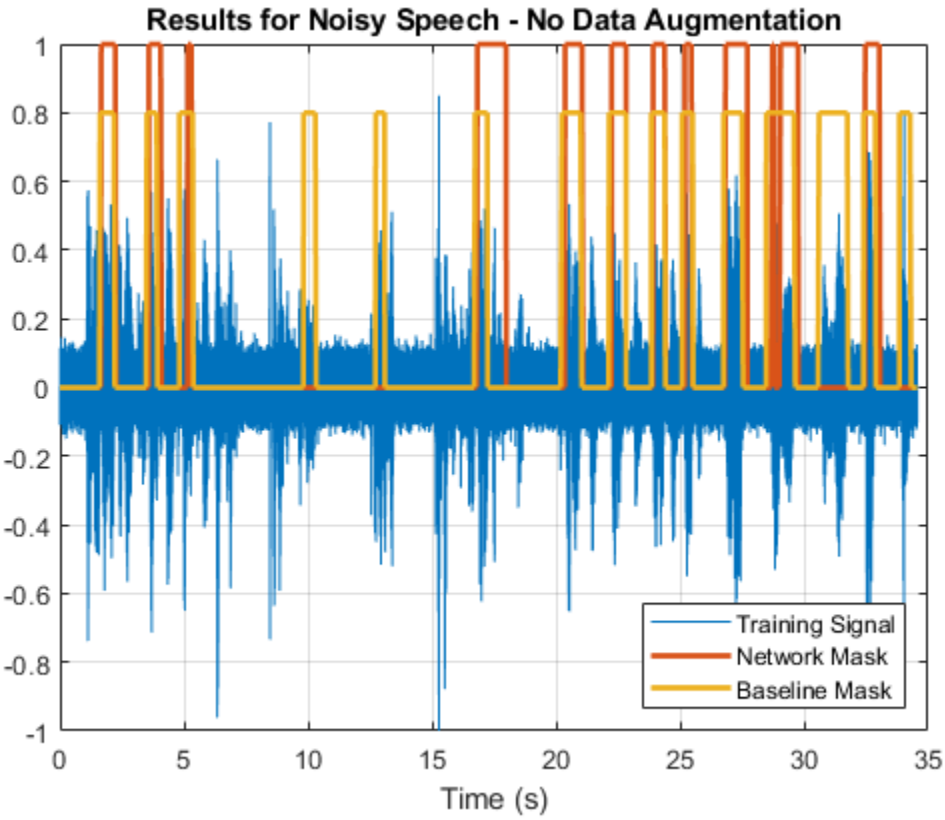
```
v = double(v) - 1;
v = repmat(v,HopLength,1);
v = v(:);
```

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` 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: -0.1243
```

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
        [coeffs,delta,deltaDelta] = mfcc(sentence,fs,'WindowLength',WindowLength,'OverlapLength'
        featureMatrix = [coeffs delta deltaDelta];
        featureMatrix(~isfinite(featureMatrix)) = 0;

        hopLength = WindowLength - OverlapLength;
        range = hopLength * (1:size(coeffs,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;
        catVect = categorical(featureMask);
        catVect = addcats(catVect,{'1'});
        localMasks{count} = catVect;
```

```
        count = count + 1;

    end

    TrainingFeatures = [TrainingFeatures;localFeatures];
    TrainingMasks = [TrainingMasks;localMasks];

end
fprintf('Training feature extraction took %f seconds.\n',toc)
```

```
Training feature extraction took 41.517132 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{:});
M = mean(featuresMatrix);
S = std(featuresMatrix);
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;
```

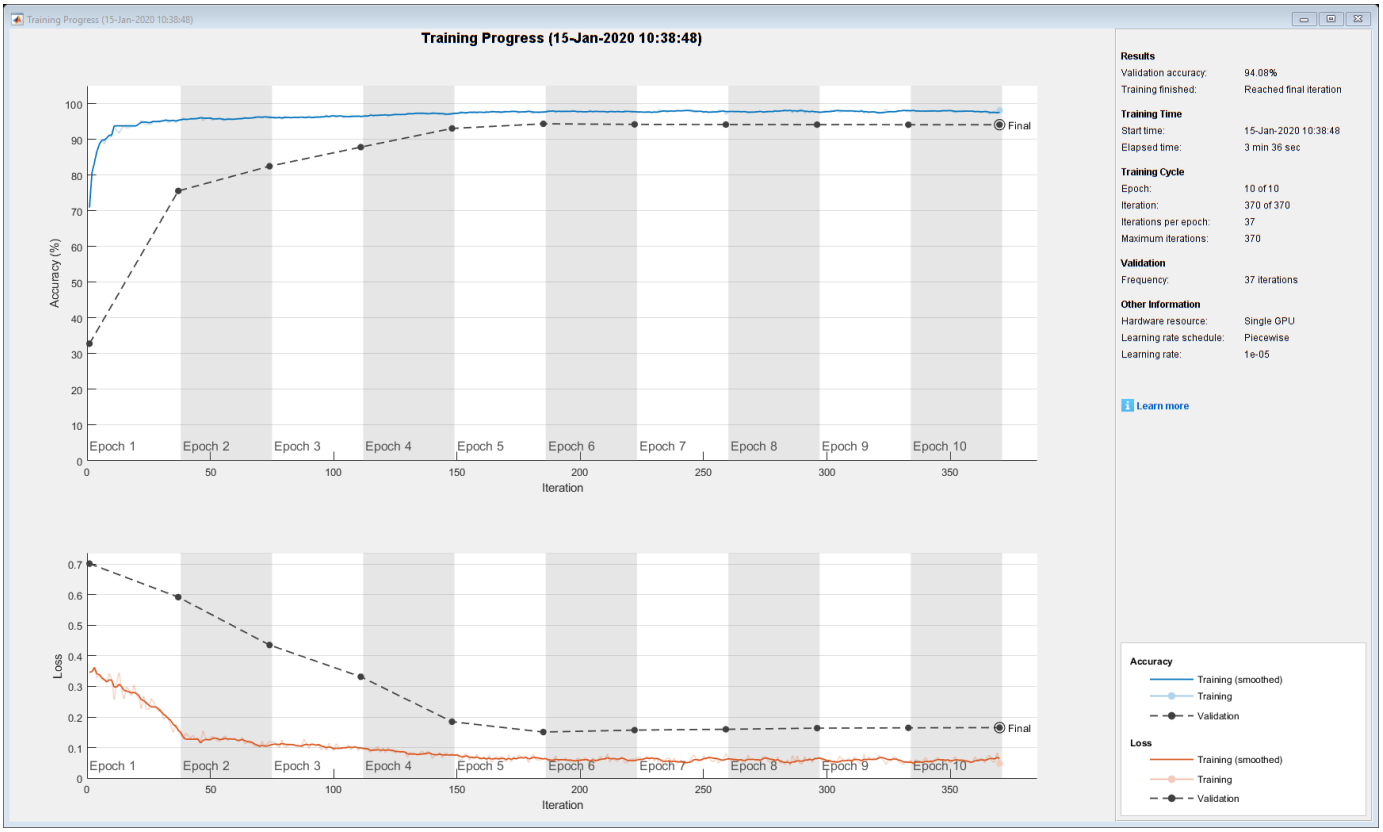
Retrain Network with Augmented Dataset

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.

```
if ~reduceDataset
    [KWSNet,netInfo] = trainNetwork(TrainingFeatures,TrainingMasks,layers,options);
else
    load('KWSNet.mat','KWSNet');%#ok
end
```

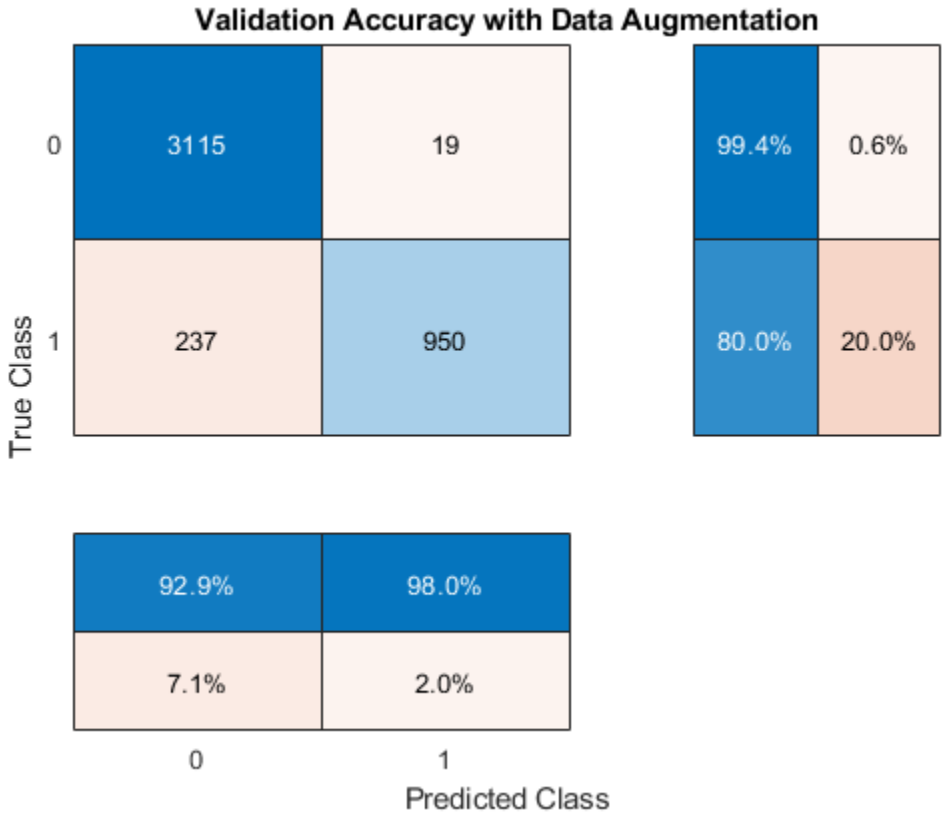


Verify the network accuracy on the validation signal.

```
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";
```

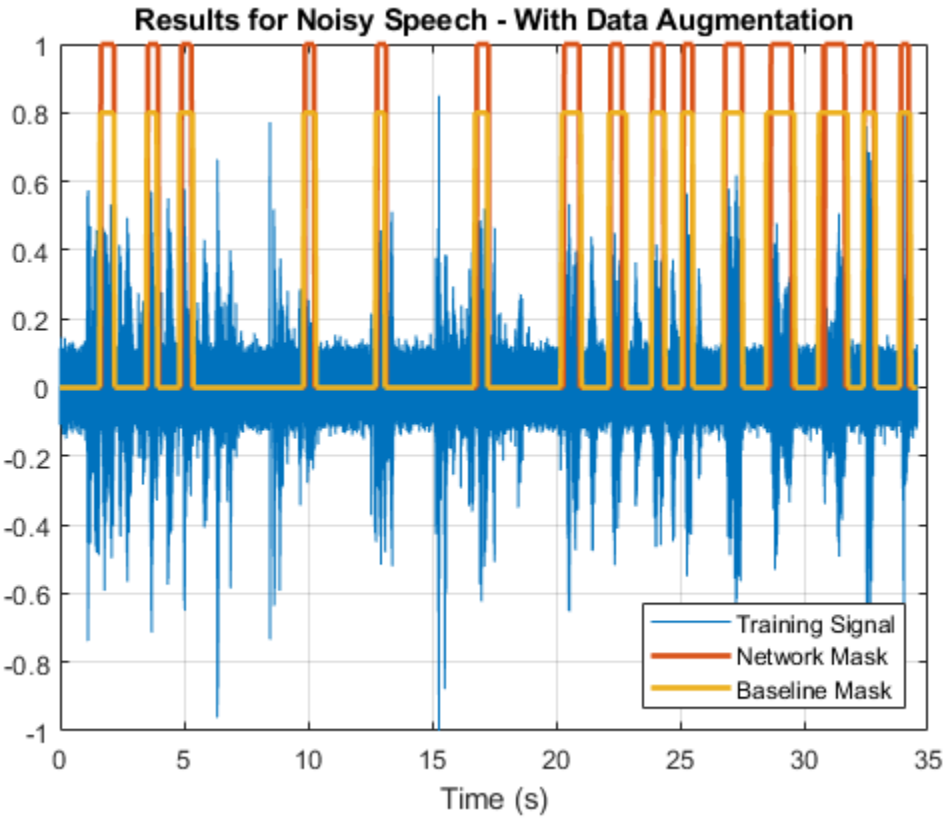


Listen to the identified keyword regions.

```
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` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Long Short-Term Memory Networks” on page 1-53
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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 12-0 .

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 12-0 . 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` 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

Load the pretrained network, the `audioFeatureExtractor` 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`.

```
load('network_Audio_SER.mat','net','afe','normalizers');
```

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 = ;
emotion = ;

adsSubset = subset(ads,ads.Labels.Speaker==speaker & ads.Labels.Emotion == emotion);

audio = read(adsSubset);
sound(audio,fs)
```

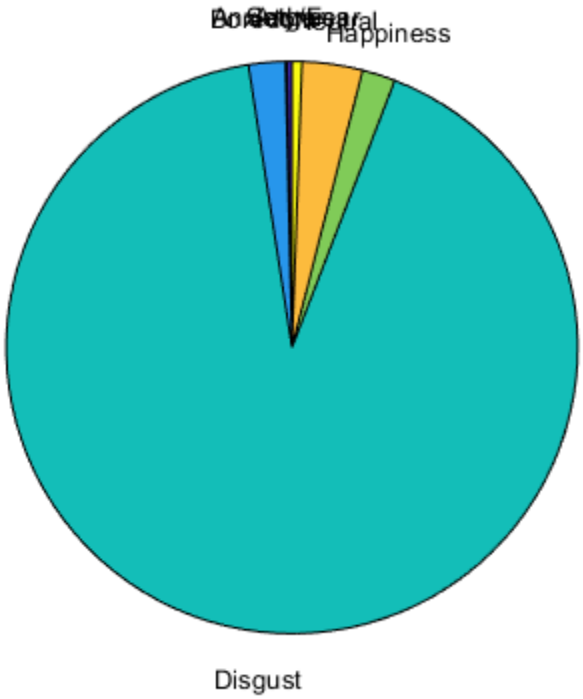
Use the `audioFeatureExtractor` object to extract the features and then transpose them so that time is along rows. Normalize the features and then convert them to 20-element sequences with 10-element overlap, which corresponds to approximately 600 ms windows with 300 ms overlap. Use the supporting function, HelperFeatureVector2Sequence on page 12-0 , to convert the array of feature vectors to sequences.

```
features = (extract(afe,audio))';

featuresNormalized = (features - normalizers.Mean)./normalizers.StandardDeviation;
```

```
numOverlap = ;
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 = ;
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`. 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 = ;
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 (59.69 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` 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 5 min 14 sec
Evaluation completed in 5 min 14 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.

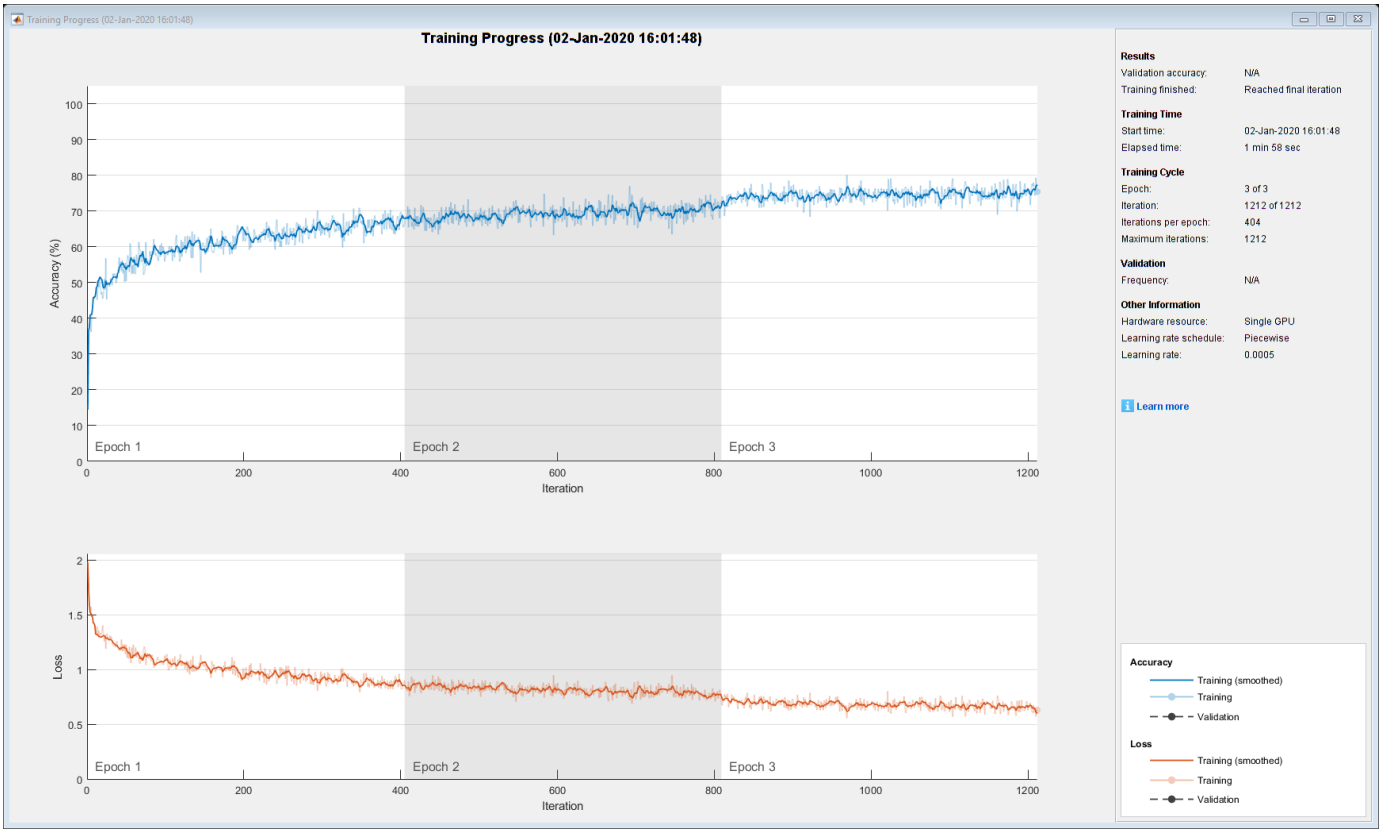
```
dropoutProb1 = 0.3;
numUnits = 200;
dropoutProb2 = 0.6;
layers = [ ...
    sequenceInputLayer(size(sequencesTrain{1},1))
    dropoutLayer(dropoutProb1)
    bilstmLayer(numUnits,"OutputMode","last")
    dropoutLayer(dropoutProb2)
    fullyConnectedLayer(numel(categories(emptyEmotions)))
    softmaxLayer
    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 = ;
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 12-0 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);
```

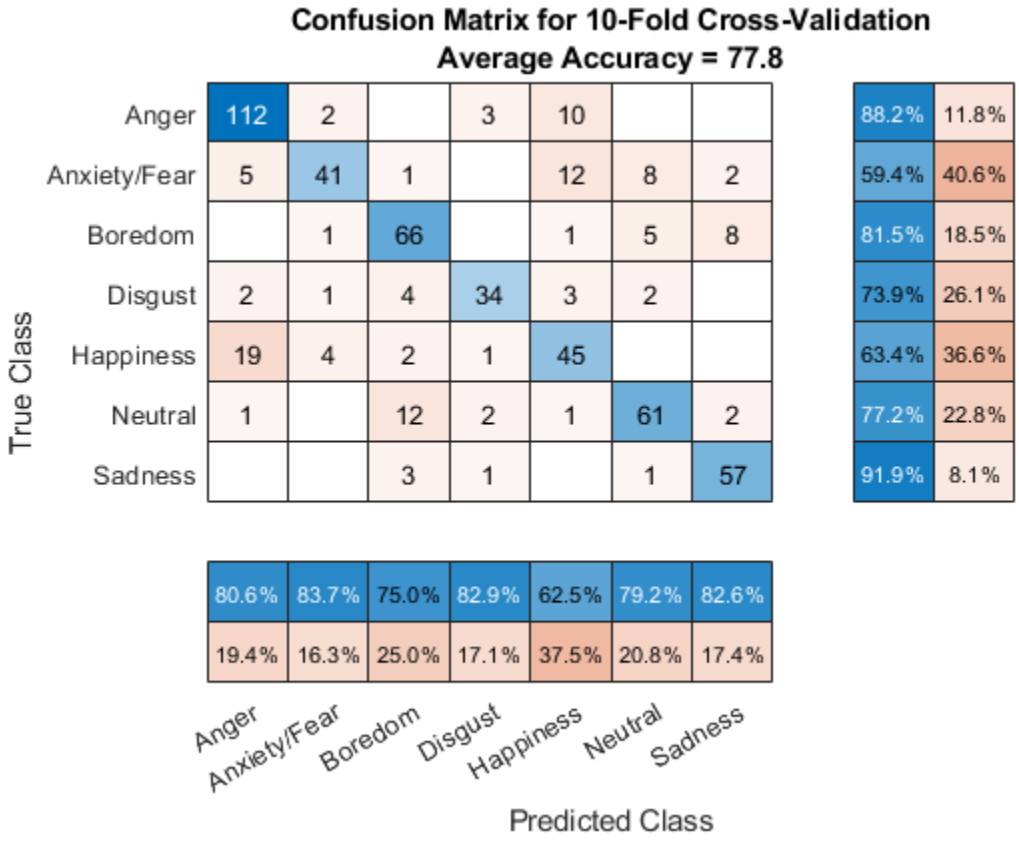
Print the accuracy per fold and plot the 10-fold confusion chart.

```
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 = 81.6
Fold 2, Accuracy = 77.6
Fold 3, Accuracy = 83.7
Fold 4, Accuracy = 78.9
Fold 5, Accuracy = 81.8
Fold 6, Accuracy = 71.4
Fold 7, Accuracy = 70.5
Fold 8, Accuracy = 91.3
Fold 9, Accuracy = 73.2
Fold 10, Accuracy = 67.6
```

```
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',val
```

```
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<
            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,featu
[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` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Long Short-Term Memory Networks” on page 1-53
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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','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.

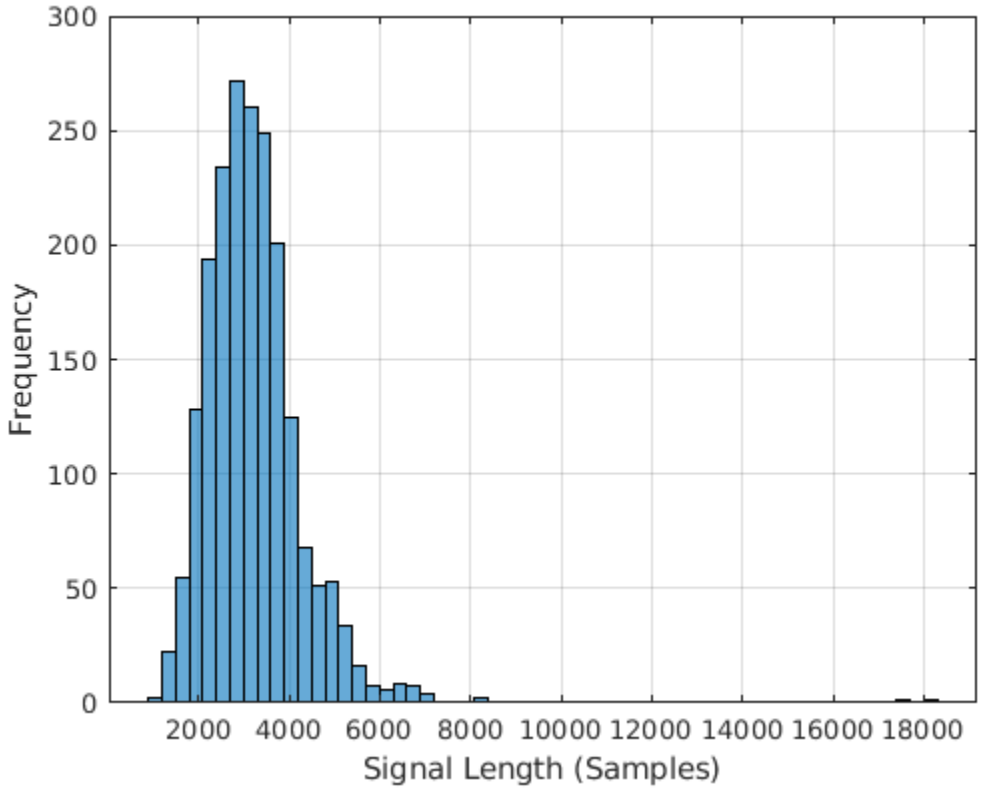
```
ads.Labels = helpergenLabels(ads);
summary(ads.Labels)
```

```
     0      200
     1      200
     2      200
     3      200
     4      200
     5      200
     6      200
     7      200
     8      200
     9      200
```

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` 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       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
```

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 = 96.8125
```

The estimated generalization accuracy is approximately 97%. Use the trained SVM to predict the spoken-digit classes in the test set.

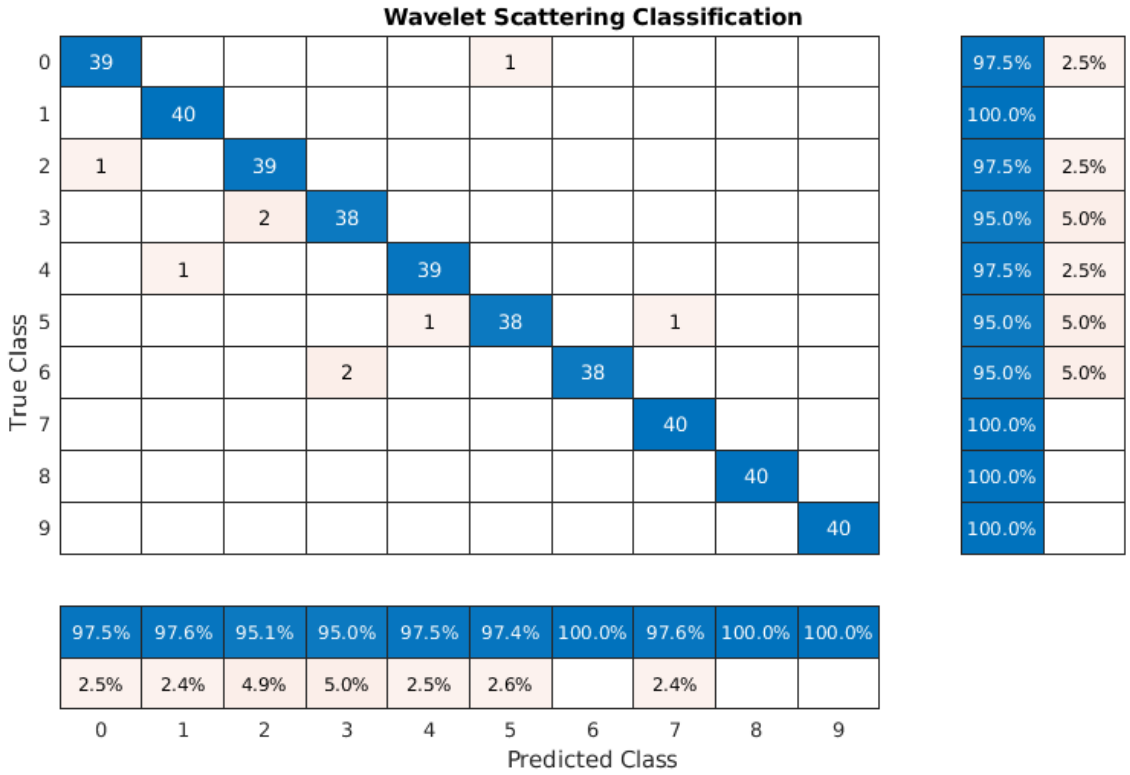
```
predLabels = predict(classificationSVM,TestFeatures);
testAccuracy = sum(predLabels==adsTest.Labels)/numel(predLabels)*100
```

```
testAccuracy = 97.7500
```

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.

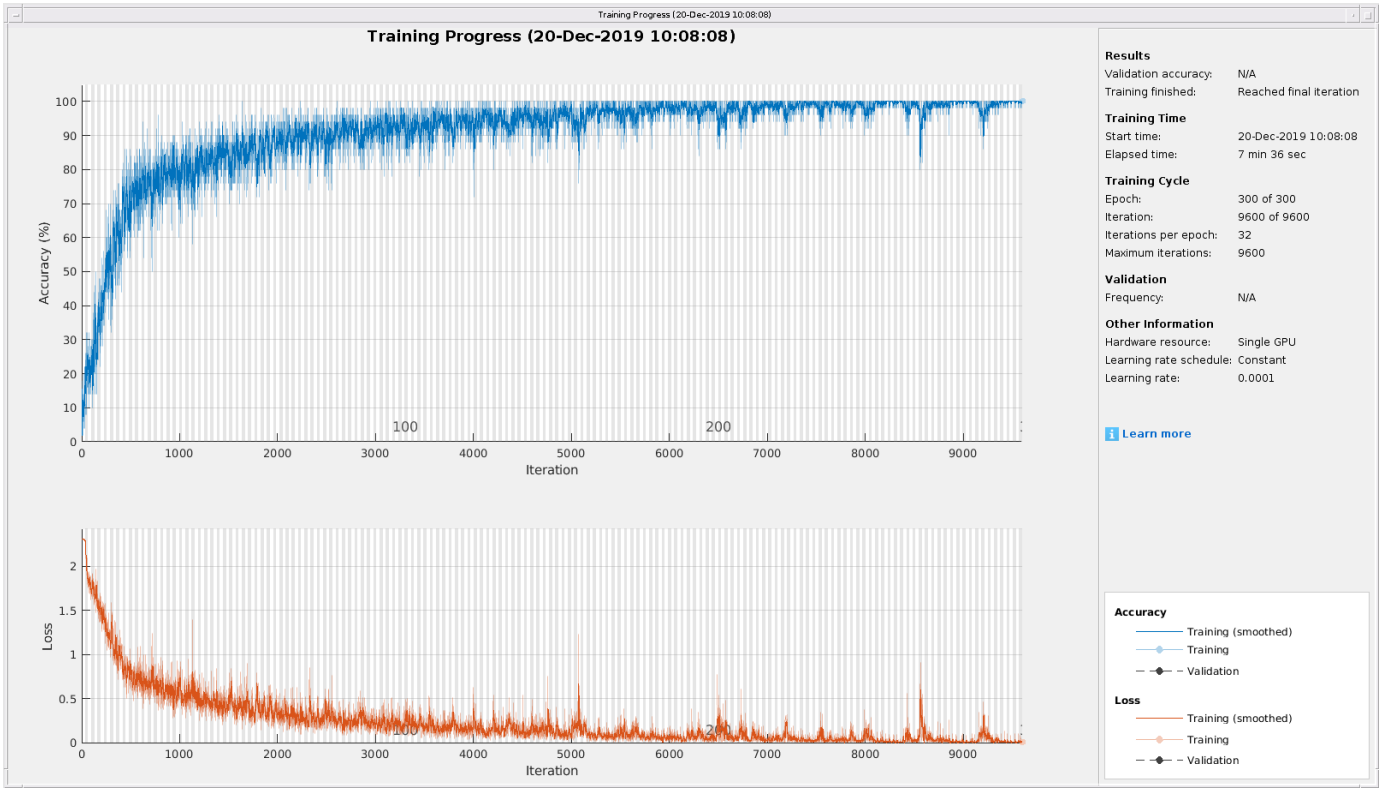
```
[inputSize, ~] = size(TrainFeatures{1});
YTrain = adsTrain.Labels;

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 = 94
```

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'`.

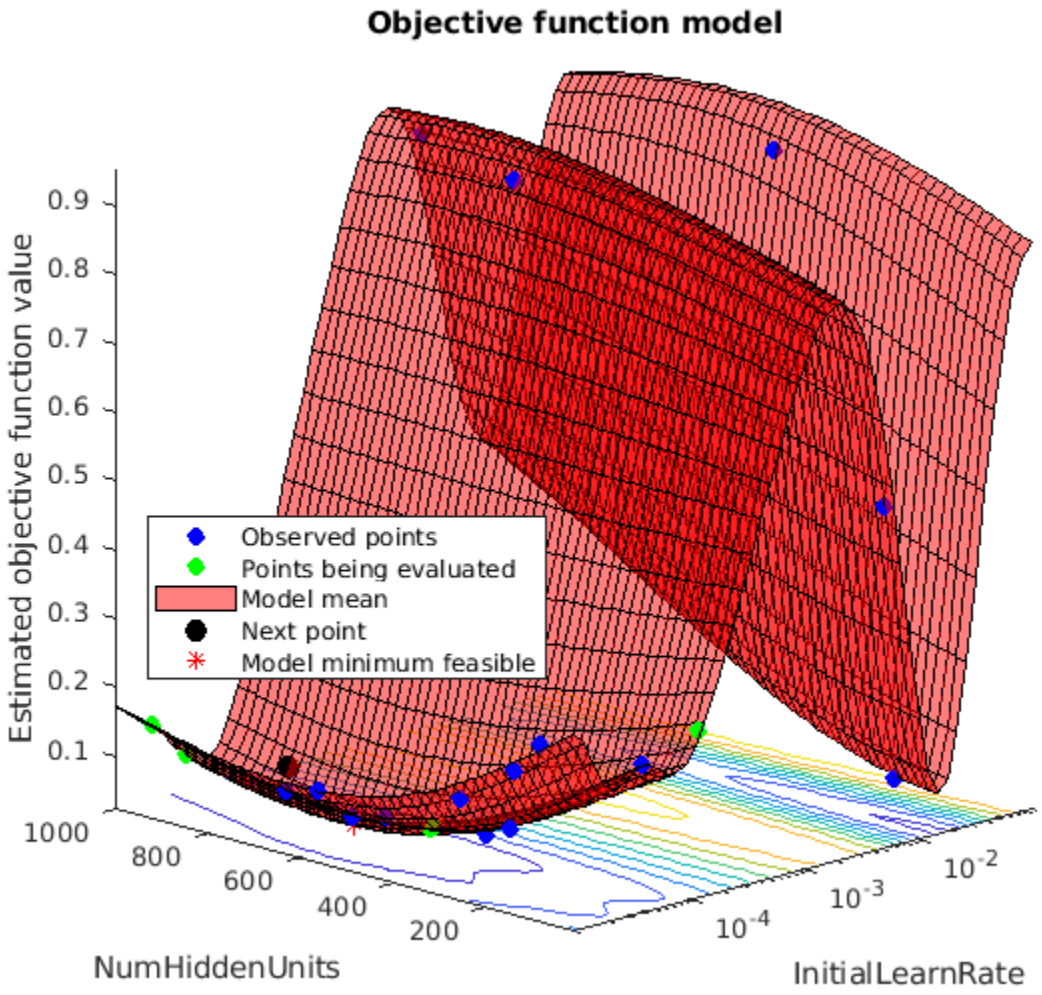
```
optVars = [
    optimizableVariable('InitialLearnRate',[1e-5, 1e-1],'Transform','log')
    optimizableVariable('NumHiddenUnits',[10, 1000],'Type','integer')
    ];
```

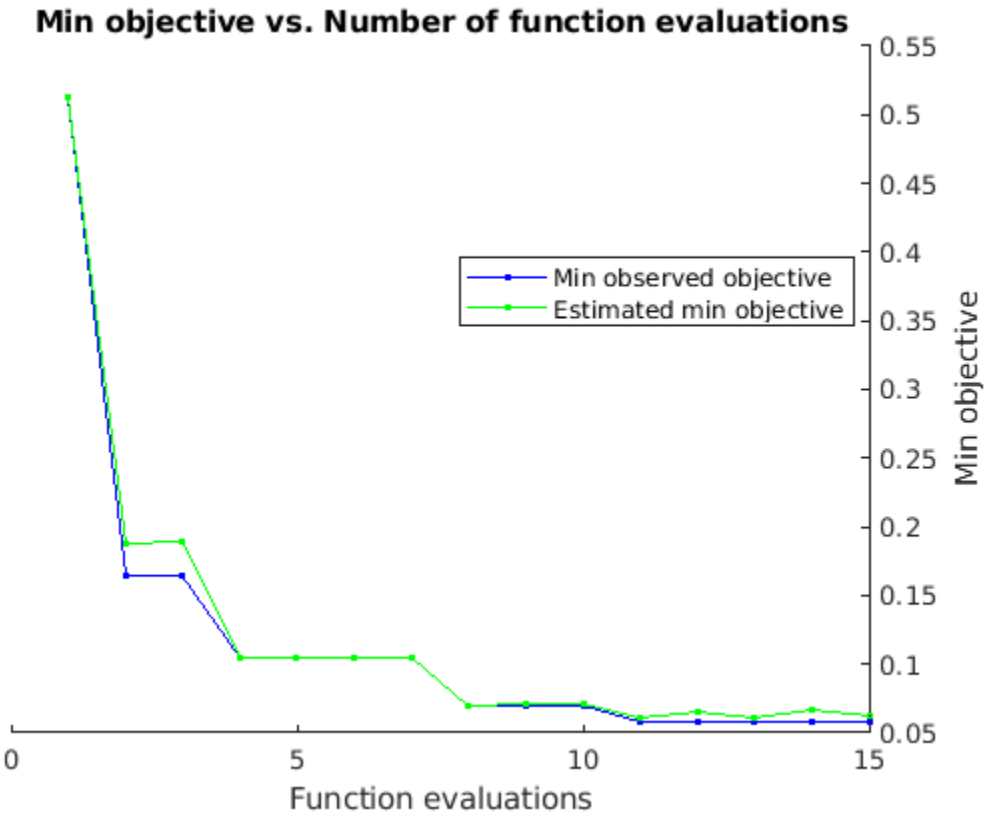
Bayesian optimization is computationally intensive and can take several hours to finish. For the purposes of this example, set `optimizeCondition` to `false` to load predetermined optimized

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
    load 0.02.mat
end
```

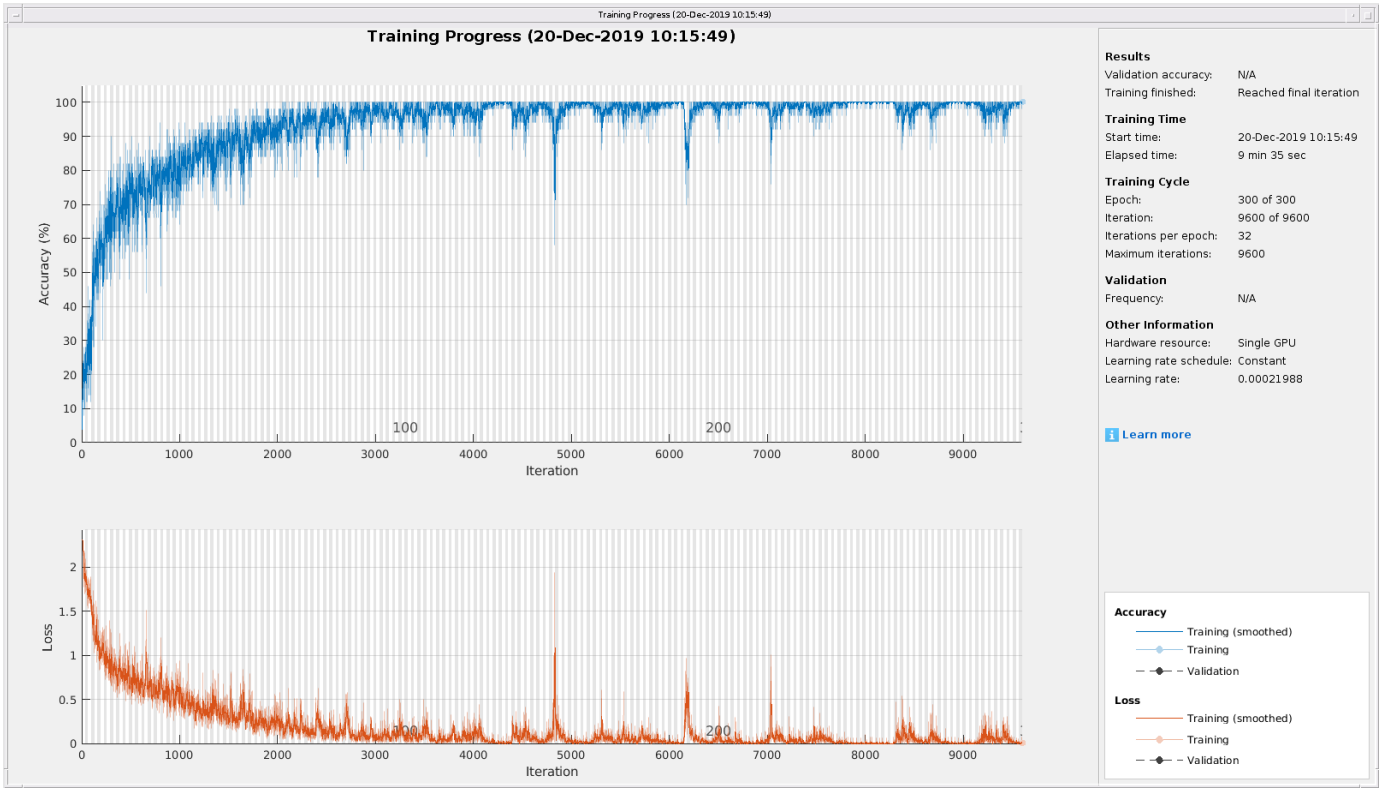
If you perform Bayesian optimization, figures similar to the following are generated to track the objective function values with the corresponding hyperparameter values and the number of iterations. You can increase the number of Bayesian optimization iterations to ensure that the global minimum of the 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 = 97.7500
```

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 1600
Processed 1000 files out of 1600
Processed 1500 files out of 1600
...done
```

```
XTrain = log10(XTrain + epsil);

XTest = helperspeechSpectrograms(adsTest,segmentDuration,frameDuration,hopDuration,numBands);
```

```
Computing speech spectrograms...
...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
```

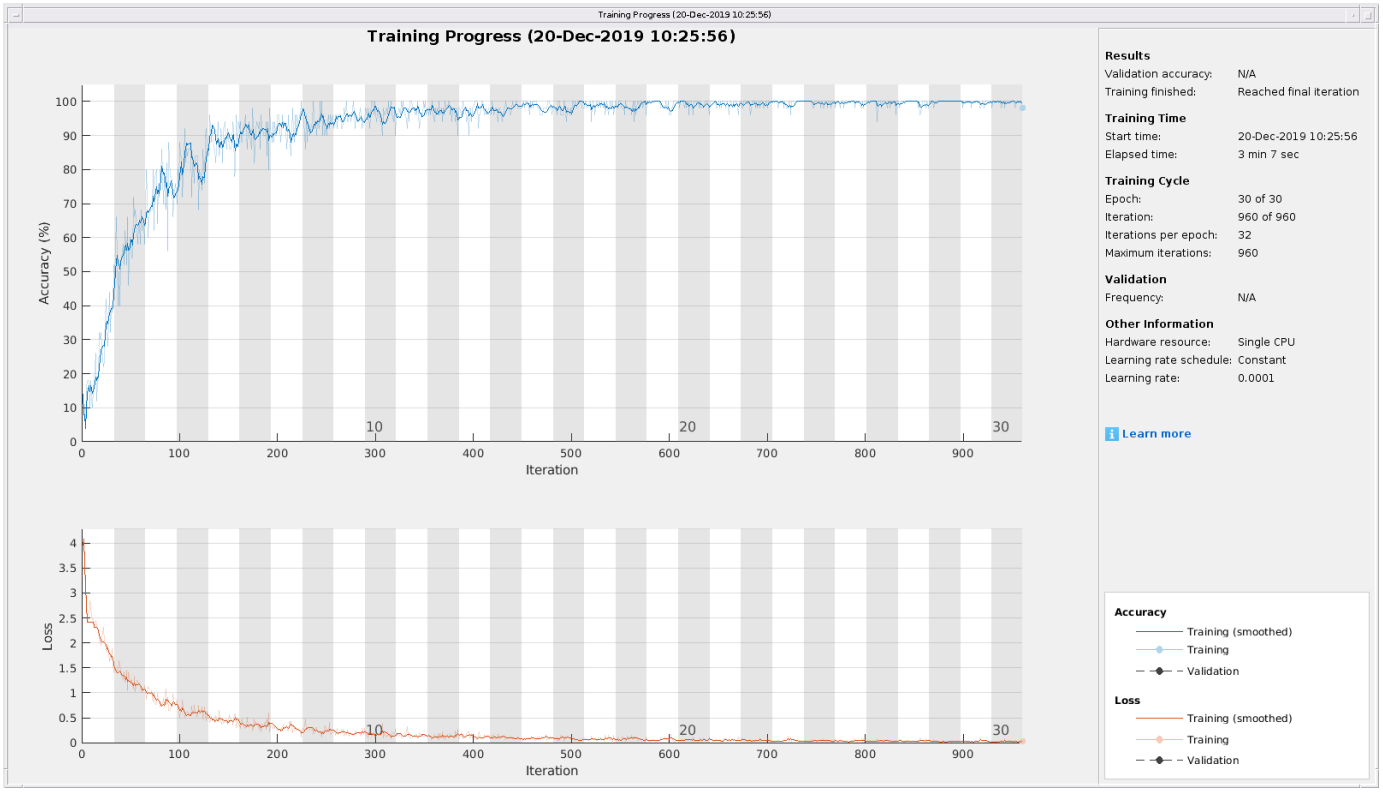
```
    reluLayer

    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);
```



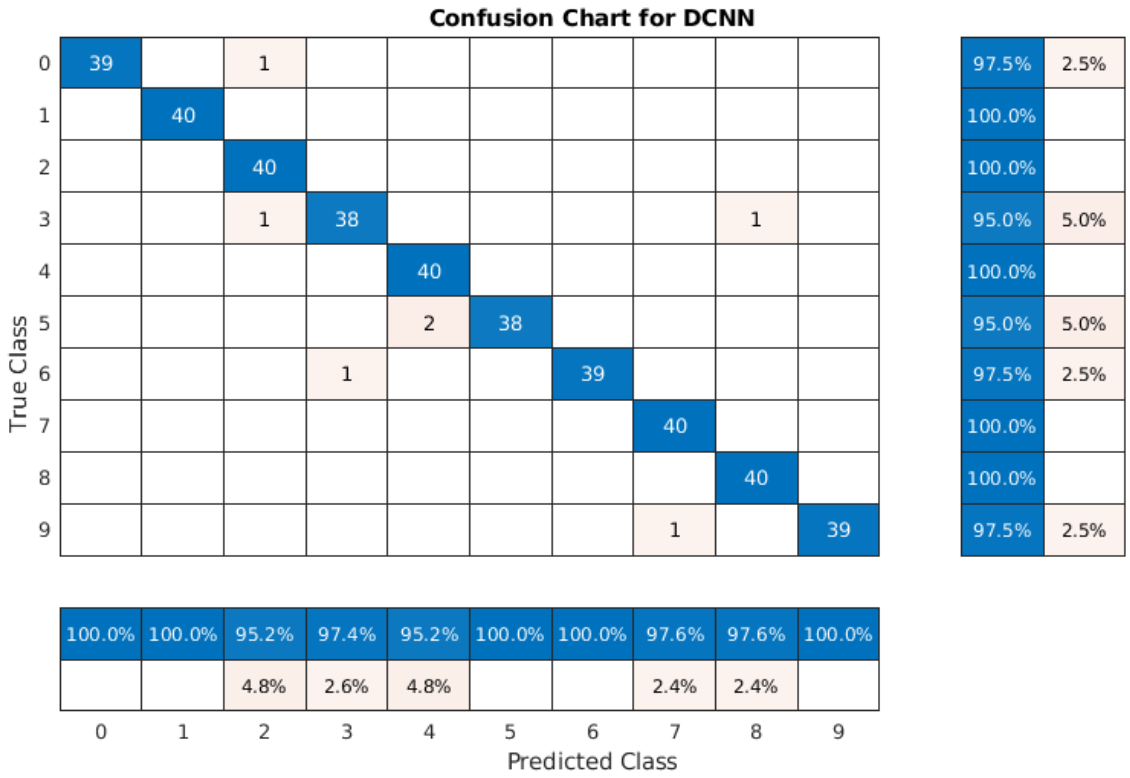
Use the trained network to predict the digit labels for the test set.

```
[Ypredicted,probs] = classify(trainedNet,XTest,'ExecutionEnvironment','CPU');
cnnAccuracy = sum(Ypredicted==YTest)/numel(YTest)*100
```

```
cnnAccuracy = 98.2500
```

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';
```



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, ...
        'WindowLength',frameLength, ...
        '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

`trainNetwork` | `trainingOptions`

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.

Problem Summary

Load audio files containing male and female speech sampled at 4 kHz. Listen to the audio files individually for reference.

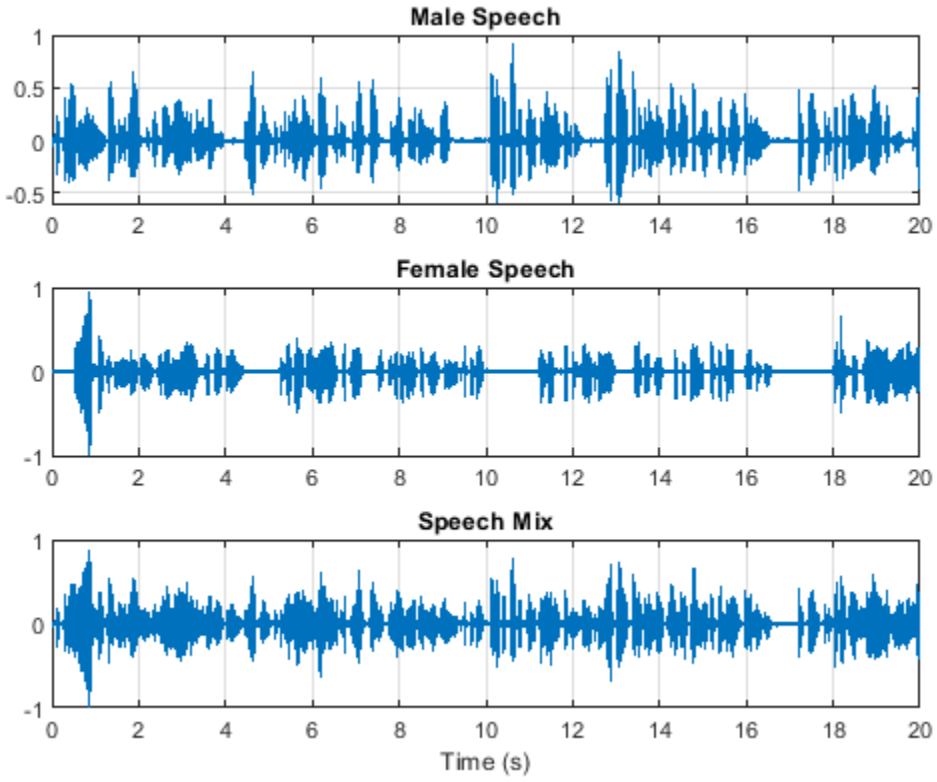
```
[mSpeech,Fs] = audioread("MaleSpeech-16-4-mono-20secs.wav");
sound(mSpeech,Fs)

[fSpeech] = audioread("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
```



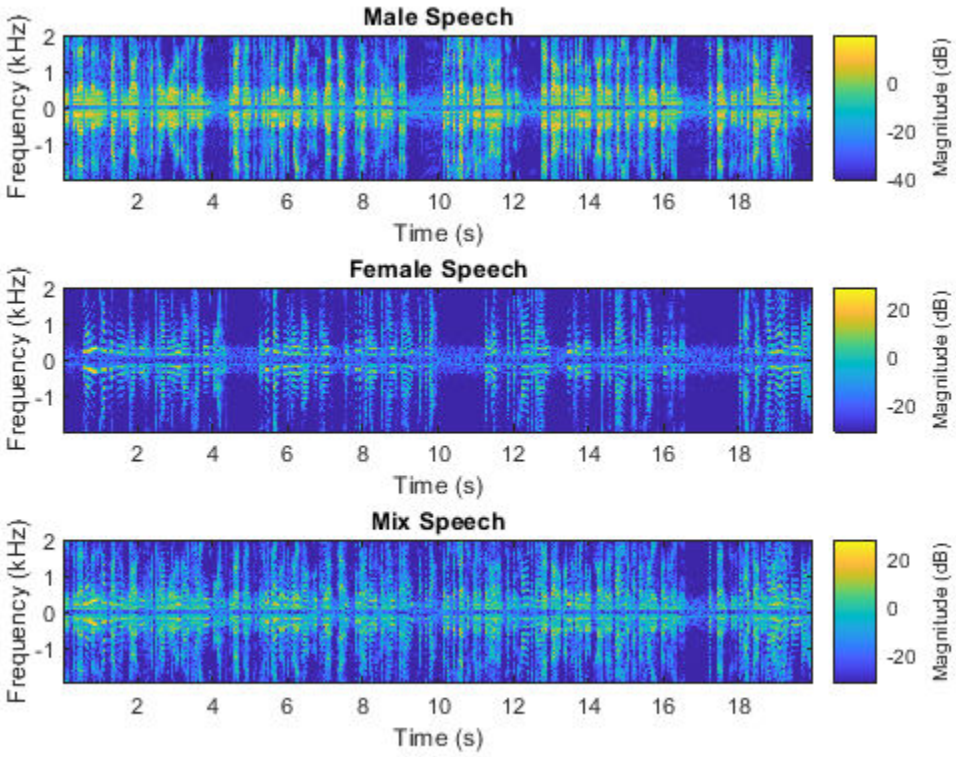
Listen to the mix audio.

```
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)
title("Male Speech")
subplot(3,1,2)
stft(fSpeech,Fs,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength)
title("Female Speech")
subplot(3,1,3)
stft(mix,Fs,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength)
title("Mix Speech")
```



Source Separation Using Ideal Time-Frequency Masks

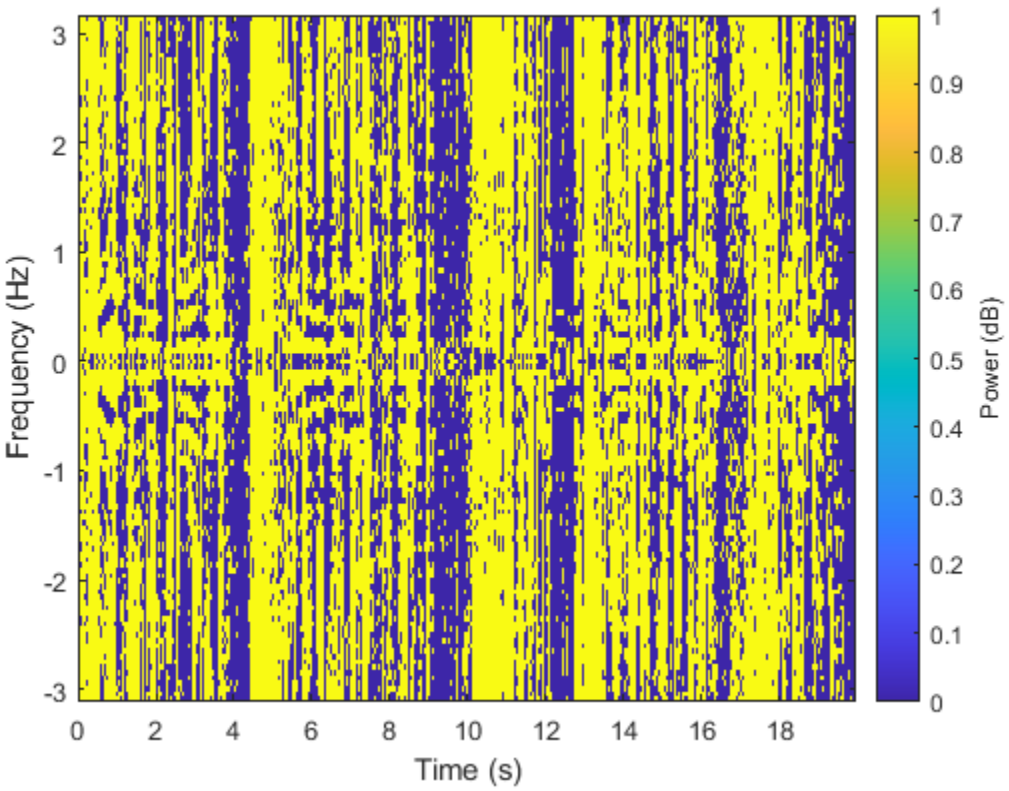
The application of a TF mask has been shown to be an effective method for separating desired audio signals from competing sounds. A TF mask is a matrix of the same size as the underlying STFT. The mask is multiplied element-by-element with the underlying STFT to isolate the desired source. The TF mask 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);
P_F        = stft(fSpeech,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength);
[P_mix,F]  = stft(mix,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength);
binaryMask = abs(P_M) >= abs(P_F);

figure(3)
plotMask(binaryMask,WindowLength - OverlapLength,F,Fs)
```



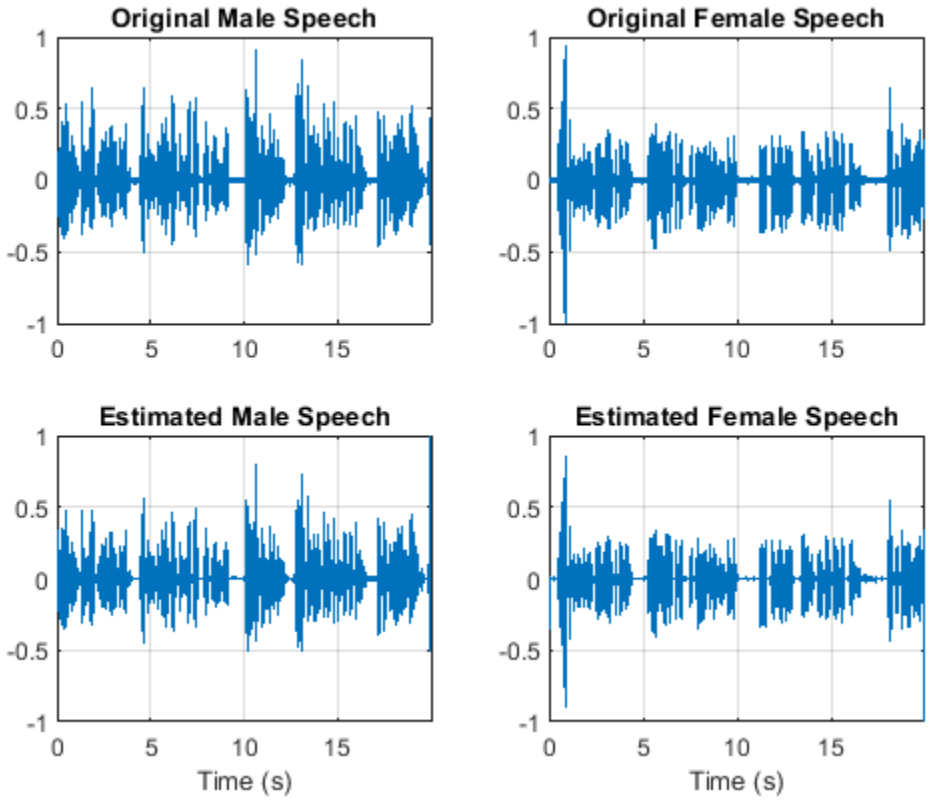
Estimate the male speech STFT by multiplying the mix STFT by the male speaker's binary mask. Estimate the female speech STFT by multiplying the mix STFT by the inverse of the male speaker's binary mask.

```
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)
fSpeech_Hard = istft(P_F_Hard,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength);

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].

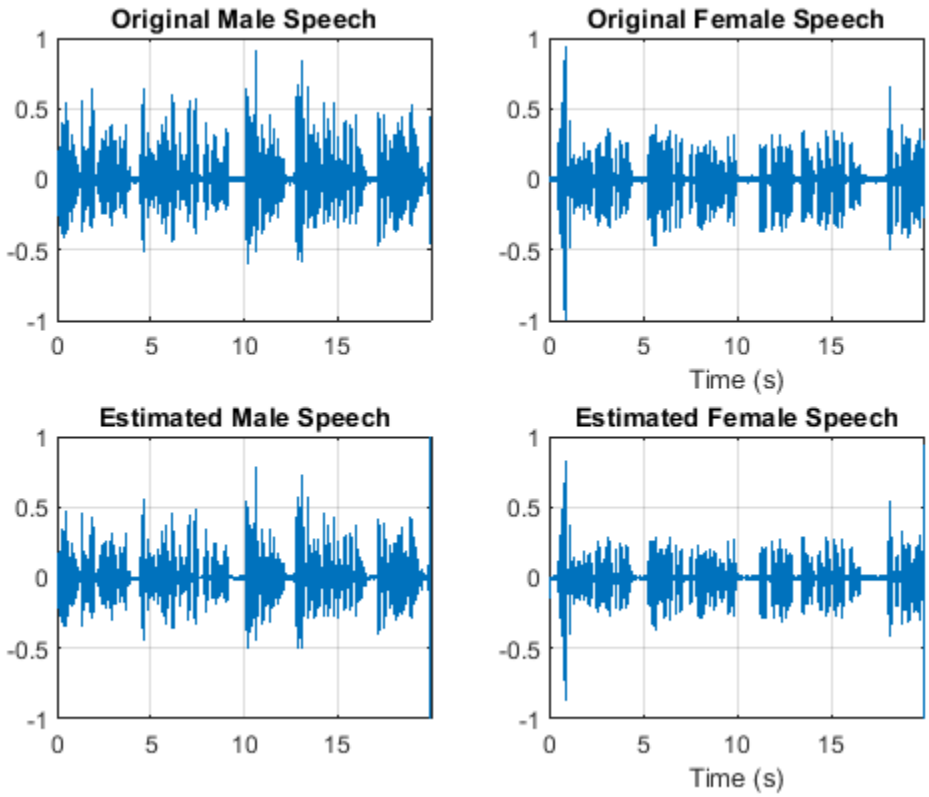
Compute the soft mask for the male speaker. Estimate the STFT of the male speaker by multiplying the mix STFT by the male speaker's soft mask. Estimate the STFT of the female speaker by multiplying the mix STFT by the female speaker's soft mask.

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);
fSpeech_Soft = istft(P_F_Soft,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLength);
```

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
```



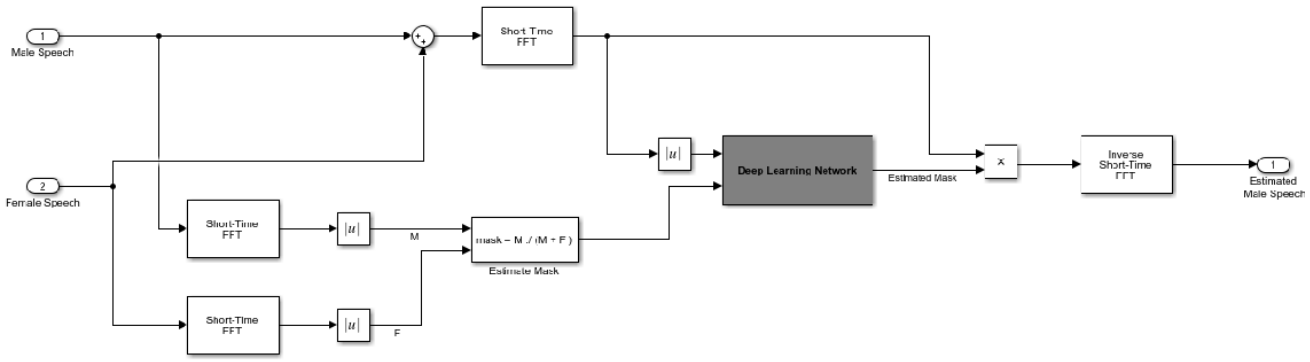

```
sound(mSpeech_Soft,Fs)
```

```
sound(fSpeech_Soft,Fs)
```

Mask Estimation Using Deep Learning

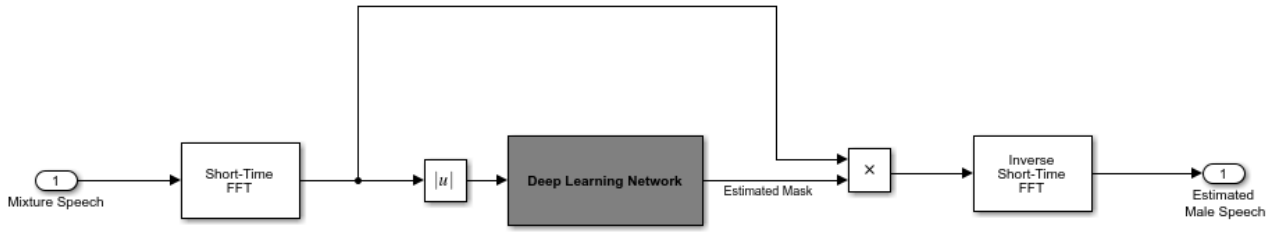
The goal of the deep learning network in this example is to estimate the ideal soft mask described above. The network estimates the mask corresponding to the male speaker. The female speaker mask is derived directly from the male mask.

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(maleTrainingAudioFile);
femaleSpeechTrain = audioread(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(maleValidationAudioFile);
femaleSpeechValidate = audioread(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);
P_M    = abs(stft(maleSpeechTrain,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLeng
P_F    = abs(stft(femaleSpeechTrain,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLe
```

Since the STFT input is real, the STFTs are conjugate symmetric. Reduce the STFTs to their unique samples.

```
N      = 1 + FFTLength/2;
P_mix0 = P_mix0(N-1:end,:);
P_M    = P_M(N-1:end,:);
P_F    = P_F(N-1:end,:);
```

Take the log of the mix STFT. Normalize the values by their mean and standard deviation.

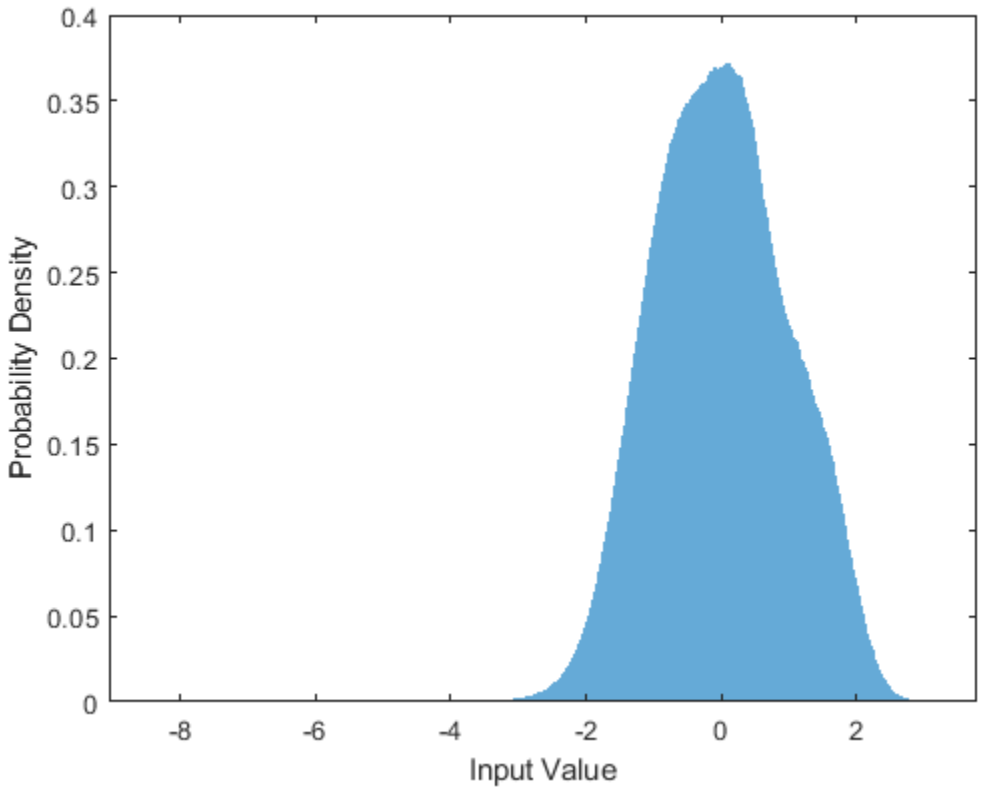
```
P_mix = log(abs(P_mix0) + eps);
MP    = mean(P_mix(:));
SP    = std(P_mix(:));
P_mix = (P_mix - MP) / SP;
```

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);
P_Val_M    = abs(stft(maleSpeechValidate,'Window',win,'OverlapLength',OverlapLength,'FFTLength',
P_Val_F    = abs(stft(femaleSpeechValidate,'Window',win,'OverlapLength',OverlapLength,'FFTLength

P_Val_mix0 = P_Val_mix0(N-1:end,:);
P_Val_M    = P_Val_M(N-1:end,:);
P_Val_F    = P_Val_F(N-1:end,:);

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.

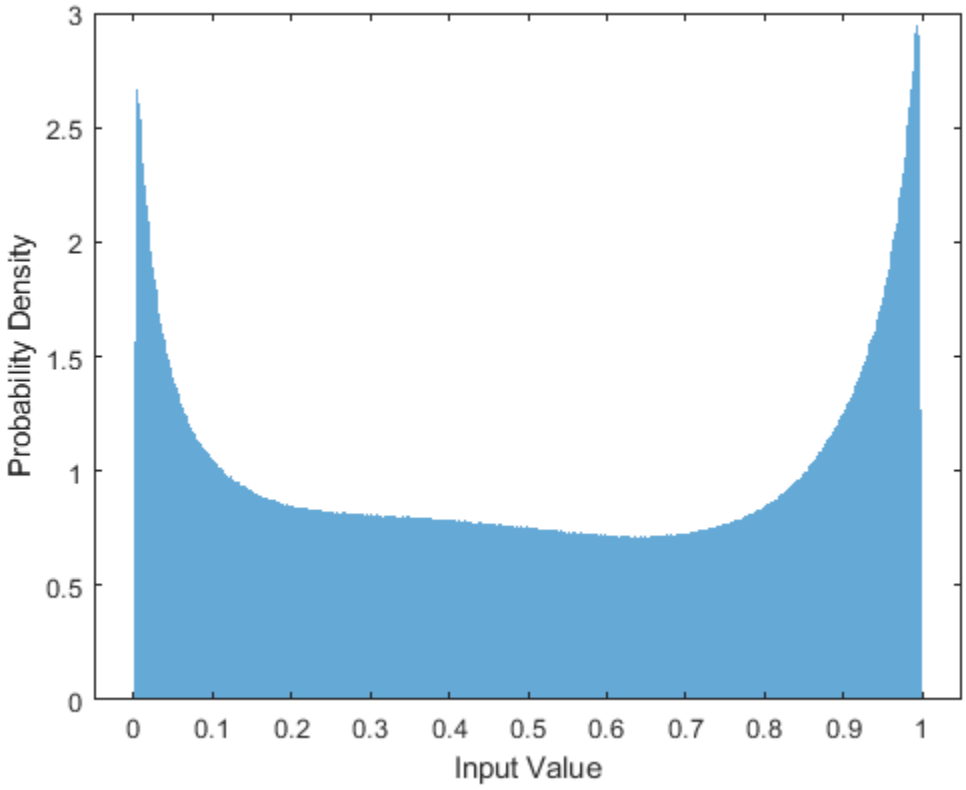
```
maskTrain = P_M ./ (P_M + P_F + eps);
```

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

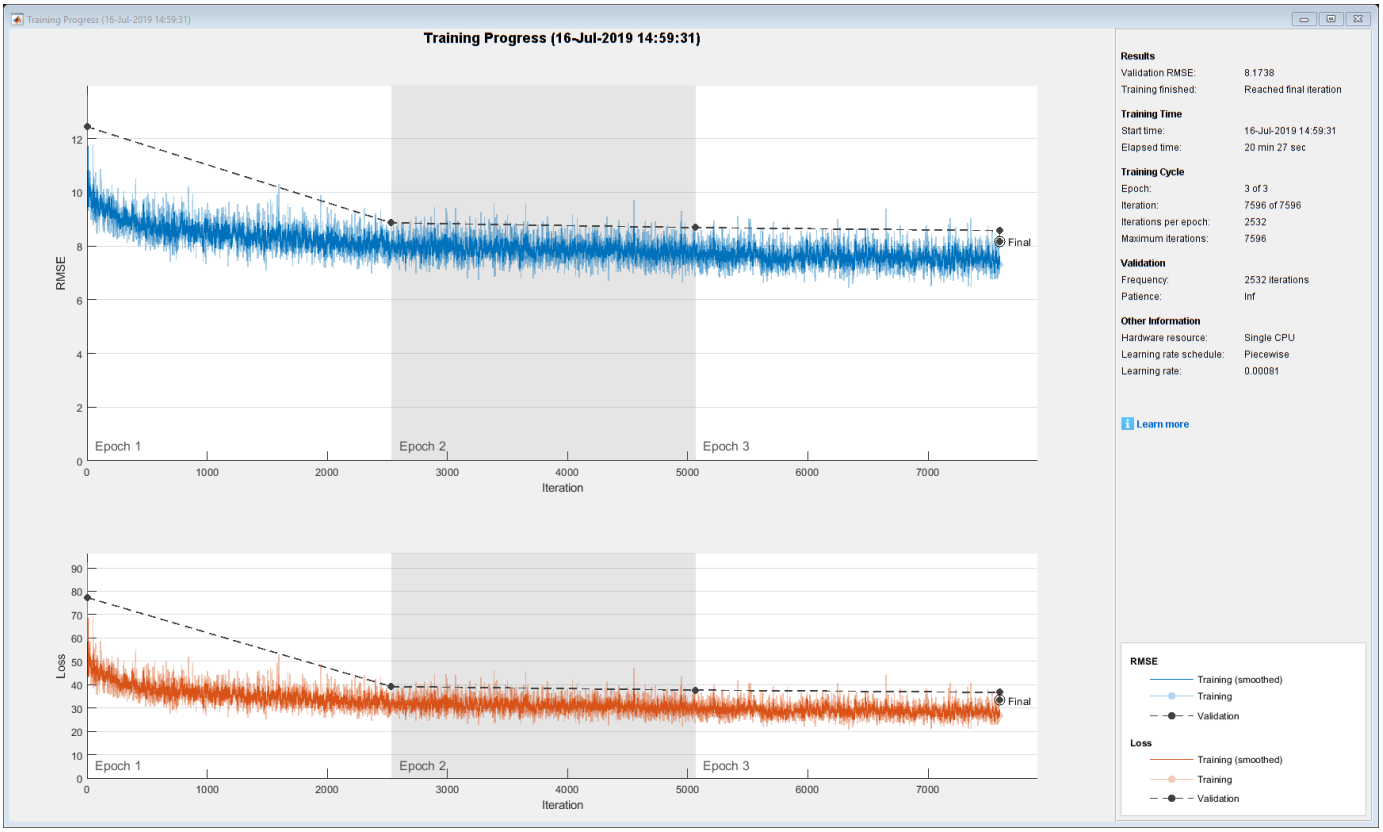
time a certain number of epochs (1) has passed. Set `ValidationData` to the validation predictors and targets. Set `ValidationFrequency` such that the validation mean square error is computed once per epoch. This example uses the adaptive moment estimation (ADAM) solver.

```
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
```



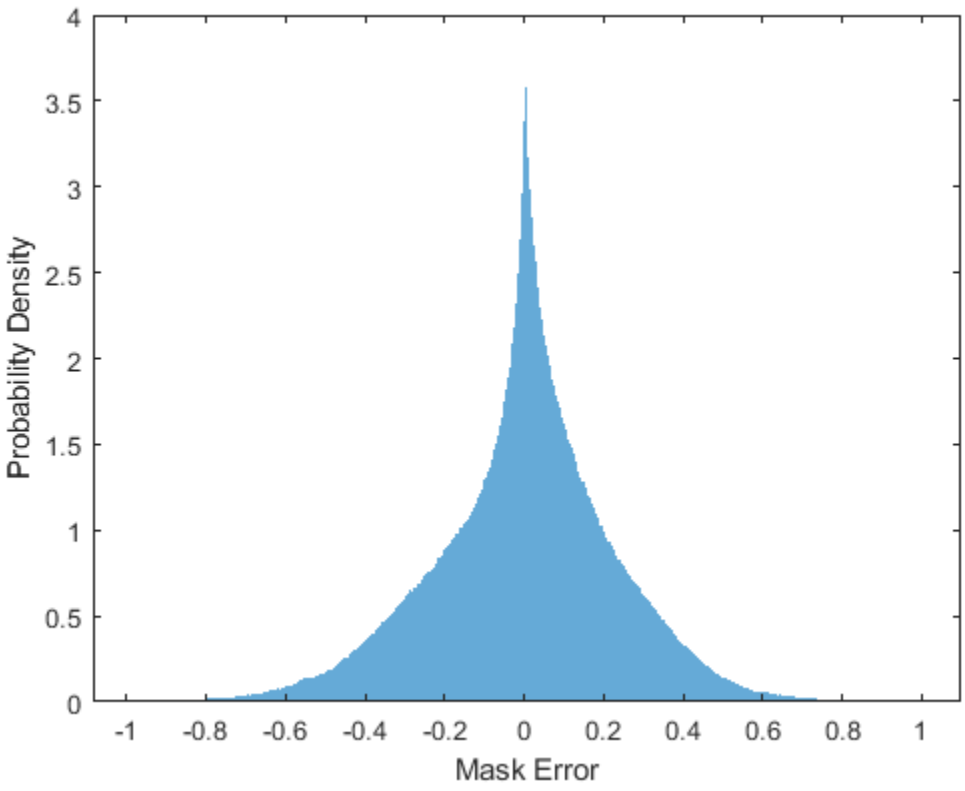
Pass the validation predictors to the network. The output is the estimated mask. Reshape the estimated mask.

```
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;
```

Convert the one-sided STFT to a centered STFT.

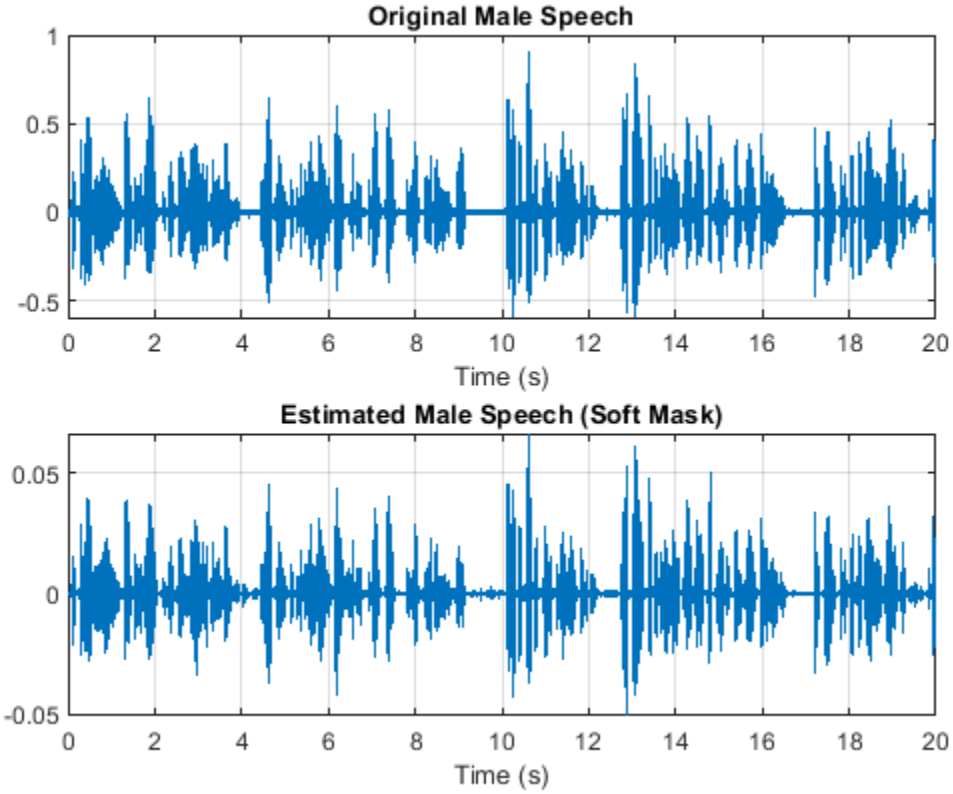
```
P_Male = [conj(P_Male(end-1:-1:2,:)) ; P_Male];
```

Use the ISTFT to get the estimated male audio signal. Scale the audio.

```
maleSpeech_est_soft = istft(P_Male, 'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLe
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
```



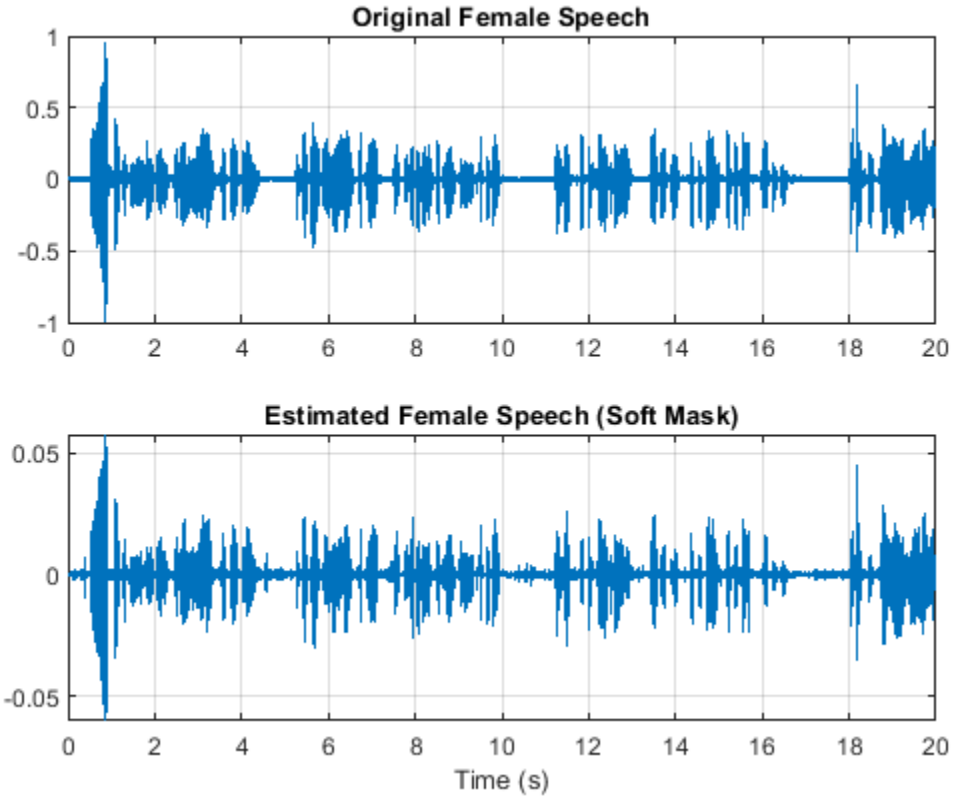

```
sound(maleSpeech_est_soft(range),Fs)
```

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;

P_Female = [conj(P_Female(end-1:-1:2,:)) ; P_Female];

femaleSpeech_est_soft = istft(P_Female,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FF
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.

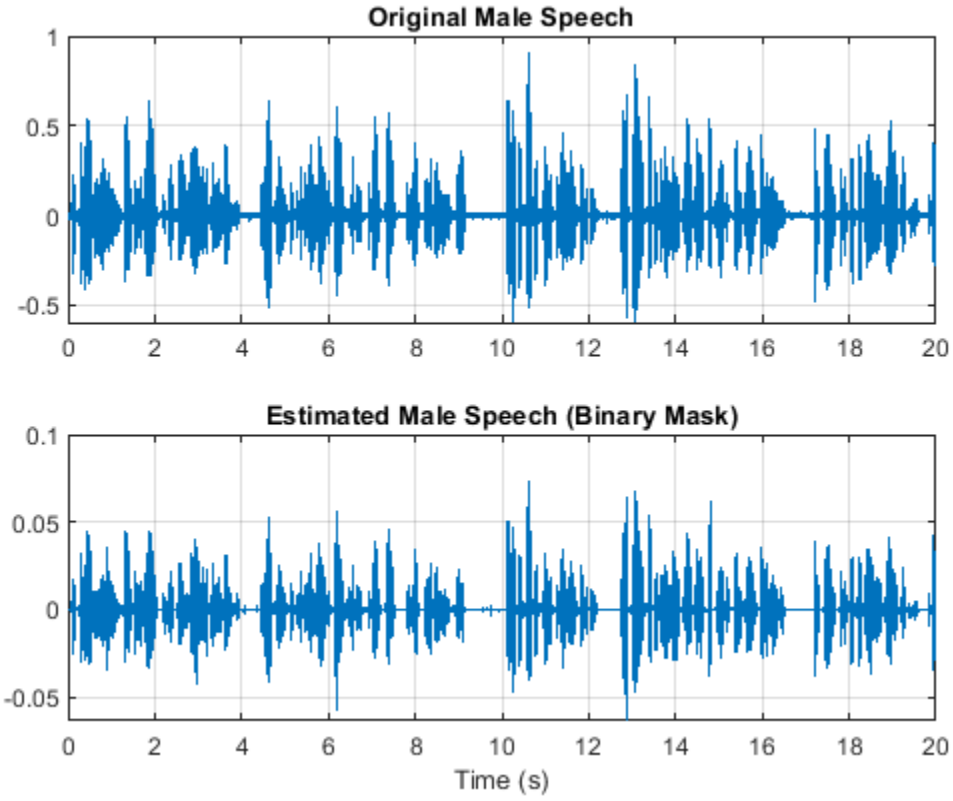
```
HardMaleMask   = SoftMaleMask >= 0.5;
HardFemaleMask = SoftMaleMask < 0.5;
```

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;

P_Male = [conj(P_Male(end-1:-1:2,:)) ; P_Male];

maleSpeech_est_hard = istft(P_Male,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FFTLeng
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
```



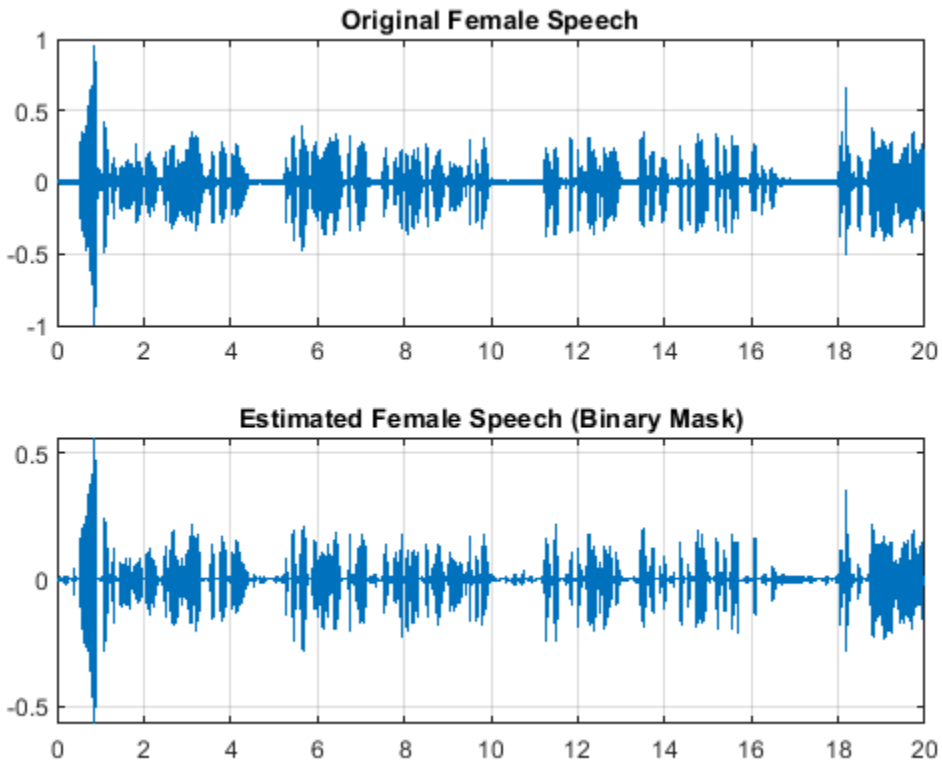

```
sound(maleSpeech_est_hard(range),Fs)
```

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;

P_Female = [conj(P_Female(end-1:-1:2,:)) ; P_Female];

femaleSpeech_est_hard = istft(P_Female,'Window',win,'OverlapLength',OverlapLength,'FFTLength',FF
femaleSpeech_est_hard = femaleSpeech_est_hard / max(femaleSpeech_est_hard);
```

Visualize the estimated and original female speech signals. Listen to the estimated female speech.

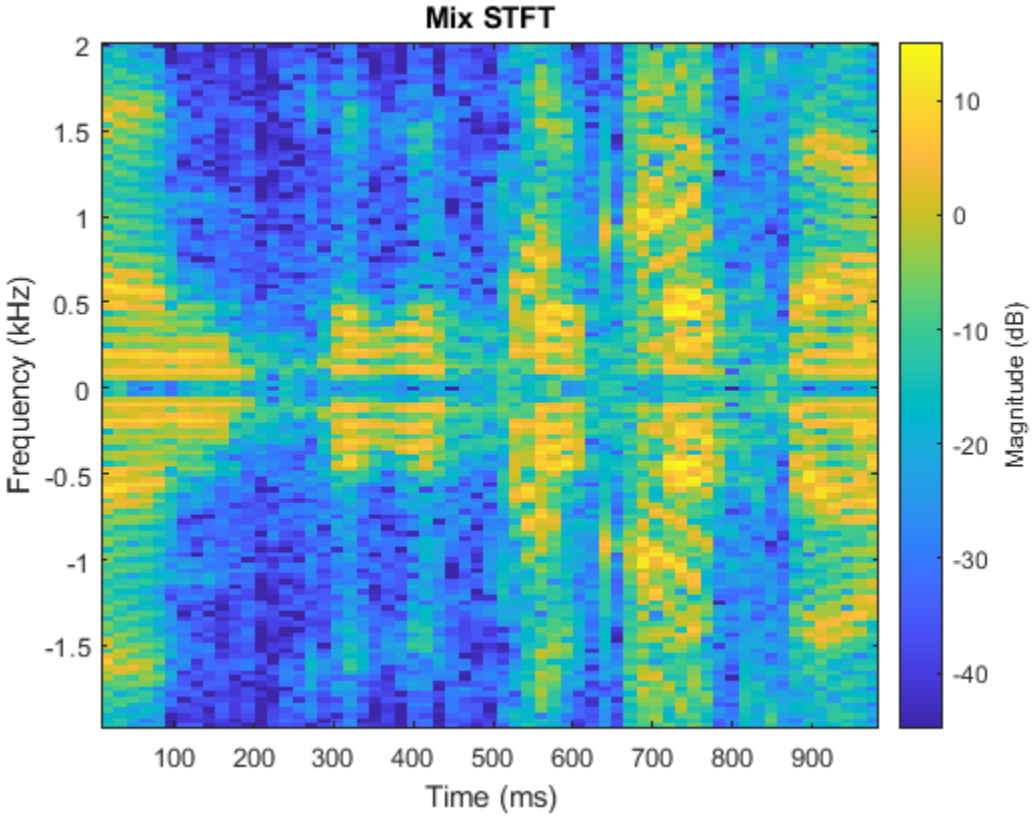
```
range = (numel(win):numel(maleSpeech_est_soft)-numel(win));
t = range * (1/Fs);

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
```



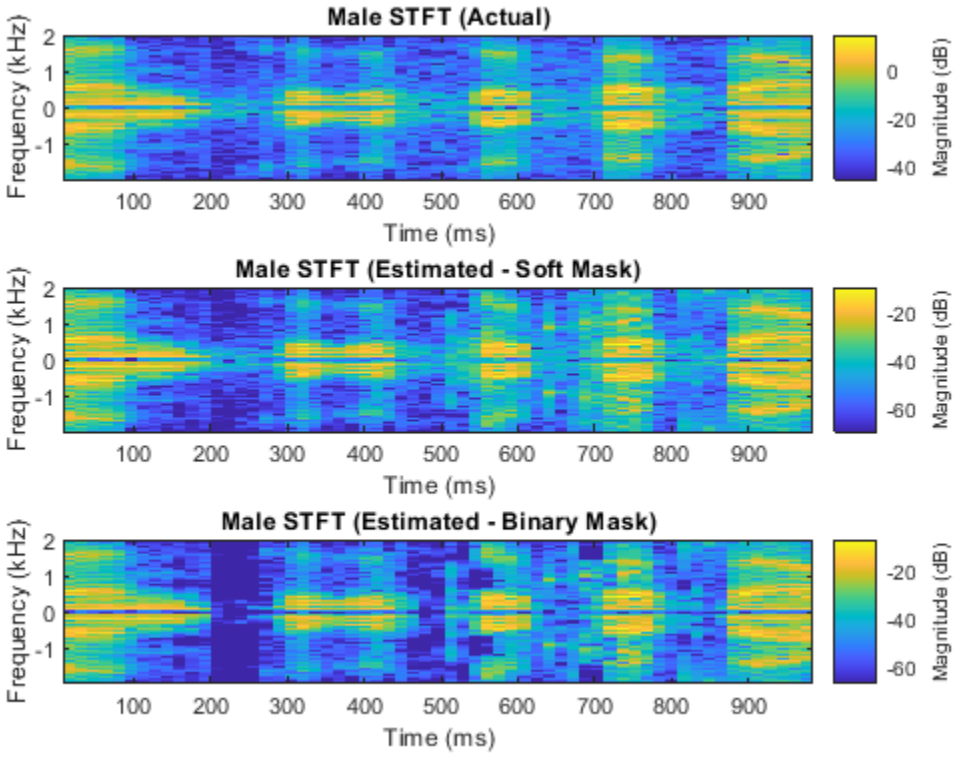

```
sound(femaleSpeech_est_hard(range),Fs)
```

Compare STFTs of a one-second segment for mix, original female and male, and estimated female and male, respectively.

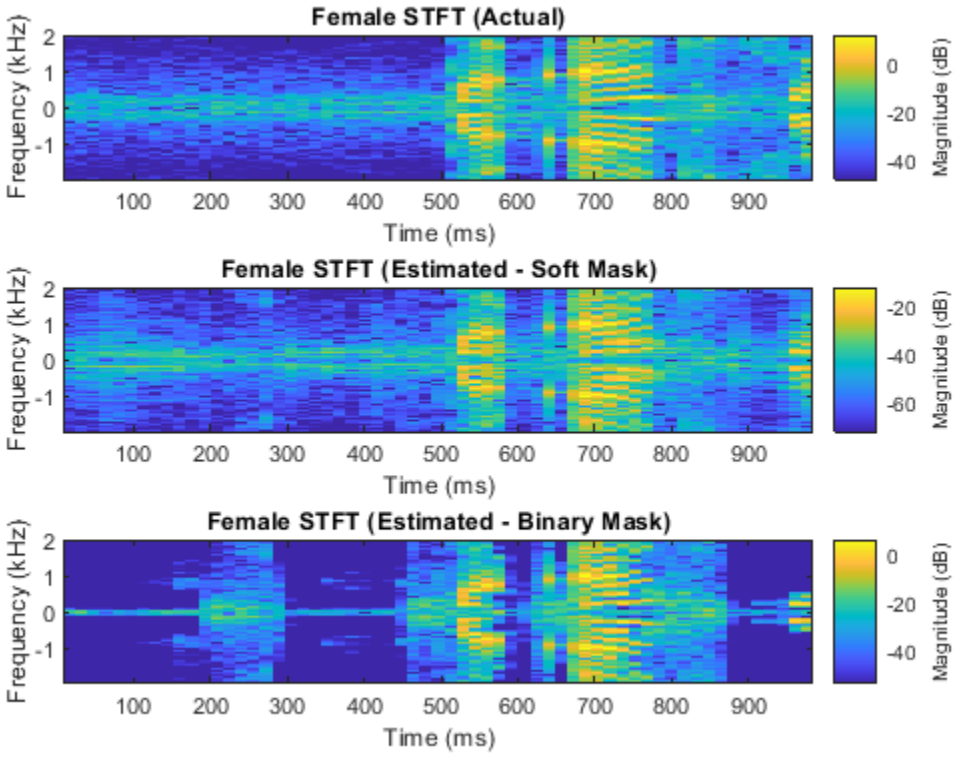
```
range = 7e4:7.4e4;

figure(13)
stft(mixValidate(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Mix STFT")
```




```
figure(14)
subplot(3,1,1)
stft(maleSpeechValidate(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Male STFT (Actual)")
subplot(3,1,2)
stft(maleSpeech_est_soft(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Male STFT (Estimated - Soft Mask)")
subplot(3,1,3)
stft(maleSpeech_est_hard(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Male STFT (Estimated - Binary Mask)");
```




```
figure(15)
subplot(3,1,1)
stft(femaleSpeechValidate(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Female STFT (Actual)")
subplot(3,1,2)
stft(femaleSpeech_est_soft(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
title("Female STFT (Estimated - Soft Mask)")
subplot(3,1,3)
stft(femaleSpeech_est_hard(range),Fs,'Window',win,'OverlapLength',64,'FFTLength',FFTLength)
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

`trainNetwork` | `trainingOptions`

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.

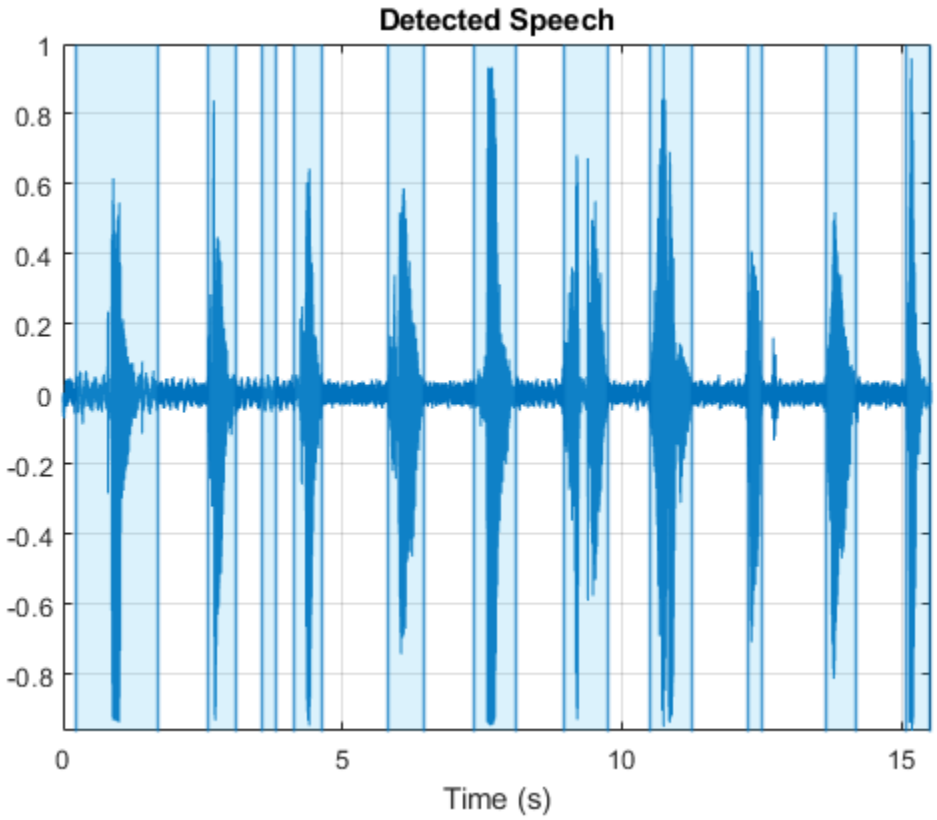
This example trains a voice activity detection bidirectional LSTM network with feature sequences of spectral characteristics and a harmonic ratio metric.

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` 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)
```

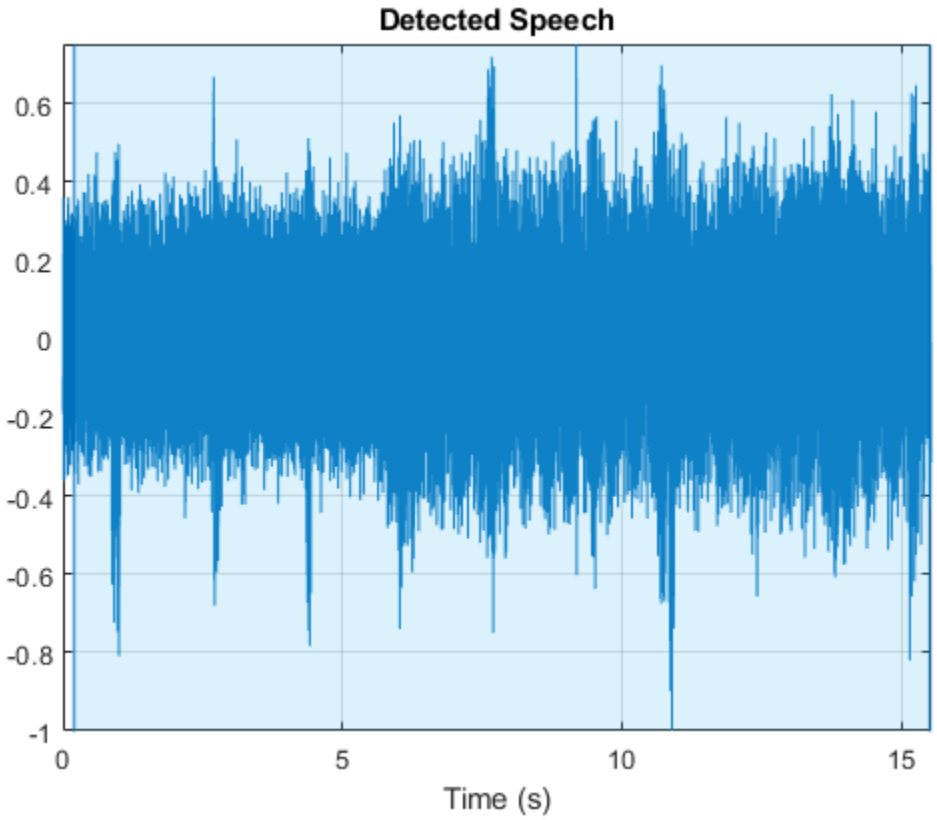


Corrupt the audio signal with washing machine noise at a -20 dB SNR. Listen to the corrupted audio.

```
[noise,fileFs] = audioread('WashingMachine-16-8-mono-200secs.wav');
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)
```



Load a pretrained network and a configured `audioFeatureExtractor` object. The network was trained to detect speech in a low SNR environments given features output from the `audioFeatureExtractor` object.

```
load('Audio_VoiceActivityDetectionExample.mat','speechDetectNet','afe')
```

```
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.

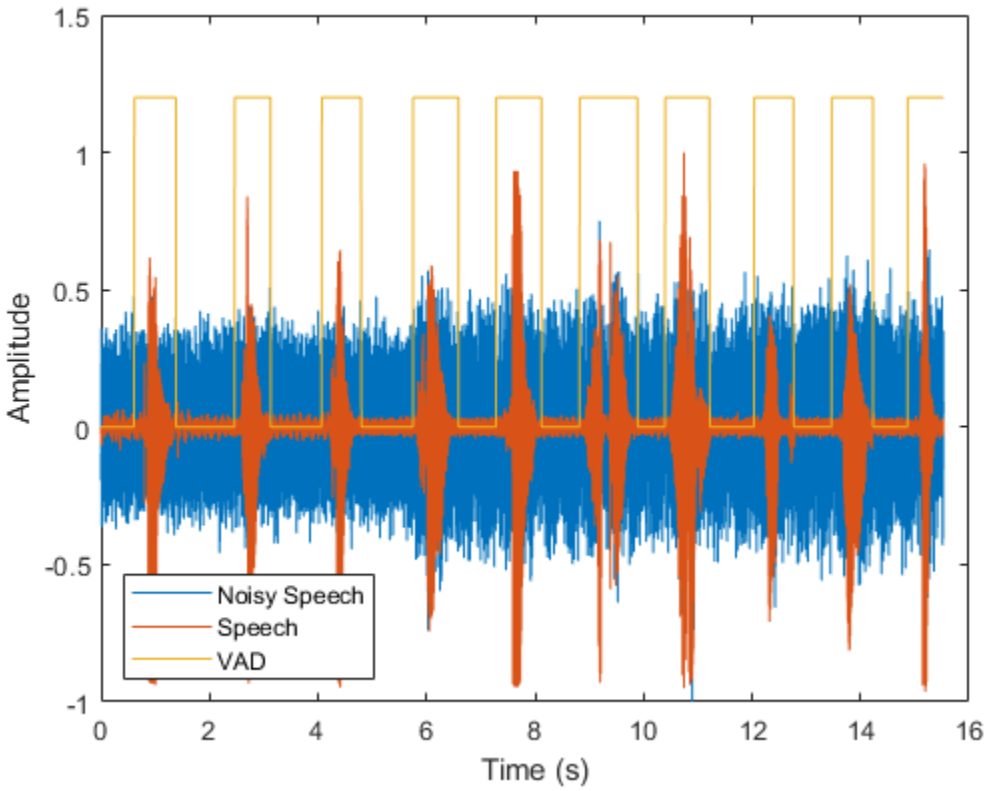
```
features = extract(afe,noisySpeech);
features = (features - mean(features,1)) ./ std(features,[],1);
features = features';
```

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.

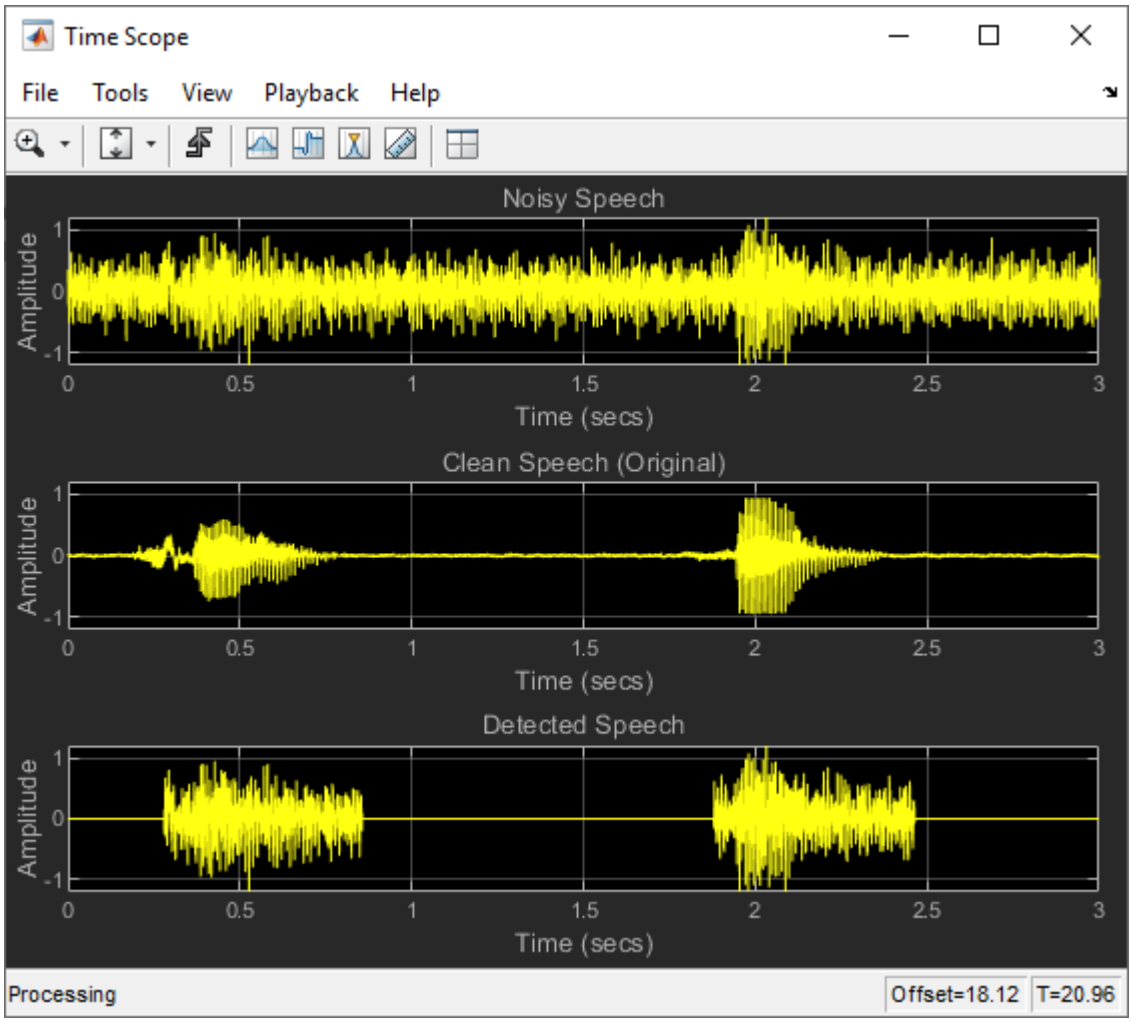
```
audiowrite('Speech.wav',speech,fs)
audiowrite('Noise.wav',noise,fs)
```

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 = ;

sequenceLength = ;
sequenceHop = ;
SNR = ;
noiseGain = 10^(-SNR/20) * norm(speech) / norm(noise);

signalToListenTo = ;
```

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 12-0 .

```
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

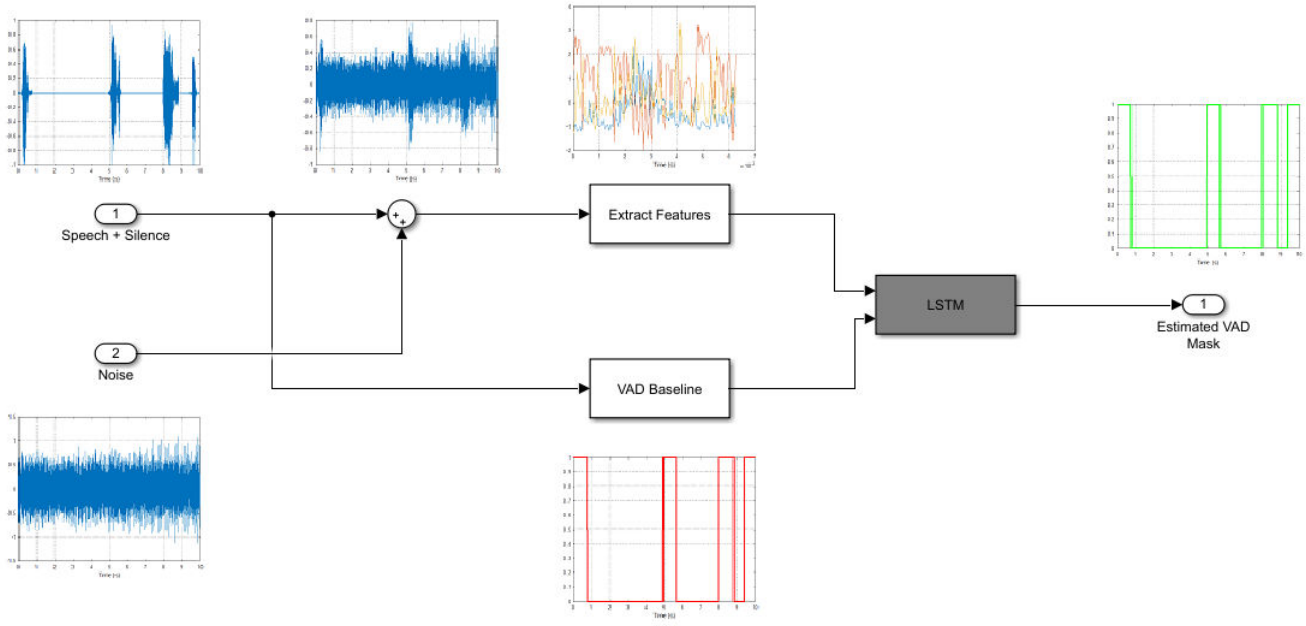
Training:

1. Create an `audioDatastore` 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.

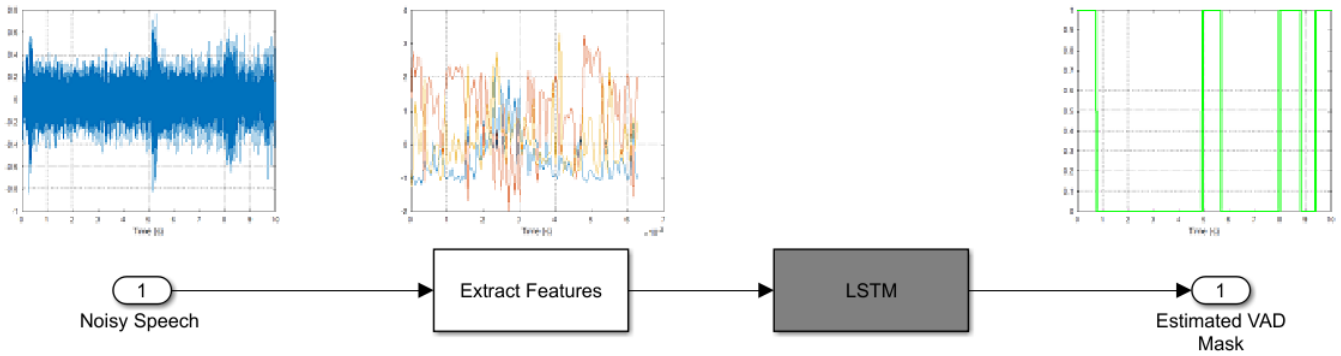
Prediction:

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 12-0 .

```
url = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz'

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading Google speech commands data set (1.9 GB)...')
    untar(url,datasetFolder)
end
```

Create an `audioDatastore` that points to the data set.

```
ads = audioDatastore(datasetFolder,"Includesubfolders",true);
```

The data set contains background noise files that are not used in this example. Use `subset` to create a new datastore that does not have the background noise files.

```
indices = cellfun(@(c)~contains(c,"_background_noise_"),ads.Files);
ads = subset(ads,indices);
```

Split Data into Training and Validation

The data set folder contains text files that list which audio files should be in the validation set and which audio files should be in the test set. These predefined validation and test sets do not contain utterances of the same word by the same person, so it is better to use these predefined sets than to select a random subset of the whole data set.

Because this example trains a single network, it only uses the validation set and not the test set to evaluate the trained model. If you train many networks and choose the network with the highest validation accuracy as your final network, then you can use the test set to evaluate the final network.

Read the list of validation files.

```
c = importdata(fullfile(datasetFolder,'validation_list.txt'));
filesValidation = string(c);
```

Read the list of test files.

```
c = importdata(fullfile(datasetFolder,'testing_list.txt'));
filesTest = string(c);
```

Determine which files in the datastore should go to validation set and which should go to training set. Use `subset` to create two new datastores.

```
files = ads.Files;
sf = split(files,filesep);
isValidation = ismember(sf(:,end-1) + "/" + sf(:,end),filesValidation);
isTest = ismember(sf(:,end-1) + "/" + sf(:,end),filesTest);

adsValidation = subset(ads,isValidation);
adsTrain = subset(ads,~isValidation & ~isTest);
```

Create Speech-Plus-Silence Training Signal

Read the contents of an audio file using `read`. Get the sample rate from the `adsInfo` struct.

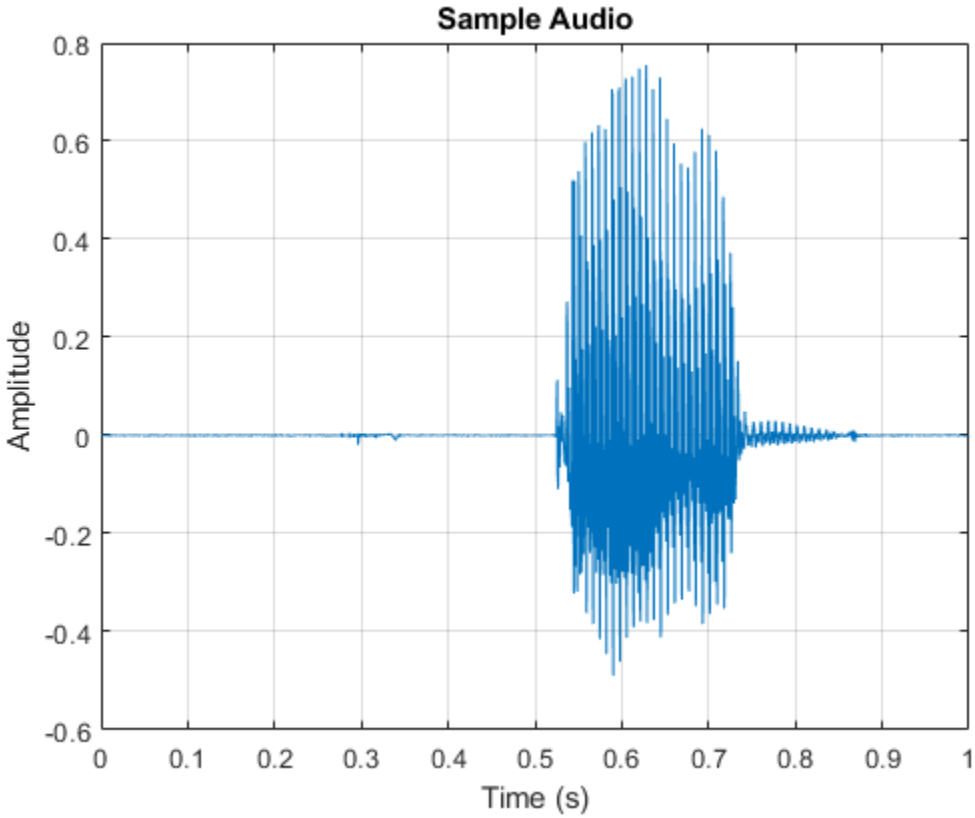
```
[data,adsInfo] = read(adsTrain);
Fs = adsInfo.SampleRate;
```

Listen to the audio signal using the sound command.

```
sound(data,Fs)
```

Plot the audio signal.

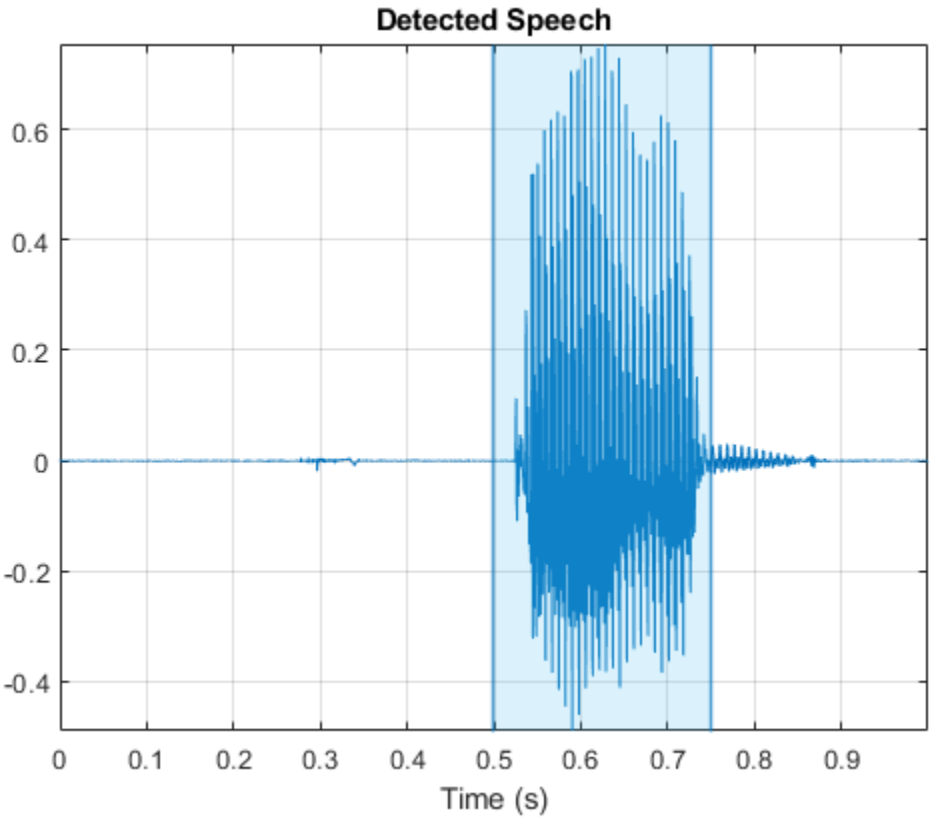
```
timeVector = (1/Fs) * (0:numel(data)-1);
plot(timeVector,data)
ylabel("Amplitude")
xlabel("Time (s)")
title("Sample Audio")
grid on
```



The signal has non-speech portions (silence, background noise, etc) that do not contain useful speech information. This example removes silence using the `detectSpeech` 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.

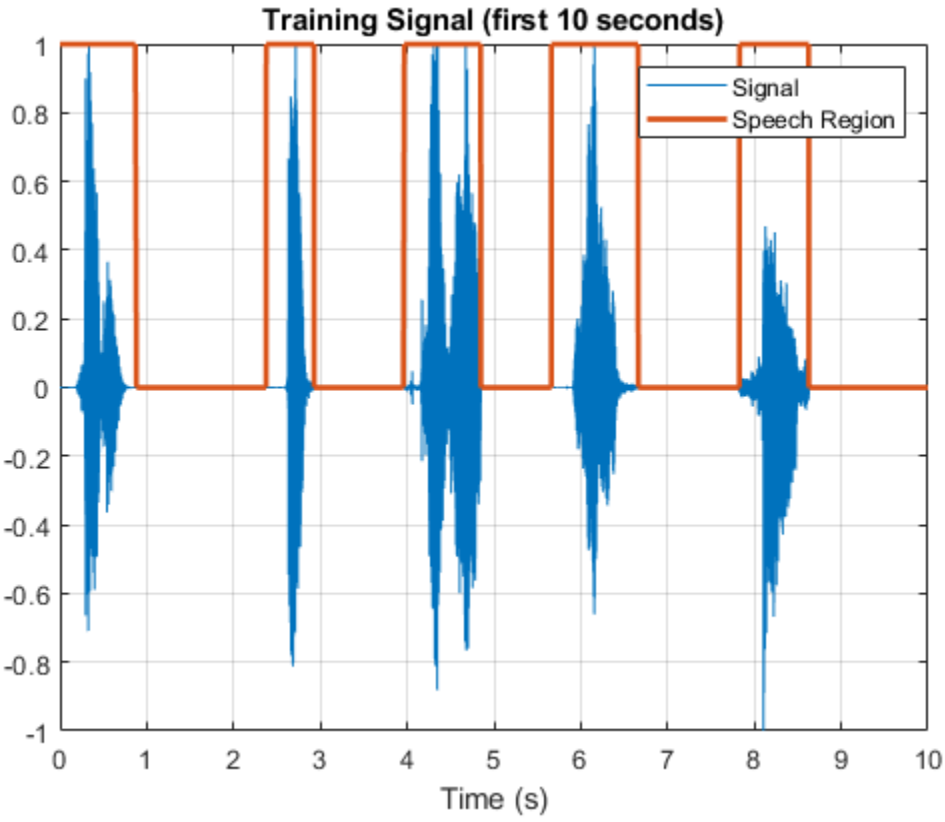
```
maxSilenceSegment = 2;
```

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.

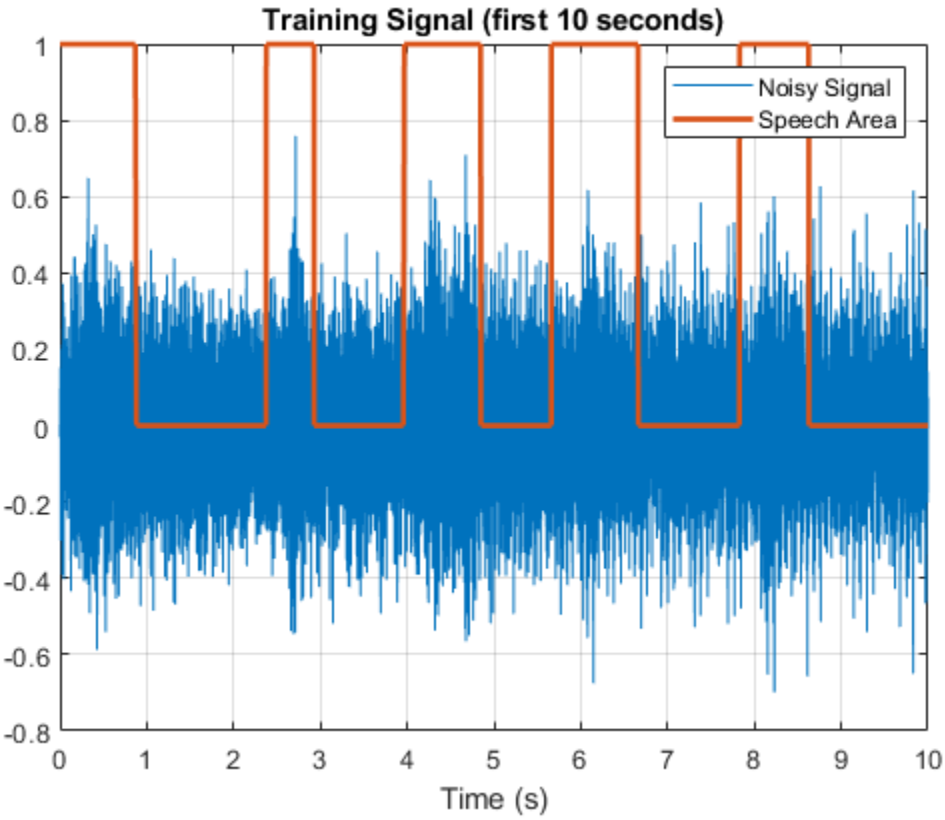
```
noise = audioread("WashingMachine-16-8-mono-1000secs.wav");
noise = resample(noise,2,1);
```

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.

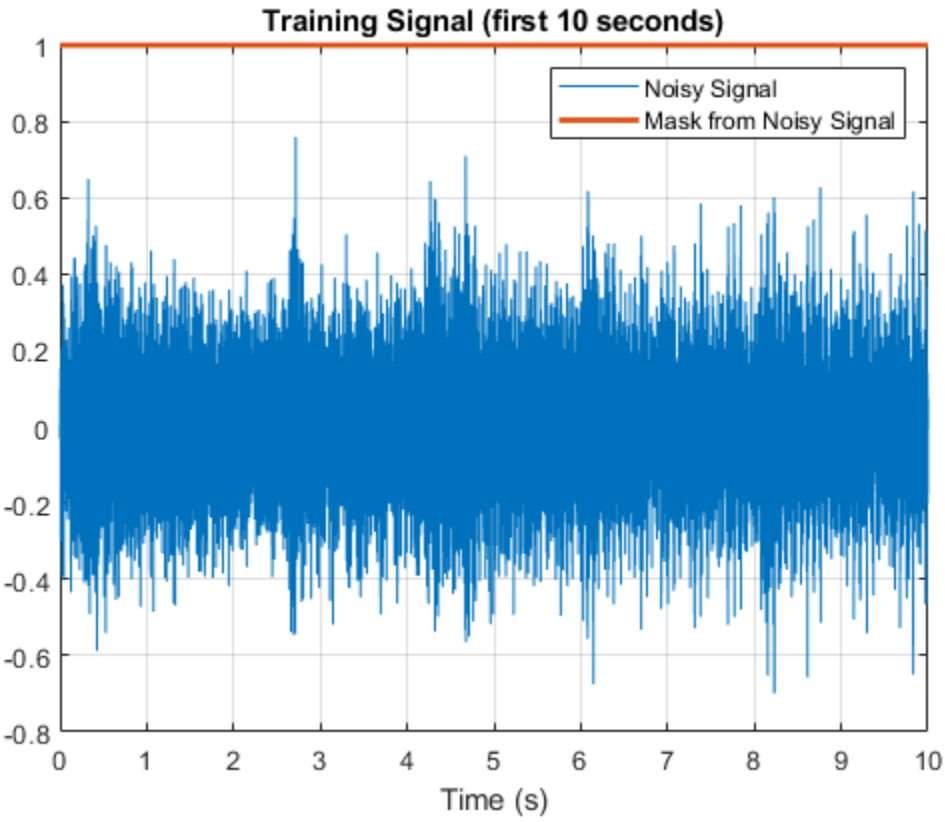
```
sound(audioTrainingNoisy(range),Fs)
```

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.wav");
noise = resample(noise,2,1);
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`
2. `spectralCrest`

3 spectralEntropy
4 spectralFlux
5 spectralKurtosis
6 spectralRolloffPoint
7 spectralSkewness
8 spectralSlope
9 harmonicRatio

This example uses `audioFeatureExtractor` 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 = 124999
```

```
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+windowLength
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.

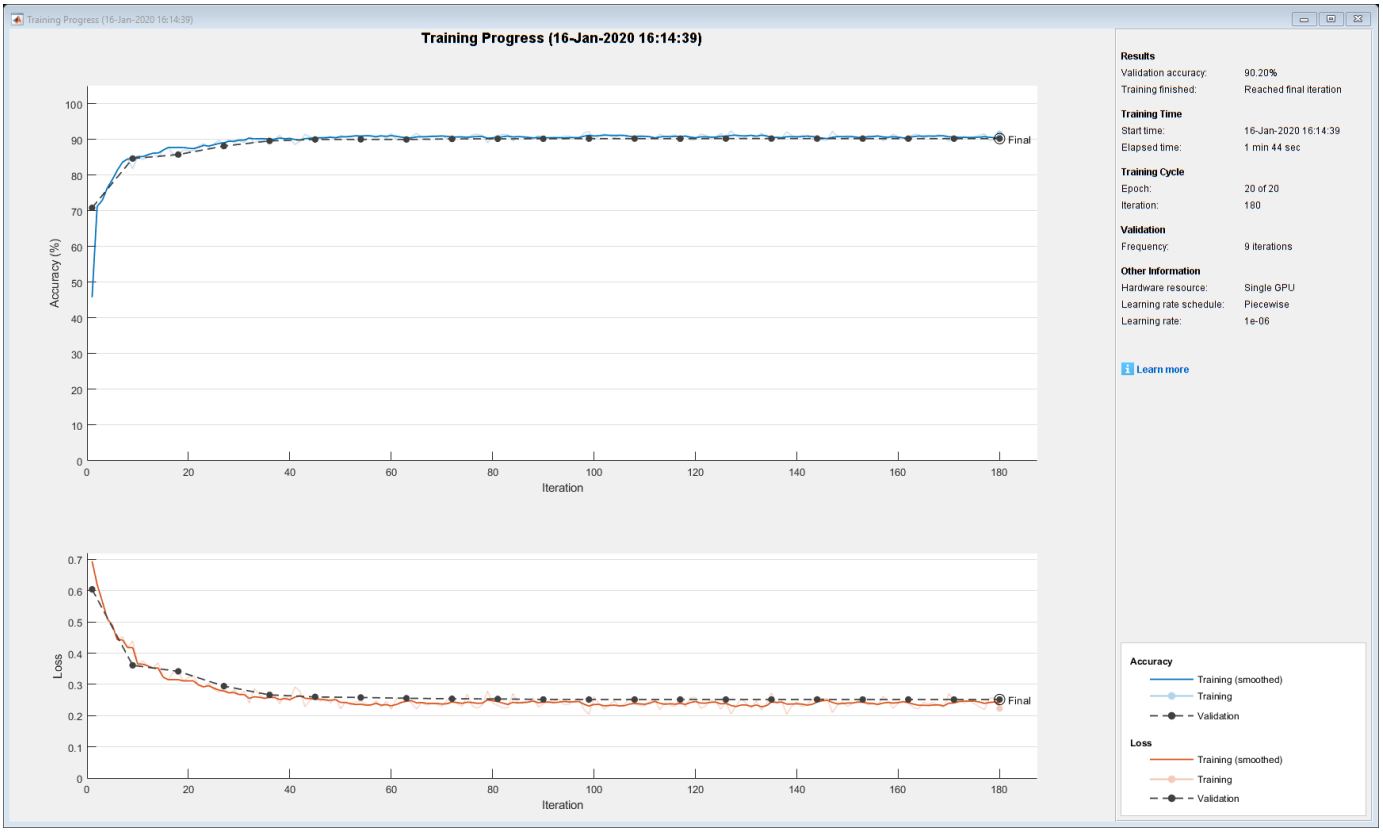
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
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: 90.195313 percent.
```

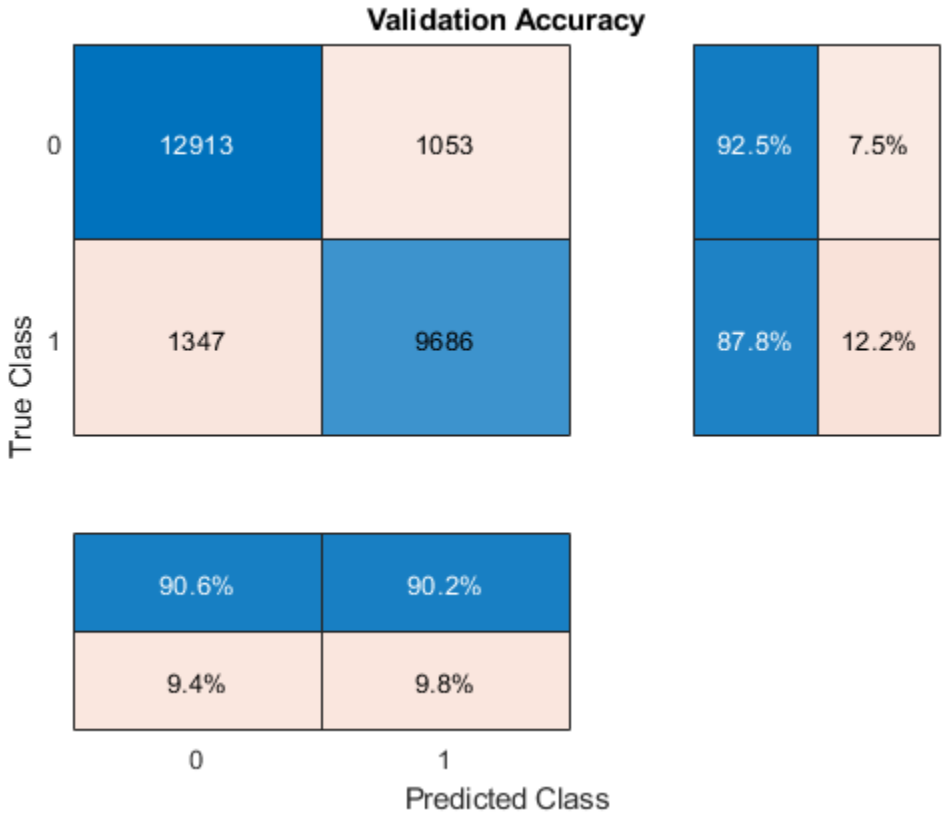
Use Trained Network to Detect Voice Activity

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 = ;
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` 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` object and a `dsp.MovingAverage` 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` 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 `dsp.TimeScope` to visualize the original speech signal, the noisy signal that the network is applied to, and the decision output from the network.

```
scope = dsp.TimeScope('SampleRate',fs, ...
    'TimeSpan',3, ...
    'BufferLength',fs*3*3, ...
    'YLimits',[-1.2 1.2], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'NumInputPorts',3, ...
    'LayoutDimensions',[3,1], ...
    'Title','Noisy Speech');
scope.ActiveDisplay = 2;
scope.Title = 'Clean Speech (Original)';
scope.ActiveDisplay = 3;
scope.Title = 'Detected Speech';
```

Create an `audioDeviceWriter` 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

`trainNetwork` | `trainingOptions`

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.

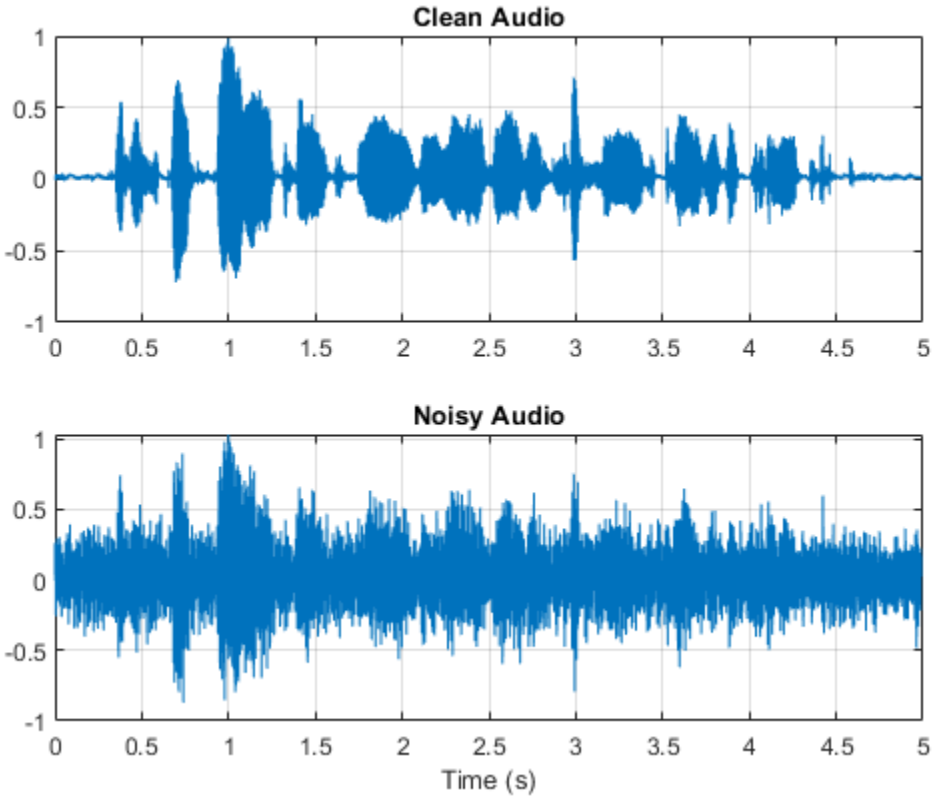
```
noise = audioread("WashingMachine-16-8-mono-1000secs.wav");

% 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 the Mozilla Common Voice dataset [1 on page 12-0] to train and test the deep learning networks. The dataset contains 48 kHz recordings of subjects speaking short sentences. Download the dataset and untar the downloaded file. Set `PathToDatabase` to the location of the data.

```
datafolder = PathToDatabase;
```

Use `audioDatastore` to create a datastore for all files in the dataset.

```
ads0 = audioDatastore(fullfile(datafolder,"clips"));
```

Use `readtable` to read the metadata associated with the audio files from the training set. The metadata is contained in the `train.tsv` file. Inspect the first few rows of the metadata.

```
metadata = readtable(fullfile(datafolder,"train.tsv"),"FileType","text");
head(metadata)
```

```
ans=8×8 table
                                                                client_id
    ____________________________________________________________________________________________________________

    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
```

```
{'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
{'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
{'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
{'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
{'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
{'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
{'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
```

Find the files in the datastore corresponding to the training set.

```
csvFiles = metadata.path;
adsFiles = ads0.Files;
[~,adsFiles,ext] = cellfun(@(x)fileparts(x),adsFiles,'UniformOutput',false);
[~,indA] = intersect(strcat(adsFiles,ext),csvFiles);
```

Create a subset training set from the large dataset.

```
ads = subset(ads0,indA);
```

You will train the deep learning networks on a subset of the files. Create a datastore subset containing the first 1000 files of the datastore.

```
numFilesInSubset = 1000;
ads = subset(ads,1:numFilesInSubset);
```

Use `read` to get the contents of the first file in the datastore.

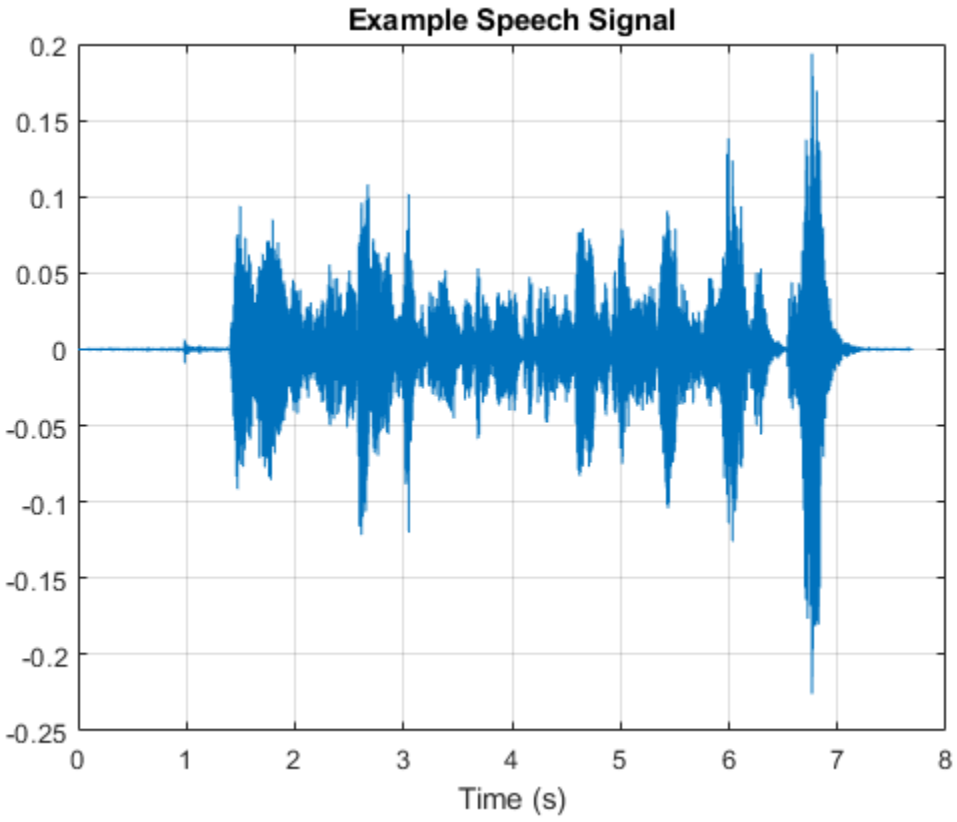
```
[audio,adsInfo] = read(ads);
```

Listen to the speech signal.

```
sound(audio,adsInfo.SampleRate)
```

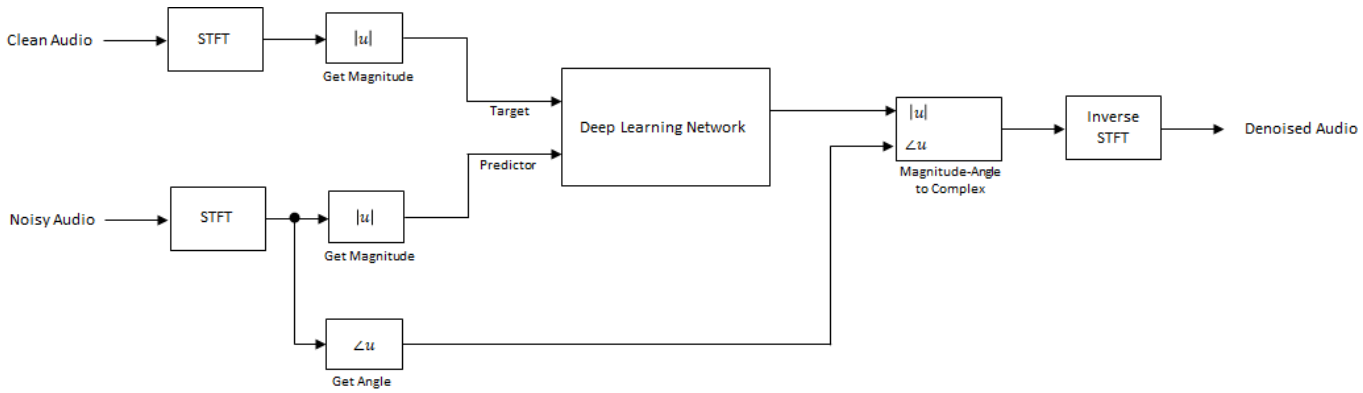
Plot the speech signal.

```
figure
t = (1/adsInfo.SampleRate) * (0:numel(audio)-1);
plot(t,audio)
title("Example Speech Signal")
xlabel("Time (s)")
grid on
```

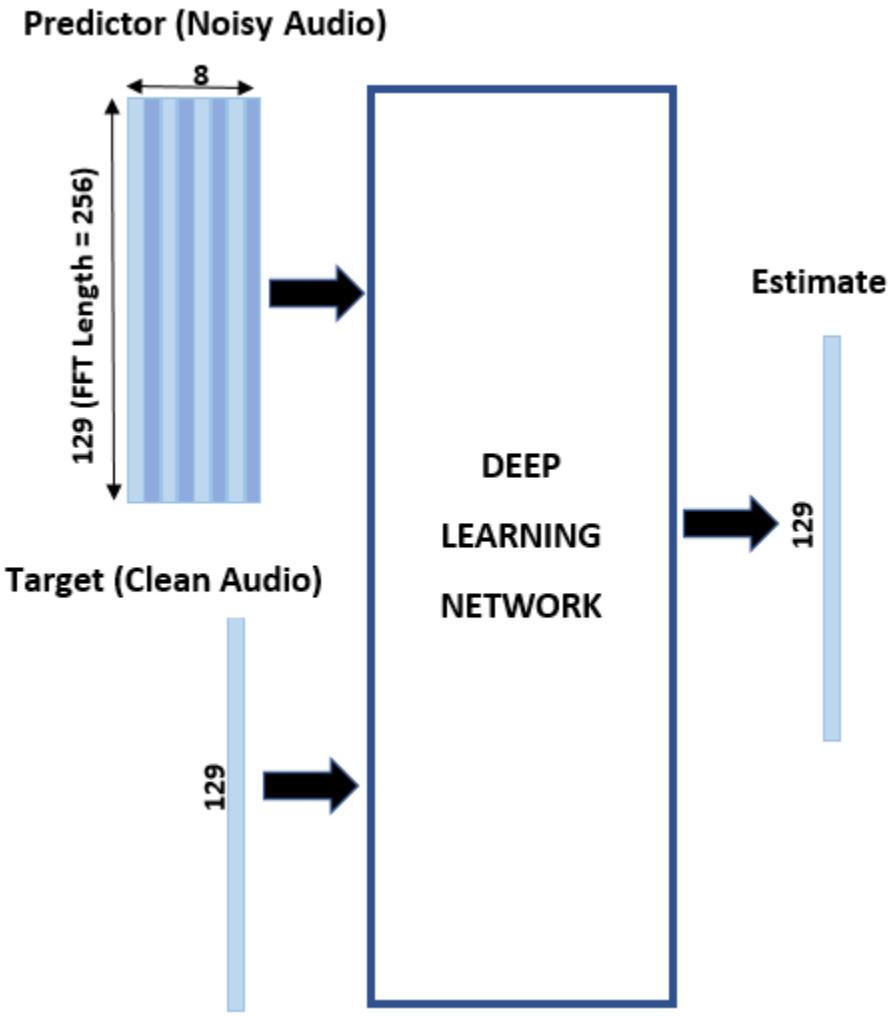


Deep Learning System Overview

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 12-0].



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` 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(ads);
```

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` 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   334
```

```
predictors = stftSegments;
size(predictors)
```

```
ans = 1×3

   129     8   334
```

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(ads)
T = tall(ads)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

T =

  M×1 tall cell array

    {369264×1 double}
    {129648×1 double}
    {142320×1 double}
    {203376×1 double}
    {213744×1 double}
    {190704×1 double}
    {160752×1 double}
    {193008×1 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 12-0 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 50 sec
Evaluation completed in 1 min 48 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));
```

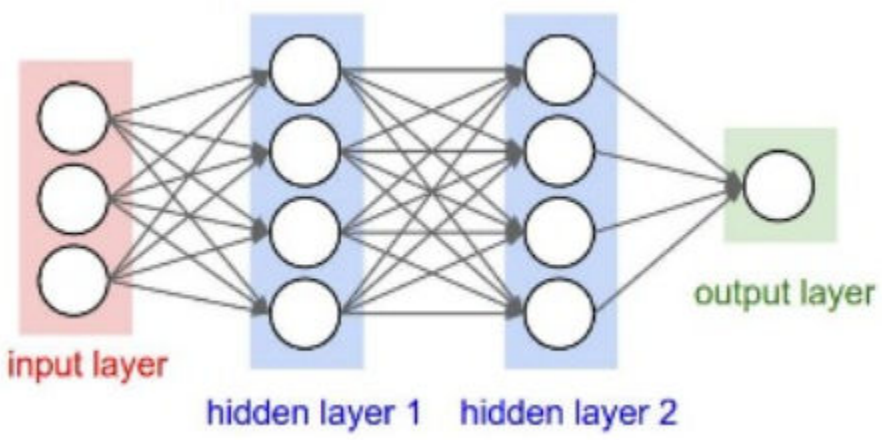
You will use 1% of the data for validation during training. Validation is useful to detect scenarios where the network is overfitting the training data.

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
    ];
```

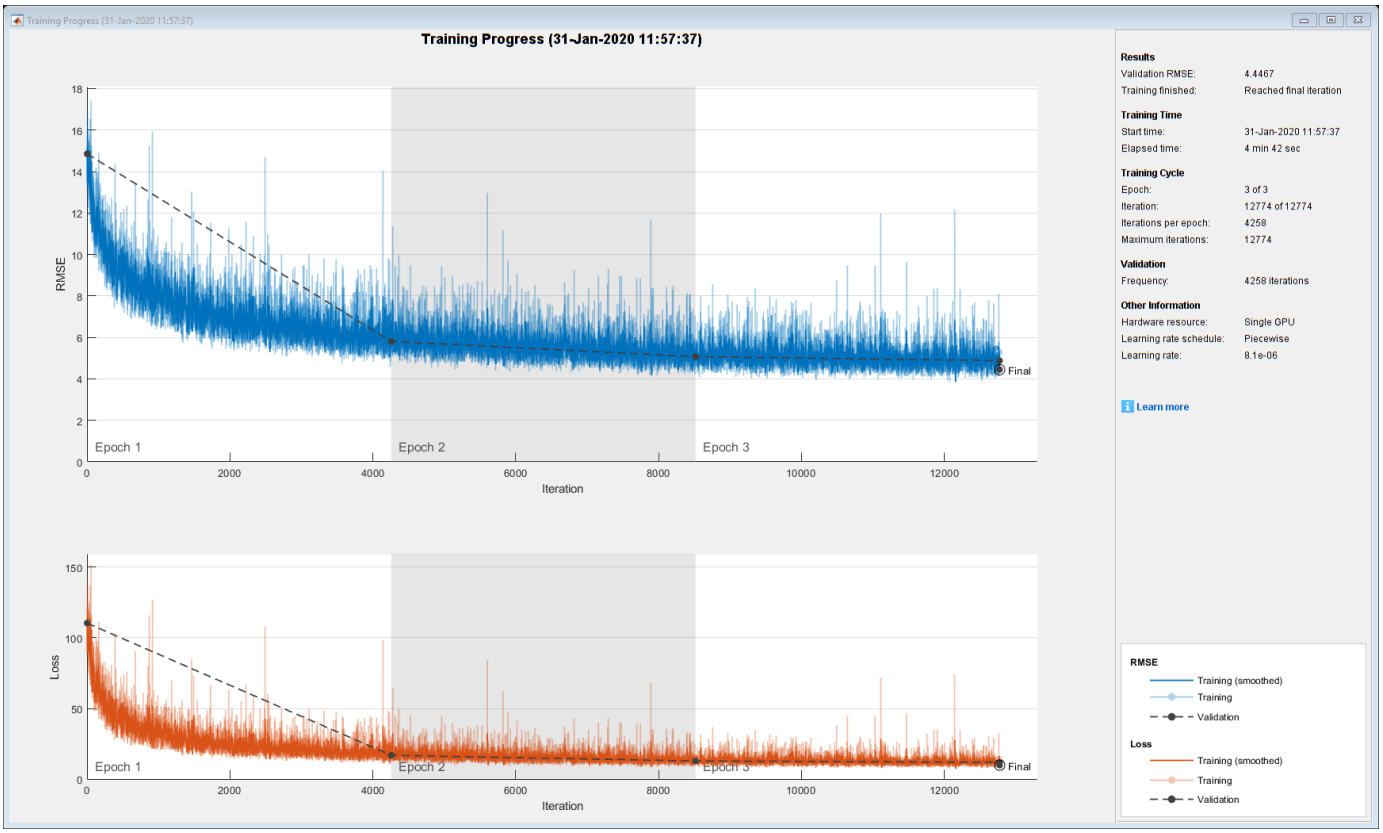
Next, specify the training options for the network. Set `MaxEpochs` to `3` so that the network makes 3 passes through the training data. Set `MiniBatchSize` of `128` so that the network looks at 128 training signals at a time. Specify `Plots` as `"training-progress"` to generate plots that show the training 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 load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = ;
if doTraining
    denoiseNetFullyConnected = trainNetwork(trainPredictors,trainTargets,layers,options);
else
    s = load("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 12-0]. 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 12-0], 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
          ];
```

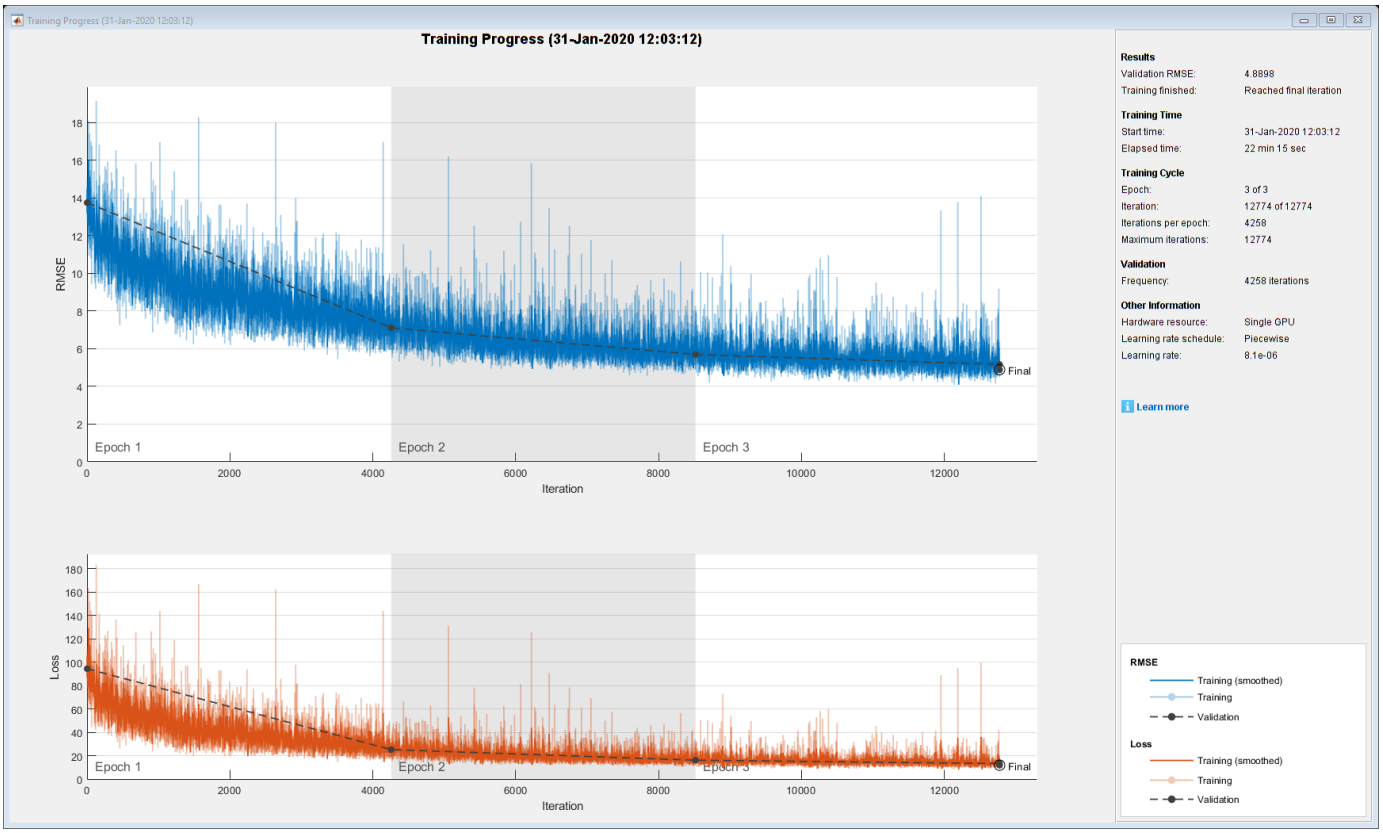
The training options are identical to the options for the fully connected network, except that the dimensions of the validation target signals are permuted to be consistent with the dimensions expected by the regression layer.

```
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 load a pre-trained network instead of training a network from scratch, set `doTraining` to `false`.

```
doTraining = ;
if doTraining
    denoiseNetFullyConvolutional = trainNetwork(trainPredictors,permute(trainTargets,[3 1 2 4]),
else
    s = load("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

You will create a validation dataset using the same approach you used for the training dataset. Use the `readtable` function to read the metadata associated with validation files.

```
metadata = readtable(fullfile(datafolder,"test.tsv"),"FileType","text");
```

Locate the validation files in the datastore.

```
csvFiles = metadata.path;
adsFiles = ads0.Files;
[~,adsFiles,ext] = cellfun(@(x)fileparts(x),adsFiles,'UniformOutput',false);
[~,indA] = intersect(strcat(adsFiles,ext),csvFiles);
```

Create a validation datastore from the large datastore.

```
ads = subset(ads0,indA);
```

Create a datastore subset containing the first 100 files of the datastore.

```
ads = subset(ads,1:100);
```

Shuffle the files in the datastore.

```
ads = shuffle(ads);
```

Read the contents of a file from the datastore.

```
[cleanAudio,adsInfo] = read(ads);
```

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.wav");
```

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];
```

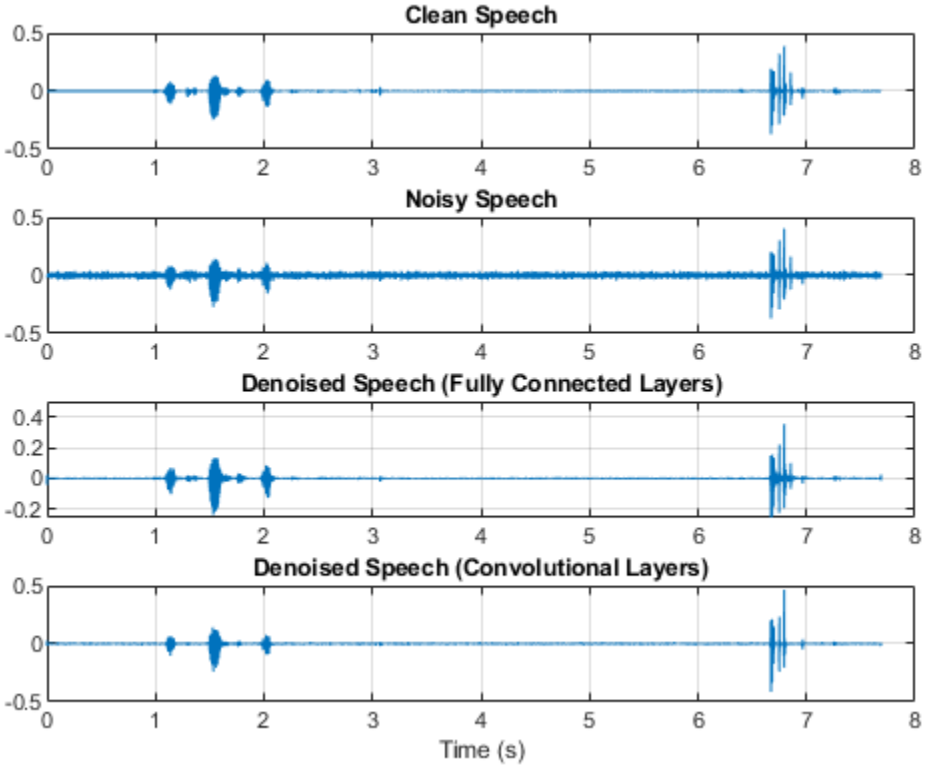
Compute the denoised speech signals. `istft` performs the inverse STFT. Use the phase of the noisy STFT vectors to reconstruct the time-domain signal.

```
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)))
```

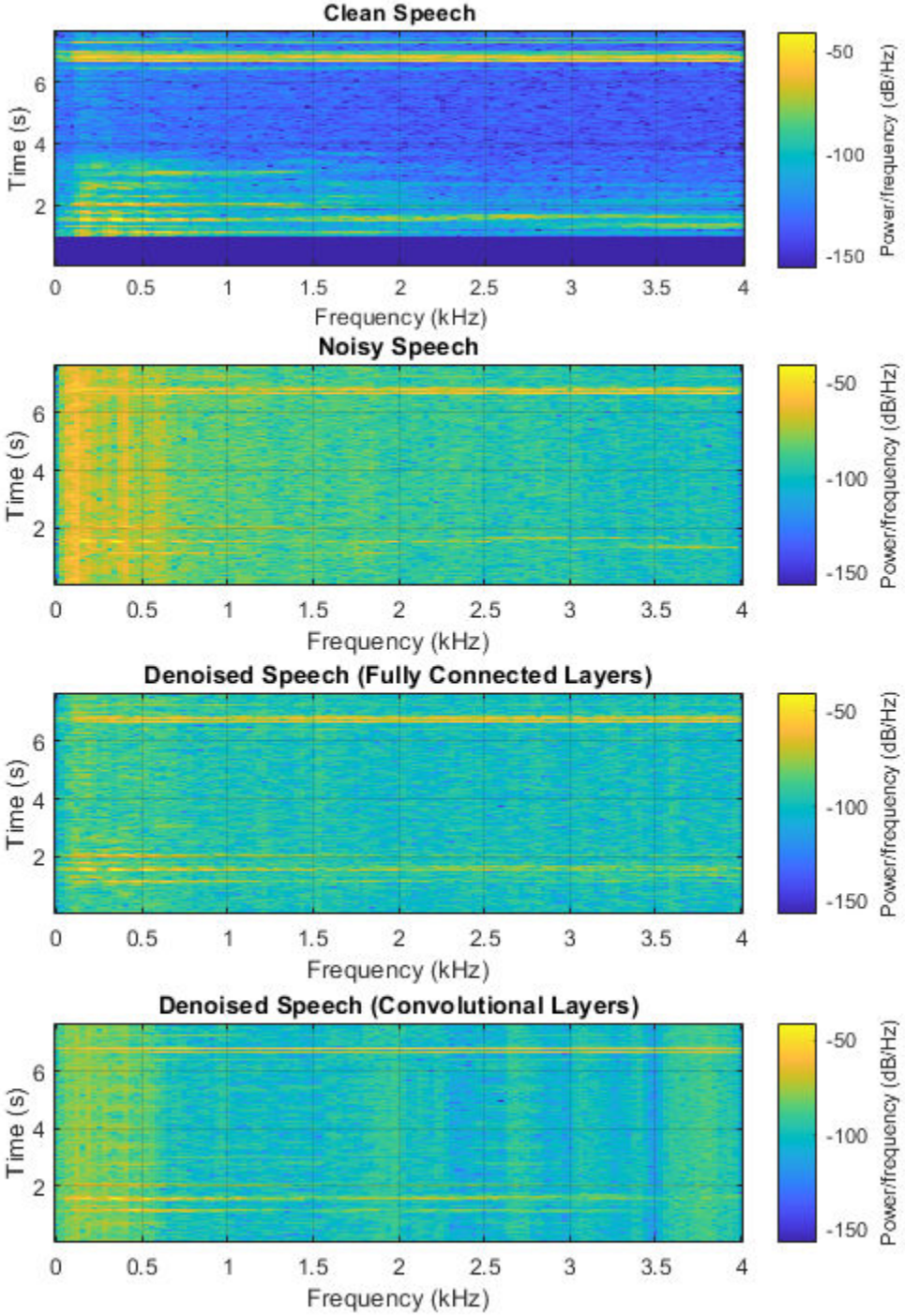
```
title("Clean Speech")
grid on

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.

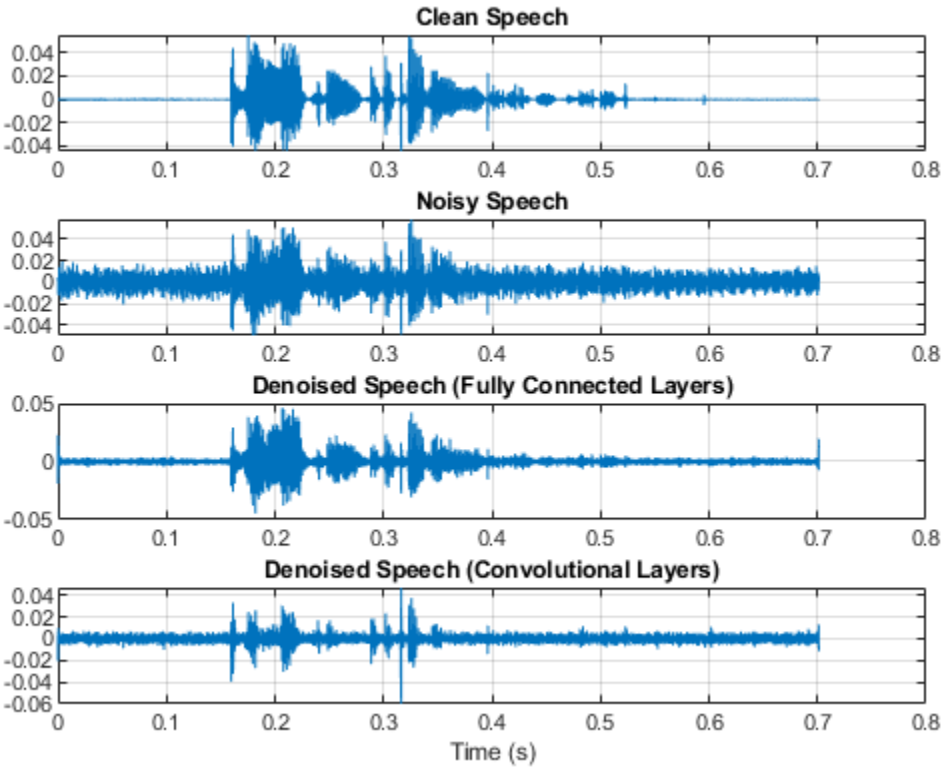
```
sound(denoisedAudioFullyConvolutional,fs)
```

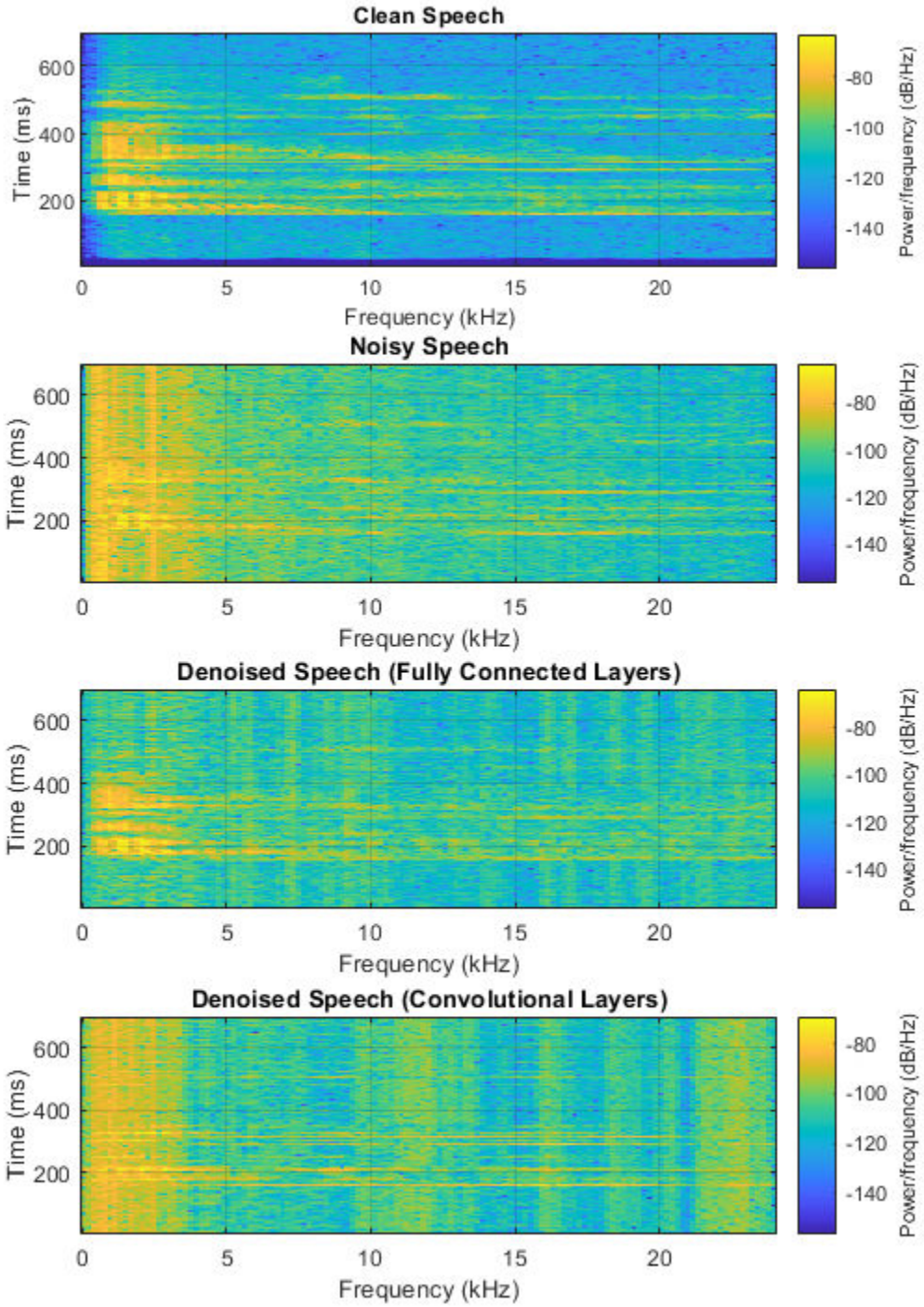
Listen to clean speech.

```
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
```



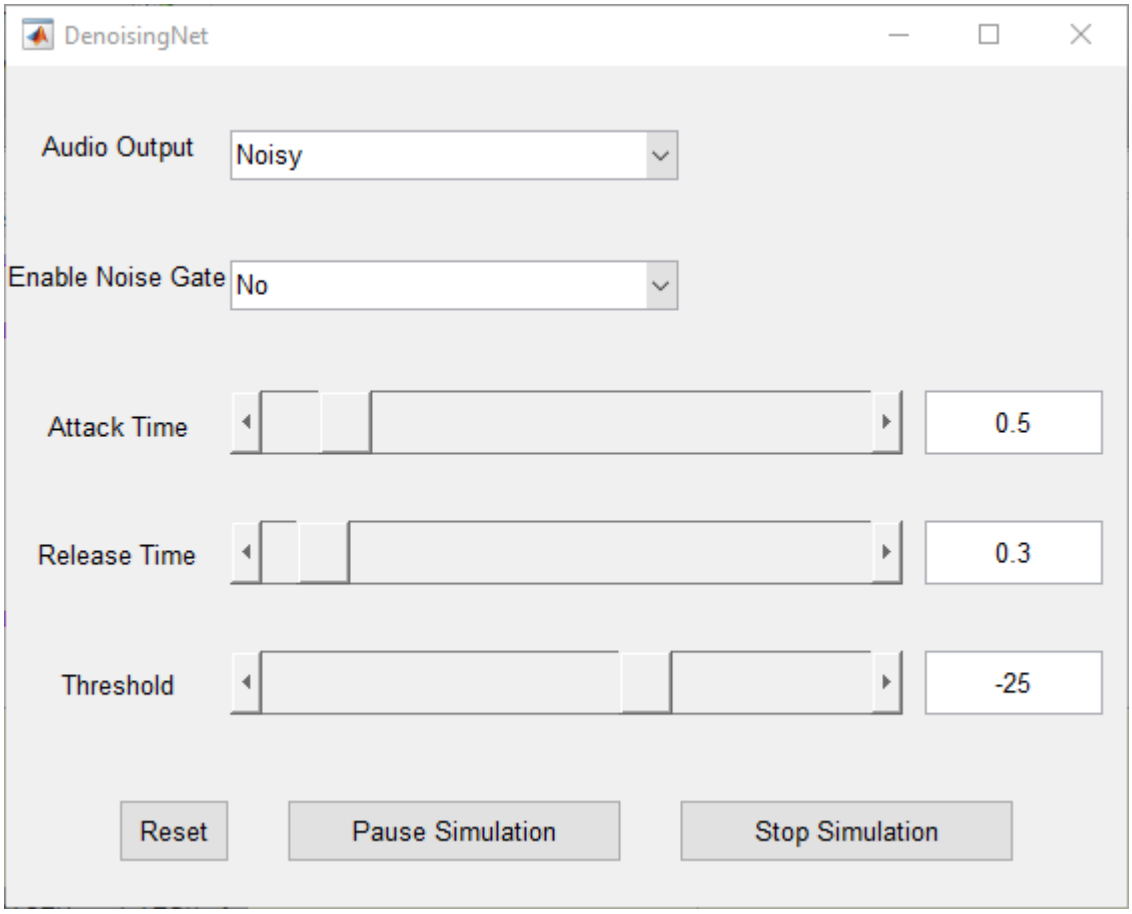

Clean Speech
Noisy Speech
Denoised Speech (Fully Connected Layers)
Denoised Speech (Convolutional Layers)
Time (ms)
600
400
200
0
5
10
15
20
Frequency (kHz)
-80
-100
-120
-140
Power/frequency (dB/Hz)

Real-Time Application

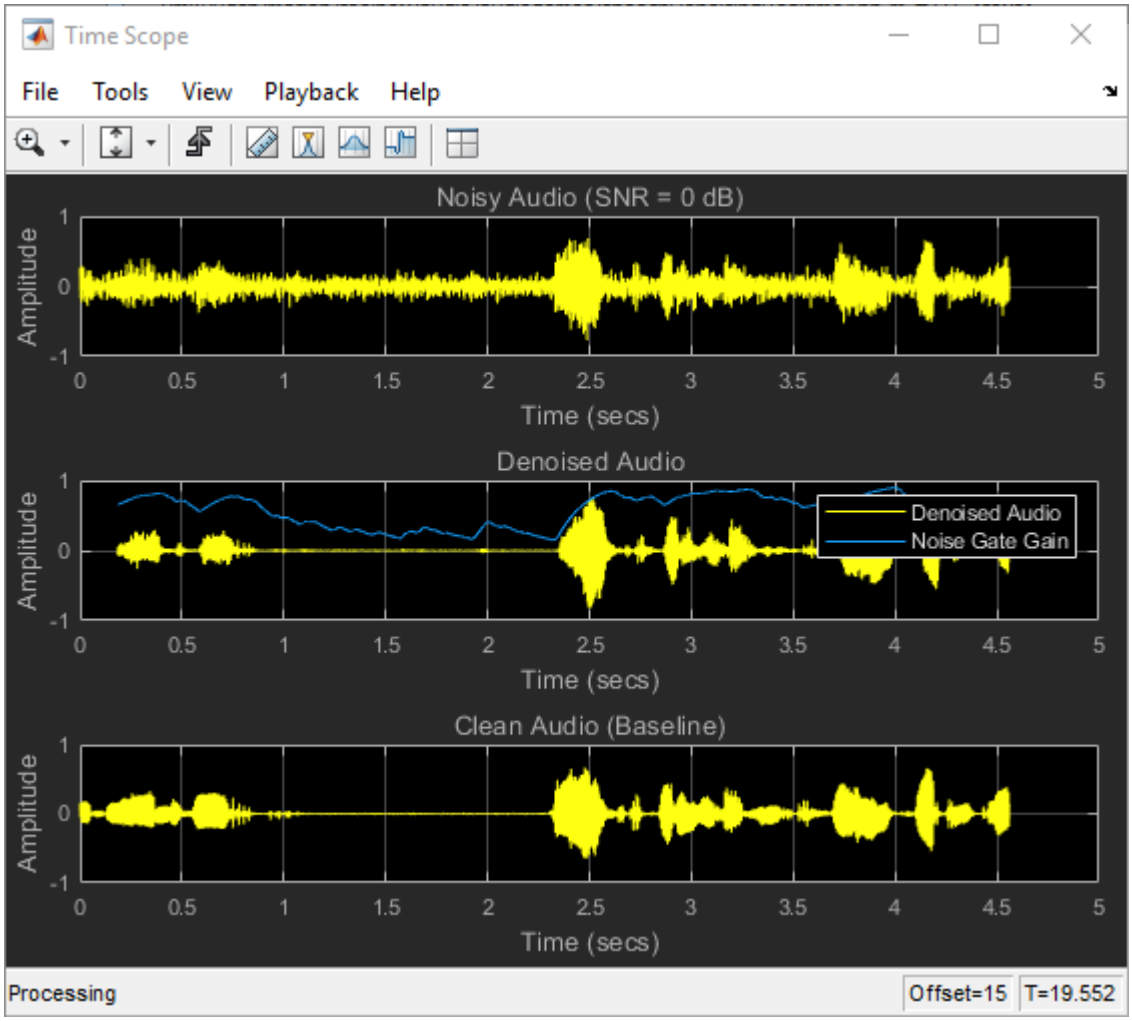
The procedure in the previous section passes the entire spectrum of the noisy signal to `predict`. This is not suitable for real-time applications where low latency is a requirement.

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

`trainNetwork` | `trainingOptions`

More About

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

Classify Gender Using LSTM Networks

This example shows how to classify the gender of a speaker using deep learning. The example uses a Bidirectional Long Short-Term Memory (BiLSTM) network and Gammatone Cepstral Coefficients (gtcc), pitch, harmonic ratio, and several spectral shape descriptors.

Introduction

Gender classification based on speech signals is an essential component of many audio systems, such as automatic speech recognition, speaker recognition, and content-based multimedia indexing.

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. 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 bidirectional LSTM layers.

This example trains the LSTM network with sequences of gammatone cepstrum coefficients (`gtcc`), pitch estimates (`pitch`), harmonic ratio (`harmonicRatio`), and several spectral shape descriptors ("Spectral Descriptors" (Audio Toolbox)).

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.

Preprocess Audio Data

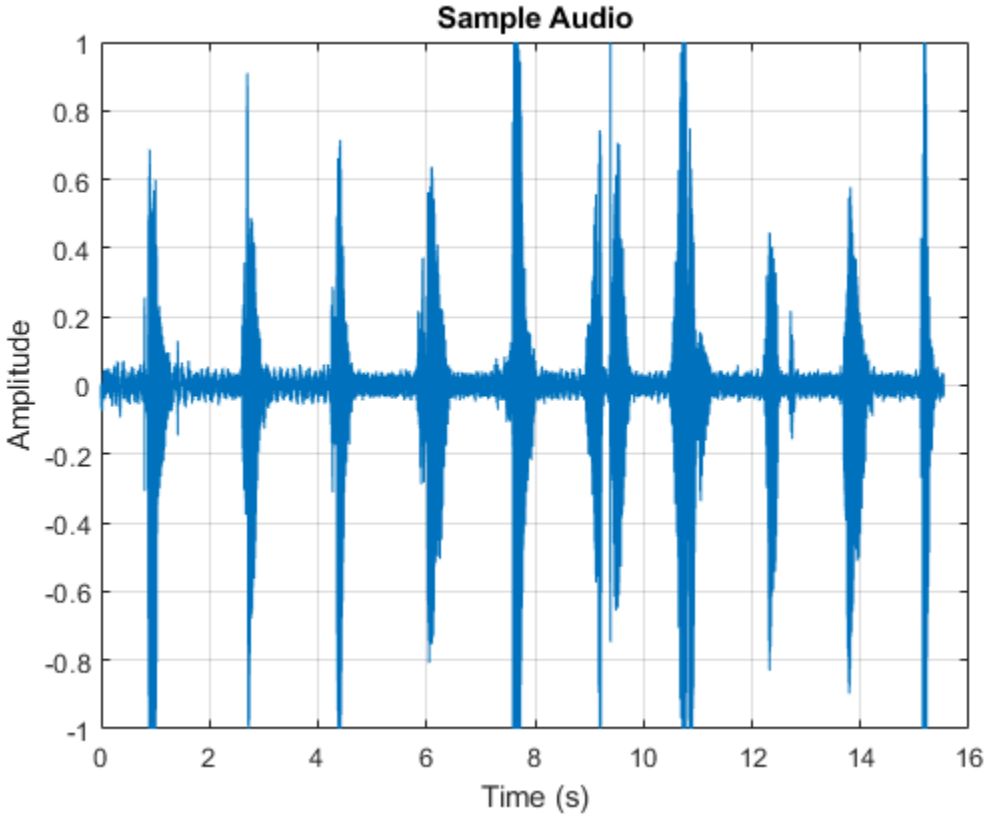
The BiLSTM network used in this example works best when using sequences of feature vectors. To illustrate the preprocessing pipeline, this example walks through the steps for a single audio file.

Read the contents of an audio file containing speech. The speaker gender is male.

```
[audioIn,Fs] = audioread('Counting-16-44p1-mono-15secs.wav');
labels = {'male'};
```

Plot the audio signal and then listen to it using the `sound` command.

```
timeVector = (1/Fs) * (0:size(audioIn,1)-1);
figure
plot(timeVector,audioIn)
ylabel("Amplitude")
xlabel("Time (s)")
title("Sample Audio")
grid on
```




```
sound(audioIn,Fs)
```

The speech signal has silence segments that do not contain useful information pertaining to the gender of the speaker. Use `detectSpeech` to locate segments of speech in the audio signal.

```
speechIndices = detectSpeech(audioIn,Fs);
```

Create an `audioFeatureExtractor` to extract features from the audio data. A speech signal is dynamic in nature and changes over time. It is assumed that speech signals are stationary on short time scales and their processing is often done in windows of 20-40 ms. Specify 30 ms windows with 20 ms overlap.

```
extractor = audioFeatureExtractor( ...
    "SampleRate",Fs, ...
    "Window",hamming(round(0.03*Fs),"periodic"), ...
    "OverlapLength",round(0.02*Fs), ...
    ...
    "gtcc",true, ...
    "gtccDelta",true, ...
    "gtccDeltaDelta",true, ...
    ...
    "SpectralDescriptorInput","melSpectrum", ...
    "spectralCentroid",true, ...
    "spectralEntropy",true, ...
    "spectralFlux",true, ...
    "spectralSlope",true, ...
    ...
```

```
    "pitch",true, ...
    "harmonicRatio",true);
```

Extract features from each audio segment. The output from `audioFeatureExtractor` is a `numFeatureVectors`-by-`numFeatures` array. The `sequenceInputLayer` used in this example requires time to be along the second dimension. Permute the output array so that time is along the second dimension.

```
featureVectorsSegment = {};
for ii = 1:size(speechIndices,1)
    featureVectorsSegment{end+1} = ( extract(extractor,audioIn(speechIndices(ii,1):speechIndices
end
numSegments = size(featureVectorsSegment)
```

```
numSegments = 1×2

     1    11
```

```
[numFeatures,numFeatureVectorsSegment1] = size(featureVectorsSegment{1})
```

```
numFeatures = 45
```

```
numFeatureVectorsSegment1 = 124
```

Replicate the labels so that they are in one-to-one correspondence with segments.

```
labels = repelem(labels,size(speechIndices,1))
```

```
labels = 1×11 cell
    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}    {'male'}
```

When using a `sequenceInputLayer`, it is often advantageous to use sequences of consistent length. Convert the arrays of feature vectors into sequences of feature vectors. Use 20 feature vectors per sequence with 5 feature vector overlap.

```
featureVectorsPerSequence = 20;
featureVectorOverlap = 5;
hopLength = featureVectorsPerSequence - featureVectorOverlap;

idx1 = 1;
featuresTrain = {};
sequencePerSegment = zeros(numel(featureVectorsSegment),1);
for ii = 1:numel(featureVectorsSegment)
    sequencePerSegment(ii) = max(floor((size(featureVectorsSegment{ii},2) - featureVectorsPerSequ
    idx2 = 1;
    for j = 1:sequencePerSegment(ii)
        featuresTrain{idx1,1} = featureVectorsSegment{ii}(:,idx2:idx2 + featureVectorsPerSequence
        idx1 = idx1 + 1;
        idx2 = idx2 + hopLength;
    end
end
```

For conciseness, the helper function HelperFeatureVector2Sequence on page 12-0 encapsulates the above processing and is used throughout the rest of the example.

Replicate the labels so that they are in one-to-one correspondence with the training set.

```
labels = repelem(labels,sequencePerSegment);
```

The result of the preprocessing pipeline is a `NumSequence`-by-1 cell array of `NumFeatures`-by-`FeatureVectorsPerSequence` matrices. Labels is a `NumSequence`-by-1 array.

```
NumSequence = numel(featuresTrain)
```

```
NumSequence = 27
```

```
[NumFeatures,FeatureVectorsPerSequence] = size(featuresTrain{1})
```

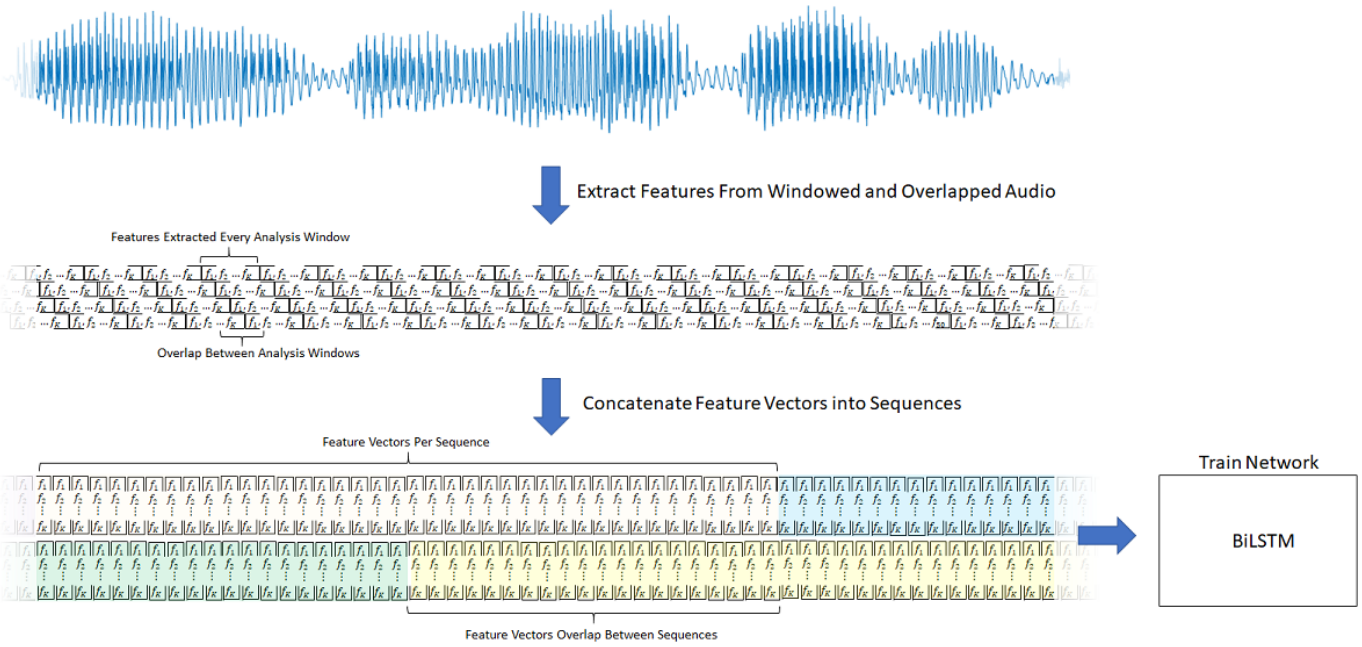
```
NumFeatures = 45
```

```
FeatureVectorsPerSequence = 20
```

```
NumSequence = numel(labels)
```

```
NumSequence = 27
```

The figure provides an overview of the feature extraction used per detected speech region.



Create Training and Test Datastores

This example uses the Mozilla Common Voice dataset [1] on page 12-0 . The dataset contains 48 kHz recordings of subjects speaking short sentences. Download the dataset and untar the downloaded file. Set `PathToDatabase` to the location of the data.

```
datafolder = PathToDatabase;
```

Use `audioDatastore` to create a datastore for all files in the dataset.

```
loc = fullfile(datafolder,"clips");
ads = audioDatastore(loc);
```

Since only a fraction of dataset files are annotated with gender information, use both the training and validation sets to train the network. Use the test set to validate the network. Use `readtable` to read

the metadata associated with the audio files. The metadata is contained in the `train.tsv, dev.tsv, and test.tsv` files. Inspect the first few rows of the training metadata.

```
metadataTrain = readtable(fullfile(datafolder,"train.tsv"),"FileType","text");
metadataDev = readtable(fullfile(datafolder,"dev.tsv"),"FileType","text");
metadataTrain = [metadataTrain;metadataDev];

head(metadataTrain)
```

```
ans=8×8 table
                                                                client_id
    ____________________________________________________________________________________________________________

    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
    {'4f29be8fe932d773576dd3df5e111929f4e22422322450983695eaa8625a12659cd3e999a061a29ebe7178383
    {'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
    {'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
    {'4f3b69348cb65923dff20efe0eaef4fbc8797f9c2240447ae48764e36fab63867dbf6947bfb8ff623cab4f1d1e
```

Remove rows of the metadata that do not contain gender information. Remove rows from the metadata that do not contain age information, or if the age information indicates a teenager.

```
containsGenderInfo = contains(metadataTrain.gender,'male') | contains(metadataTrain.gender,'fema
isAdult = ~contains(metadataTrain.age,'teens') & ~isempty(metadataTrain.age);
highUpVotes = metadataTrain.up_votes >= 3;
metadataTrain(~containsGenderInfo | ~isAdult | ~highUpVotes,:) = [];
trainFiles = fullfile(loc,metadataTrain.path);
```

Subset the datastore to only include files corresponding to adult speakers with gender information.

```
[~,idxA,idxB] = intersect(ads.Files,trainFiles);
adsTrain = subset(ads,idxA);
adsTrain.Labels = metadataTrain.gender(idxB);
```

Use `countEachLabel` to inspect the gender breakdown of the training set.

```
labelDistribution = countEachLabel(adsTrain)
```

```
labelDistribution=2×2 table
    Label     Count
    ______    _____

    female    1554
    male      4491
```

Use `splitEachLabel` to reduce the training datastore so that there are an equal number of male and female speakers.

```
numFilesPerGender = min(labelDistribution.Count);
adsTrain = splitEachLabel(adsTrain,numFilesPerGender);
countEachLabel(adsTrain)
```

```
ans=2×2 table
    Label     Count
```

```
    ______    _____

    female    1554
    male      1554
```

Create the validation set using the same steps.

```
metadataValidation = readtable(fullfile(datafolder,"test.tsv"),"FileType","text");
containsGenderInfo = contains(metadataValidation.gender,'male') | contains(metadataValidation.ger
isAdult = ~contains(metadataValidation.age,'teens') & ~isempty(metadataValidation.age);
metadataValidation(~containsGenderInfo | ~isAdult,:) = [];
validationFiles = fullfile(loc,metadataValidation.path);
[~,idxA,idxB] = intersect(ads.Files,validationFiles);
adsValidation = subset(ads,idxA);
adsValidation.Labels = metadataValidation.gender(idxB);
countEachLabel(adsValidation)
```

```
ans=2×2 table
    Label     Count
    ______    _____

    female     312
    male      1608
```

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 training dataset by a factor of 20
    adsTrain = splitEachLabel(adsTrain,round(numel(adsTrain.Files) / 2 / 20));
    adsValidation = splitEachLabel(adsValidation,20);
end
```

Create Training and Validation Sets

Determine the sample rate of audio files in the data set, and then update the sample rate, window, and overlap length of the audio feature extractor.

```
[~,adsInfo] = read(adsTrain);
Fs = adsInfo.SampleRate;
extractor.SampleRate = Fs;
extractor.Window = hamming(round(0.03*Fs),"periodic");
extractor.OverlapLength = round(0.02*Fs);
```

To speed up processing, distribute computations over multiple workers. If you have Parallel Computing Toolbox™, the example partitions the datastore so that the feature extraction occurs in parallel across available workers. Determine the optimal number of partitions for your system. If you do not have Parallel Computing Toolbox™, the example uses a single worker.

```
if ~isempty(ver('parallel')) && ~reduceDataset
    pool = gcp;
    numPar = numpartitions(adsTrain,pool);
else
    numPar = 1;
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

In a loop:

1. Read from the audio datastore.
2. Detect regions of speech.
3. Extract feature vectors from the regions of speech.

Replicate the labels so that they are in one-to-one correspondence with the feature vectors.

```
labelsTrain = [];
featureVectors = {};

% Loop over optimal number of partitions
parfor ii = 1:numPar

    % Partition datastore
    subds = partition(adsTrain,numPar,ii);

    % Preallocation
    featureVectorsInSubDS = {};
    segmentsPerFile = zeros(numel(subds.Files),1);

    % Loop over files in partitioned datastore
    for jj = 1:numel(subds.Files)

        % 1. Read in a single audio file
        audioIn = read(subds);

        % 2. Determine the regions of the audio that correspond to speech
        speechIndices = detectSpeech(audioIn,Fs);

        % 3. Extract features from each speech segment
        segmentsPerFile(jj) = size(speechIndices,1);
        features = cell(segmentsPerFile(jj),1);
        for kk = 1:size(speechIndices,1)
            features{kk} = ( extract(extractor,audioIn(speechIndices(kk,1):speechIndices(kk,2))))
        end
        featureVectorsInSubDS = [featureVectorsInSubDS;features(:)];

    end
    featureVectors = [featureVectors;featureVectorsInSubDS];

    % Replicate the labels so that they are in one-to-one correspondance
    % with the feature vectors.
    repedLabels = repelem(subds.Labels,segmentsPerFile);
    labelsTrain = [labelsTrain;repedLabels(:)];
end
```

In classification applications, 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.

```
allFeatures = cat(2,featureVectors{:});
allFeatures(isinf(allFeatures)) = nan;
```

```
M = mean(allFeatures,2,'omitnan');
S = std(allFeatures,0,2,'omitnan');
featureVectors = cellfun(@(x)(x-M)./S,featureVectors,'UniformOutput',false);
for ii = 1:numel(featureVectors)
    idx = find(isnan(featureVectors{ii}));
    if ~isempty(idx)
        featureVectors{ii}(idx) = 0;
    end
end
```

Buffer the feature vectors into sequences of 20 feature vectors with 10 overlap. If a sequence has less than 20 feature vectors, drop it.

```
[featuresTrain,trainSequencePerSegment] = HelperFeatureVector2Sequence(featureVectors,featureVec
```

Replicate the labels so that they are in one-to-one correspondence with the sequences.

```
labelsTrain = repelem(labelsTrain,[trainSequencePerSegment{:}]);
labelsTrain = categorical(labelsTrain);
```

Create the validation set using the same steps used to create the training set.

```
labelsValidation = [];
featureVectors = {};
valSegmentsPerFile = [];
parfor ii = 1:numPar
    subds = partition(adsValidation,numPar,ii);
    featureVectorsInSubDS = {};
    valSegmentsPerFileInSubDS = zeros(numel(subds.Files),1);
    for jj = 1:numel(subds.Files)
        audioIn = read(subds);
        speechIndices = detectSpeech(audioIn,Fs);
        numSegments = size(speechIndices,1);
        features = cell(valSegmentsPerFileInSubDS(jj),1);
        for kk = 1:numSegments
            features{kk} = ( extract(extractor,audioIn(speechIndices(kk,1):speechIndices(kk,2)))
        end
        featureVectorsInSubDS = [featureVectorsInSubDS;features(:)];
        valSegmentsPerFileInSubDS(jj) = numSegments;
    end
    repedLabels = repelem(subds.Labels,valSegmentsPerFileInSubDS);
    labelsValidation = [labelsValidation;repedLabels(:)];
    featureVectors = [featureVectors;featureVectorsInSubDS];
    valSegmentsPerFile = [valSegmentsPerFile;valSegmentsPerFileInSubDS];
end

featureVectors = cellfun(@(x)(x-M)./S,featureVectors,'UniformOutput',false);
for ii = 1:numel(featureVectors)
    idx = find(isnan(featureVectors{ii}));
    if ~isempty(idx)
        featureVectors{ii}(idx) = 0;
    end
end

[featuresValidation,valSequencePerSegment] = HelperFeatureVector2Sequence(featureVectors,featureV
labelsValidation = repelem(labelsValidation,[valSequencePerSegment{:}]);
labelsValidation = categorical(labelsValidation);
```

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 a hidden bidirectional LSTM layer with an output size of 50 and output a sequence. Then, specify a bidirectional LSTM layer with an output size of 50 and output the last element of the sequence. This command instructs the bidirectional LSTM layer to map its input into 50 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(featuresTrain{1},1))
    bilstmLayer(50,"OutputMode","sequence")
    bilstmLayer(50,"OutputMode","last")
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer];
```

Next, specify the training options for the classifier. Set `MaxEpochs` to 4 so that the network makes 4 passes through the training data. Set `MiniBatchSize` of 256 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. Specify `Shuffle` as `"every-epoch"` to shuffle the training sequence at the beginning of each epoch. Specify `LearnRateSchedule` to `"piecewise"` to decrease the learning rate by a specified factor (0.1) every time a certain number of epochs (1) has passed.

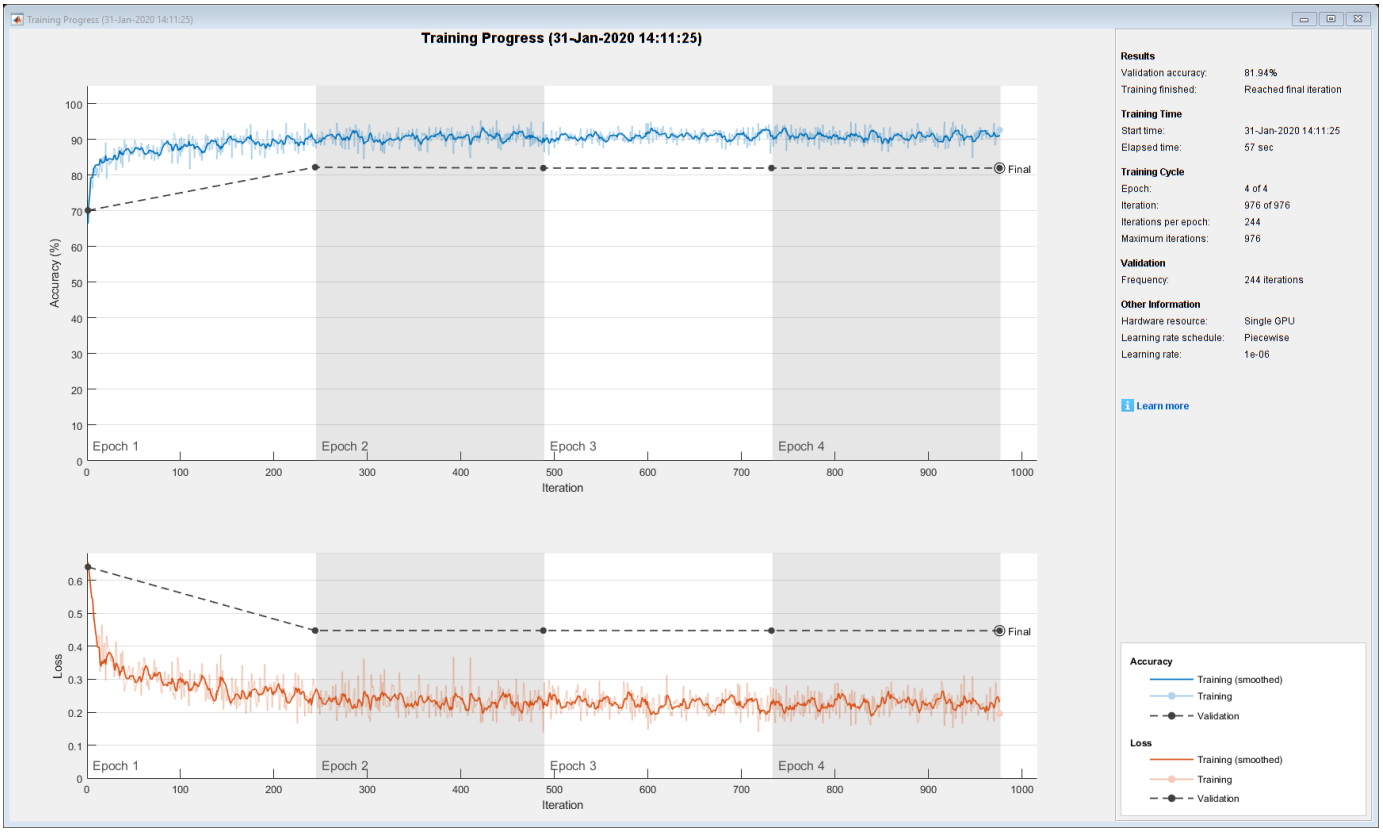
This example uses the adaptive moment estimation (ADAM) solver. ADAM performs better with recurrent neural networks (RNNs) like LSTMs than the default stochastic gradient descent with momentum (SGDM) solver.

```
miniBatchSize = 256;
validationFrequency = floor(numel(labelsTrain)/miniBatchSize);
options = trainingOptions("adam", ...
    "MaxEpochs",4, ...
    "MiniBatchSize",miniBatchSize, ...
    "Plots","training-progress", ...
    "Verbose",false, ...
    "Shuffle","every-epoch", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropFactor",0.1, ...
    "LearnRateDropPeriod",1, ...
    'ValidationData',{featuresValidation,labelsValidation}, ...
    'ValidationFrequency',validationFrequency);
```

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.

```
net = trainNetwork(featuresTrain,labelsTrain,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 at the start of training, or 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.

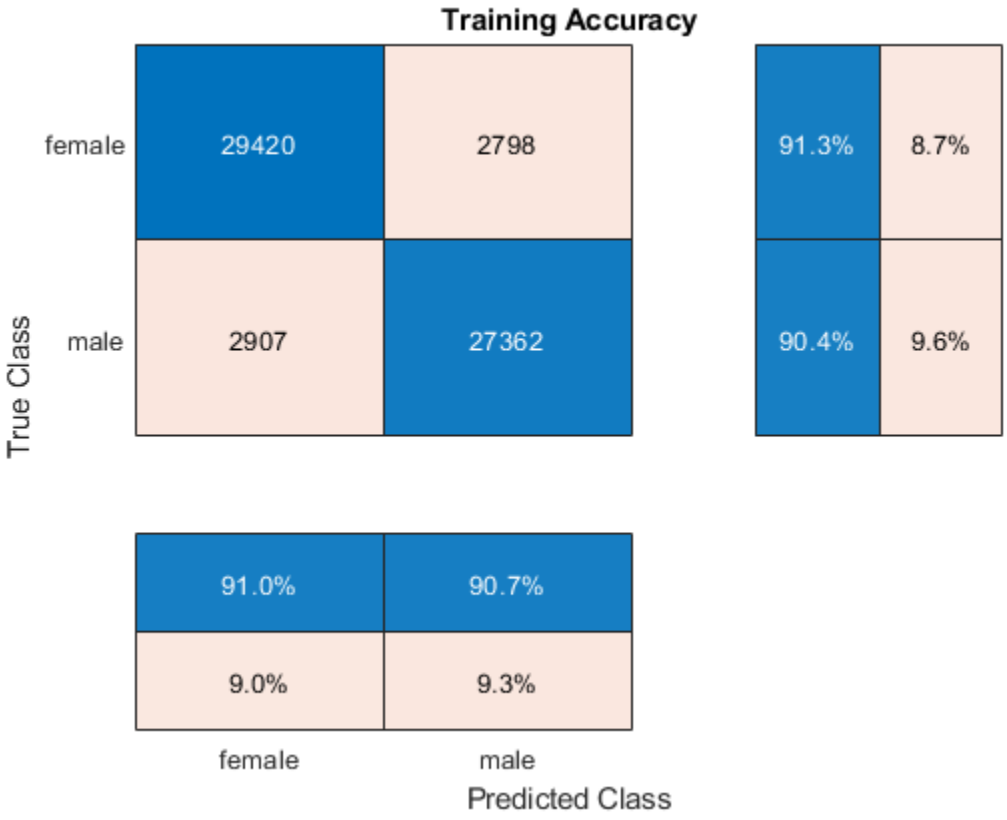
Visualize the Training Accuracy

Calculate the training accuracy, which represents the accuracy of the classifier on the signals on which it was trained. First, classify the training data.

```
prediction = classify(net,featuresTrain);
```

Plot the confusion matrix. Display the precision and recall for the two classes by using column and row summaries.

```
figure
cm = confusionchart(categorical(labelsTrain),prediction,'title','Training Accuracy');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



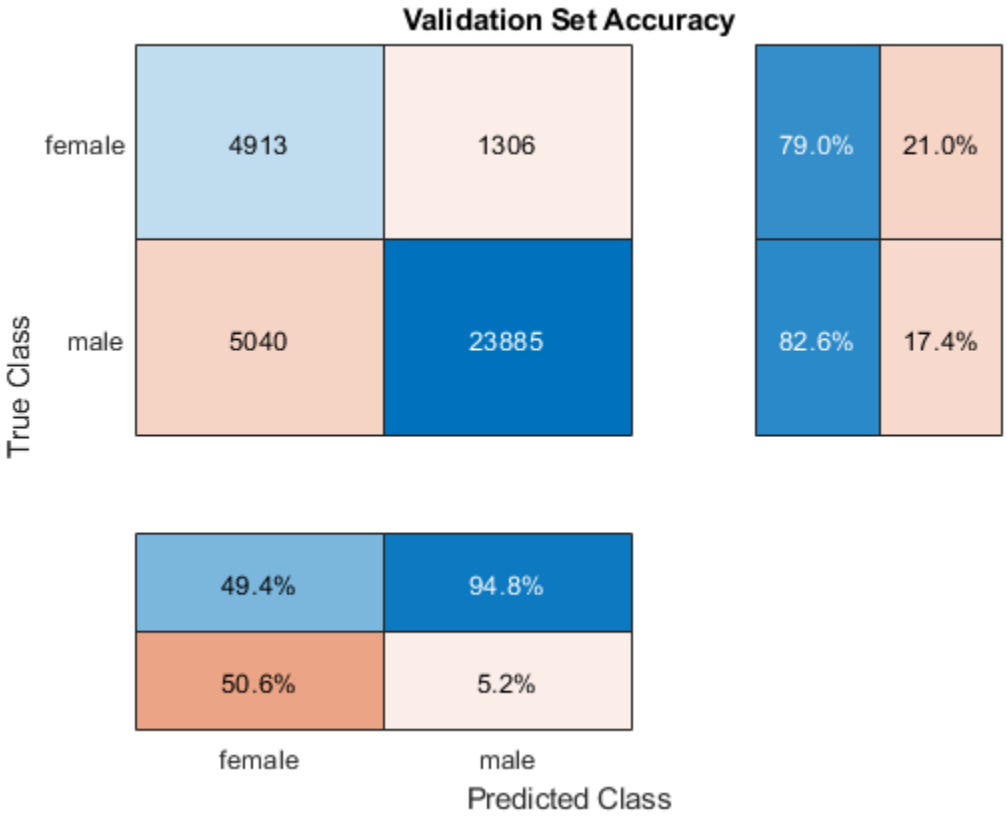
Visualize the Validation Accuracy

Calculate the validation accuracy. First, classify the training data.

```
[prediction,probabilities] = classify(net,featuresValidation);
```

Plot the confusion matrix. Display the precision and recall for the two classes by using column and row summaries.

```
figure
cm = confusionchart(categorical(labelsValidation),prediction,'title','Validation Set Accuracy');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



The example generated multiple sequences from each training speech file. Higher accuracy can be achieved by considering the output class of all sequences corresponding to the same file, and applying a "max-rule" decision, where the class with the segment with the highest confidence score is selected.

Determine the number of sequences generated per file in the validation set.

```
sequencePerFile = zeros(size(valSegmentsPerFile));
valSequencePerSegmentMat = cell2mat(valSequencePerSegment);
idx = 1;
for ii = 1:numel(valSegmentsPerFile)
    sequencePerFile(ii) = sum(valSequencePerSegmentMat(idx:idx+valSegmentsPerFile(ii)-1));
    idx = idx + valSegmentsPerFile(ii);
end
```

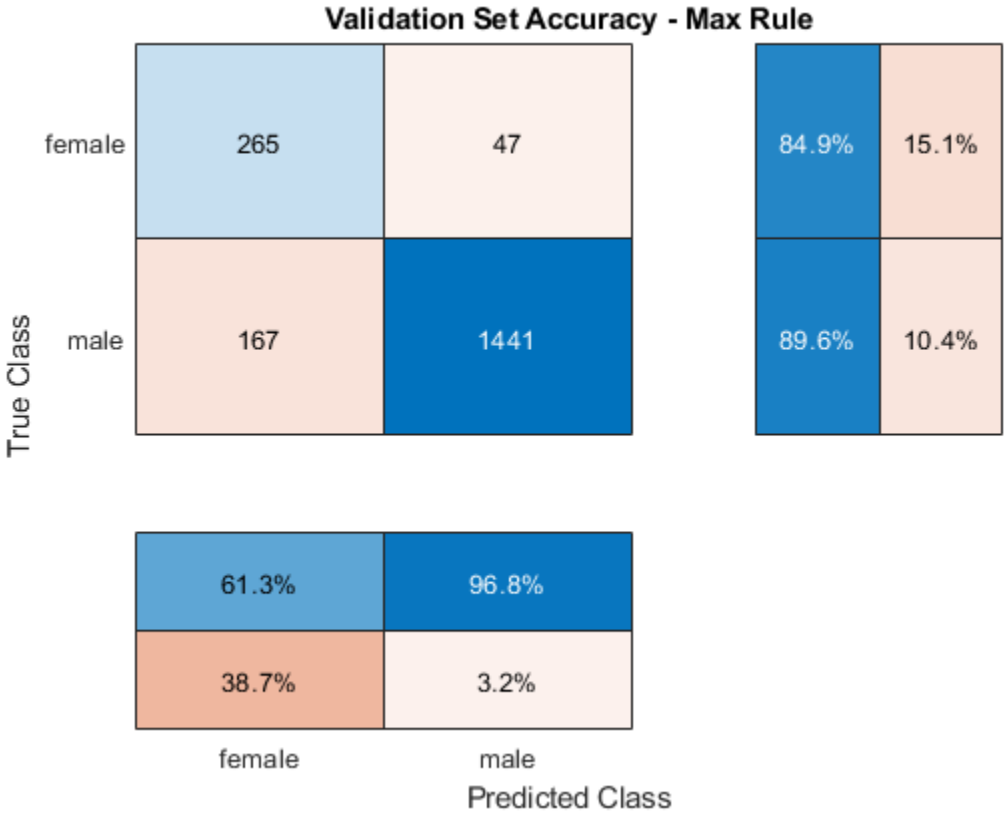
Predict the gender from each training file by considering the output classes of all sequences generated from the same file.

```
numFiles = numel(adsValidation.Files);
actualGender = categorical(adsValidation.Labels);
predictedGender = actualGender;
scores = cell(1,numFiles);
counter = 1;
cats = unique(actualGender);
for index = 1:numFiles
    scores{index} = probabilities(counter: counter + sequencePerFile(index) - 1,:);
    m = max(mean(scores{index},1),[],1);
```

```
    if m(1) >= m(2)
        predictedGender(index) = cats(1);
    else
        predictedGender(index) = cats(2);
    end
    counter = counter + sequencePerFile(index);
end
```

Visualize the confusion matrix on the majority-rule predictions.

```
figure
cm = confusionchart(actualGender,predictedGender,'title','Validation Set Accuracy - Max Rule');
cm.ColumnSummary = 'column-normalized';
cm.RowSummary = 'row-normalized';
```



References

[1] Mozilla Common Voice Dataset

Appendix - Supporting Functions

```
function [sequences,sequencePerSegment] = HelperFeatureVector2Sequence(features,featureVectorsPe
if featureVectorsPerSequence <= featureVectorOverlap
    error('The number of overlapping feature vectors must be less than the number of feature vec
end

hopLength = featureVectorsPerSequence - featureVectorOverlap;
idx1 = 1;
```

```
sequences = {};
sequencePerSegment = cell(numel(features),1);
for ii = 1:numel(features)
    sequencePerSegment{ii} = max(floor((size(features{ii},2) - featureVectorsPerSequence)/hopLeng
    idx2 = 1;
    for j = 1:sequencePerSegment{ii}
        sequences{idx1,1} = features{ii}(:,idx2:idx2 + featureVectorsPerSequence - 1); %#ok<AGROW
        idx1 = idx1 + 1;
        idx2 = idx2 + hopLength;
    end
end


end
```

See Also

`lstmLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Long Short-Term Memory Networks” on page 1-53

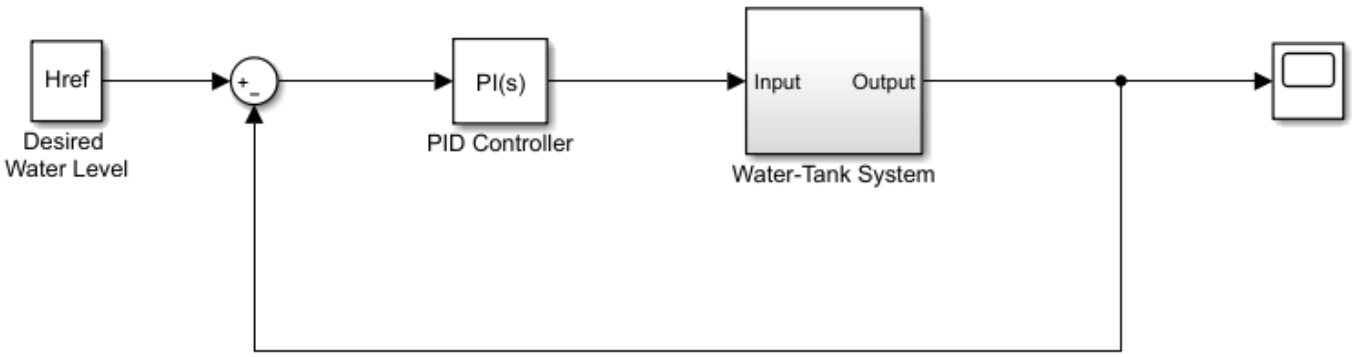
Reinforcement Learning Examples

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).

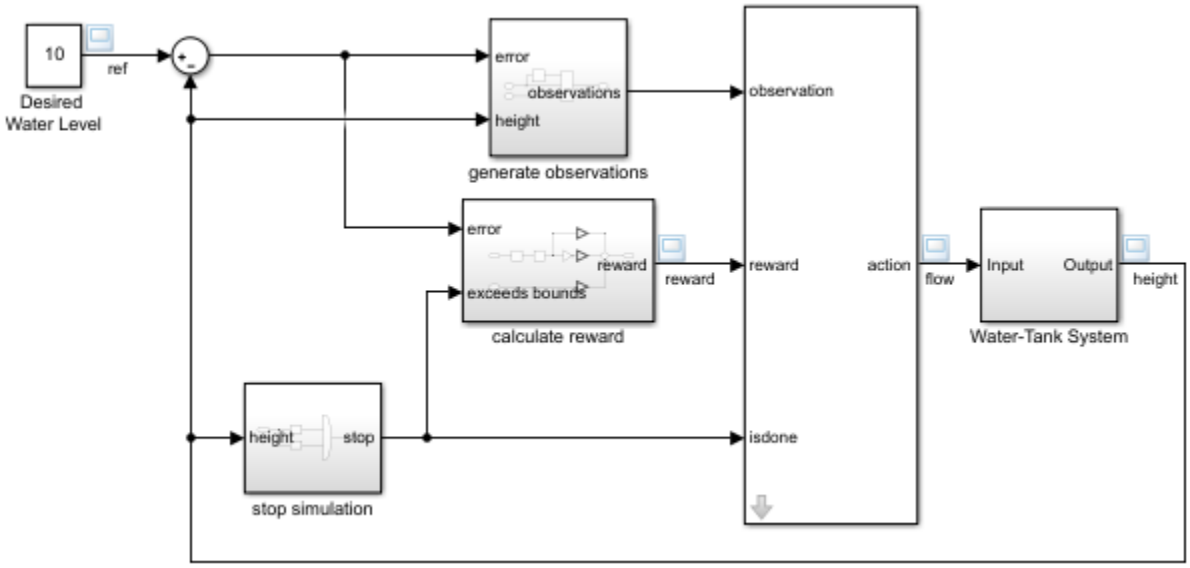


Modify the original model by making the following changes:

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 Environments for Reinforcement Learning” (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` and `rlFiniteSetSpec`.
- 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)
```

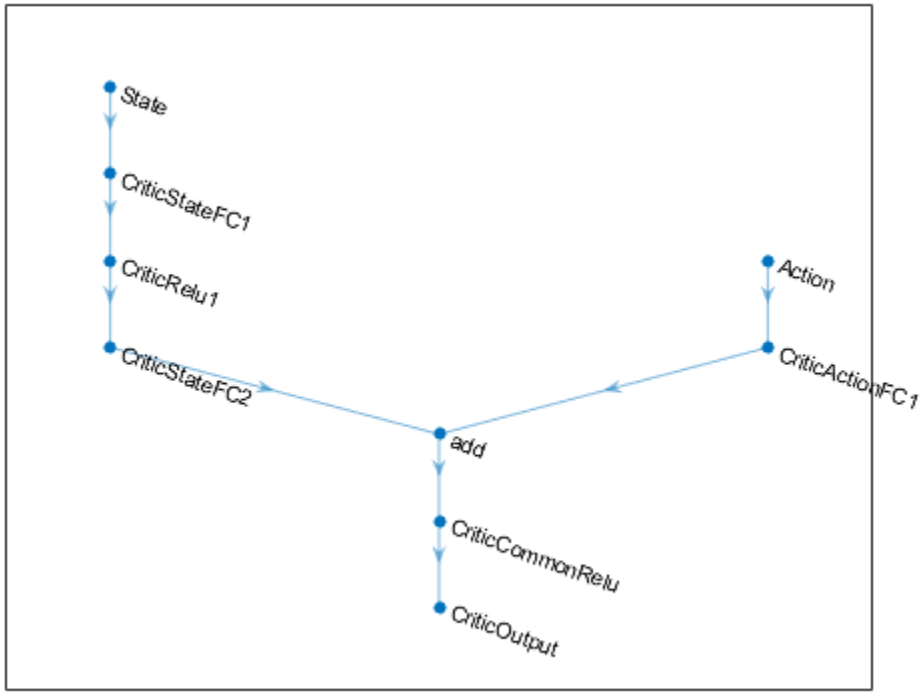
Create DDPG Agent

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 = [
    imageInputLayer([numObservations 1 1],'Normalization','none','Name','State')
    fullyConnectedLayer(50,'Name','CriticStateFC1')
    reluLayer('Name','CriticRelu1')
    fullyConnectedLayer(25,'Name','CriticStateFC2')];
actionPath = [
    imageInputLayer([numActions 1 1],'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`.

```
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`.

```
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`.

```
actorNetwork = [
    imageInputLayer([numObservations 1 1],'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`.

```
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`.

```
agent = rlDDPGAgent(actor,critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

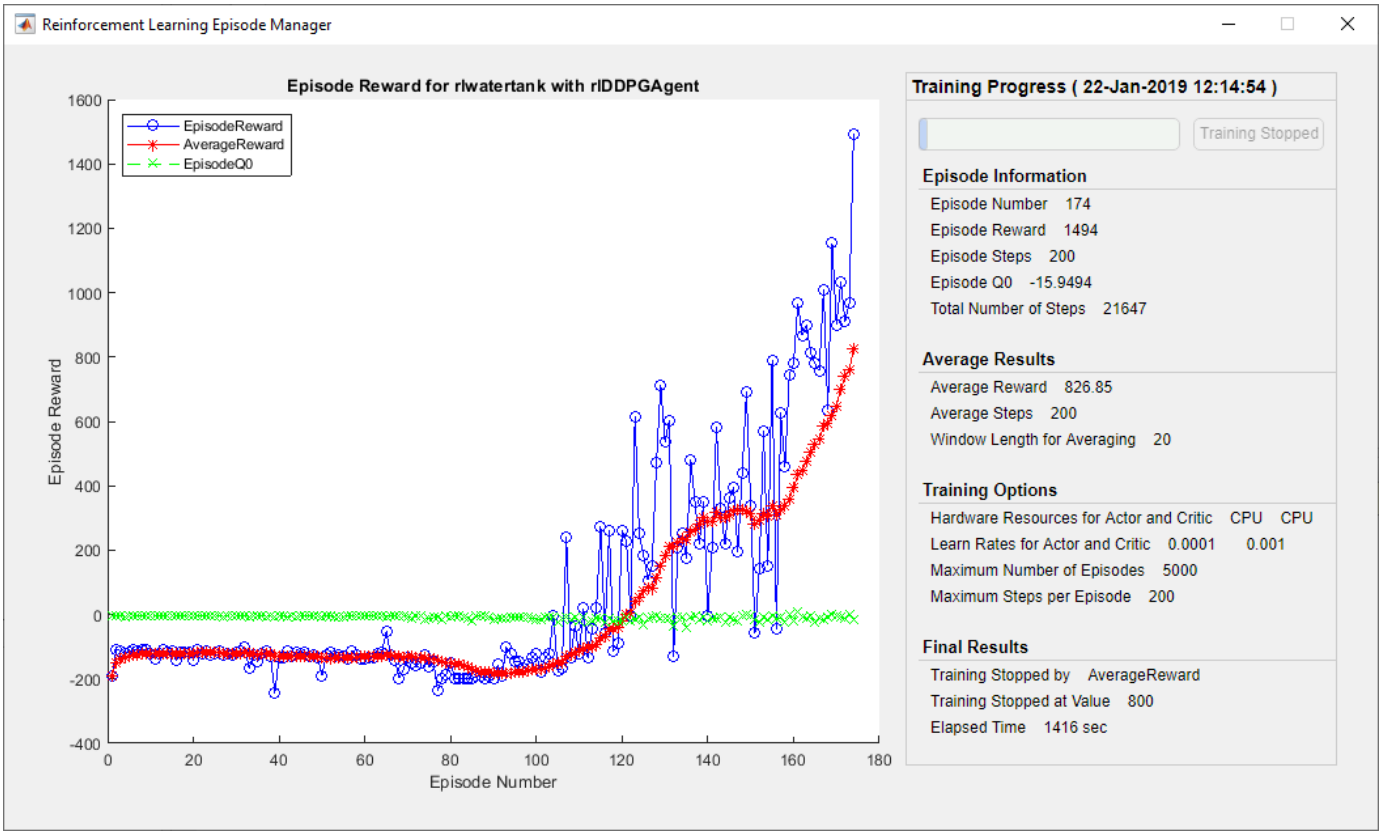
- Run each training for at most `5000` episodes. Specify that each episode lasts for at most `200` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the 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`.

```
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` 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`.

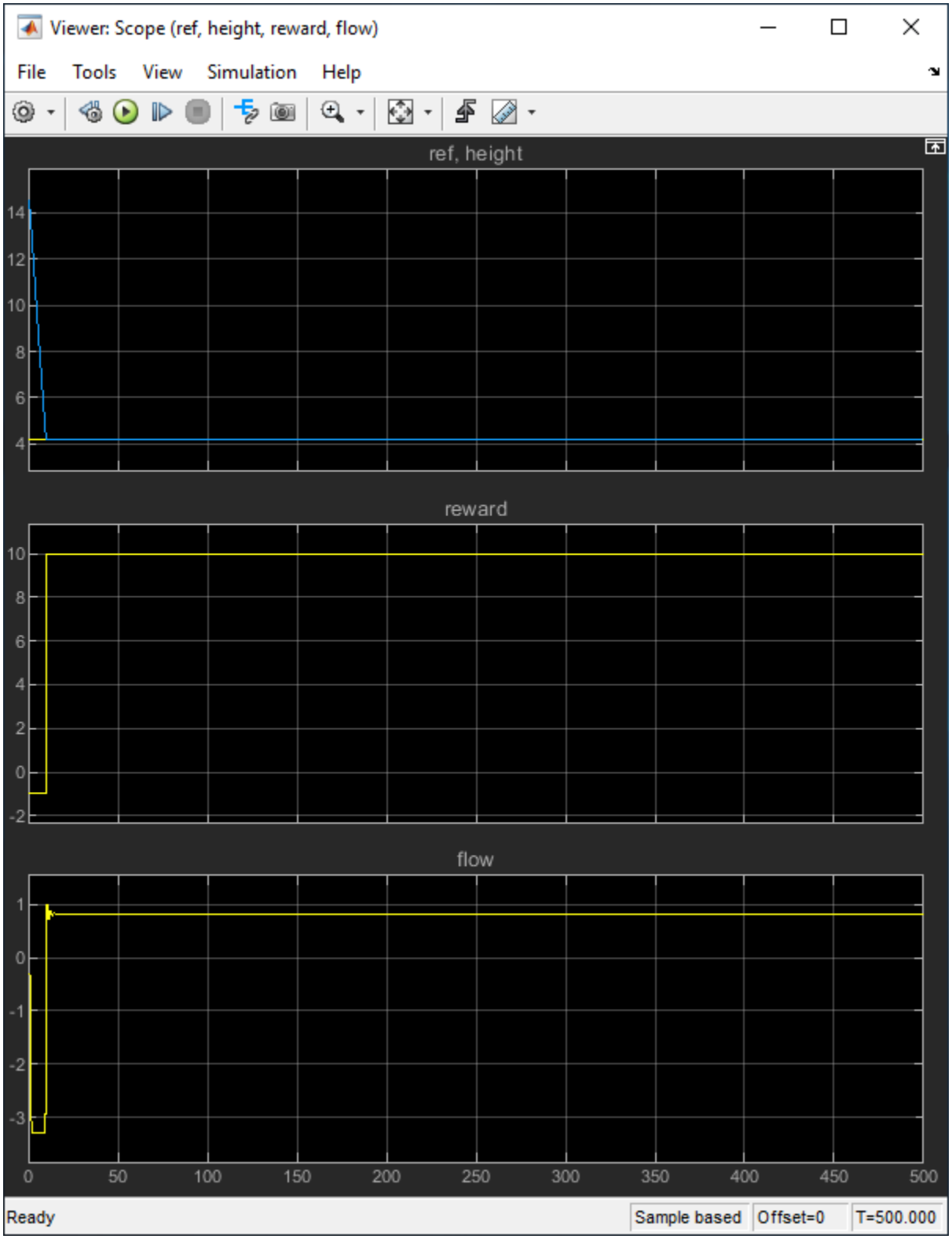
```
doTraining = false;

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 Environments for Reinforcement Learning” (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 is initially hanging 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` Nm
- The observations from the environment are an image indicating the location of the pendulum's mass and the pendulum angular velocity.
- The reward r_t, provided at every time step, is:

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

where:

- 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` Nm.

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)
```

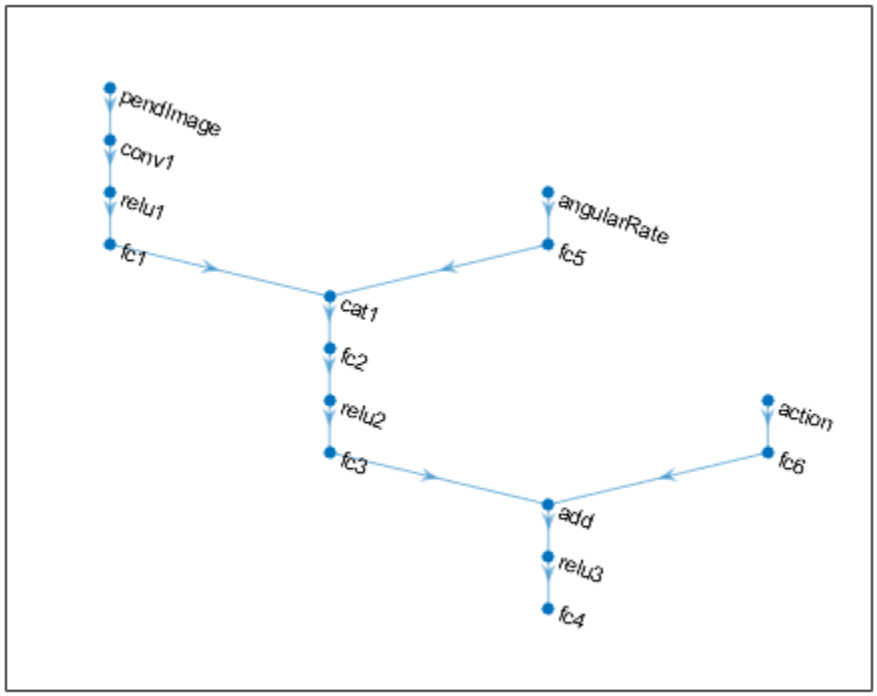
Create DDPG Agent

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`.

```
criticOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);
```

Uncomment the following line to use the GPU to accelerate training of the critic CNN.

```
% 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`.

```
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.

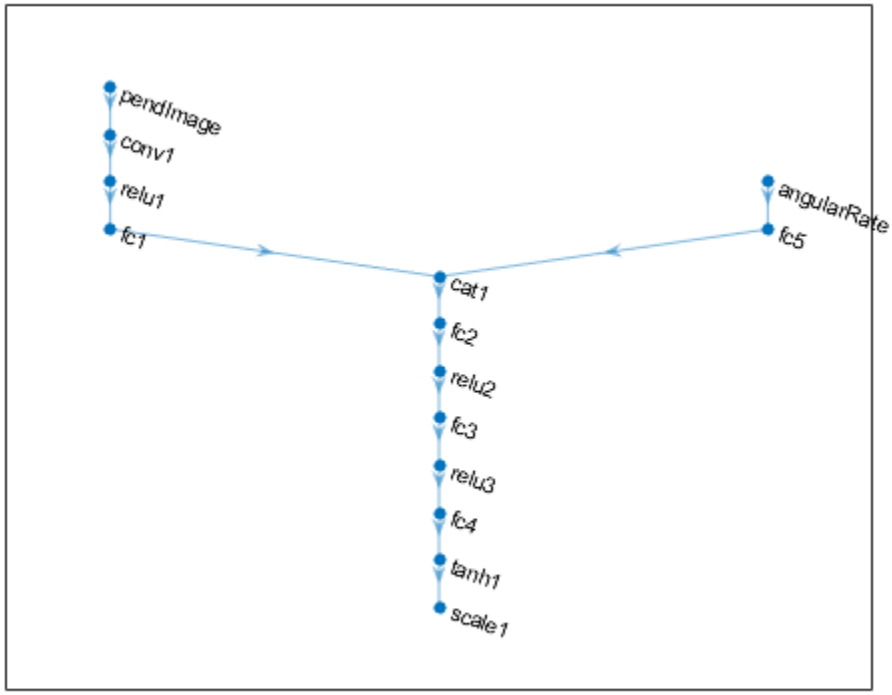
```
% actorOptions.UseDevice = 'gpu';
```

Create the actor representation using the specified neural network and options. For more information, see `rlDeterministicActorRepresentation`.

```
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`.

```
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`.

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

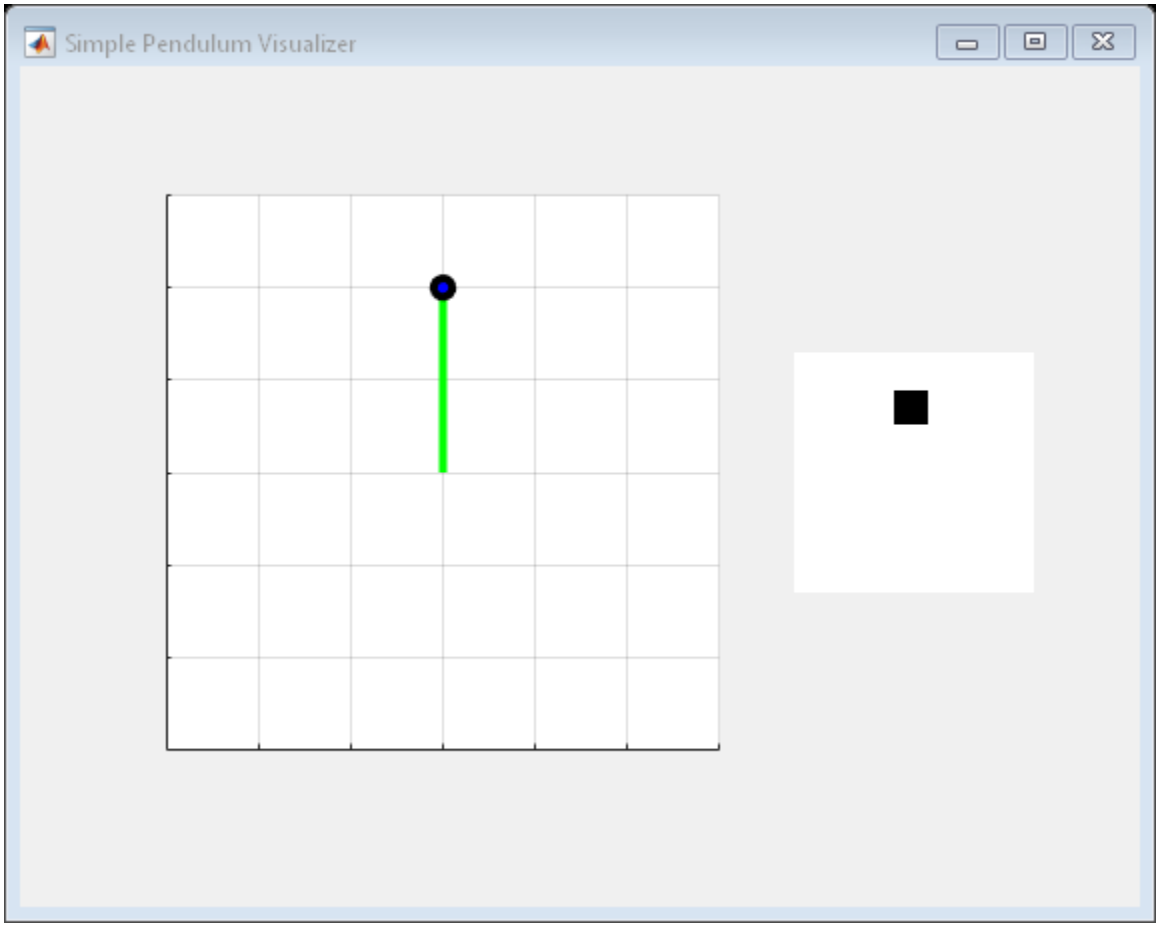
- Run each training for at most `5000` episodes, with each episode lasting at most `400` time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option).
- Stop training when the agent receives a moving average cumulative reward greater than `-740` (over ten consecutive episodes)

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = 400;
trainingOptions = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-740);
```

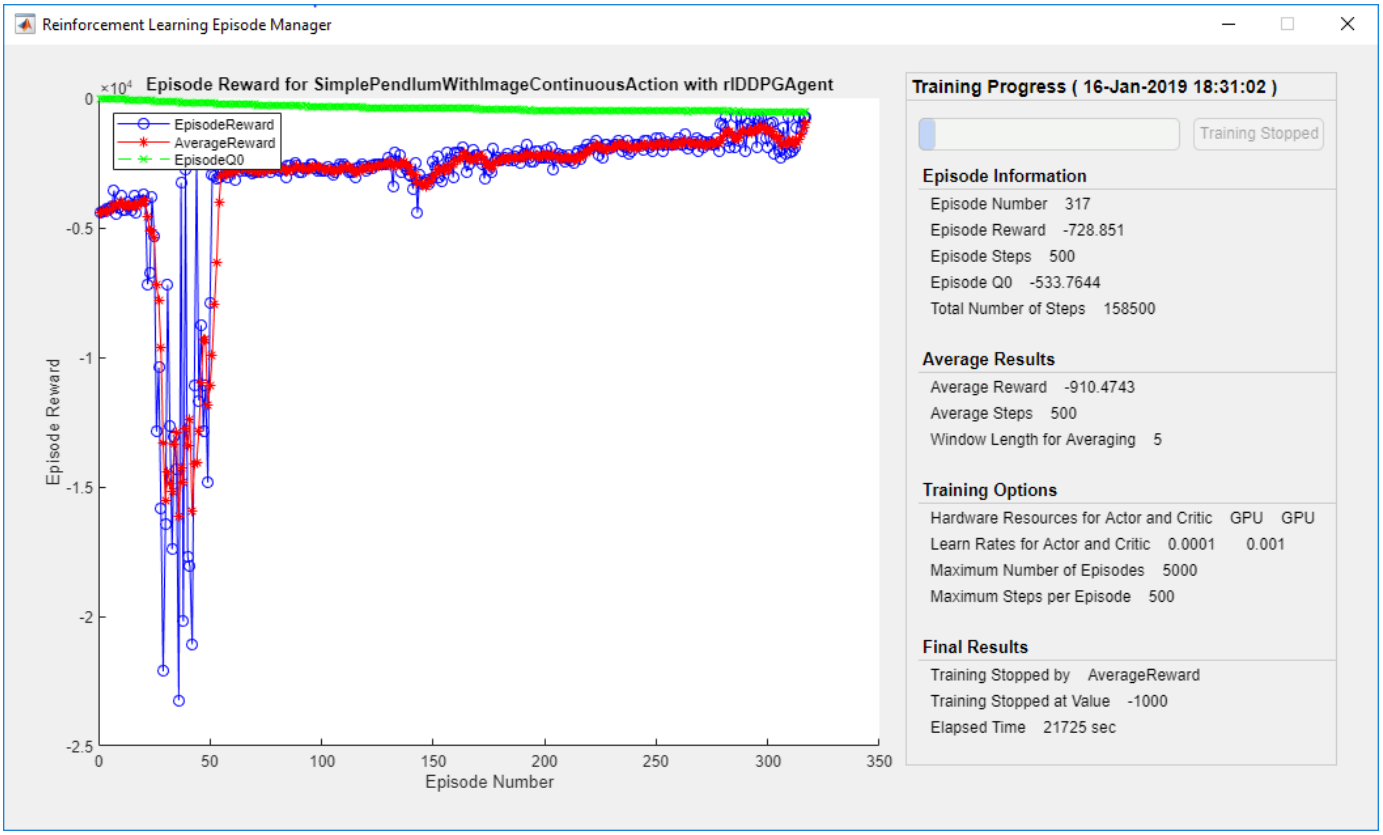
The pendulum system can be visualized with `plot` during training or simulation.

```
plot(env)
```



Train the agent using the `train` 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`.

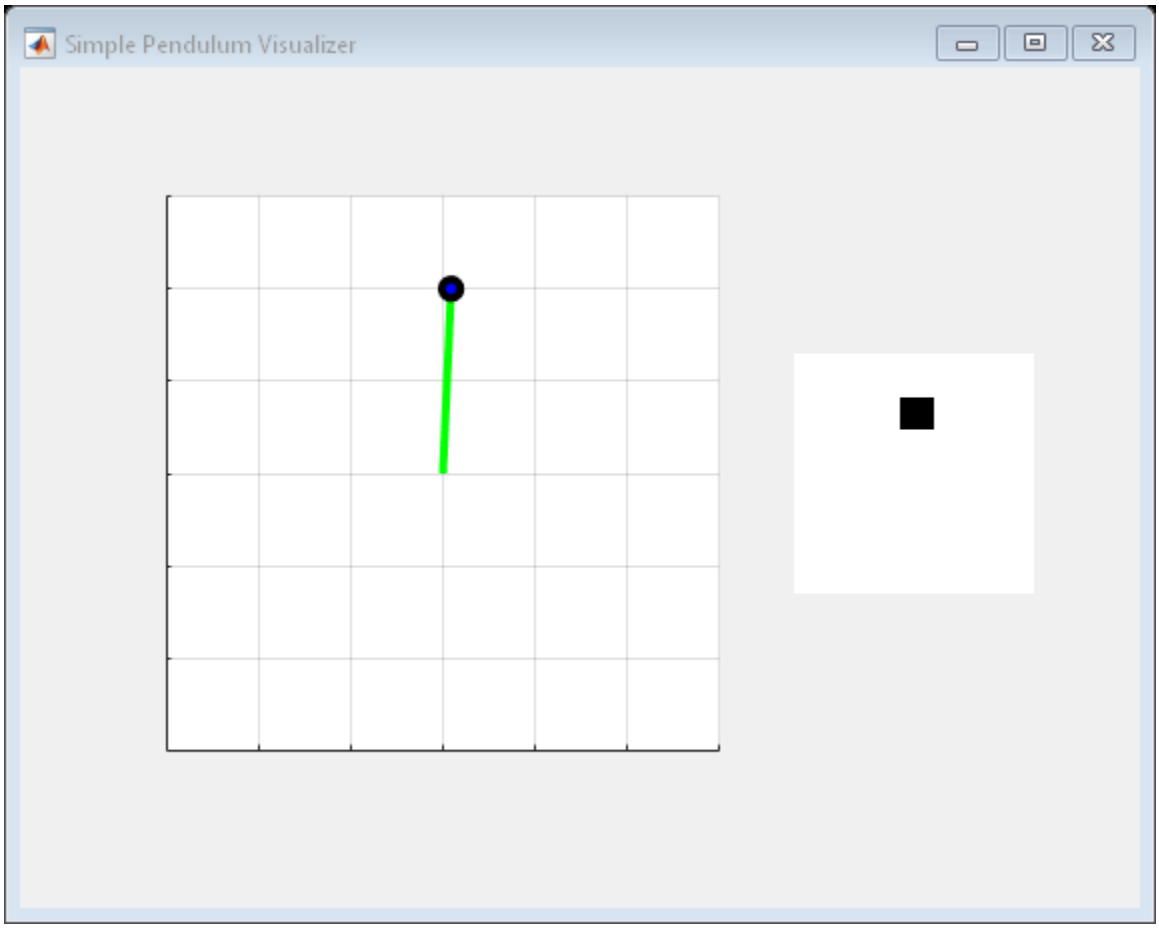
```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainingOptions);
else
    % Load pretrained agent for the example.
    load('SimplePendulumWithImageDDPG.mat','agent')
end
```



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` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',500);
experience = sim(env,agent,simOptions);
```



See Also

`train`

More About

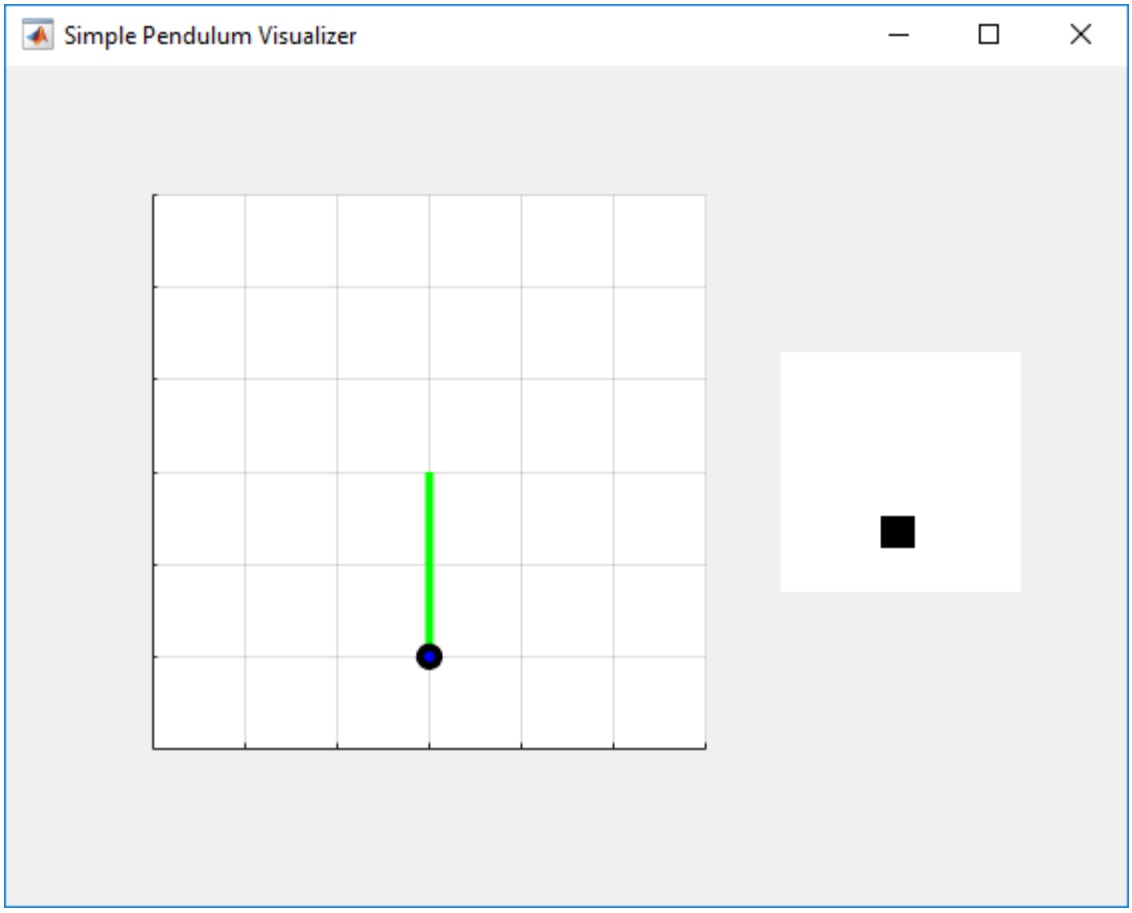
- “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox)
- “Train Reinforcement Learning Agents” (Reinforcement Learning Toolbox)
- “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox)

Create Agent Using Deep Network Designer and Train Using Image Observations

This example shows how to create a deep Q-learning network (DQN) agent using the Deep Network Designer app to swing up and balance a pendulum modeled in MATLAB®. 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 is initially hanging 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` Nm.
- 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:

$$r_t = -\left(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001u_{t-1}^2\right)$$

where:

- 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 50x50 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` Nm.

```
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)
```

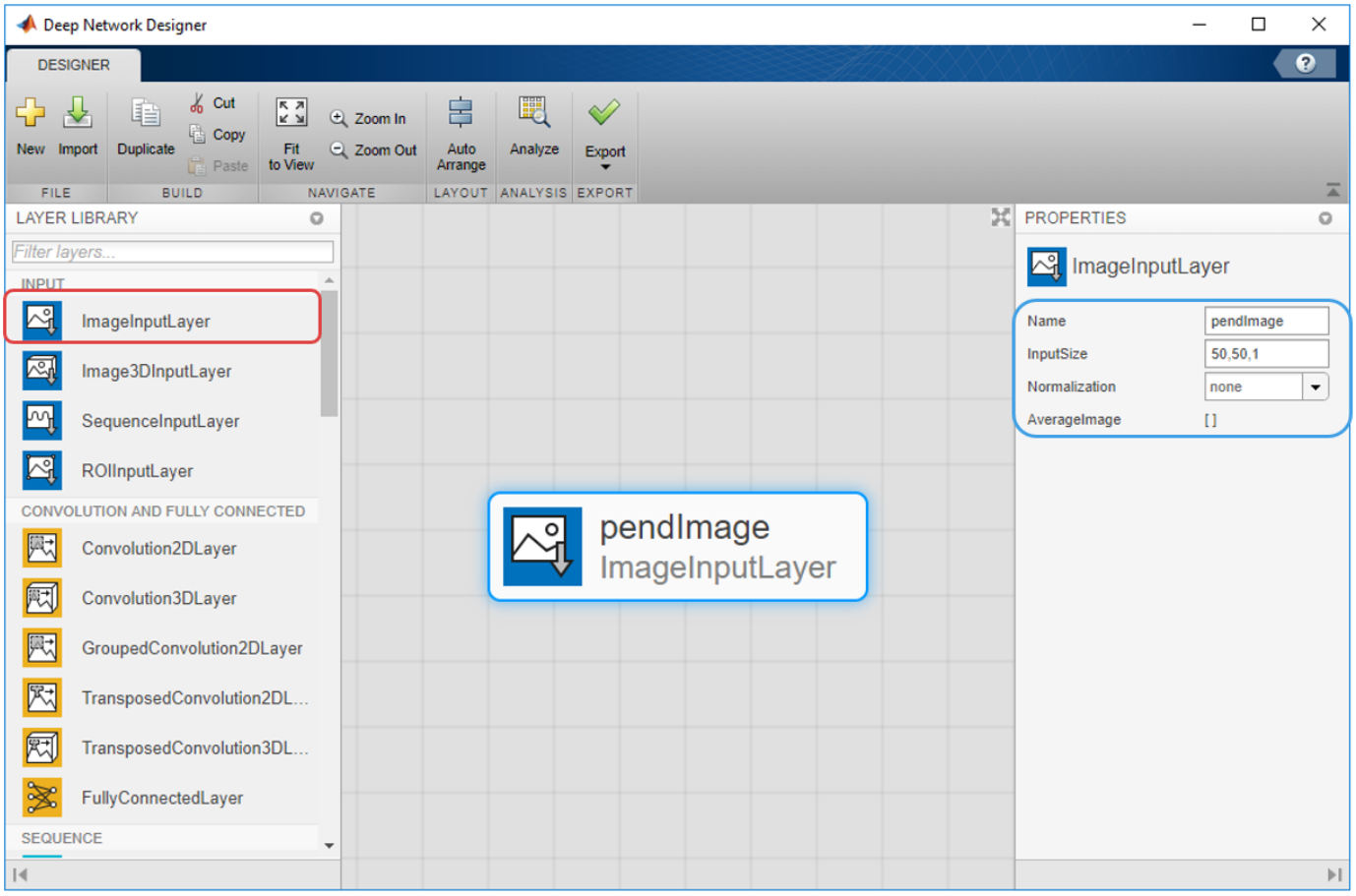
Construct Critic Network Using Deep Network Designer

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, the two observations 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).

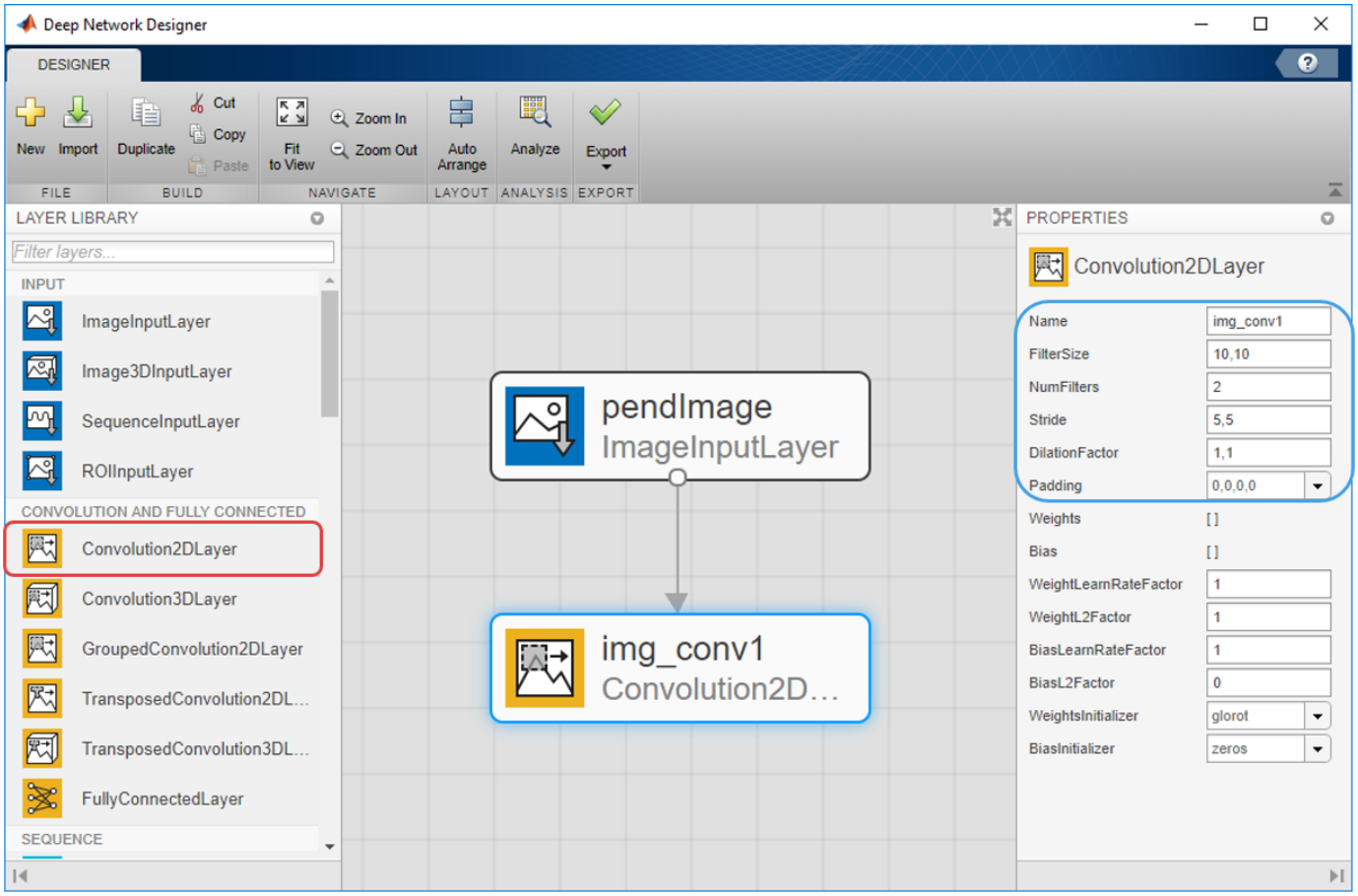
You can interactively construct the critic network using the Deep Network Designer app. To do so, you first create separate input paths for each observation and action. These paths learn lower-level features from their respective inputs. You then create a common output path which combines the outputs from the input paths.

Create Image Observation Path

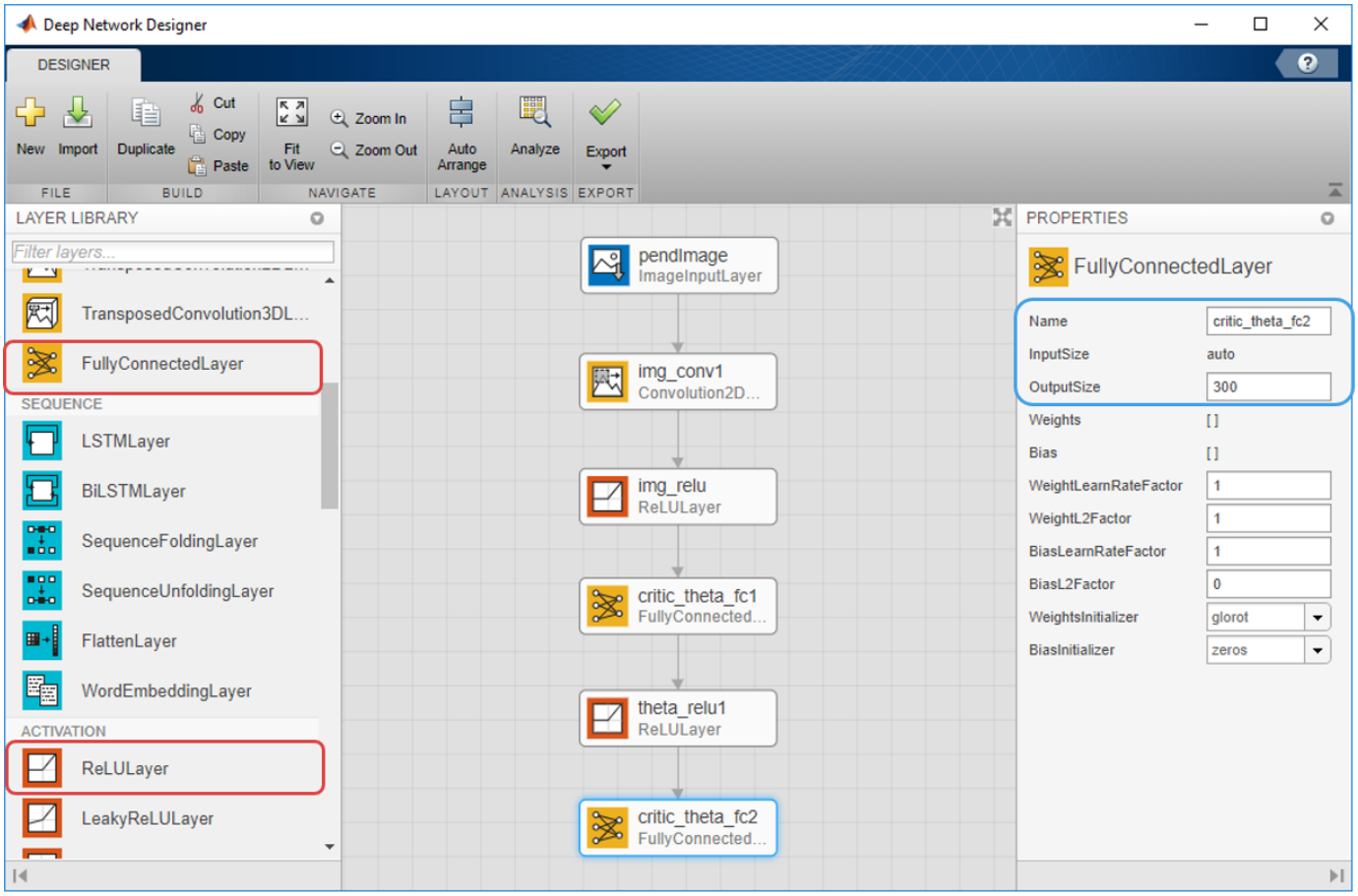
To create the image observation path, first drag an `ImageInputLayer` from the **Layer Library** pane to the canvas. Set the layer **InputSize** to `50,50,1` for the image observation, and set **Normalization** to `none`.



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`. The **OutputSize** of the two `FullyConnectedLayer` layers are 400 and 300, respectively.



Create All Input Paths and Output Path

Construct the other input paths and output path in similar fashion. For this example, use the following options:

Angular Velocity Path (scalar input):

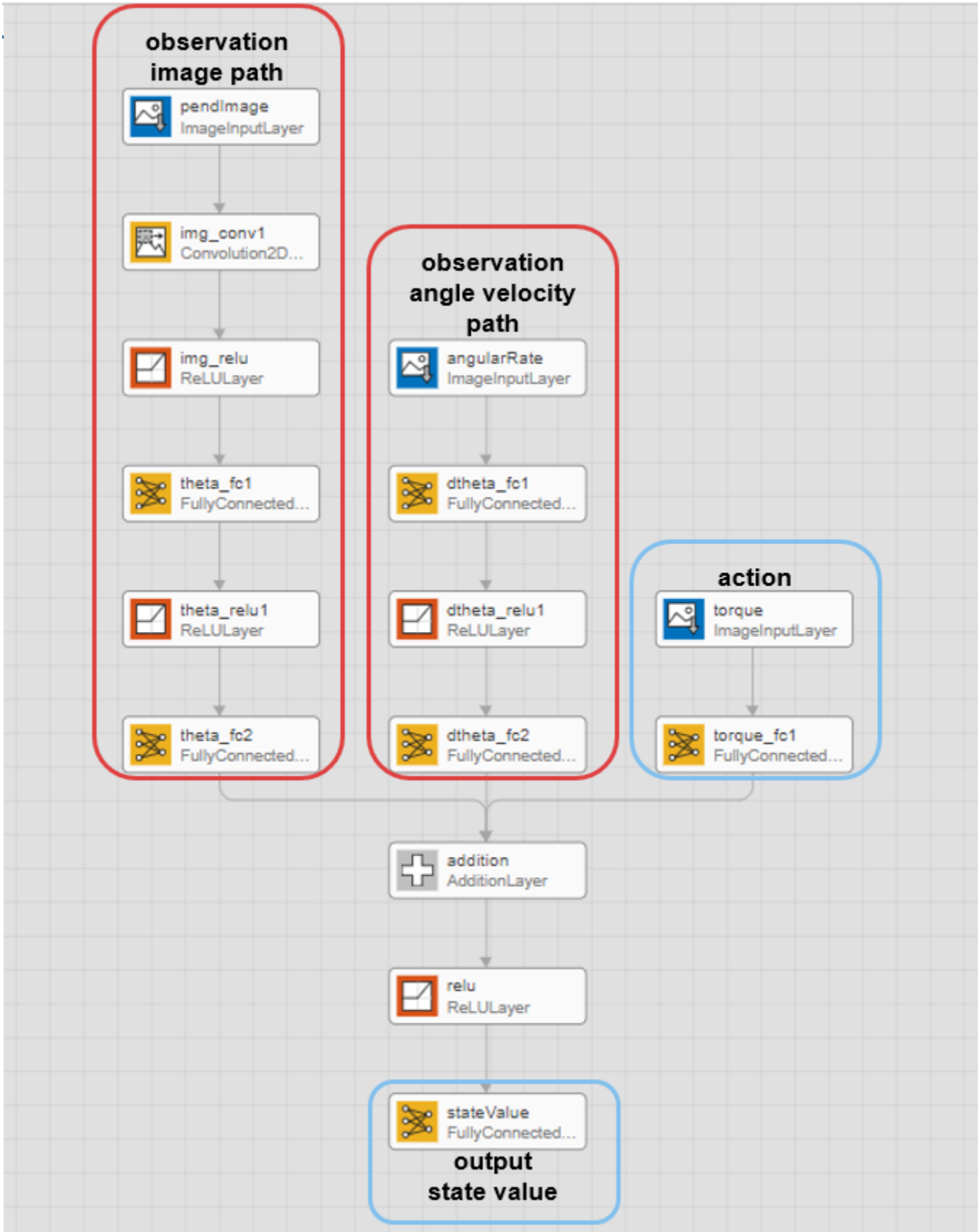
- `ImageInputLayer`: **InputSize** = `1,1` and **Normalization** = `none`
- `FullyConnectedLayer`: **OutputSize** = `400`
- `ReLULayer`
- `FullyConnectedLayer`: **OutputSize** = `300`

Action Path (scalar input):

- `ImageInputLayer`: **InputSize** = `1,1` and **Normalization** = `none`
- `FullyConnectedLayer`: **OutputSize** = `300`

Output Path:

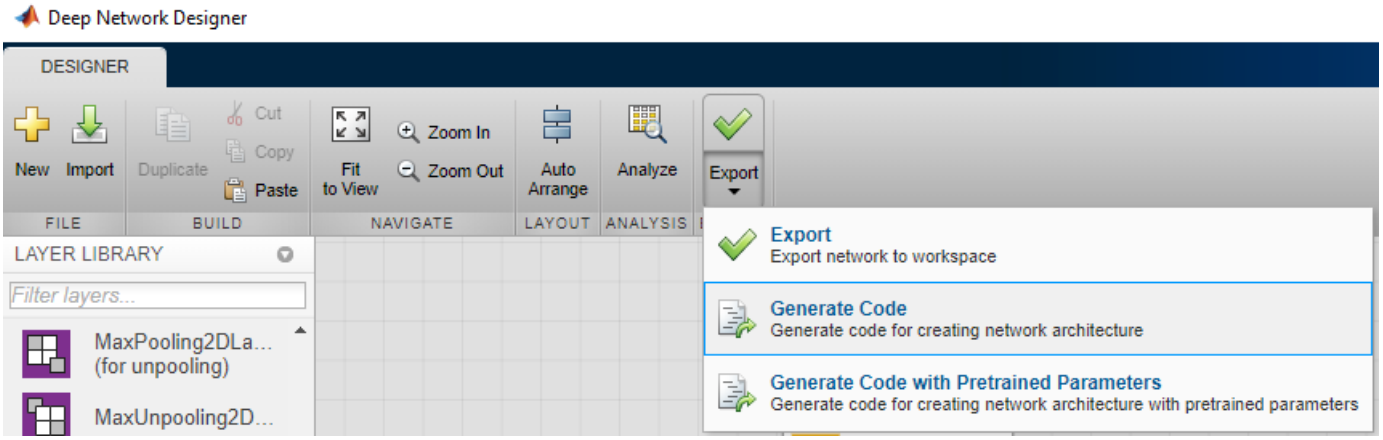
- `AdditionLayer`: Connect the output of all input paths to the input of this layer.
- `ReLULayer`
- `FullyConnectedLayer`: **OutputSize** = `1` for the scalar value function.


observation image path
pendImage ImageInputLayer
img_conv1 Convolution2D...
img_relu ReLULayer
theta_fc1 FullyConnected...
theta_relu1 ReLULayer
theta_fc2 FullyConnected...
observation angle velocity path
angularRate ImageInputLayer
dtheta_fc1 FullyConnected...
dtheta_relu1 ReLULayer
dtheta_fc2 FullyConnected...
action
torque ImageInputLayer
torque_fc1 FullyConnected...
addition AdditionLayer
relu ReLULayer
stateValue FullyConnected...
output state value

Export Network from Deep Network Designer

To export the network to the MATLAB workspace, in the **Deep Network Designer**, click **Export**. The **Deep Network Designer** exports the network to 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**.

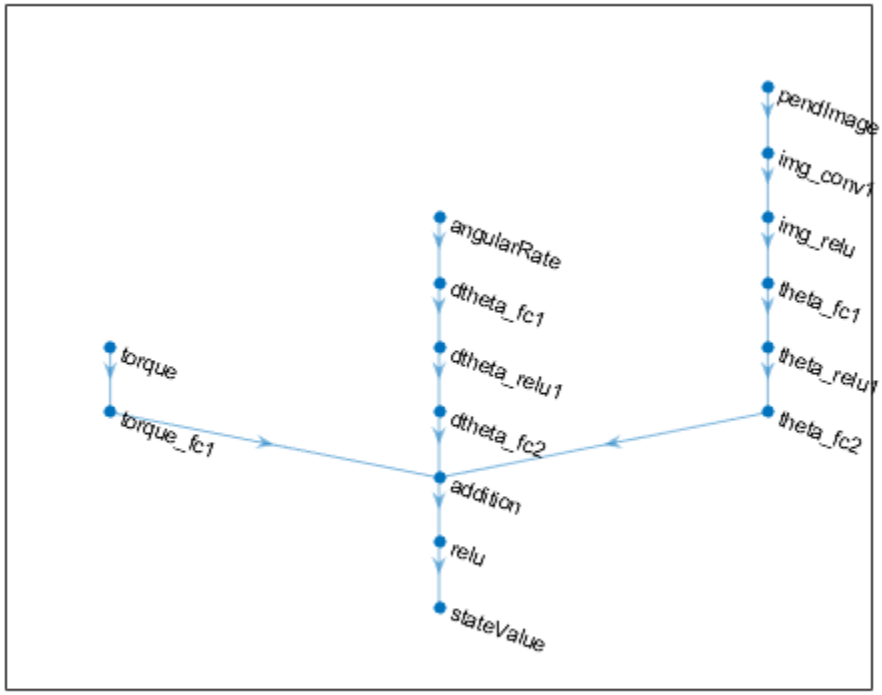


The generated code is:

```
lgraph = layerGraph();
layers = [
    imageInputLayer([1 1 1],"Name","torque","Normalization","none")
    fullyConnectedLayer(300,"Name","torque_fc1")];
lgraph = addLayers(lgraph,layers);
layers = [
    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,layers);
layers = [
    imageInputLayer([50 50 1],"Name","pendImage","Normalization","none")
    convolution2dLayer([10 10],2,"Name","img_conv1","Stride",[5 5])
    reluLayer("Name","img_relu")
    fullyConnectedLayer(400,"Name","theta_fc1")
    reluLayer("Name","theta_relu1")
    fullyConnectedLayer(300,"Name","theta_fc2")];
lgraph = addLayers(lgraph,layers);
layers = [
    additionLayer(3,"Name","addition")
    reluLayer("Name","relu")
    fullyConnectedLayer(1,"Name","stateValue")];
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,"torque_fc1","addition/in3");
lgraph = connectLayers(lgraph,"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`.

```
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`.

```
critic = rlQValueRepresentation(lgraph,obsInfo,actInfo,...
    'Observation',{'pendImage','angularRate'},'Action',{'torque'},criticOpts);
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
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`.

```
agent = rlDQNAgent(critic,agentOpts);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

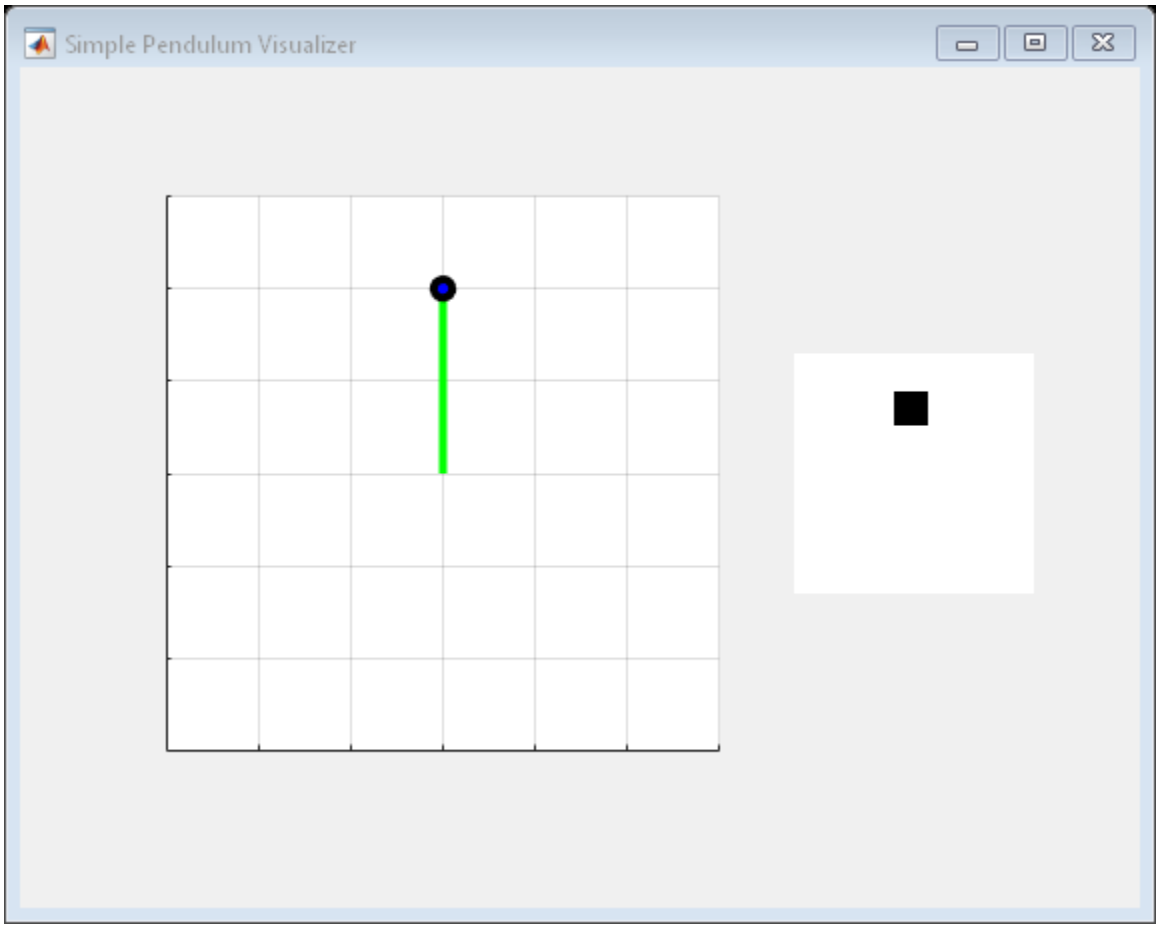
- Run each training for at most 1000 episodes, with each episode lasting at most 500 time steps.
- Display the training progress in the Episode Manager dialog box (set the `Plots` option) and disable the command line display (set the `Verbose` option to `false`).
- Stop training when the agent receives an average cumulative reward greater than `-1000` over 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`.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',5000,...
    'MaxStepsPerEpisode',500,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','AverageReward',...
    'StopTrainingValue',-1000);
```

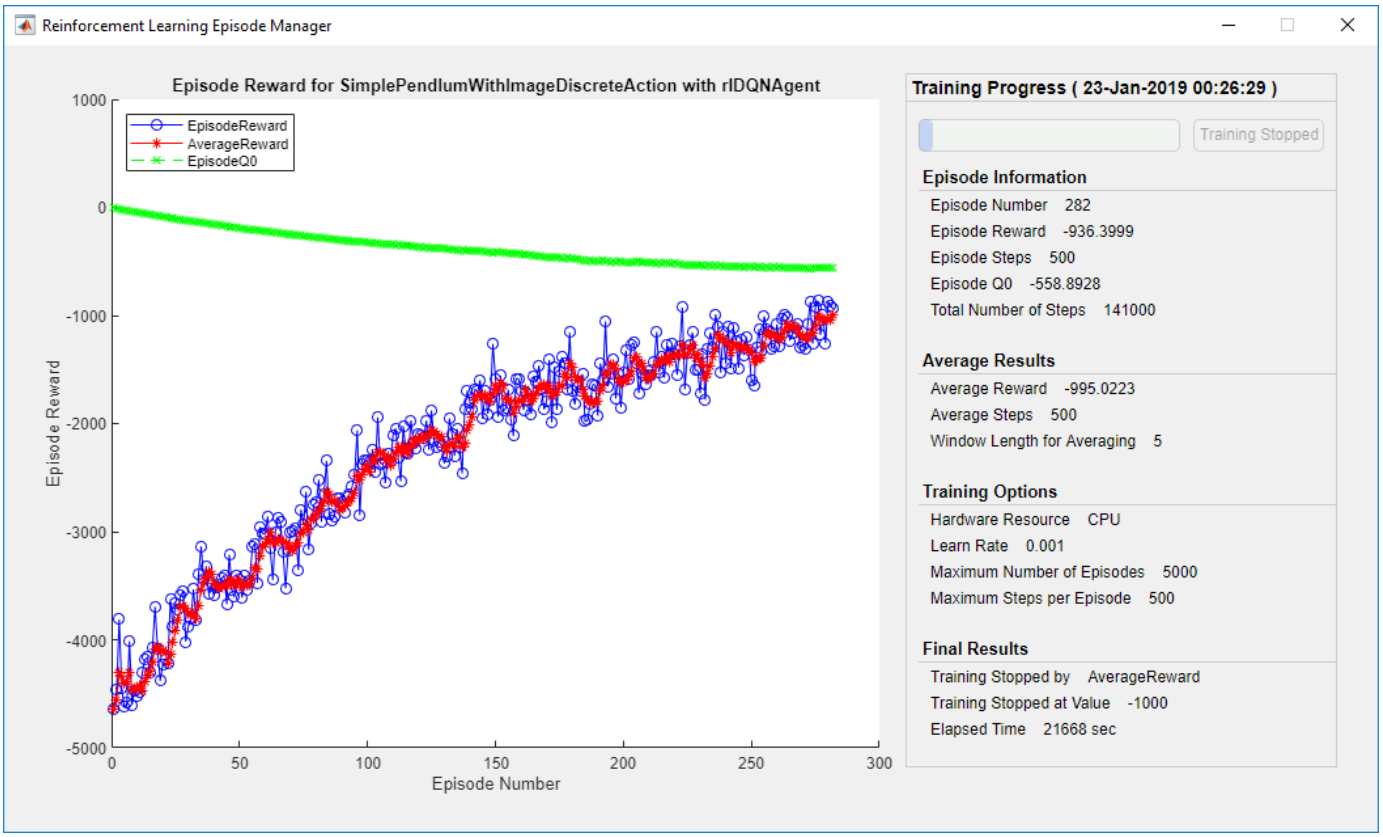
The pendulum system can be visualized with `plot(env)` during training or simulation.

```
plot(env)
```



Train the agent using the `train` 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`.

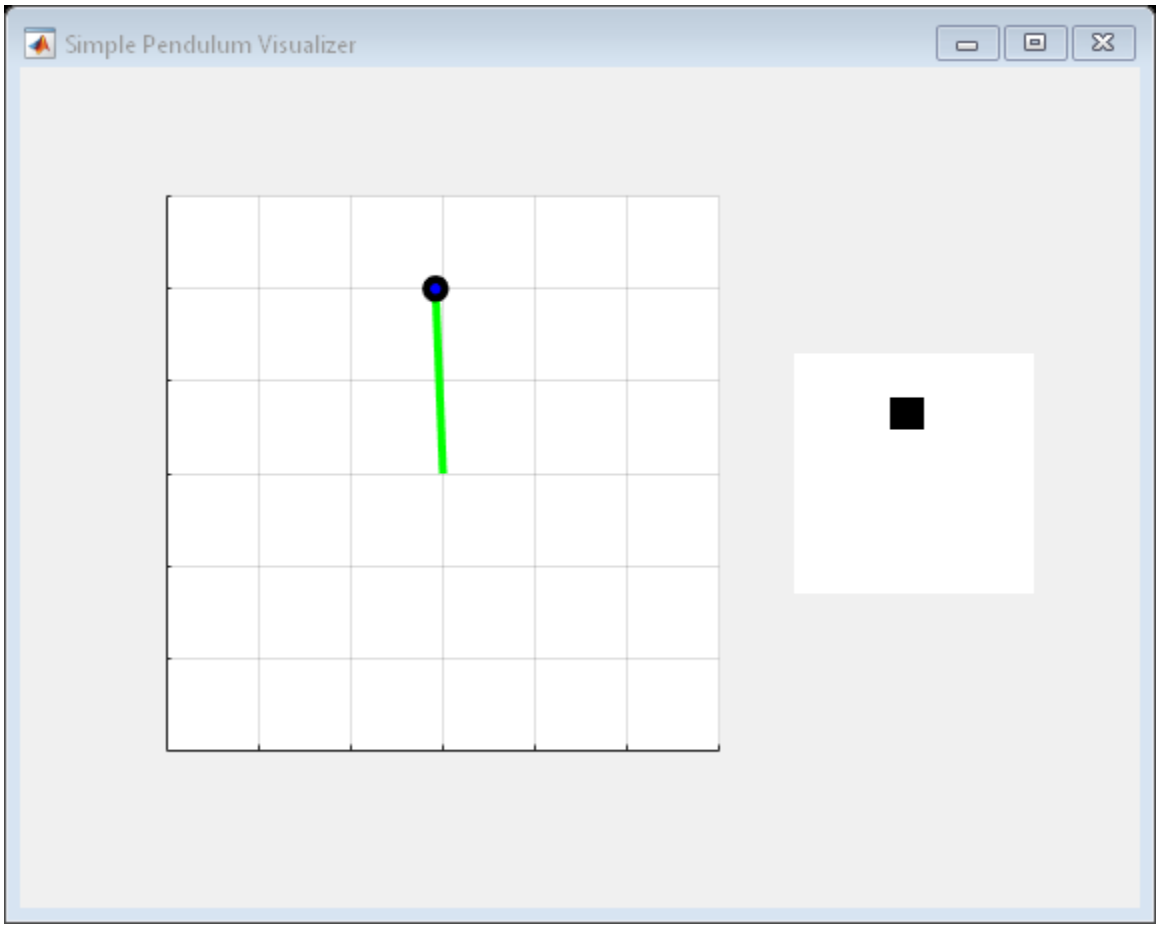
```
doTraining = false;

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` and `sim`.

```
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)

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 where the orientation of the robot is also randomized. The robot has two thrusters mounted on the side of the body which 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 and setup initial model variables.

```
mdl = 'rlFlyingRobotEnv';
open_system(mdl)

% initial model state variables
theta0 = 0;
x0 = -15;
y0 = 0;

% sample time
Ts = 0.4;

% simulation length
Tf = 30;
```

For this model:

- The goal orientation is `0` radians (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 with `rlSimulinkEnv` with the generated 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. See `flyingRobotResetFcn` for details of the reset function.

```
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 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 = [
    imageInputLayer([numObs 1 1],'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 = [
    imageInputLayer([numAct 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSize,'Name','fc5')];

% create the layerGraph
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);

% connect actionPath to obervationPath
criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

Specify options for the critic using `rlRepresentationOptions`.

```
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`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'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 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`.

```
actorNetwork = [
    imageInputLayer([numObs 1 1],'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`.

```
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`.

```
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 ten 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`.

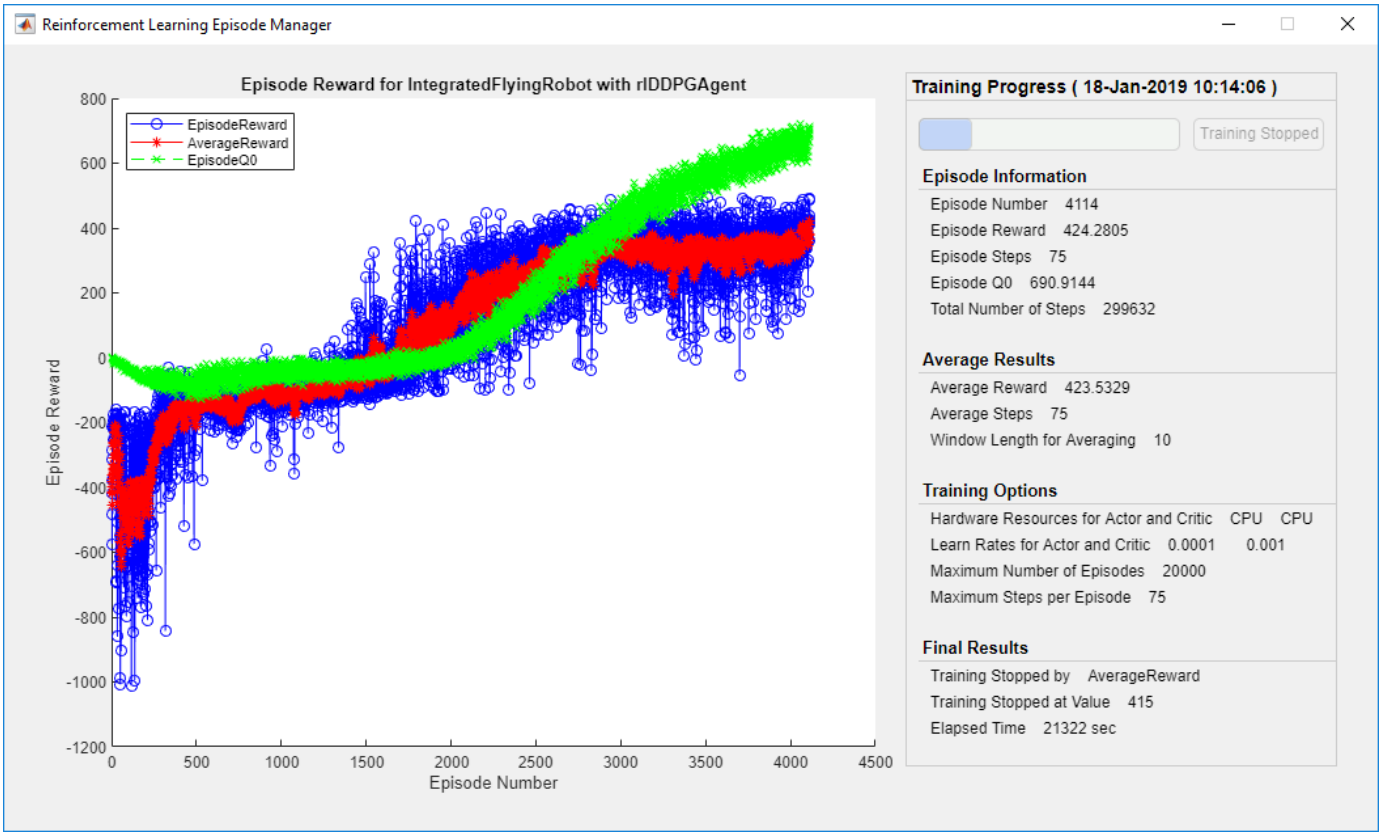
For more information, see `rlTrainingOptions`.

```
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` 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,trainingOptions);
else
```

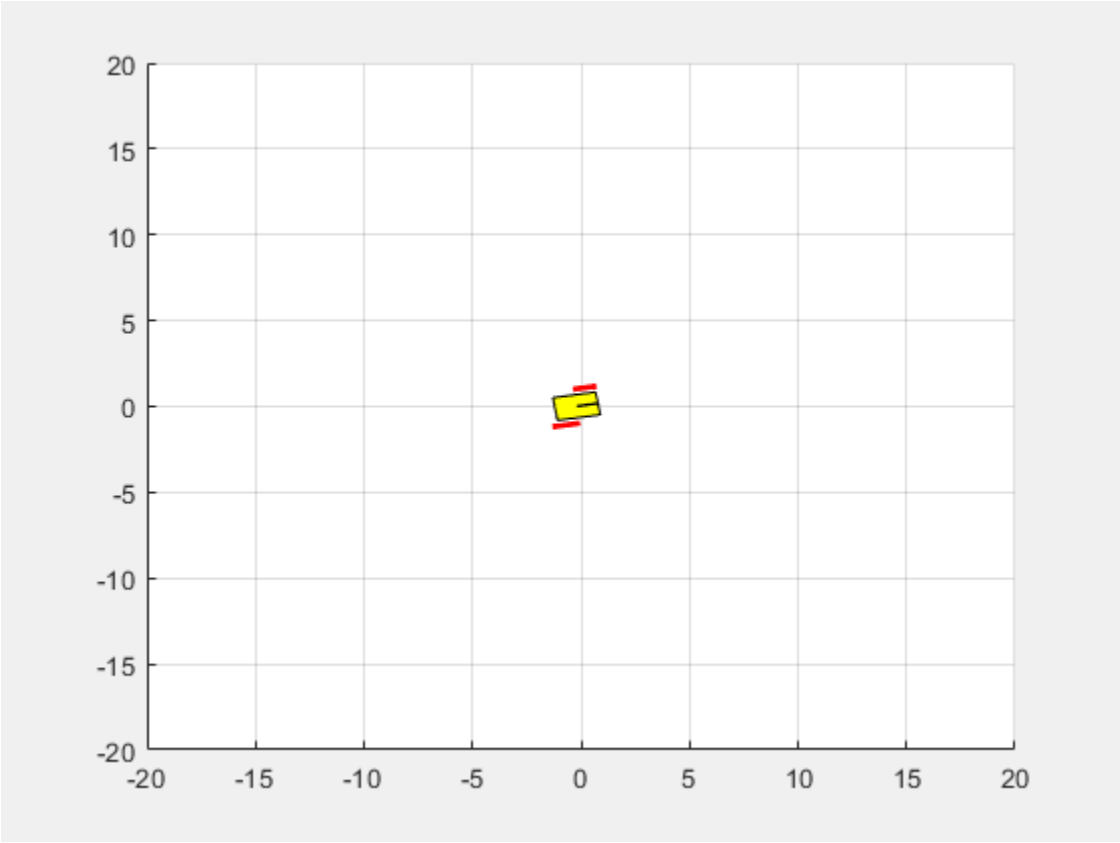
```
    % Load pretrained agent for the example.
    load('FlyingRobotDDPG.mat','agent')
end
```



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` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',maxsteps);
experience = sim(env,agent,simOptions);
```



See Also

`rlDDPGAgent` | `train`

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, modeled in Simscape™ Multibody™, to walk using both a deep deterministic policy gradient (DDPG) agent and a Twin-Delayed deep deterministic policy gradient (TD3) agent and compares the performance of these trained agents.

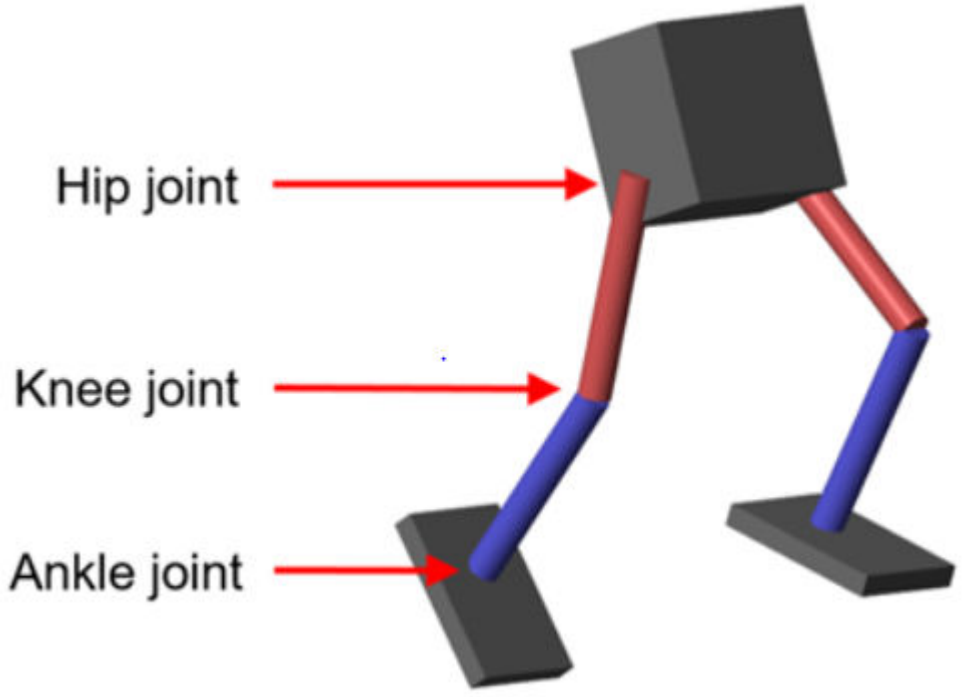
For more information on these agents, see “Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox) and “Twin-Delayed Deep Deterministic Policy Gradient Agents” (Reinforcement Learning Toolbox).

For the purpose of comparison in this example, both agents are trained on the biped robot environment with same model parameters. The agents are also configured to have the following settings in common.

- Initial condition strategy of the biped robot
- Network structure of actor and critic, inspired by [2]
- Options for actor and critic representations
- Training options (sample time, discount factor, mini-batch size, experience buffer length, exploartion 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.

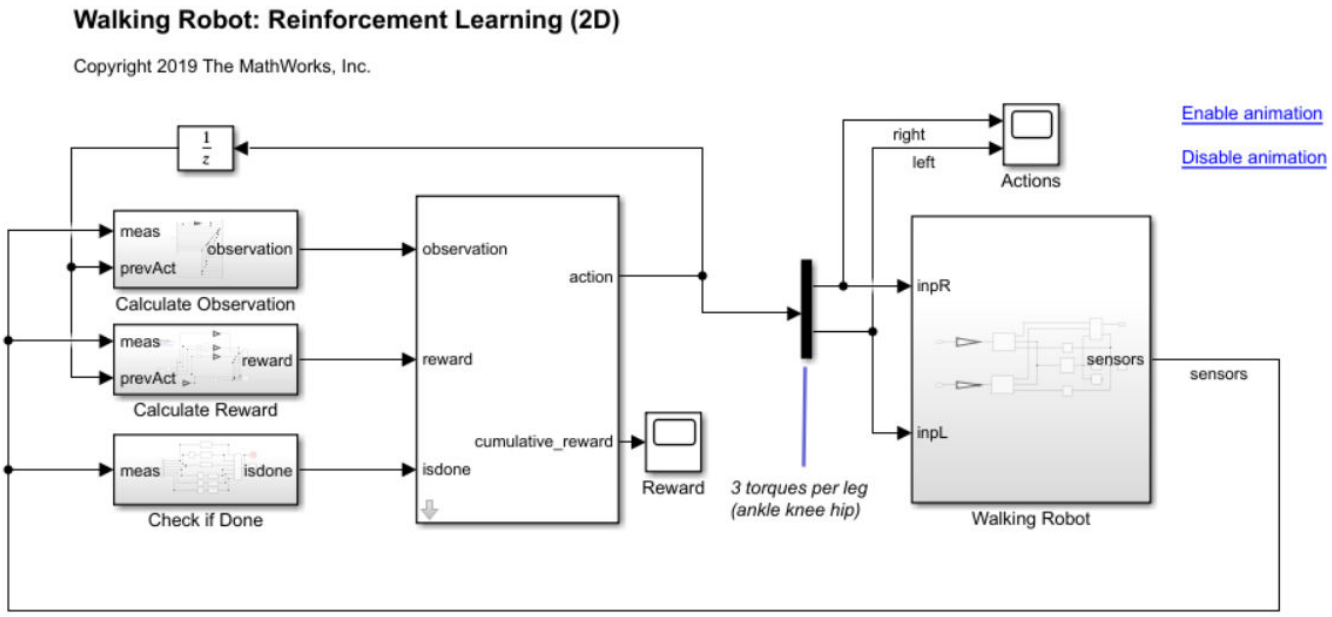


Load parameters of the model to the MATLAB® workspace.

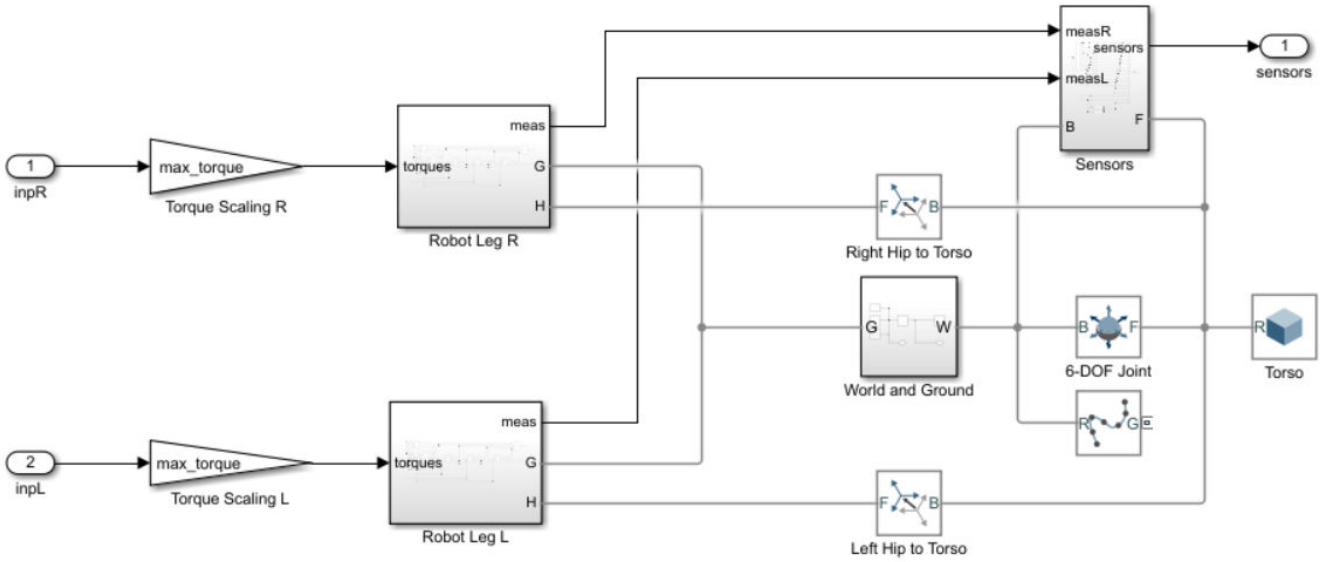
```
robotParametersRL
```

Open the Simulink model.

```
mdl = 'rlWalkingBipedRobot';
open_system(mdl)
```



The robot is modeled using Simscape Multibody.



For this model:

- The neutral `0` radian position is with both legs straight and the ankles flat.
- The foot contact is modeled using the Spatial Contact Force block from Simscape Multibody.

- The agent can control 3 individual joints, the ankle, knee, and hip, on both legs of the robot by applying torque signals from -3 to 3 Nm. The actual computed action signals are normalized between -1 and 1.

The environment provides 29 observations to the agent. The observations are:

- 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 3 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 Z direction (fallen) or more than 1 m in Y direction (lateral motion).
- The absolute value of either the roll, pitch, or yaw is greater than 0.7854 radians.

The reward function r_t, which is provided at every time step, is inspired by [1]. 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.

$$\boldsymbol{r_t} = v_x - 3y^2 - 50\hat{z}^2 + 25\frac{\mathrm{Ts}}{\mathrm{Tf}} - 0.02\sum_i {u_{t-1}^i}^2$$

Here:

- v_x is the translation velocity in 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.

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 performt he 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 using an actor representation. The actor and critic networks for this example are inspired by [2].

For details on the creating the DDPG agent, see the `createDDPGAgent` helper function. For information on configuring DDPG agent options, see `rlDDPGAgentOptions`.

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

TD3 agent approximates the long-term reward given observations and actions using 2 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 this Q value is then used to update the policy (actor) of the agent, 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 — Clipped noise is added to target actions to smooth 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 it is recommended to delay the actor network update, as it 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 values estimates and results in a more high 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`.

Specify Training Options and Train Agent

For this example, the training options for the DDPG and TD3 agents are the same. These options are based on the following requirements.

- 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`.

```
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);
```

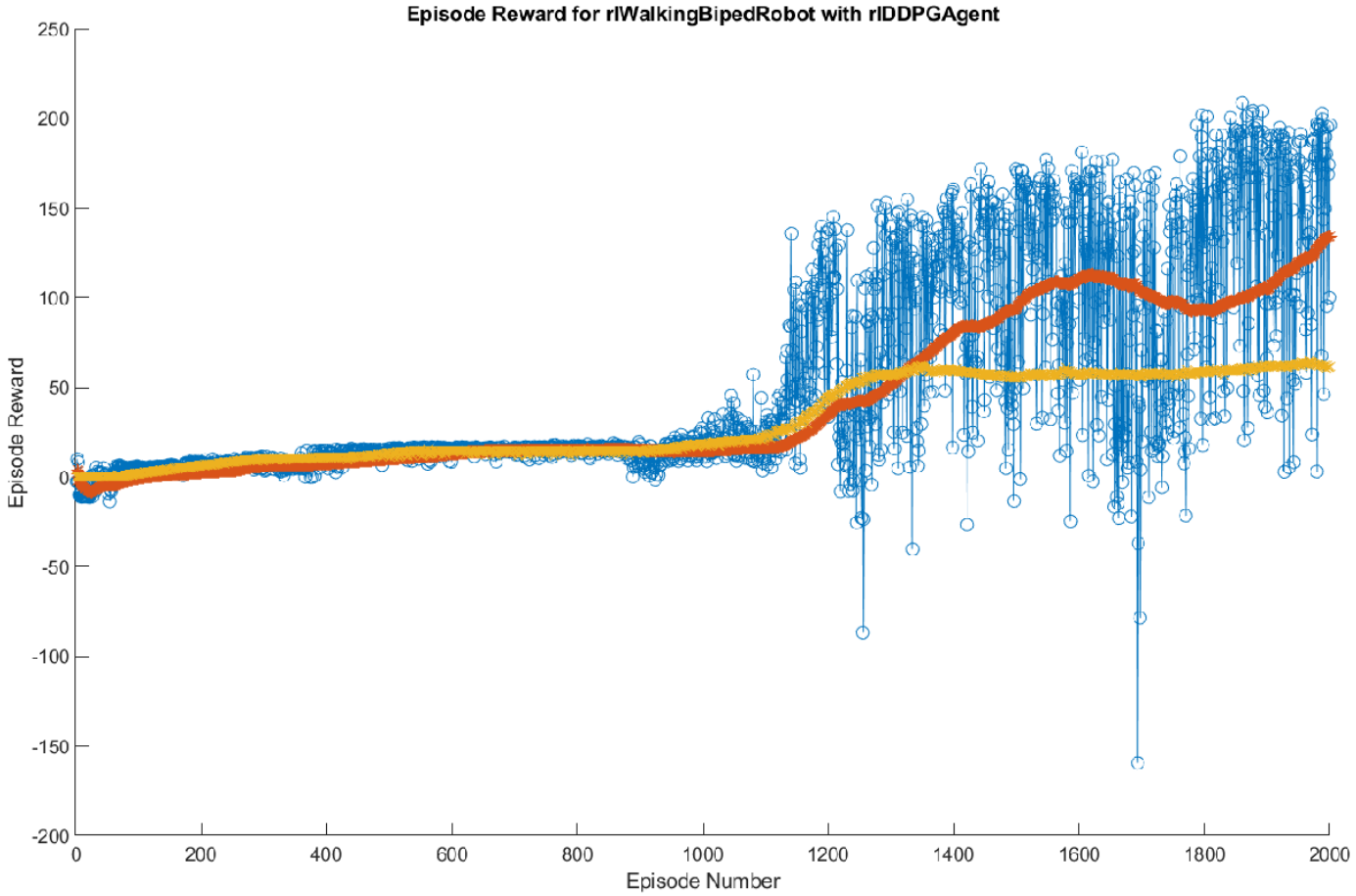
Specify the following training options to train the agent in parallel training mode. If you do not have Parallel Computing Toolbox™ software installed, set `UseParallel` to `false`.

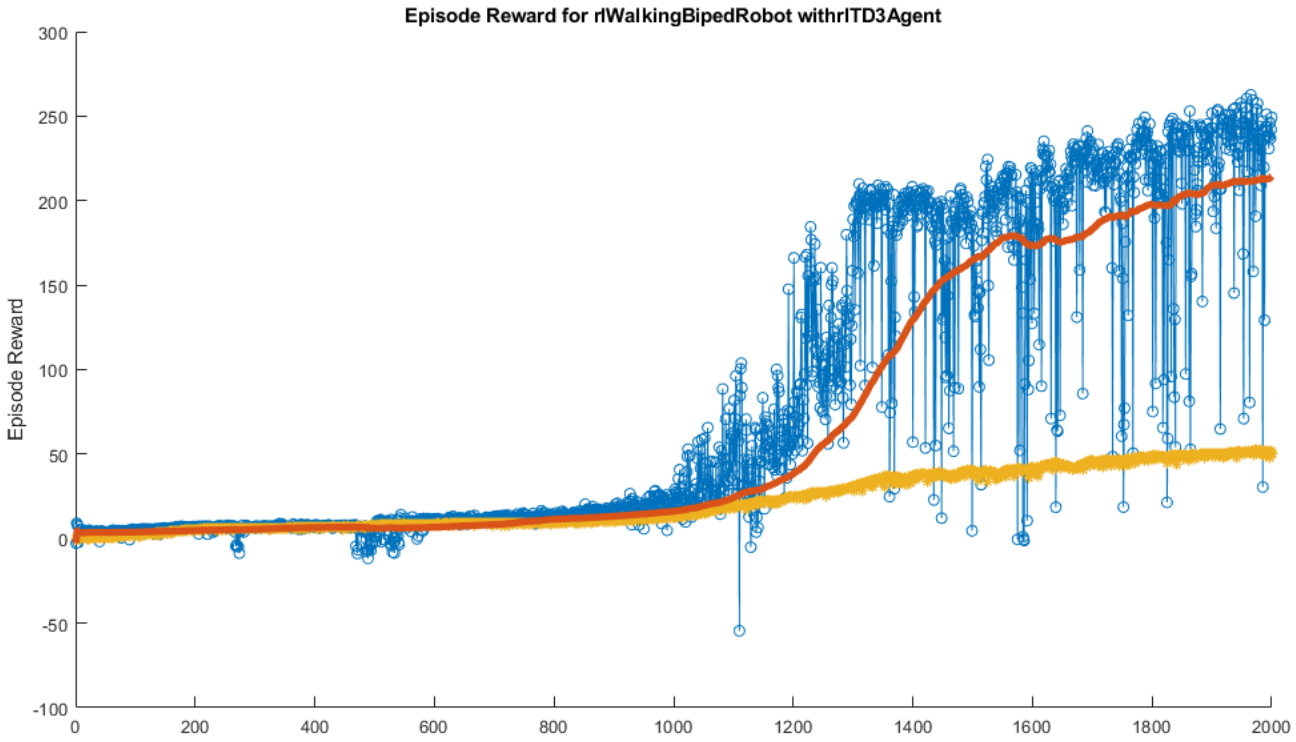
- Set the `UseParallel` option to `true`.
- Train the agent in parallel asynchronously.
- After every 32 steps, have each worker send experiences to the host. DDPG and TD3 agents require workers to send experiencesw to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = 'async';
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;
trainOpts.ParallelizationOptions.DataToSendFromWorkers = 'Experiences';
```

Train the agent using the `train` 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 below. The pretrained agents were trained in parallel using four workers.

```
doTraining = false;
if doTraining
    % Train the agent.
    trainingStats = train(agent,env,trainOpts);
else
    % Load pretrained agent for the selected agent.
    if strcmp(AgentSelection,'DDPG')
       load('rlWalkingBipedRobotDDPG.mat','agent')
    else
       load('rlWalkingBipedRobotTD3.mat','agent')
    end
end
```





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.

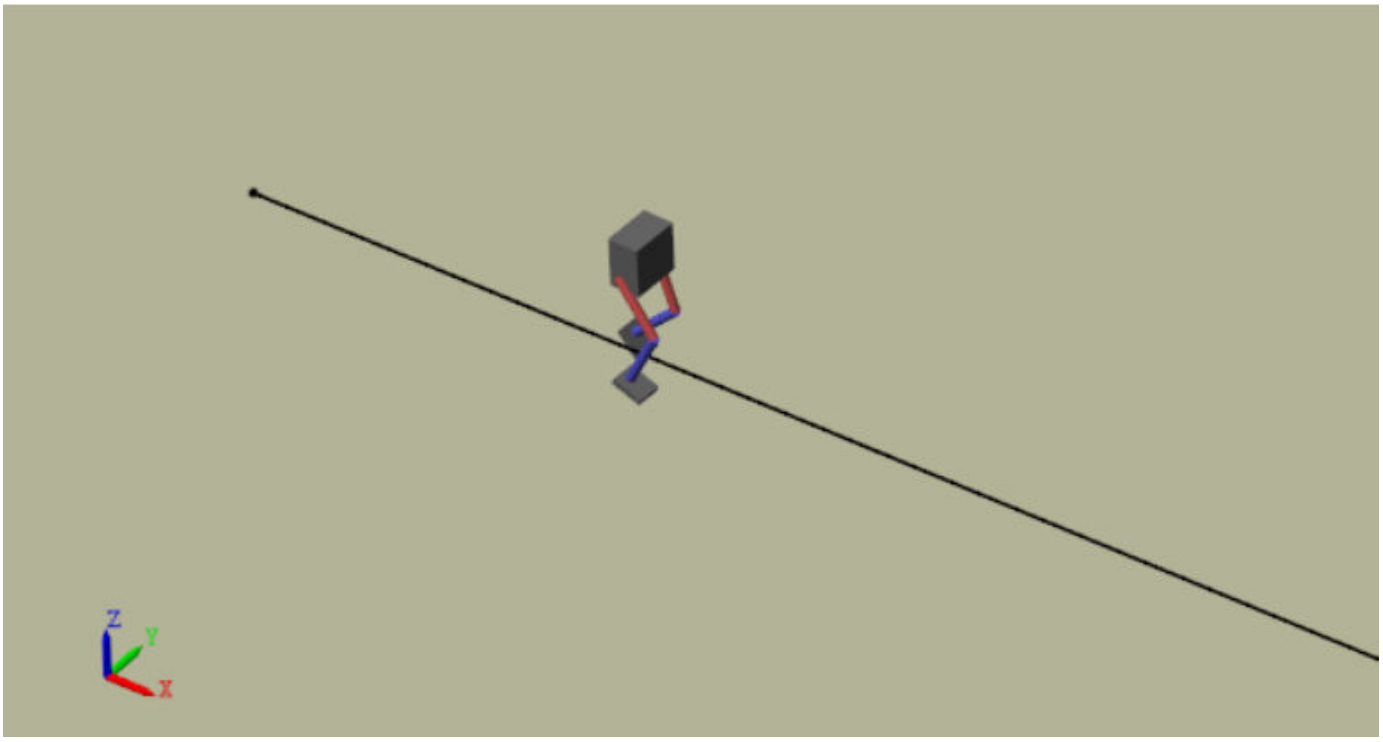
Simulate Trained Agents

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` and `sim`.

```
simOptions = rlSimulationOptions('MaxSteps',maxSteps);
experience = sim(env,agent,simOptions);
```

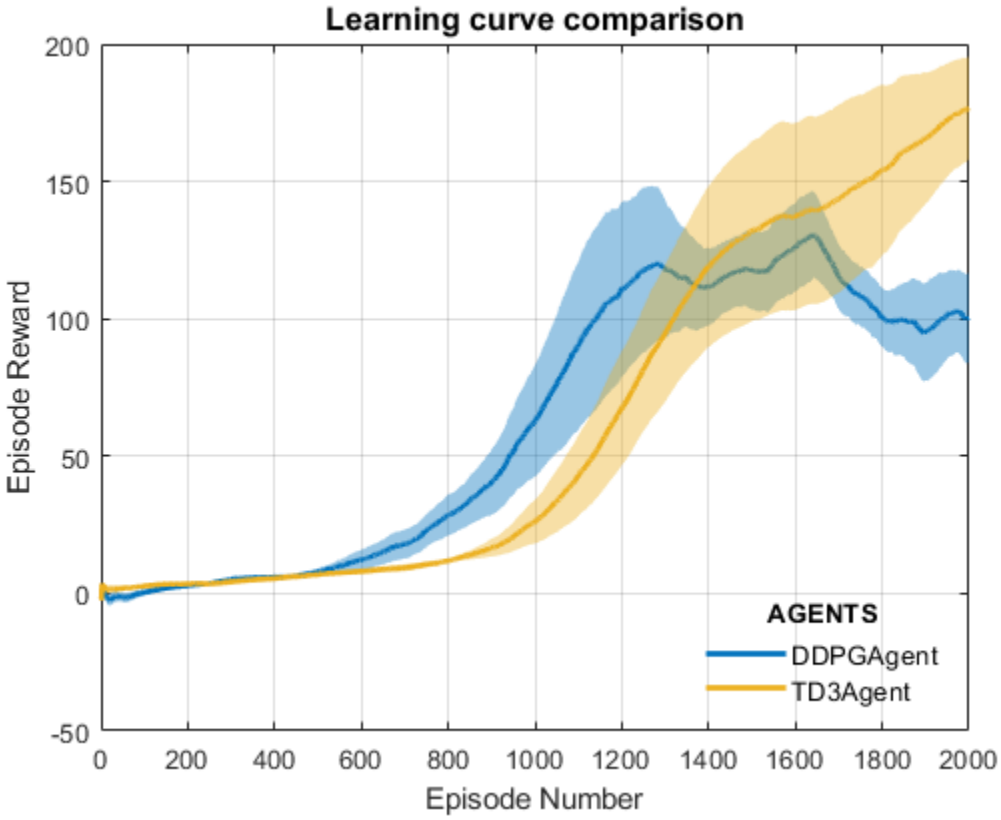


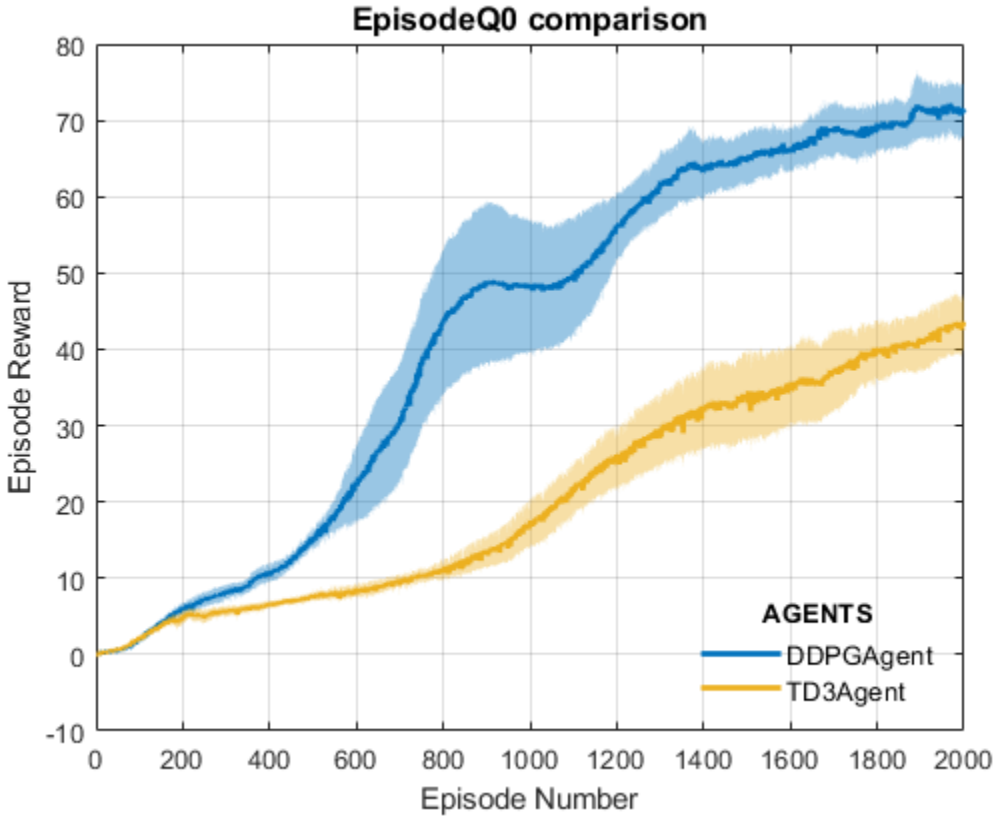
Compare Agent Performance

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:

1. The DDPG appears to pick up learning faster (around episode number 600 on an average) but hits a local minimum. TD3 starts slower but eventually achieves higher rewards than DDPG as it avoids overestimation of Q values.
2. 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:

1. 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 minimum of two Q functions. This behavior is further enhanced because of delayed updates to the targets.
2. Although the TD3 estimate for these 2000 episodes is low, the TD3 agent shows a steady increase in the episode Q0 value,s unlike 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.

References

[1] Heess, Nicolas, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, et al. 'Emergence of Locomotion Behaviours in Rich Environments'. *ArXiv:1707.02286 [Cs]*, 10 July 2017. https://arxiv.org/abs/1707.02286.

[2] 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'. *ArXiv:1509.02971 [Cs, Stat]*, 5 July 2019. https://arxiv.org/abs/1509.02971.

[3] Fujimoto, Scott, Herke van Hoof, and David Meger. 'Addressing Function Approximation Error in Actor-Critic Methods'. *ArXiv:1802.09477 [Cs, Stat]*, 22 October 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 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;
```

Considering the physical limitations of the vehicle dynamics, the acceleration is constrained to the range `[-3,2]` (m/s^2).

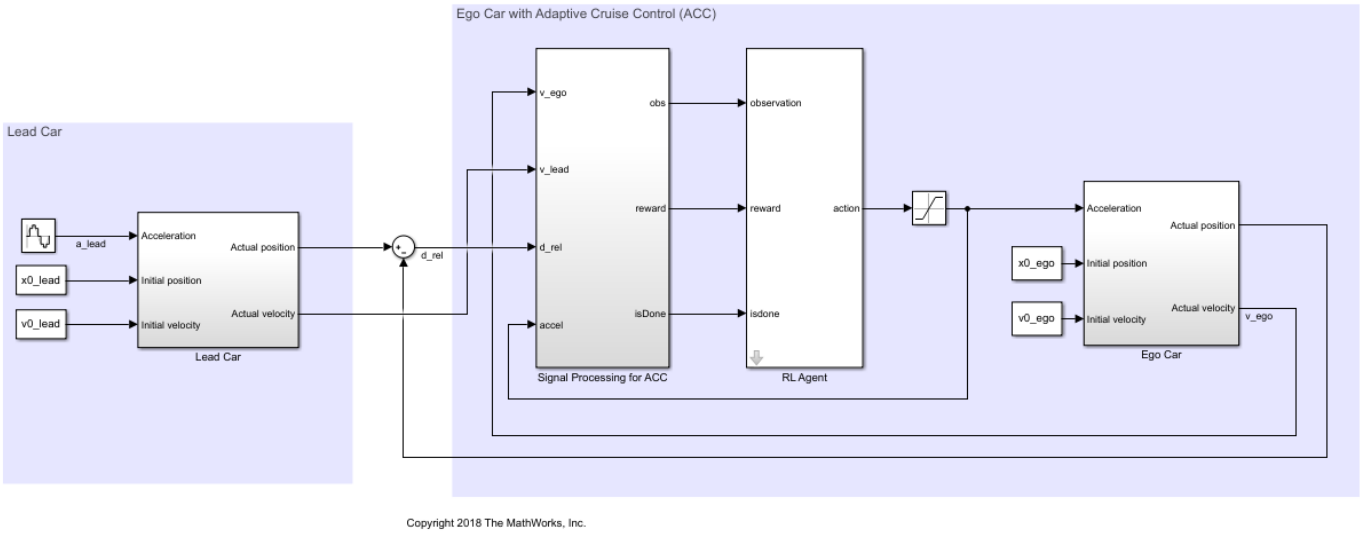
```
amin_ego = -3;
amax_ego = 2;
```

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 ego car V_{ref} is defined as follows. If relative distance is less than safe distance, ego car tracks the minimum of lead car velocity and driver-set velocity. In this manner, the ego car maintains some distance from lead car. If the relative distance is greater than safe distance, the ego car tracks driver-set velocity. In this example, safe distance is defined as a linear function of 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 info
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';
% action Info
actionInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
actionInfo.Name = 'acceleration';
% define environment
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.

```
% randomize initial positions of lead car
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng('default')
```

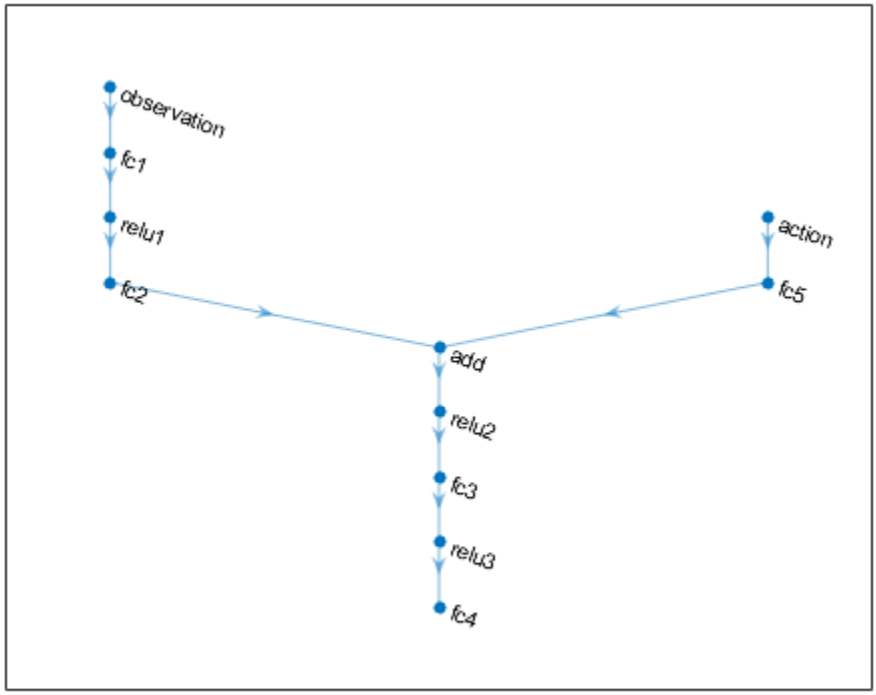
Create DDPG agent

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 = [
    imageInputLayer([3 1 1],'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 = [
    imageInputLayer([1 1 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`.

```
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`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'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 neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    imageInputLayer([3 1 1],'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`.

```
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`.

```
agent = rlDDPGAgent(actor,critic,agentOptions);
```

Train Agent

To train the agent, first specify the training options. For this example, use the following options:

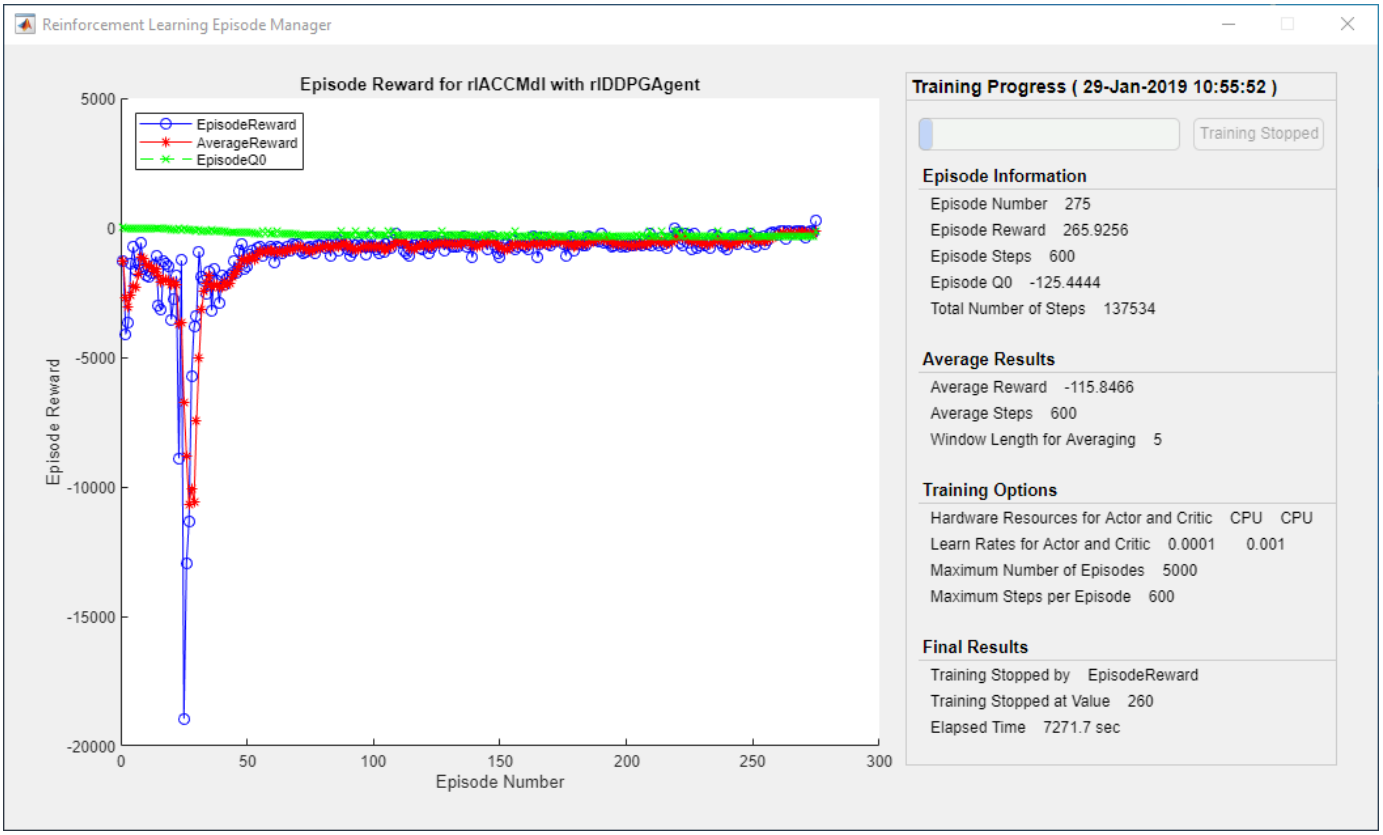
- Run each training episode for at most `5000` episodes, with each episode lasting at most 600 time steps.
- Display the training progress in the Episode Manager dialog box.
- Stop training when the agent receives an episode reward greater than 260.

For more information, see `rlTrainingOptions`.

```
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` function. This 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 pretrained agent for the example.
    load('SimulinkACCDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, uncomment the following commands and simulate it within the Simulink environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

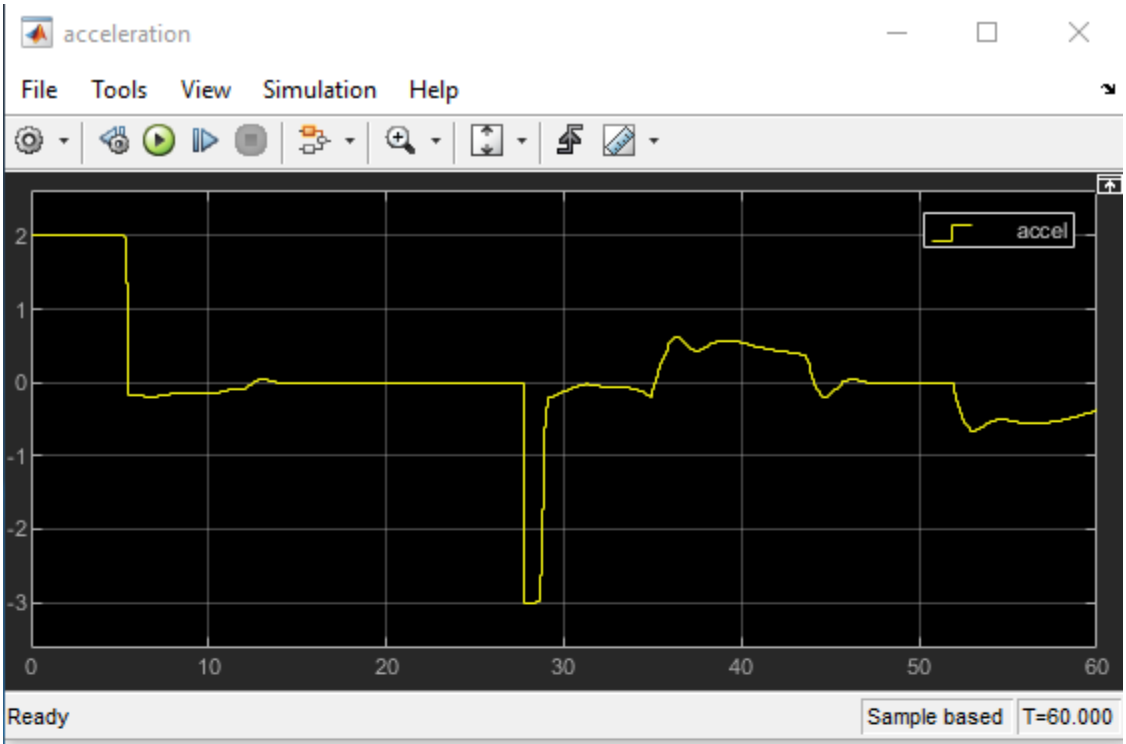
```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

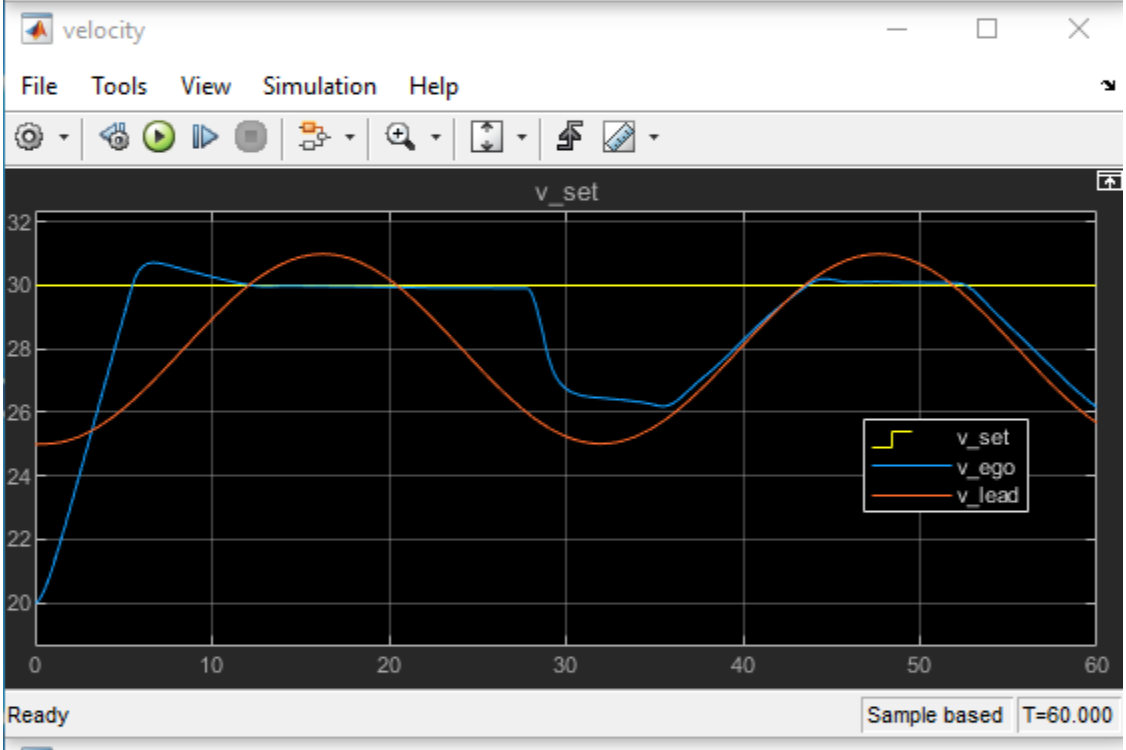
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

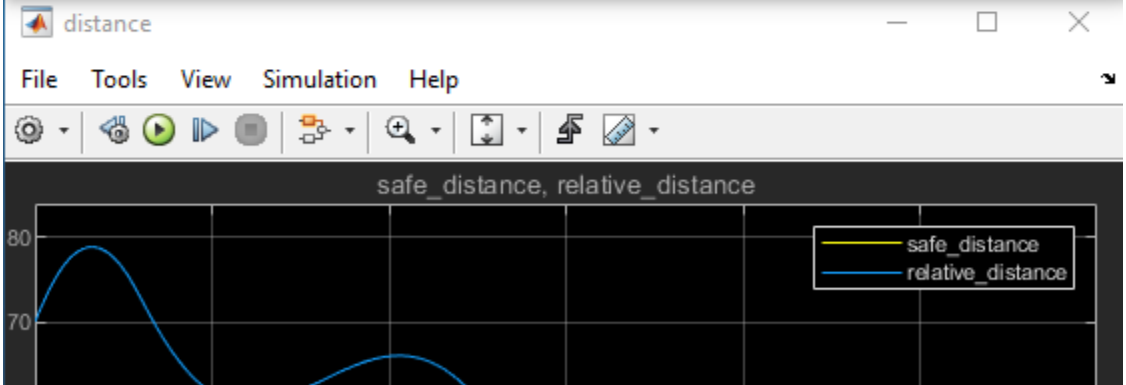
```
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when lead car is 70 (m) ahead of ego car.

- In the first 28 seconds, relative distance is greater than safe distance (bottom plot), therefore 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, relative distance is less than safe distance (bottom plot), therefore 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, 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)
```

Local Function

```
function in = localResetFcn(in)
% reset initial position of 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 host agent. The host agent updates its parameters as follows.

- For asynchronous training, the host 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 host agent waits to receive experiences from all of the workers and learns from these experiences. The host then sends updated parameters to all the workers at the same time. Then, all workers continue to generate experiences using the updated parameters.

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.

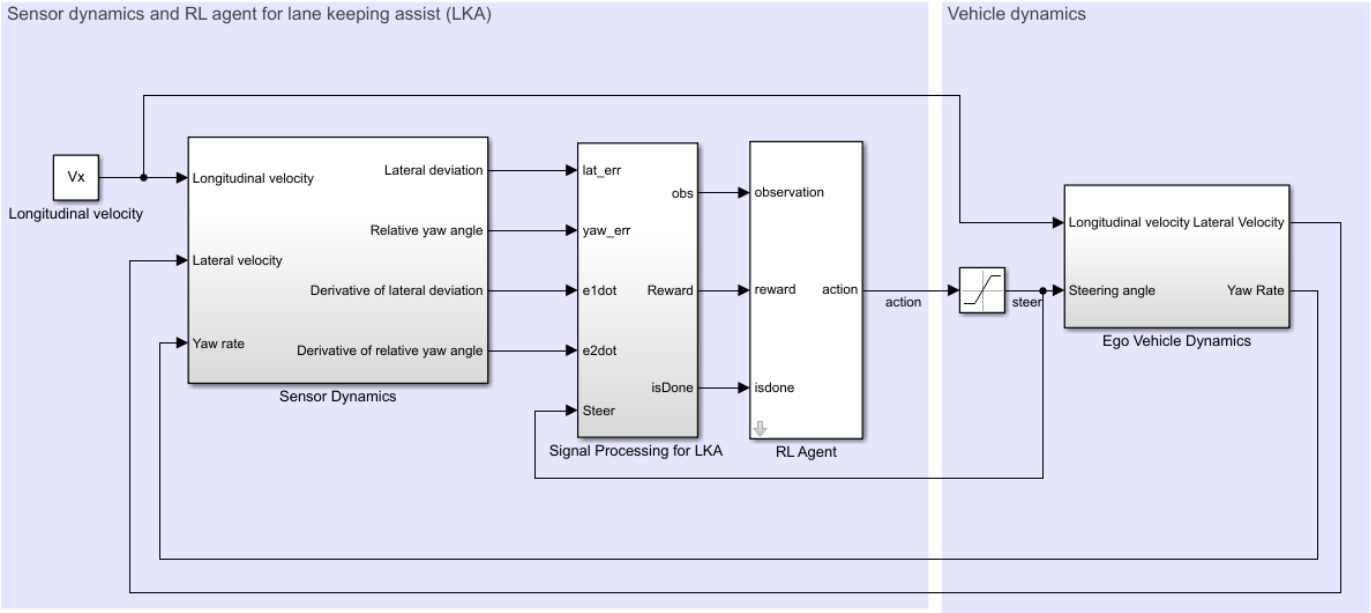
```
u_min = -0.5;
u_max = 0.5;
```

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.

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

A DQN 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 deep neural network value function representation, see “Create Policy and Value Function Representations” (Reinforcement Learning Toolbox).

```
L = 24; % number of neurons
statePath = [
    imageInputLayer([6 1 1],'Normalization','none','Name','state')
    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 = [
    imageInputLayer([1 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(L, 'Name', 'fc5')];

criticNetwork = layerGraph(statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');
```

Specify options for the critic representation using `rlRepresentationOptions`.

```
criticOpts = rlRepresentationOptions('LearnRate',1e-3,'GradientThreshold',1);
```

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`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,'Observation',{'state'}
```

To create the DQN agent, first specify the DQN agent options using `rlDQNAgentOptions`.

```
agentOpts = rlDQNAgentOptions(...
    'SampleTime',Ts,...
    'UseDoubleDQN',true,...
    'TargetSmoothFactor',1e-3,...
    'DiscountFactor',0.99,...
    'ExperienceBufferLength',1e6,...
    'MiniBatchSize',64);
```

Then create the DQN agent using the specified critic representation and agent options. For more information, see `rlDQNAgent`.

```
agent = rlDQNAgent(critic,agentOpts);
```

Parallel Training Options

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 `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 `-2.5`.

For more information, see `rlTrainingOptions`.

```
maxepisodes = 5000;
maxsteps = ceil(T/Ts);
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes, ...
    'MaxStepsPerEpisode',maxsteps, ...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue', -1,...
    'SaveAgentCriteria','EpisodeReward',...
    'SaveAgentValue',-2.5);
```

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 host.
- DQN agent requires workers to send "`experiences`" to the host.

```
trainOpts.UseParallel = true;
trainOpts.ParallelizationOptions.Mode = "async";
```

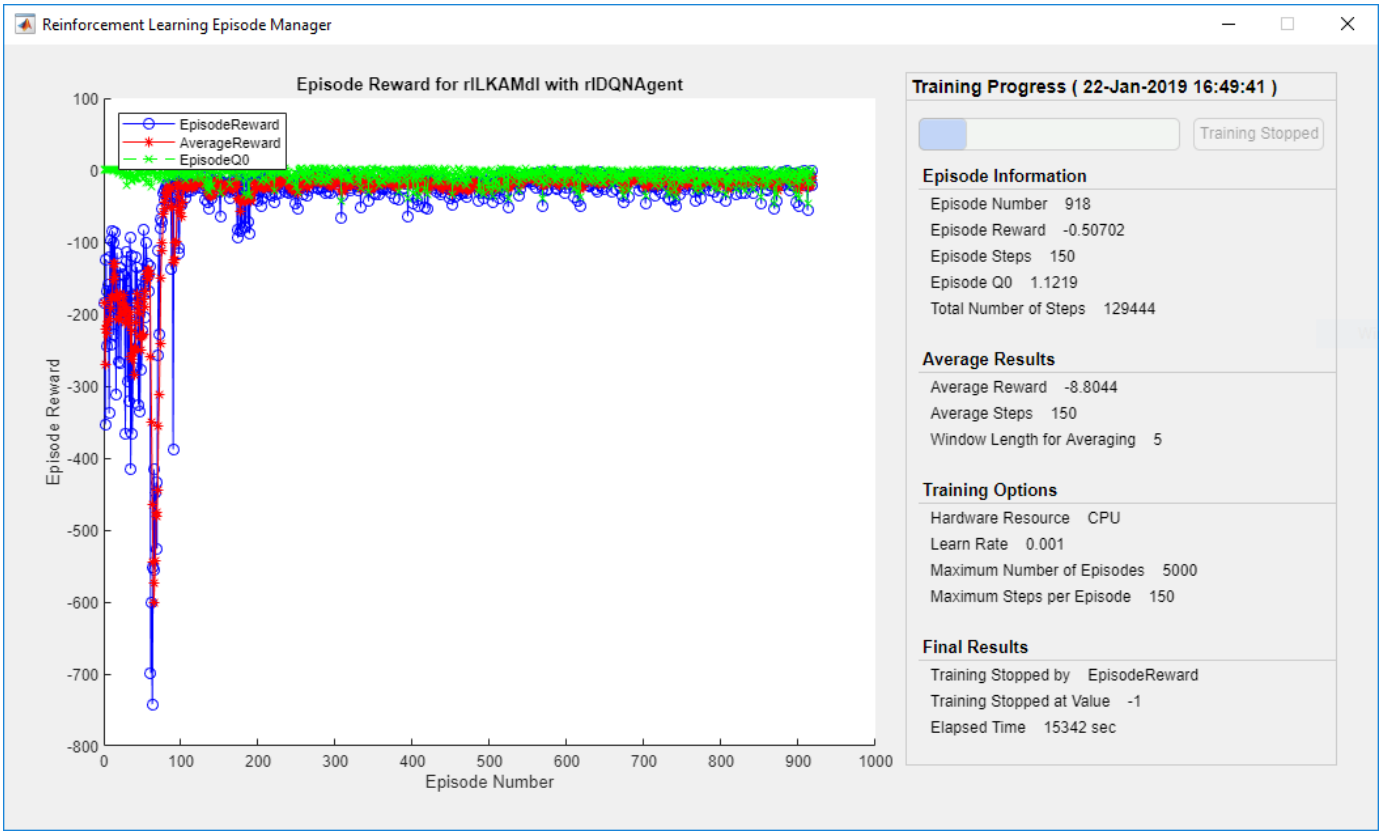
```
trainOpts.ParallelizationOptions.DataToSendFromWorkers = "experiences";
trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 30;
trainOpts.ParallelizationOptions.WorkerRandomSeeds = -1;
```

For more information, see `rlTrainingOptions`.

Train Agent

Train the agent using the `train` 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

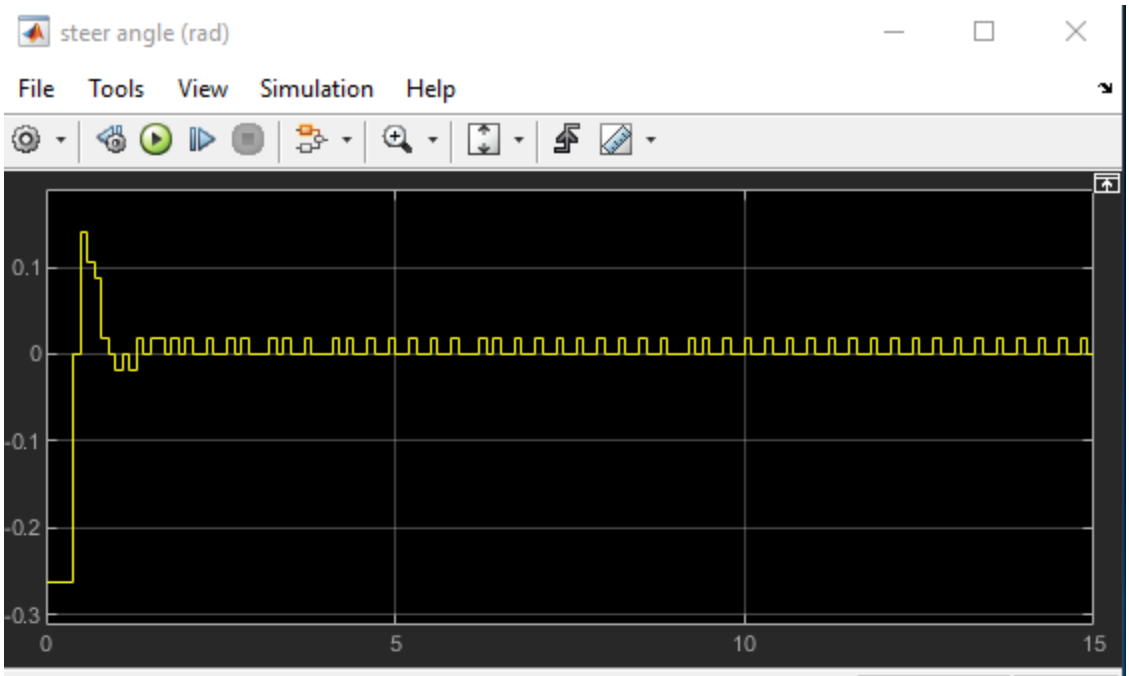
To validate the performance of the trained agent, uncomment the following two lines and simulate the agent within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

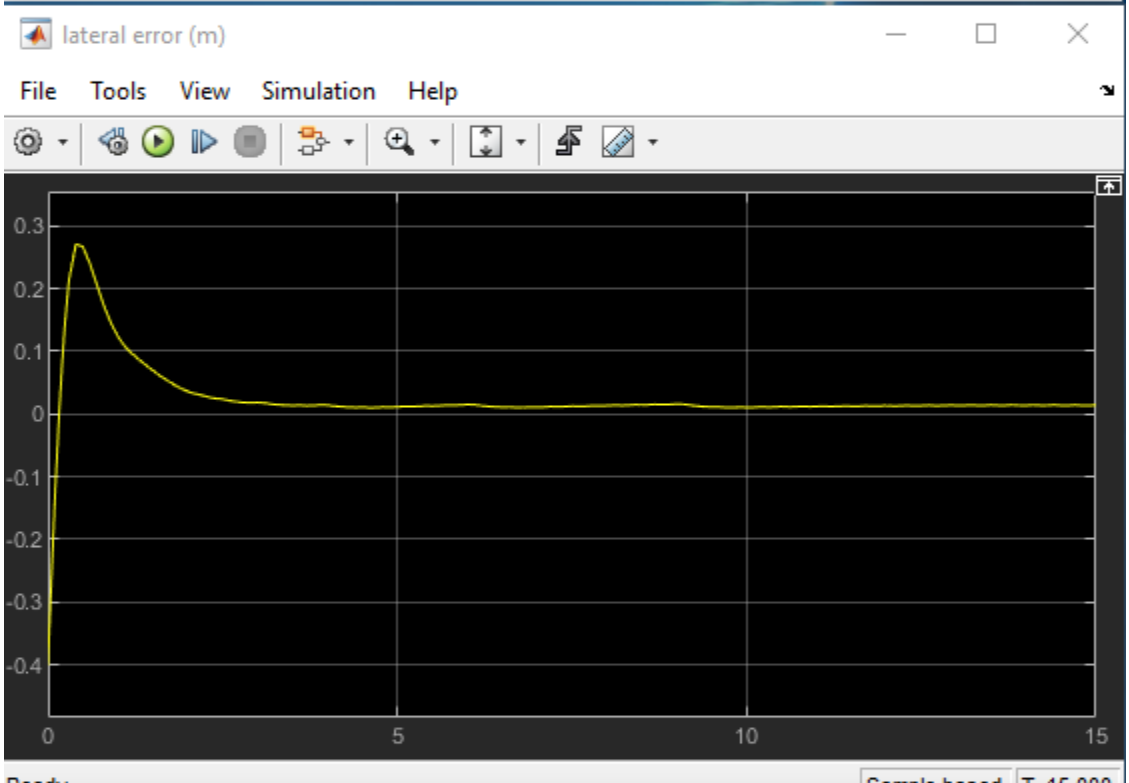
```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

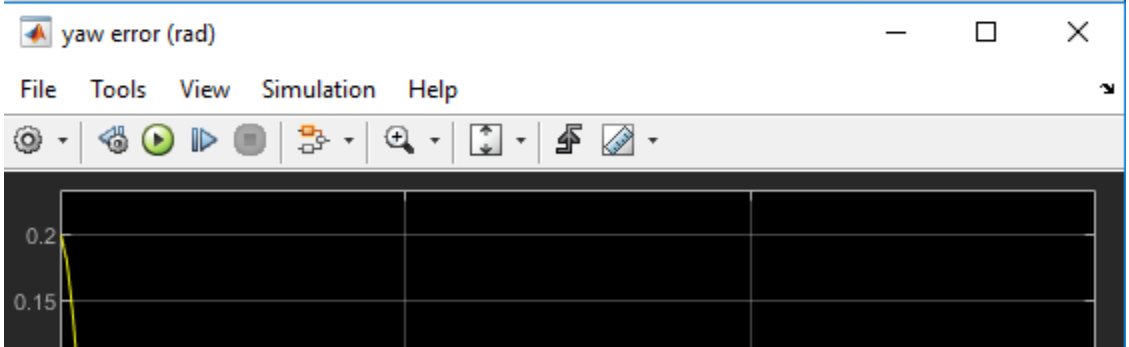
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
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


yaw error (rad)
File Tools View Simulation Help
0.2
0.15

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 ego car and a simple longitudinal model for 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, while also 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. 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 standstill default spacing (m), time gap (s) and driver-set velocity (m/s).

```
D_default = 10;
t_gap = 1.4;
v_set = 28;
```

Considering the physical limitations of the vehicle dynamics, the acceleration is constrained 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 relative yaw angle is -0.1 rad.

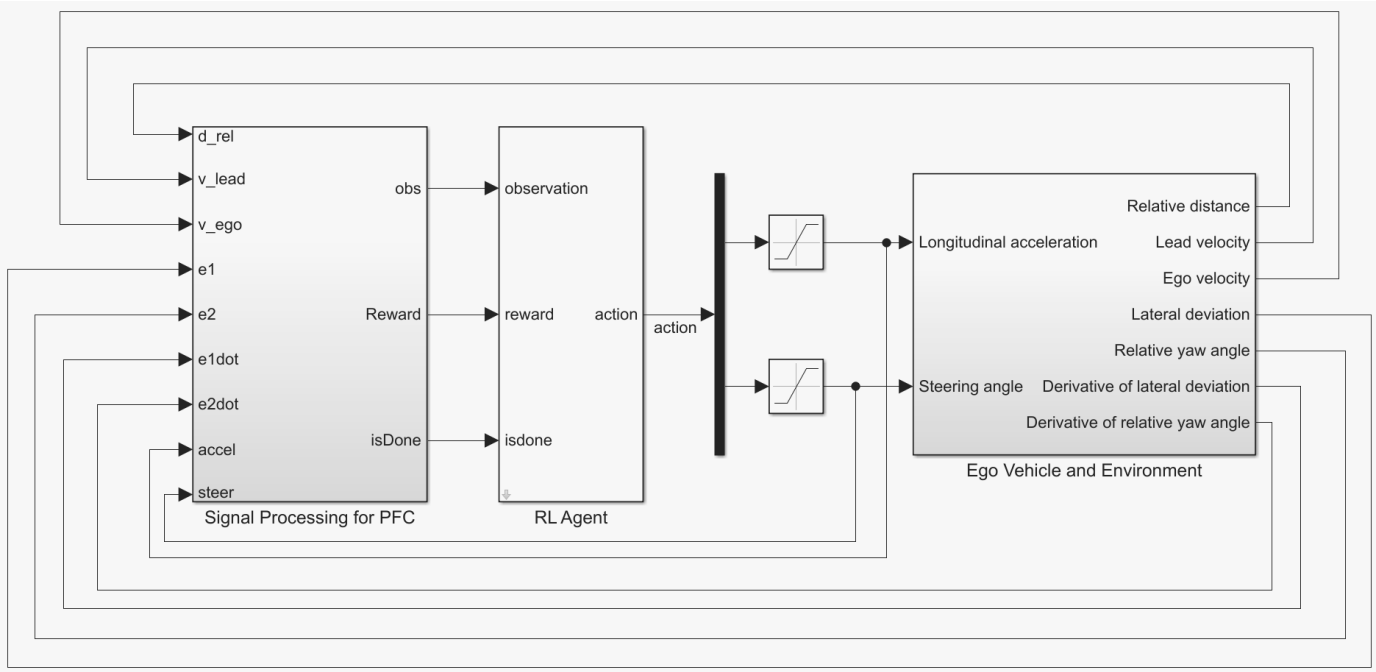
```
rho = 0.001;
e1_initial = 0.2;
e2_initial = -0.1;
```

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 value between -15 degrees (-0.2618 rad) to 15 degrees (0.2618 rad).
- The reference velocity for ego car V_{ref} is defined as follows. If relative distance is less than safe distance, ego car tracks the minimum of lead car velocity and driver-set velocity. In this manner, ego car maintains some distance from lead car. If relative distance is greater than safe distance, ego car tracks driver-set velocity. In this example, safe distance is defined as a linear function of 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 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 is terminated when lateral deviation $|e_1| > 1$ or longitudinal velocity $V_{ego} < 0.5$ or relative distance between 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: $F_t = 1$if 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, 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 info
observationInfo = rlNumericSpec([9 1],'LowerLimit',-inf*ones(9,1),'UpperLimit',inf*ones(9,1));
observationInfo.Name = 'observations';
% action Info
actionInfo = rlNumericSpec([2 1],'LowerLimit',[-3;-0.2618],'UpperLimit',[2;0.2618]);
actionInfo.Name = 'accel;steer';
% define environment
env = rlSimulinkEnv(mdl,agentblk,observationInfo,actionInfo);
```

To define the initial conditions, specify an environment reset function using an anonymous function handle.

```
% randomize initial positions of lead car, lateral deviation and relative
% yaw angle
env.ResetFcn = @(in)localResetFcn(in);
```

Fix the random generator seed for reproducibility.

```
rng(0)
```

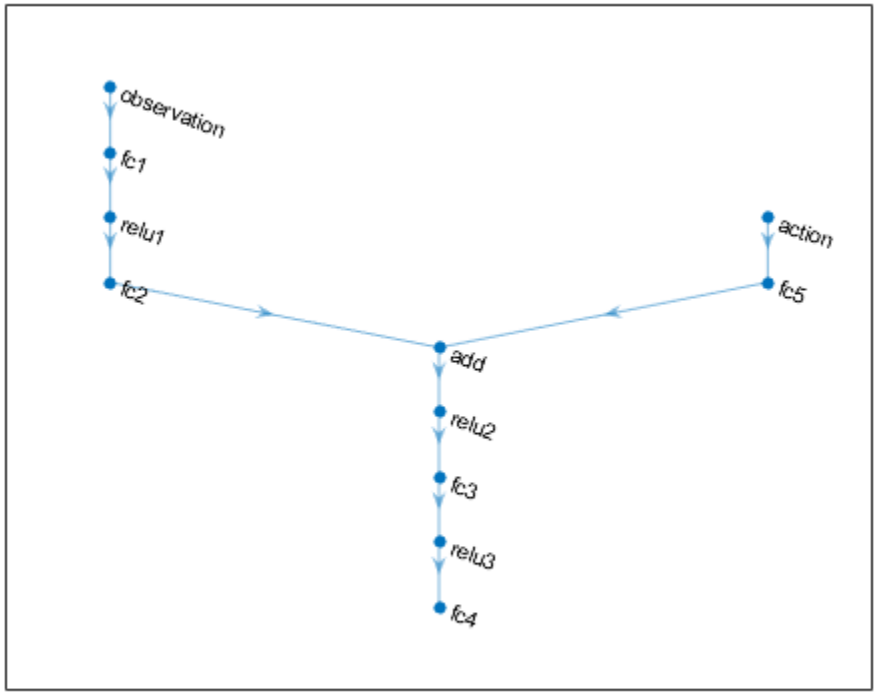
Create DDPG agent

A DDPG agent approximates the long-term reward given observations and actions 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 = [
    imageInputLayer([9 1 1],'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 = [
    imageInputLayer([2 1 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.

```
figure
plot(criticNetwork)
```



Specify options for the critic representation using `rlRepresentationOptions`.

```
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`.

```
critic = rlQValueRepresentation(criticNetwork,observationInfo,actionInfo,...
    'Observation',{'observation'},'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 neural network with one input, the observation, and one output, the action.

Construct the actor similarly to the critic. For more information, see `rlDeterministicActorRepresentation`.

```
actorNetwork = [
    imageInputLayer([9 1 1],'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',reshape([2.5;0.2618],[1,1,2]),'Bias',reshape([-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`.

```
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`.

```
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`.

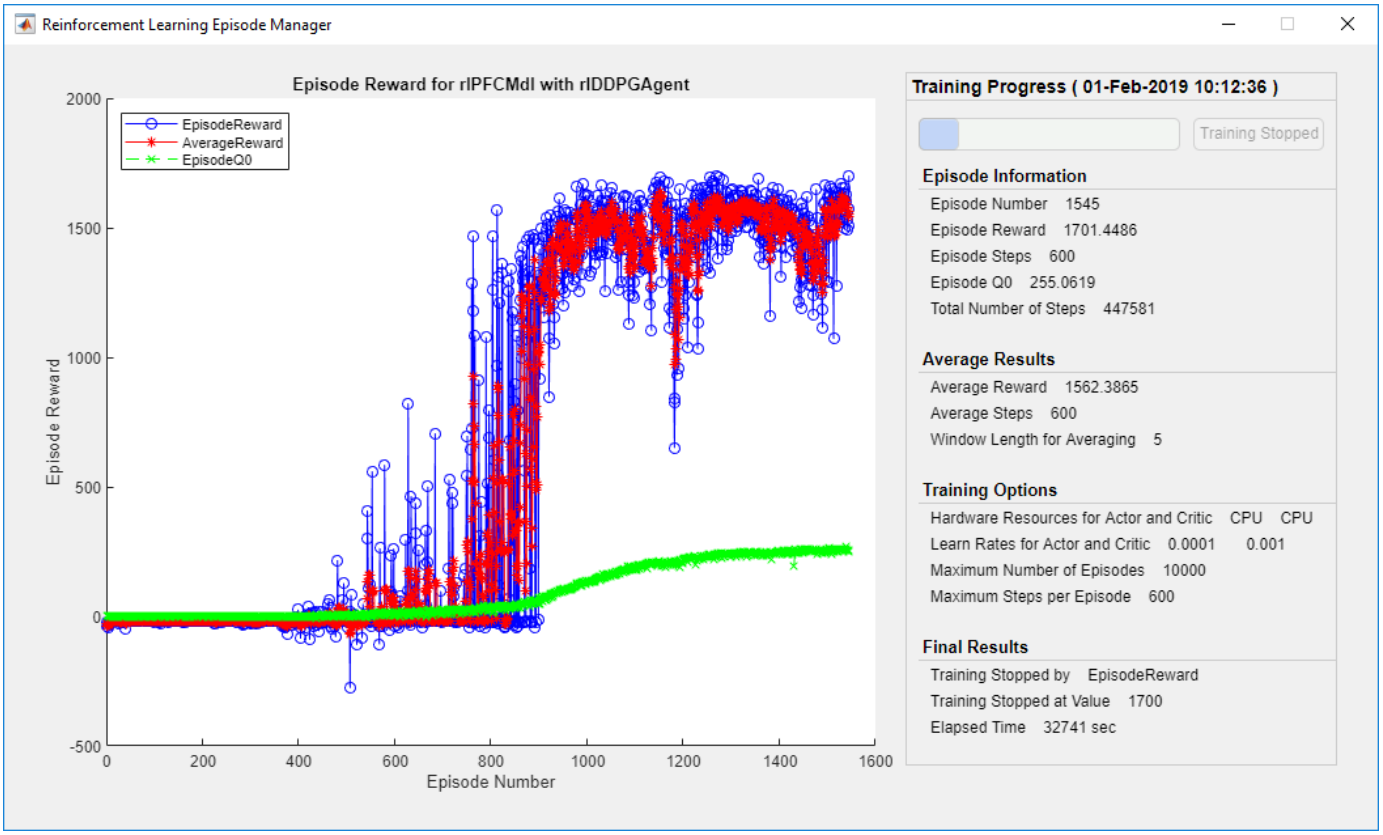
For more information, see `rlTrainingOptions`.

```
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` function. This 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 pretrained agent for the example.
    load('SimulinkPFCDDPG.mat','agent')
end
```



Simulate DDPG Agent

To validate the performance of the trained agent, uncomment the following two lines and simulate it within the environment. For more information on agent simulation, see `rlSimulationOptions` and `sim`.

```
% simOptions = rlSimulationOptions('MaxSteps',maxsteps);
% experience = sim(env,agent,simOptions);
```

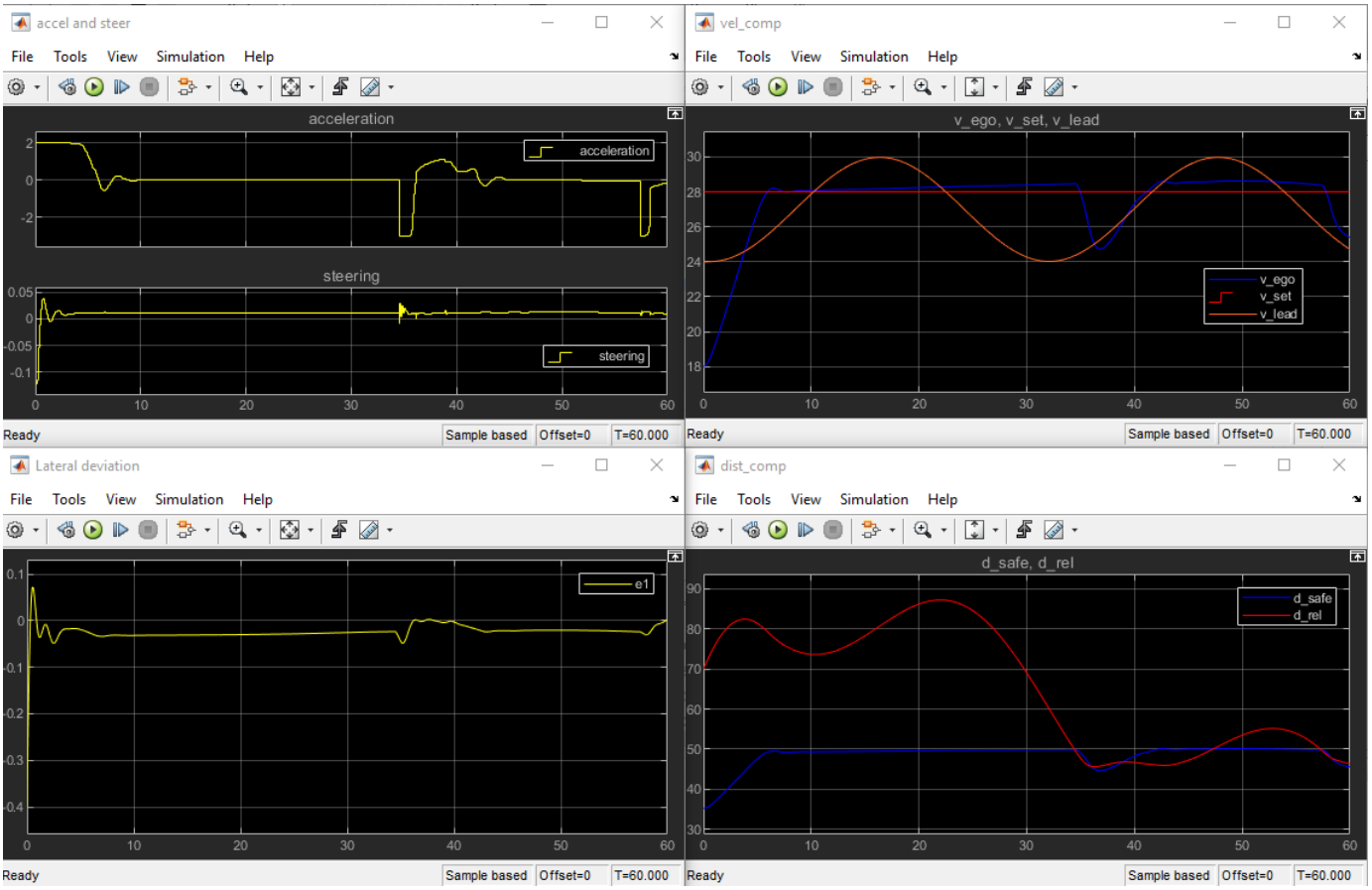
To demonstrate the trained agent using deterministic initial conditions, simulate the model in Simulink.

```
e1_initial = -0.4;
e2_initial = 0.1;
x0_lead = 80;
sim(mdl)
```

The following plots show the simulation results when lead car is 70 (m) ahead of ego car.

- In the first 35 seconds, relative distance is greater than safe distance (bottom right plot), thus ego car tracks set velocity (top right plot). To speed up and reach the set velocity, acceleration is mostly non-negative (top left plot).

- From 35 to 42 seconds, relative distance is mostly less than safe distance (bottom right plot), thus ego car tracks the minimum of lead velocity and set velocity. Since lead velocity is less than set velocity (top right plot), to track lead velocity, acceleration becomes non-zero (top left plot).
- From 42 to 58 seconds, ego car tracks set velocity (top right plot) and acceleration remains zero (top left plot).
- From 58 to 60 seconds, relative distance becomes less than safe distance (bottom right plot), thus ego car slows down and tracks lead velocity.
- The bottom left plot shows the lateral deviation. As shown in the plot, the lateral deviation is greatly decreased within one second. The lateral deviation remains less than 0.05 m.



Close Simulink model.

```
bdclose(mdl)
```

Local Function

```
function in = localResetFcn(in)
% reset
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)

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 14-0 . 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 1-53.
- 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))';
```

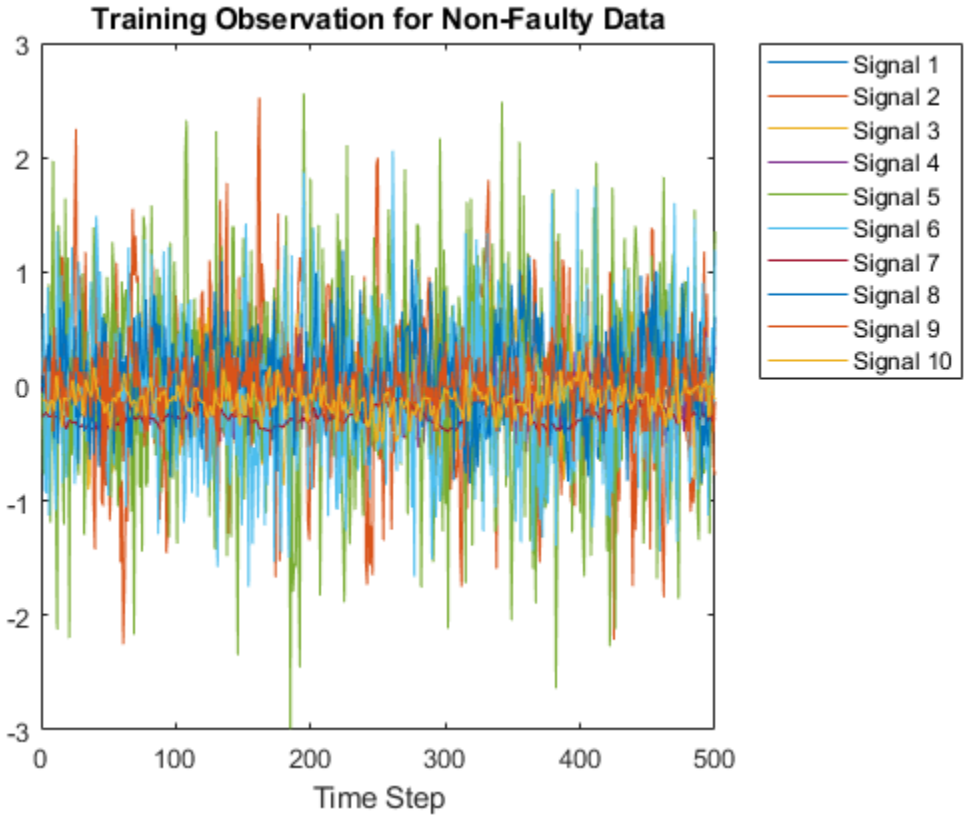
Use the helper function `helperNormalize` to apply normalization to each cell in the three data sets based on the mean and standard deviation of the training data.

```
Xtrain = helperNormalize(Xtrain, tMean, tSigma);
XVal = helperNormalize(XVal, tMean, tSigma);
Xtest = helperNormalize(Xtest, tMean, tSigma);
```

Visualize Data

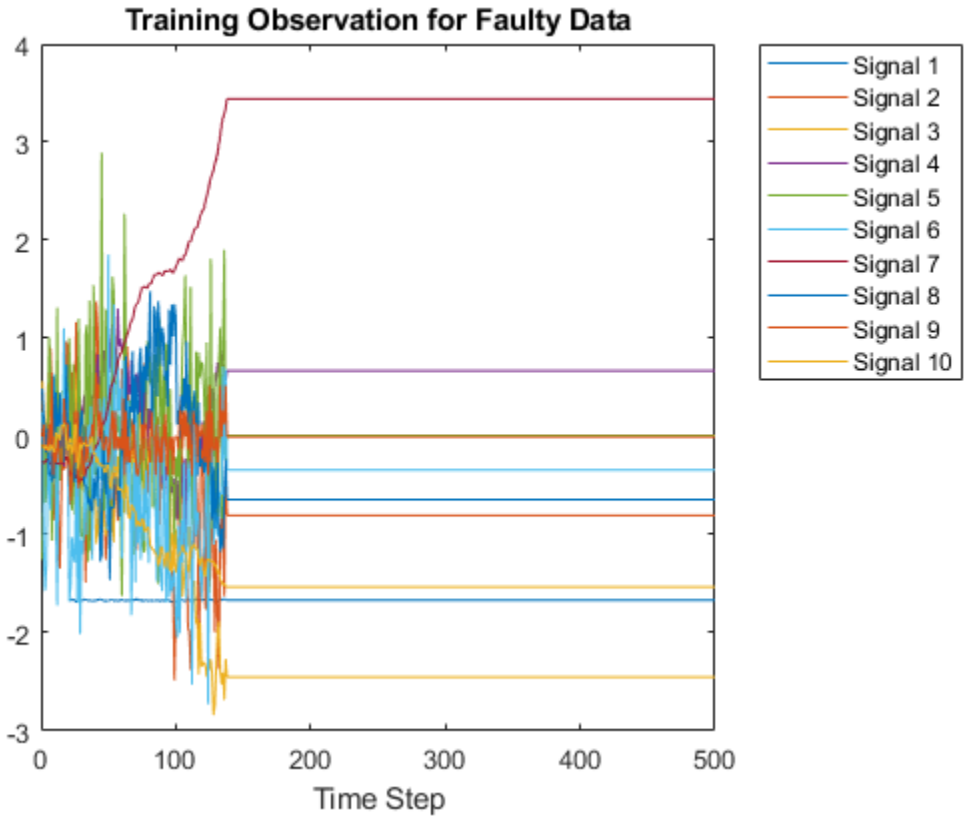
The `Xtrain` data set contains 400 fault-free simulations followed by 6800 faulty simulations.Visualize the fault-free and faulty data. First, create a plot of the fault-free data. For the purposes of this example, plot and label only 10 signals in the `Xtrain` data set to create an easy-to-read figure.

```
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 14-0 . 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. If a GPU is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher), the software uses the GPU. Otherwise, the software uses CPU. 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.

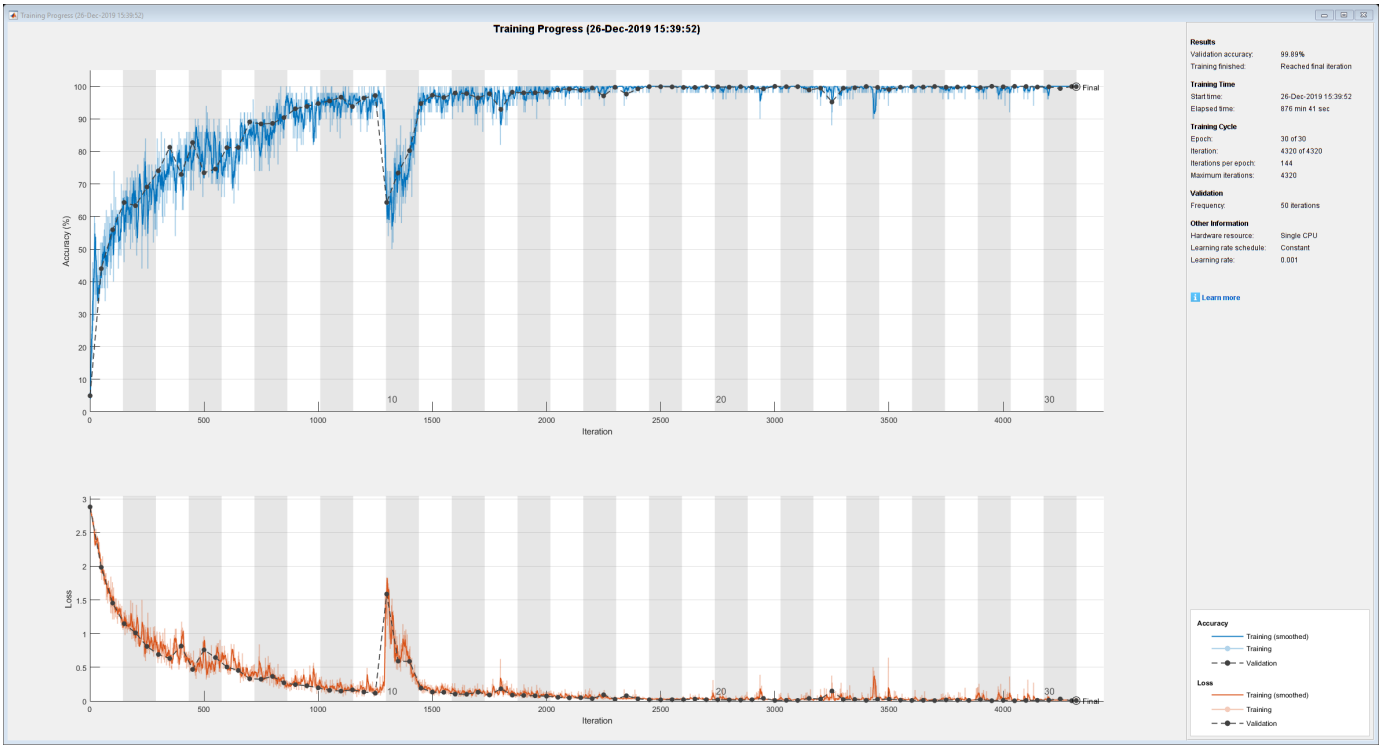
For more information on training options for deep learning, see `trainingOptions`.

```
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.

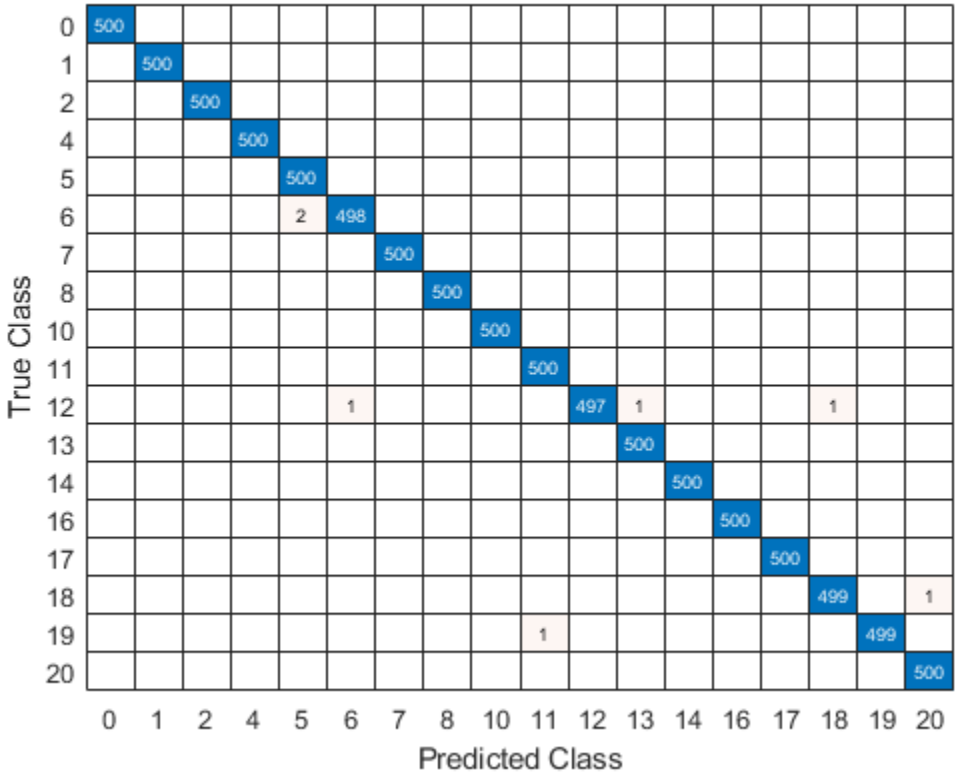
```
acc = sum(Ypred == Ytest)./numel(Ypred)
```

```
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` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Long Short-Term Memory Networks” on page 1-53
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

Automatic Differentiation

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 1-23.

This topic explains the architecture of deep learning layers and how to define custom layers to use for your problems.

Type	Description
Layer	Define a custom deep learning layer and specify optional learnable parameters. For an example showing how to define a custom layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17. For an example showing how to define a custom layer with multiple inputs, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28.
Classification Output Layer	Define a custom classification output layer and specify a loss function. 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 15-39.
Regression Output Layer	Define a custom regression output layer and specify a loss function. 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 15-54.

Layer Templates

You can use the following templates to define new layers.

Intermediate Layer Template

This template outlines the structure of an intermediate layer with learnable parameters. If the layer does not have learnable parameters, then you can omit the `properties (learnable)` section. For an example showing how to define a layer with learnable parameters, see “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17.

```
classdef myLayer < nnet.layer.Layer

    properties
        % (Optional) Layer properties.
```

```
        % Layer properties go here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Layer learnable parameters go here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Layer constructor function goes here.
        end

        function [Z1, …, Zm] = predict(layer, X1, …, Xn)
            % Forward input data through the layer at prediction time and
            % output the result.
            %
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function

            % Layer forward function for prediction goes here.
        end

        function [Z1, …, Zm, memory] = forward(layer, X1, …, Xn)
            % (Optional) Forward input data through the layer at training
            % time and output the result and a memory value.
            %
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function
            %         memory      - Memory value for custom backward propagation

            % Layer forward function for training goes here.
        end

        function [dLdX1, …, dLdXn, dLdW1, …, dLdWk] = ...
                backward(layer, X1, …, Xn, Z1, …, Zm, dLdZ1, …, dLdZm, memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer             - Layer to backward propagate through
            %         X1, ..., Xn       - Input data
            %         Z1, ..., Zm       - Outputs of layer forward function
            %         dLdZ1, ..., dLdZm - Gradients propagated from the next layers
            %         memory            - Memory value from forward function
            % Outputs:
            %         dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
            %                             inputs
            %         dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
            %                             learnable parameter

            % Layer backward function goes here.
        end
    end
end
```

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 15-39.

```
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 15-54.

```
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.

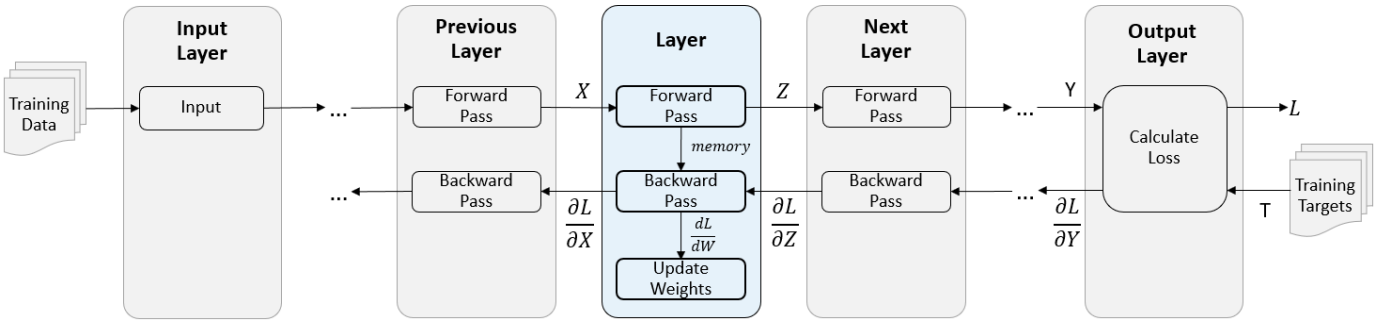
When making a forward pass through the network, each layer takes the outputs of the previous layers, applies a function, and then outputs (forward propagates) the results to the next layers.

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 the next layers.

At the end of a forward pass of the network, the output layer calculates the loss L between the predictions Y and the true 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 Properties

Declare the layer properties in the `properties` section of the class definition.

By default, custom intermediate layers have these properties:

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 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`	The 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`	The 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` in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` in the layer constructor. For an example, see "Define Custom Deep Learning Layer with Multiple Inputs" on page 15-28.

Learnable Parameters

Declare the layer learnable parameters in the `properties (Learnable)` section of the class definition. If the layer has no learnable parameters, then you can omit the `properties (Learnable)` section.

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.
`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,'MyParameterName',value)` and `layer = setL2Factor(layer,'MyParameterName',value)`, respectively.

To get the value of the learning rate factor and the L2 factor of a learnable parameter, use the syntaxes `getLearnRateFactor(layer,'MyParameterName')` and `getL2Factor(layer,'MyParameterName')` 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);
```

Forward Functions

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 you define the function `forward` and custom backward function, then you must assign a value to the argument `memory`, which you can use during backward propagation.

The syntax for `predict` is

```
[Z1,…,Zm] = predict(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs and `Z1,…,Zm` are the `m` layer outputs. The values `n` and `m` must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

The syntax for `forward` is

```
[Z1,…,Zm,memory] = forward(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs, `Z1,…,Zm` are the `m` layer outputs, and `memory` is the memory of the layer.

Tip If the number of inputs to `forward` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj` for `j`=1,...,`NumOutputs` and `varargout{NumOutputs+1}` corresponds to `memory`.

The dimensions of the inputs depend on the type of data and the output of the connected layers:

Layer Input	Input Size	Observation Dimension
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

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 15-194. 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.

To define a custom backward function, create a function named `backward`.

The syntax for `backward` is

```
[dLdX1,…,dLdXn,dLdW1,…,dLdWk] = backward(layer,X1,…,Xn,Z1,…,Zm,dLdZ1,…,dLdZm,memory)
```

where:

- `X1,…,Xn` are the `n` layer inputs
- `Z1,…,Zm` are the `m` outputs of the layer forward functions
- `dLdZ1,…,dLdZm` are the gradients backward propagated from the next layer
- `memory` is the memory output of `forward` if `forward` is defined, otherwise, `memory` is `[]`.

For the outputs, `dLdX1,…,dLdXn` are the derivatives of the loss with respect to the layer inputs and `dLdW1,…,dLdWk` are the derivatives of the loss with respect to the `k` learnable 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 `varargin{i}` corresponds to `Xi` for `i`=1,...,`NumInputs`, `varargin{NumInputs+j}` and `varargin{NumInputs +NumOutputs+j}` correspond to `Zj` and `dLdZj`, respectively, for `j`=1,...,`NumOutputs`, and `varargin{end}` 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 `varargout{i}` corresponds to `dLdXi` for `i`=1,

...,`NumInputs` and `varargout{NumInputs+t}` corresponds to `dLdWt` for `t`=1,...,`k`, where `k` is the number of learnable parameters.

The values of `X1,…,Xn` and `Z1,…,Zm` are the same as in the forward functions. The dimensions of `dLdZ1,…,dLdZm` are the same as the dimensions of `Z1,…,Zm`, respectively.

The dimensions and data type of `dLdX1,…,dLdxn` are the same as the dimensions and data type of `X1,…,Xn`, respectively. The dimensions and data types of `dLdW1`,...,`dLdWk` are the same as the dimensions and data types of `W1`,...,`Wk`, respectively.

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 `dLdW1`,...,`dLdWk` during the backward pass.

For an example showing how to define a custom backward function, see “Specify Custom Layer Backward Function” on page 15-62.

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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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, and correctly defined gradients. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer, and `dim` specifies the dimension of the observations in the layer input data. 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 15-73.

Check Validity of Layer Using checkLayer

Check the layer validity of the custom layer `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

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 `'ObservationDimension'` to 4.

```
layer = preluLayer(20,'prelu');
validInputSize = [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: 1.5273 seconds.
```

Here, the function does not detect any issues with the layer.

Include Layer in Network

You can use a custom layer in the same way as any other layer in Deep Learning Toolbox.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

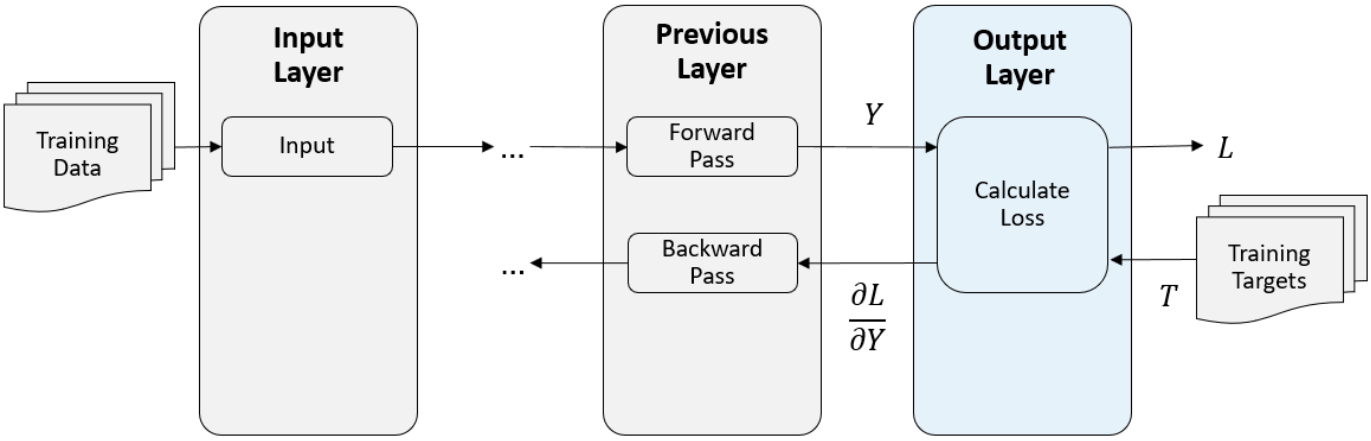
Create a layer array that includes the custom layer `preluLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    preluLayer(20,'prelu')
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Output Layer Architecture

At the end of a forward pass at training time, an output layer takes the predictions (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 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. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 `"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)`. The default value is `'auto'`.

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.

Loss Functions

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.

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 15-194. 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 15-68.

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

Regression Task	Input Size	Observation Dimension
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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

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 a custom output layer.

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.

Define a custom mean absolute error regression layer. To create this layer, save the file `maeRegressionLayer.m` in the current folder.

Load the example training data.

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

Create a layer array and include the custom 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','Verbose',false);
net = trainNetwork(XTrain,YTrain,layers,options);
```

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 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

`assembleNetwork` | `checkLayer` | `getL2Factor` | `getLearnRateFactor` | `setL2Factor` | `setLearnRateFactor`

More About

- “Check Custom Layer Validity” on page 15-73
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Regression Output Layer” on page 15-54
- “Define Custom Weighted Classification Layer” on page 15-47
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

Define Custom Deep Learning Layer with Learnable Parameters

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 1-23.

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 it can be used in MATLAB.
2. Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
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 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 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.

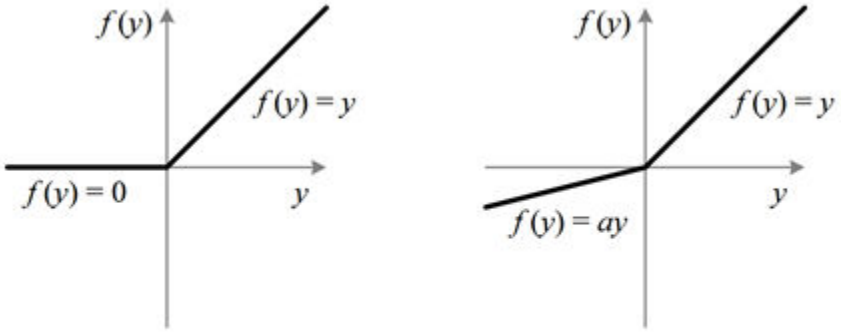


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Layer with Learnable Parameters Template

Copy the layer with learnable parameters template into a new file in MATLAB. This template outlines the structure of a layer with learnable parameters and includes the functions that define the layer behavior.

```
classdef myLayer < nnet.layer.Layer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Layer learnable parameters go here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Layer constructor function goes here.
        end

        function [Z1, …, Zm] = predict(layer, X1, …, Xn)
            % Forward input data through the layer at prediction time and
            % output the result.
            %
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function

            % Layer forward function for prediction goes here.
        end

        function [Z1, …, Zm, memory] = forward(layer, X1, …, Xn)
            % (Optional) Forward input data through the layer at training
            % time and output the result and a memory value.
            %
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function
            %         memory      - Memory value for custom backward propagation

            % Layer forward function for training goes here.
        end

        function [dLdX1, …, dLdXn, dLdW1, …, dLdWk] = ...
                backward(layer, X1, …, Xn, Z1, …, Zm, dLdZ1, …, dLdZm, memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer             - Layer to backward propagate through
            %         X1, ..., Xn       - Input data
            %         Z1, ..., Zm       - Outputs of layer forward function
            %         dLdZ1, ..., dLdZm - Gradients propagated from the next layers
            %         memory            - Memory value from forward function
            % Outputs:
            %         dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
            %                             inputs
            %         dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
```

```
            %                               learnable parameter

            % Layer backward 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 `myLayer` with `preluLayer`.

```
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:

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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.

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`	The 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`	The 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` in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` in the layer constructor. For an example, see “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28.

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 inputs 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 `preluLayer` function. Add a comment to the top of the function that explains the syntax of the function.

```
function layer = preluLayer(numChannels, name)
    % layer = preluLayer(numChannels) creates a PReLU layer with
    % numChannels channels and 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.

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` 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, name)
    % layer = preluLayer(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 `preluLayer(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 syntax for `predict` is

```
[Z1,…,Zm] = predict(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs and `Z1,…,Zm` are the `m` layer outputs. The values `n` and `m` must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` 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,…,Zm`. 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)`.

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
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

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The syntax for `forward` is

```
[Z1,…,Zm,memory] = forward(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs, `Z1,…,Zm` are the `m` layer outputs, and `memory` is the memory of the layer.

Tip If the number of inputs to `forward` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj` for `j`=1,...,`NumOutputs` and `varargout{NumOutputs+1}` corresponds to `memory`.

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 like `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 of 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 15-194.

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, name)
            % layer = preluLayer(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.

            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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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 `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

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 `'ObservationDimension'` to 4.

```
layer = preluLayer(20,'prelu');
validInputSize = [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: 1.5273 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;
```

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder. Create a layer array including the custom layer `preluLayer`.

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    batchNormalizationLayer
    preluLayer(20,'prelu')
    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 |       00:00:13 |       71.88% |       0.8378 |          0.0010 |
|       3 |         100 |       00:00:26 |       85.94% |       0.4878 |          0.0010 |
|       4 |         150 |       00:00:40 |       88.28% |       0.4068 |          0.0010 |
|       6 |         200 |       00:00:55 |       96.09% |       0.1690 |          0.0010 |
|       7 |         250 |       00:01:08 |       97.66% |       0.1368 |          0.0010 |
|       8 |         300 |       00:01:23 |       99.22% |       0.0744 |          0.0010 |
|       9 |         350 |       00:01:40 |       99.22% |       0.0592 |          0.0010 |
|      10 |         390 |       00:01:51 |      100.00% |       0.0465 |          0.0010 |
|=========================================================================================|
```

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 = 0.9188
```

References

[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

`assembleNetwork` | `checkLayer`

More About

- “Define Custom Deep Learning Layers” on page 15-2

- “Check Custom Layer Validity” on page 15-73
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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 1-23.

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 it can be used in MATLAB.
2. Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
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 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.

Layer with Learnable Parameters Template

Copy the layer with learnable parameters template into a new file in MATLAB. This template outlines the structure of a layer with learnable parameters and includes the functions that define the layer behavior.

```
classdef myLayer < nnet.layer.Layer

    properties
        % (Optional) Layer properties.

        % Layer properties go here.
    end

    properties (Learnable)
        % (Optional) Layer learnable parameters.

        % Layer learnable parameters go here.
    end

    methods
        function layer = myLayer()
            % (Optional) Create a myLayer.
            % This function must have the same name as the class.

            % Layer constructor function goes here.
        end

        function [Z1, …, Zm] = predict(layer, X1, …, Xn)
            % Forward input data through the layer at prediction time and
            % output the result.
            %
```

```
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function

            % Layer forward function for prediction goes here.
        end

        function [Z1, …, Zm, memory] = forward(layer, X1, …, Xn)
            % (Optional) Forward input data through the layer at training
            % time and output the result and a memory value.
            %
            % Inputs:
            %         layer       - Layer to forward propagate through
            %         X1, ..., Xn - Input data
            % Outputs:
            %         Z1, ..., Zm - Outputs of layer forward function
            %         memory      - Memory value for custom backward propagation

            % Layer forward function for training goes here.
        end

        function [dLdX1, …, dLdXn, dLdW1, …, dLdWk] = ...
                backward(layer, X1, …, Xn, Z1, …, Zm, dLdZ1, …, dLdZm, memory)
            % (Optional) Backward propagate the derivative of the loss
            % function through the layer.
            %
            % Inputs:
            %         layer             - Layer to backward propagate through
            %         X1, ..., Xn       - Input data
            %         Z1, ..., Zm       - Outputs of layer forward function
            %         dLdZ1, ..., dLdZm - Gradients propagated from the next layers
            %         memory            - Memory value from forward function
            % Outputs:
            %         dLdX1, ..., dLdXn - Derivatives of the loss with respect to the
            %                             inputs
            %         dLdW1, ..., dLdWk - Derivatives of the loss with respect to each
            %                             learnable parameter

            % Layer backward 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 `myLayer` with `weightedAdditionLayer`.

```
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:

Property	Description
`Name`	Layer name, specified as a character vector or a string scalar. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 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`	The 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`	The 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` in the layer constructor. If you are creating a layer with multiple outputs, then you must set either the `NumOutputs` or `OutputNames` 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 syntax for `predict` is

```
[Z1,…,Zm] = predict(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs and `Z1,…,Zm` are the `m` layer outputs. The values `n` and `m` must correspond to the `NumInputs` and `NumOutputs` properties of the layer.

Tip If the number of inputs to `predict` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.

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
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

The `forward` function propagates the data forward through the layer at *training time* and also outputs a memory value.

The syntax for `forward` is

```
[Z1,…,Zm,memory] = forward(layer,X1,…,Xn)
```

where `X1,…,Xn` are the `n` layer inputs, `Z1,…,Zm` are the `m` layer outputs, and `memory` is the memory of the layer.

Tip If the number of inputs to `forward` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj` for `j=1,...,NumOutputs` and `varargout{NumOutputs+1}` corresponds to `memory`.

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 like `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 of 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 15-194.

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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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 |       00:00:00 |       12.50% |       2.2951 |          0.0010 |
|       2 |          50 |       00:00:17 |       72.66% |       0.7880 |          0.0010 |
|       3 |         100 |       00:00:37 |       89.84% |       0.3001 |          0.0010 |
|       4 |         150 |       00:00:57 |       94.53% |       0.1555 |          0.0010 |
|       6 |         200 |       00:01:17 |       99.22% |       0.0385 |          0.0010 |
|       7 |         250 |       00:01:38 |       99.22% |       0.0361 |          0.0010 |
|       8 |         300 |       00:01:59 |      100.00% |       0.0112 |          0.0010 |
|       9 |         350 |       00:02:19 |       99.22% |       0.0249 |          0.0010 |
|      10 |         390 |       00:02:33 |      100.00% |       0.0045 |          0.0010 |
|=========================================================================================|
```

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

`analyzeNetwork` | `checkLayer` | `trainNetwork`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Check Custom Layer Validity” on page 15-73
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 `"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)`. The default value is `'auto'`.

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 15-194.

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 15-194.

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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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.

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 0.49493 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 |       00:00:01 |        9.38% |       0.9944 |          0.0100 |
|       2 |          50 |       00:00:07 |       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 |       00:00:24 |       95.31% |       0.0772 |          0.0100 |
|       7 |         250 |       00:00:30 |       97.66% |       0.0446 |          0.0100 |
|       8 |         300 |       00:00:36 |       99.22% |       0.0201 |          0.0100 |
|       9 |         350 |       00:00:42 |       99.22% |       0.0262 |          0.0100 |
|      11 |         400 |       00:00:47 |      100.00% |       0.0080 |          0.0100 |
|      12 |         450 |       00:00:53 |      100.00% |       0.0059 |          0.0100 |
|      13 |         500 |       00:00:59 |      100.00% |       0.0092 |          0.0100 |
|      15 |         550 |       00:01:05 |      100.00% |       0.0064 |          0.0100 |
|      16 |         600 |       00:01:11 |      100.00% |       0.0020 |          0.0100 |
|      17 |         650 |       00:01:16 |      100.00% |       0.0039 |          0.0100 |
|      18 |         700 |       00:01:22 |      100.00% |       0.0023 |          0.0100 |
|      20 |         750 |       00:01:28 |      100.00% |       0.0024 |          0.0100 |
|      21 |         800 |       00:01:33 |      100.00% |       0.0019 |          0.0100 |
|      22 |         850 |       00:01:38 |      100.00% |       0.0017 |          0.0100 |
|      24 |         900 |       00:01:43 |      100.00% |       0.0020 |          0.0100 |
|      25 |         950 |       00:01:49 |      100.00% |       0.0012 |          0.0100 |
|      26 |        1000 |       00:01:54 |      100.00% |       0.0011 |          0.0100 |
|      27 |        1050 |       00:02:00 |       99.22% |       0.0104 |          0.0100 |
|      29 |        1100 |       00:02:06 |      100.00% |       0.0012 |          0.0100 |
|      30 |        1150 |       00:02:12 |      100.00% |       0.0011 |          0.0100 |
|      30 |        1170 |       00:02:15 |       99.22% |       0.0079 |          0.0100 |
|=========================================================================================|
```

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

`assembleNetwork` | `checkLayer` | `classificationLayer`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Check Custom Layer Validity” on page 15-73
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

Define Custom Weighted Classification 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 and create a custom weighted classification output layer with weighted cross entropy loss. Use a weighted classification layer for classification problems with an imbalanced distribution of classes. For an example showing how to use a weighted classification layer in a network, see “Speech Command Recognition Using Deep Learning” on page 4-17.

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 weighted classification layer computes the weighted cross entropy loss for classification problems. Weighted cross entropy is an error measure between two continuous random variables. For prediction scores Y and training targets T, the weighted cross entropy loss between Y and T is given by

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} w_i T_{ni}\log(Y_{ni}),$$

where N is the number of observations, K is the number of classes, and w is a vector of weights for each class.

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 `weightedClassificationLayer`.

```
classdef weightedClassificationLayer < 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 = weightedClassificationLayer()
            ...
        end

        ...
     end
```

Save the Layer

Save the layer class file in a new file named `weightedClassificationLayer.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. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 `"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)`. The default value is `'auto'`.

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 requires an additional property to save the class weights. Specify the property `ClassWeights` in the `properties` section.

```
properties
    % Vector of weights corresponding to the classes in the training
    % data
    ClassWeights
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 input argument `classWeights` to assign to the `ClassWeights` property. Also specify an optional input argument `name` to assign to the `Name` property at creation. Add a comment to the top of the function that explains the syntaxes of the function.

```
function layer = weightedClassificationLayer(classWeights, name)
    % layer = weightedClassificationLayer(classWeights) creates a
    % weighted cross entropy loss layer. classWeights is a row
    % vector of weights corresponding to the classes in the order
    % that they appear in the training data.
```

```
        %
        % layer = weightedClassificationLayer(classWeights, name)
        % additionally 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 optional input argument `name`.

```
function layer = weightedClassificationLayer(classWeights, name)
    % layer = weightedClassificationLayer(classWeights) creates a
    % weighted cross entropy loss layer. classWeights is a row
    % vector of weights corresponding to the classes in the order
    % that they appear in the training data.
    %
    % layer = weightedClassificationLayer(classWeights, name)
    % additionally specifies the layer name.

    % Set class weights
    layer.ClassWeights = classWeights;

    % Set layer name
    if nargin == 2
        layer.Name = name;
    end

    % Set layer description
    layer.Description = 'Weighted cross entropy';
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 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

Classification Task	Input Size	Observation Dimension
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 weighted classification layer computes the weighted cross entropy loss for classification problems. Weighted cross entropy is an error measure between two continuous random variables. For prediction scores *Y* and training targets *T*, the weighted cross entropy loss between *Y* and *T* is given by

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} w_i T_{ni}\log(Y_{ni}),$$

where *N* is the number of observations, *K* is the number of classes, and *w* is a vector of weights for each class.

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 weighted cross
    % entropy loss between the predictions Y and the training
    % targets T.

    N = size(Y,4);
    Y = squeeze(Y);
    T = squeeze(T);
    W = layer.ClassWeights;

    loss = -sum(W*(T.*log(Y)))/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 15-194.

Completed Layer

View the completed classification output layer class file.

```
classdef weightedClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % Vector of weights corresponding to the classes in the training
        % data
        ClassWeights
    end
```

```
    methods
        function layer = weightedClassificationLayer(classWeights, name)
            % layer = weightedClassificationLayer(classWeights) creates a
            % weighted cross entropy loss layer. classWeights is a row
            % vector of weights corresponding to the classes in the order
            % that they appear in the training data.
            %
            % layer = weightedClassificationLayer(classWeights, name)
            % additionally specifies the layer name.

            % Set class weights
            layer.ClassWeights = classWeights;

            % Set layer name
            if nargin == 2
                layer.Name = name;
            end

            % Set layer description
            layer.Description = 'Weighted cross entropy';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the weighted cross
            % entropy loss between the predictions Y and the training
            % targets T.

            N = size(Y,4);
            Y = squeeze(Y);
            T = squeeze(T);
            W = layer.ClassWeights;

            loss = -sum(W*(T.*log(Y)))/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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

The MATLAB functions used in `forwardLoss` in `weightedClassificationLayer` all support `dlarray` objects, so the layer is GPU compatible.

Check Output Layer Validity

Check the validity of the custom classification output layer `weightedClassificationLayer`.

Define a custom weighted classification layer. To create this layer, save the file `weightedClassificationLayer.m` in the current folder.

Create an instance of the layer. Specify the class weights as a vector with three elements corresponding to three classes.

```
classWeights = [0.1 0.7 0.2];
layer = weightedClassificationLayer(classWeights);
```

Check that the layer is valid using `checkLayer`. Set the valid input size to the typical size of a single observation input to the layer. The layer expects a 1-by-1-by-*K*-by-*N* array input, where *K* is the number of classes and *N* is the number of observations in the mini-batch.

```
numClasses = numel(classWeights);
validInputSize = [1 1 numClasses];
checkLayer(layer,validInputSize,'ObservationDimension',4);
```

```
Skipping GPU tests. No compatible GPU device found.

Running nnet.checklayer.TestOutputLayerWithoutBackward
........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     8 Passed, 0 Failed, 0 Incomplete, 2 Skipped.
     Time elapsed: 2.3813 seconds.
```

The test summary reports the number of passed, failed, incomplete, and skipped tests.

See Also

`assembleNetwork` | `checkLayer` | `classificationLayer`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Check Layer Validity” on page 15-73
- “Define Custom Regression Output Layer” on page 15-54
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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. To include a layer in a layer graph, you must specify a nonempty unique layer name. If you train a series network with the

layer and `Name` is set to `''`, then the software automatically assigns a name to the layer 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 `"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)`. The default value is `'auto'`.

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 15-194.

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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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);
```

```
Running nnet.checklayer.TestOutputLayerWithoutBackward
..........
Done nnet.checklayer.TestOutputLayerWithoutBackward
__________

Test Summary:
     10 Passed, 0 Failed, 0 Incomplete, 0 Skipped.
     Time elapsed: 0.088094 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 |       00:00:08 |        14.27 |         11.3 |          0.0100 |
|       3 |         100 |       00:00:16 |        13.45 |         10.5 |          0.0100 |
|       4 |         150 |       00:00:23 |        10.35 |          8.2 |          0.0100 |
|       6 |         200 |       00:00:30 |        10.28 |          8.0 |          0.0100 |
|       7 |         250 |       00:00:36 |        10.49 |          7.9 |          0.0100 |
|       8 |         300 |       00:00:42 |         9.21 |          7.3 |          0.0100 |
|       9 |         350 |       00:00:49 |         9.18 |          7.0 |          0.0100 |
|      11 |         400 |       00:00:55 |        10.49 |          8.2 |          0.0100 |
|      12 |         450 |       00:01:01 |         8.12 |          6.1 |          0.0100 |
|      13 |         500 |       00:01:11 |         9.13 |          6.0 |          0.0100 |
|      15 |         550 |       00:01:21 |         9.85 |          7.4 |          0.0100 |
|      16 |         600 |       00:01:30 |         8.70 |          6.4 |          0.0100 |
```

```
|      17 |          650 |       00:01:37 |         8.21 |          6.1 |          0.0100 |
|      18 |          700 |       00:01:43 |         9.05 |          6.3 |          0.0100 |
|      20 |          750 |       00:01:49 |         8.05 |          6.0 |          0.0100 |
|      21 |          800 |       00:01:55 |         7.80 |          5.7 |          0.0100 |
|      22 |          850 |       00:02:01 |         6.86 |          5.4 |          0.0100 |
|      24 |          900 |       00:02:08 |         7.37 |          5.5 |          0.0100 |
|      25 |          950 |       00:02:17 |         7.00 |          5.0 |          0.0100 |
|      26 |         1000 |       00:02:24 |         6.99 |          5.0 |          0.0100 |
|      27 |         1050 |       00:02:33 |         8.29 |          6.6 |          0.0100 |
|      29 |         1100 |       00:02:43 |         8.34 |          6.8 |          0.0100 |
|      30 |         1150 |       00:02:52 |         6.26 |          4.5 |          0.0100 |
|      30 |         1170 |       00:02:55 |         6.90 |          5.1 |          0.0100 |
|=========================================================================================|
```

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

`assembleNetwork` | `checkLayer` | `regressionLayer`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Check Layer Validity” on page 15-73
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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 1-23.

The example "Define Custom Deep Learning Layer with Learnable Parameters" on page 15-17 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 it can be used in MATLAB.
2. Declare the layer properties – specify the properties of the layer and which parameters are learned during training.
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 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 15-194. 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 15-17 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 15-17. 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, name)
            % layer = preluLayer(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.

            Z = max(X,0) + layer.Alpha .* min(0,X);
        end
    end
end
```

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 syntax for `backward` is

```
[dLdX1,…,dLdXn,dLdW1,…,dLdWk] = backward(layer,X1,…,Xn,Z1,…,Zm,dLdZ1,…,dLdZm,memory)
```

where:

- `X1,…,Xn` are the `n` layer inputs
- `Z1,…,Zm` are the `m` outputs of the layer forward functions
- `dLdZ1,…,dLdZm` are the gradients backward propagated from the next layer
- `memory` is the memory output of `forward` if `forward` is defined, otherwise, `memory` is `[]`.

For the outputs, `dLdX1,…,dLdXn` are the derivatives of the loss with respect to the layer inputs and `dLdW1,…,dLdWk` are the derivatives of the loss with respect to the `k` learnable 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 `varargin{i}`

corresponds to `Xi` for `i=1,...,NumInputs`, `varargin{NumInputs+j}` and `varargin{NumInputs+NumOutputs+j}` correspond to `Zj` and `dLdZj`, respectively, for `j=1,...,NumOutputs`, and `varargin{end}` 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 `varargout{i}` corresponds to `dLdXi` for `i=1,...,NumInputs` and `varargout{NumInputs+t}` corresponds to `dLdWt` for `t=1,...,k`, where `k` is the number of learnable parameters.

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, name)
            % layer = preluLayer(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.

            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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

References

[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

`checkLayer` | `layerGraph`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Check Custom Layer Validity” on page 15-73
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Specify Custom Output Layer Backward Loss Function” on page 15-68

- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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 1-23.

The example “Define Custom Weighted Classification Layer” on page 15-47 shows how to define and create a custom weighted classification output layer with weighted cross entropy 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 15-194. 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 Weighted Classification Layer” on page 15-47 shows how to create a weighted classification layer.

A weighted classification layer computes the weighted cross entropy loss for classification problems. Weighted cross entropy is an error measure between two continuous random variables. For prediction scores Y and training targets T, the weighted cross entropy loss between Y and T is given by

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{K} w_i T_{ni}\log(Y_{ni}),$$

where N is the number of observations, K is the number of classes, and w is a vector of weights for each class.

View the layer created in the example “Define Custom Weighted Classification Layer” on page 15-47. This layer does not have a `backwardLoss` function.

```
classdef weightedClassificationLayer < nnet.layer.ClassificationLayer

    properties
```

```
        % Vector of weights corresponding to the classes in the training
        % data
        ClassWeights
    end

    methods
        function layer = weightedClassificationLayer(classWeights, name)
            % layer = weightedClassificationLayer(classWeights) creates a
            % weighted cross entropy loss layer. classWeights is a row
            % vector of weights corresponding to the classes in the order
            % that they appear in the training data.
            %
            % layer = weightedClassificationLayer(classWeights, name)
            % additionally specifies the layer name.

            % Set class weights
            layer.ClassWeights = classWeights;

            % Set layer name
            if nargin == 2
                layer.Name = name;
            end

            % Set layer description
            layer.Description = 'Weighted cross entropy';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the weighted cross
            % entropy loss between the predictions Y and the training
            % targets T.

            N = size(Y,4);
            Y = squeeze(Y);
            T = squeeze(T);
            W = layer.ClassWeights;

            loss = -sum(W*(T.*log(Y)))/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 weighted cross entropy loss with respect to the predictions Y is given by

$$\frac{\delta L}{\delta Y_i} = -\frac{1}{N}\frac{w_i T_i}{Y_i},$$

where N is the number of observations and w is a vector of weights for each class.

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 weighted cross entropy loss with respect to the
    % predictions Y.

    [~,~,K,N] = size(Y);
    Y = squeeze(Y);
    T = squeeze(T);
    W = layer.ClassWeights;

    dLdY = -(W'.*T./Y)/N;
    dLdY = reshape(dLdY,[1 1 K N]);
end
```

Complete Layer

View the completed layer class file.

```
classdef weightedClassificationLayer < nnet.layer.ClassificationLayer

    properties
        % Vector of weights corresponding to the classes in the training
        % data
        ClassWeights
    end

    methods
        function layer = weightedClassificationLayer(classWeights, name)
            % layer = weightedClassificationLayer(classWeights) creates a
            % weighted cross entropy loss layer. classWeights is a row
            % vector of weights corresponding to the classes in the order
            % that they appear in the training data.
            %
            % layer = weightedClassificationLayer(classWeights, name)
            % additionally specifies the layer name.

            % Set class weights
            layer.ClassWeights = classWeights;

            % Set layer name
            if nargin == 2
                layer.Name = name;
            end

            % Set layer description
            layer.Description = 'Weighted cross entropy';
        end

        function loss = forwardLoss(layer, Y, T)
            % loss = forwardLoss(layer, Y, T) returns the weighted cross
            % entropy loss between the predictions Y and the training
            % targets T.

            N = size(Y,4);
```

```
            Y = squeeze(Y);
            T = squeeze(T);
            W = layer.ClassWeights;

            loss = -sum(W*(T.*log(Y)))/N;
        end

            function dLdY = backwardLoss(layer, Y, T)
            % dLdY = backwardLoss(layer, Y, T) returns the derivatives of
            % the weighted cross entropy loss with respect to the
            % predictions Y.

            [~,~,K,N] = size(Y);
            Y = squeeze(Y);
            T = squeeze(T);
            W = layer.ClassWeights;

            dLdY = -(W'.*T./Y)/N;
            dLdY = reshape(dLdY,[1 1 K 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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. For more information on working with GPUs in MATLAB, see “GPU Computing in MATLAB” (Parallel Computing Toolbox).

See Also

`checkLayer` | `layerGraph`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Check Custom Layer Validity” on page 15-73
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Specify Custom Layer Backward Function” on page 15-62
- “List of Deep Learning Layers” on page 1-23

- “Deep Learning Tips and Tricks” on page 1-45

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, and correctly defined gradients. To check that a layer is valid, run the following command:

```
checkLayer(layer,validInputSize,'ObservationDimension',dim)
```

where `layer` is an instance of the layer, `validInputSize` is a vector or cell array specifying the valid input sizes to the layer, and `dim` specifies the dimension of the observations in the layer input data. For large input sizes, the gradient checks take longer to run. To speed up the tests, specify a smaller valid input size.

Check Layer Validity

Check the validity of the example custom layer `preluLayer`.

Define a custom PReLU layer. To create this layer, save the file `preluLayer.m` in the current folder.

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,'prelu');
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.

Running nnet.checklayer.TestLayerWithoutBackward
.........
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     9 Passed, 0 Failed, 0 Incomplete, 8 Skipped.
     Time elapsed: 1.0705 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 `'ObservationDimension'` to 4.

```
layer = preluLayer(20,'prelu');
validInputSize = [5 5 20];
checkLayer(layer,validInputSize,'ObservationDimension',4)
```

```
Skipping GPU tests. No compatible GPU device found.

Running nnet.checklayer.TestLayerWithoutBackward
.......... ...
Done nnet.checklayer.TestLayerWithoutBackward
__________

Test Summary:
     13 Passed, 0 Failed, 0 Incomplete, 4 Skipped.
     Time elapsed: 0.43429 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` does not error.
`forwardDoesNotError`	When specified, `forward` does not error.
`forwardPredictAreConsistentInSize`	When `forward` is specified, `forward` and `predict` output values of the same size.
`backwardDoesNotError`	When specified, `backward` does not error.
`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.

Test	Description
`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.

The tests `predictIsConsistentInType`, `forwardIsConsistentInType`, and `backwardIsConsistentInType` 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`.

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'` name-value pair. 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 syntax for `predict` is `[Z1,…,Zm] = predict(layer,X1,…,Xn)` where `X1,…,Xn` are the `n` layer inputs and `Z1,…,Zm` are the `m` layer outputs. The values `n` and `m` must correspond to the `NumInputs` and `NumOutputs` properties of the layer. **Tip** If the number of inputs to `predict` can vary, then use `varargin` instead of `X1,…,Xn`.

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.	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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj`.
`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 syntax for `forward` is `[Z1,…,Zm,memory] = forward(layer,X1,…,X` where `X1,…,Xn` are the `n` layer inputs, `Z1,…,Zm` are the `m` layer outputs, and `memory` is the memory of the layer. **Tip** If the number of inputs to `forward` 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,…,Zm`. In this case, `varargout` is a cell array of the outputs, where `varargout{j}` corresponds to `Zj` for `j`=1,...,`NumOutputs` and `varargout{NumOutputs+1}` corresponds to `memory`.
`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.	
`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 syntax for `backward` is `[dLdX1,…,dLdXn,dLdW1,…,dLdWk] = backward(layer,X1` where: • `X1,…,Xn` are the `n` layer inputs

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.	• `Z1,…,Zm` are the `m` outputs of the layer forward functions • `dLdZ1,…,dLdZm` are the gradients backward propagated from the next layer • `memory` is the memory output of `forward` if `forward` is defined, otherwise, `memory` is `[]`. For the outputs, `dLdX1, …,dLdXn` are the derivatives of the loss with respect to the layer inputs and `dLdW1, …,dLdWk` are the derivatives of the loss with respect to the `k` learnable 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 `varargin{i}` corresponds to `Xi` for `i=1,...,NumInputs`, `varargin{NumInputs+j}` and `varargin{NumInputs +NumOutputs+j}` correspond to `Zj` and `dLdZj`, respectively, for `j=1,...,NumOutputs`, and `varargin{end}` 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 `varargout{i}` corresponds to `dLdXi` for `i=1, ...,NumInputs` and

Test Diagnostic	Description	Possible Solution
		`varargout{NumInputs+t}` corresponds to `dLdWt` for `t`=1, ...,`k`, where `k` is the number of learnable parameters. **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 15-194.

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'` name-value pair. 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 15-194.
`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 15-194.
`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 syntax for `backward` is

```
[dLdX1,…,dLdXn,dLdW1,…,dLdWk] = backward(layer,X1,…,Xn,Z1,…,Zm,dLdZ1,…,dLdZm,memory)
```

where:

- `X1,…,Xn` are the `n` layer inputs
- `Z1,…,Zm` are the `m` outputs of the layer forward functions
- `dLdZ1,…,dLdZm` are the gradients backward propagated from the next layer
- `memory` is the memory output of `forward` if `forward` is defined, otherwise, `memory` is `[]`.

For the outputs, `dLdX1,…,dLdXn` are the derivatives of the loss with respect to the layer inputs and `dLdW1,…,dLdWk` are the derivatives of the loss with respect to the `k` learnable 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 `varargin{i}` corresponds to `Xi` for `i=1,...,NumInputs`, `varargin{NumInputs+j}` and `varargin{NumInputs +NumOutputs+j}` correspond to `Zj` and `dLdZj`, respectively, for `j=1,...,NumOutputs`, and `varargin{end}` 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 `varargout{i}` corresponds to `dLdXi` for `i=1, ...,NumInputs` and `varargout{NumInputs+t}` corresponds to `dLdWt` for `t=1,...,k`, where `k` is the number of learnable parameters.

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`.

Test Diagnostic	Description	Possible Solution
`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 15-194.

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 15-194.

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 15-194.

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 15-194. 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 CUDA enabled NVIDIA GPU with compute capability 3.0 or higher. 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 like `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 of 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 15-194.

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 15-194.

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 15-194. 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 15-118. Alternatively, define a custom backward function by creating a function named `backward`. To learn more, see “Backward Function” on page 15-9.
`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 15-194.
`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 15-194.

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 15-194. 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 15-118. Alternatively, define a custom backward loss function by creating a function named `backwardLoss`. To learn more, see “Loss Functions” on page 15-12.
`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 15-194.

See Also

`analyzeNetwork` | `checkLayer`

More About

- “Define Custom Deep Learning Layers” on page 15-2
- “Define Custom Deep Learning Layer with Learnable Parameters” on page 15-17
- “Define Custom Deep Learning Layer with Multiple Inputs” on page 15-28
- “Define Custom Classification Output Layer” on page 15-39
- “Define Custom Weighted Classification Layer” on page 15-47
- “Define Custom Regression Output Layer” on page 15-54
- “Specify Custom Layer Backward Function” on page 15-62
- “Specify Custom Output Layer Backward Loss Function” on page 15-68
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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)
```

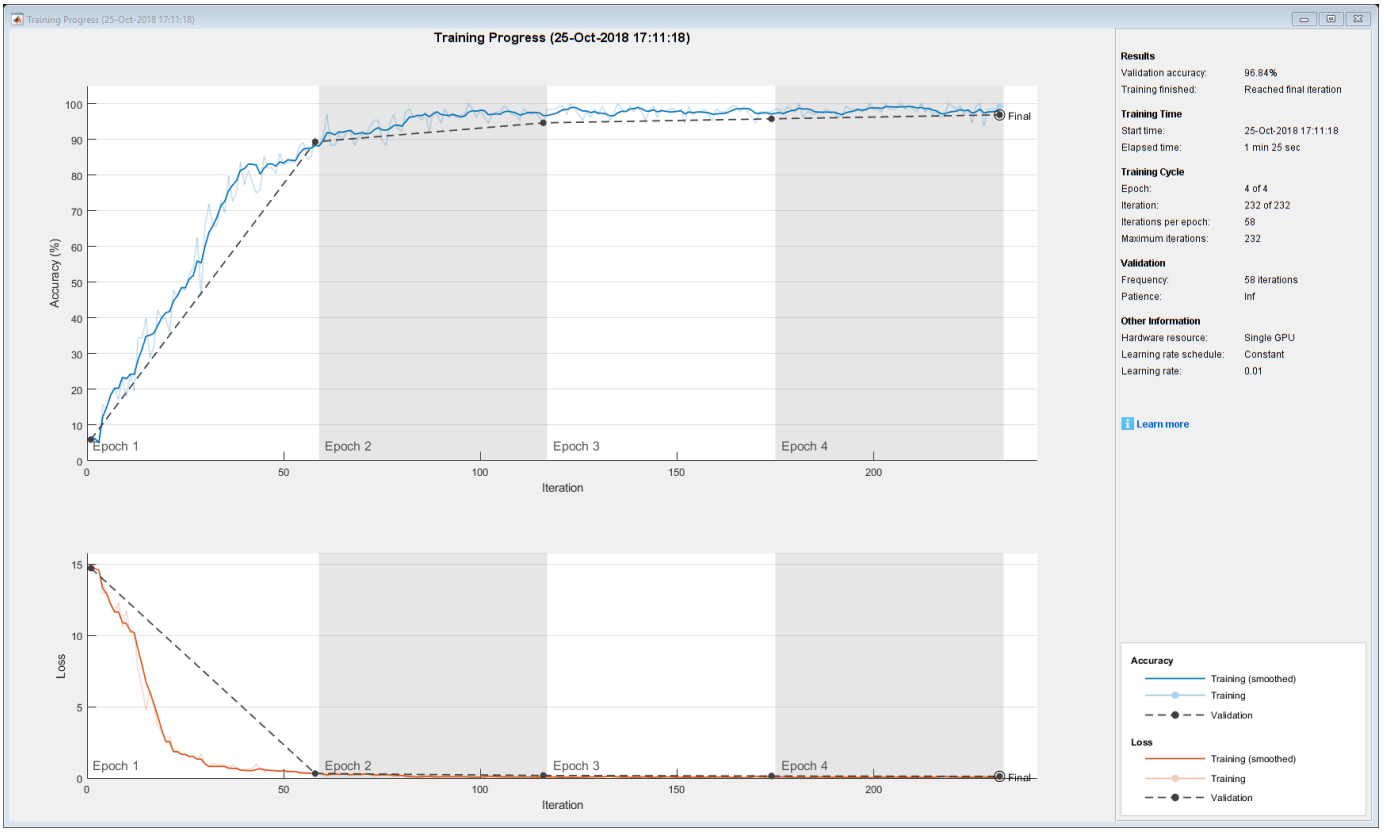
```
    leakyReluLayer
    convolution2dLayer(3,16,'WeightsInitializer',@leakyHe)
    leakyReluLayer
    convolution2dLayer(3,32,'WeightsInitializer',@leakyHe)
    leakyReluLayer
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
```

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 (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. 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

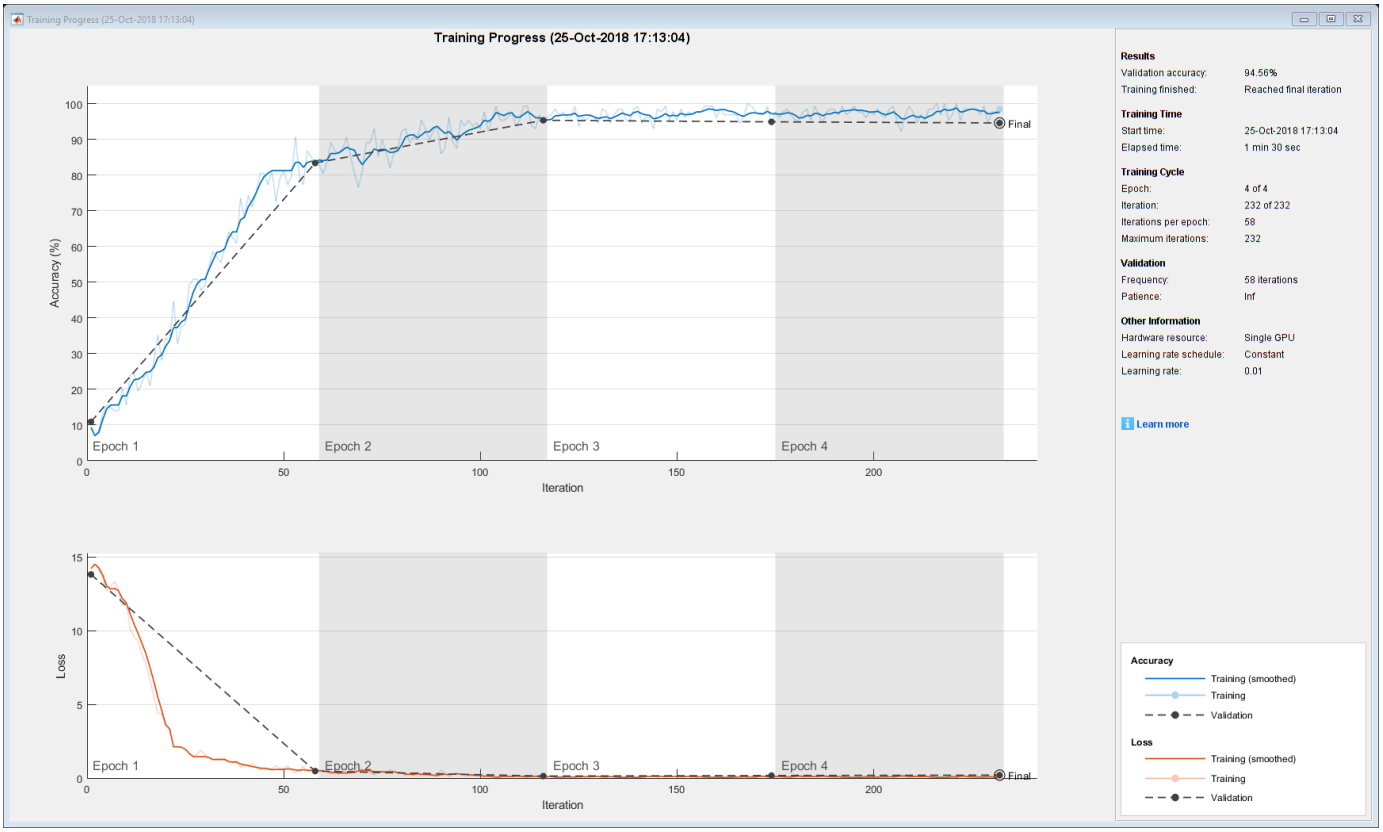
The `leakyHe` function accepts the optional input argument `scale`. To input extra variables into the custom weight initialization function, specify the function as an anonymous function that accepts a single input `sz`. To do this, replace instances of `@leakyHe` with `@(sz) leakyHe(sz,scale)`. Here, 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.

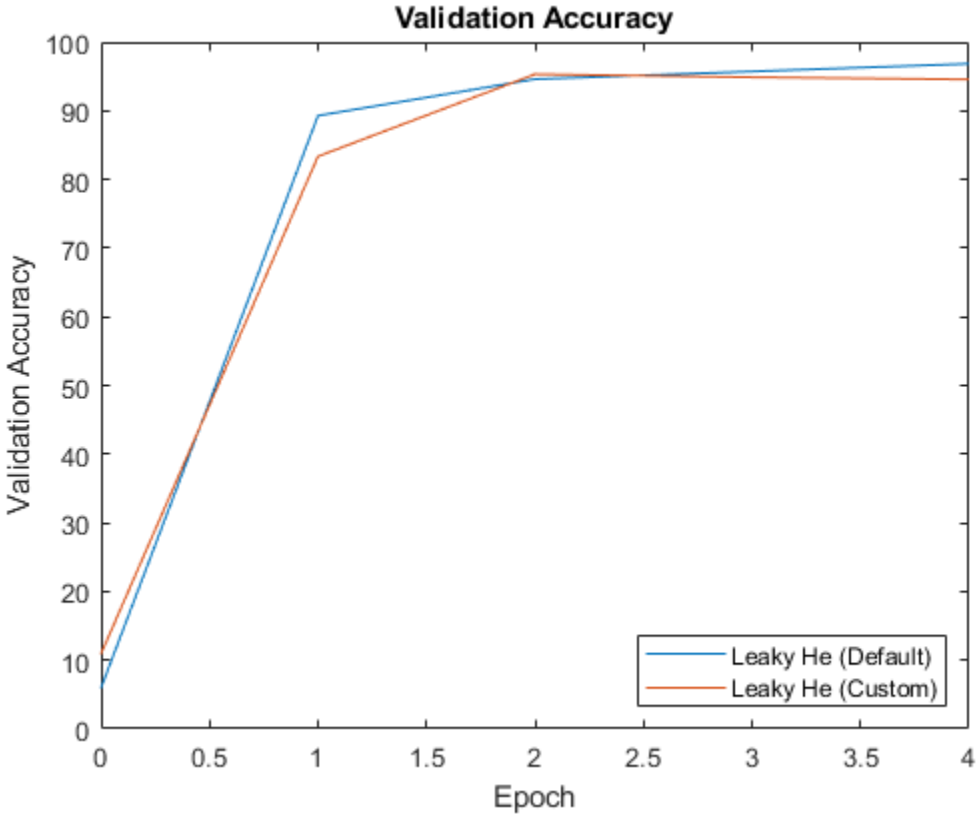
```
validationAccuracy = [
    infoDefault.ValidationAccuracy;
    infoCustom.ValidationAccuracy];
```

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 15-95
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “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');
```

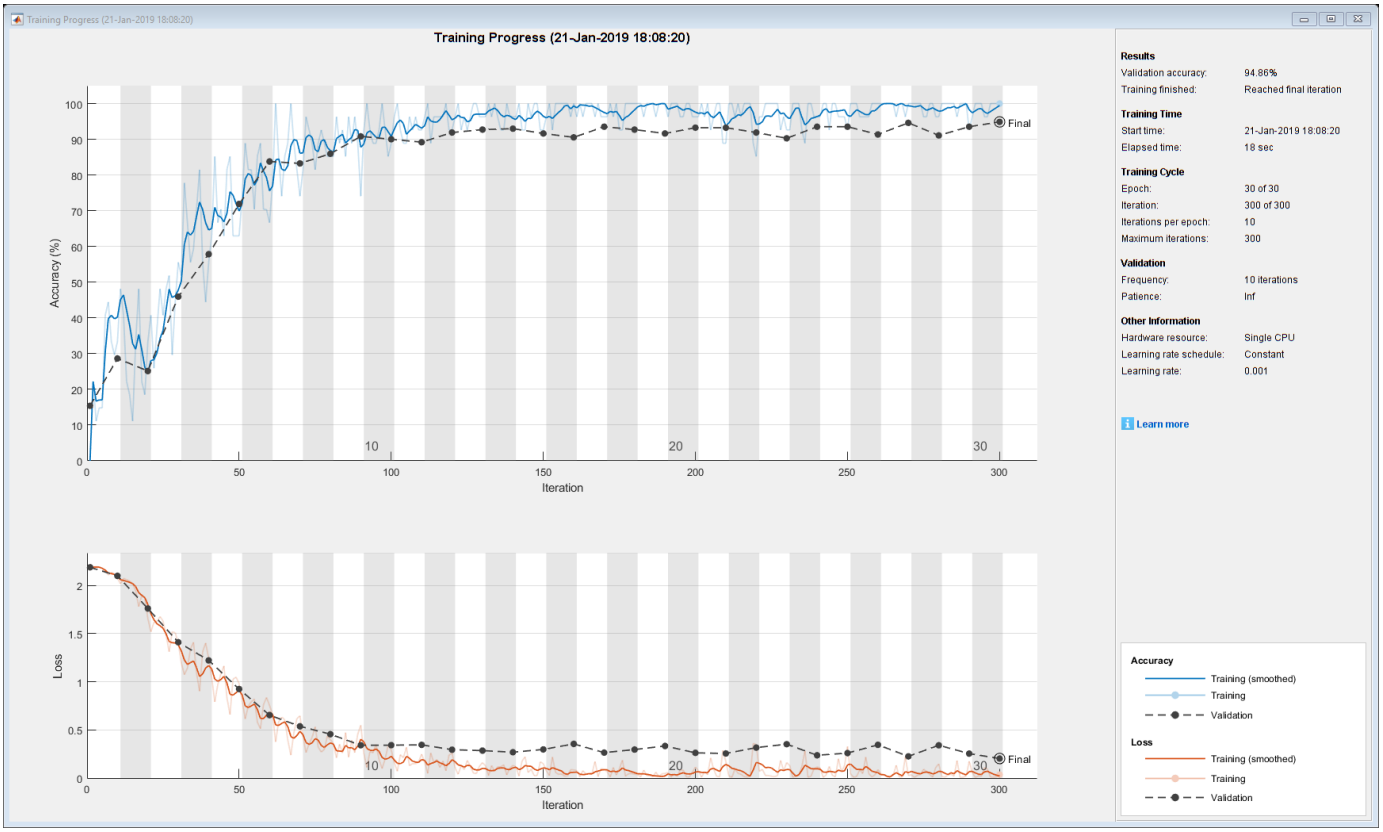
Glorot Initializer

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);
```



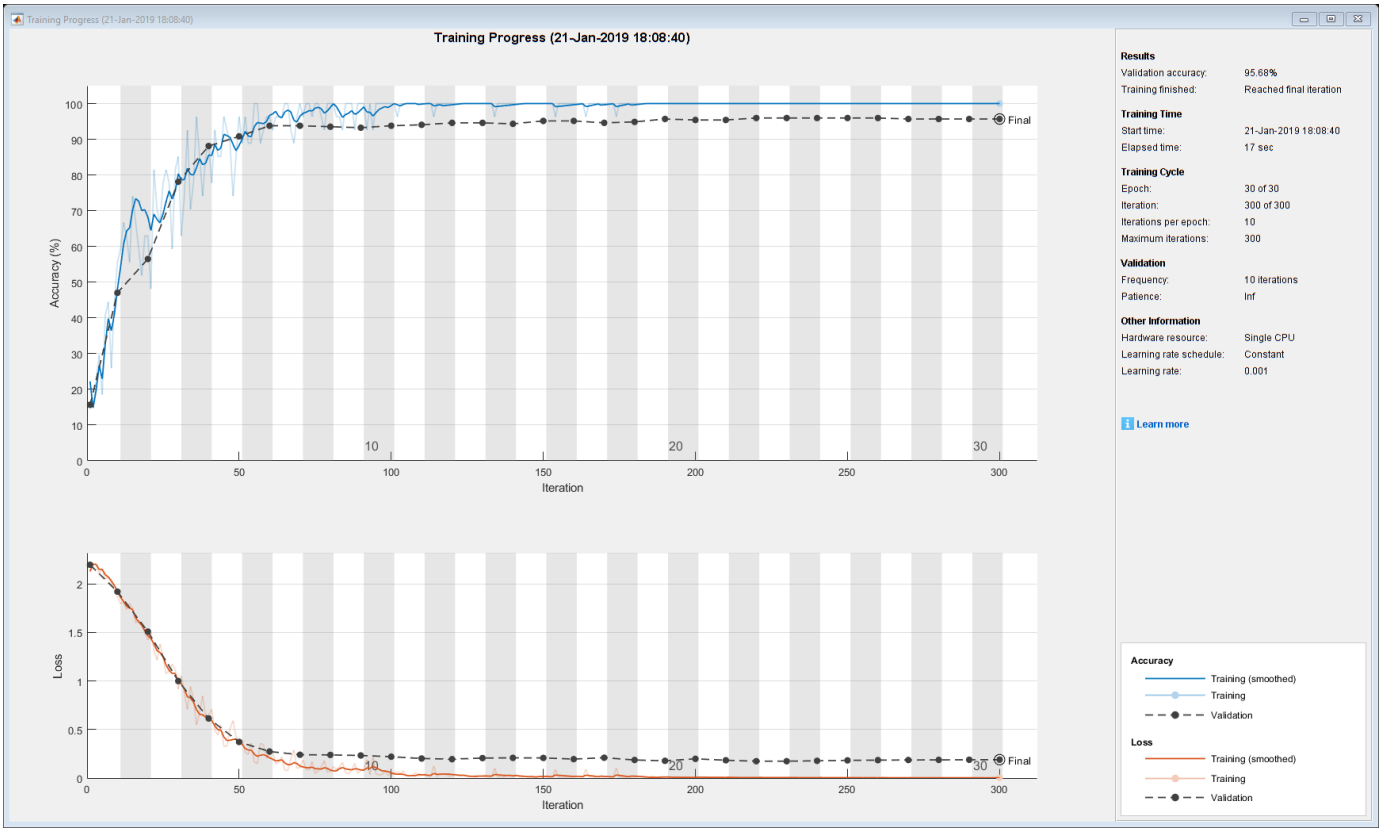
He Initializer

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);
```



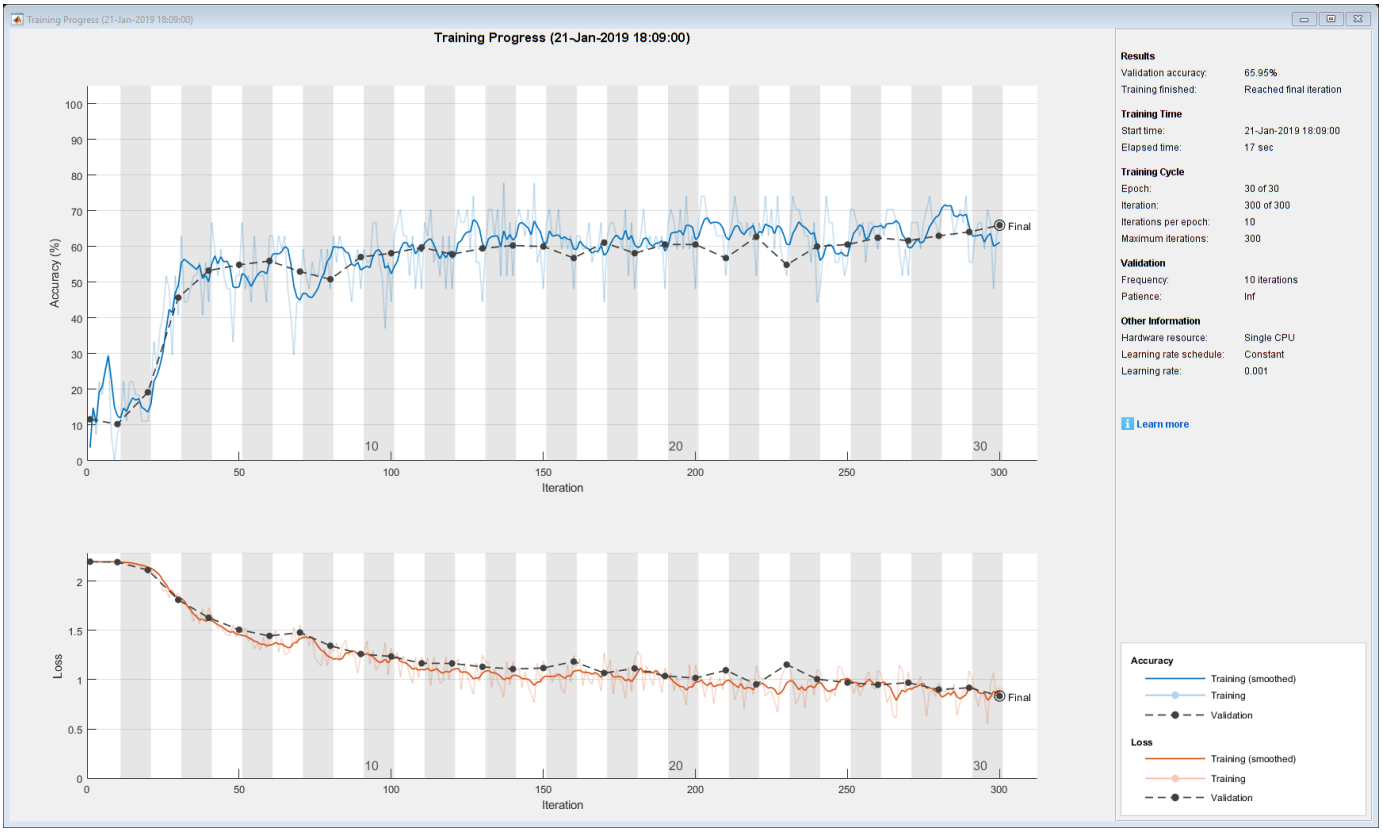
Narrow-Normal Initializer

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.

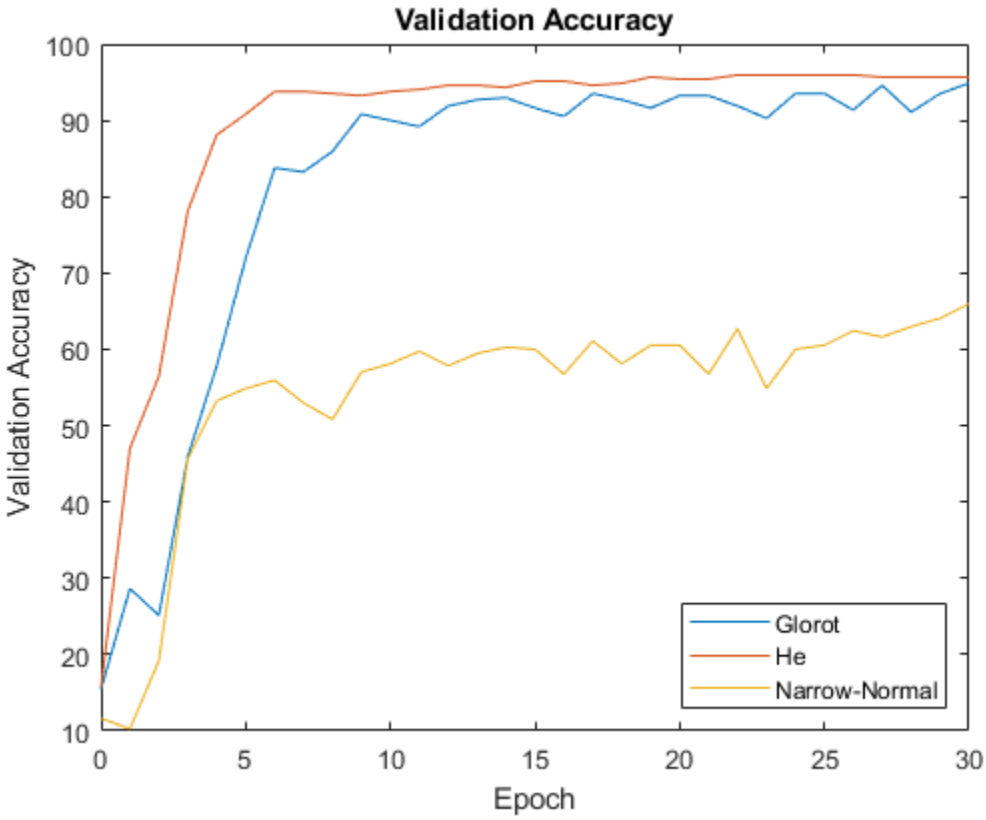
```
validationAccuracy = [
    infoGlorot.ValidationAccuracy;
    infoHe.ValidationAccuracy;
    infoNarrowNormal.ValidationAccuracy];
```

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 15-89
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “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.

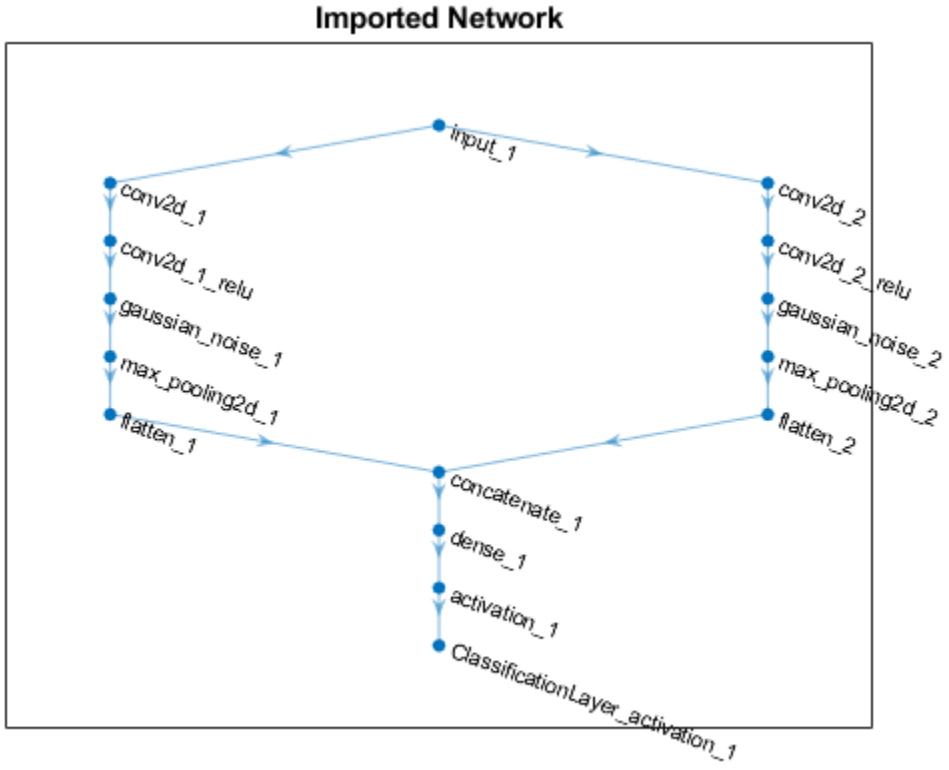
```
filename = 'digitsDAGnetwithnoise.h5';
lgraph = importKerasLayers(filename,'ImportWeights',true);
```

```
Warning: Unable to import some Keras layers, because they are not yet supported by the Deep Learn
```

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.

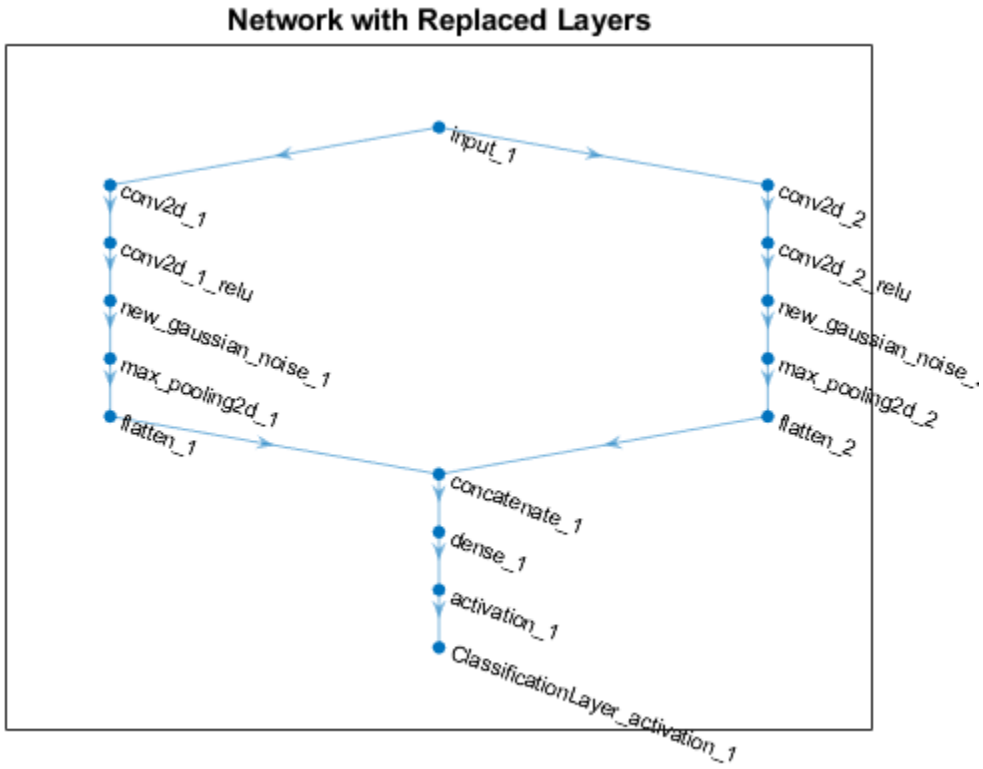
```
gnLayer1 = gaussianNoiseLayer(1.5,'new_gaussian_noise_1');
gnLayer2 = gaussianNoiseLayer(0.7,'new_gaussian_noise_2');
```

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'
      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]
      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: [15×1 nnet.cnn.layer.Layer]
    Connections: [15×2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_activation_1'}
```

See Also

`DAGNetwork` | `assembleNetwork` | `findPlaceholderLayers` | `importKerasLayers` | `importKerasNetwork` | `layerGraph` | `replaceLayer` | `trainNetwork`

Related Examples

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

- “Pretrained Deep Neural Networks” on page 1-12
- “Define Custom Deep Learning Layers” on page 15-2

Assemble Multiple-Output Network for Prediction

Instead of using the model function for prediction, you can assemble the network into a `DAGNetwork` ready for prediction using the `functionToLayerGraph` and `assembleNetwork` functions. This lets you use the `predict` function.

Load Model Function and 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.

Assemble Network for Prediction

Define an anonymous function with a fixed set of model parameters, the model state, and set the `doTraining` option to `false`.

```
doTraining = false;
fun = @(dlX) model(dlX,parameters,doTraining,state);
```

Convert the model function to a layer graph using the `functionToLayerGraph` function. Create a variable `dlX` that contains a mini-batch of data with the expected format.

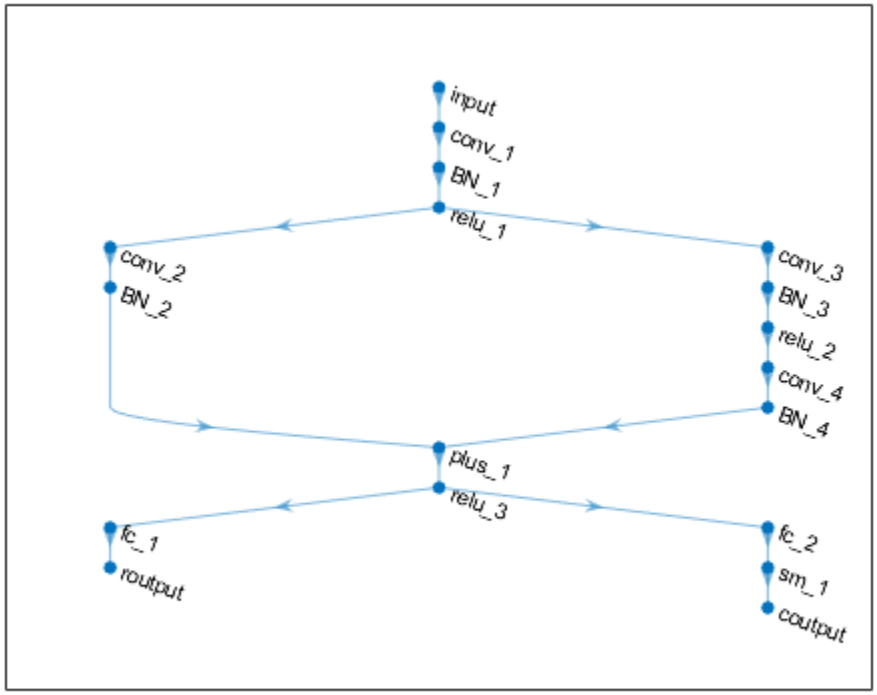
```
X = rand(28,28,1,128,'single');
dlX = dlarray(X,'SSCB');
lgraph = functionToLayerGraph(fun,dlX);
```

The layer graph output by the `functionToLayerGraph` function does not include input and output layers. Add an input layer, a classification layer, and a regression layer to the layer graph using the `addLayers` and `connectLayers` functions.

```
layers = imageInputLayer([28 28 1],'Name','input','Normalization','none');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'input','conv_1');
layers = classificationLayer('Classes',classNames,'Name','coutput');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'sm_1','coutput');
layers = regressionLayer('Name','routput');
lgraph = addLayers(lgraph,layers);
lgraph = connectLayers(lgraph,'fc_1','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: [18×1 nnet.cnn.layer.Layer]
    Connections: [18×2 table]
     InputNames: {'input'}
    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 = 0.9644
```

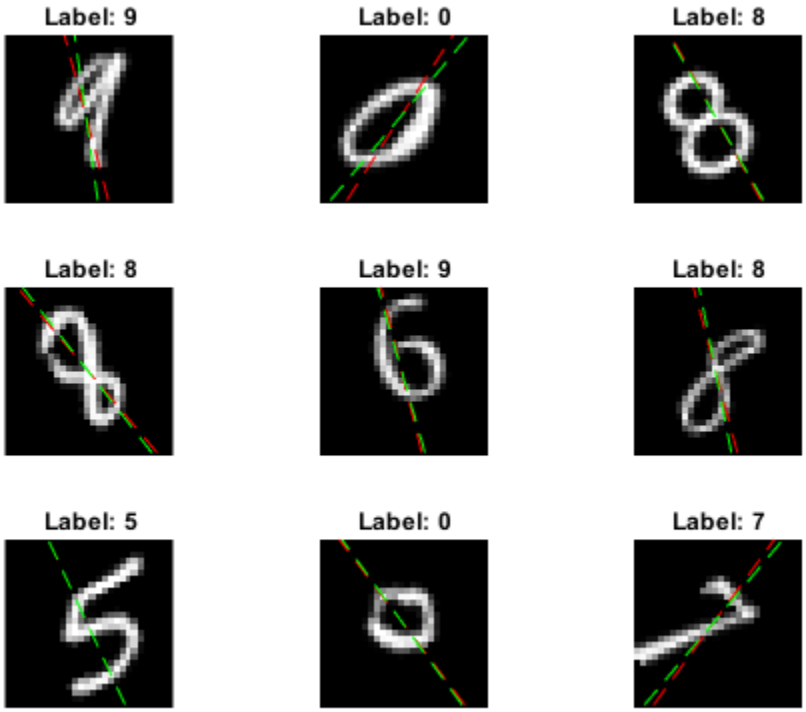
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean((Y2Pred - Y2Test).^2))
```

```
angleRMSE = single
    5.8081
```

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
```



Model Function

The function model takes the input data dlX, the model parameters `parameters`, 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(dlX,parameters,doTraining,state)

% Convolution
W = parameters.conv1.Weights;
B = parameters.conv1.Bias;
dlY = dlconv(dlX,W,B,'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,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)
W = parameters.convSkip.Weights;
B = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,W,B,'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
W = parameters.conv2.Weights;
B = parameters.conv2.Bias;
dlY = dlconv(dlY,W,B,'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,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,Offset,Scale,trainedMean,trainedVariance);
end
dlY = relu(dlY);

% Convolution
W = parameters.conv3.Weights;
B = parameters.conv3.Bias;
dlY = dlconv(dlY,W,B,'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,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 (angles)
W = parameters.fc1.Weights;
B = parameters.fc1.Bias;
dlY2 = fullyconnect(dlY,W,B);

% Fully connect, softmax (labels)
W = parameters.fc2.Weights;
B = parameters.fc2.Bias;
dlY1 = fullyconnect(dlY,W,B);
dlY1 = softmax(dlY1);

end
```

See Also

`assembleNetwork` | `batchnorm` | `dlarray` | `dlconv` | `fullyconnect` | `functionToLayerGraph` | `predict` | `relu` | `softmax`

More About

- “Multiple-Input and Multiple-Output Networks” on page 1-21
- “Make Predictions Using Model Function” on page 15-173
- “Train Network with Multiple Outputs” on page 3-54
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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 15-121.

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.

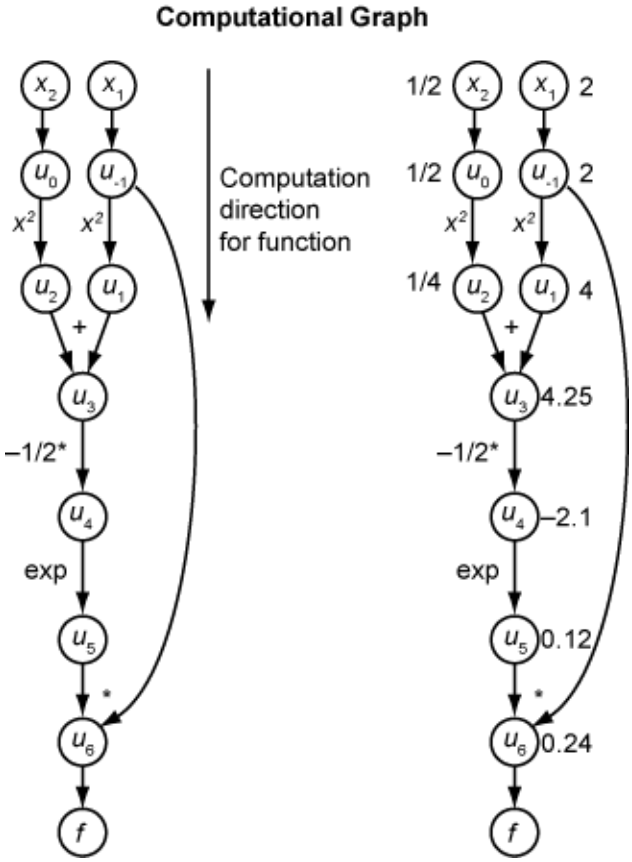
Forward Mode

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 $f(x)$.



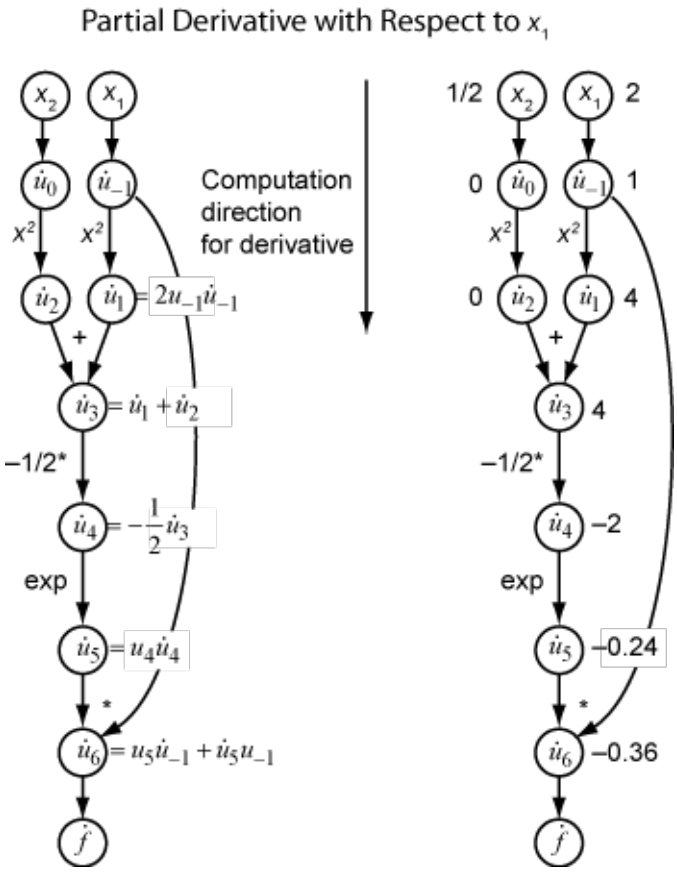
To compute the gradient of $f(x)$ using forward mode, you compute the same graph in the same direction, but modify the computation based on the elementary rules of differentiation. To further simplify the calculation, you fill in the value of the derivative of each subexpression u_i as you go. To compute 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

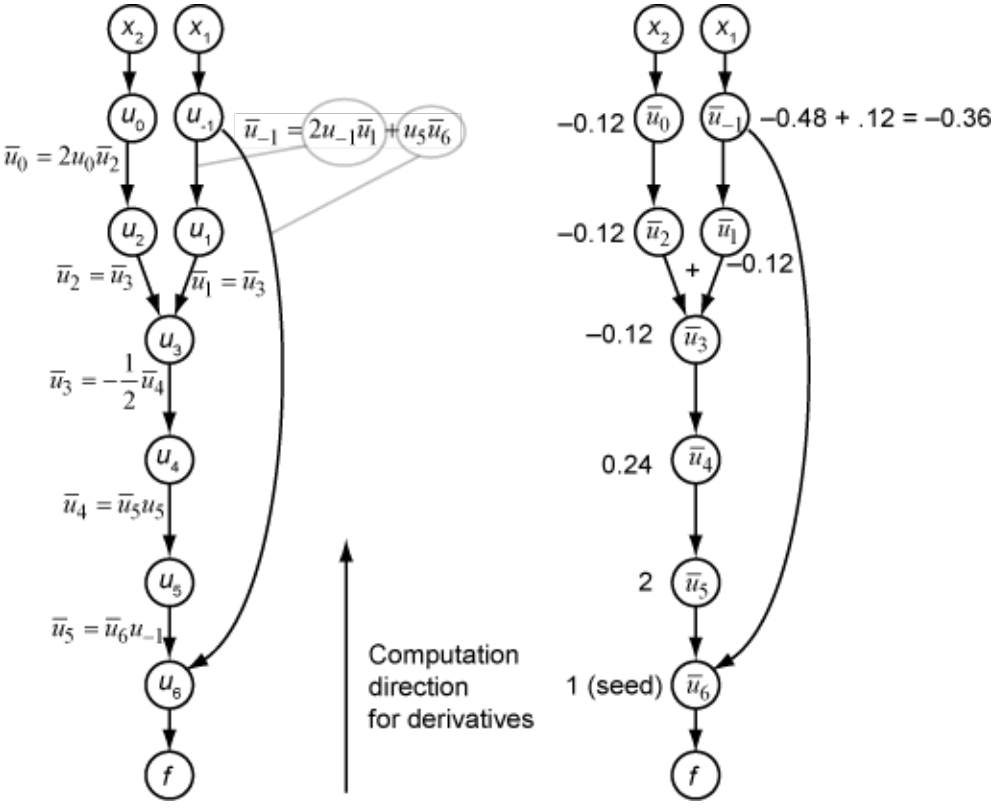
$$\bar{u}_{-1} = \bar{u}_1 2u_{-1} + \bar{u}_6u_5.$$

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` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Use Automatic Differentiation In Deep Learning Toolbox” on page 15-117
- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

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 15-194.

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 simple `dlfeval`-`dlgradient`-`dlarray` syntax, see "Grad-CAM Reveals the Why Behind Deep Learning Decisions" on page 5-8. For a more complex example using a custom training loop, see "Train Generative Adversarial Network (GAN)" on page 3-72. For further details on custom training using automatic differentiation, see "Define Custom Training Loops, Loss Functions, and Networks" on page 15-121.

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 “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8.

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 15-112. 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.

- Use only supported functions. See “List of Functions with dlarray Support” on page 15-194. 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
```

- Currently, `dlarray` does not allow higher order derivatives. In other words, you cannot calculate a second derivative by calling `dlgradient` twice.

See Also

`dlarray` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Automatic Differentiation Background” on page 15-112
- “List of Functions with dlarray Support” on page 15-194

- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Grad-CAM Reveals the Why Behind Deep Learning Decisions” on page 5-8
- “Train Generative Adversarial Network (GAN)” on page 3-72

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 Custom Training Loops” on page 15-121.

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 15-2. 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 15-122. For networks that cannot be created using layer graphs, you can define custom networks as a function. To learn more, see “Define Custom Networks” on page 15-122.

Custom training loops, loss functions, and networks use automatic differentiation to automatically compute the model gradients. To learn more, see “Automatic Differentiation Background” on page 15-112.

Define Custom Training Loops

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 learn rate schedule), then you can define your own custom training loop using automatic differentiation.

For an example showing how to train a network with a custom learn rate schedule, see “Train Network Using Custom Training Loop” on page 15-134.

Update Learnable Parameters Using Automatic Differentiation

To update the learnable parameters, you must first calculate the gradients of the loss with respect to the learnable parameters.

Create a function of the form `gradients = modelGradients(dlnet,dlX,dlT)`, where `dlnet` is the network, `dlX` are the input predictors, `dlT` are the targets, and `gradients` are 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 models defined as a function, you do not need to pass a network as an input argument.

To use automatic differentiation, you call `dlgradient` to compute the gradient of the function, and `dlfeval` to set up or update the computational graph. These functions use a `dlarray` to manage the data structures and enable tracing of evaluation.

To update the network weights, 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 an example showing how to create a model gradients function to train a generative adversarial network (GAN) that generates images, see “Train Generative Adversarial Network (GAN)” on page 3-72.

Specify Loss Functions

When using `dlnetwork` objects, do not use output layers, instead you must calculate the loss manually in the model gradients function. You can use the following functions to compute the loss:

Function	Description
`softmax`	The softmax activation operation applies the softmax function to the channel dimension of the input data.
`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.
`mse`	The half mean squared error operation computes the half mean squared error loss between network predictions and target values for regression tasks.

Alternatively, you can use custom loss function by creating a function of the form `loss = myLoss(Y,T)`, where `Y` is the network predictions, `T` are the targets, and `loss` is the returned loss.

Use the loss value when computing gradients for updating the network weights.

For an example showing how to create a model gradients function to train a generative adversarial network (GAN) that generates images using a custom loss function, see “Train Generative Adversarial Network (GAN)” on page 3-72.

Define Custom Networks

For most tasks, you can use a pretrained network or define your own network as a layer graph. To learn more about pretrained networks, see “Pretrained Deep Neural Networks” on page 1-12. For a list of layers supported by `dlnetwork` objects, see “Supported Layers”.

For architectures that cannot be created using layer graphs, you can define a custom model as a function of the form `[dlY1,...,dlYM] = model(dlX1,...,dlXN,parameters)`, where `dlX1,...,dlXN` correspond to the input data for the `N` model inputs, `parameters` contains the network parameters, and `dlY1,...,dlYM` correspond to the `M` model outputs. To train a custom network, use a custom training loop

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 15-194.

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 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 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.
`dlconv`	The convolution operation applies sliding filters to the input data. Use 1-D and 2-D filters with ungrouped or grouped convolutions and 3-D filters with ungrouped convolutions.
`dltranspconv`	The transposed convolution operation upsamples feature maps.
`fullyconnect`	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
`gru`	The gated recurrent unit (GRU) operation allows a network to learn dependencies between time steps in time series and sequence data.
`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.

Function	Description
`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.
`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.

See Also

`dlarray` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Train Network Using Custom Training Loop” on page 15-134
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Functions with dlarray Support” on page 15-194
- “Automatic Differentiation Background” on page 15-112

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 learn rate schedule), then you can define your own custom training loop 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 15-126
RMSProp solver	• `solverName` • `'Epsilon'`	"Root Mean Square Propagation (RMSProp)" on page 15-126
SGDM solver	• `solverName` • `'Momentum'`	"Stochastic Gradient Descent with Momentum (SGDM)" on page 15-126
Learn rate	`'InitialLearnRate'`	"Learn Rate" on page 15-126
Learn rate schedule	• `'LearnRateSchedule'` • `'LearnRateDropPeriod'` • `'LearnRateDropFactor'`	"Piecewise Learn Rate Schedule" on page 15-126
Training progress	`'Plots'`	"Plots" on page 15-127
Verbose output	• `'Verbose'` • `'VerboseFrequency'`	"Verbose Output" on page 15-128
Mini-batch size	`'MiniBatchSize'`	"Mini-Batch Size" on page 15-129
Number of epochs	`'MaxEpochs'`	"Number of Epochs" on page 15-129
Validation	• `ValidationData` • `'ValidationPatience'`	"Validation" on page 15-129
L_2 regularization	`'L2Regularization'`	"L2 Regularization" on page 15-131
Gradient clipping	• `'GradientThreshold'` • `'GradientThresholdMethod'`	"Gradient Clipping" on page 15-131
Single CPU or GPU training	`'ExecutionEnvironment'`	"Single CPU or GPU Training" on page 15-132
Checkpoints	`'CheckpointPath'`	"Checkpoints" on page 15-132

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 15-129.

```
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(1,2,1)
        lineAccuracyValidation = animatedline;

        subplot(1,2,2)
        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 15-129.

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 15-129.

```
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 15-129.

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. Train on a single GPU, convert the data to `gpuArray` objects. Using a GPU requires Parallel Computing Toolbox and a CUDA enabled NVIDIA GPU with compute capability 3.0 or higher.

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` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `dlupdate` | `rmspropupdate` | `sgdmupdate`

More About

- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Train Network Using Custom Training Loop” on page 15-134
- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Automatic Differentiation Background” on page 15-112

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.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

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'}
```

Define Model Gradients Function

Create the function `modelGradients`, listed at the end of the example, that 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` and the corresponding loss.

Specify Training Options

Train with a minibatch size of 128 for 5 epochs.

```
numEpochs = 5;
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;
```

Visualize 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 CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

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 epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- 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 `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
```

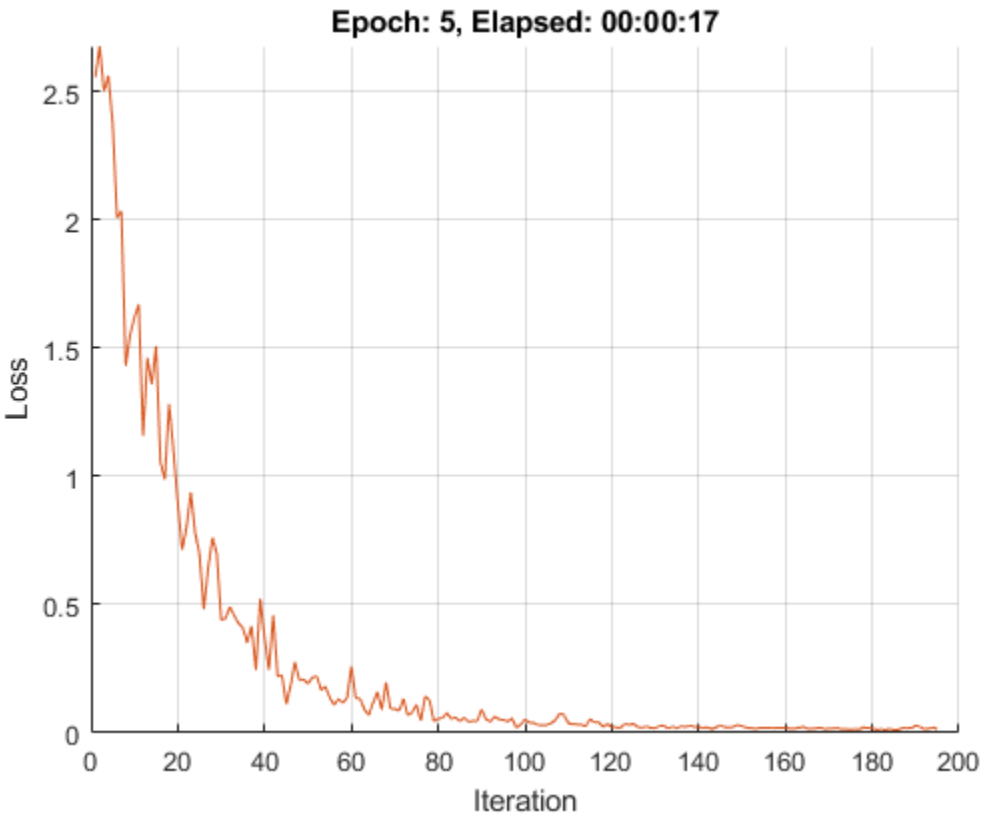
Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Train the network.

```
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);

iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and convert the labels to dummy
        % variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);

        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c,YTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(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, state, and loss using dlfeval and the
        % modelGradients function and update the network state.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);
        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.
        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.

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

Convert the data to a `dlarray` object with dimension format `'SSCB'`. For GPU prediction, also convert the data to `gpuArray`.

```
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

Classify the images using `modelPredictions` function, listed at the end of the example and find the classes with the highest scores.

```
dlYPred = modelPredictions(dlnet,dlXTest,miniBatchSize);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.9910
```

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);

end
```

Model Predictions Function

The `modelPredictions` function takes a `dlnetwork` object `dlnet`, an array of input data `dlX`, and a mini-batch size, and outputs the model predictions by iterating over mini-batches of the specified size.

```
function dlYPred = modelPredictions(dlnet,dlX,miniBatchSize)

numObservations = size(dlX,4);
numIterations = ceil(numObservations / miniBatchSize);

numClasses = dlnet.Layers(11).OutputSize;
dlYPred = zeros(numClasses,numObservations,'like',dlX);

for i = 1:numIterations
    idx = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);

    dlYPred(:,idx) = predict(dlnet,dlX(:,:,:,idx));
end

end
```

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `forward` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Update Batch Normalization Statistics in Custom Training Loop” on page 15-140
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23

- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

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

Load the digits data.

```
[XTrain,YTrain] = digitTrain4DArrayData;
classes = categories(YTrain);
numClasses = numel(classes);
```

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 with a mini-batch size of 128 for 5 epochs. 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 on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

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 epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the data to `gpuArray` objects.
- 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
```

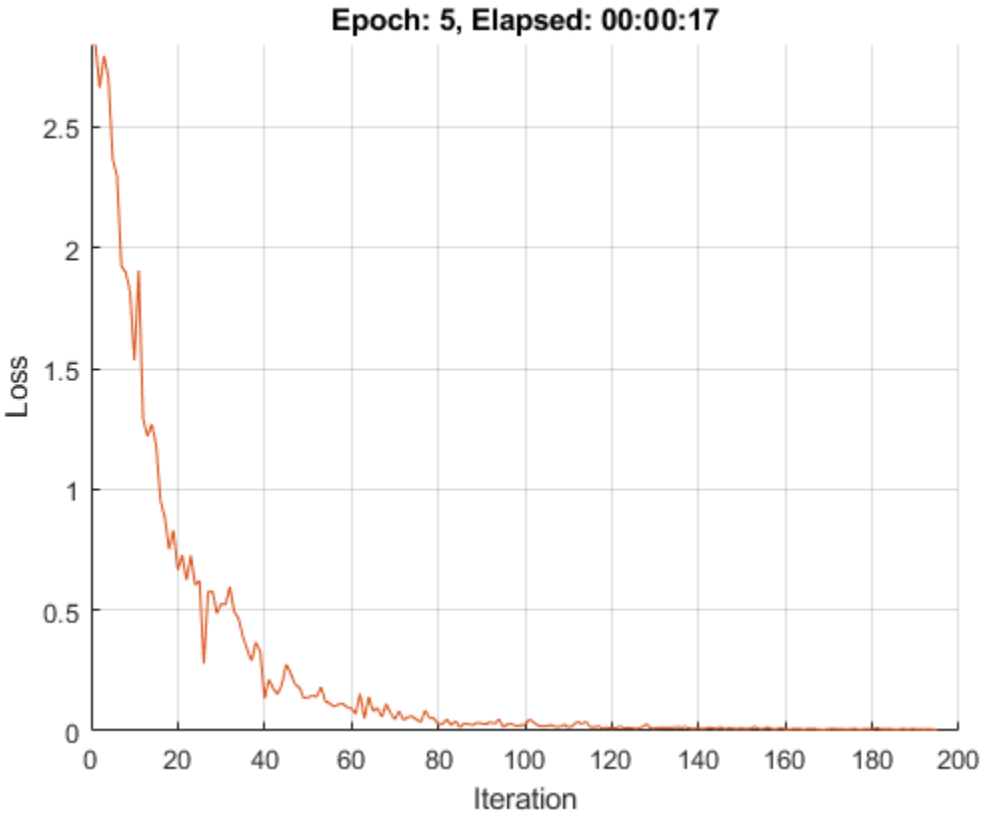
Initialize the velocity parameter for the SGDM solver.

```
velocity = [];
```

Train the network.

```
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);

iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Shuffle data.
    idx = randperm(numel(YTrain));
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);

    % Loop over mini-batches.
    for i = 1:numIterationsPerEpoch
        iteration = iteration + 1;

        % Read mini-batch of data and convert the labels to dummy
        % variables.
        idx = (i-1)*miniBatchSize+1:i*miniBatchSize;
        X = XTrain(:,:,:,idx);

        Y = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y(c,YTrain(idx)==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(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, state, and loss using dlfeval and the
        % modelGradients function and update the network state.
        [gradients,state,loss] = dlfeval(@modelGradients,dlnet,dlX,Y);
        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.

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

Convert the data to a `dlarray` object with dimension format `'SSCB'`. For GPU prediction, also convert the data to `gpuArray`.

```
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

Classify the images using the `modelPredictions` function, listed at the end of the example, and find the classes with the highest scores.

```
dlYPred = modelPredictions(dlnet,dlXTest,miniBatchSize);
[~,idx] = max(extractdata(dlYPred),[],1);
YPred = classes(idx);
```

Evaluate the classification accuracy.

```
accuracy = mean(YPred == YTest)
```

```
accuracy = 0.9948
```

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`, an array of input data `dlX`, and a mini-batch size, and returns the model predictions by iterating over mini-batches of the specified size.

```
function dlYPred = modelPredictions(dlnet,dlX,miniBatchSize)

numObservations = size(dlX,4);
numIterations = ceil(numObservations / miniBatchSize);

numClasses = dlnet.Layers(11).OutputSize;
dlYPred = zeros(numClasses,numObservations,'like',dlX);

for i = 1:numIterations
    idx = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);

    dlYPred(:,idx) = predict(dlnet,dlX(:,:,:,idx));
end
```

```
end
```

See Also

`adamupdate` | `dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `forward` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

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. To read a mini-batch of data from the datastore, set the `ReadSize` property to the mini-batch size.

For each mini-batch:

- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU prediction, convert to `gpuArray` objects.
- Make predictions using the `predict` function.
- Determine the class labels by finding the maximum scores.

Specify the prediction options. Specify a mini-batch size of 128 and make predictions on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
miniBatchSize = 128;
executionEnvironment = "auto";
```

Set the read size property of the image datastore to the mini-batch size.

```
imds.ReadSize = miniBatchSize;
```

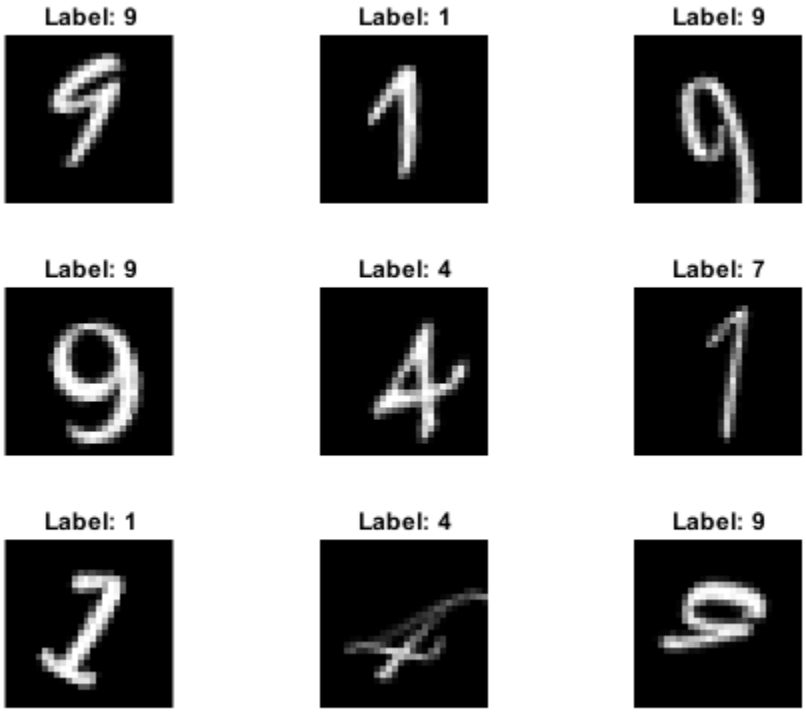
Make predictions by looping over the mini-batches of data.

```
numObservations = numel(imds.Files);
YPred = strings(1,numObservations);
i = 1;
```

```
% Loop over mini-batches.
while hasdata(imds)

    % Read mini-batch of data.
    data = read(imds);
    X = cat(4,data{:});

    % Normalize the images.
    X = single(X)/255;

    % Convert mini-batch of data to dlarray.
    dlX = dlarray(X,'SSCB');

    % If training on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dlX = gpuArray(dlX);
    end

    % Make predictions using the predict function.
    dlYPred = predict(dlnet,dlX);

    % Determine corresponding classes.
    [~,idxTop] = max(extractdata(dlYPred),[],1);
    idxMiniBatch = i:min((i+miniBatchSize-1),numObservations);
    YPred(idxMiniBatch) = classes(idxTop);

    i = i + miniBatchSize;
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 = YPred(idx(i));
    imshow(I)
    title("Label: " + label)
end
```



See Also

`dlarray` | `dlnetwork` | `predict`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45
- “Automatic Differentiation Background” on page 15-112

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.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;
classNames = categories(YTrain);
numClasses = numel(classNames);
numObservations = numel(YTrain);
```

View some images from the training data.

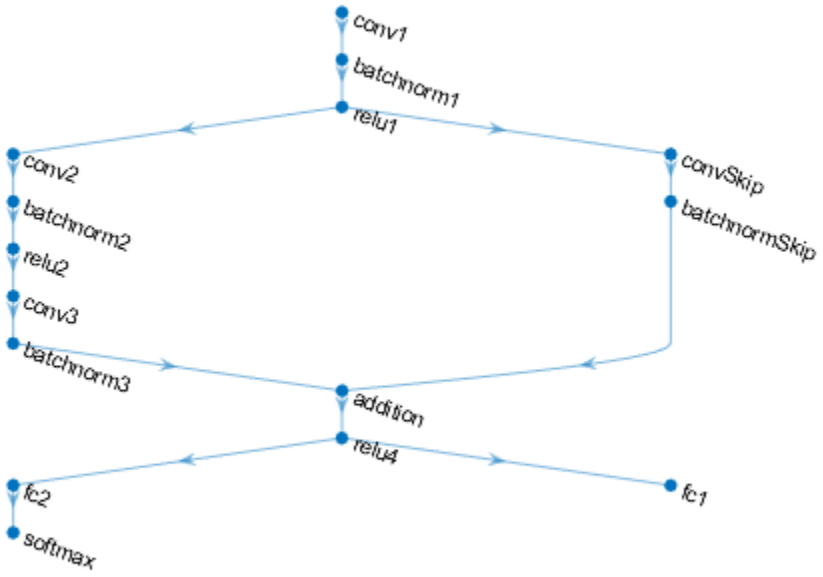
```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

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.
- 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 "conv_1") and `ParameterName` is the name of the parameter (for example, "Weights").

Create a struct `parameters` containing the model parameters. Initialize the learnable layer weights using the example function `initializeGaussian`, listed at the end of the example. Initialize the learnable layer biases with zeros. Initialize the batch normalization offset and scale parameters with zeros and ones, 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. Initialize the batch normalization trained mean and trained variance states with zeros and ones, respectively.

```
parameters.conv1.Weights = dlarray(initializeGaussian([5,5,1,16]));
parameters.conv1.Bias = dlarray(zeros(16,1,'single'));
```

```
parameters.batchnorm1.Offset = dlarray(zeros(16,1,'single'));
parameters.batchnorm1.Scale = dlarray(ones(16,1,'single'));
state.batchnorm1.TrainedMean = zeros(16,1,'single');
state.batchnorm1.TrainedVariance  = ones(16,1,'single');

parameters.convSkip.Weights = dlarray(initializeGaussian([1,1,16,32]));
parameters.convSkip.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnormSkip.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnormSkip.Scale = dlarray(ones(32,1,'single'));
state.batchnormSkip.TrainedMean = zeros(32,1,'single');
state.batchnormSkip.TrainedVariance = ones(32,1,'single');

parameters.conv2.Weights = dlarray(initializeGaussian([3,3,16,32]));
parameters.conv2.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm2.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm2.Scale = dlarray(ones(32,1,'single'));
state.batchnorm2.TrainedMean = zeros(32,1,'single');
state.batchnorm2.TrainedVariance  = ones(32,1,'single');

parameters.conv3.Weights = dlarray(initializeGaussian([3,3,32,32]));
parameters.conv3.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm3.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm3.Scale = dlarray(ones(32,1,'single'));
state.batchnorm3.TrainedMean = zeros(32,1,'single');
state.batchnorm3.TrainedVariance  = ones(32,1,'single');

parameters.fc2.Weights = dlarray(initializeGaussian([10,6272]));
parameters.fc2.Bias = dlarray(zeros(numClasses,1,'single'));

parameters.fc1.Weights = dlarray(initializeGaussian([1,6272]));
parameters.fc1.Bias = dlarray(zeros(1,1,'single'));
```

View the struct of the parameters.

```
parameters
```

```
parameters = struct with fields:
            conv1: [1×1 struct]
       batchnorm1: [1×1 struct]
         convSkip: [1×1 struct]
    batchnormSkip: [1×1 struct]
            conv2: [1×1 struct]
       batchnorm2: [1×1 struct]
            conv3: [1×1 struct]
       batchnorm3: [1×1 struct]
              fc2: [1×1 struct]
              fc1: [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]
    batchnormSkip: [1×1 struct]
       batchnorm2: [1×1 struct]
       batchnorm3: [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 input data `dlX`, the model parameters `parameters`, 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 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";

numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU if one is available. This requires Parallel Computing Toolbox™. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

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 epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert to `gpuArray` objects.
- 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
```

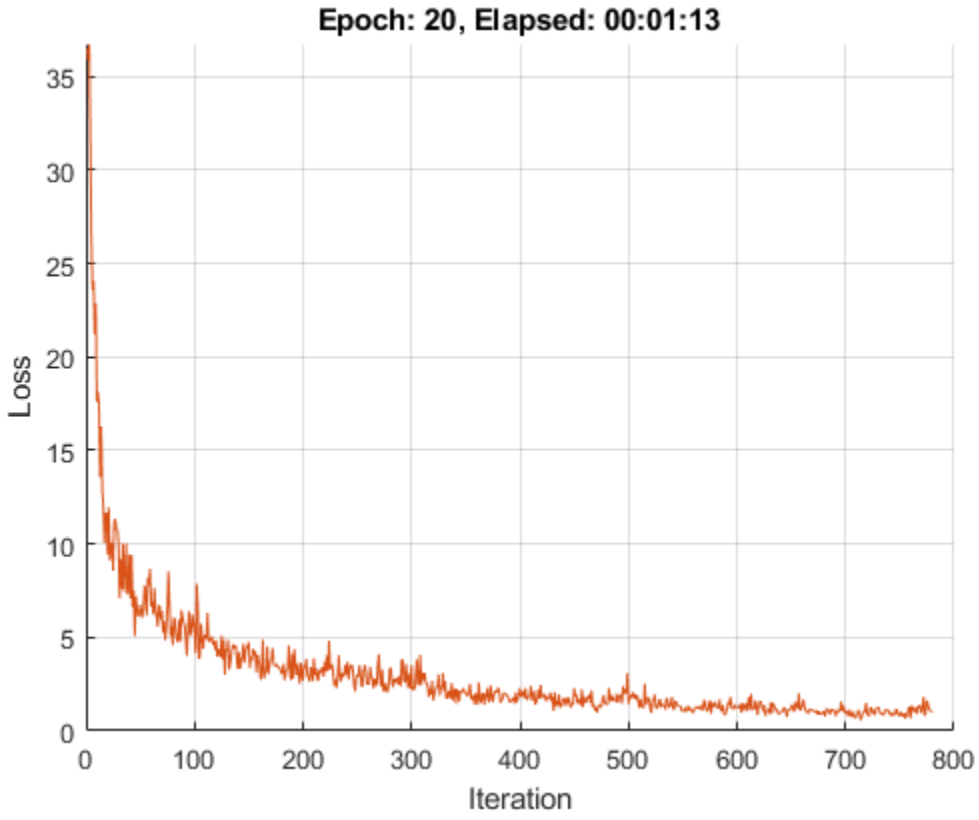
Initialize parameters for Adam.

```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    idx = randperm(numObservations);
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);
    anglesTrain = anglesTrain(idx);

    % 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.
        X = XTrain(:,:,:,idx);

        Y1 = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y1(c,YTrain(idx)==classNames(c)) = 1;
        end
```

```
            Y2 = anglesTrain(idx)';
            Y2 = single(Y2);

            % Convert mini-batch of data to dlarray.
            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, state, and loss using dlfeval and the
            % modelGradients function.
            [gradients,state,loss] = dlfeval(@modelGradients, dlX, Y1, Y2, parameters, 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

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

Convert the data to a `dlarray` object with dimension format `'SSCB'`. For GPU prediction, also convert the data to `gpuArray`.

```
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

To predict the labels and angles of the validation data, use the model function with the `doTraining` option set to `false`.

```
doTraining = false;
[dlYPred,anglesPred] = model(dlXTest, parameters,doTraining,state);
```

Evaluate the classification accuracy.

```
[~,idx] = max(extractdata(dlYPred),[],1);
labelsPred = classNames(idx);
accuracy = mean(labelsPred==YTest)
```

```
accuracy = 0.9852
```

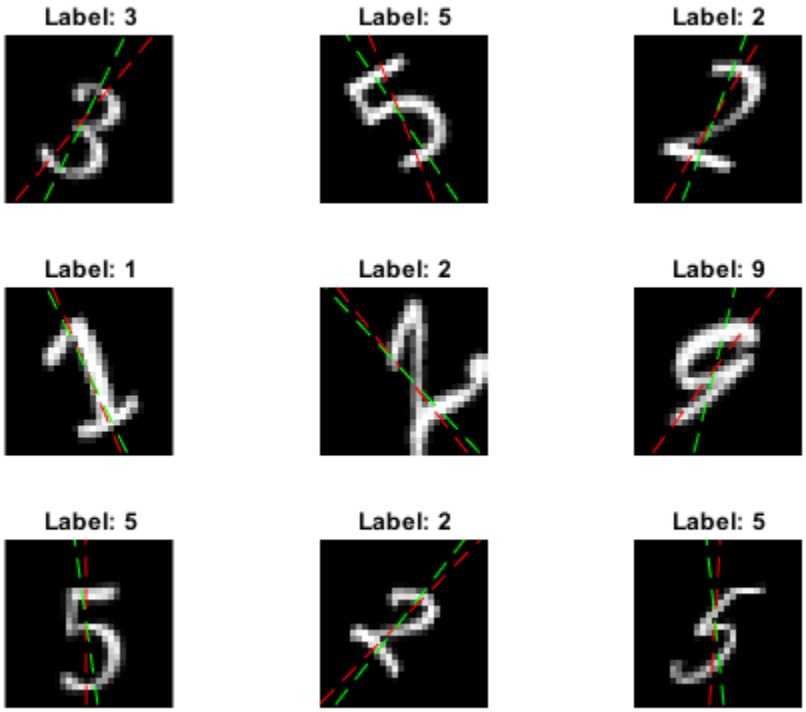
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean((extractdata(anglesPred) - anglesTest').^2))
```

```
angleRMSE =

  gpuArray single

   10.4900
```

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 = extractdata(anglesPred(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(labelsPred(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes the input data `dlX`, the model parameters `parameters`, 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(dlX,parameters,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,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',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,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',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,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 (angles)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

% Fully connect, softmax (labels)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

end
```

Model Gradients Function

The `modelGradients` function, takes 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(dlX,T1,T2,parameters,state)

doTraining = true;
[dlY1,dlY2,state] = model(dlX,parameters,doTraining,state);

lossLabels = crossentropy(dlY1,T1);
lossAngles = mse(dlY2,T2);

loss = lossLabels + 0.1*lossAngles;
gradients = dlgradient(loss,parameters);

end
```

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
```

See Also

`batchnorm` | `crossentropy` | `dlarray` | `dlconv` | `dlfeval` | `dlgradient` | `fullyconnect` | `relu` | `sgdmupdate` | `softmax`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Update Batch Normalization Statistics Using Model Function” on page 15-161
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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.

```
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;
classNames = categories(YTrain);
numClasses = numel(classNames);
numObservations = numel(YTrain);
```

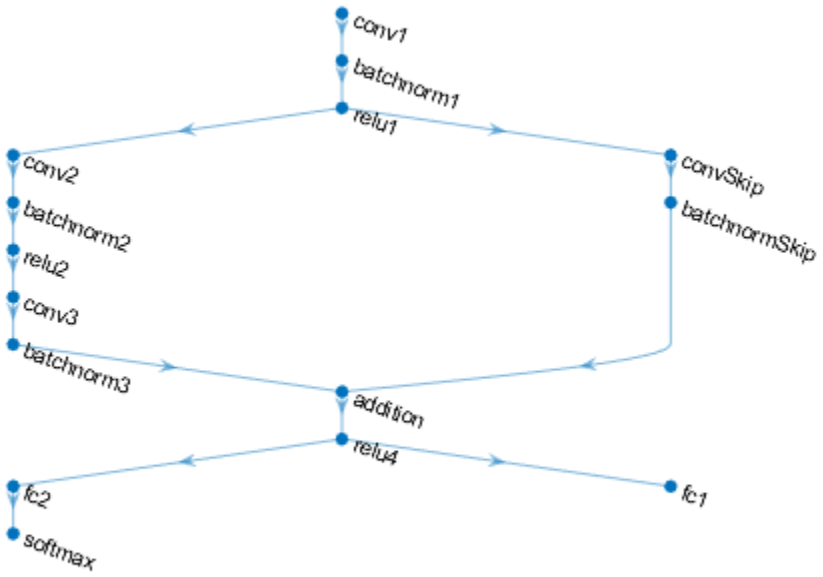
View some images from the training data.

```
idx = randperm(numObservations,64);
I = imtile(XTrain(:,:,:,idx));
figure
imshow(I)
```

Define Deep Learning Model

Define the following network, which predicts both labels and angles of rotation.

- A block of convolution, batch normalization, ReLU operations with 16 5-by-5 filters
- A branch of two blocks of convolution and batch normalization operations, each with 32 3-by-3 filters and separated by a ReLU operation
- A skip connection with a block of convolution and batch normalization operations with 32 1-by-1 convolutions
- Combine both branches using an addition operation followed by a ReLU operation
- For the regression output, a branch with a fully connect operation of size 1 (the number of responses)
- For classification output, a branch with a fully connect 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, `conv_1`), and `ParameterName` is the name of the parameter (for example, `Weights`).

Create a struct `parameters` containing the model parameters. Initialize the learnable layer weights using the example function `initializeGaussian`, listed at the end of the example. Initialize the learnable layer biases with zeros. Initialize the batch normalization offset and scale parameters with zeros and ones, respectively.

To perform training and inference using batch normalization layers, you must also manage the network state. Before prediction, you must specify the data set mean and variance derived from the training data. Create a struct `state` containing the state parameters. Initialize the batch normalization trained mean and trained variance states with zeros and ones, respectively.

```
parameters.conv1.Weights = dlarray(initializeGaussian([5,5,1,16]));
parameters.conv1.Bias = dlarray(zeros(16,1,'single'));

parameters.batchnorm1.Offset = dlarray(zeros(16,1,'single'));
parameters.batchnorm1.Scale = dlarray(ones(16,1,'single'));
state.batchnorm1.TrainedMean = zeros(16,1,'single');
state.batchnorm1.TrainedVariance  = ones(16,1,'single');

parameters.convSkip.Weights = dlarray(initializeGaussian([1,1,16,32]));
parameters.convSkip.Bias = dlarray(zeros(32,1,'single'));
```

```
parameters.batchnormSkip.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnormSkip.Scale = dlarray(ones(32,1,'single'));
state.batchnormSkip.TrainedMean = zeros(32,1,'single');
state.batchnormSkip.TrainedVariance = ones(32,1,'single');

parameters.conv2.Weights = dlarray(initializeGaussian([3,3,16,32]));
parameters.conv2.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm2.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm2.Scale = dlarray(ones(32,1,'single'));
state.batchnorm2.TrainedMean = zeros(32,1,'single');
state.batchnorm2.TrainedVariance  = ones(32,1,'single');

parameters.conv3.Weights = dlarray(initializeGaussian([3,3,32,32]));
parameters.conv3.Bias = dlarray(zeros(32,1,'single'));

parameters.batchnorm3.Offset = dlarray(zeros(32,1,'single'));
parameters.batchnorm3.Scale = dlarray(ones(32,1,'single'));
state.batchnorm3.TrainedMean = zeros(32,1,'single');
state.batchnorm3.TrainedVariance  = ones(32,1,'single');

parameters.fc2.Weights = dlarray(initializeGaussian([numClasses,6272]));
parameters.fc2.Bias = dlarray(zeros(numClasses,1,'single'));

parameters.fc1.Weights = dlarray(initializeGaussian([1,6272]));
parameters.fc1.Bias = dlarray(zeros(1,1,'single'));
```

View the struct of the state.

```
state
```

```
state = struct with fields:
       batchnorm1: [1×1 struct]
    batchnormSkip: [1×1 struct]
       batchnorm2: [1×1 struct]
       batchnorm3: [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 input data `dlX`, the model parameters `parameters`, 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";

numIterationsPerEpoch = floor(numObservations./miniBatchSize);
```

Train on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.

```
executionEnvironment = "auto";
```

Train Model

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 epoch, display the training progress.

For each mini-batch:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the data to `gpuArray` objects.
- 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.

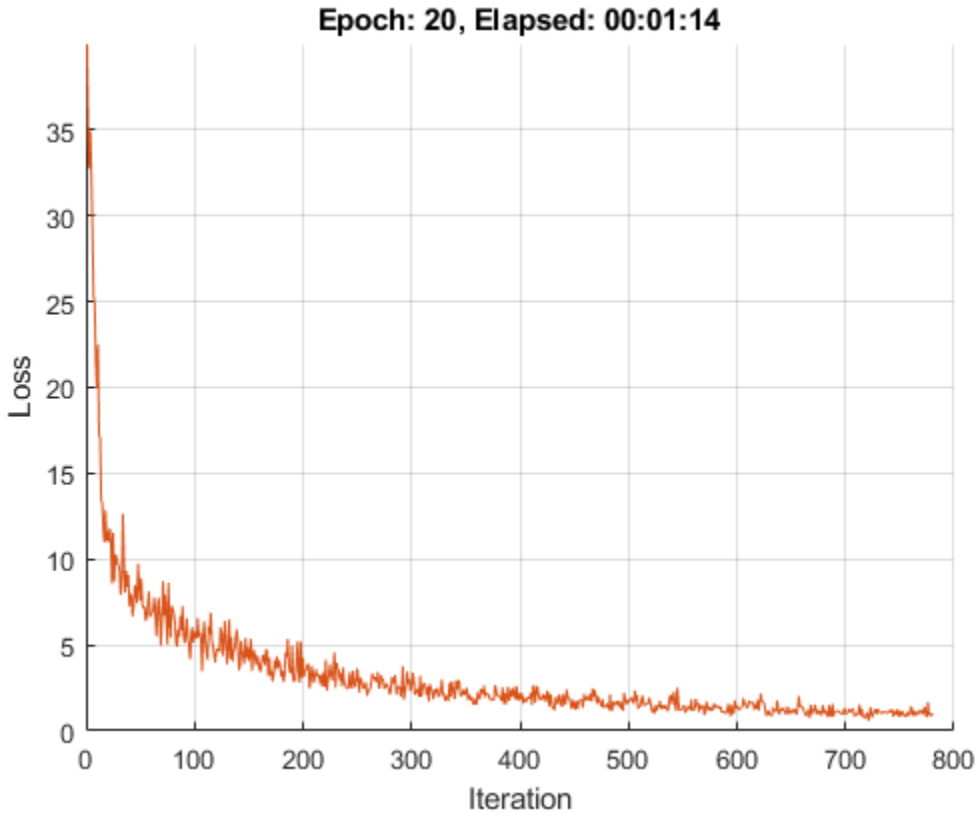
```
if plots == "training-progress"
    figure

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

Initialize the parameters for the Adam solver.

```
trailingAvg = [];
trailingAvgSq = [];
```

Train the model.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs

    % Shuffle data.
    idx = randperm(numObservations);
    XTrain = XTrain(:,:,:,idx);
    YTrain = YTrain(idx);
    anglesTrain = anglesTrain(idx);

    % 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.
        X = XTrain(:,:,:,idx);

        Y1 = zeros(numClasses, miniBatchSize, 'single');
        for c = 1:numClasses
            Y1(c,YTrain(idx)==classNames(c)) = 1;
        end

        Y2 = anglesTrain(idx)';
        Y2 = single(Y2);

        % Convert mini-batch of data to dlarray.
        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, state, and loss using dlfeval and the
        % modelGradients function.
        [gradients,state,loss] = dlfeval(@modelGradients, dlX, Y1, Y2, parameters, 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.

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

Convert the data to a `dlarray` object with dimension format `'SSCB'`. For GPU prediction, also convert the data to `gpuArray`.

```
dlXTest = dlarray(XTest,'SSCB');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlXTest = gpuArray(dlXTest);
end
```

To predict the labels and angles of the validation data, use the `modelPredictions` function, listed at the end of the example.

```
[dlYPred,anglesPred] = modelPredictions(parameters,state,dlXTest,miniBatchSize);
```

Evaluate the classification accuracy.

```
[~,idx] = max(extractdata(dlYPred),[],1);
labelsPred = classNames(idx);
accuracy = mean(labelsPred==YTest)
```

```
accuracy = 0.9912
```

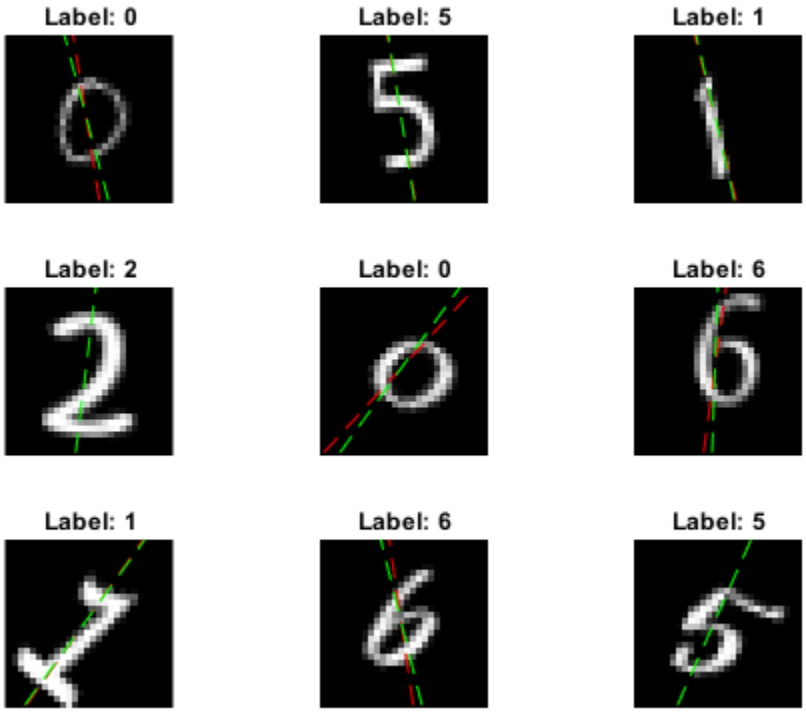
Evaluate the regression accuracy.

```
angleRMSE = sqrt(mean((extractdata(anglesPred) - anglesTest').^2))
```

```
angleRMSE =

    6.1576
```

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 = extractdata(anglesPred(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(labelsPred(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function `model` takes as input the input data `dlX`, the model parameters `parameters`, 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(dlX,parameters,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,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',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,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',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,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 (angles)
weights = parameters.fc1.Weights;
bias = parameters.fc1.Bias;
dlY2 = fullyconnect(dlY,weights,bias);

% Fully connect, softmax (labels)
weights = parameters.fc2.Weights;
bias = parameters.fc2.Bias;
dlY1 = fullyconnect(dlY,weights,bias);
dlY1 = softmax(dlY1);

end
```

Model Gradients Function

The `modelGradients` function takes as input 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(dlX,T1,T2,parameters,state)

doTraining = true;
[dlY1,dlY2,state] = model(dlX,parameters,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, an array of input data `dlX`, and a mini-batch size, and returns the model predictions by iterating over mini-batches of the specified size using the `model` function with the `doTraining` option set to `false`.

```
function [dlYPred,anglesPred] = modelPredictions(parameters,state,dlX,miniBatchSize)

doTraining = false;

numObservations = size(dlX,4);
```

```
numIterations = ceil(numObservations / miniBatchSize);

numClasses = size(parameters.fc2.Weights,1);

dlYPred = zeros(numClasses,numObservations,'like',dlX);
anglesPred = zeros(1,numObservations,'like',dlX);

for i = 1:numIterations
    idx = (i-1)*miniBatchSize+1:min(i*miniBatchSize,numObservations);

    [dlYPred(:,idx),anglesPred(idx)] = model(dlX(:,:,:,idx), parameters,doTraining,state);
end

end
```

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
```

See Also

`batchnorm` | `crossentropy` | `dlarray` | `dlconv` | `dlfeval` | `dlgradient` | `fullyconnect` | `relu` | `sgdmupdate` | `softmax`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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');
```

Make Predictions

Loop over the mini-batches of the test data and make predictions using a custom prediction loop.

For each mini-batch:

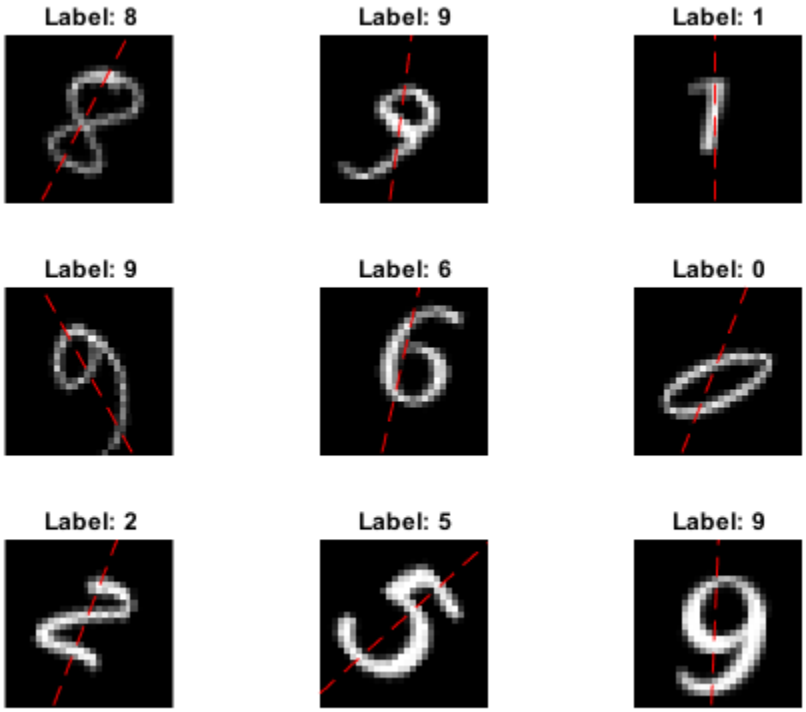
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- Make predictions on a GPU if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU with compute capability 3.0 or higher.
- Make predictions by calling the model function with the `doTraining` option set to `false`.
- Determine the class labels by finding the maximum scores.

```
miniBatchSize = 128;
executionEnvironment = "auto";
doTraining = false;

imds.ReadSize = miniBatchSize;
numObservations = numel(imds.Files);
Y1Pred = strings(1,numObservations);
Y2Pred = zeros(1,numObservations);
```

```
i = 1;

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

    % Read mini-batch of data.
    data = read(imds);
    X = cat(4,data{:});

    % Normalize the images.
    X = single(X)/255;

    % Convert mini-batch of data to dlarray.
    dlX = dlarray(X,'SSCB');

    % If making predictions on a GPU, then convert data to gpuArray.
    if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
        dlX = gpuArray(dlX);
    end

    % Make predictions using the predict function.
    [dlY1Pred,dlY2Pred] = model(dlX,parameters,doTraining,state);

    % Determine corresponding classes.
    [~,idxTop] = max(extractdata(dlY1Pred),[],1);
    idxMiniBatch = i:min((i+miniBatchSize-1),numObservations);
    Y1Pred(idxMiniBatch) = classNames(idxTop);

    Y2Pred(idxMiniBatch) = gather(extractdata(dlY2Pred));

    i = i + miniBatchSize;
end
```

View some of the images with their predictions.

```
idx = randperm(numel(imds.Files),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 = Y2Pred(idx(i));
    plot(offset*[1-tand(thetaPred) 1+tand(thetaPred)],[sz 0],'r--')

    hold off
    label = string(Y1Pred(idx(i)));
    title("Label: " + label)
end
```



Model Function

The function model takes the input data `dlX`, the model parameters `parameters`, 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(dlX,parameters,doTraining,state)

% Convolution
W = parameters.conv1.Weights;
B = parameters.conv1.Bias;
dlY = dlconv(dlX,W,B,'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,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)
W = parameters.convSkip.Weights;
B = parameters.convSkip.Bias;
dlYSkip = dlconv(dlY,W,B,'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
W = parameters.conv2.Weights;
B = parameters.conv2.Bias;
dlY = dlconv(dlY,W,B,'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,trainedVariance);

    % Update state
    state.batchnorm2.TrainedMean = trainedMean;
    state.batchnorm2.TrainedVariance = trainedVariance;
else
    dlY = batchnorm(dlY,Offset,Scale,trainedMean,trainedVariance);
end
dlY = relu(dlY);

% Convolution
W = parameters.conv3.Weights;
B = parameters.conv3.Bias;
dlY = dlconv(dlY,W,B,'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,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 (angles)
W = parameters.fc1.Weights;
B = parameters.fc1.Bias;
dlY2 = fullyconnect(dlY,W,B);

% Fully connect, softmax (labels)
W = parameters.fc2.Weights;
B = parameters.fc2.Bias;
dlY1 = fullyconnect(dlY,W,B);
dlY1 = softmax(dlY1);

end
```

See Also

`batchnorm` | `dlarray` | `dlconv` | `dlfeval` | `dlgradient` | `fullyconnect` | `relu` | `sgdmupdate` | `softmax`

More About

- “Multiple-Input and Multiple-Output Networks” on page 1-21
- “Assemble Multiple-Output Network for Prediction” on page 15-106
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Train Network Using Custom Training Loop” on page 15-134
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “List of Functions with dlarray Support” on page 15-194

Train Network Using Cyclical Learn Rate for Snapshot Ensembling

This example shows how to train a network to classify images of objects using a cyclical learn rate schedule and snapshot ensembling for better test accuracy. In the example, you learn how to use a cosine function for the learn 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 learn rate: for each iteration, the solver uses the learn 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 learn rate, and `M` is the number of cycles/snapshots. This learn 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);
auimdsTest = augmentedImageDatastore(imageSize, XTest, YTest);
```

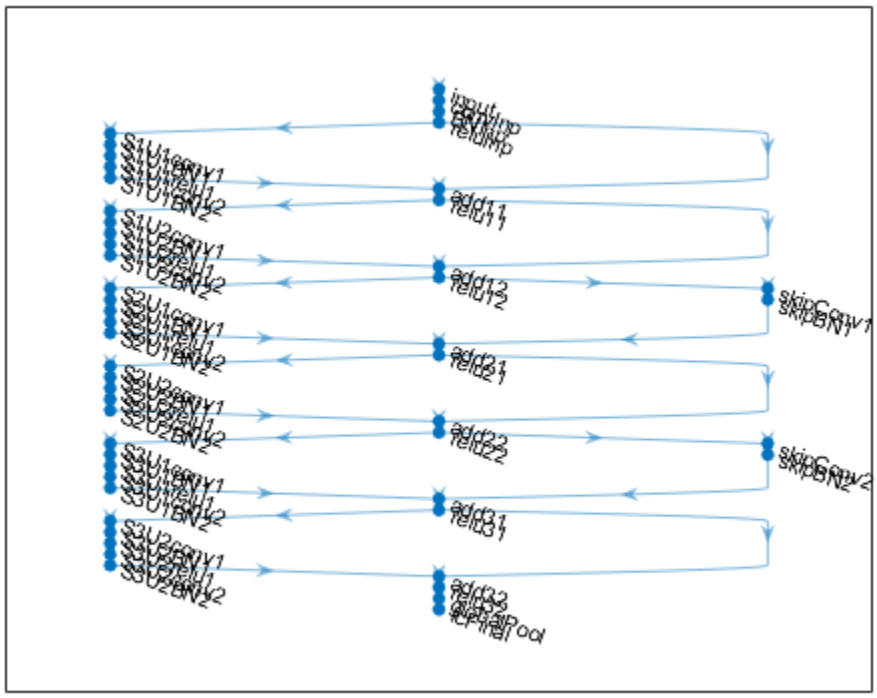
Define Network Architecture

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.

```
velocity = [];
numEpochs = 200;
miniBatchSize = 64;
augimdsTrain.MiniBatchSize = miniBatchSize;
numObservations = numel(YTrain);
numIterationsPerEpoch = floor(numObservations./miniBatchSize);
momentum = 0.9;
weightDecay = 1e-4;
```

Specify the training options specific to the cyclical learn rate. `Alpha0` is the initial learn rate and `numSnapshots` is the number of cycles or snapshots taken during training.

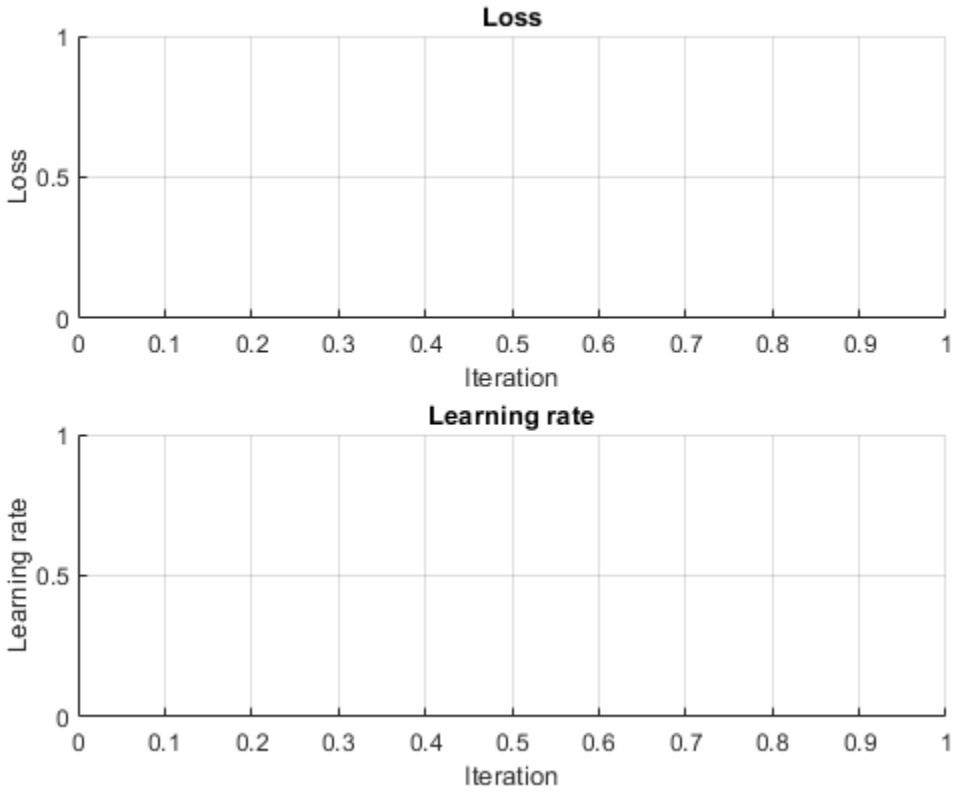
```
alpha0 = 0.1;
numSnapshots = 5;
epochsPerSnapshot = numEpochs./numSnapshots;
iterationsPerSnapshot = ceil(numObservations./miniBatchSize)*numEpochs./numSnapshots;
modelPrefix = "SnapshotEpoch";
```

Train on a GPU if one is available (requires Parallel Computing Toolbox™).

```
executionEnvironment = "auto";
```

Initialize the training figure.

```
[lossLine, learnRateLine] = plotLossAndLearnRate();
```



Train Model

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:

- Convert the labels to dummy variables.
- Convert the data to `dlarray` objects with underlying type single and specify the dimension labels `'SSCB'` (spatial, spatial, channel, batch).
- For GPU training, convert the mini-batch data to `gpuArray` objects.
- Evaluate the model gradients and loss using `dlfeval` and the `modelGradients` function.
- Update the state of the nonlearnable parameters of the network.
- Determine the learn rate for the cyclical learn rate schedule.
- Update the network parameters using the `sgdmupdate` function.
- Plot the loss and learn rate at each iteration.

For this example, the training took approximately 18h on a NVIDIA™ GeForce GTX 1080.

```
iteration = 0;
start = tic;

% Loop over epochs.
for epoch = 1:numEpochs
    % Reset image datastore.
    reset(augimdsTrain);

    % Shuffle data.
    augimdsTrain = shuffle(augimdsTrain);

    % Save snapshot model.
    if ~mod(epoch,epochsPerSnapshot)
        save(modelPrefix + epoch + ".mat",'dlnet');
    end

    % Loop over mini-batches.
    while hasdata(augimdsTrain)
        iteration = iteration + 1;

        % Read mini-batch of data.
        data = read(augimdsTrain);

        % Concatenate the inputs.
        Xdata = data{:,1};
        X = cat(4,Xdata{:});

        % Convert the labels to dummy variables.
        TrueClasses = data{:,2};
        Y = zeros(numClasses, numel(TrueClasses), 'single');
        for c = 1:numClasses
            Y(c,TrueClasses==classes(c)) = 1;
        end

        % Convert mini-batch of data to dlarray.
        dlX = dlarray(single(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 loss using dlfeval and the
```

```
        % modelGradients function.
        [gradients, loss, state] = dlfeval(@modelGradients,dlnet,dlX,Y,weightDecay);

        % Update the state of nonlearnable parameters.
        dlnet.State = state;

        % Determine learn rate for cyclical learn 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

        % Plot loss and learn rate for current iteration.
        loss = double(gather(extractdata(loss)));
        addpoints(lossLine, iteration, loss);
        addpoints(learnRateLine, iteration, learnRate);
        drawnow

    end

    % Display the training progress.
    D = duration(0,0,toc(start),'Format','hh:mm:ss');
    disp( ...
        "Epoch: " + epoch + ", " + ...
        "Loss: " + num2str(loss) + ", " + ...
        "Elapsed: " + string(D))
end
```

```
Epoch: 1, Loss: 1.784, Elapsed: 00:05:59
Epoch: 2, Loss: 1.1388, Elapsed: 00:11:32
Epoch: 3, Loss: 1.0594, Elapsed: 00:17:02
Epoch: 4, Loss: 1.3154, Elapsed: 00:22:34
Epoch: 5, Loss: 0.7968, Elapsed: 00:28:08
Epoch: 6, Loss: 0.85319, Elapsed: 00:33:47
Epoch: 7, Loss: 0.5477, Elapsed: 00:39:26
Epoch: 8, Loss: 1.1158, Elapsed: 00:45:14
Epoch: 9, Loss: 0.39734, Elapsed: 00:50:47
Epoch: 10, Loss: 0.47909, Elapsed: 00:56:26
Epoch: 11, Loss: 0.68743, Elapsed: 01:02:08
Epoch: 12, Loss: 0.45266, Elapsed: 01:07:51
Epoch: 13, Loss: 1.0058, Elapsed: 01:13:27
Epoch: 14, Loss: 0.64816, Elapsed: 01:19:14
Epoch: 15, Loss: 1.1039, Elapsed: 01:25:05
Epoch: 16, Loss: 1.2072, Elapsed: 01:30:41
Epoch: 17, Loss: 0.43167, Elapsed: 01:36:24
```

```
Epoch: 18, Loss: 1.1376, Elapsed: 01:42:40
Epoch: 19, Loss: 0.76857, Elapsed: 01:48:40
Epoch: 20, Loss: 0.77434, Elapsed: 01:54:22
Epoch: 21, Loss: 0.98595, Elapsed: 01:59:57
Epoch: 22, Loss: 0.78628, Elapsed: 02:05:32
Epoch: 23, Loss: 0.55069, Elapsed: 02:11:07
Epoch: 24, Loss: 0.52066, Elapsed: 02:16:43
Epoch: 25, Loss: 0.44842, Elapsed: 02:22:18
Epoch: 26, Loss: 0.40094, Elapsed: 02:27:55
Epoch: 27, Loss: 0.78839, Elapsed: 02:33:30
Epoch: 28, Loss: 0.47829, Elapsed: 02:39:05
Epoch: 29, Loss: 0.21833, Elapsed: 02:44:41
Epoch: 30, Loss: 0.5759, Elapsed: 02:50:16
Epoch: 31, Loss: 1.1089, Elapsed: 02:55:50
Epoch: 32, Loss: 0.37353, Elapsed: 03:01:25
Epoch: 33, Loss: 0.30851, Elapsed: 03:07:01
Epoch: 34, Loss: 0.34735, Elapsed: 03:12:36
Epoch: 35, Loss: 0.28772, Elapsed: 03:18:11
Epoch: 36, Loss: 0.31045, Elapsed: 03:23:47
Epoch: 37, Loss: 0.28555, Elapsed: 03:29:22
Epoch: 38, Loss: 0.897, Elapsed: 03:34:58
Epoch: 39, Loss: 0.69014, Elapsed: 03:40:33
Epoch: 40, Loss: 0.26282, Elapsed: 03:46:17
Epoch: 41, Loss: 1.0086, Elapsed: 03:51:53
Epoch: 42, Loss: 0.47303, Elapsed: 03:57:27
Epoch: 43, Loss: 1.3765, Elapsed: 04:03:02
Epoch: 44, Loss: 0.54884, Elapsed: 04:08:39
Epoch: 45, Loss: 0.38778, Elapsed: 04:14:14
Epoch: 46, Loss: 0.74121, Elapsed: 04:19:49
Epoch: 47, Loss: 0.78481, Elapsed: 04:25:25
Epoch: 48, Loss: 0.44624, Elapsed: 04:31:01
Epoch: 49, Loss: 0.81747, Elapsed: 04:36:38
```

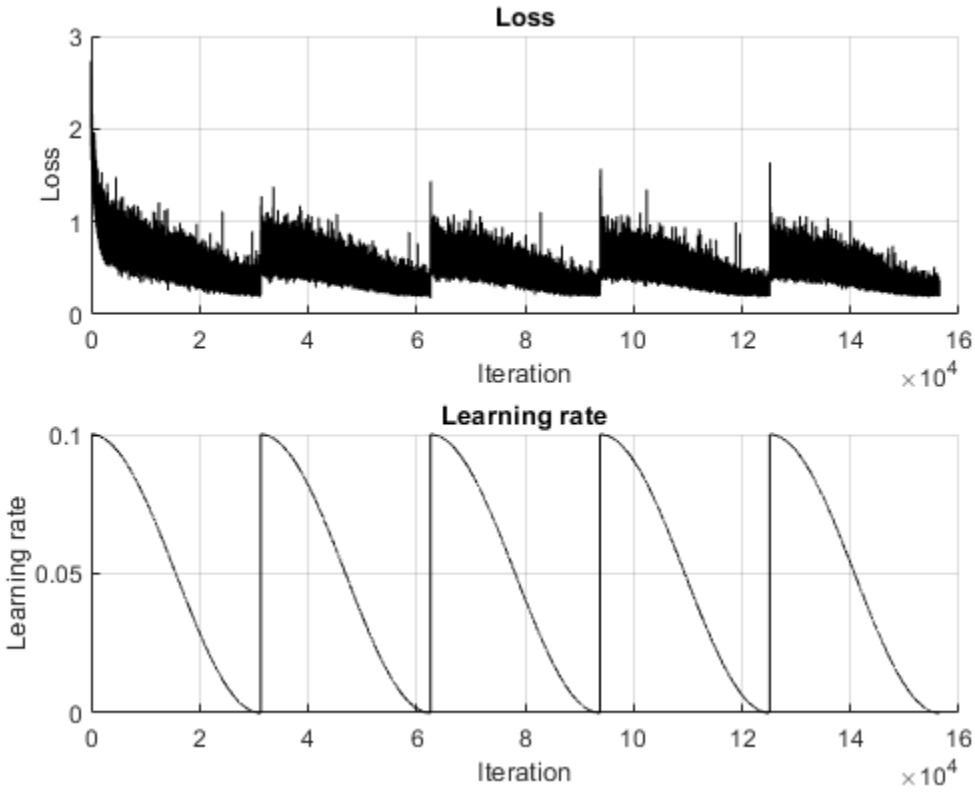
```
Epoch: 50, Loss: 0.40319, Elapsed: 04:42:14
Epoch: 51, Loss: 0.87757, Elapsed: 04:47:51
Epoch: 52, Loss: 1.0567, Elapsed: 04:53:27
Epoch: 53, Loss: 0.29019, Elapsed: 04:59:03
Epoch: 54, Loss: 0.92056, Elapsed: 05:04:40
Epoch: 55, Loss: 0.45776, Elapsed: 05:10:16
Epoch: 56, Loss: 1.0265, Elapsed: 05:15:52
Epoch: 57, Loss: 0.55256, Elapsed: 05:21:29
Epoch: 58, Loss: 1.0822, Elapsed: 05:27:06
Epoch: 59, Loss: 0.78332, Elapsed: 05:32:44
Epoch: 60, Loss: 0.48247, Elapsed: 05:38:20
Epoch: 61, Loss: 0.86749, Elapsed: 05:43:58
Epoch: 62, Loss: 0.64667, Elapsed: 05:49:34
Epoch: 63, Loss: 0.64563, Elapsed: 05:55:10
Epoch: 64, Loss: 0.58239, Elapsed: 06:00:46
Epoch: 65, Loss: 0.29219, Elapsed: 06:06:23
Epoch: 66, Loss: 0.37627, Elapsed: 06:11:59
Epoch: 67, Loss: 0.34035, Elapsed: 06:17:35
Epoch: 68, Loss: 0.34809, Elapsed: 06:23:11
Epoch: 69, Loss: 0.61085, Elapsed: 06:28:47
Epoch: 70, Loss: 0.42018, Elapsed: 06:34:24
Epoch: 71, Loss: 0.3739, Elapsed: 06:40:00
Epoch: 72, Loss: 0.23083, Elapsed: 06:45:37
Epoch: 73, Loss: 0.21324, Elapsed: 06:51:14
Epoch: 74, Loss: 0.18931, Elapsed: 06:56:55
Epoch: 75, Loss: 0.88882, Elapsed: 07:02:31
Epoch: 76, Loss: 0.36844, Elapsed: 07:08:09
Epoch: 77, Loss: 0.76548, Elapsed: 07:13:46
Epoch: 78, Loss: 0.42548, Elapsed: 07:19:24
Epoch: 79, Loss: 0.29112, Elapsed: 07:25:01
Epoch: 80, Loss: 0.17333, Elapsed: 07:30:45
Epoch: 81, Loss: 0.50322, Elapsed: 07:36:22
```

```
Epoch: 82, Loss: 0.40387, Elapsed: 07:41:58
Epoch: 83, Loss: 0.3939, Elapsed: 07:47:34
Epoch: 84, Loss: 0.79005, Elapsed: 07:53:11
Epoch: 85, Loss: 0.51953, Elapsed: 07:58:46
Epoch: 86, Loss: 0.65925, Elapsed: 08:04:24
Epoch: 87, Loss: 0.49915, Elapsed: 08:10:01
Epoch: 88, Loss: 0.58721, Elapsed: 08:15:38
Epoch: 89, Loss: 0.57397, Elapsed: 08:21:15
Epoch: 90, Loss: 0.51315, Elapsed: 08:26:53
Epoch: 91, Loss: 0.42037, Elapsed: 08:32:30
Epoch: 92, Loss: 0.41111, Elapsed: 08:38:06
Epoch: 93, Loss: 0.71338, Elapsed: 08:43:43
Epoch: 94, Loss: 0.31452, Elapsed: 08:49:21
Epoch: 95, Loss: 0.35696, Elapsed: 08:54:58
Epoch: 96, Loss: 0.56142, Elapsed: 09:00:36
Epoch: 97, Loss: 0.69246, Elapsed: 09:06:15
Epoch: 98, Loss: 0.40288, Elapsed: 09:11:53
Epoch: 99, Loss: 0.67491, Elapsed: 09:17:31
Epoch: 100, Loss: 0.70555, Elapsed: 09:23:08
Epoch: 101, Loss: 0.45978, Elapsed: 09:28:47
Epoch: 102, Loss: 0.3963, Elapsed: 09:34:27
Epoch: 103, Loss: 0.60798, Elapsed: 09:40:05
Epoch: 104, Loss: 0.41759, Elapsed: 09:45:45
Epoch: 105, Loss: 0.45068, Elapsed: 09:51:23
Epoch: 106, Loss: 1.103, Elapsed: 09:57:02
Epoch: 107, Loss: 0.29916, Elapsed: 10:02:41
Epoch: 108, Loss: 0.64019, Elapsed: 10:08:21
Epoch: 109, Loss: 0.26558, Elapsed: 10:13:59
Epoch: 110, Loss: 0.41303, Elapsed: 10:19:38
Epoch: 111, Loss: 0.74221, Elapsed: 10:25:18
Epoch: 112, Loss: 0.48748, Elapsed: 10:30:56
Epoch: 113, Loss: 0.27348, Elapsed: 10:36:35
```

```
Epoch: 114, Loss: 0.51661, Elapsed: 10:42:14
Epoch: 115, Loss: 0.27831, Elapsed: 10:47:54
Epoch: 116, Loss: 0.35103, Elapsed: 10:53:33
Epoch: 117, Loss: 0.19571, Elapsed: 10:59:11
Epoch: 118, Loss: 0.37368, Elapsed: 11:04:50
Epoch: 119, Loss: 0.18644, Elapsed: 11:10:29
Epoch: 120, Loss: 0.48589, Elapsed: 11:16:16
Epoch: 121, Loss: 0.74257, Elapsed: 11:21:57
Epoch: 122, Loss: 0.65423, Elapsed: 11:27:37
Epoch: 123, Loss: 0.35185, Elapsed: 11:33:17
Epoch: 124, Loss: 0.81636, Elapsed: 11:38:55
Epoch: 125, Loss: 0.49292, Elapsed: 11:44:34
Epoch: 126, Loss: 0.9133, Elapsed: 11:50:14
Epoch: 127, Loss: 0.80498, Elapsed: 11:55:53
Epoch: 128, Loss: 0.59473, Elapsed: 12:01:33
Epoch: 129, Loss: 0.60313, Elapsed: 12:07:12
Epoch: 130, Loss: 0.5426, Elapsed: 12:12:50
Epoch: 131, Loss: 1.3471, Elapsed: 12:18:29
Epoch: 132, Loss: 0.35591, Elapsed: 12:24:08
Epoch: 133, Loss: 0.75186, Elapsed: 12:29:49
Epoch: 134, Loss: 0.98765, Elapsed: 12:35:29
Epoch: 135, Loss: 0.65345, Elapsed: 12:41:08
Epoch: 136, Loss: 0.78963, Elapsed: 12:46:48
Epoch: 137, Loss: 0.38269, Elapsed: 12:52:27
Epoch: 138, Loss: 0.5309, Elapsed: 12:58:06
Epoch: 139, Loss: 0.4119, Elapsed: 13:03:45
Epoch: 140, Loss: 0.93898, Elapsed: 13:09:26
Epoch: 141, Loss: 0.45791, Elapsed: 13:15:04
Epoch: 142, Loss: 0.70093, Elapsed: 13:20:43
Epoch: 143, Loss: 0.84997, Elapsed: 13:26:23
Epoch: 144, Loss: 0.27732, Elapsed: 13:32:05
Epoch: 145, Loss: 0.51171, Elapsed: 13:37:44
```

```
Epoch: 146, Loss: 0.81123, Elapsed: 13:43:24
Epoch: 147, Loss: 0.5678, Elapsed: 13:49:04
Epoch: 148, Loss: 0.58568, Elapsed: 13:54:44
Epoch: 149, Loss: 0.3952, Elapsed: 14:00:23
Epoch: 150, Loss: 0.31967, Elapsed: 14:06:03
Epoch: 151, Loss: 0.44051, Elapsed: 14:11:46
Epoch: 152, Loss: 0.99278, Elapsed: 14:17:27
Epoch: 153, Loss: 0.87306, Elapsed: 14:23:07
Epoch: 154, Loss: 0.34008, Elapsed: 14:28:47
Epoch: 155, Loss: 0.4687, Elapsed: 14:34:27
Epoch: 156, Loss: 0.22836, Elapsed: 14:40:07
Epoch: 157, Loss: 0.23204, Elapsed: 14:45:48
Epoch: 158, Loss: 0.36854, Elapsed: 14:51:28
Epoch: 159, Loss: 0.35363, Elapsed: 14:57:08
Epoch: 160, Loss: 0.37937, Elapsed: 15:02:55
Epoch: 161, Loss: 0.7725, Elapsed: 15:08:36
Epoch: 162, Loss: 0.59353, Elapsed: 15:14:15
Epoch: 163, Loss: 0.57963, Elapsed: 15:19:54
Epoch: 164, Loss: 0.54625, Elapsed: 15:25:35
Epoch: 165, Loss: 0.65612, Elapsed: 15:31:15
Epoch: 166, Loss: 0.73254, Elapsed: 15:36:56
Epoch: 167, Loss: 0.4483, Elapsed: 15:42:37
Epoch: 168, Loss: 0.36817, Elapsed: 15:48:17
Epoch: 169, Loss: 0.57539, Elapsed: 15:53:57
Epoch: 170, Loss: 1.0026, Elapsed: 15:59:37
Epoch: 171, Loss: 0.95288, Elapsed: 16:05:17
Epoch: 172, Loss: 0.83053, Elapsed: 16:10:59
Epoch: 173, Loss: 0.41976, Elapsed: 16:16:39
Epoch: 174, Loss: 0.44098, Elapsed: 16:22:19
Epoch: 175, Loss: 0.58823, Elapsed: 16:28:00
Epoch: 176, Loss: 0.67325, Elapsed: 16:33:41
Epoch: 177, Loss: 0.27045, Elapsed: 16:39:21
```

```
Epoch: 178, Loss: 0.66652, Elapsed: 16:45:02
Epoch: 179, Loss: 1.0097, Elapsed: 16:50:43
Epoch: 180, Loss: 0.40372, Elapsed: 16:56:23
Epoch: 181, Loss: 0.39175, Elapsed: 17:02:04
Epoch: 182, Loss: 0.40741, Elapsed: 17:07:45
Epoch: 183, Loss: 0.35398, Elapsed: 17:13:25
Epoch: 184, Loss: 0.63228, Elapsed: 17:19:05
Epoch: 185, Loss: 0.35308, Elapsed: 17:24:45
Epoch: 186, Loss: 0.46854, Elapsed: 17:30:27
Epoch: 187, Loss: 0.51346, Elapsed: 17:36:08
Epoch: 188, Loss: 0.71886, Elapsed: 17:41:48
Epoch: 189, Loss: 0.73986, Elapsed: 17:47:29
Epoch: 190, Loss: 0.46669, Elapsed: 17:53:10
Epoch: 191, Loss: 0.40962, Elapsed: 17:58:51
Epoch: 192, Loss: 0.25007, Elapsed: 18:04:31
Epoch: 193, Loss: 0.45651, Elapsed: 18:10:12
Epoch: 194, Loss: 0.20788, Elapsed: 18:15:52
Epoch: 195, Loss: 0.32097, Elapsed: 18:21:32
Epoch: 196, Loss: 0.28159, Elapsed: 18:27:15
Epoch: 197, Loss: 0.20396, Elapsed: 18:32:56
Epoch: 198, Loss: 0.30823, Elapsed: 18:38:37
Epoch: 199, Loss: 0.28583, Elapsed: 18:44:18
```




```
Epoch: 200, Loss: 0.32877, Elapsed: 18:50:07
```

Create Snapshot Ensemble

Combine the M snapshots of the network taken during training to form a final ensemble. The ensemble predictions correspond to the average of the output of the fully connected layer from all M individual models.

```
YPredictions = zeros(numClasses,numel(YTest),numSnapshots);
modelAccuracy = zeros(numSnapshots+1,1);
modelName = cell(numSnapshots+1,1);
for m = 1:numSnapshots
    modelName{m} = modelPrefix + m*epochsPerSnapshot;
    load(modelName{m} + ".mat");
    YPredictions(:,:,m) = gather(extractdata(predict(dlnet, dlarray(single(XTest),'SSCB'))));
    modelAccuracy(m) = computeAccuracy(YPredictions(:,:,m), YTest, classes);
    disp(modelName{m} + " accuracy: " + modelAccuracy(m) + "%")
end
```

```
SnapshotEpoch40 accuracy: 88.04%
SnapshotEpoch80 accuracy: 86.78%
SnapshotEpoch120 accuracy: 87.53%
SnapshotEpoch160 accuracy: 87.07%
SnapshotEpoch200 accuracy: 88.39%
```

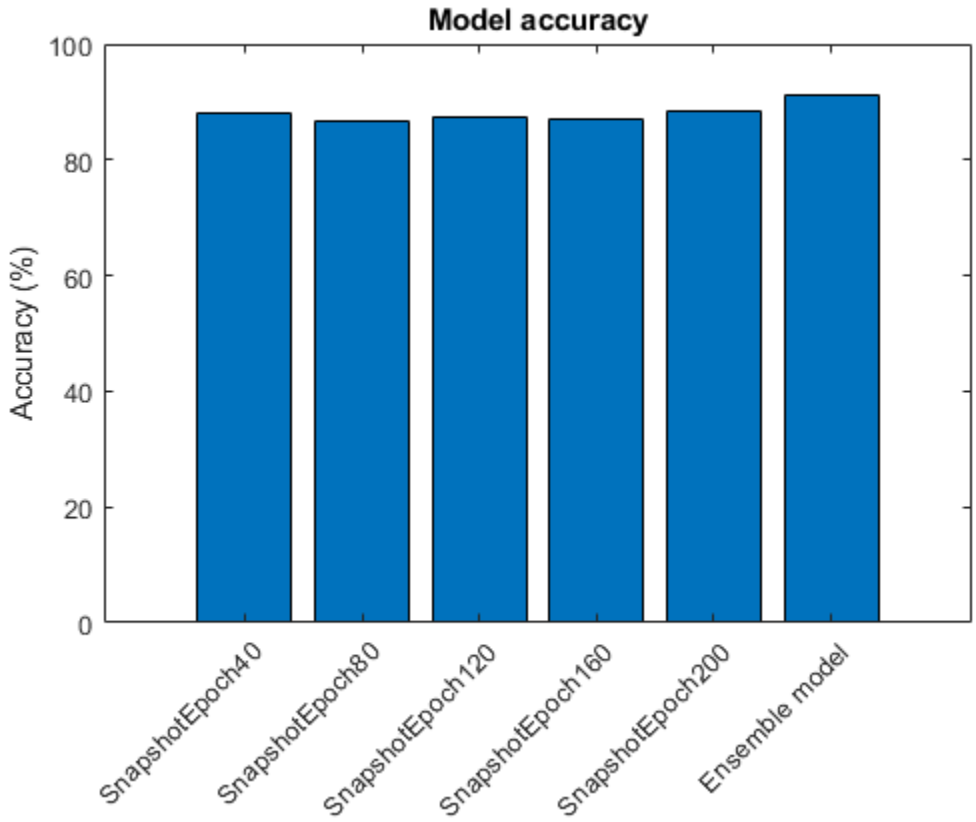
```
modelAccuracy(end) = computeAccuracy(mean(YPredictions,3), YTest, classes);
modelName{end} = "Ensemble model";
disp("Ensemble accuracy: " + modelAccuracy(end) + "%")
```

```
Ensemble accuracy: 91.13%
```

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

modelGradients 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.Parameter
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
```

computeAccuracy Function

The `computeAccuracy` function uses the network predictions, the true labels, and the number of classes to calculate the accuracy.

```
function accuracy = computeAccuracy(YPredictions, YTest, classes)
[~,I] = max(YPredictions,[],1);
C = classes(I);
accuracy = 100*(sum(C==YTest)/numel(C));
end
```

plotLossAndLearnRate Function

The `plotLossAndLearnRate` function plots the loss and learn rate at each iteration during training.

```
function [lossLine, learnRateLine] = plotLossAndLearnRate()
figure('Name','Training Progress');
clf
subplot(2,1,1); lossLine = animatedline;
title('Loss');
xlabel('Iteration')
ylabel('Loss')
grid on
subplot(2,1,2); learnRateLine = animatedline;
title('Learning rate');
xlabel('Iteration')
ylabel('Learning rate')
grid on
end
```

convolutionalUnit Function

`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] 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

`dlarray` | `dlfeval` | `dlgradient` | `dlnetwork` | `layerGraph` | `sgdmupdate` | `sigmoid`

More About

- “Train Generative Adversarial Network (GAN)” on page 3-72
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121
- “Make Predictions Using Model Function” on page 15-173
- “Specify Training Options in Custom Training Loop” on page 15-125
- “Automatic Differentiation Background” on page 15-112

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 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 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.
`dlconv`	The convolution operation applies sliding filters to the input data. Use 1-D and 2-D filters with ungrouped or grouped convolutions and 3-D filters with ungrouped convolutions.
`dltranspconv`	The transposed convolution operation upsamples feature maps.
`fullyconnect`	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
`gru`	The gated recurrent unit (GRU) operation allows a network to learn dependencies between time steps in time series and sequence data.
`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.

Function	Description
`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.
`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`.
`dlmtimes`	This function applies matrix multiplication to each page of two input `dlarray` objects. This is sometimes called batch matrix multiplication.
`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`.
`finddim`	This function finds the indices of `dlarray` dimensions with a given dimension label.
`stripdims`	This function removes the data format from a `dlarray`.

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`.
`cos`	
`cosh`	
`cot`	
`csc`	
`exp`	
`log`	• The output `dlarray` has the same data format as the input `dlarray`. • Because `dlarray` does not support complex numbers, the input `dlarray` must have nonnegative values.
`sec`	The output `dlarray` has the same data format as the input `dlarray`.
`sign`	
`sin`	
`sinh`	
`sqrt`	• The output `dlarray` has the same data format as the input `dlarray`. • Because `dlarray` does not support complex numbers, the input `dlarray` must have nonnegative values.
`tan`	The output `dlarray` has the same data format as the input `dlarray`.
`tanh`	
`uminus`, `-`	

Binary Element-wise Operators

Function	Notes and Limitations
`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 15-203.
`plus`, `+`	

Function	Notes and Limitations
`power`, `.^`	• 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 15-203. • Because `dlarray` does not support complex numbers, the software generates an error if any element of the output is complex.
`rdivide`, `./`	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 15-203.
`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`.
`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`	• When you find the maximum or minimum elements of a single `dlarray`, the output

Function	Notes and Limitations
`min`	`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 15-203.
`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`.

Other Math Operators

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 15-205.
	• All inputs must be real scalars. The output `dlarray` is unformatted.

`mrdivide`, `/`	The second `dlarray` input must be a scalar. The output `dlarray` has the same data format as the first `dlarray` input.
`mtimes`, `*`	One `dlarray` be formatted 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.

Logical Operations

Function	Notes and Limitations
`all`	The output `dlarray` has the same data format as the input `dlarray`.

Function	Notes and Limitations
`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 15-203.
`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 15-203.
`ge`, `>=`	
`gt`, `>`	
`le`, `<=`	
`lt`, `<`	
`ne`, `~=`	
`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 15-203.
`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.

Function	Notes and Limitations
`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 `dlarray` inputs are formatted, then the output `dlarray` is formatted with the same data format.
`horzcat`	
`vertcat`	

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`.

Function	Notes and Limitations

`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`.
`logical`	For a nonscalar input `dlarray`, the output is a `dlarray` that contains data of type `logical`. However, if the input is scalar, then the output is a basic (non-`dlarray`) logical value. To avoid this behavior, use `dlX~=0` instead of `logical(dlX)`. The command `dlX~=0` always returns a `dlarray` that contains data of type `logical`, even for scalar inputs.
`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.

Function	Notes and Limitations
`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 Identification Functions

Function	Notes and Limitations
`isfloat`	The software applies the function to the underlying data of an input `dlarray`.
`islogical`	
`isnumeric`	
`isreal`	Because `dlarray` does not support complex numbers, this function always returns `true` for a `dlarray` input.

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

Function	Notes and Limitations
`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`	
`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 15-203.

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` | `dlfeval` | `dlgradient` | `dlnetwork`

More About

- “Automatic Differentiation Background” on page 15-112
- “Define Custom Training Loops, Loss Functions, and Networks” on page 15-121

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” (MATLAB). 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™
`bigimageDatastore`	Datastore to manage blocks of single 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 16-3. For more information on how to preprocess data read from datastores, see “Transform and Combine Datastores” on page 16-4.

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” (MATLAB). 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

To use an image datastore as a source of training data, use the `imds` argument of `trainNetwork`. To use all other types of datastore as a source of training data, use the `ds` argument of `trainNetwork`. 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 (with the exception of `ImageDatastore`) must return data as either a cell array or a table.

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.

For networks with multiple inputs, the datastore must be a combined or transformed datastore that returns a cell array with (`numInputs`+1) columns containing the predictors and the responses, where `numInputs` is the number of network inputs and `numResponses` is the number of responses. For `i` less than or equal to `numInputs`, the `i`th element of the cell array corresponds to the input `layers.InputNames(i)`, where `layers` is the layer graph defining the network architecture. The last column of the cell array corresponds to the responses.

The table shows sample output of calling the `read` function for datastore `ds`.

Type of Network	Format of Read Data	Example Output

Single-input	Two-column cell array	`data = read(ds)`
		
		`data =`
		
		`4×2 cell array`
		
		`{28×28 double} {[7]}`
		`{28×28 double} {[7]}`
		`{28×28 double} {[9]}`
		`{28×28 double} {[9]}`

	Two-column table	`data = read(ds)`
		
		`data =`
		
		`4×2 table`
		
		`input response`
		`_____ ________`
		
		`{28×28 double} 7`
		`{28×28 double} 7`
		`{28×28 double} 9`
		`{28×28 double} 9`

Type of Network	Format of Read Data	Example Output

Multiple-input	(numInputs+1)-column cell array	`data = read(ds)`
		
		`data =`
		
		`4×3 cell array`
		
		`{28×28 double} {128×128 double}`
		`{28×28 double} {128×128 double}`
		`{28×28 double} {128×128 double}`
		`{28×28 double} {128×128 double}`

Inference

For inference using `predict`, `classify`, and `activations`, a datastore is only required to yield the columns corresponding to the predictors. The inference functions ignore additional columns of data beyond the first.

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.

For networks with multiple inputs, use a combined datastore to combine multiple data sources of data.

Transform Datastores

The `transform` function creates an altered form of a datastore, called an underlying datastore, by transforming the data read by the underlying datastore.

- 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” (MATLAB).
- 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” (MATLAB).

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 a function that performs the desired transformations of data, `data`, read from the underlying datastore. The function 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 two datastores of the same length to create the format expected of training and validation data. Combining datastores maintains the parity between the datastores. Each call to the `read` function of the resulting `CombinedDatastore` returns data from corresponding parts of 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`. Image datastores return data as a cell array, therefore 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 `ImageDatastores`, `PixelLabelDatastores`, and `TransformedDatastores`. 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. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

Many built-in datastores are already partitionable because they support the `partition` function. Transformed and combined datastores are partitionable if their underlying datastores are partitionable. 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.

Datastores do not support specifying the `'Shuffle'` name-value pair argument of `trainingOptions` as `'none'`.

See Also

`combine` | `read` | `trainNetwork` | `trainingOptions` | `transform`

Related Examples

- “Prepare Datastore for Image-to-Image Regression” on page 16-80
- “Classify Text Data Using Convolutional Neural Network” on page 4-82

More About

- “Getting Started with Datastore” (MATLAB)
- “Select Datastore for File Format or Application” (MATLAB)
- “Develop Custom Datastore” (MATLAB)

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 no 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.

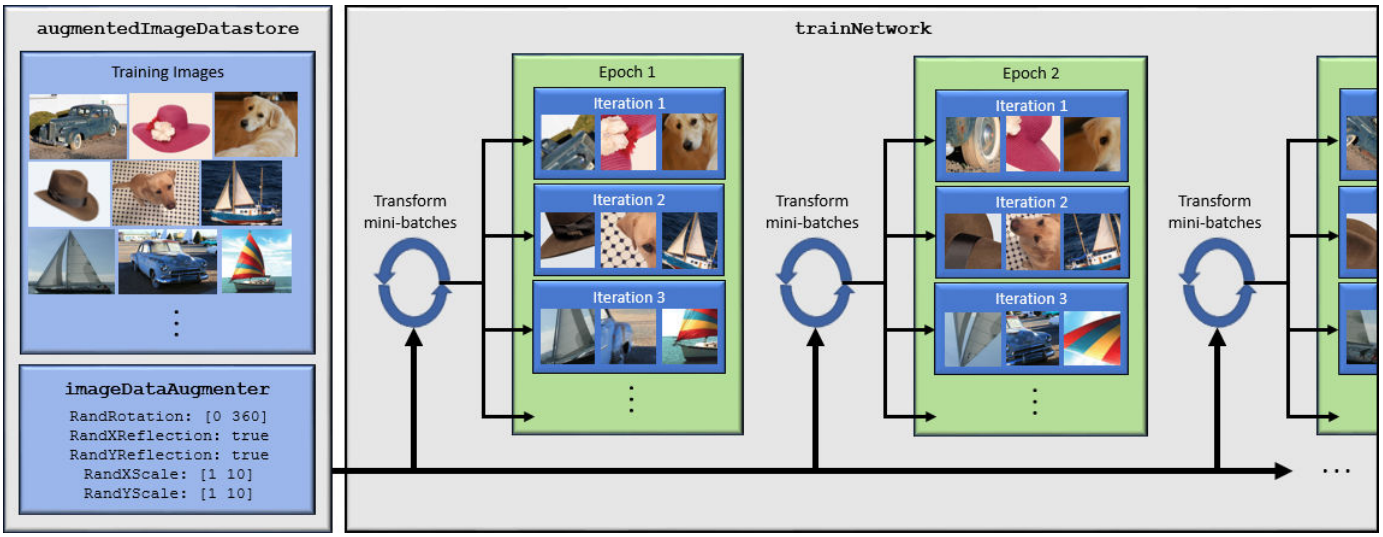
Resizing Option	Data Format	Resizing Function	Sample Code
Rescaling	• 3-D array representing a single color or multispectral image • 3-D array representing a stack of grayscale images • 4-D array representing a stack of images	`imresize`	`im = imresize(I,outputSize);` `outputSize` specifies the dimensions 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. 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 AlexNet” 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 8-74.
`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 16-2.

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

`ImageDatastore` | `combine` | `imresize` | `trainNetwork` | `transform`

Related Examples

- “Train Network with Augmented Images”
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Prepare Datastore for Image-to-Image Regression” on page 16-80

More About

- “Datastores for Deep Learning” on page 16-2
- “Preprocess Volumes for Deep Learning” on page 16-12
- “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`. 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. For more information, see “Datastores for Deep Learning” on page 16-2.

The table shows how to create an image or pixel label datastore for each of the supported file formats. 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	Create Pixel Label Datastore
MAT	`volds = imageDatastore(filepath, ... 'FileExtensions','.mat','ReadFcn',@(x) matRead(x));` `matRead` is a custom function that you write to read data from a .MAT file. For a sample implementation, see Define Custom Function to Read MAT Files on page 16-13.	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID 'FileExtensions','.mat','ReadFcn',@(x) matRead(x));` `matRead` is a custom function that you write to read data from a .MAT file. For a sample implementation, see Define Custom Function to Read MAT Files on page 16-13.
DICOM volume in single file	`volds = imageDatastore(filepath, ... 'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));` For more information about the DICOM file format, see `dicomread`.	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID 'FileExtensions','.dcm','ReadFcn',@(x) dicomread(x));` For more information about the DICOM file format, see `dicomread`.
DICOM volume in multiple files	Create an `ImageDatastore` from a collection of DICOM files by following these steps. • 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 an `ImageDatastore` from the collection of .MAT files. For an example of these steps, see Read Multi-File DICOM Volumes on page 16-13.	Create a `PixelLabelDatastore` from a collection of DICOM files by following these steps. • 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 a `PixelLabelDatastore` from the collection of .MAT files, class names, and pixel label IDs. For an example of these steps, see Read Multi-File DICOM Volumes on page 16-13.

Image File Format	Create Image Datastore	Create Pixel Label Datastore
NIfTI	`volds = imageDatastore(filepath, ... 'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));` For more information about the NIfTI file format, see `niftiread`.	`pxds = pixelLabelDatastore(filepath,classNames,pixelLabelID 'FileExtensions','.nii','ReadFcn',@(x) niftiread(x));` For more information about the NIfTI file format, see `niftiread`.

Define Custom Function to Read MAT Files

To read data from a .MAT file, you must define a custom read function. For example, this code creates 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)
% data = matRead(filename) reads the image data in the MAT-file filename

inp = load(filename);
f = fields(inp);
data = inp.(f{1});
end
```

Customize your custom read function according to how your image data is stored in .MAT files.

Example: Prepare Datastore Containing Single and Multi-File DICOM Volumes

This example shows how to create an `imageDatastore` or `PixelLabelDatastore` from a set of DICOM files that comprise a 3-D volume.

Specify the directory that contains the DICOM files. The directory can include files that contain a 2-D image, files that contain a complete 3-D volume, and files that contain 2-D slices of a 3-D volume. The datastore will include only the files that contain 3-D data, either as a complete 3-D volume in a single file or as 2-D slices of a 3-D volume.

```
dicomDir = fullfile(matlabroot,'toolbox','images','imdata');
```

Gather details about the DICOM files by using the `dicomCollection` function. This function returns the details as a table, where each row represents a single study. The function aggregates the files of a multi-file DICOM volume into a single study. The file names of a multi-file DICOM volume are listed in a string array in the `Filenames` variable.

```
collection = dicomCollection(dicomDir,'IncludeSubfolders',true)
```

Create a directory to store the processed DICOM volumes.

```
matFileDir = fullfile(tempdir,'MATFiles');
if ~exist(matFileDir,'dir')
    mkdir(matFileDir)
end
```

For every study in the collection, get the file names that comprise the study. Try reading the data of the study by using the `dicomreadVolume` function.

- If the data is contained in multiple files, then `dicomreadVolume` runs successfully and returns the complete volume in a single 4-D array. This volume can be included in the datastore.

- If the data is contained in a single file, then `dicomreadVolume` does not run successfully. In this case, read the data by using the `dicomread` function.
  - If `dicomread` returns a 4-D array, then the study contains a complete 3-D volume that can be included in the datastore.
  - If `dicomread` returns a 2-D matrix or 3-D array, then the study contains a single 2-D image. In this case, omit the image data from the datastore and move on to the next study in the collection.

For complete volumes returned in a 4-D array, write the data to a .MAT file. This example also writes the absolute file name of the DICOM file, `'dicomFileName'`, as a second variable. For multiple file DICOM volumes, `'dicomFileName'` is a string array of all individual DICOM files.

```
for idx = 1:numel(collection.Row)
    dicomFileName = collection.Filenames{idx};
    if length(dicomFileName) > 1
        matFileName = fileparts(dicomFileName(1));
        matFileName = split(matFileName,filesep);
        matFileName = replace(strtrim(matFileName(end))," ","_");
    else
        [~,matFileName] = fileparts(dicomFileName);
    end
    matFileName = fullfile(matFileDir,matFileName);

    try
        V = dicomreadVolume(collection,collection.Row{idx});
    catch ME
        V = dicomread(dicomFileName);
        if ndims(V)<4
            % Skip files that are not volumes
            continue;
        end
    end

    % For multi-file DICOM, dicomFileName is a string array.
    save(matFileName,'V','dicomFileName');

end
```

If the volumes represent image data, then create an `imageDatastore` from the .MAT files containing the volumes. You can specify the `ReadFcn` property as the `matRead` function from Define Custom Function to Read MAT Files on page 16-13.

```
imdsdicom = imageDatastore(matFileDir,'FileExtensions','.mat', ...
    'ReadFcn',@matReader);
```

If the volumes represent pixel label data, then create a `PixelLabelDatastore` from the .MAT files containing the volumes. You can specify the `ReadFcn` property as the `matRead` function from Define Custom Function to Read MAT Files on page 16-13. The arguments `classNames` and `pixelLabelID` are vectors that specify the mapping of voxel label values to class names.

```
pxdsdicom = pixelLabelDatastore(matFileDir,classNames,pixelLabelID, ...
    'FileExtensions','.mat','ReadFcn',@(x) matRead(x));
```

Associate 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 several benefits over `combine`.

- `randomPatchExtractionDatastore` supports parallel training, multi-GPU training, and prefetch reading. Specify parallel or multi-GPU training using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Specify prefetch reading using the `'DispatchInBackground'` name-value pair 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`.

`RandomPatchExtractionDatastore` does not support the `DataAugmentation` property for volumetric data. To apply random affine transformations to volumetric data, you must use `transform`.

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.

Example: Transform Volumetric Data in Image Datastore

This sample code shows how to transform volumetric data in image datastore `volds` using an arbitrary preprocessing pipeline defined in the function `preprocessVolumetricIMDS`. The example assumes that the `ReadSize` of `volds` is greater than 1.

```
dsTrain = transform(volds,@(x) preprocessVolumetricIMDS(x,inputSize));
```

Define the `preprocessVolumetricIMDS` function that performs the desired transformations of data read from the underlying datastore. 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 dataOut = preprocessVolumetricIMDS(data,inputSize)

numRows = size(data,1);
dataOut = cell(numRows,1);

for idx = 1:numRows

    % Perform randomized 90 degree rotation about the z-axis
    data = imrotate3(data{idx,1},90*(randi(4)-1),[0 0 1]);

    % Resize the volume to the size expected by the network
    dataClean = imresize(data,inputSize);

    % Add zero-mean Gaussian noise with a normalized variance of 0.01
    dataNoisy = imnoise(dataClean,'gaussian',0.01);

    % Return the preprocessed data
    dataOut(idx) = dataNoisy;

end
end
```

Example: Transform Volumetric Data in Random Patch Extraction Datastore

This sample code shows how to transform volumetric data in random patch extraction datastore `volds` using an arbitrary preprocessing pipeline defined in the function `preprocessVolumetricPatchDS`. The example assumes that the `ReadSize` of `volds` is 1.

```
dsTrain = transform(volds,@preprocessVolumetricPatchDS);
```

Define the `preprocessVolumetricPatchDS` function that performs the desired transformations of data read from the underlying 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 dataOut = preprocessVolumetricPatchDS(data)

img = data(1);
resp = data(2);

% 5 augmentations: nil,rot90,fliplr,flipud,rot90(fliplr)
augType = {@(x) x,@rot90,@fliplr,@flipud,@(x) rot90(fliplr(x))};

rndIdx = randi(5,1);
imgOut = augType{rndIdx}(img);
respOut = augType{rndIdx}(resp);

% Return the preprocessed data
dataOut = table(imgOut,respOut};

end
```

See Also

`imageDatastore` | `pixelLabelDatastore` | `randomPatchExtractionDatastore` | `trainNetwork` | `transform`

Related Examples

- “3-D Brain Tumor Segmentation Using Deep Learning” on page 8-112

More About

- “Datastores for Deep Learning” on page 16-2
- “Deep Learning in MATLAB” on page 1-2
- “Create Functions in Files” (MATLAB)

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 16-2. 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.

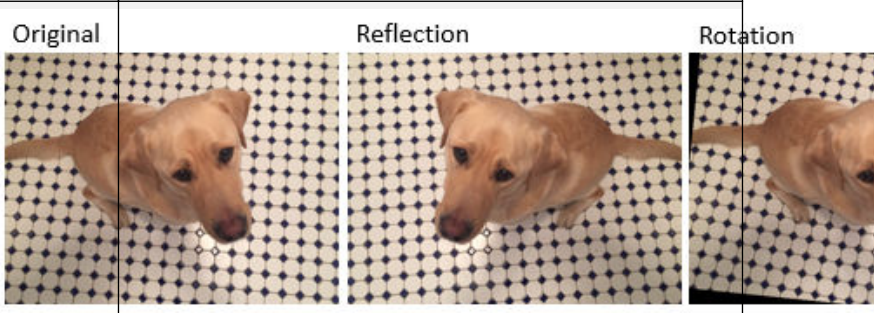
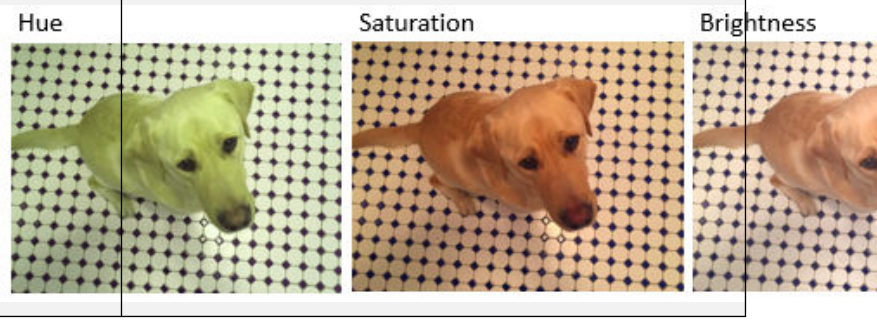
To augment training data, start by loading your data into a datastore. For more information, see “Datastores for Deep Learning” on page 16-2. Some built-in datastores apply a specific and limited set of augmentation to data for specific applications. You can also apply your own set of augmentation operations 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.

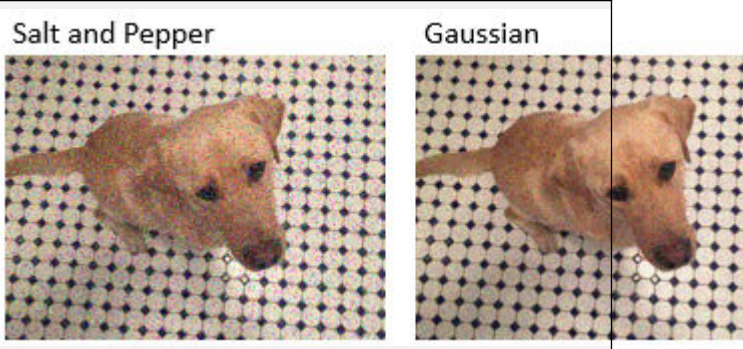
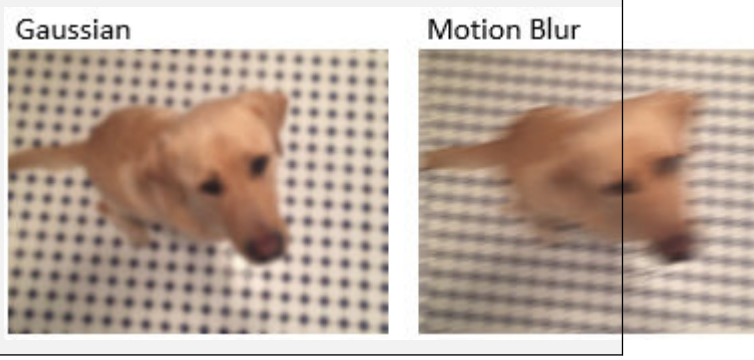
Image Processing Applications

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 16-34.

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	• `centerCropW` `indow2d`, `centerCropW` `indow3d` • `randomCropW` `indow2d`, `randomCropW` `indow3d`	
Jitter color	Randomly adjust image hue, saturation, brightness, or contrast	• `jitt` `HSV`	

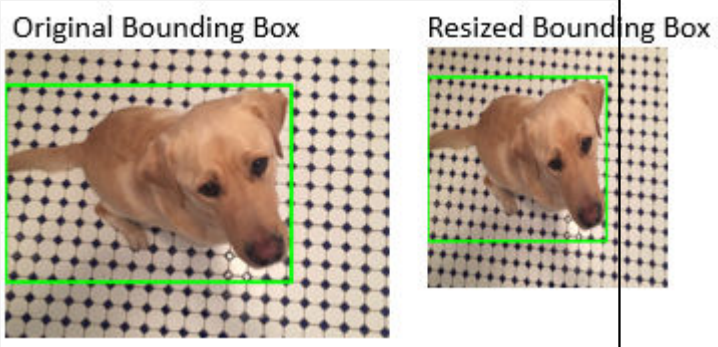
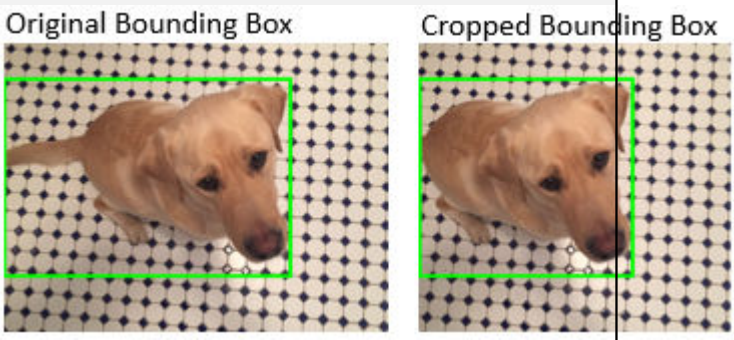
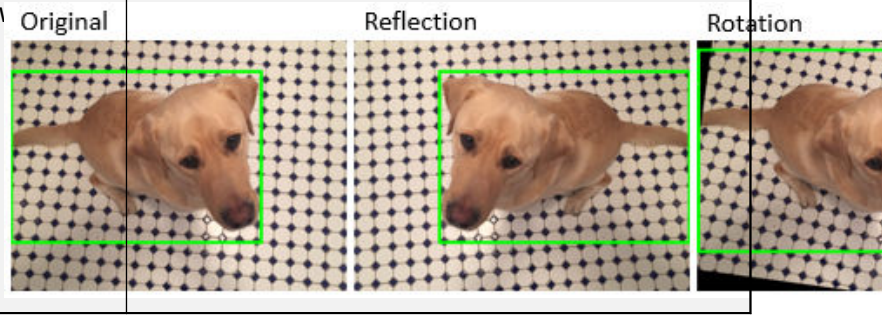
Processing Type	Description	Sample Functions	Sample Output
Simulate noise	Add random Gaussian, Poisson, salt and pepper, or multiplicative noise	• `imnoise`	
Simulate blur	Add Gaussian or directional motion blur	• `imgaussfilt`, `imgaussfilt3` • `imfilter`	

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 16-67. For more information, see “Getting Started with Object Detection Using Deep Learning” (Computer Vision Toolbox).

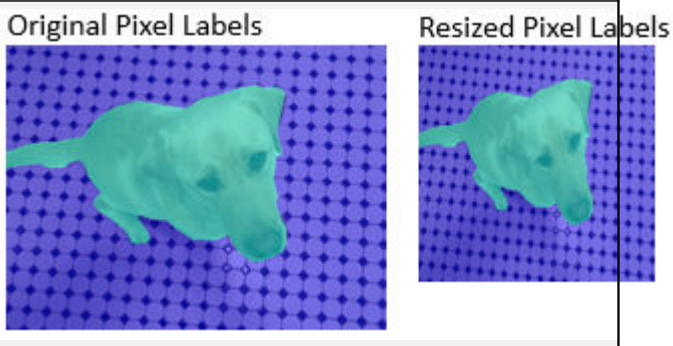
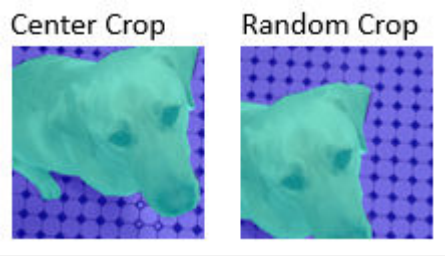
Processing Type	Description	Sample Functions	Sample Output
Resize bounding boxes	Resize bounding boxes by a fixed scaling factor or to a target size	• `bboxresize`	
Crop bounding boxes	Crop a bounding box to a target size from the center or a random position	• `bboxcrop`	
Warp bounding boxes	Apply reflection, rotation, scale, shear, and translation to bounding boxes	• `bbox`	

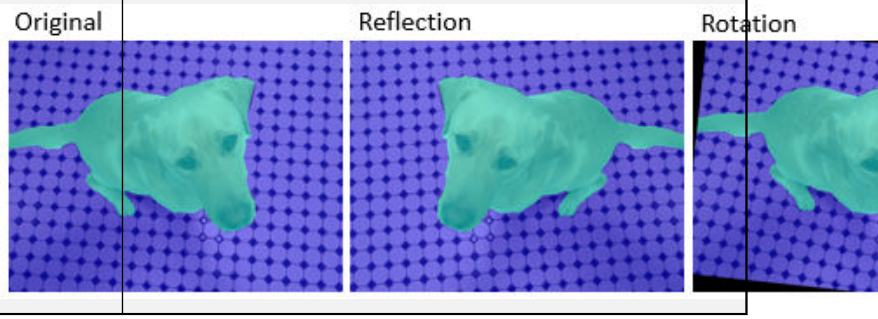
Semantic Segmentation

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 16-57. 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` • `randomCropWindow2d`, `randomCropWindow3d`	

Warp pixel labels	Apply random reflection, rotation, scale, shear, and translation to pixel label images	• `rand`
		`e2d`,
		`rand`
		`e3d`
			

Signal Processing Applications

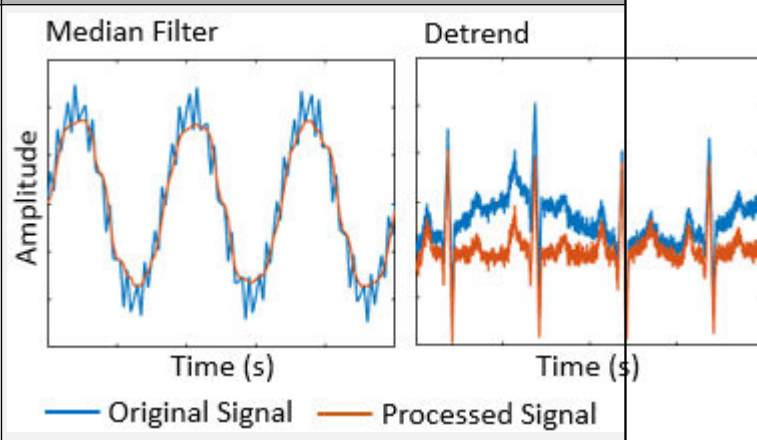
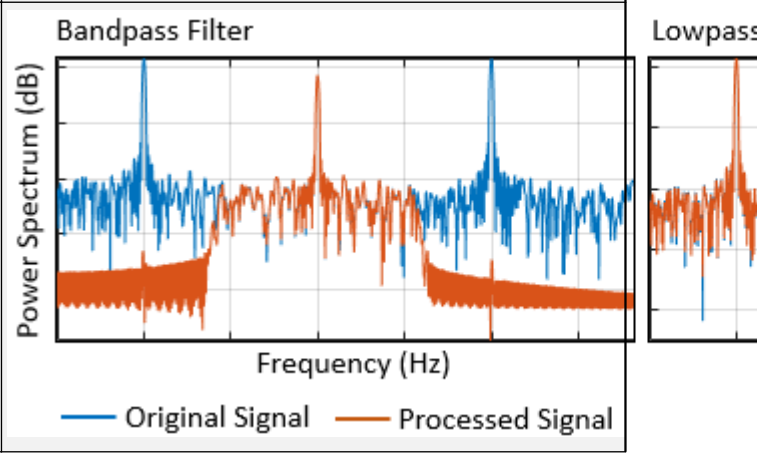
Signal Processing Toolbox™ enables you to denoise, smooth, detrend, and resample signals. You can augment training data with noise, multipath fading, and synthetic signals such as pulses and chirps. You can also create labeled sets of signals by using the **Signal Labeler** app and the `labeledSignalSet` object. For an example that shows how to create and apply these transformations, see “Waveform Segmentation Using Deep Learning” on page 11-42.

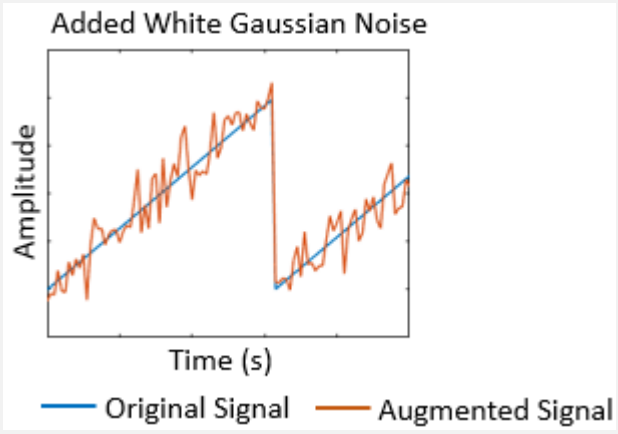
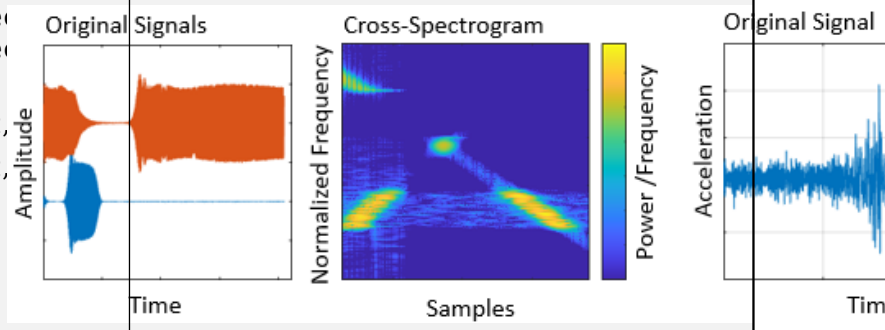
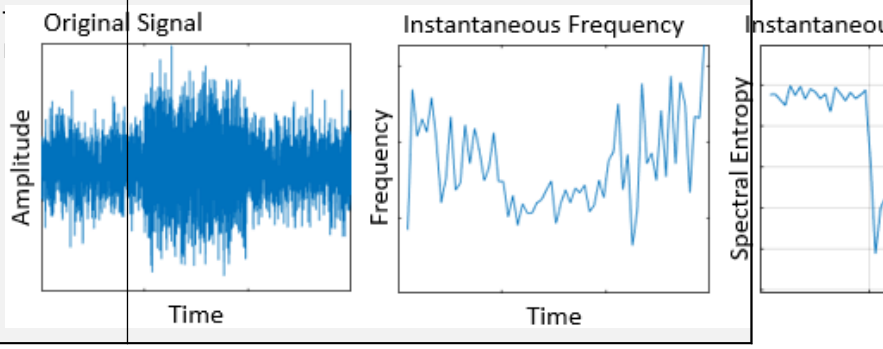
Wavelet Toolbox™ and Signal Processing Toolbox enable you to generate 2-D time-frequency representations of time series data that you can use as image inputs for signal classification applications. For an example, see “Classify Time Series Using Wavelet Analysis and Deep Learning” on page 11-93. Similarly, 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 11-60.

You can process signal data using the functions in the table as well as any other functionality in each toolbox.

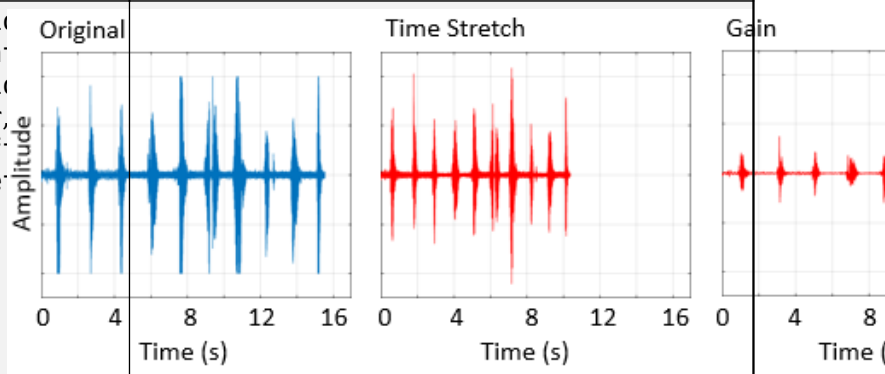
Processing Type	Description	Sample Functions	Sample Output
Clean signals	• Apply median filtering or moving average to signal • Remove polynomial trend • Resample signal to new fixed rate	• `medfilt1`, `smoothdata` • `detrend` • `downsample`, `interp`, `upsample`	
Filter signals	• Perform lowpass, highpass, 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`	

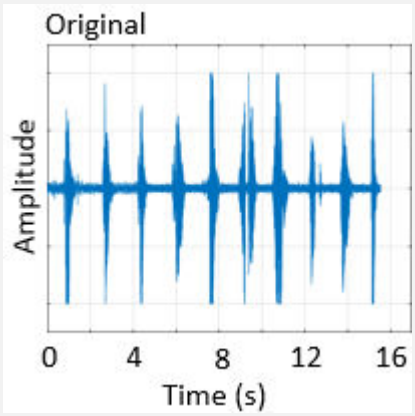
Processing Type	Description	Sample Functions	Sample Output
Augment signals	• Add white Gaussian noise to signal using Communications Toolbox • Adjust time information of the signal, and perform multipath fading using Communications Toolbox • Add synthetic chirps and waveforms	• `awgn` • `chirp`, `square`, `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`	

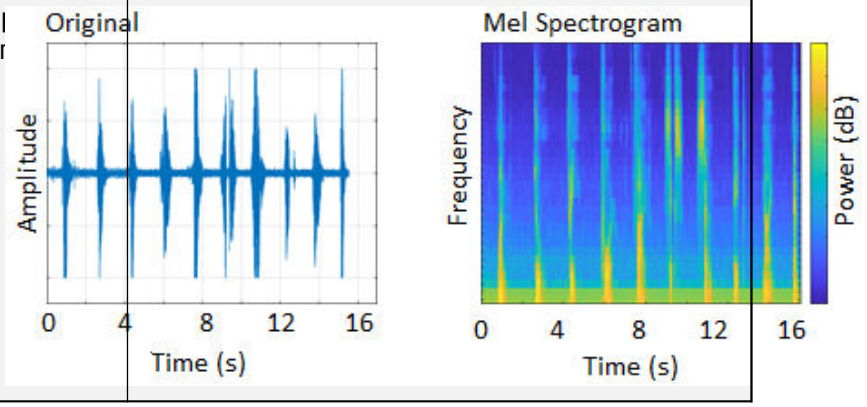
Audio Processing Applications

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

`combine` | `read` | `trainNetwork` | `trainingOptions` | `transform`

More About

- "Datastores for Deep Learning" on page 16-2
- "Select Datastore for File Format or Application" (MATLAB)

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 16-94.

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 16-28.
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 16-32.

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 16-32.

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

1 Begin defining your class. Inherit from the base class `matlab.io.Datastore` and the `matlab.io.datastore.MiniBatchable` class.

```
classdef MySequenceDatastore < matlab.io.Datastore & ...
                               matlab.io.datastore.MiniBatchable
```

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” (MATLAB).
- 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 *D*-by-*S*, where *D* 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 16-28 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” (MATLAB).

See Also

`trainNetwork`

Related Examples

- “Train Network Using Custom Mini-Batch Datastore for Sequence Data” on page 16-94

More About

- “Getting Started with Datastore” (MATLAB)
- “Develop Custom Datastore” (MATLAB)
- “Developing Classes — Typical Workflow” (MATLAB)
- “Testing Guidelines for Custom Datastores” (MATLAB)
- “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 16-0
- Cropping Transformations on page 16-0
- Color Transformations on page 16-0
- Synthetic Noise on page 16-0
- Synthetic Blur on page 16-0

The example then shows how to apply augmentation to image data in datastores on page 16-0 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 16-80.

Read and display a sample image. To compare the effect of the different types of image augmentation, each transformation uses the same input image.

```
peppers = imread('kobi.png');
imshow(peppers)
```

Random Image Warping Transformations

The `randomAffine2d` 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 two-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` by using the `affineOutputView` 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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

Shear

Create a horizontal shear transformation with shear angle selected randomly from the range [-30,30].

```
tform = randomAffine2d('XShear',[-30 30]);
outputView = affineOutputView(size(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView);
imshow(augmented)
```

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(peppers),tform);
augmented = imwarp(peppers,tform,'OutputView',outputView,'FillValues',[128 128 128]);
imshow(augmented)
```

Cropping Transformations

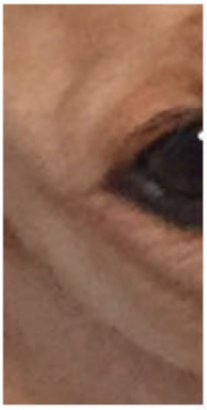
To create output images of a desired size, use the `randomCropWindow2d` and `centerCropWindow2d` functions. Be careful to select a cropping window that includes the desired content in the image.

Specify the desired size of the cropped region as a 2-element vector of the form [*height, width*].

```
targetSize = [200,100];
```

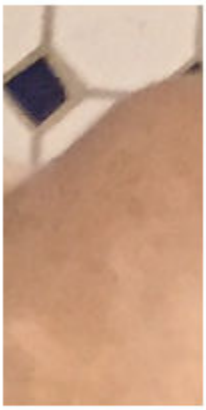
Crop the image to the target size from the center of the image.

```
win = centerCropWindow2d(size(peppers),targetSize);
peppersCenterCrop = imcrop(peppers,win);
imshow(peppersCenterCrop)
```



Crop the image to the target size from a random location in the image.

```
win = randomCropWindow2d(size(peppers),targetSize);
peppersRandomCrop = imcrop(peppers,win);
imshow(peppersRandomCrop)
```



Color Transformations

You can randomly adjust the hue, saturation, brightness, and contrast of a color image by using the `jitterColorHSV` 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.

```
augmented = jitterColorHSV(peppers,'Hue',[0.05 0.15]);
montage({peppers,augmented})
```

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.

```
augmented = jitterColorHSV(peppers,'Saturation',[-0.4 -0.1]);
montage({peppers,augmented})
```

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.

```
augmented = jitterColorHSV(peppers,'Brightness',[-0.3 -0.1]);
montage({peppers,augmented})
```

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.

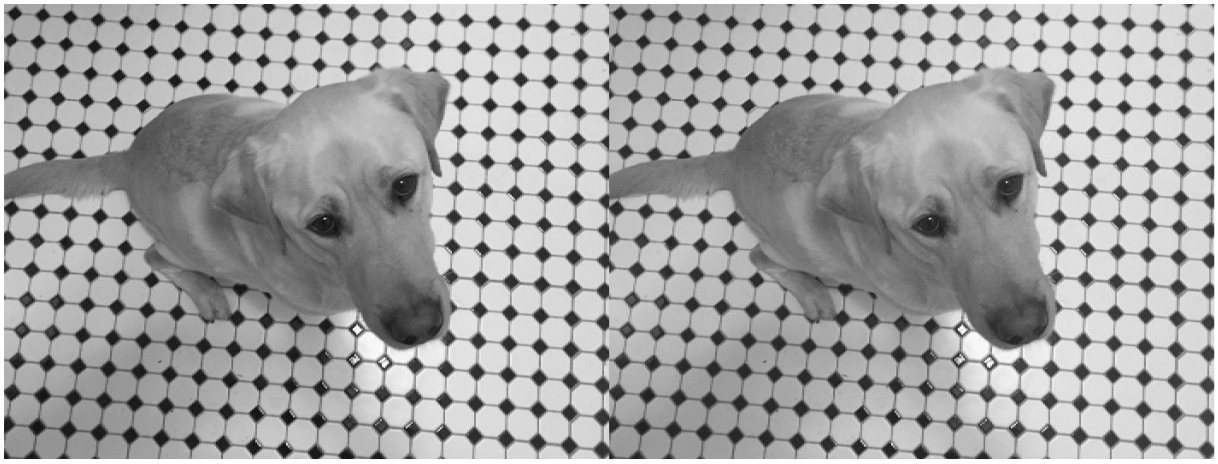
```
augmented = jitterColorHSV(peppers,'Contrast',[1.2 1.4]);
montage({peppers,augmented})
```

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.

```
peppersGray = rgb2gray(im2double(peppers));
contrastFactor = 1-0.2*rand;
brightnessOffset = 0.3*(rand-0.5);
peppersJittered = peppersGray.*contrastFactor + brightnessOffset;
peppersJittered = im2uint8(peppersJittered);
montage({peppersGray,peppersJittered})
```



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
    peppersGray = repmat(rgb2gray(peppers),[1 1 3]);
end
imshow(peppersGray)
```

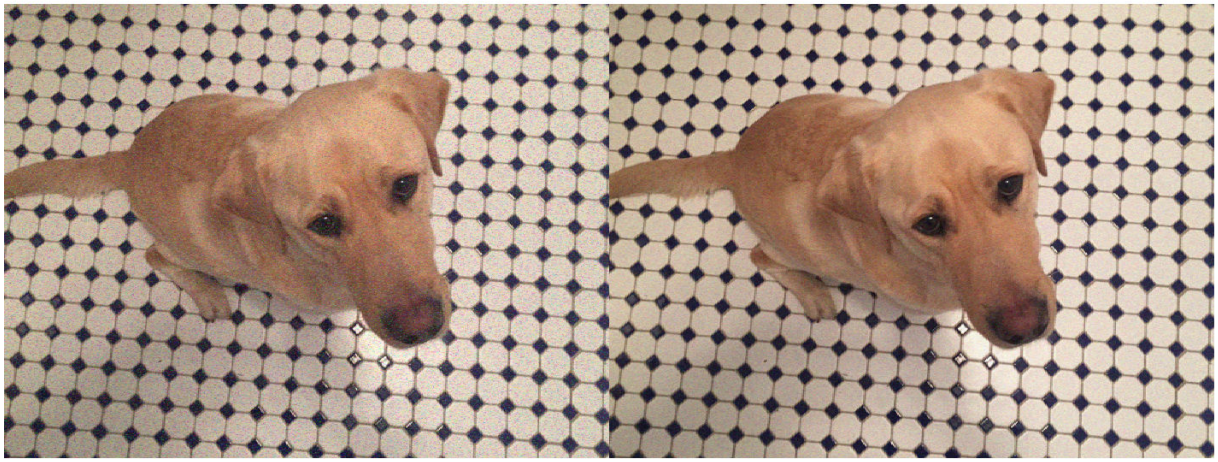
Other Image Processing Operations

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` 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.

```
peppersSaltAndPepper = imnoise(peppers,'salt & pepper',0.1);
peppersGaussian = imnoise(peppers,'gaussian');
montage({peppersSaltAndPepper,peppersGaussian})
```



Synthetic Blur

To apply randomized Gaussian blur to an image, use the `imgaussfilt` function. You can specify the amount of smoothing.

```
sigma = 1+5*rand;
blurredPeppers = imgaussfilt(peppers,sigma);
imshow(blurredPeppers)
```

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.

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")
```

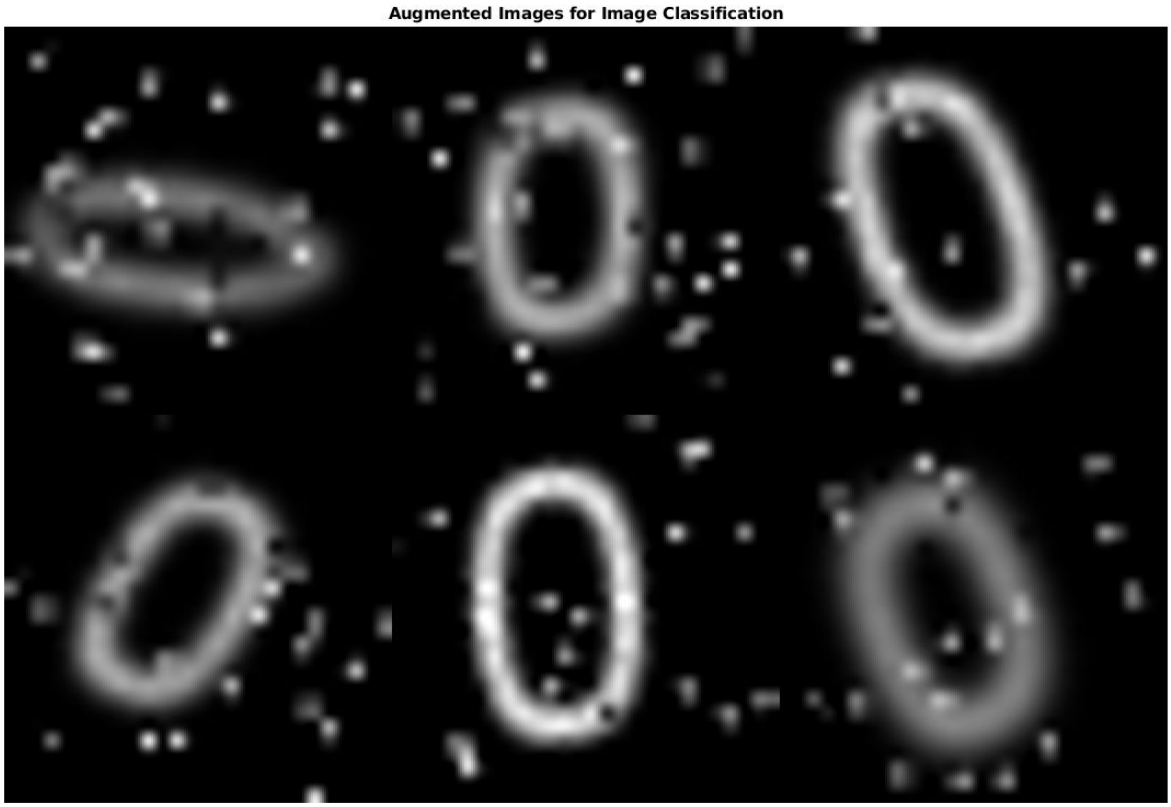
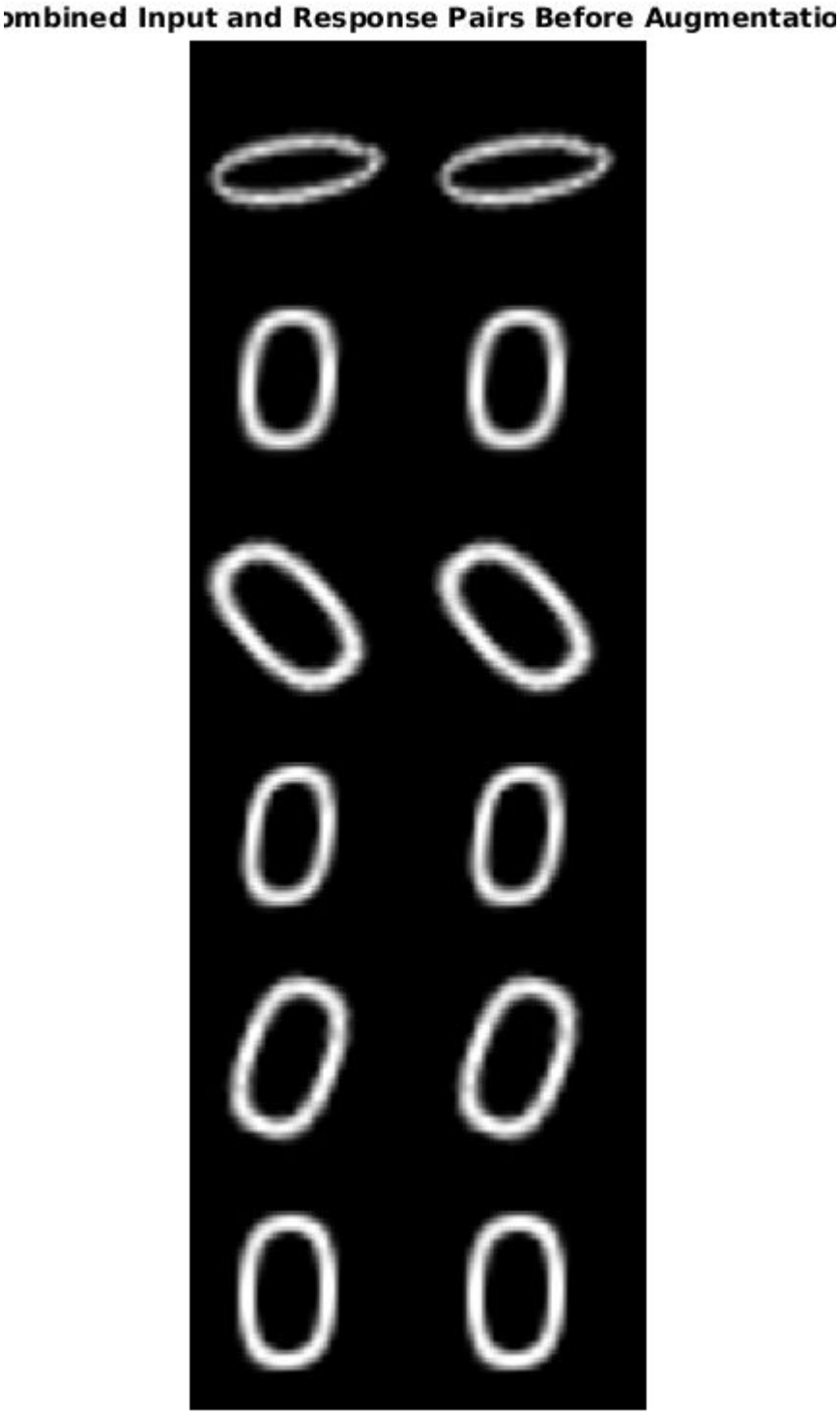


Image Regression

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")
```

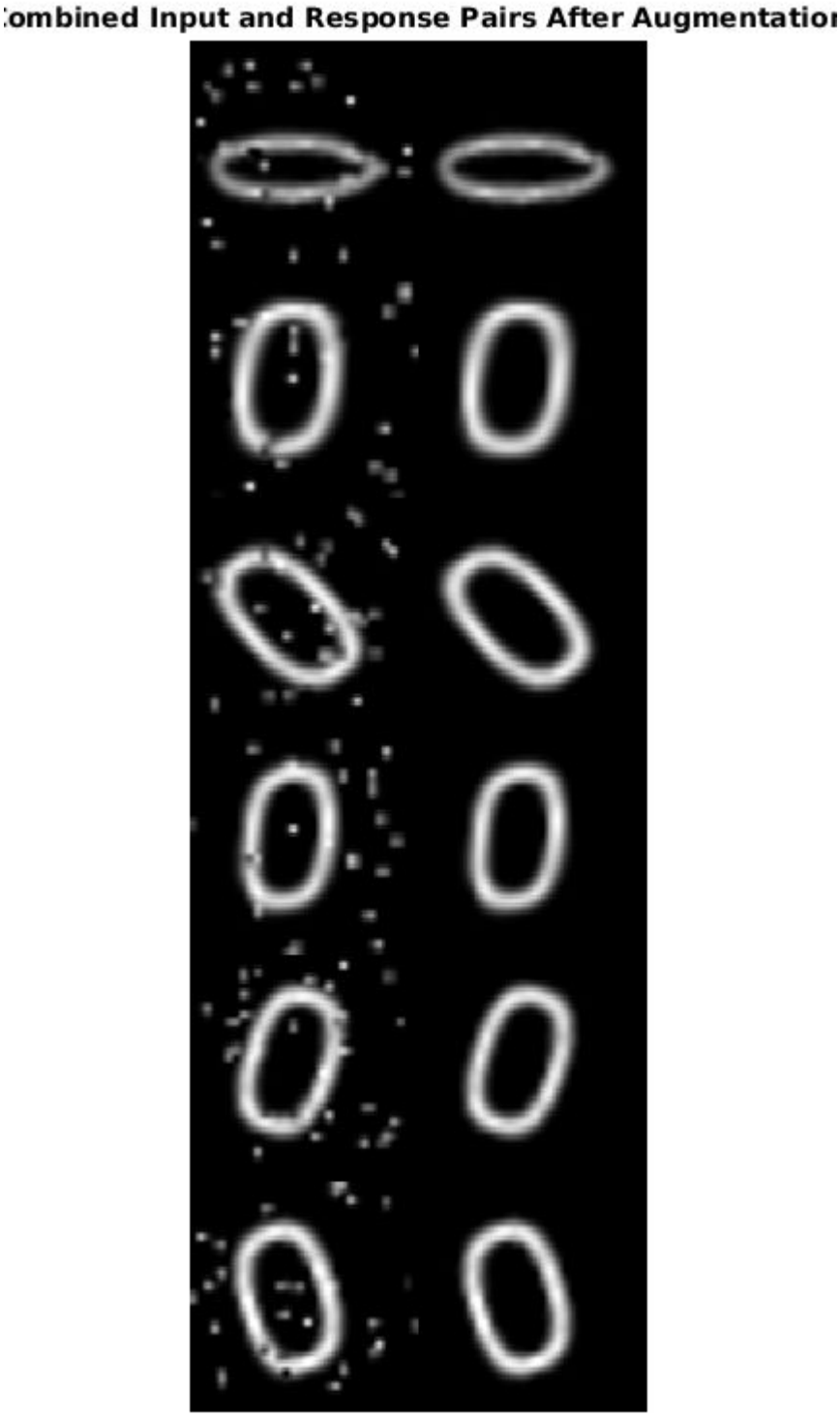


Augment each pair of training images with a series of image processing operations:

- 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 Augmentatio

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 16-80.

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 network response image.

```
function [dataOut,info] = classificationAugmentationPipeline(dataIn,info)

dataOut = cell([size(dataIn,1),2]);

for idx = 1:size(dataIn,1)
    temp = dataIn{idx};

    % 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 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)

    inputImage = im2single(imresize(dataIn{idx,1},[32 32]));
    targetImage = im2single(imresize(dataIn{idx,2},[32 32]));

    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

`combine` | `transform`

Related Examples

- “Prepare Datastore for Image-to-Image Regression” on page 16-80

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 16-19
- “Preprocess Images for Deep Learning” on page 16-8

Augment Pixel Labels for Semantic Segmentation

This example shows how to use MATLAB®, Computer Vision Toolbox™, and Image Processing Toolbox™ 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 16-0
- Crop Image and Pixel Labels on page 16-0
- Warp Image and Pixel Labels on page 16-0

The example then shows how to apply augmentation to semantic segmentation training data in datastores on page 16-0 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.

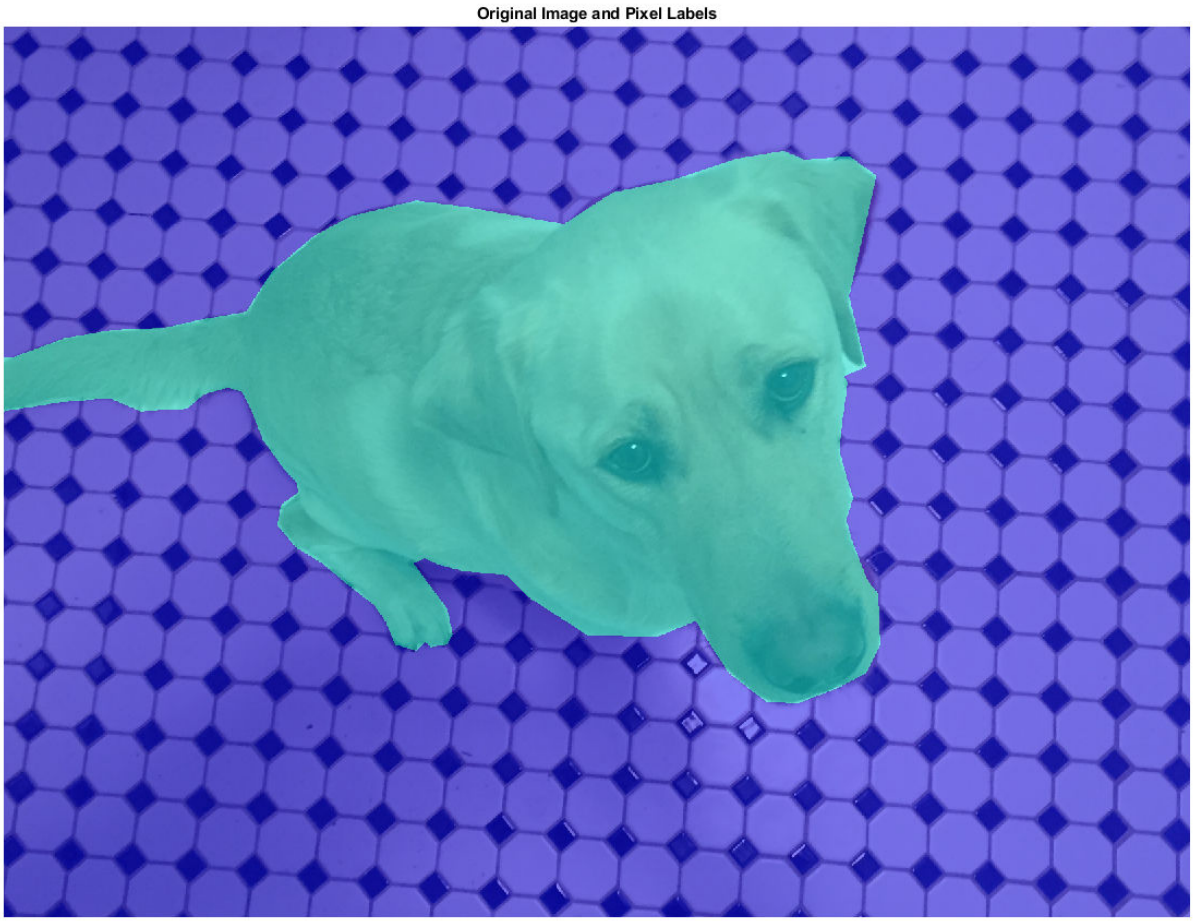
```
filenameLabels = 'kobiPixelLabeled.png';
L = imread(filenameLabels);
classes = ["floor","dog"];
ids = [1 2];
```

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')
```



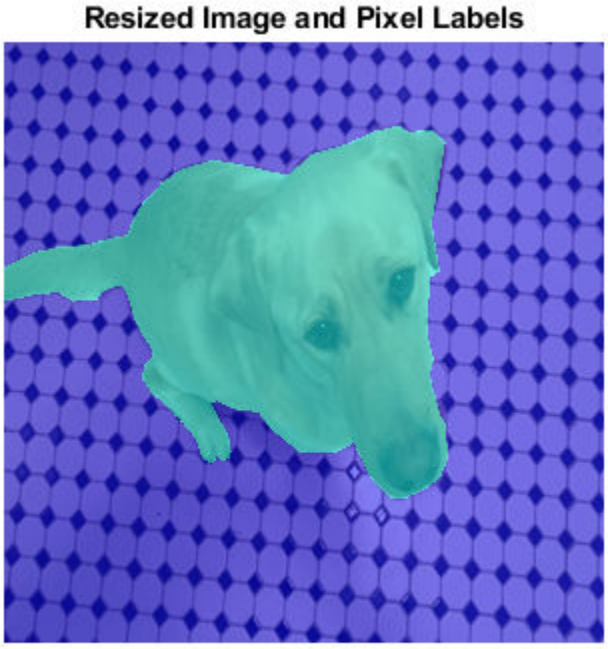
Resize Image and Pixel Labels

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 `randomCropWindow2d` and `centerCropWindow2d` 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*].

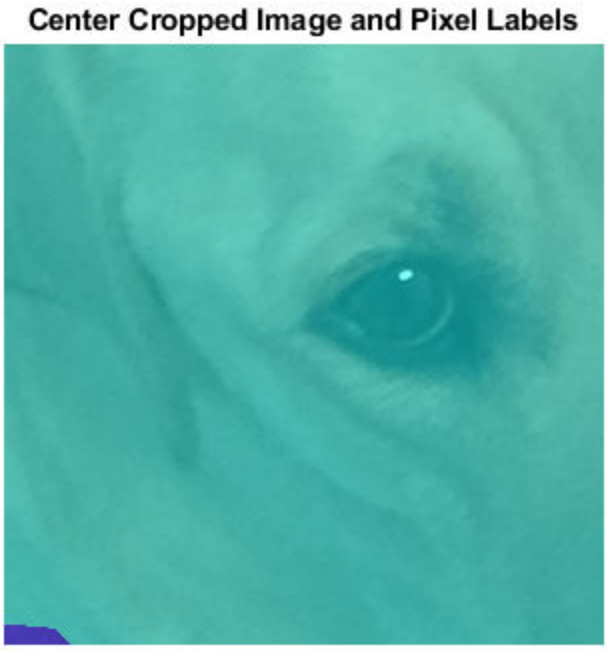
```
targetSize = [300 300];
```

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')
```

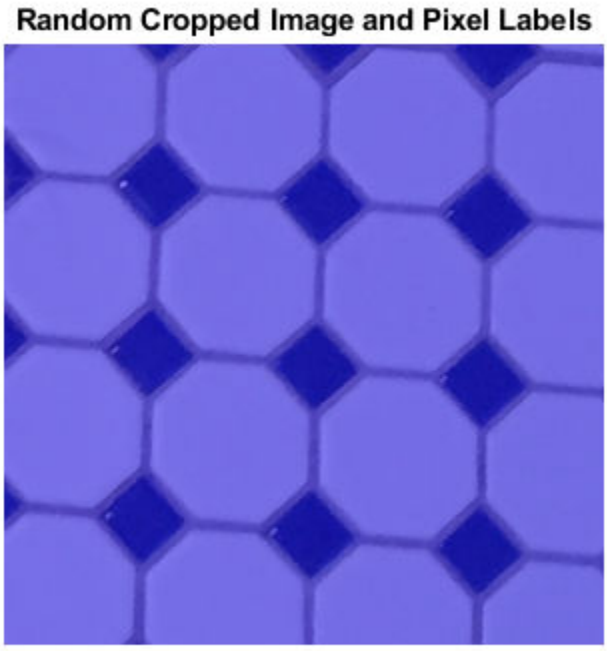


Crop the image to the target size from a random position in the image.

```
win = randomCropWindow2d(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` 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`. Control the spatial bounds and resolution of the warped output by using the `affineOutputView` 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.

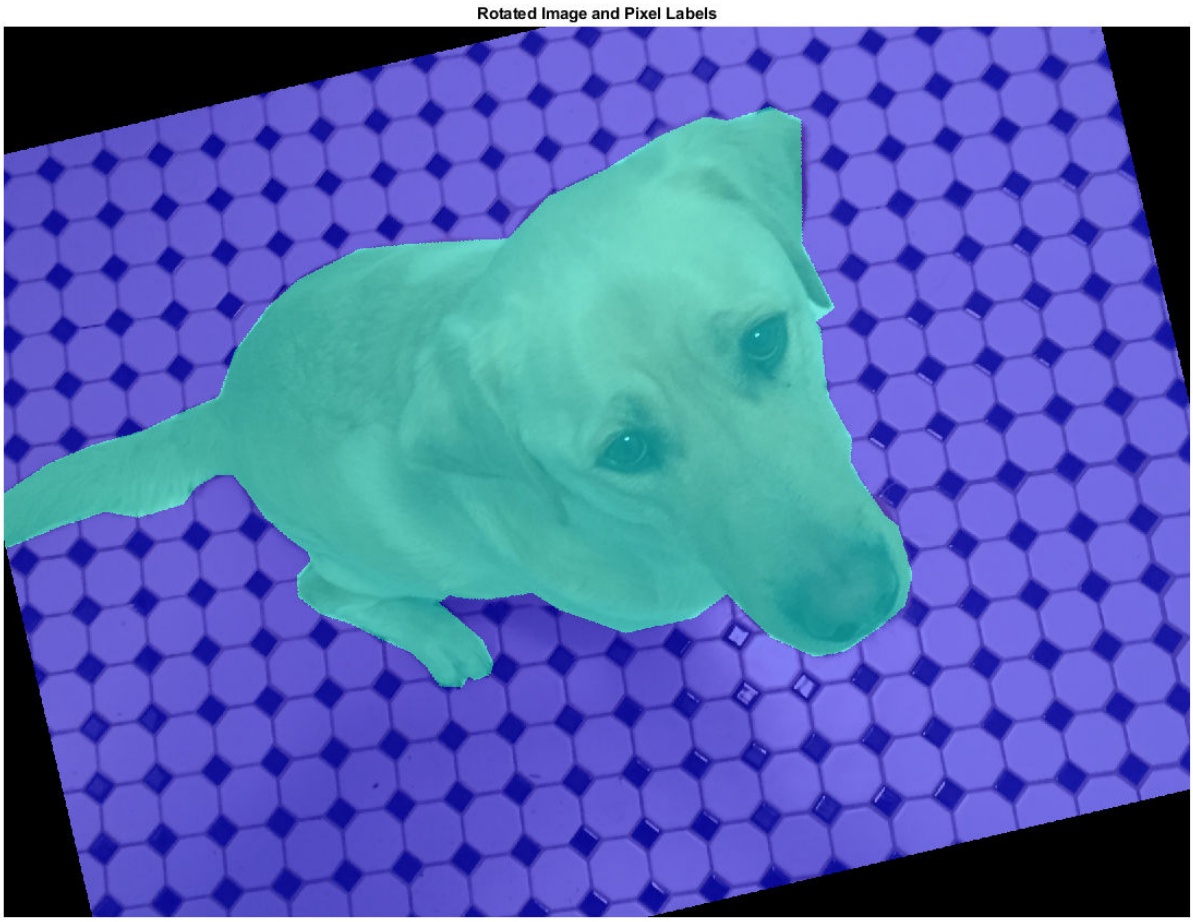
```
rout = affineOutputView(size(I),tform);
```

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.

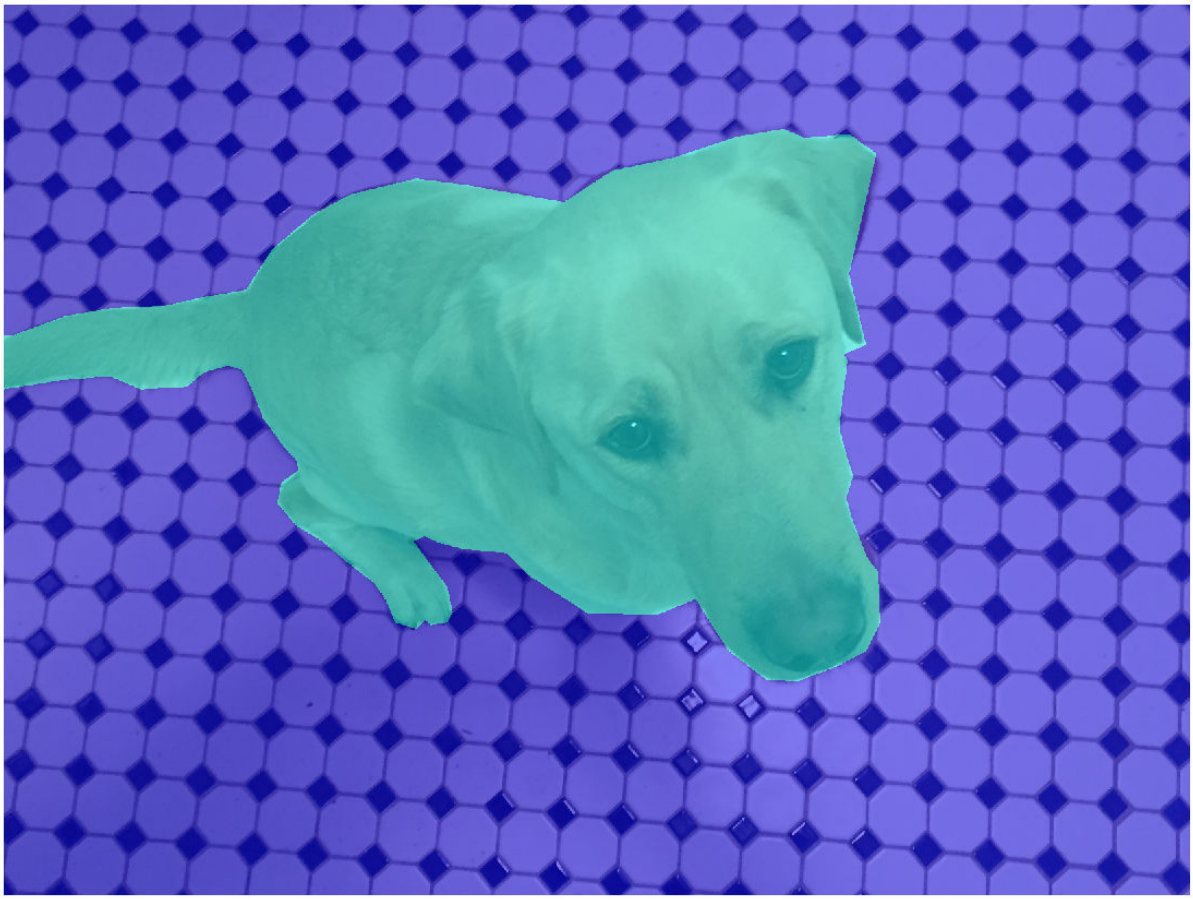
```
trainingData = combine(imds,pxds);
```

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.

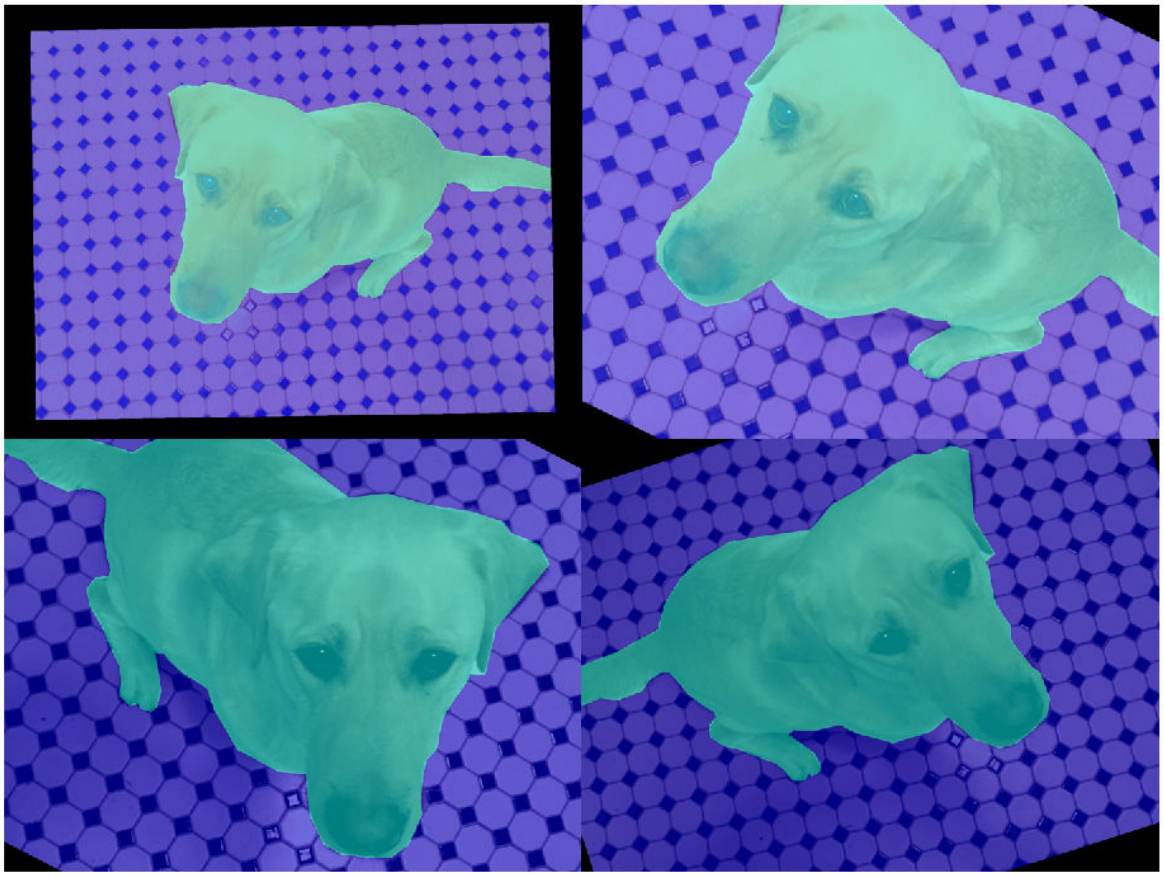
```
augmentedTrainingData = transform(trainingData,@jitterImageColorAndWarp);
```

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.

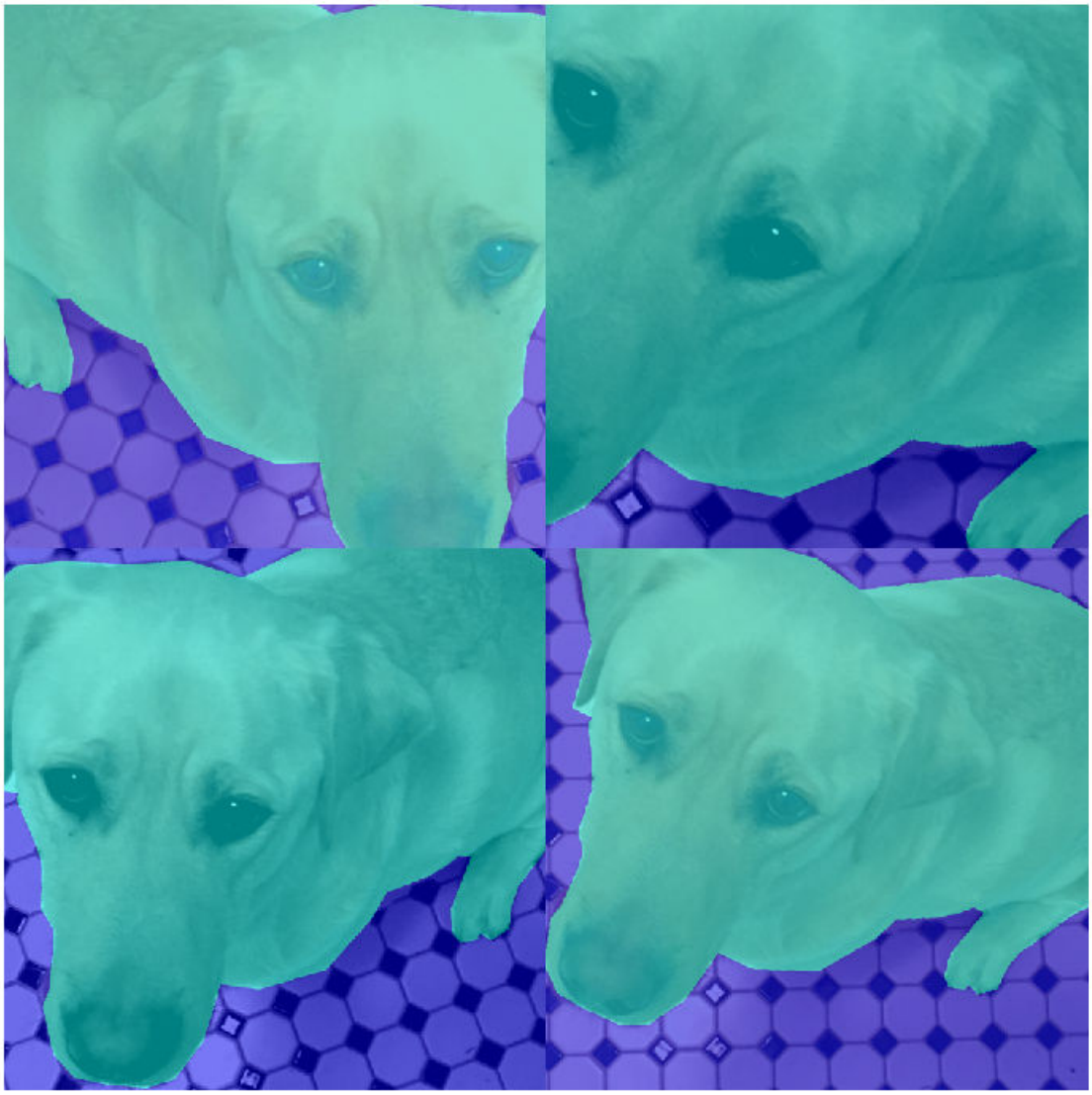
```
targetSize = [800 800];
preprocessedTrainingData = transform(augmentedTrainingData,...
    @(data)centerCropImageAndLabel(data,targetSize));
```

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

`centerCropWindow2d` | `randomAffine2d` | `randomCropWindow2d`

Related Examples

- “Augment Images for Deep Learning Workflows Using Image Processing Toolbox” on page 16-34
- “Semantic Segmentation Using Deep Learning” on page 8-74

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 16-19
- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Augment Bounding Boxes for Object Detection

This example shows how to use MATLAB®, Computer Vision Toolbox™, and Image Processing Toolbox™ 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 16-0
- Crop Image and Bounding Box on page 16-0
- Warp Image and Bounding Box on page 16-0

The example then shows how to apply augmentation to training data in datastores on page 16-0 using a combination of multiple types of transformations.

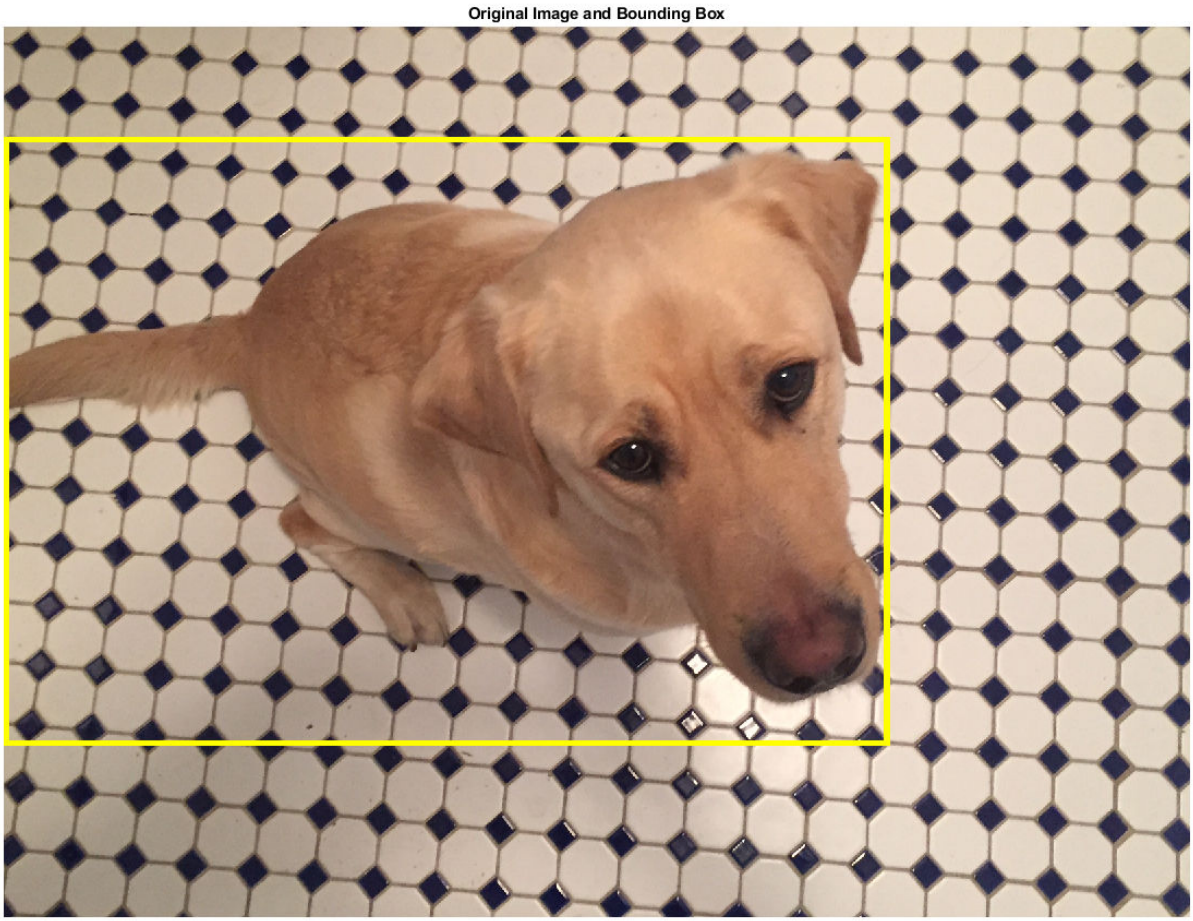
You can use augmented training data to train a network. For an example showing how to train an object detection network, see “Object Detection Using Faster R-CNN Deep Learning” (Computer Vision Toolbox).

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.

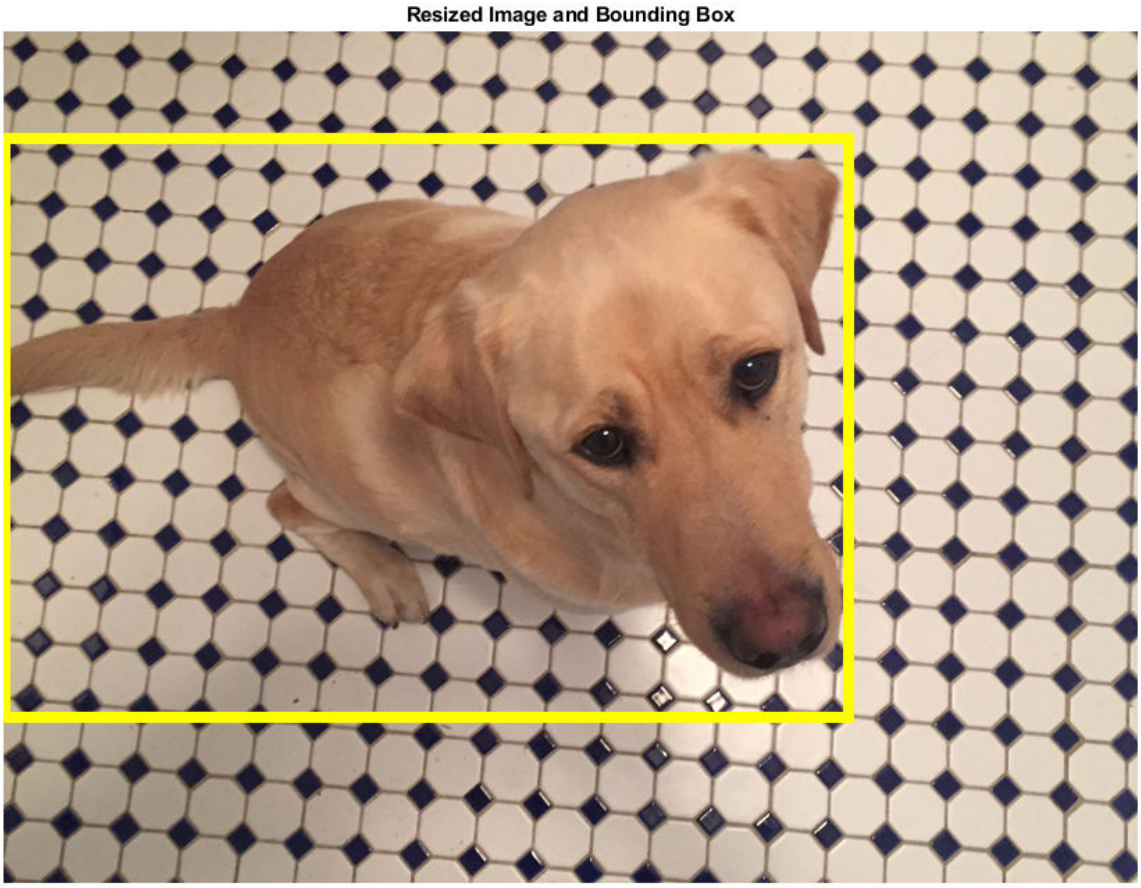
```
scale = 1/2;
J = imresize(I,scale);
```

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 `randomCropWindow2d` and `centerCropWindow2d` 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 = [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.

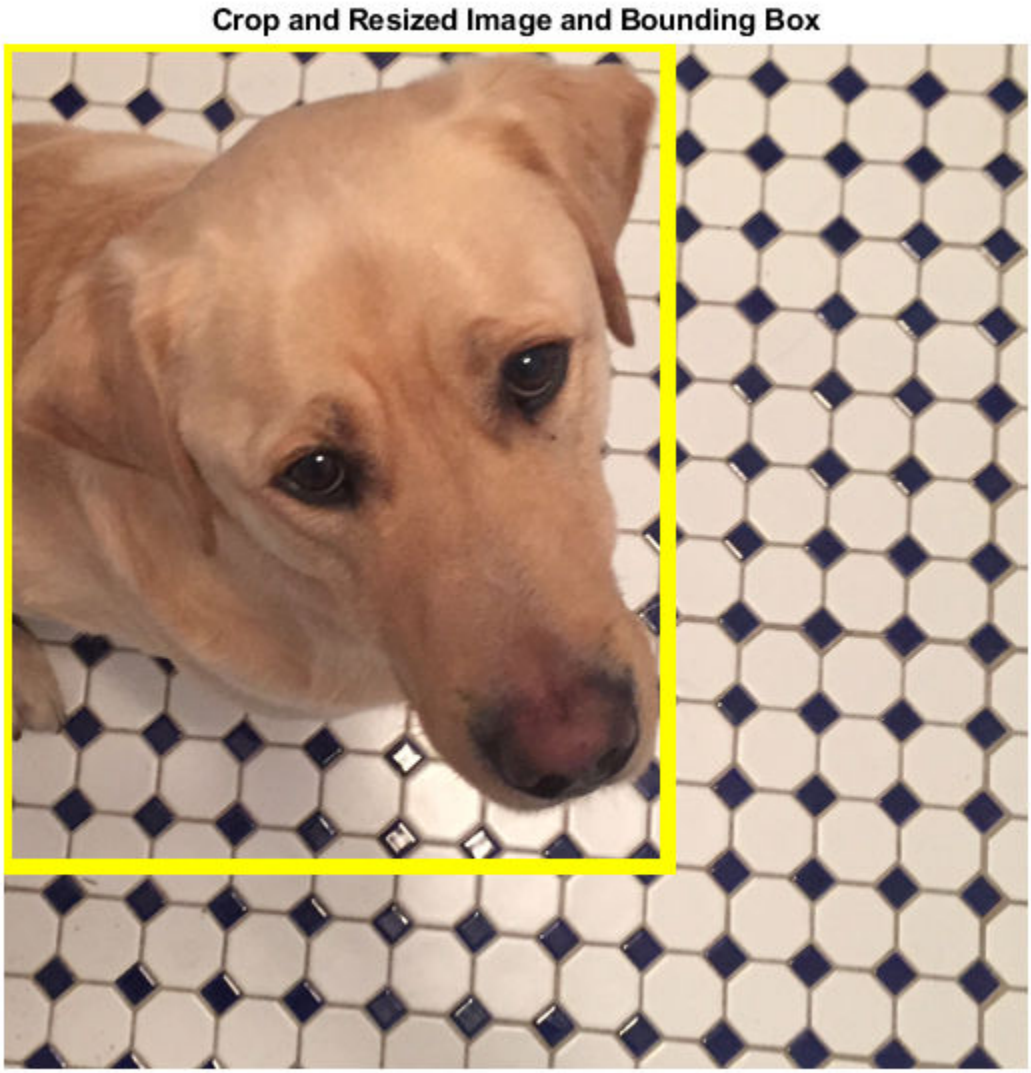
```
cropSize = [1024 1024];
win = randomCropWindow2d(size(I),cropSize);
J = imcrop(I,win);
croppedBox = bboxcrop(bbox,win,"OverlapThreshold",0.5);
```

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` 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`. Warp bounding boxes by using `bboxwarp`. Control the spatial bounds and resolution of the warped output by using the `affineOutputView` 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.

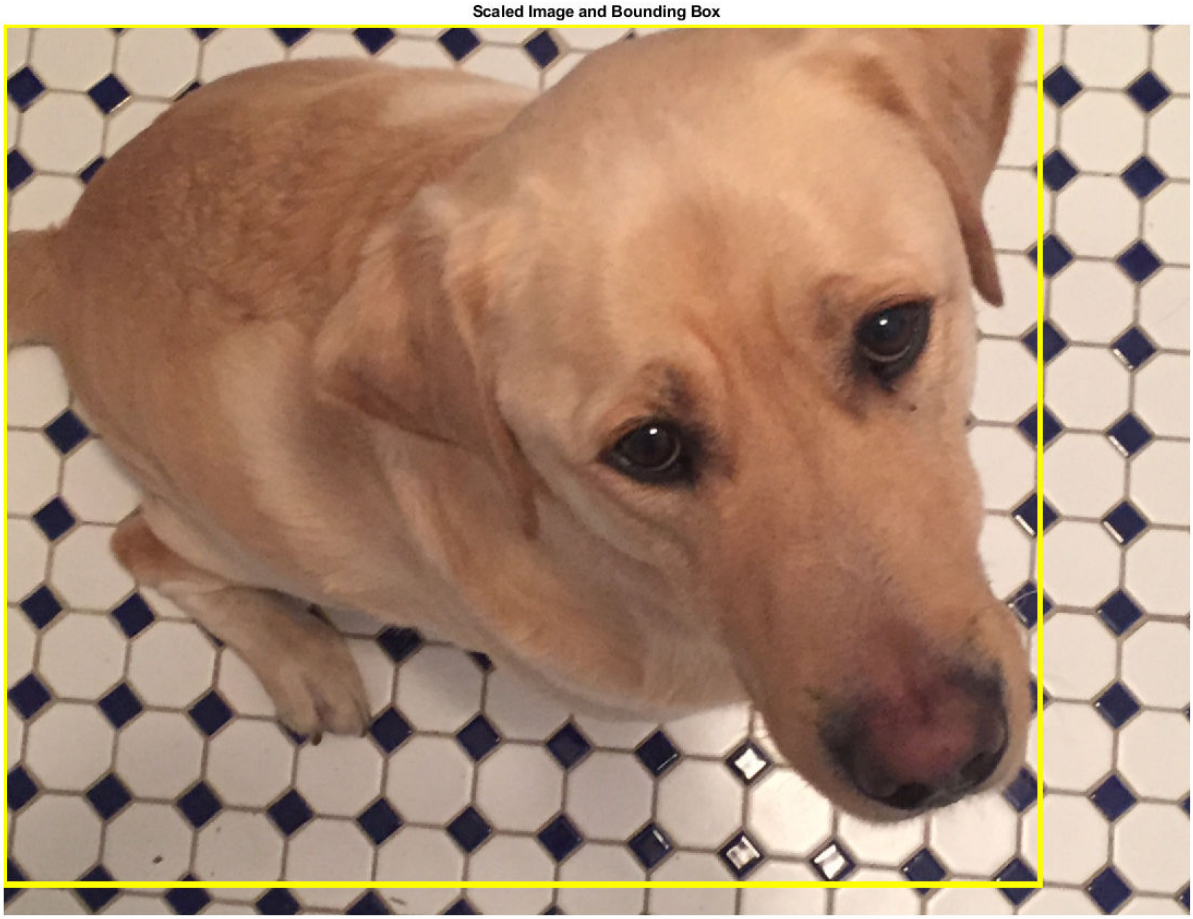
```
rout = affineOutputView(size(I),tform);
```

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);
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`.

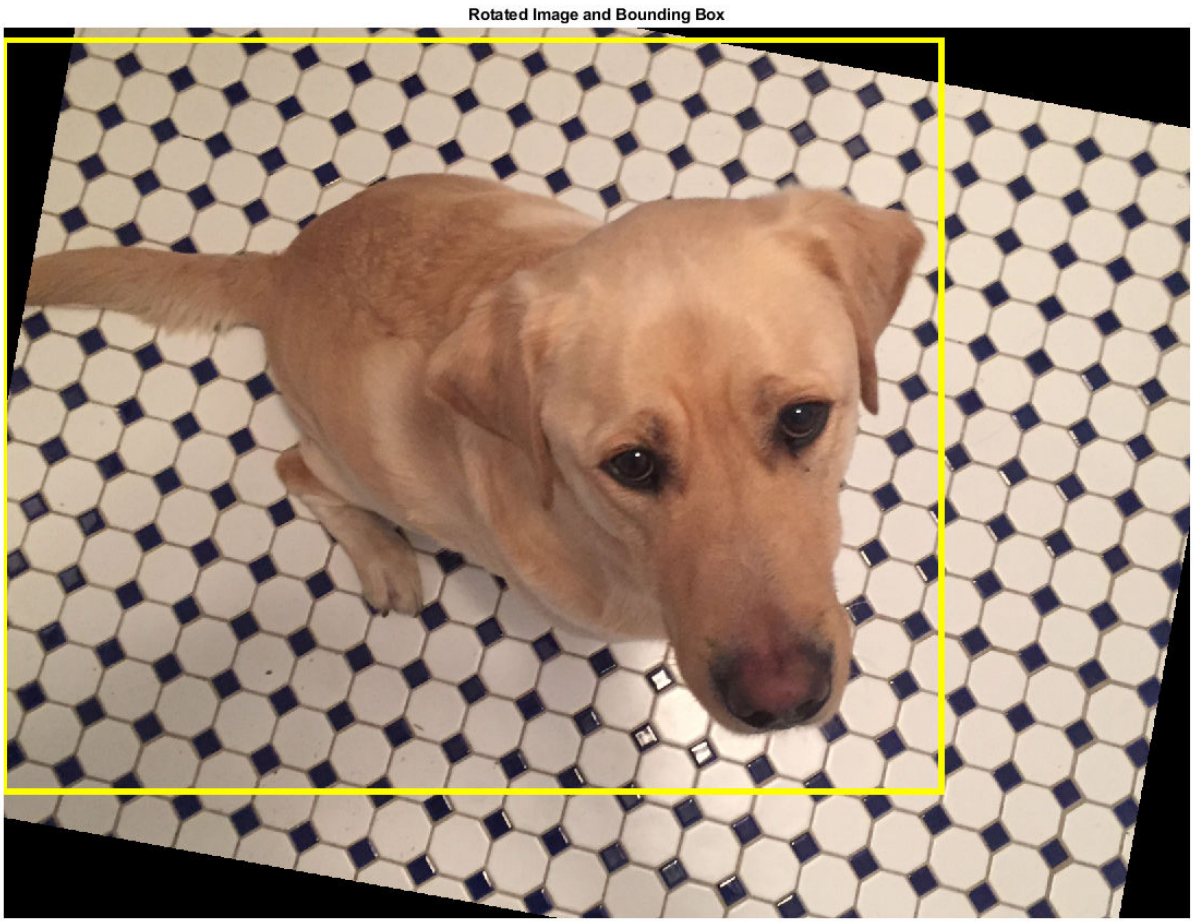
```
rout = affineOutputView(size(I),tform);
```

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.

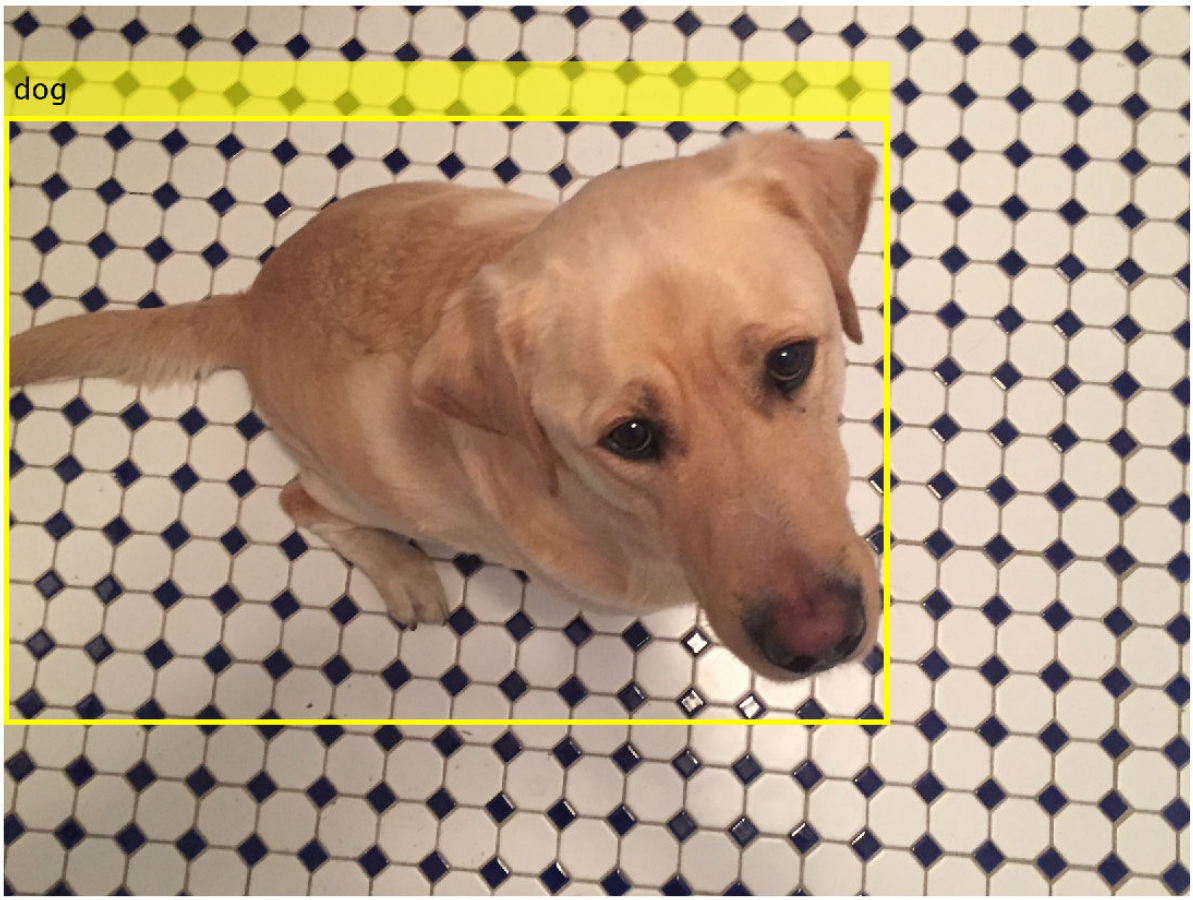
```
trainingData = combine(imds,blds);
```

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));
```

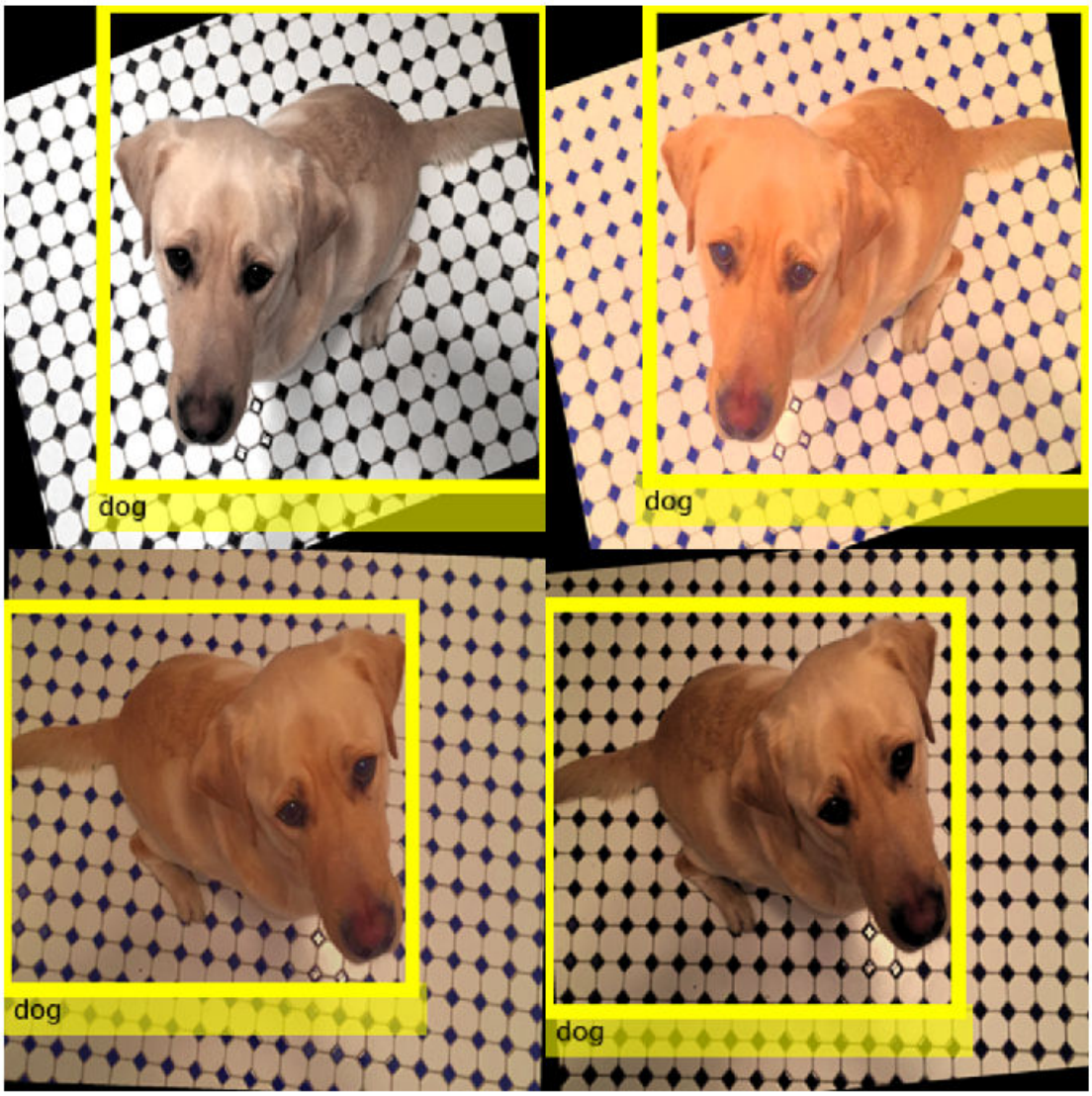
Read all of the preprocessed data.

```
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

`bboxcrop` | `bboxresize` | `bboxwarp` | `centerCropWindow2d` | `imcrop` | `imresize` | `randomCropWindow2d`

Related Examples

- “Augment Images for Deep Learning Workflows Using Image Processing Toolbox” on page 16-34
- “Train Object Detector Using R-CNN Deep Learning” on page 8-131

More About

- “Preprocess Data for Domain-Specific Deep Learning Applications” on page 16-19
- “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` 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. Combined datastores (created when you use the `combine` function) do not support shuffling, so specify the `Shuffle` parameter as `'never'`.

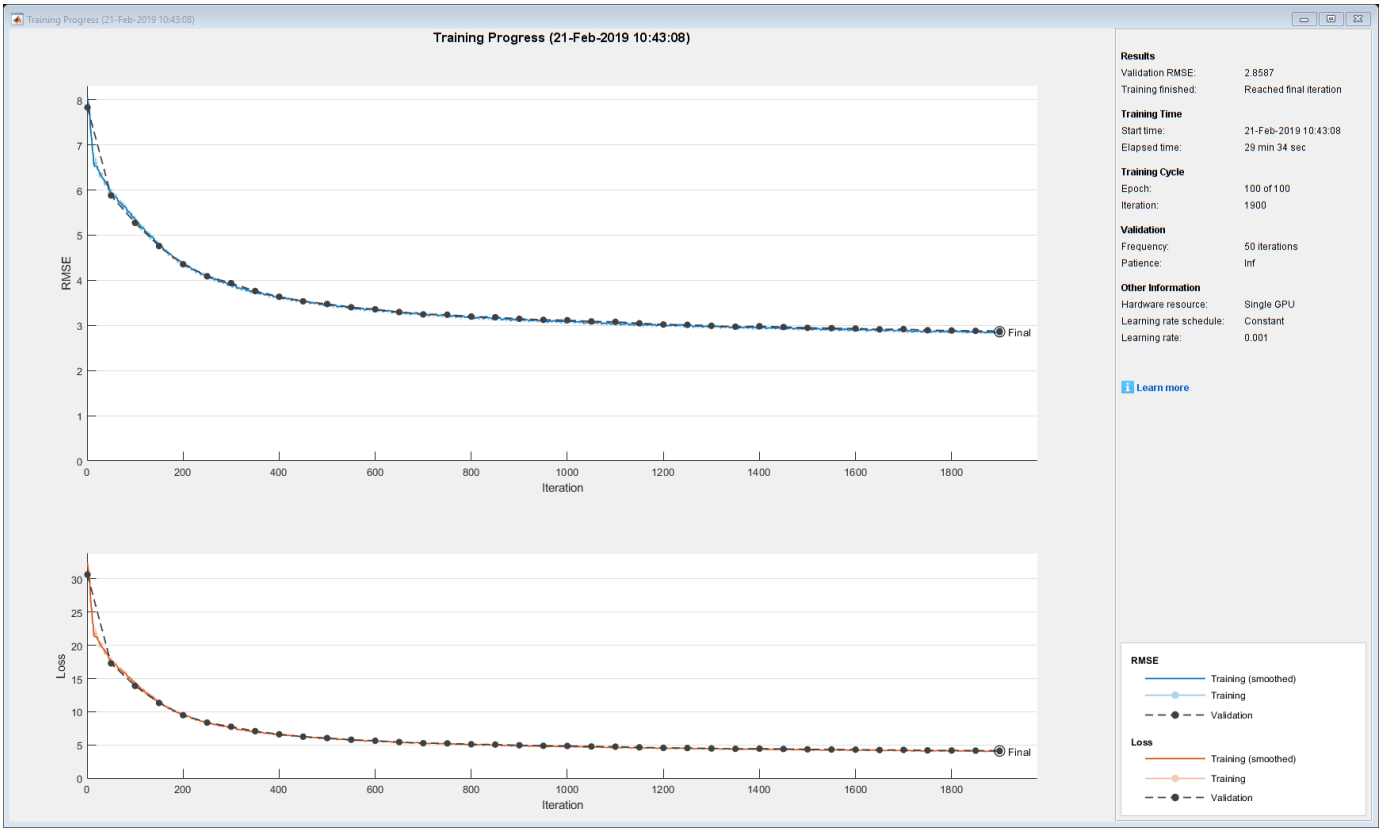
```
options = trainingOptions('adam', ...
    'MaxEpochs',100, ...
    'MiniBatchSize',imds.ReadSize, ...
    'ValidationData',dsVal, ...
    'Shuffle','never', ...
    'Plots','training-progress', ...
    'Verbose',false);
```

Train the Network

Now that the data source and training options are configured, train the convolutional autoencoder network using the `trainNetwork` function. A CUDA-capable NVIDIA™ GPU with compute capability 3.0 or higher is highly recommended for training.

Note: Training takes approximately 25 minutes on an NVIDIA™ Titan XP GPU.

```
net = trainNetwork(dsTrain,layers,options);
```



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
    18.6498
```

```
psnrDenoised = psnr(ypred(:,:,:,1),ref)
```

```
psnrDenoised = single
    21.8640
```

The PSNR of the output image is higher than the noisy input image, as expected.

Summary

This example showed how to use the `transform` and `combine` functions of `ImageDatastore` to set up the data preprocessing required for training and evaluating a convolutional autoencoder on the digit data set.

Supporting Functions

The `addNoise` helper function adds salt and pepper noise to images by using the `imnoise` 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

`combine` | `imageDatastore` | `trainNetwork` | `trainingOptions` | `transform`

See Also

Related Examples

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

More About

- “Datastores for Deep Learning” on page 16-2

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];
```

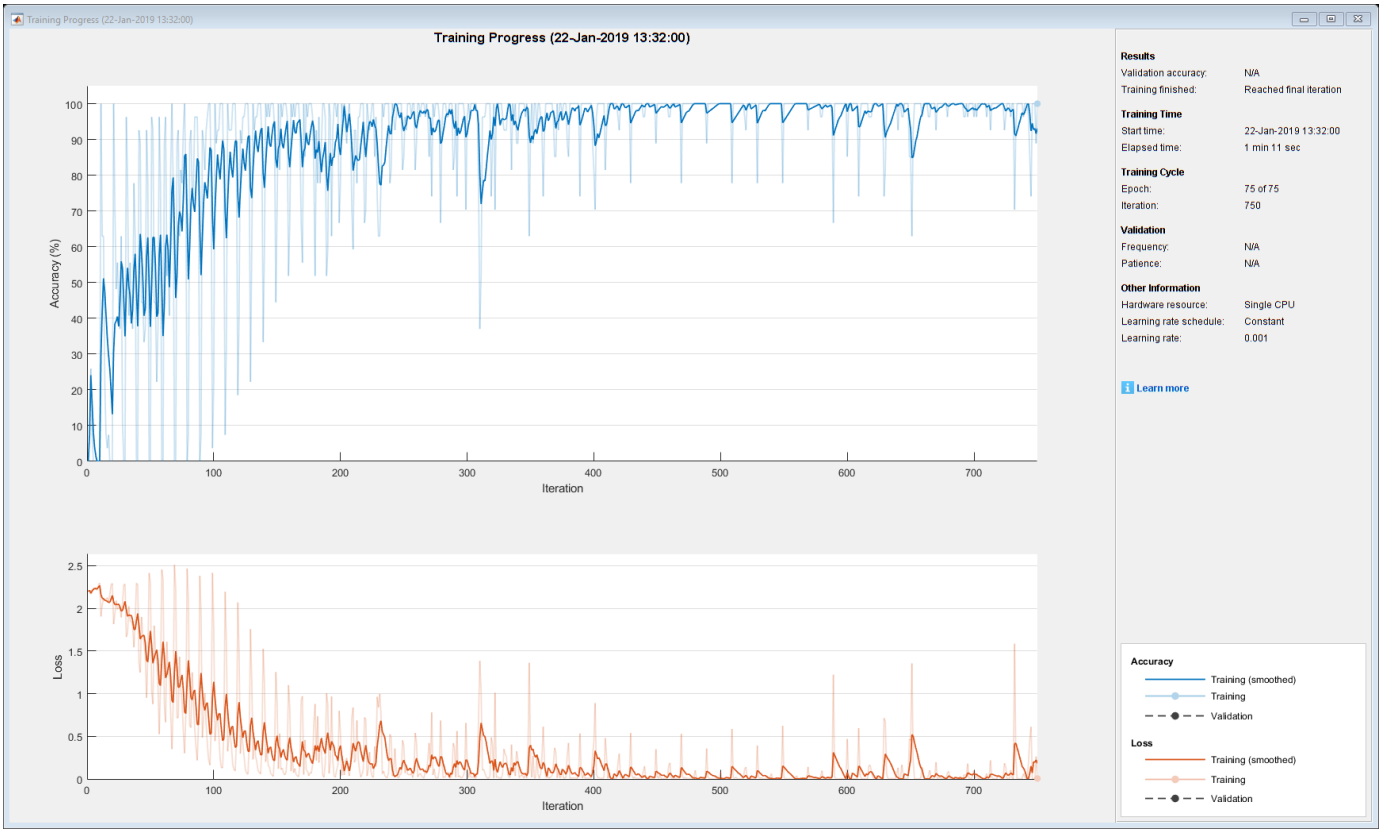
Specify the training options. Set the solver to `'adam'` and `'GradientThreshold'` to 2. Set the mini-batch size to 27 and set the maximum number of epochs to 75. The datastores do not support shuffling, so set `'Shuffle'` to `'never'`.

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

`combine` | `lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions` | `transform`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Long Short-Term Memory Networks” on page 1-53
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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 16-28.

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

`lstmLayer` | `sequenceInputLayer` | `trainNetwork` | `trainingOptions`

Related Examples

- “Develop Custom Mini-Batch Datastore” on page 16-28
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Sequence-to-Sequence Classification Using Deep Learning” on page 4-34
- “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39
- “Long Short-Term Memory Networks” on page 1-53
- “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];
```

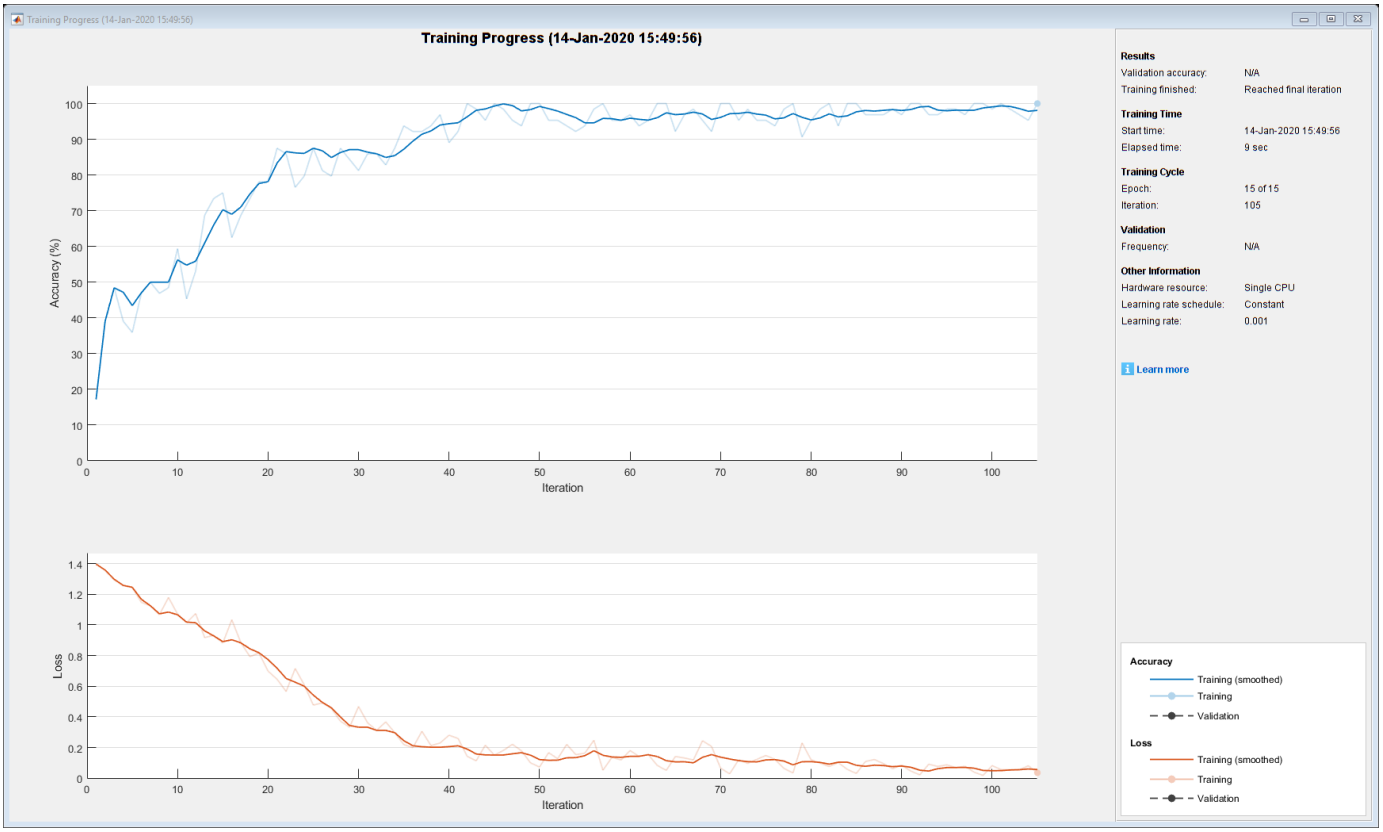
Specify the training options. Specify the solver to be `'adam'` and the gradient threshold to be 2. The datastore does not support shuffling, so set `'Shuffle'`, to `'never'`. Validate the network once per epoch. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). 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.

```
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

`doc2sequence` | `fastTextWordEmbedding` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `transform` | `wordEmbeddingLayer`

Related Examples

- “Sequence Classification Using Deep Learning” on page 4-2
- “Time Series Forecasting Using Deep Learning” on page 4-9
- “Long Short-Term Memory Networks” on page 1-53
- “List of Deep Learning Layers” on page 1-23
- “Deep Learning Tips and Tricks” on page 1-45

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.m`. 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 16-28.

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.

To create the datastore, first save the custom mini-batch datastore `textDatastore.m` to the path. For more information about creating custom mini-batch datastores, see “Develop Custom Mini-Batch Datastore” on page 16-28.

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 16-28. To monitor the training progress, set the `'Plots'` option to `'training-progress'`. To suppress verbose output, set `'Verbose'` to `false`.

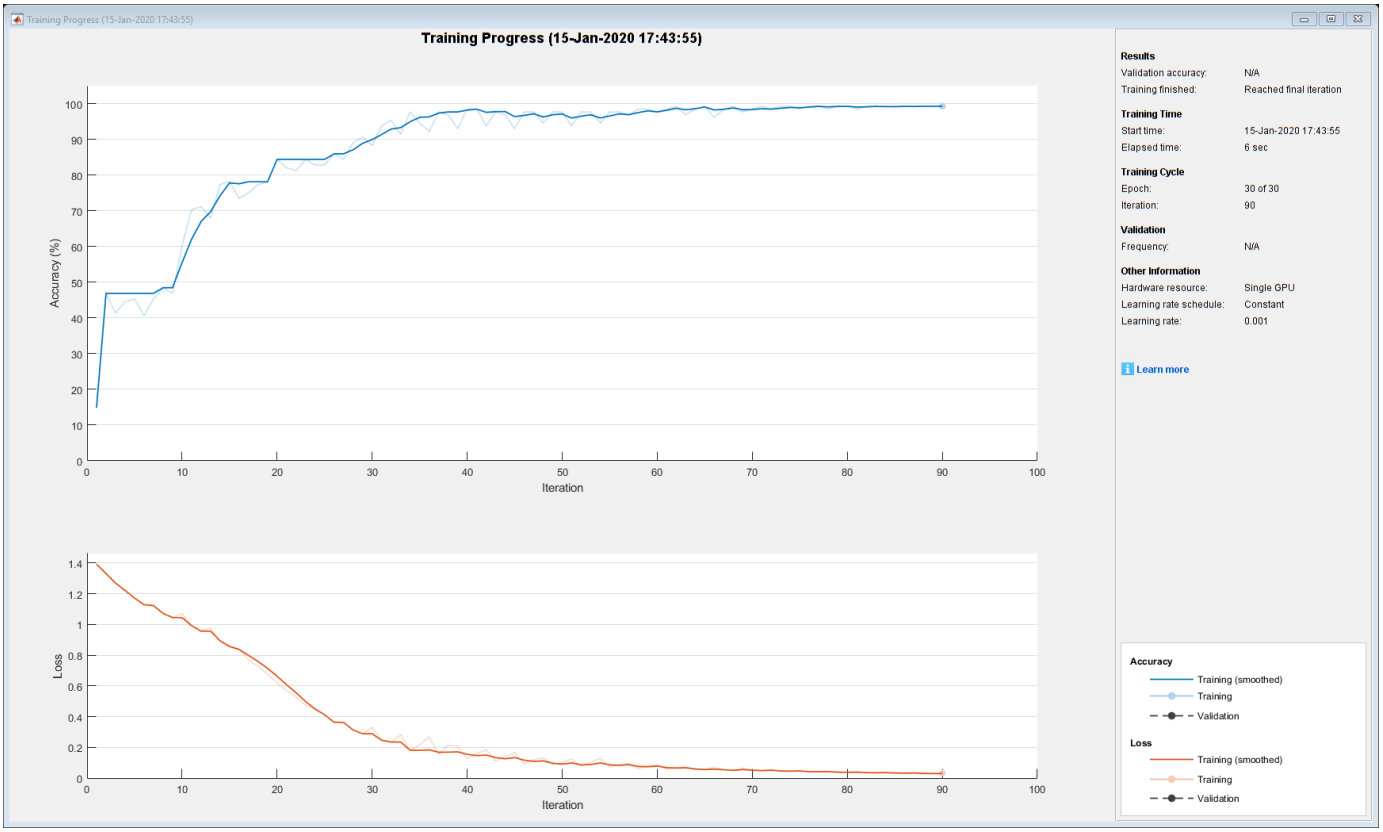
By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses the CPU. To specify the execution environment manually, use the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Training on a CPU can take significantly longer than training on a GPU.

```
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.m`.

```
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

`doc2sequence` | `extractHTMLText` | `findElement` | `htmlTree` | `lstmLayer` | `sequenceInputLayer` | `tokenizedDocument` | `trainNetwork` | `trainingOptions` | `wordEmbeddingLayer` | `wordcloud`

Related Examples

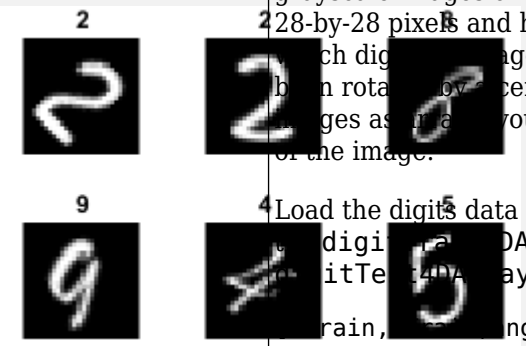
- “Generate Text Using Deep Learning” on page 4-131
- “Create Simple Text Model for Classification” (Text Analytics Toolbox)
- “Analyze Text Data Using Topic Models” (Text Analytics Toolbox)
- “Analyze Text Data Using Multiword Phrases” (Text Analytics Toolbox)
- “Train a Sentiment Classifier” (Text 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.

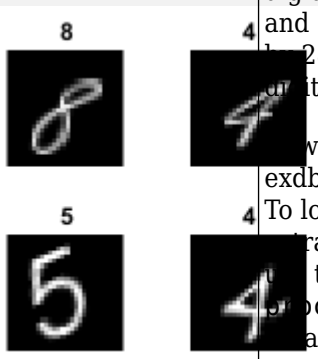
Image Data Sets

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 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 5-49 and “Train Convolutional Neural Network for Regression” on page 3-46.
		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 3-40.
		Image classification

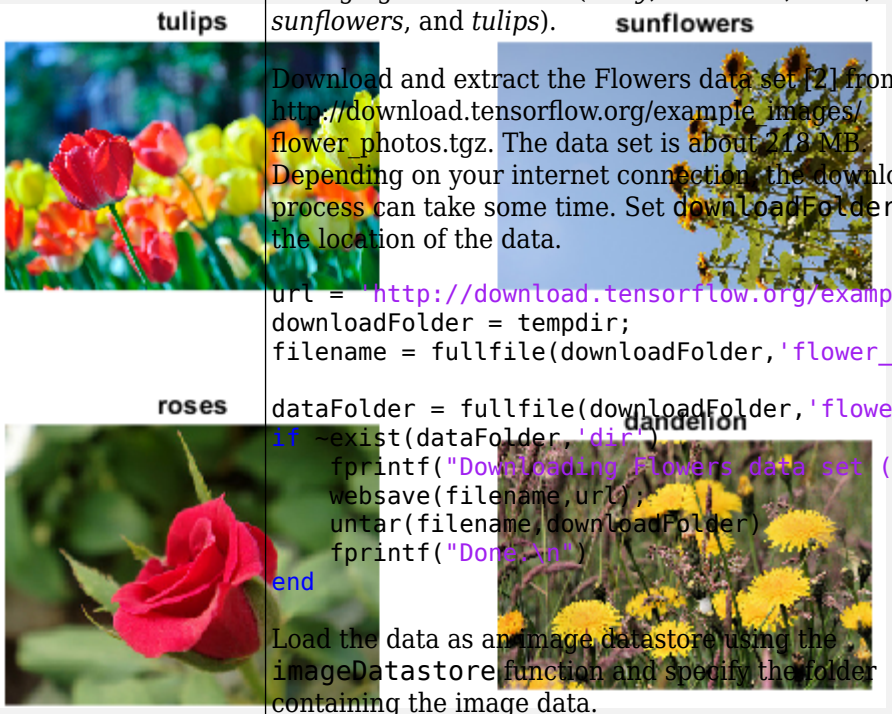
Data Set	Description	Task

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, extract and locate the files in the working directory, then load the helper functions `processImagesMNIST` and `processLabelsMNIST`, which are used in the example "Train Variational Autoencoder (VAE) to Generate Images" on page 3-124.
	
	`oldpath = addpath(fullfile(matlabroot,'examples','nnet','main'));`
	`filenameImagesTrain = 'train-images.idx3-ubyte';`
	`filenameLabelsTrain = 'train-labels.idx1-ubyte';`
	`filenameImagesTest = 't10k-images.idx3-ubyte';`
	`filenameLabelsTest = 't10k-labels.idx1-ubyte';`
	
	`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 3-124.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Image classification

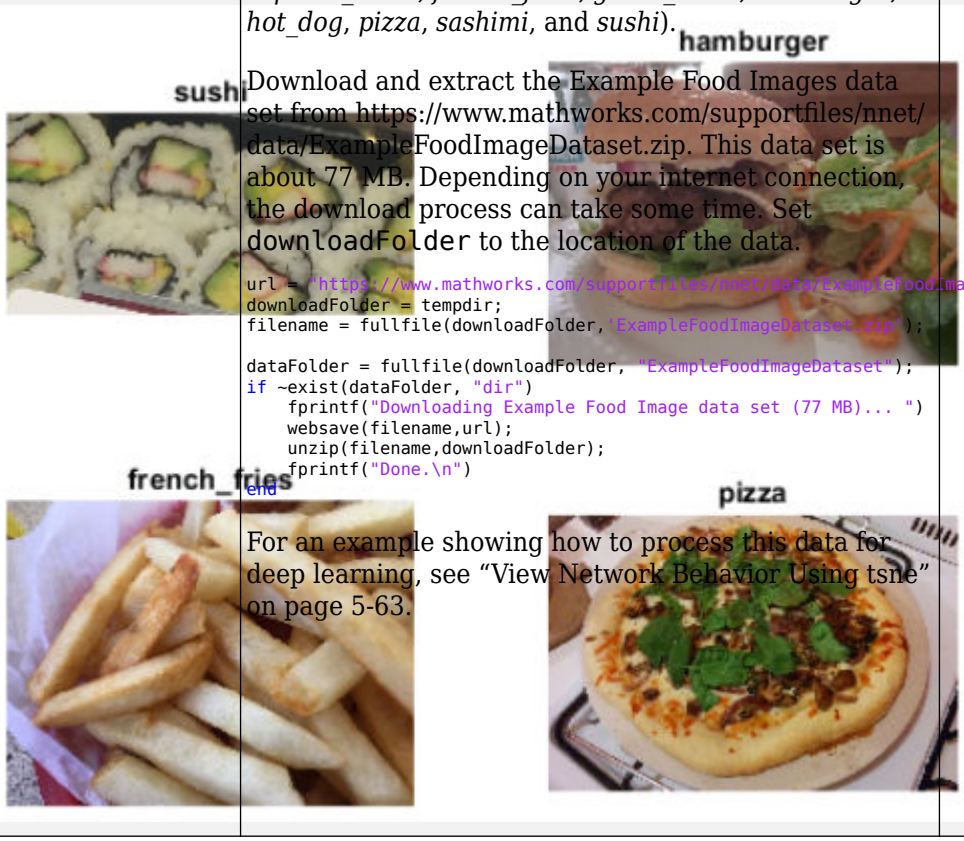
Data Set	Description	Task

Omniglot	The Omniglot data set contains character sets for 50 alphabets, divided into 30 sets for training and 20 sets for testing. 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 [1] 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 3-96.	

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*).
	
	Download and extract the Flowers data set [2] from http://download.tensorflow.org/example_images/flower_photos.tgz. The data set is about 218 MB. Depending on your internet connection, the download process can take some time. 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 3-72.
		Image classification

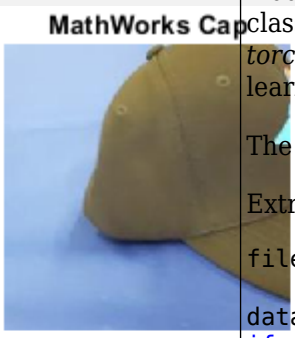
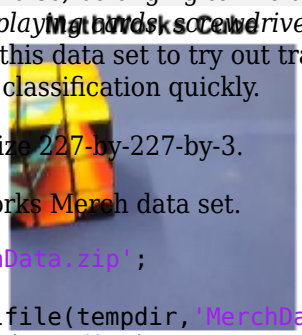
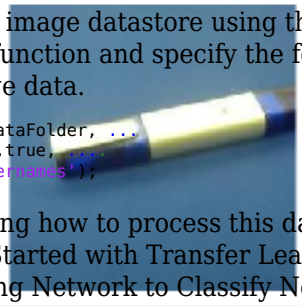
Data Set	Description	Task

Example Food Images	The Example Food Images data set contains 978 photographs of food in nine classes (*ceaser_salad*, *caprese_salad*, *french_fries*, *greek_salad*, *hamburger*, *hot_dog*, *pizza*, *sashimi*, and *sushi*).
	 
	Download and extract the Example Food Images data set from https://www.mathworks.com/supportfiles/nnet/data/ExampleFoodImageDataset.zip. This data set is about 77 MB. Depending on your internet connection, the download process can take some time. Set `downloadFolder` to the location of the data.
	`url = "https://www.mathworks.com/supportfiles/nnet/data/ExampleFoodImageDataset.zip";`
	`downloadFolder = tempdir;`
	`filename = fullfile(downloadFolder,'ExampleFoodImageDataset.zip');`
	`dataFolder = fullfile(downloadFolder, "ExampleFoodImageDataset");`
	`if ~exist(dataFolder, "dir")`
	`  fprintf("Downloading Example Food Image data set (77 MB)... ")`
	`  websave(filename,url);`
	`  unzip(filename,downloadFolder);`
	`  fprintf("Done.\n")`
	`end`
	For an example showing how to process this data for deep learning, see “View Network Behavior Using tsne” on page 5-63.
		Image classification

Data Set	Description	Task

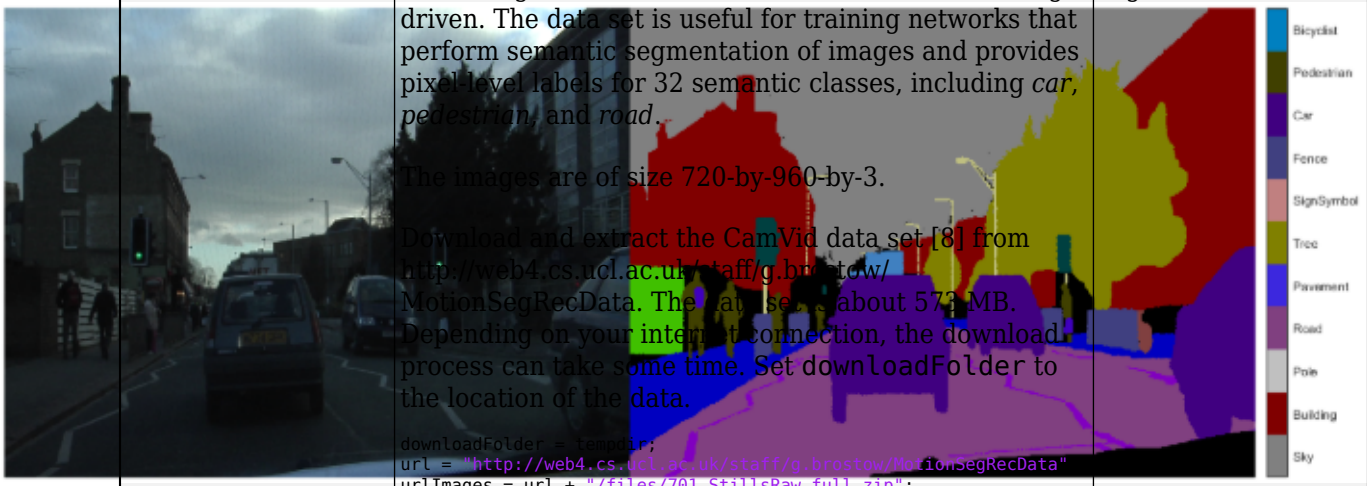
CIFAR-10
 
 	
(Representative example)
	The CIFAR-10 data set contains 60,000 color images of size 32-by-32 pixels, belonging to 10 classes (*airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*).
	
	There are 6000 images per class and the data set is split into 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 [7] from https://www.cs.toronto.edu/%7Ekriz/cifar-10-matlab.tar.gz. The data set is about 175 MB. Depending on your internet connection, the download process can take some time. 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

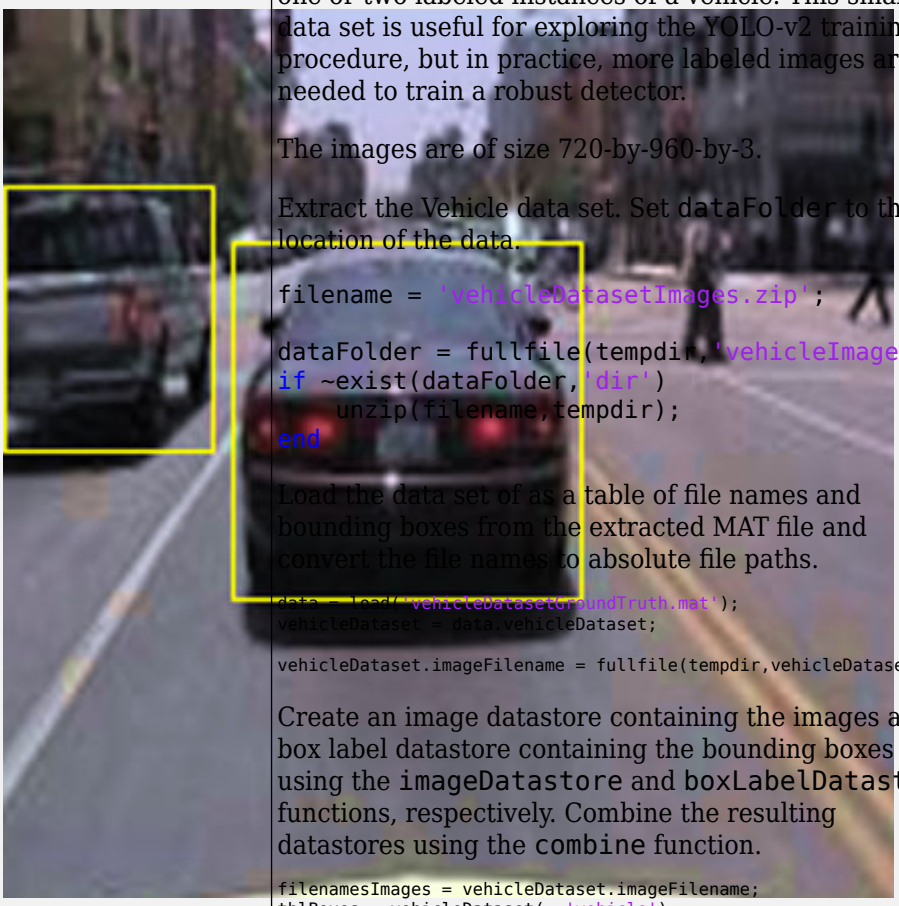
MathWorks® Merch    	This is a small data set containing 75 images of MathWorks merchandise, belonging to five different classes (*cap*, *cube*, *playing cards*, *screwdriver*, and *torch*). You can use this data set to try out transfer learning and image classification quickly.
	
	The images are of 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. 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 [8] from http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData. The data set is about 573 MB. Depending on your internet connection, the download process can take some time. 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 the 32 original classes in the data set into 11 classes. To get the label IDs, use the helper function `camvidPixelLabelIDs`, which is used in the example “Semantic Segmentation Using Deep Learning” on page 8-74.
	
	`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 8-74.
		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 of 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 8-64.
		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. The data contains labeled training, validation, and test sets, with 18 object class labels including *road markings*, *tree*, and *building*.
	
	Download the RIT-18 data set [9] from http://www.cis.rit.edu/%7Ermk6217/rit18_data.mat. The data set is about 3 GB. Depending on your internet connection, the download process can take some time. 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 8-95.
		Semantic segmentation

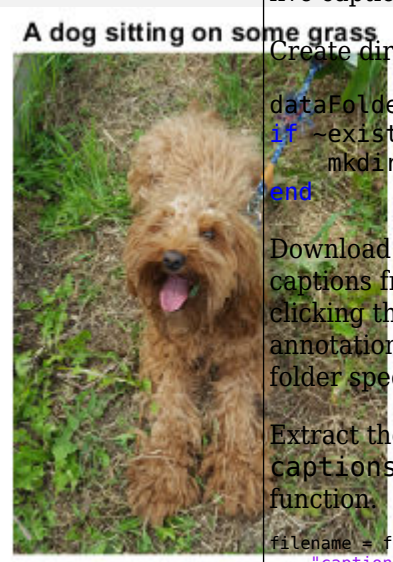
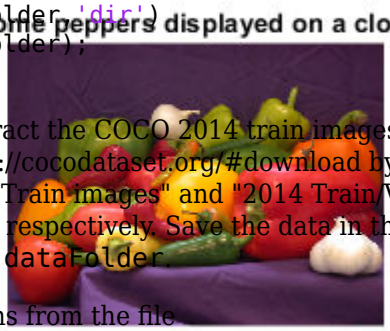
Data Set	Description	Task

BraTS	The BraTS data set contains MRI scans of brain tumors, namely gliomas, which are the most common primary brain malignancies.
	
	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.
	
	Create a directory to store the BraTS data set [10].
	
	`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. The data set is about 7 GB. Depending on your internet connection, the download process can take some time.
	
	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 8-112.
		Semantic segmentation

Data Set	Description	Task

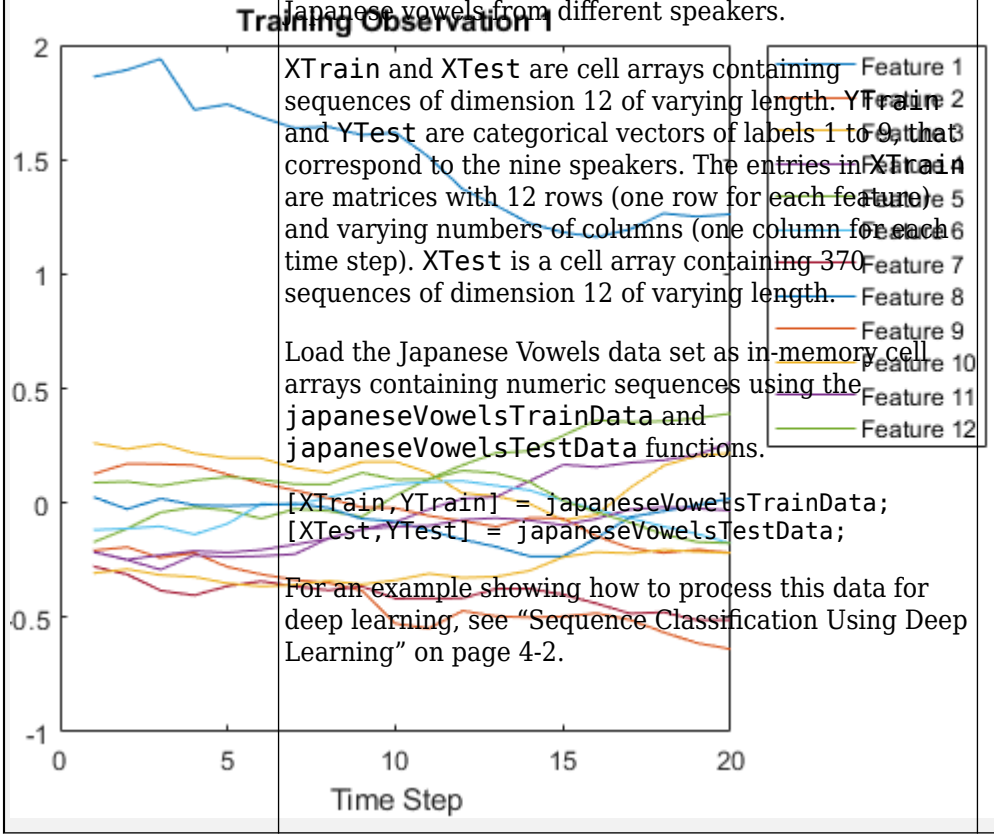
Camelyon16	The data from the Camelyon16 challenge contains a total of 400 WSIs of lymph nodes from two independent sources, separated into 270 training images and 130 test images. The WSIs are stored as TIF files in a striped format with an 11-level pyramidal structure.
	
	The training data set consists of 159 WSIs of normal lymph nodes and 111 whole-slide images (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 [11].
	
	`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 dataset" 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.
	
	The data set is about 2 GB. Depending on your internet connection, the download process can take some time.
	
	For an example showing how to process this data for deep learning, see “Deep Learning Classification of Large Multiresolution Images” on page 9-51.
		Image classification (large images)

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 http://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 struct 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 4-149.
		Image captioning

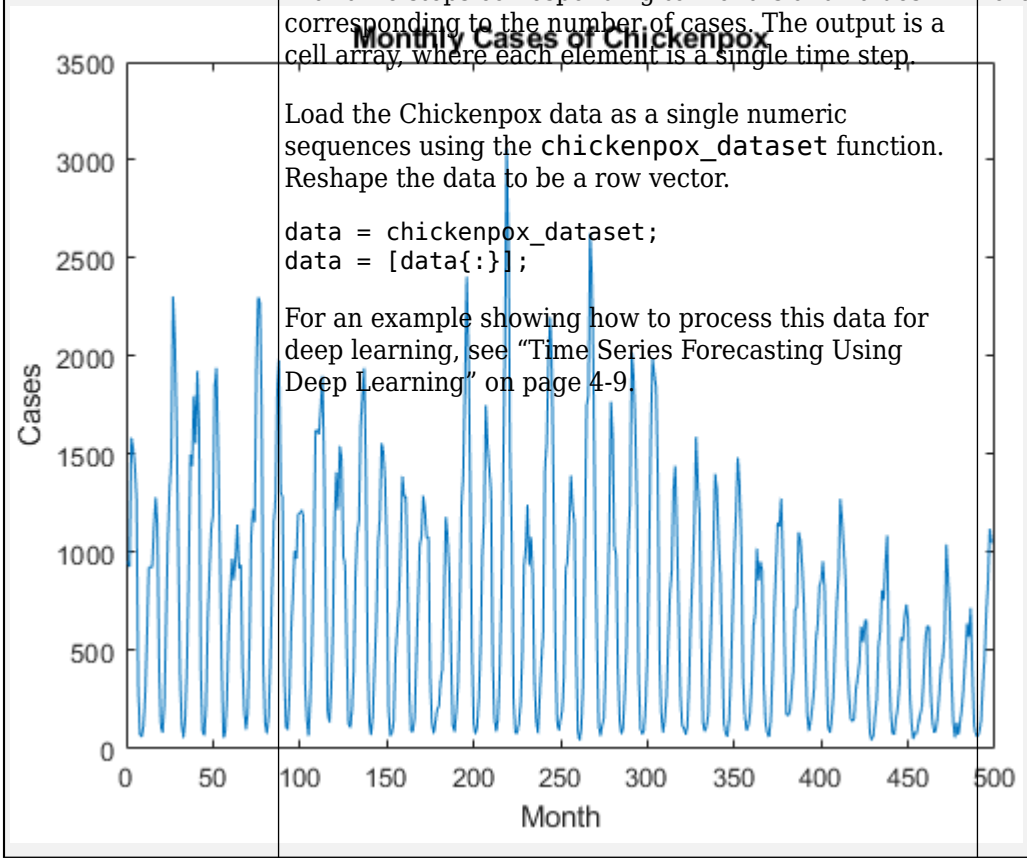
Time Series and Signal Data Sets

Data	Description	Task

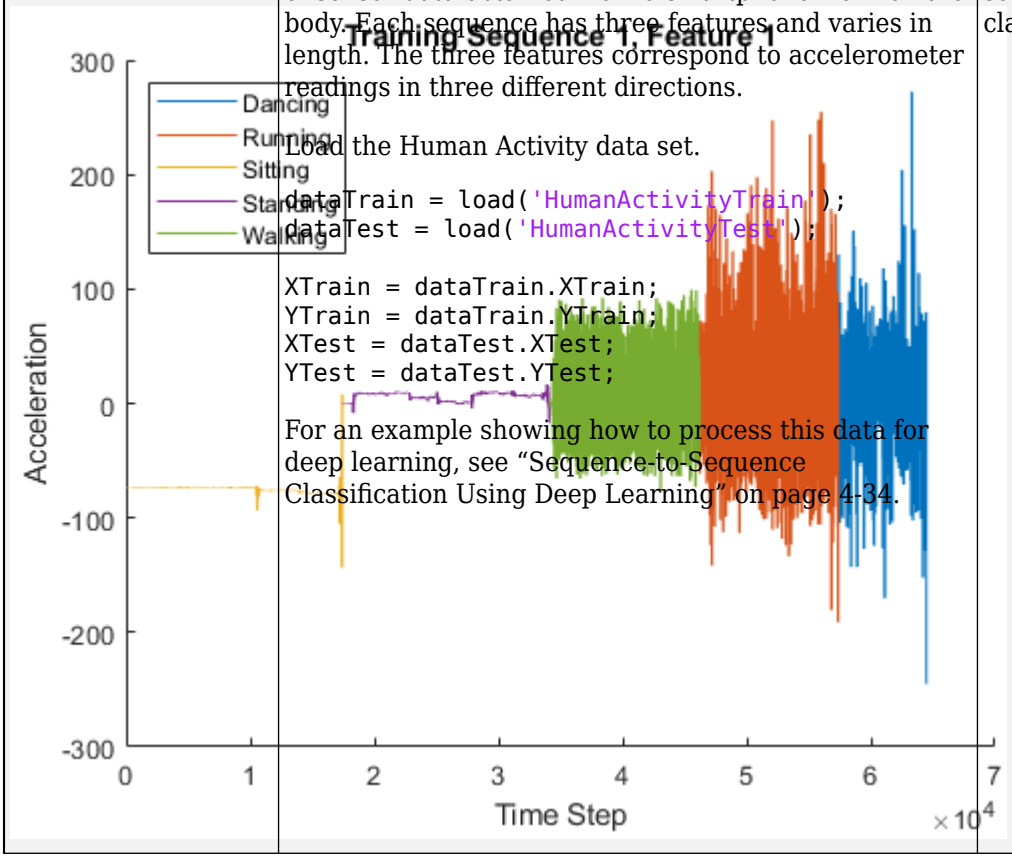
Japanese Vowels	The Japanese Vowels data set [12] [13] contains preprocessed sequences representing utterances of Japanese vowels from different speakers.
	 
	`XTrain` and `XTest` are cell arrays containing sequences of dimension 12 of varying length. `YTrain` and `YTest` are categorical vectors of labels 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 4-9.
		Time-series forecasting

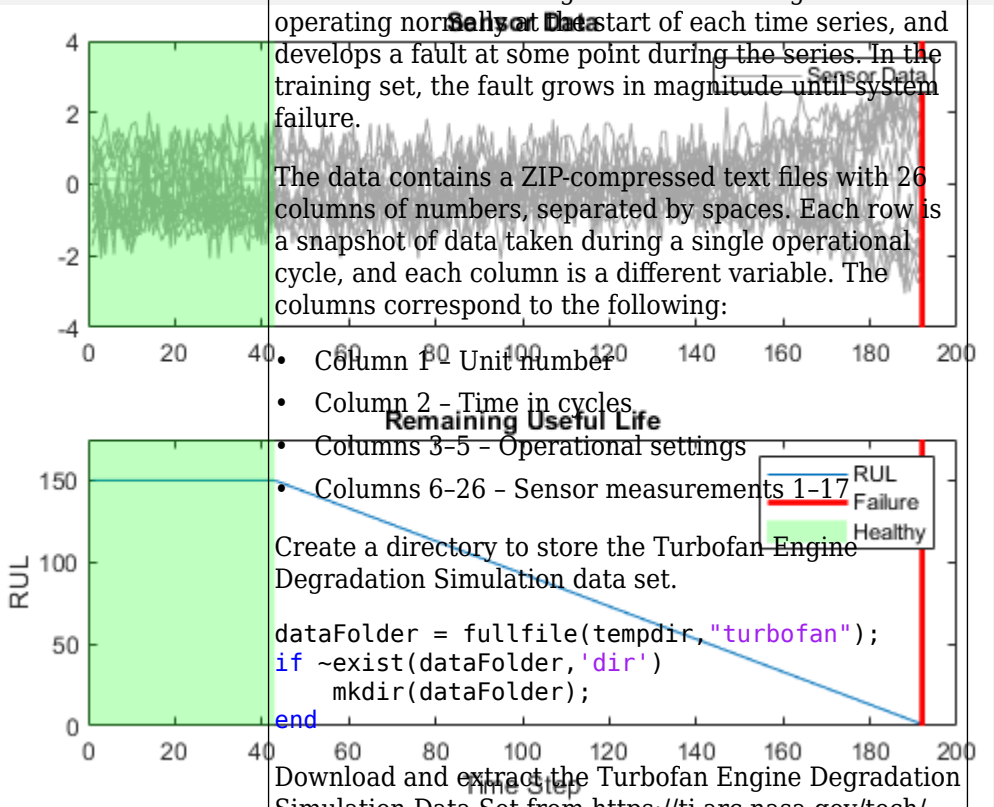


Data	Description	Task

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 4-34.
		Sequence-to-sequence classification

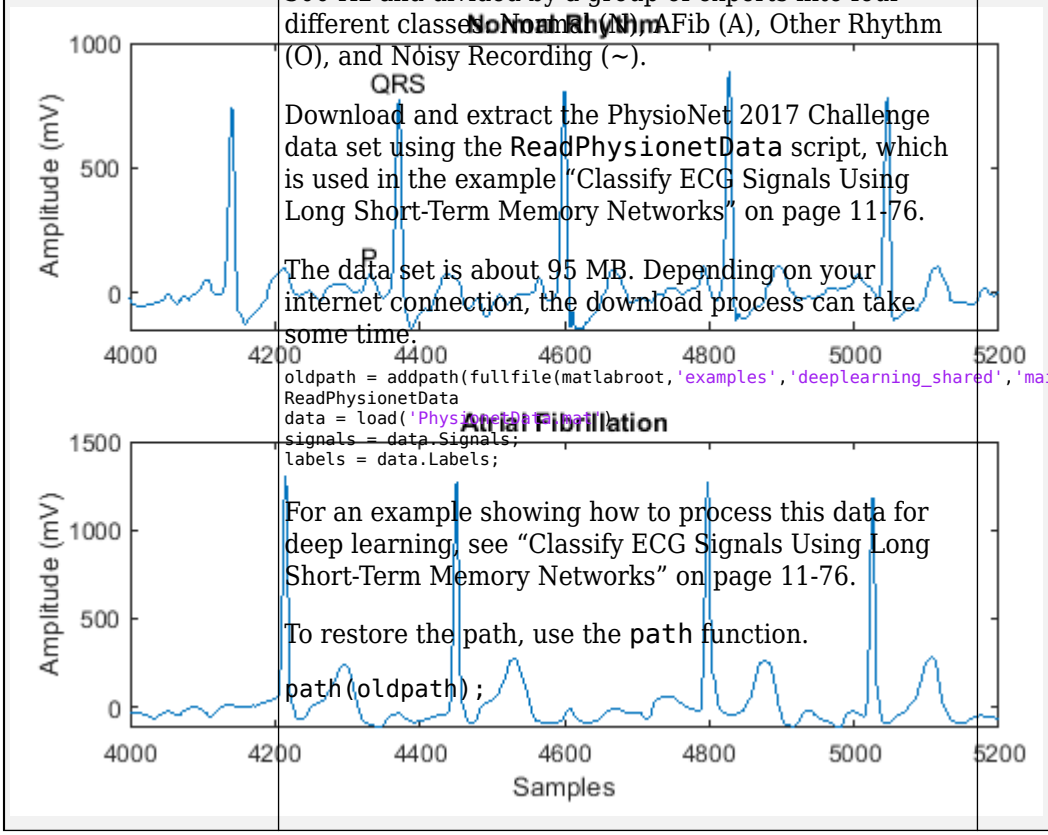
Data	Description	Task

Turbofan Engine Degradation Simulation	Each time series of the Turbofan Engine Degradation Simulation data set [14] 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 1–17
	
	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 4-39.
	
	`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

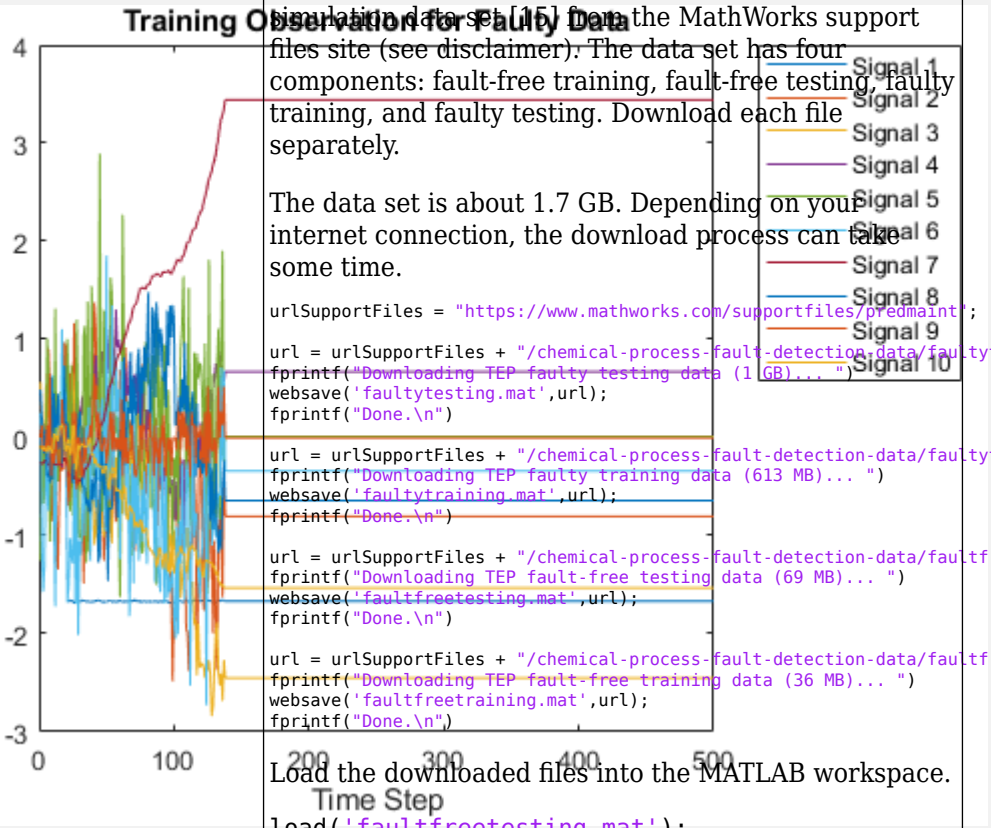


Data	Description	Task
	For an example showing how to process this data for deep learning, see “Sequence-to-Sequence Regression Using Deep Learning” on page 4-39. To restore the path, use the `path` function. `path(oldpath);`	

PhysioNet 2017 Challenge	The PhysioNet 2017 Challenge data set [16] consists of a set of electrocardiogram (ECG) recordings sampled at 300 Hz and divided by a group of experts into four different classes: Normal Rhythm (N), AFib (A), Other Rhythm (O), and Noisy Recording (~).
	
	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 11-76.
	
	The data set is about 95 MB. Depending on your internet connection, the download process can take some time.
	
	`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 11-76.
	
	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. Download the Tennessee Eastman Process (TEP) simulation data set [15] 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 about 1.7 GB. Depending on your internet connection, the download process can take some time. Load the downloaded files into the MATLAB workspace. For an example showing how to process this data for deep learning, see “Chemical Process Fault Detection Using Deep Learning” on page 14-2.	Sequence-to-label classification

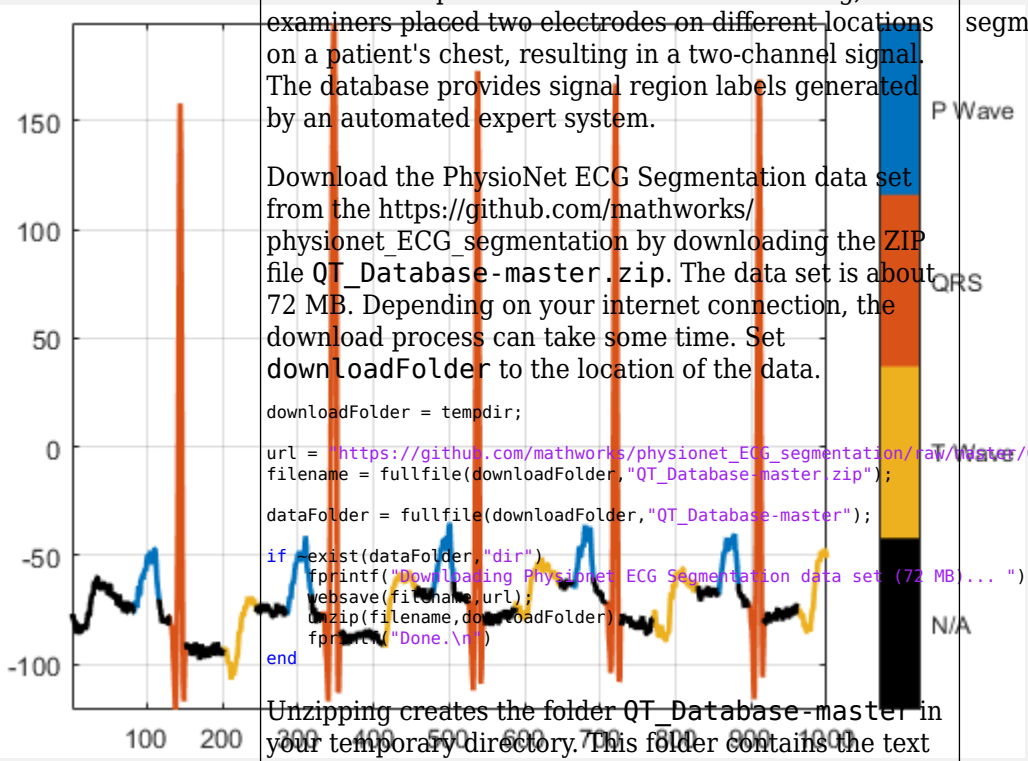



```
urlSupportFiles = "https://www.mathworks.com/supportfiles/predmaint";

url = urlSupportFiles + "/chemical-process-fault-detection-data/faultytesting.mat";
fprintf("Downloading TEP faulty testing data (1 GB)... ")
websave('faultytesting.mat',url);
fprintf("Done.\n")

url = urlSupportFiles + "/chemical-process-fault-detection-data/faultytraining.mat";
fprintf("Downloading TEP faulty training data (613 MB)... ")
websave('faultytraining.mat',url);
fprintf("Done.\n")

url = urlSupportFiles + "/chemical-process-fault-detection-data/faultfreetesting.mat";
fprintf("Downloading TEP fault-free testing data (69 MB)... ")
websave('faultfreetesting.mat',url);
fprintf("Done.\n")

url = urlSupportFiles + "/chemical-process-fault-detection-data/faultfreetraining.mat";
fprintf("Downloading TEP fault-free training data (36 MB)... ")
websave('faultfreetraining.mat',url);
fprintf("Done.\n")
```

```
load('faultfreetesting.mat');
load('faultfreetraining.mat');
load('faultytesting.mat');
load('faultytraining.mat');
```

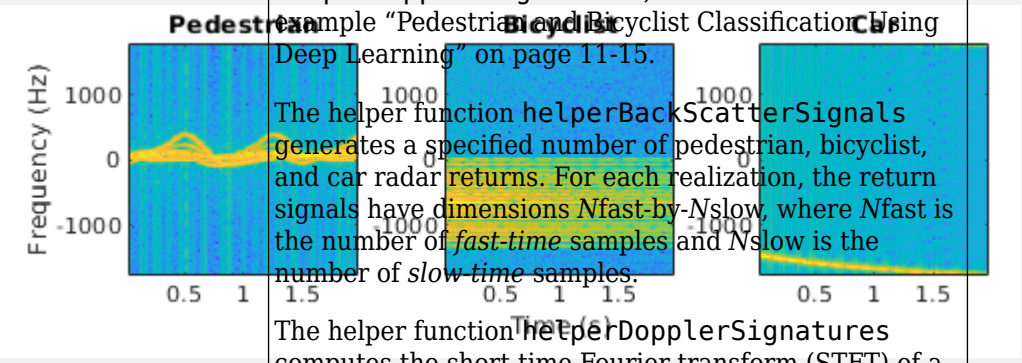
Data	Description	Task

PhysioNet ECG Segmentation	The PhysioNet ECG Segmentation data set [16] [17] consists of roughly 15 minutes of ECG recordings 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.
	
	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 about 72 MB. Depending on your internet connection, the download process can take some time. 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 11-42.
		Sequence-to-label classification, waveform segmentation

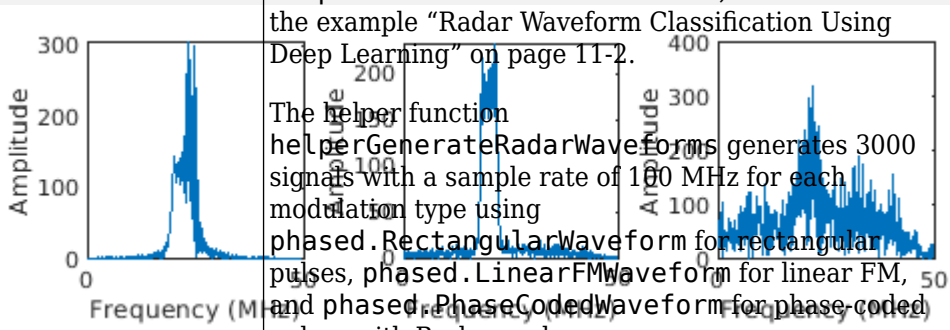


Data	Description	Task

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" on page 11-15.
	
	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" on page 11-15.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Sequence-to-label classification

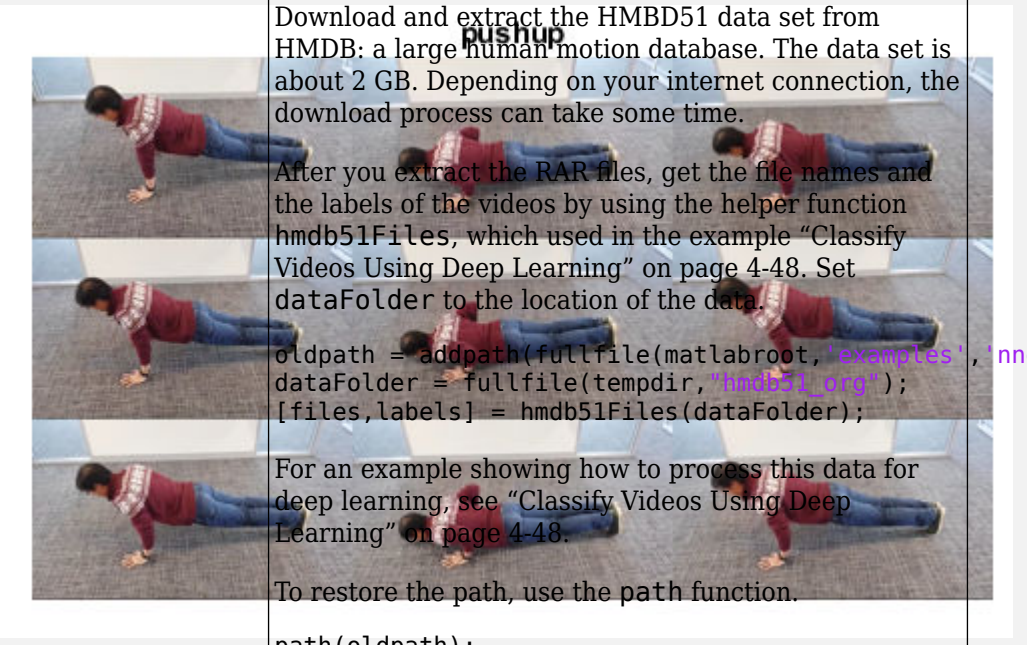


Data	Description	Task

Generated waveforms 	Generate rectangular, linear FM, and phase coded waveforms using the helper function `helperGenerateRadarWaveforms`, which is used in the example “Radar Waveform Classification Using Deep Learning” on page 11-2.
	
	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 Waveform Classification Using Deep Learning” on page 11-2.
	
	To restore the path, use the `path` function.
	
	`path(oldpath);`
		Sequence-to-label classification

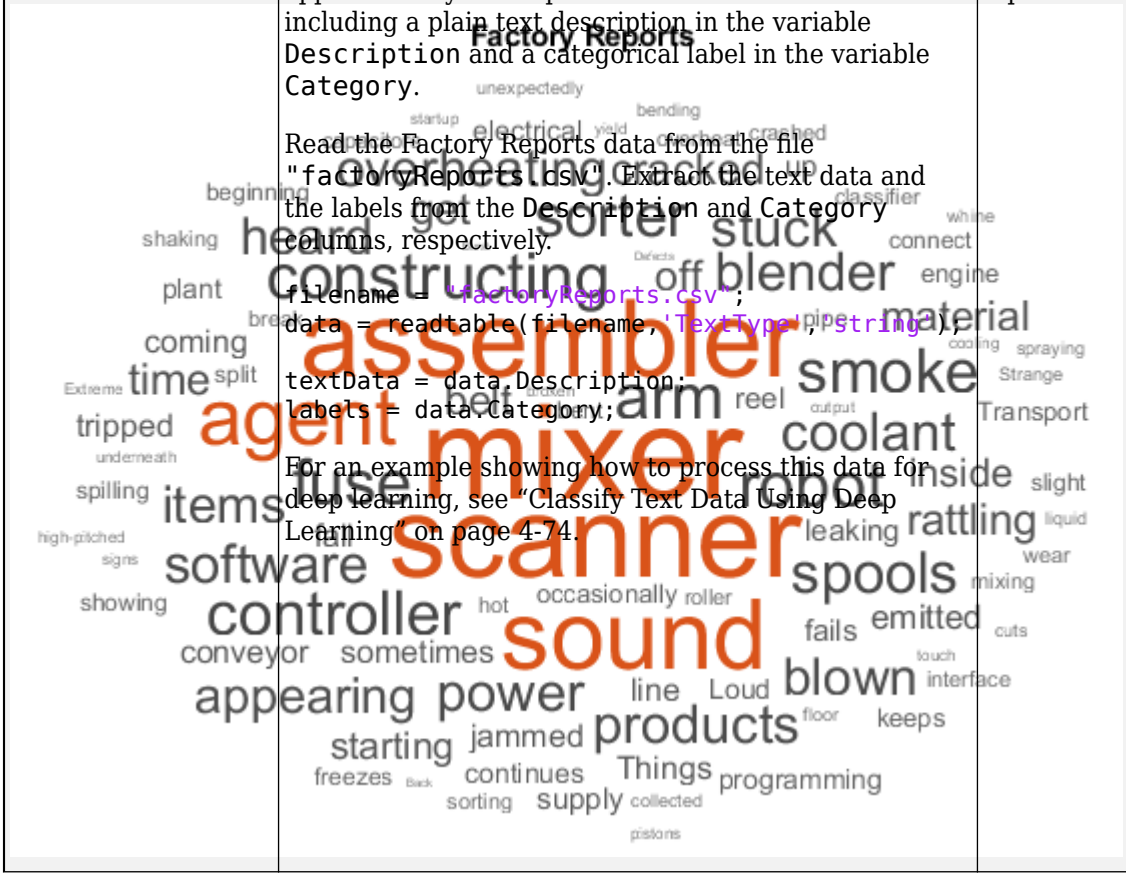
Video Data Sets

Data	Description	Task

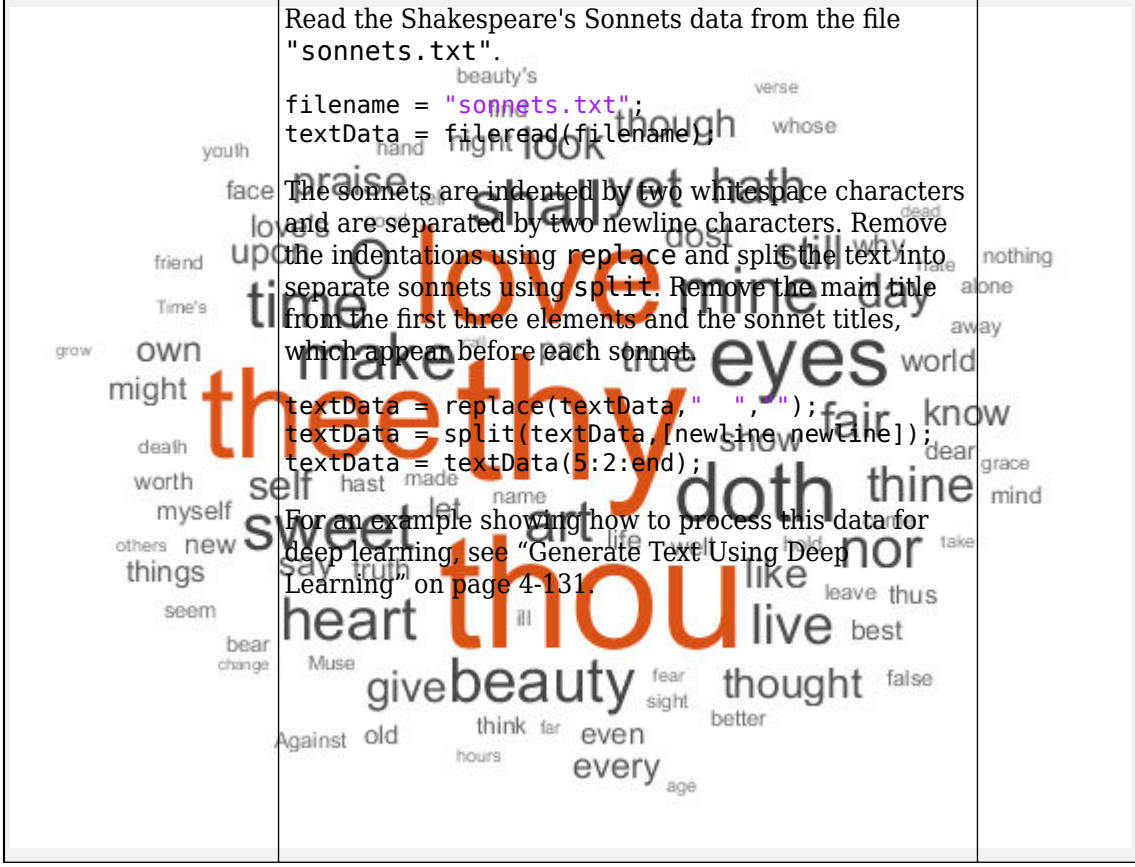
HMDB: a large human motion database
 
 	
(Representative example)
	The HMBD51 data set contains about 2 GB of video data for 7000 clips from 51 classes, such as *drink*, *run*, and *pushup*.
	
	Download and extract the HMBD51 data set from HMDB: a large human motion database. The data set is about 2 GB. Depending on your internet connection, the download process can take some time.
	
	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 4-48. 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 4-48.
	
	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 4-74.
		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.	Topic modeling, text generation



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 4-131.

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 4-91.
		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 4-143.
		Topic modeling, text generation

Data	Description	Task

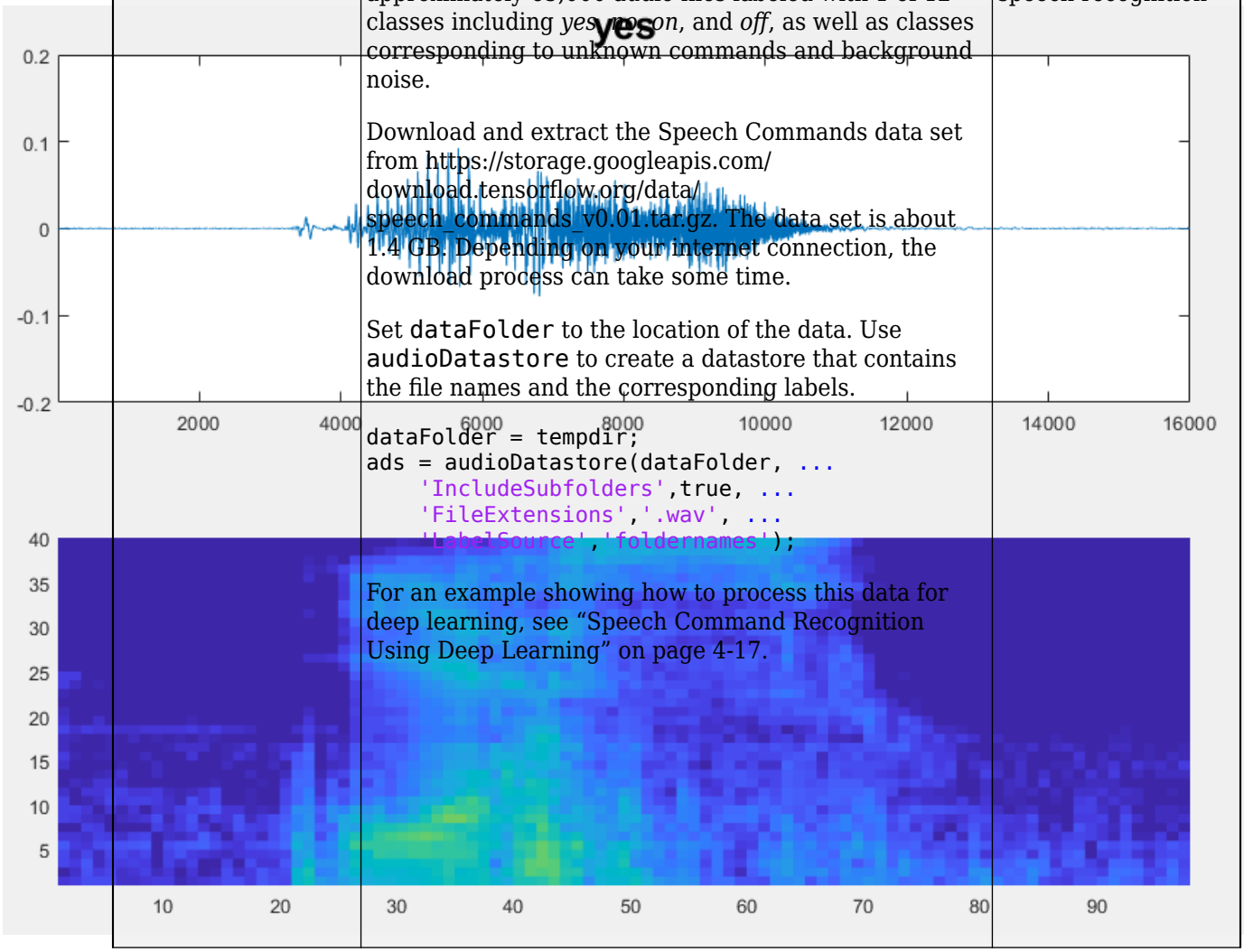
Weekend updates	The file `weekendUpdates.xlsx` contains example social media status updates containing the hashtags "#weekend" and "#vacation". This data set requires Text Analytics Toolbox.
	
	Extract the text data from the file `weekendUpdates.xlsx` using the `readtable` function and extract the text 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 "Create Simple Preprocessing Function" (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 4-111.
		Sequence-to-sequence translation

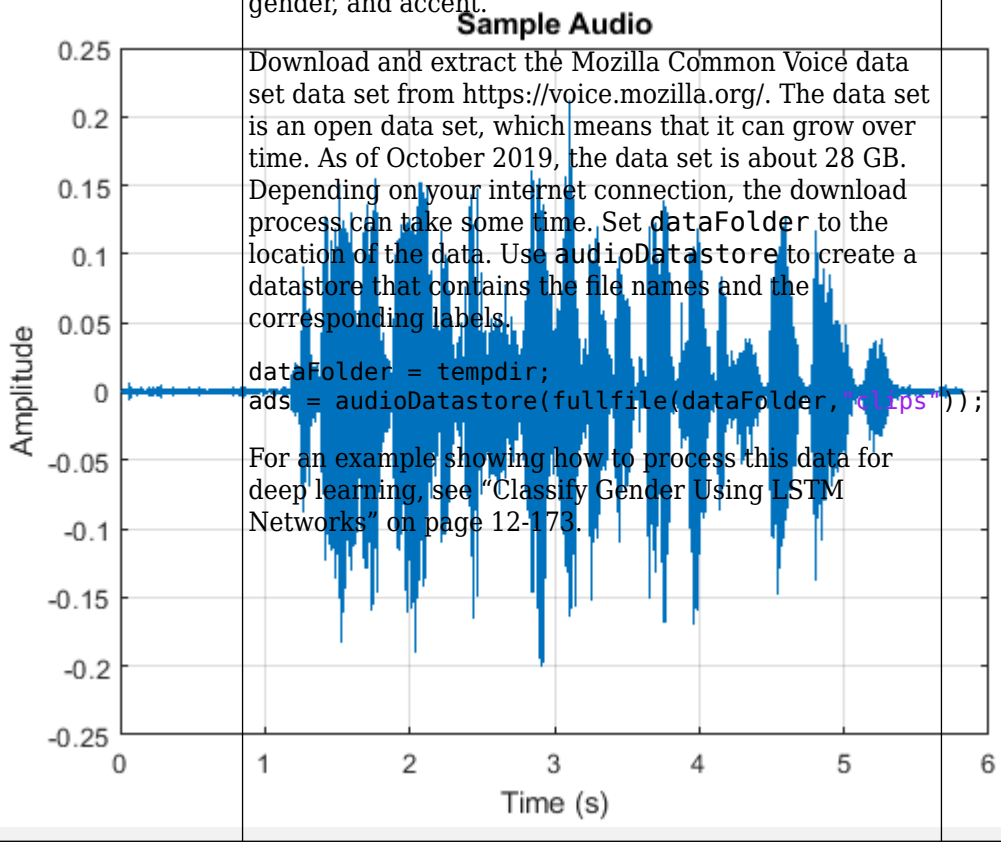
Audio Data Sets

Data	Description	Task

Speech Commands	The Speech Commands data set [18] 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.
	
	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. Depending on your internet connection, the download process can take some time.
	
	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 4-17.
		Audio classification, speech recognition

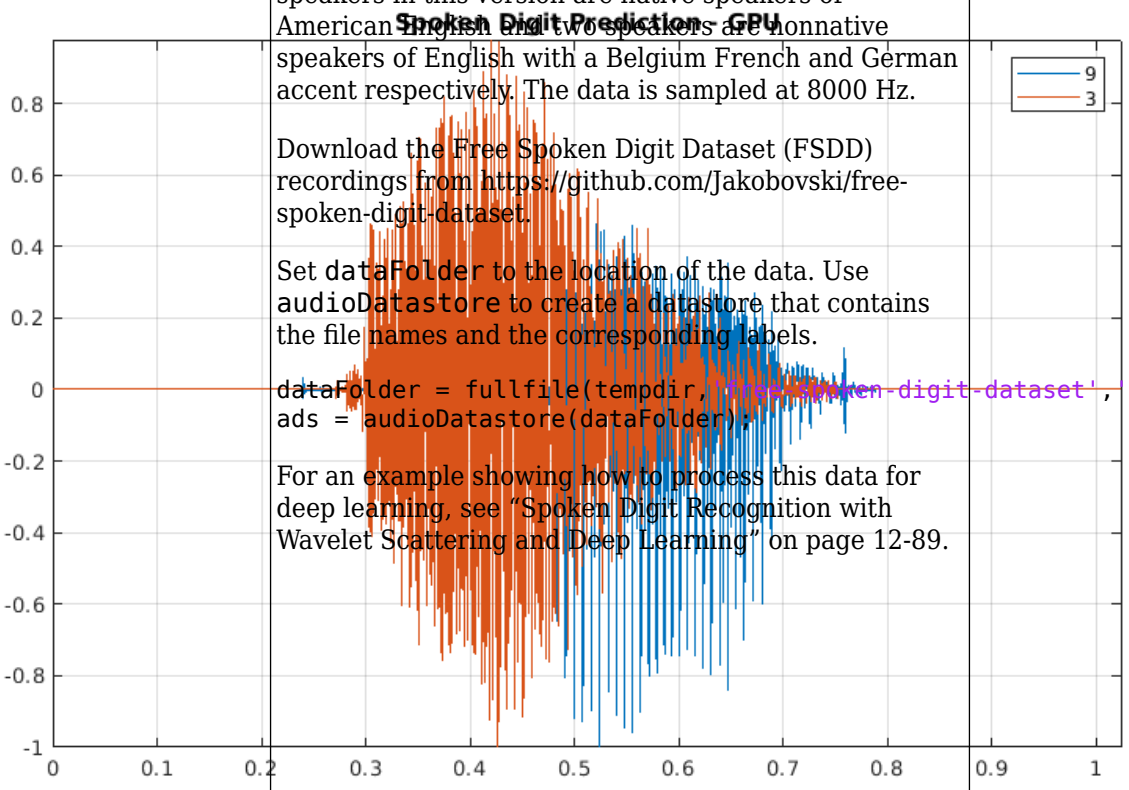


Data	Description	Task

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, gender, 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. Depending on your internet connection, the download process can take some time. 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"));`
	For an example showing how to process this data for deep learning, see "Classify Gender Using LSTM Networks" on page 12-173.
		Audio classification, speech recognition.

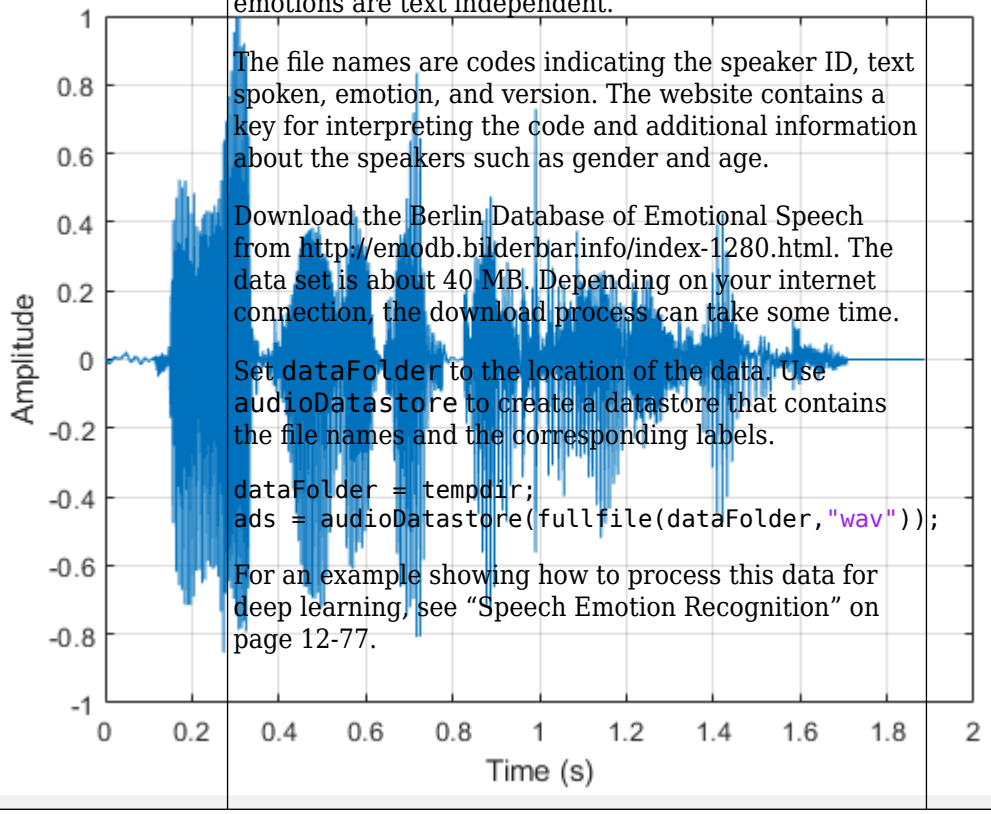
Data	Description	Task

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 12-89.
		Audio classification, speech recognition.

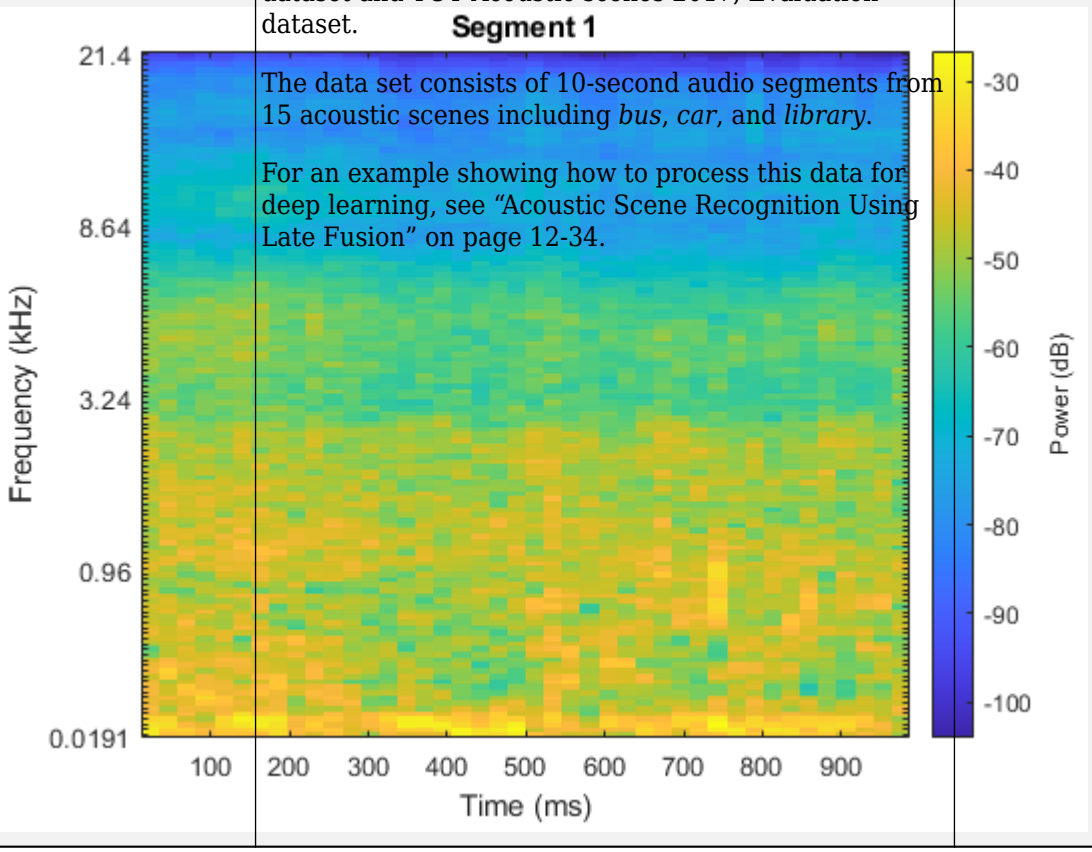


Data	Description	Task

Berlin Database of Emotional Speech	The Berlin Database of Emotional Speech [19] 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.
	
	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.
	
	Download the Berlin Database of Emotional Speech from http://emodb.bilderbar.info/index-1280.html. The data set is about 40 MB. Depending on your internet connection, the download process can take some time.
	
	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 12-77.
		Audio classification, speech recognition.



Data	Description	Task
TUT Acoustic scenes 2017	Download and extract the TUT Acoustic scenes 2017 data set from TUT Acoustic scenes 2017, Development dataset and TUT Acoustic scenes 2017, Evaluation dataset. The data set consists of 10-second audio segments from 15 acoustic scenes including *bus*, *car*, and *library*. For an example showing how to process this data for deep learning, see "Acoustic Scene Recognition Using Late Fusion" on page 12-34.	Acoustic scene classification



References

Mesaros, Annamaria, Toni Heittola, and Tuomas Virtanen. "Acoustic Scene Classification: An Overview of DCASE 2017 Challenge Entries." Proceedings of the International Workshop on Acoustic Signal Enhancement (2018):

411–415.

[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. http://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] 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

[13] 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

[14] 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

[15] 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.

[16] 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* 101, No. 23, 2000, pp. e215–e220. https://circ.ahajournals.org/content/101/23/e215.full

[17] 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.

[18] 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.

[19] 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.

See Also

`trainNetwork` | `trainingOptions`

More About

- Deep Network Designer
- “Pretrained Deep Neural Networks” on page 1-12
- “Create Simple Deep Learning Network for Classification” on page 3-40
- “Train Deep Learning Network to Classify New Images” on page 3-6
- “Deep Learning in MATLAB” on page 1-2

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.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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` 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: To view the report, open('codegen/mex/mobilenetv2_predict/html/repor
```

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.

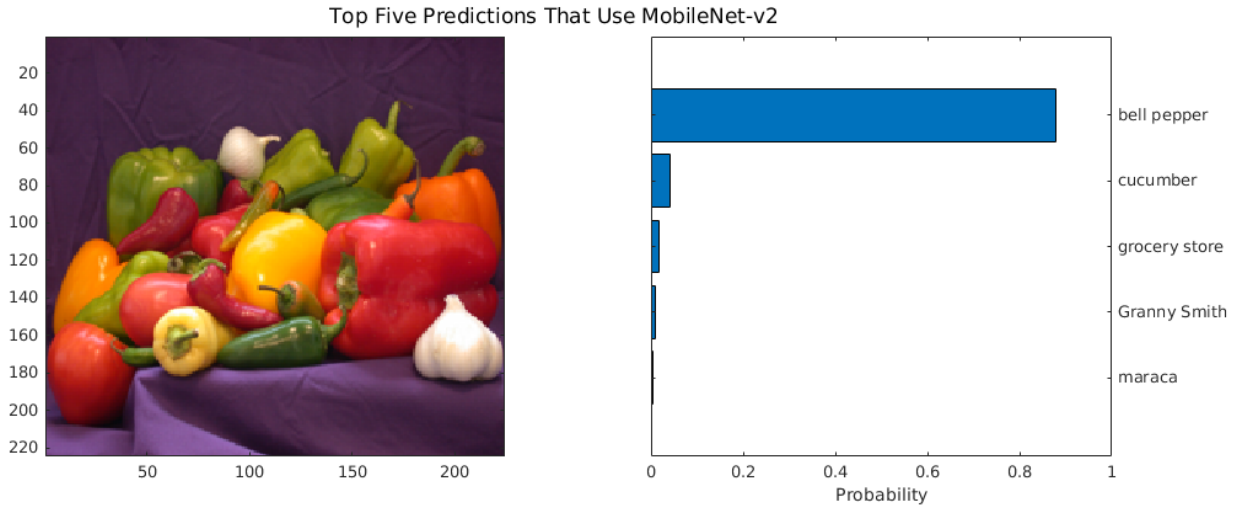
```
im = imread('peppers.png');
imshow(im);
```

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')
```



Classification of Video

The included helper function mobilenet_live.m grabs frames from a webcam, performs prediction, and displays the classification results on each of the captured video frames. This example uses the `webcam` function that is supported by the MATLAB® Support Package for USB Webcams™. You can download and install the support package through the Support Package Installer.

```
type('mobilenet_live.m')
```

```
% Copyright 2017-2019 The MathWorks, Inc.

function mobilenet_live

% Connect to a camera
camera = webcam;

% The labels with top 5 prediction scores are
% mapped to corresponding labels
net = mobilenetv2();
classnames = net.Layers(end).ClassNames;

imfull = zeros(224,400,3, 'uint8');
```

```
fps = 0;

ax = axes;

while true
    % Take a picture
    ipicture = camera.snapshot;

    % Resize and cast the picture to single
    picture = imresize(ipicture,[224,224]);

    % Call MEX function for MobileNet-v2 prediction
    tic;
    pout = mobilenetv2_predict(single(picture));
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % top 5 scores
    [top5labels, scores] = getTopFive(pout,classnames);

    % display
    if isvalid(ax)
        dispResults(ax, imfull, picture, top5labels, scores, fps);
    else
        break;
    end
end

end

function dispResults(ax, imfull, picture, top5labels, scores, fps)
for k = 1:3
    imfull(:,177:end,k) = picture(:,:,k);
end

h = imshow(imfull, 'InitialMagnification',200, 'Parent', ax);
scol = 1;
srow = 20;
text(get(h, 'Parent'), scol, srow, sprintf('MobileNet-v2 Demo'), 'color', 'w', 'FontSize', 20);
srow = srow + 20;

text(get(h, 'Parent'), scol, srow, sprintf('Fps = %2.2f', fps), 'color', 'w', 'FontSize', 15);
srow = srow + 20;
for k = 1:5
    t = text(get(h, 'Parent'), scol, srow, top5labels{k}, 'color', 'w','FontSize', 15);
    pos = get(t, 'Extent');
    text(get(h, 'Parent'), pos(1)+pos(3)+5, srow, sprintf('%2.2f%%', scores(k)), 'color', 'w', 'F
    srow = srow + 20;
end

drawnow;
end

function [labels, scores] = getTopFive(predictOut,classnames)
[val,indx] = sort(predictOut, 'descend');
```

```
scores = val(1:5)*100;
labels = classnames(indx(1:5));
end
```

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 Add-Ons (MATLAB).

```
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: To view the report, open('codegen/mex/resnet_predict/html/report.mlda
```

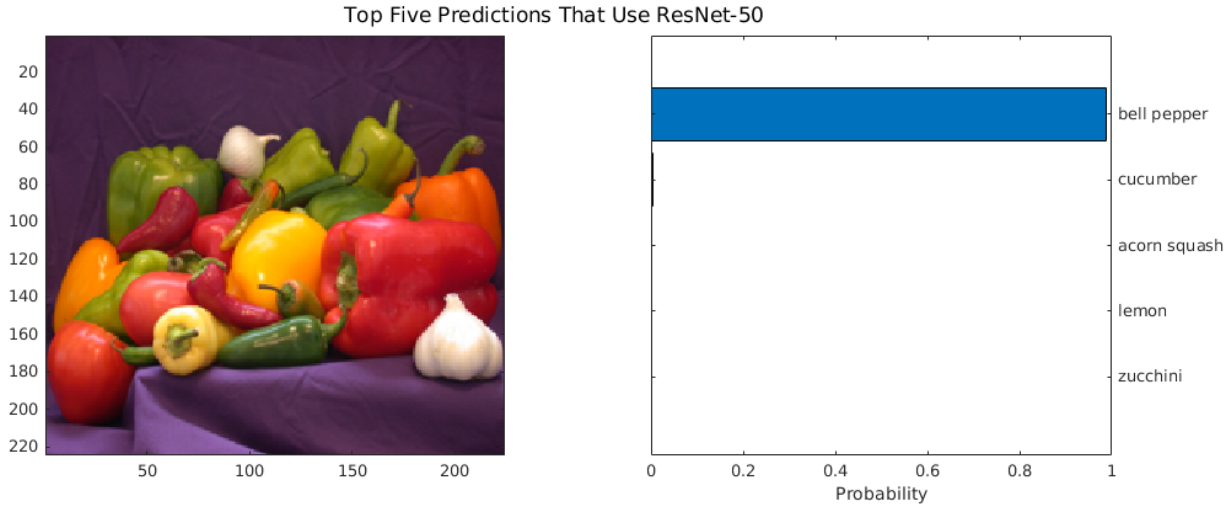
Call `resnet_predict_mex` on the input image.

```
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 Add-Ons (MATLAB).

```
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
```

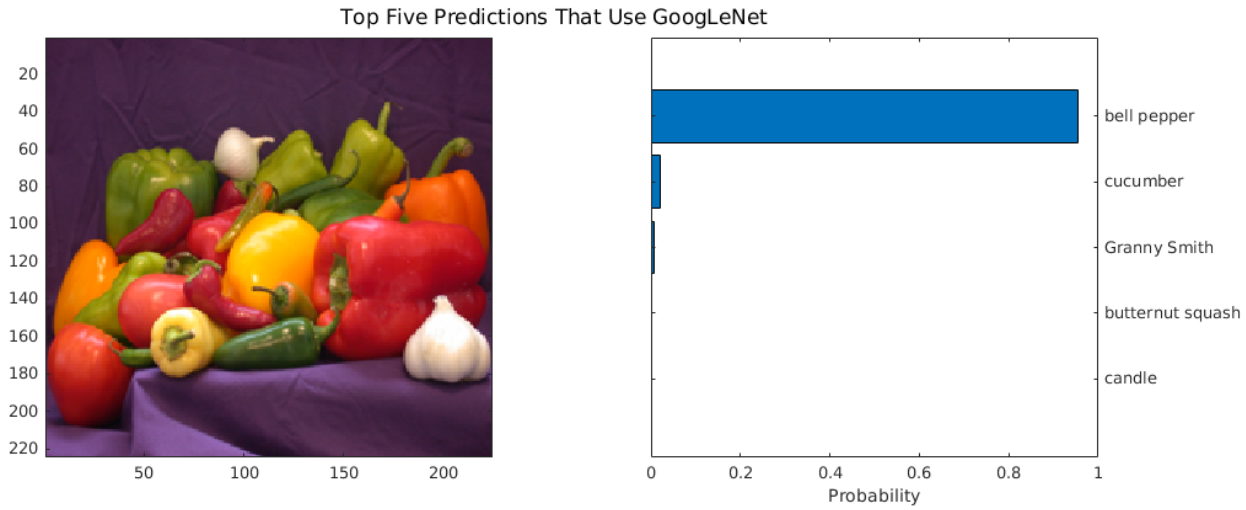
```
Code generation successful: To view the report, open('codegen/mex/googlenet_predict/html/report.m
```

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.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` 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.

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 2018-2019 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();
```

```
Downloading pretrained SegNet (107 MB)...
```

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` 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: To view the report, open('codegen/mex/segnet_predict/html/report.mlda
```

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
- “Computer Vision Using Deep Learning”

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 with compute capability 3.2 or higher.
- 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 Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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] = getLaneDetectionNetwork();
```

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.

```
laneNet.Layers
```

```
ans =

  23x1 Layer array with layers:

     1   'data'          Image Input                   227x227x3 images with 'zerocenter' normali
     2   'conv1'         Convolution                   96 11x11x3 convolutions with stride [4  4]
     3   'relu1'         ReLU                          ReLU
     4   'norm1'         Cross Channel Normalization   cross channel normalization with 5 channel
     5   'pool1'         Max Pooling                   3x3 max pooling with stride [2  2] and pad
     6   'conv2'         Convolution                   256 5x5x48 convolutions with stride [1  1]
     7   'relu2'         ReLU                          ReLU
```

```
     8   'norm2'         Cross Channel Normalization   cross channel normalization with 5 channe
     9   'pool2'         Max Pooling                   3x3 max pooling with stride [2  2] and pa
    10   'conv3'         Convolution                   384 3x3x256 convolutions with stride [1
    11   'relu3'         ReLU                          ReLU
    12   'conv4'         Convolution                   384 3x3x192 convolutions with stride [1
    13   'relu4'         ReLU                          ReLU
    14   'conv5'         Convolution                   256 3x3x192 convolutions with stride [1
    15   'relu5'         ReLU                          ReLU
    16   'pool5'         Max Pooling                   3x3 max pooling with stride [2  2] and pa
    17   'fc6'           Fully Connected               4096 fully connected layer
    18   'relu6'         ReLU                          ReLU
    19   'drop6'         Dropout                       50% dropout
    20   'fcLane1'       Fully Connected               16 fully connected layer
    21   'fcLane1Relu'   ReLU                          ReLU
    22   'fcLane2'       Fully Connected               6 fully connected layer
    23   'output'        Regression Output             mean-squared-error with 'leftLane_a', 'le
```

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` 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++';
codegen -args {ones(227,227,3,'single'),ones(1,6,'double'),ones(1,6,'double')} -config cfg detec
```

```
Code generation successful: To view the report, open('codegen/lib/detect_lane/html/report.mldatx
```

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);
  real32_T *getInputDataPointer();
  real32_T *getOutputDataPointer();
  ~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)
```

```
.                                   cnn_lanenet_data_offset.bin
..                                  cnn_lanenet_data_scale.bin
DeepLearningNetwork.cu              cnn_lanenet_fc6_b.bin
DeepLearningNetwork.h               cnn_lanenet_fc6_w.bin
DeepLearningNetwork.o               cnn_lanenet_fcLane1_b.bin
```

```
MWCNNLayerImpl.cu                       cnn_lanenet_fcLane1_w.bin
MWCNNLayerImpl.hpp                      cnn_lanenet_fcLane2_b.bin
MWCNNLayerImpl.o                        cnn_lanenet_fcLane2_w.bin
MWCudaDimUtility.cu                     cnn_lanenet_responseNames.txt
MWCudaDimUtility.hpp                    codeInfo.mat
MWCudaDimUtility.o                      codedescriptor.dmr
MWElementwiseAffineLayer.cpp            coder_array.h
MWElementwiseAffineLayer.hpp            compileInfo.mat
MWElementwiseAffineLayer.o              detect_lane.a
MWElementwiseAffineLayerImpl.cu         detect_lane.cu
MWElementwiseAffineLayerImpl.hpp        detect_lane.h
MWElementwiseAffineLayerImpl.o          detect_lane.o
MWElementwiseAffineLayerImplKernel.cu   detect_lane_data.cu
MWElementwiseAffineLayerImplKernel.o    detect_lane_data.h
MWFusedConvReLULayer.cpp                detect_lane_data.o
MWFusedConvReLULayer.hpp                detect_lane_initialize.cu
MWFusedConvReLULayer.o                  detect_lane_initialize.h
MWFusedConvReLULayerImpl.cu             detect_lane_initialize.o
MWFusedConvReLULayerImpl.hpp            detect_lane_ref.rsp
MWFusedConvReLULayerImpl.o              detect_lane_rtw.mk
MWKernelHeaders.hpp                     detect_lane_rtwutil.cu
MWTargetNetworkImpl.cu                  detect_lane_rtwutil.h
MWTargetNetworkImpl.hpp                 detect_lane_rtwutil.o
MWTargetNetworkImpl.o                   detect_lane_terminate.cu
buildInfo.mat                           detect_lane_terminate.h
cnn_api.cpp                             detect_lane_terminate.o
cnn_api.hpp                             detect_lane_types.h
cnn_api.o                               examples
cnn_lanenet_conv1_b.bin                 gpu_codegen_info.mat
cnn_lanenet_conv1_w.bin                 html
cnn_lanenet_conv2_b.bin                 interface
cnn_lanenet_conv2_w.bin                 predict.cu
cnn_lanenet_conv3_b.bin                 predict.h
cnn_lanenet_conv3_w.bin                 predict.o
cnn_lanenet_conv4_b.bin                 rtw_proj.tmw
cnn_lanenet_conv4_w.bin                 rtwtypes.h
cnn_lanenet_conv5_b.bin
cnn_lanenet_conv5_w.bin
```

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.cpp`.

```
type main_lanenet.cpp
```

```
/* Copyright 2016 The MathWorks, Inc. */

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "opencv2/opencv.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, CV_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",CV_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, cvPoint(200,30), CV_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

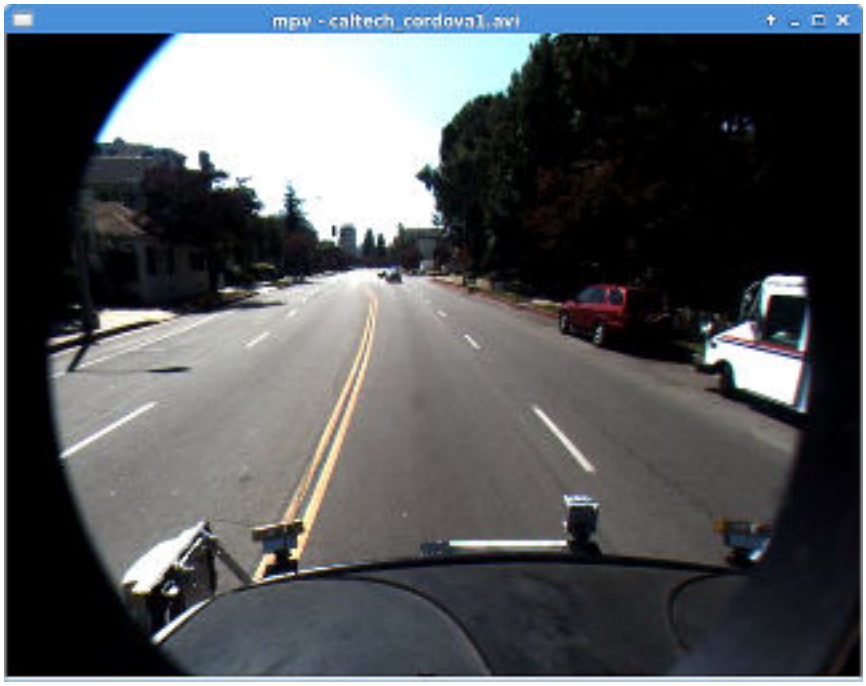
```
if ~exist('./caltech_cordova1.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/caltech_cordova1.avi';
    websave('caltech_cordova1.avi', url);
end
```

Build Executable

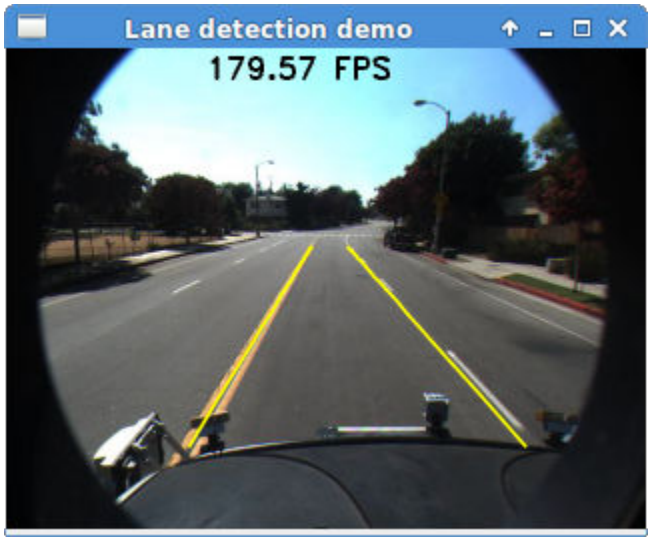
```
if ispc
    setenv('MATLAB_ROOT', matlabroot);
    vcvarsall = mex.getCompilerConfigurations('C++').Details.CommandLineShell;
    setenv('VCVARSALL', vcvarsall);
    [~,~] = system('make_win_lane_detection.bat');
```

```
    cd(codegendir);
    [status,cmdout] = system('lanenet.exe ..\..\..\caltech_cordova1.avi');
else
    setenv('MATLAB_ROOT', matlabroot);
    [~,~] = system('make -f Makefile_lane_detection.mk');
    cd(codegendir);
    [status,cmdout] = system('./lanenet ../../../caltech_cordova1.avi');
end
```

Input Screenshot



Output Screenshot



See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Computer Vision Using Deep Learning”

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™.

Prerequisites

- CUDA enabled NVIDIA® GPU with compute capability 3.5 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see "Third-party Products" (GPU Coder). For setting up the environment variables, see "Setting Up the Prerequisite Products" (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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 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` command by using the `coder.typeof` 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: To view the report, open('codegen/mex/lstmnet_predict/html/report.mlc
```

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))
```

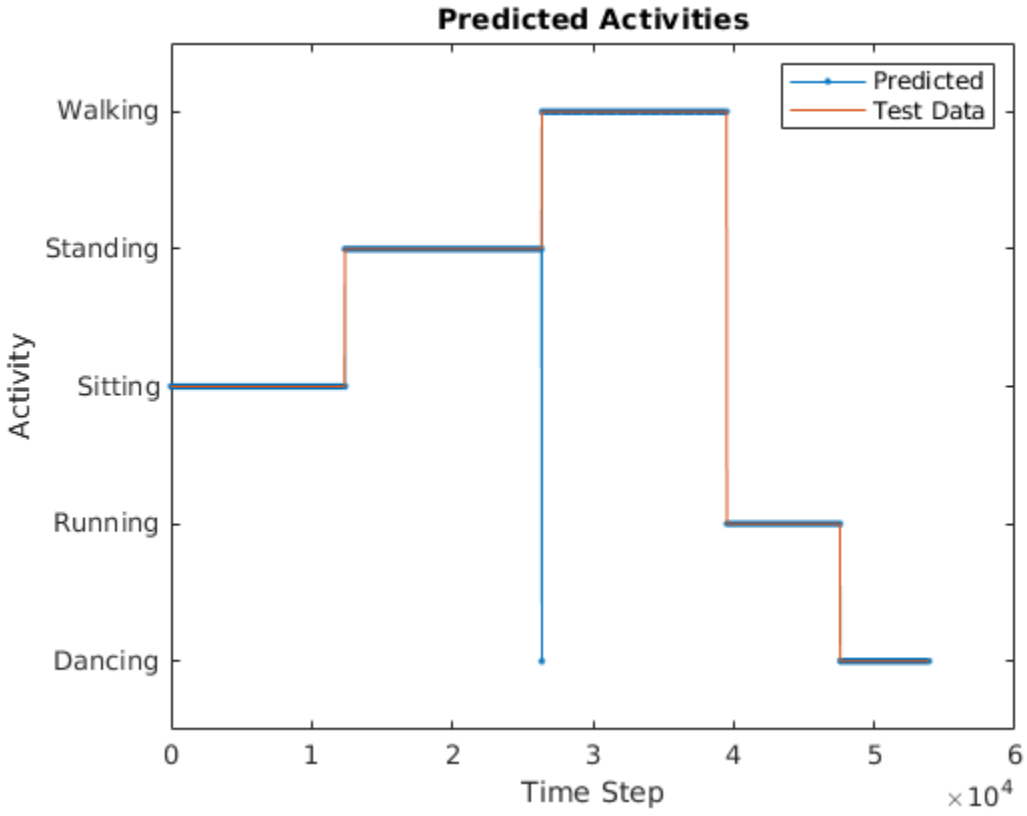
```
  Columns 1 through 6

     Sitting      Sitting      Sitting      Sitting      Sitting      Sitting
```

```
  Columns 7 through 10

     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))
```

```
  Columns 1 through 6

     Sitting      Sitting      Sitting      Sitting      Sitting      Sitting

  Columns 7 through 10

     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

YPred3 = lstmnet_predict_mex(XValidate);
```

```
Code generation successful: To view the report, open('codegen/mex/lstmnet_predict/html/report.ml
```

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 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: To view the report, open('codegen/mex/lstmnet_predict_and_update/htm
```

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 Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder™ Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Get Pretrained DAGNetwork

Load the pretrained `MobileNet-v2` network available in the `Deep Learning Toolbox Model for MobileNet-v2 Network`.

```
net = mobilenetv2;
```

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_
```

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).

Prerequisites

- CUDA enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see "Third-party Products" (GPU Coder). For setting up the environment variables, see "Setting Up the Prerequisite Products" (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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 = getYOLOv2();
```

```
Downloading pretrained detector (98 MB)...
```

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)
```

```
%   Copyright 2018-2019 The MathWorks, Inc.

persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

% pass in input
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);

% convert categorical labels to cell array of charactor vectors for MATLAB
% execution
if coder.target('MATLAB')
    labels = cellstr(labels);
end

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
```

Run MEX Code Generation

To generate CUDA code for the yolov2_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` 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 [224,224,3]. This value corresponds to the input layer size of YOLOv2.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg yolov2_detect -args {ones(224,224,3,'uint8')} -report

Code generation successful: To view the report, open('codegen/mex/yolov2_detect/html/report.mldat
```

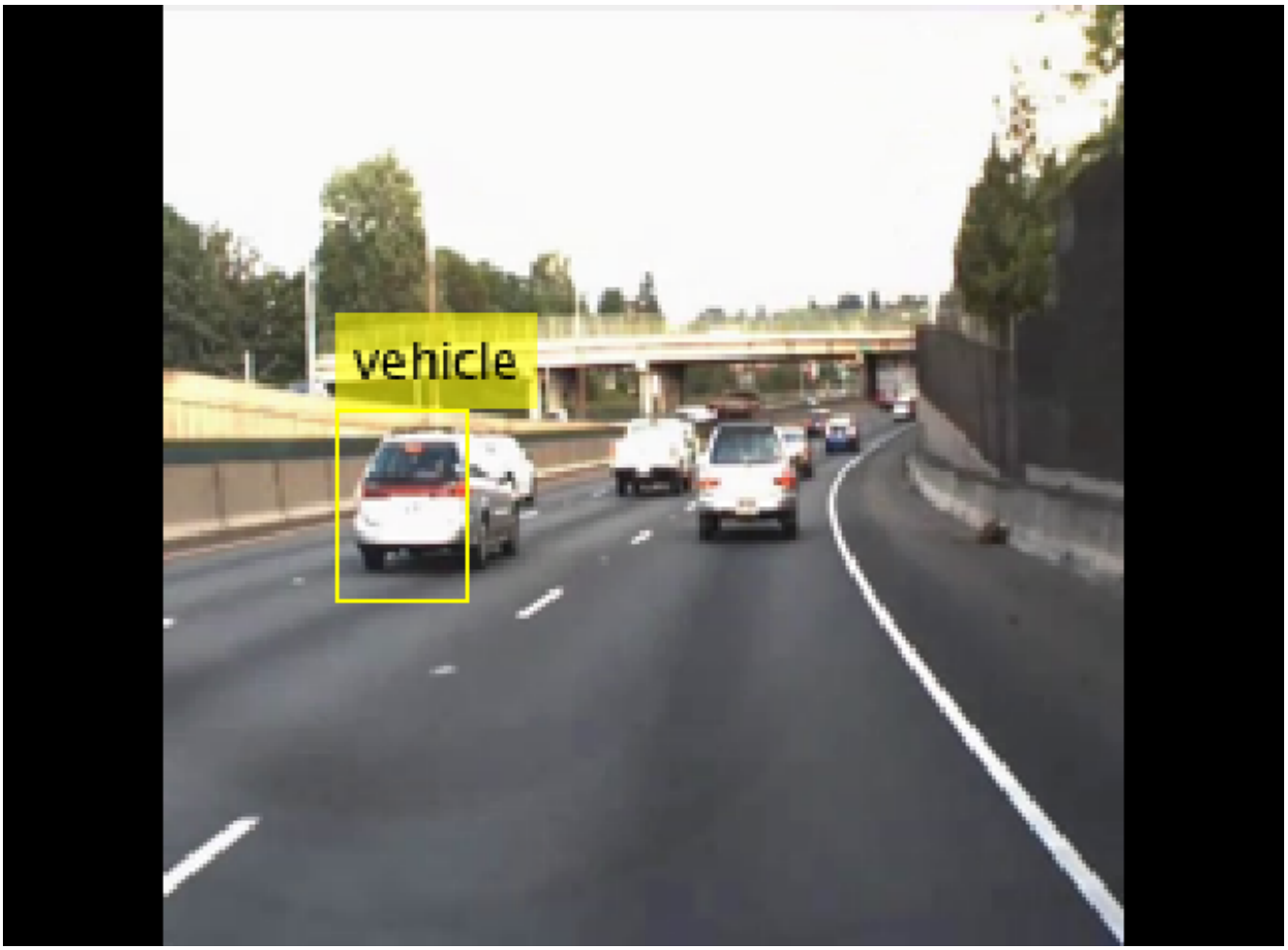
Run Generated MEX

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);
    step(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." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Integrating Deep Learning with GPU Coder into Simulink

This example shows how to integrate the CUDA® code generated for a deep learning network into Simulink®. GPU coder™ does not support code generation for Simulink blocks but you can still use the computational power of GPUs in Simulink by generating a dynamic linked library (dll) with GPU Coder and then integrating it into Simulink as an S-Function block by using the legacy code tool. For more information, see legacy_code. To illustrate this concept, the example uses “Lane Detection Optimized with GPU Coder” (GPU Coder). The original example used a C++ file with OpenCV functions to read the frames, draw lanes, and overlay frame rate information on the video output. This example uses Simulink blocks from the Computer Vision System Toolbox™ to perform the same operations.

Prerequisites

- CUDA enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder™ Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

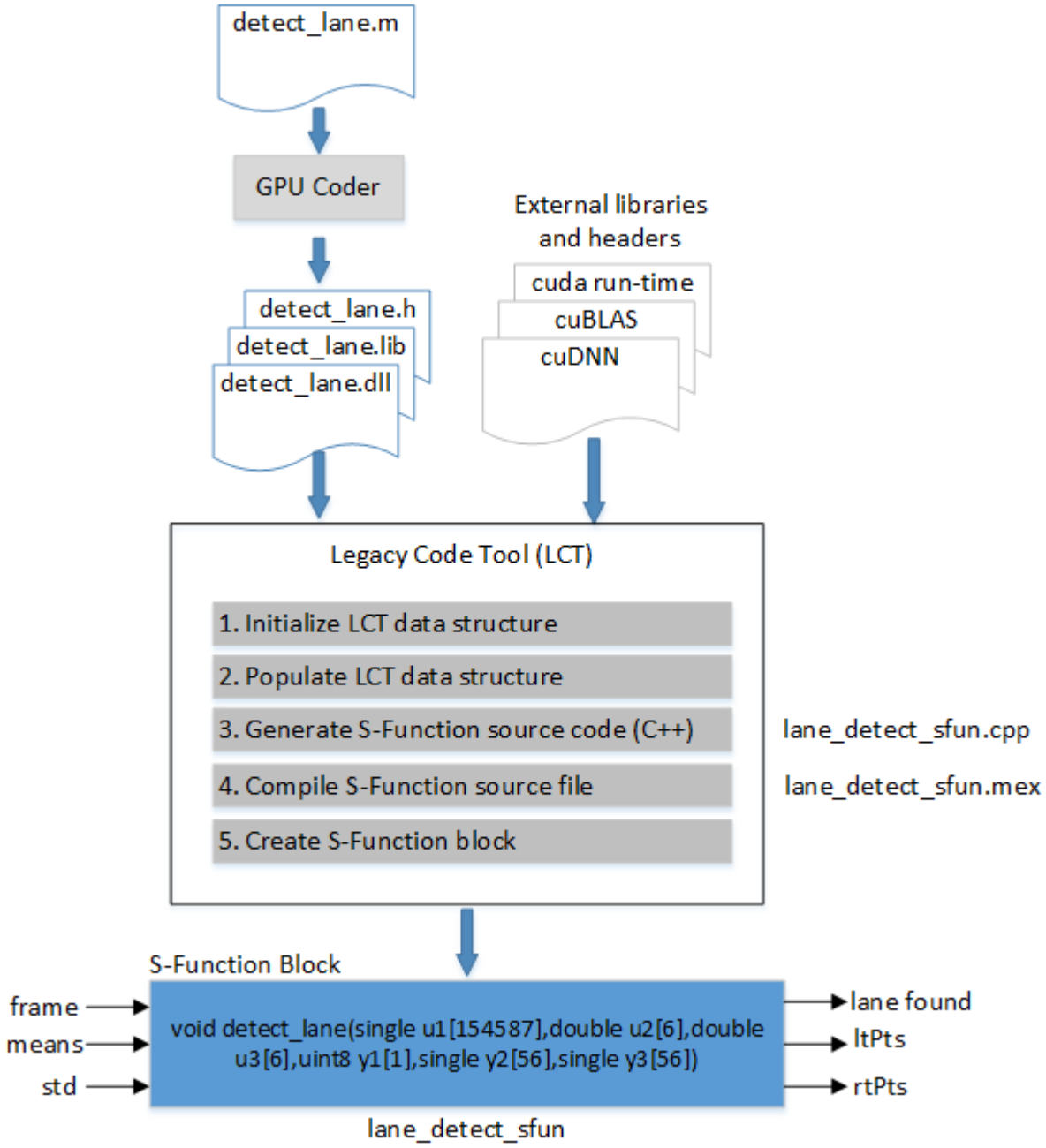
Verify GPU Environment

Use the `coder.checkGpuInstall` 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);
```

Workflow

This diagram illustrates the general procedure for using the `Legacy Code Tool` to integrate the CUDA code generated for a deep learning network into Simulink.



Get Pretrained SeriesNetwork

```
[laneNet,coeffMeans,coeffStds] = getLaneDetectionNetwork();
```

The architecture of the pretrained `SeriesNetwork` is similar to `AlexNet` except that the last few layers are replaced by a smaller, fully connected layer and regression output layer. This network takes an image input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by a 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 that describe the parabolic equation for the left and right lane boundaries. The

variables `coeffStds` and `coeffMeans` contain the mean and std values from the trained network. These values are required during simulation.

Main Entry Point Function

This example uses the detect_lane.m entry-point function. The `detect_lane` function computes the x and y coordinates corresponding to the lane positions from the a, b, and c parameters. The `detect_lane` function also performs computations that map the x and y coordinates to image coordinates.

Generate a Dynamic Link Library (DLL) for the Function

To run the `detect_lane` function on the GPU from Simulink, generate a shared library by using GPU Coder. The inputs to the `detect_lane` function are the video frame, mean, and std values. The values passed by using the `-args` option reflect the size of these inputs. Copy the generated library to the top-level folder.

```
Isize = single(zeros(227,227));

cfg = coder.gpuConfig('dll');
cfg.TargetLang = 'C++';
cfg.GenerateReport = true;
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -args {ones(227,227,3,'single'),ones(1,6,'double'),ones(1,6,'double')} -config cfg detec

if ispc
    copyfile(fullfile(pwd, 'codegen','dll', 'detect_lane','detect_lane.dll'), pwd);
else
    copyfile(fullfile(pwd, 'codegen','dll', 'detect_lane','detect_lane.so'), pwd);
end

Code generation successful: To view the report, open('codegen/dll/detect_lane/html/report.mldatx
```

Generate and Compile S-Function

The lane detection example depends on the NVIDIA CUDA run time, cuBLAS, and the cuDNN library. The `Legacy Code Tool` data structure specifies:

- A name for the S-function
- Specifications for the existing C++ function
- All library and header files required for compilation and the file paths
- Options for the generated S-function

After defining the structure, use the legacy_code function to:

- Initialize the `Legacy Code Tool` data structure for the C++ function
- Generate an S-function for use during simulation
- Compile and link the generated S-function into a dynamically loadable executable (MEX)
- Generate a masked S-function block for calling the generated S-function

```
srcPath = fullfile(pwd, 'codegen', 'dll', 'detect_lane');

if ispc
    cuPath = getenv('CUDA_PATH');
    cudaLibPath = fullfile(cuPath,'lib','x64');
```

```
    cudaIncPath = fullfile(cuPath,'include');

    cudnnPath = getenv('NVIDIA_CUDNN');
    cudnnIncPath = fullfile(cudnnPath,'include');
    cudnnLibPath = fullfile(cudnnPath,'lib','x64');

    libs = {'detect_lane.lib','cudart.lib','cublas.lib','cudnn.lib'};

else
    [~,nvccPath] = system('which nvcc');
    nvccPath = regexp(nvccPath, '[\f\n\r]', 'split');
    cuPath = erase(nvccPath{1},'/bin/nvcc');
    cudaLibPath = fullfile(cuPath,'lib64');
    cudaIncPath = fullfile(cuPath,'include');

    cudnnPath = getenv('NVIDIA_CUDNN');
    cudnnIncPath = fullfile(cudnnPath,'include');
    cudnnLibPath = fullfile(cudnnPath,'lib64');

    [~,cmdout] = system('ldconfig -p | grep "libcublas.so "');
    pathStrIdx =  strfind(cmdout,'/usr/');
    cublasLibPath = fileparts(cmdout(33:end));
    cublasIncPath = '/usr/include';

    libs = {'detect_lane.so','libcudart.so','libcublas.so','libcudnn.so'};
end

headerPath = {srcPath;cudnnIncPath;cudaIncPath;cublasIncPath};
libPath = {srcPath;cudnnLibPath;cudaLibPath;cublasLibPath};

% Define the Legacy Code Tool data structure
def = legacy_code('initialize');
def.SFunctionName = 'lane_detect_sfun';
def.OutputFcnSpec = 'void detect_lane(single u1[154587],double u2[6],double u3[6],uint8 y1[1],si
def.IncPaths = headerPath;
def.HeaderFiles = {'detect_lane.h'};
def.LibPaths = libPath;
def.HostLibFiles = libs;
def.Options.useTlcWithAccel = false;
def.Options.language = 'C++';

legacy_code('sfcn_cmex_generate', def);
status = evalc("legacy_code('compile', def)");
```

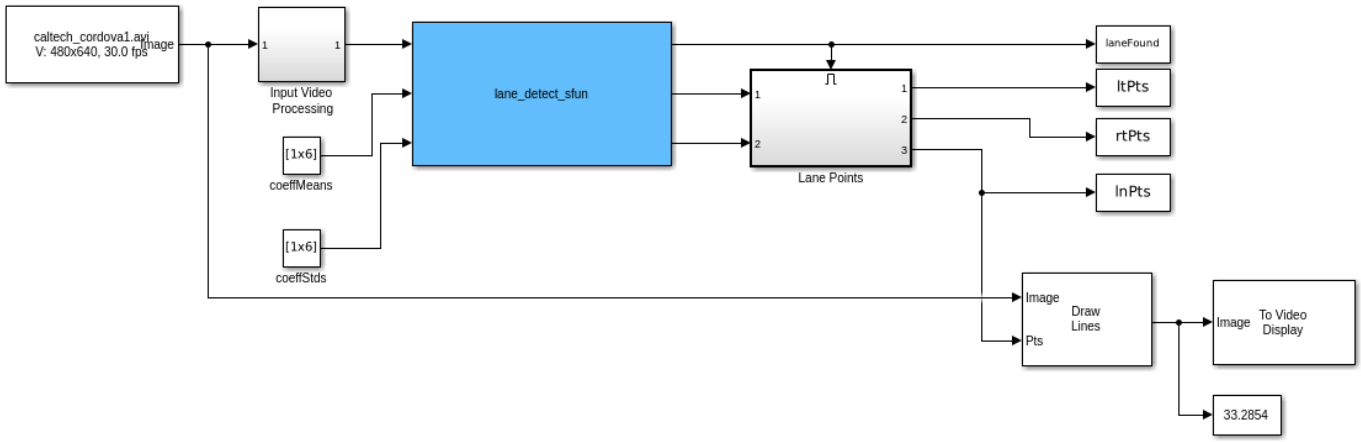
The `OutputFcnSpec` argument specifies the function that the S-function calls at each time step. The `detect_lane.h` header file in the codegen folder provides the function specification information. Map the `detect_lane` function arguments to the Simulink S-Function block by using a uniquely numbered `u` token for input ports and the `y` token for output ports. The code generation data types defined in tmwtypes.h must also be mapped to the data types that Simulink supports. For more information, see Declaring Legacy Code Tool Function Specifications. Because this example already contains a complete Simulink model, generation of the S-Function block is not performed. To generate the S-Function block, use:

```
legacy_code('slblock_generate', def);
```

Create Simulink Model for Lane Detection

Move all the pre- and post-processing operations in the `main_lanenet.cpp` file of the original example into Simulink. The Input Video Processing subsystem removes normalization performed by the multimedia reader block and resizes the input video frame to the input layer size of the lane detection network, 227-by-227-by-3. The subsystem then converts the three-dimensional video frame into the one-dimensional vector required by the `detect_lane` library. The Lane Points enabled subsystem processes of the left and right lane points to make them suitable for the Draw Lanes block. The Simulink model uses a video display to show lane detection on a sample video.

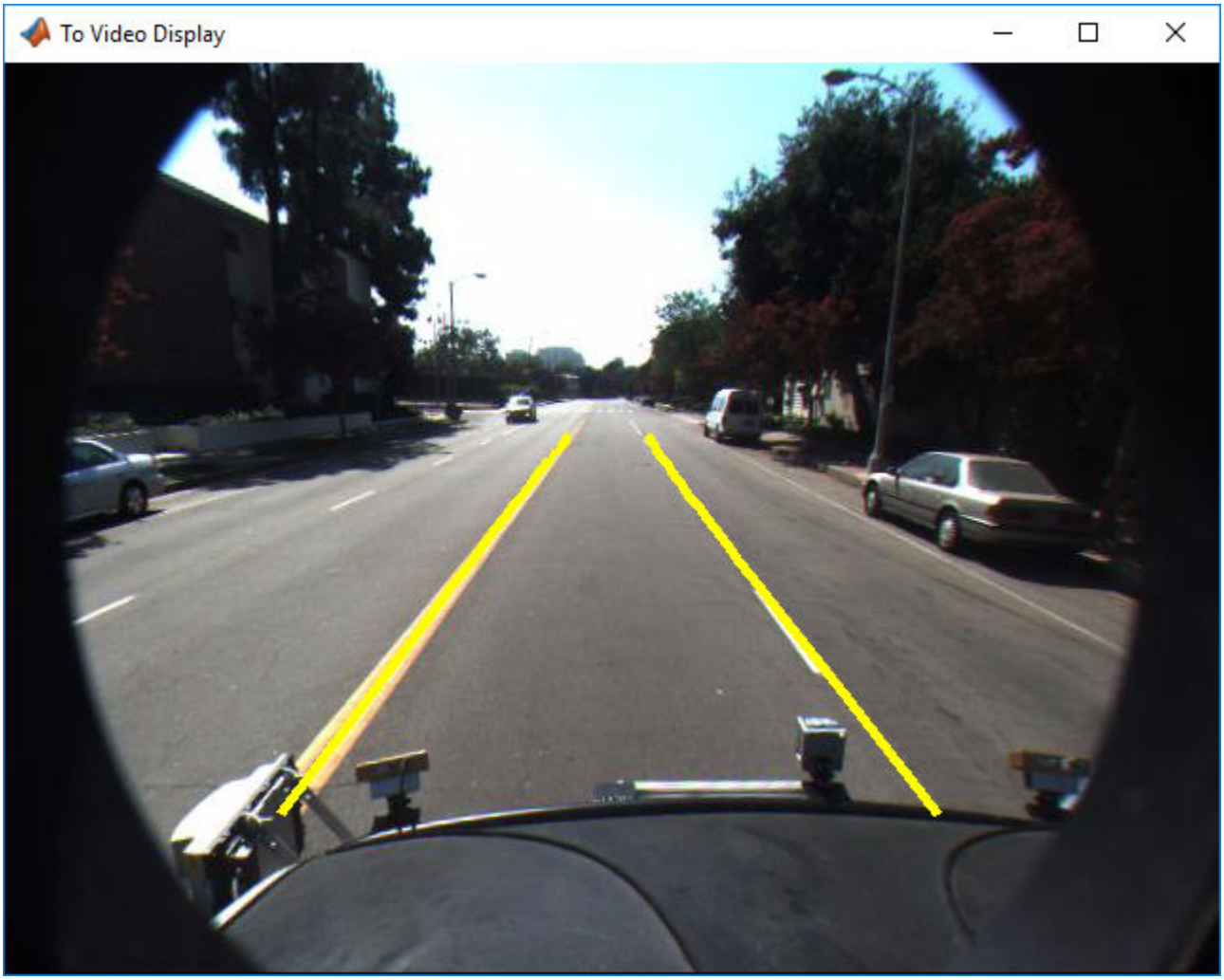
```
open_system('main_lanenet');
set_param('main_lanenet', 'SimulationCommand', 'update');
```



Run Simulink Model (Lane Detection)

To see lane detection on a sample video, run simulation.

```
sim('main_lanenet', 'timeout', 30);
```



Cleanup

Close the Simulink model.

```
close_system('main_lanenet');
```

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.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher. For the 8-bit integer example, NVIDIA GPU with compute capability 6.1 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN and TensorRT library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning support package for deep learning code generation.
- This example is not supported in MATLAB® online.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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. To learn more about finding and installing add-ons, see Get Add-Ons (MATLAB).

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 2018 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` 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: To view the report, open('codegen/mex/resnet_predict/html/report.mlda
```

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();
```

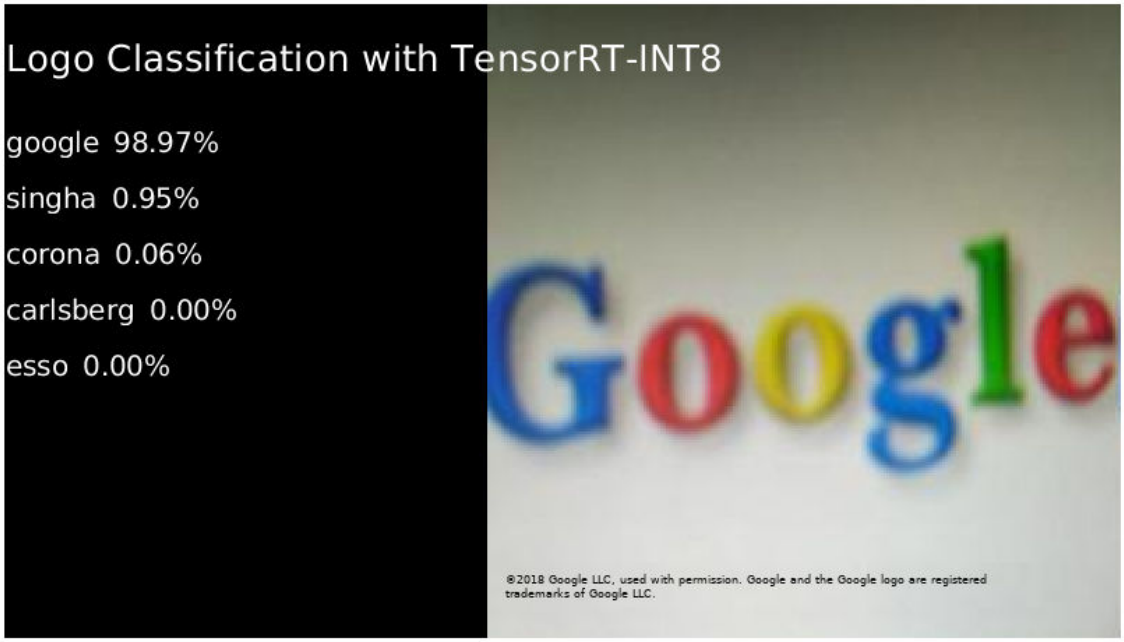
TensorRT requires a calibration data set to calibrate a network that is trained in floating-point to compute inference in 8-bit integer precision. Set the data type to int8 and the path to the calibration data set by using the `DeepLearningConfig`. `logos_dataset` is a subfolder containing images grouped by their corresponding classification labels. For int8 support, GPU compute capability must be 6.1 or higher.

```
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: To view the report, open('codegen/mex/logonet_predict/html/report.ml
```

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;
```

Deep Learning Prediction by Using Different Batch Sizes

This example demonstrates code generation with batch sizes greater than 1. This demo contains two examples, first, uses `cnncodegen` to generate code which takes in a batch of images as input. The second example creates MEX file using `codegen` and passes a batch of images as input.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN and TensorRT library.
- OpenCV 3.1.0 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 Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- This example is not supported on MATLAB® online.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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);
```

Classification by Using ResNet-50 Network

The example uses 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 Add-Ons (MATLAB). Use the `analyzeNetwork` function to display an interactive visualization of the deep learning network architecture.

```
net = resnet50;
analyzeNetwork(net);
```

Generate Code for NVIDIA GPUs by Using TensorRT Library

For an NVIDIA target with TensorRT, code generation and execution is performed on the host development computer. To run the generated code, your development computer must have an NVIDIA GPU with compute capability of at least 3.2. Use the `cnncodegen` command to generate code for the NVIDIA platform by using the `'tensorrt'` option. By default, the `cnncodegen` command generates code that uses 32-bit float-point precision for the tensor inputs to the network. In the predict call, multiple images can be batched into a single call and passed as an input. This call performs predictions over the batch of inputs in parallel. The default value of the batch size is 1.

You can specify the input batch size by using the `'batchsize'` option. During execution, the generated code expects the same batch size value to be used. Passing a different batch size value at runtime causes errors. In this example 15 images are considered as a batch.

To generate code using cuDNN specify `'cudnn'` instead of `'tensorrt'` for the `'targetlib'` option.

```
status = evalc("cnncodegen(net,'targetlib','tensorrt', 'batchsize', 15)");
```

Generated Code Description

The `presetup()` and `postsetup()` functions perform additional configuration required for TensorRT. Layer classes in the generated code folder call into the TensorRT libraries.

Main File

The main file creates and sets up the CnnMain network object with layers and weights. It uses the OpenCV `VideoCapture` method to read frames from input video. It performs prediction for each frame and fetches the output from the final fully connected layer.

Frames obtained from OpenCV `VideoCapture` object are converted from packed BGR (OpenCV) format to planar RGB (MATLAB) format. A buffer is allocated and filled with the image data. This raw buffer is an input to the network.

```
   void readBatchData(float *input, vector<Mat>& orig, int batchSize)
   {
        for (int i=0; i<batchSize; i++)
        {
           if (orig[i].empty())
           {
               orig[i] = Mat::zeros(ROWS,COLS, orig[i-1].type());
               continue;
           }

           Mat tmpIm;
           resize(orig[i], tmpIm, Size(COLS,ROWS));

           for (int j=0; j<ROWS*COLS; j++)
           {
               // BGR packed to RGB planar conversion
               input[CH*COLS*ROWS*i + 2*COLS*ROWS + j] = (float)(tmpIm.data[j*3+0]);
               input[CH*COLS*ROWS*i + 1*COLS*ROWS + j] = (float)(tmpIm.data[j*3+1]);
               input[CH*COLS*ROWS*i + 0*COLS*ROWS + j] = (float)(tmpIm.data[j*3+2]);
           }
        }
   }
```

Build and Run Executable

Download the 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
```

Use the `make` command to build the `resnet_batchSize_exe` executable. Run the executable and specify batch size as the first argument and the name of the video file as the second argument.

```
if isunix
   system(['make -f Makefile_resnet_batchsize_linux.mk ','tensorrt']);
```

```
   system('./resnet_batchSize_exe 15 object_class.avi');
elseif ispc
   system('make_resnet_batchsize_win.bat');
   system('resnet_predict.exe 15 object_class.avi');
end
```

Generate CUDA MEX for the `resnet_predict` Function

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` 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 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.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
codegen -config cfg resnet_predict -args {ones(224,224,3,batchSize,'uint8')} -report
```

```
Code generation successful: To view the report, open('codegen/mex/resnet_predict/html/report.mlda
```

Perform Prediction on Test Image Batch

```
im = imread('peppers.png');
im = imresize(im, [224,224]);
```

Concatenating 5 images since `batchSize = 5`.

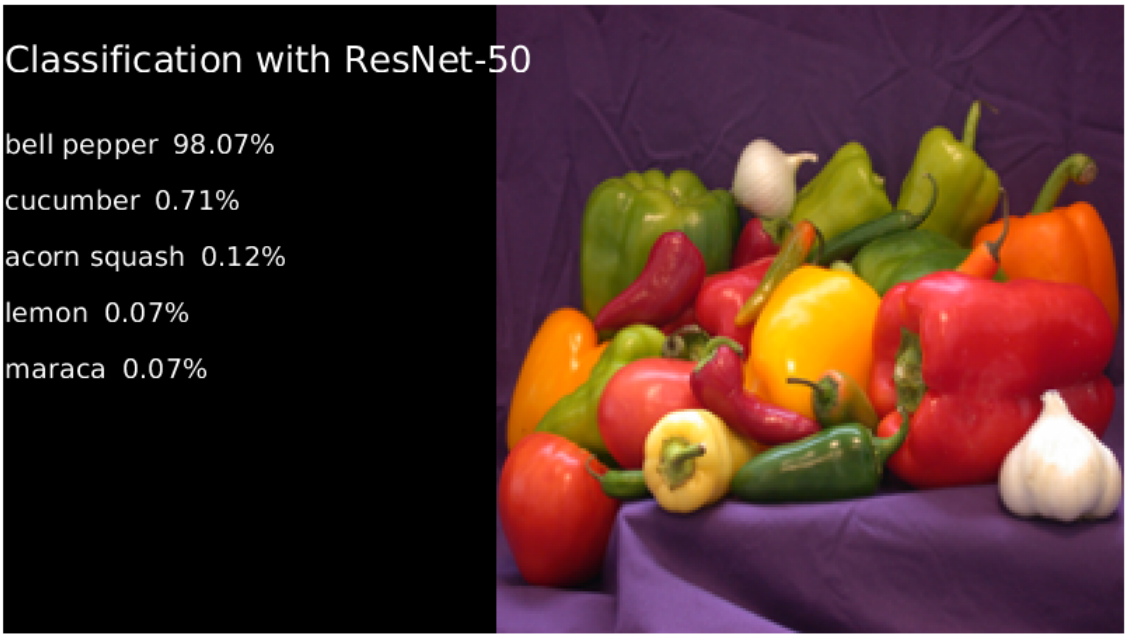
```
imBatch = cat(4,im,im,im,im,im);
predict_scores = resnet_predict_mex(imBatch);
```

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)]);
    disp(labels);
end
```

```
Top 5 predictions on image, 1
    {'bell pepper' }
    {'cucumber'    }
    {'lemon'       }
    {'acorn squash'}
    {'hamper'      }

Top 5 predictions on image, 2
    {'bell pepper' }
    {'cucumber'    }
    {'lemon'       }
    {'acorn squash'}
    {'hamper'      }
```

```
Top 5 predictions on image, 3
    {'bell pepper' }
    {'cucumber'    }
    {'lemon'       }
    {'acorn squash'}
    {'hamper'      }

Top 5 predictions on image, 4
    {'bell pepper' }
    {'cucumber'    }
    {'lemon'       }
    {'acorn squash'}
    {'hamper'      }

Top 5 predictions on image, 5
    {'bell pepper' }
    {'cucumber'    }
    {'lemon'       }
    {'acorn squash'}
    {'hamper'      }
```

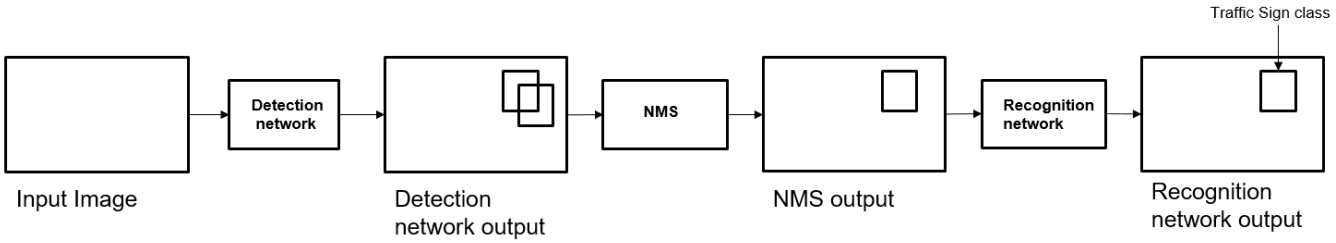


Clear the static network object that was loaded in memory.

```
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.

Prerequisites

- CUDA enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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.Layers
```

```
ans =

  58x1 Layer array with layers:

     1   'input'         Image Input             448x448x3 images
     2   'conv1'         Convolution             64 7x7x3 convolutions with stride [2  2] and pa
     3   'relu1'         Leaky ReLU              Leaky ReLU with scale 0.1
     4   'pool1'         Max Pooling             2x2 max pooling with stride [2  2] and padding
     5   'conv2'         Convolution             192 3x3x64 convolutions with stride [1  1] and
     6   'relu2'         Leaky ReLU              Leaky ReLU with scale 0.1
     7   'pool2'         Max Pooling             2x2 max pooling with stride [2  2] and padding
     8   'conv3'         Convolution             128 1x1x192 convolutions with stride [1  1] and
     9   'relu3'         Leaky ReLU              Leaky ReLU with scale 0.1
    10   'conv4'         Convolution             256 3x3x128 convolutions with stride [1  1] and
    11   'relu4'         Leaky ReLU              Leaky ReLU with scale 0.1
    12   'conv5'         Convolution             256 1x1x256 convolutions with stride [1  1] and
    13   'relu5'         Leaky ReLU              Leaky ReLU with scale 0.1
    14   'conv6'         Convolution             512 3x3x256 convolutions with stride [1  1] and
    15   'relu6'         Leaky ReLU              Leaky ReLU with scale 0.1
    16   'pool6'         Max Pooling             2x2 max pooling with stride [2  2] and padding
    17   'conv7'         Convolution             256 1x1x512 convolutions with stride [1  1] and
    18   'relu7'         Leaky ReLU              Leaky ReLU with scale 0.1
    19   'conv8'         Convolution             512 3x3x256 convolutions with stride [1  1] and
    20   'relu8'         Leaky ReLU              Leaky ReLU with scale 0.1
    21   'conv9'         Convolution             256 1x1x512 convolutions with stride [1  1] and
    22   'relu9'         Leaky ReLU              Leaky ReLU with scale 0.1
    23   'conv10'        Convolution             512 3x3x256 convolutions with stride [1  1] and
    24   'relu10'        Leaky ReLU              Leaky ReLU with scale 0.1
    25   'conv11'        Convolution             256 1x1x512 convolutions with stride [1  1] and
    26   'relu11'        Leaky ReLU              Leaky ReLU with scale 0.1
```

```
    27   'conv12'        Convolution             512 3x3x256 convolutions with stride [1  1] and
    28   'relu12'        Leaky ReLU              Leaky ReLU with scale 0.1
    29   'conv13'        Convolution             256 1x1x512 convolutions with stride [1  1] and
    30   'relu13'        Leaky ReLU              Leaky ReLU with scale 0.1
    31   'conv14'        Convolution             512 3x3x256 convolutions with stride [1  1] and
    32   'relu14'        Leaky ReLU              Leaky ReLU with scale 0.1
    33   'conv15'        Convolution             512 1x1x512 convolutions with stride [1  1] and
    34   'relu15'        Leaky ReLU              Leaky ReLU with scale 0.1
    35   'conv16'        Convolution             1024 3x3x512 convolutions with stride [1  1] and
    36   'relu16'        Leaky ReLU              Leaky ReLU with scale 0.1
    37   'pool16'        Max Pooling             2x2 max pooling with stride [2  2] and padding
    38   'conv17'        Convolution             512 1x1x1024 convolutions with stride [1  1] and
    39   'relu17'        Leaky ReLU              Leaky ReLU with scale 0.1
    40   'conv18'        Convolution             1024 3x3x512 convolutions with stride [1  1] and
    41   'relu18'        Leaky ReLU              Leaky ReLU with scale 0.1
    42   'conv19'        Convolution             512 1x1x1024 convolutions with stride [1  1] and
    43   'relu19'        Leaky ReLU              Leaky ReLU with scale 0.1
    44   'conv20'        Convolution             1024 3x3x512 convolutions with stride [1  1] and
    45   'relu20'        Leaky ReLU              Leaky ReLU with scale 0.1
    46   'conv21'        Convolution             1024 3x3x1024 convolutions with stride [1  1] an
    47   'relu21'        Leaky ReLU              Leaky ReLU with scale 0.1
    48   'conv22'        Convolution             1024 3x3x1024 convolutions with stride [2  2] an
    49   'relu22'        Leaky ReLU              Leaky ReLU with scale 0.1
    50   'conv23'        Convolution             1024 3x3x1024 convolutions with stride [1  1] an
    51   'relu23'        Leaky ReLU              Leaky ReLU with scale 0.1
    52   'conv24'        Convolution             1024 3x3x1024 convolutions with stride [1  1] an
    53   'relu24'        Leaky ReLU              Leaky ReLU with scale 0.1
    54   'fc25'          Fully Connected         4096 fully connected layer
    55   'relu25'        Leaky ReLU              Leaky ReLU with scale 0.1
    56   'fc26'          Fully Connected         539 fully connected layer
    57   'softmax'       Softmax                 softmax
    58   'classoutput'   Classification Output   crossentropyex with '1' and 538 other classes
```

The recognition network contains 14 layers including convolution, fully connected, and the classification output layers.

```
load('RecognitionNet.mat');
convnet.Layers
```

```
ans =

  14x1 Layer array with layers:

     1   'imageinput'    Image Input             48x48x3 images with 'zerocenter' normalization a
     2   'conv_1'        Convolution             100 7x7x3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     Max Pooling             2x2 max pooling with stride [2  2] and padding
     5   'conv_2'        Convolution             150 4x4x100 convolutions with stride [1  1] and
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             2x2 max pooling with stride [2  2] and padding
     8   'conv_3'        Convolution             250 4x4x150 convolutions with stride [1  1] and
     9   'maxpool_3'     Max Pooling             2x2 max pooling with stride [2  2] and padding
    10   'fc_1'          Fully Connected         300 fully connected layer
    11   'dropout'       Dropout                 90% dropout
    12   'fc_2'          Fully Connected         35 fully connected layer
    13   'softmax'       Softmax                 softmax
    14   'classoutput'   Classification Output   crossentropyex with '0' and 34 other classes
```

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 2017-2020 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` 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: To view the report, open('codegen/mex/tsdr_predict/html/report.mldat
```

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','speedLimit4
    '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},'Fo
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.

```
% Input video
v = VideoReader('stop.avi');
fps = 0;

 while hasFrame(v)
    % Take a frame
    picture = readFrame(v);
    picture = imresize(picture,[920,1632]);
    % 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

     displayDetections(picture,bboxes,clases,fps);
  end
```

Clear the static network objects that were loaded into memory.

```
clear mex;
```

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Computer Vision Using Deep Learning”

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.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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 [227 227 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.Layers
```

```
ans =

  22x1 Layer array with layers:

     1   'imageinput'    Image Input             227x227x3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution             96 5x5x3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     Max Pooling             3x3 max pooling with stride [2  2] and padding
     5   'conv_2'        Convolution             128 3x3x96 convolutions with stride [1  1] and
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             3x3 max pooling with stride [2  2] and padding
     8   'conv_3'        Convolution             384 3x3x128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     Max Pooling             3x3 max pooling with stride [2  2] and padding
    11   'conv_4'        Convolution             128 3x3x384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     Max Pooling             3x3 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
```

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 2017-2019 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` 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: To view the report, open('codegen/mex/logonet_predict/html/report.mlc
```

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', 'stel
```

```
[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), '%
    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

Pedestrian Detection

This example shows code generation for pedestrian detection application that uses deep learning. Pedestrian detection is a key issue in computer vision. Pedestrian detection has several applications in the fields of autonomous driving, surveillance, robotics, and so on.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

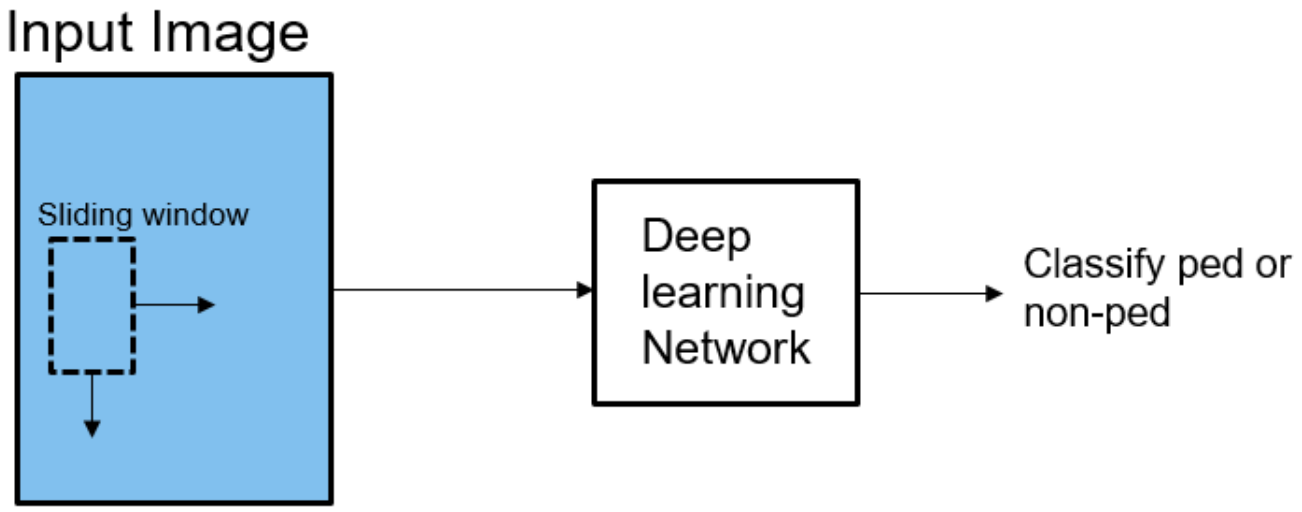
Verify GPU Environment

Use the `coder.checkGpuInstall` 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 Pedestrian Detection Network

The pedestrian detection network was trained by using images of pedestrians and non-pedestrians. This network is trained in MATLAB® by using the trainPedNet.m helper script. A sliding window approach crops patches from an image of size [64 32]. Patch dimensions are obtained from a heatmap, which represents the distribution of pedestrians in the images in the data set. It indicates the presence of pedestrians at various scales and locations in the images. In this example, patches of pedestrians close to the camera are cropped and processed. Non-Maximal Suppression (NMS) is applied on the obtained patches to merge them and detect complete pedestrians.



The pedestrian detection network contains 12 layers which include convolution, fully connected, and classification output layers.

```
load('PedNet.mat');
PedNet.Layers
```

```
ans =

  12x1 Layer array with layers:

     1   'imageinput'    Image Input                  64x32x3 images with 'zerocenter' normaliza
     2   'conv_1'        Convolution                  20 5x5x3 convolutions with stride [1  1] a
     3   'relu_1'        ReLU                         ReLU
     4   'maxpool_1'     Max Pooling                  2x2 max pooling with stride [2  2] and pad
     5   'crossnorm'     Cross Channel Normalization  cross channel normalization with 5 channe
     6   'conv_2'        Convolution                  20 5x5x20 convolutions with stride [1  1]
     7   'relu_2'        ReLU                         ReLU
     8   'maxpool_2'     Max Pooling                  2x2 max pooling with stride [2  2] and pad
     9   'fc_1'          Fully Connected              512 fully connected layer
    10   'fc_2'          Fully Connected              2 fully connected layer
    11   'softmax'       Softmax                      softmax
    12   'classoutput'   Classification Output        crossentropyex with classes 'NonPed' and
```

The `pedDetect_predict` Entry-Point Function

The pedDetect_predict.m entry-point function takes an image input and performs prediction on an image by using the deep learning network saved in the `PedNet.mat` file. The function loads the network object from the `PedNet.mat` file into a persistent variable *pednet*. Then function reuses the persistent object on subsequent calls.

```
type('pedDetect_predict.m')
```

```
function selectedBbox = pedDetect_predict(img)
%#codegen

% Copyright 2017-2019 The MathWorks, Inc.
```

```
coder.gpu.kernelfun;

persistent pednet;
if isempty(pednet)
    pednet = coder.loadDeepLearningNetwork(coder.const('PedNet.mat'),'Pedestrian_Detection');
end

[imgHt , imgWd , ~] = size(img);
VrHt = [imgHt - 30 , imgHt]; % Two bands of vertical heights are considered

% patchHt and patchWd are obtained from heat maps (heat map here refers to
% pedestrians data represented in the form of a map with different
% colors. Different colors indicate presence of pedestrians at various
% scales).
patchHt = 300;
patchWd = patchHt/3;

% PatchCount is used to estimate number of patches per image
PatchCount = ((imgWd - patchWd)/20) + 2;
maxPatchCount = PatchCount * 2;
Itmp = zeros(64 , 32 , 3 , maxPatchCount);
ltMin = zeros(maxPatchCount);
lttop = zeros(maxPatchCount);

idx = 1; % To count number of image patches obtained from sliding window
cnt = 1; % To count number of patches predicted as pedestrians

bbox = zeros(maxPatchCount , 4);
value = zeros(maxPatchCount , 1);

%% Region proposal for two bands
for VrStride = 1 : 2
    for HrStride = 1 : 20 : (imgWd - 60)  % Obtain horizontal patches with stride 20.
        ltMin(idx) = HrStride + 1;
        rtMax = min(ltMin(idx) + patchWd , imgWd);
        lttop(idx) = (VrHt(VrStride) - patchHt);
        It = img(lttop(idx): VrHt(VrStride) , ltMin(idx) : rtMax , :);
        Itmp(:,:,:,idx) = imresize(It,[64,32]);
        idx = idx + 1;
    end
end

for j = 1 : size (Itmp,4)
    score = pednet.predict(Itmp(:,:,:,j)); % Classify ROI
    % accuracy of detected box should be greater than 0.90
    if (score(1,2) > 0.80)
        bbox(cnt,:) = [ltMin(j),lttop(j), patchWd , patchHt];
        value(cnt,:) = score(1,2);
        cnt = cnt + 1;
    end

end

%% NMS to merge similar boxes
if ~isempty(bbox)
    [selectedBbox,~] = selectStrongestBbox(bbox(1:cnt-1,:),...
        value(1:cnt-1,:),'OverlapThreshold',0.002);
```

```
end
```

Generate CUDA MEX for the `pedDetect_predict` Function

Create a GPU Configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` 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 size of the input image. This value corresponds to the input layer size of pedestrian detection network.

```
% Load an input image.
im = imread('test.jpg');
im = imresize(im,[480,640]);

cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg pedDetect_predict -args {im} -report
```

```
Code generation successful: To view the report, open('codegen/mex/pedDetect_predict/html/report.m
```

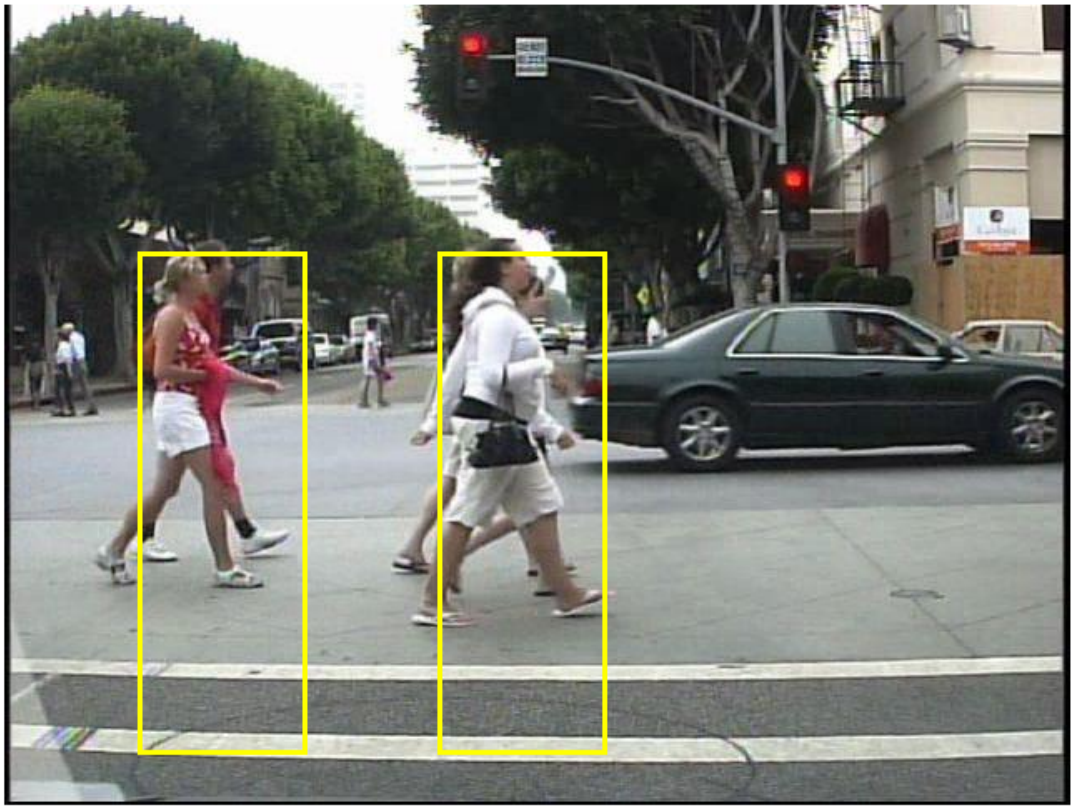
Run Generated MEX

Call `pednet_predict_mex` on the input image.

```
imshow(im);
ped_bboxes = pedDetect_predict_mex(im);
```

Display the final predictions.

```
outputImage = insertShape(im,'Rectangle',ped_bboxes,'LineWidth',3);
imshow(outputImage);
```



Classification on Video

The included helper file pedDetect_predict.m grabs frames from a video, performs prediction, and displays the classification results on each of the captured video frames.

```
v = VideoReader('LiveData.avi');
fps = 0;
while hasFrame(v)
   % Read frames from video
   im = readFrame(v);
   im = imresize(im,[480,640]);

   % Call MEX function for pednet prediction
   tic;
   ped_bboxes = pedDetect_predict_mex(im);
   newt = toc;

   % fps
   fps = .9*fps + .1*(1/newt);

   % display
   outputImage = insertShape(im,'Rectangle',ped_bboxes,'LineWidth',3);
   imshow(outputImage)
   pause(0.2)
end
```

Clear the static network object that was loaded in memory.

```
clear mex;
```

See Also

Related Examples

- “Deep Learning in MATLAB” on page 1-2
- “Computer Vision Using Deep Learning”

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.

Prerequisites

- CUDA enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see Third-party Products. For setting up the environment variables, see Environment Variables.
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

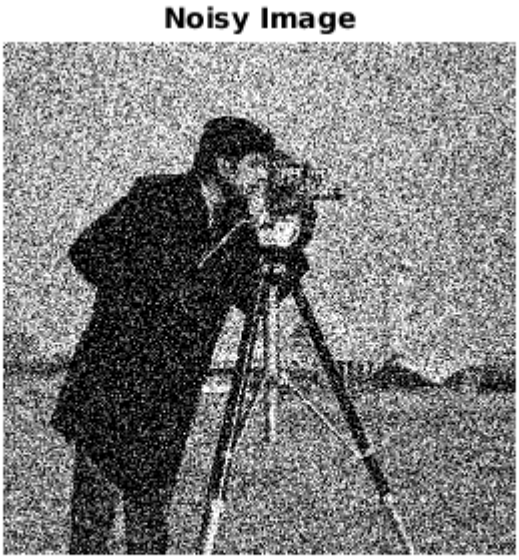
Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
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 2018-2019 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` 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: To view the report, open('codegen/mex/denoisenet_predict/html/report
```

Run Generated MEX

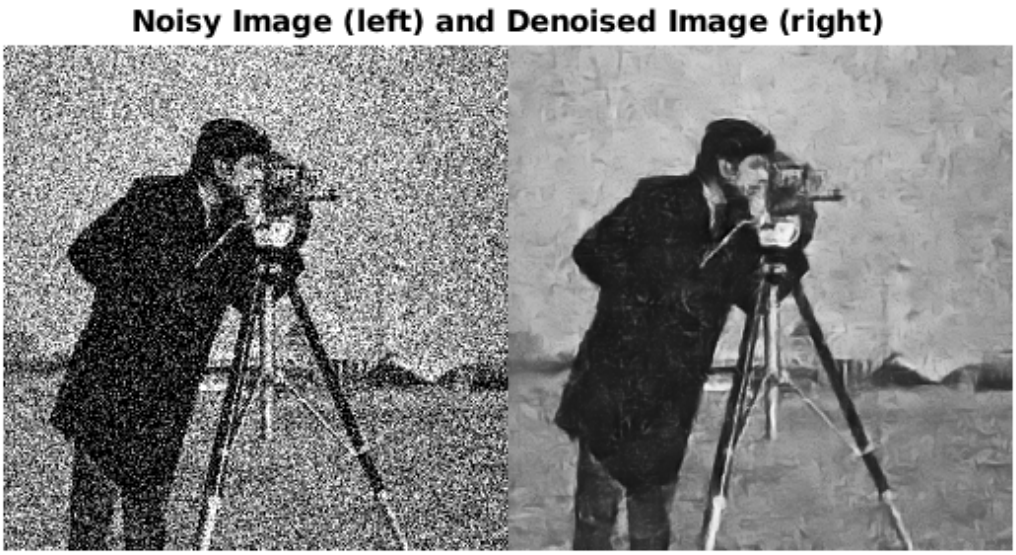
The DnCNN [1] is trained on input images having an input range [0,1]. Call the im2single function on `noisyI` to rescale the values from [0,255] to [0,1].

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
- “Image Processing Using Deep Learning”

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.

Prerequisites

- CUDA® enabled NVIDIA GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.
- Deep Learning Toolbox Model for VGG-16 Network support package. To install this support package, see `vgg16`.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).

Verify GPU Environment

Use the `coder.checkGpuInstall` 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` 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

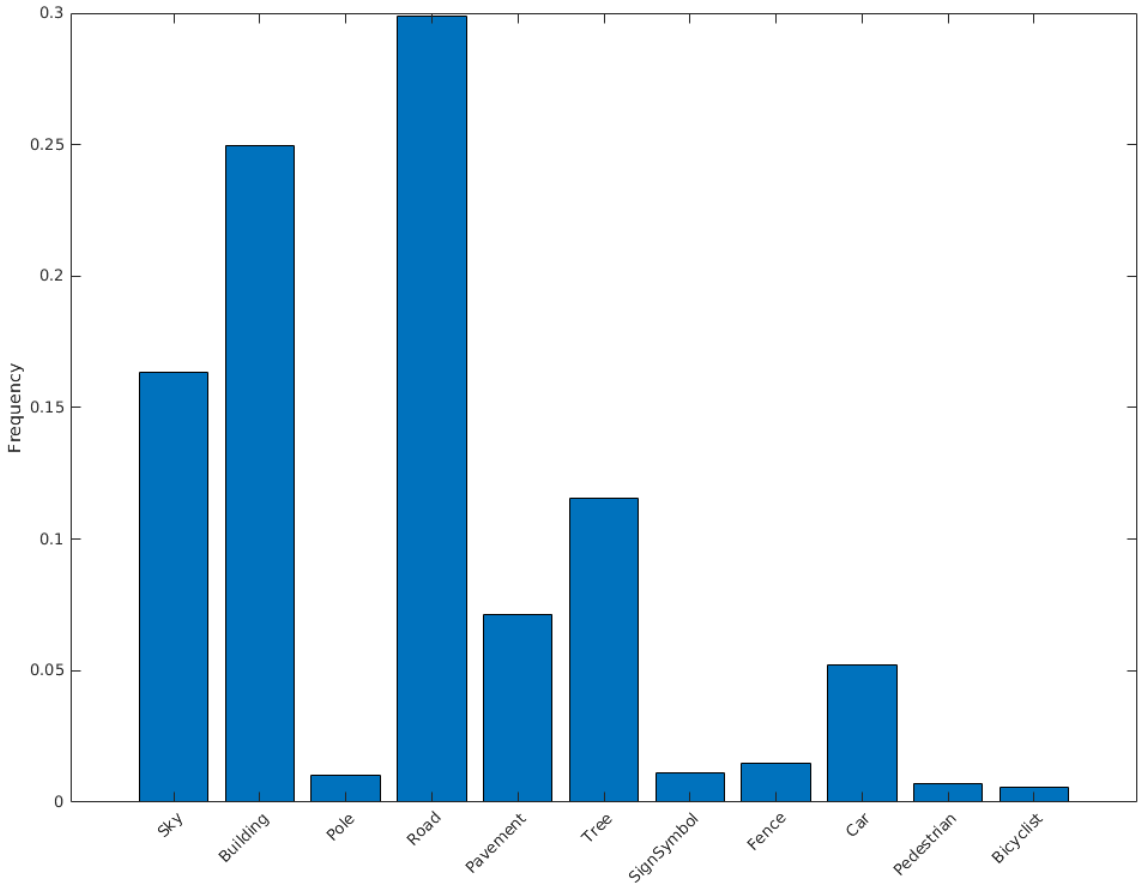
To see the distribution of class labels in the CamVid dataset, use `countEachLabel`. This function counts the number of pixels by class label.

```
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` 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` 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`.

```
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 provides more examples to the network because it helps improve the accuracy of the network. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation. Use the `imageDataAugmenter` function to specify these data augmentation parameters.

```
augmenter = imageDataAugmenter('RandXReflection',true,...
    'RandXTranslation',[-10 10],'RandYTranslation',[-10 10]);
```

The `imageDataAugmenter` function supports several other types of data augmentation. Choosing among them requires empirical analysis and is another level of hyperparameter tuning.

Start Training

Combine the training data and data augmentation selections by using the `pixelLabelImageDatastore` function. The `pixelLabelImageDatastore` function reads batches of training data, applies data augmentation, and sends the augmented data to the training algorithm.

```
pximds = pixelLabelImageDatastore(imdsTrain,pxdsTrain, ...
    'DataAugmentation',augmenter);
```

If the `doTraining` flag is true, start the training by using the `trainNetwork` function.

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.

```
if doTraining
    [net, info] = trainNetwork(pximds,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` 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 [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;
```

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 by Using 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.

Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.2 or higher.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-party Products” (GPU Coder). For setting up the environment variables, see “Setting Up the Prerequisite Products” (GPU Coder).
- GPU Coder™ Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

Verify GPU Environment

Use the `coder.checkGpuInstall` 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 patchewise on patches of size PATCHSIZE.
%
% Copyright 2018-2019 The MathWorks, Inc.
function out = segmentImageUnet(im, patchSize)

%#codegen

coder.gpu.kernelfun;

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);

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'))
    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
```

```
  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` 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');
codegen -config cfg segmentImageUnet -args {ones(size(test_data),'uint16'),coder.Constant([1024 1
```

```
Code generation successful: To view the report, open('codegen/mex/segmentImageUnet/html/report.m
```

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]);
```

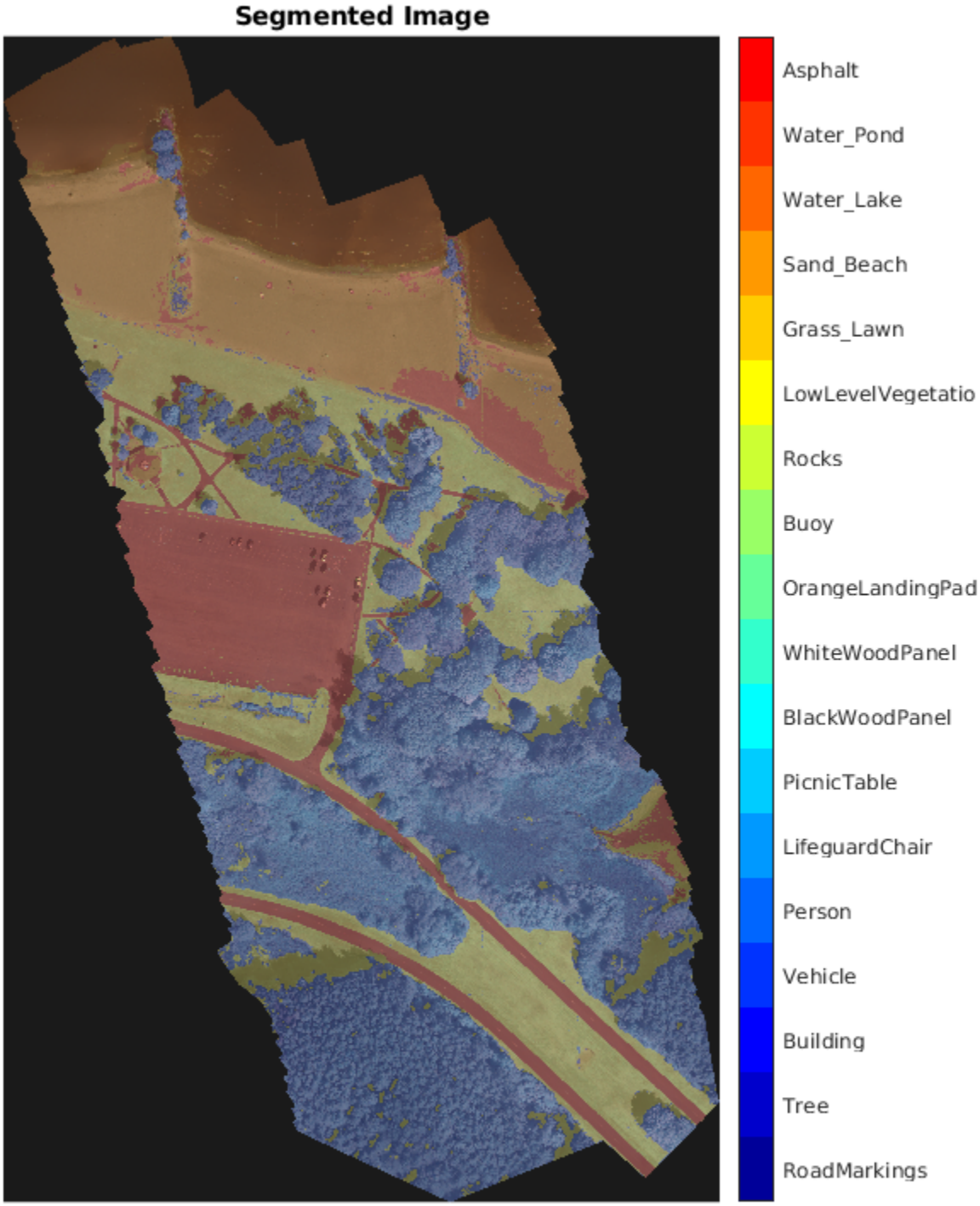
Display U-Net Segmented test_data

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
figure
imshow(B)

N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,'TickLabelInterpreter','no
colormap(cmap)
title('Segmented Image');
```


Segmented Image
Asphalt
Water_Pond
Water_Lake
Sand_Beach
Grass_Lawn
LowLevelVegetatio
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™

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/ar
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:tar
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

Code Generation for Deep Learning on Raspberry Pi

This example shows how to generate and deploy code for prediction on a Raspberry Pi™ by using `codegen` with the MATLAB Support Package for Raspberry Pi Hardware.

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.

Prerequisites

- ARM processor that supports the NEON extension
- ARM Compute Library (on the target ARM hardware)
- Open Source Computer Vision Library
- Environment variables for the compilers and libraries
- MATLAB® Coder™
- The support package MATLAB Coder Interface for Deep Learning
- Deep Learning Toolbox™
- MATLAB Support Package for Raspberry Pi Hardware

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.

The `squeezenet_raspi_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_arm_predict` function loads the SqueezeNet network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

So that the generated executable program links against the OpenCV libraries, the `squeezenet_arm_predict` function specifies linker options by using `coder.updateBuildInfo`.

```
type squeezenet_raspi_predict
```

```
% Copyright 2018 The MathWorks, Inc.

function out = squeezenet_raspi_predict(in)
%#codegen

% A persistent object mynet is used to load the DAGNetwork 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 net;
opencv_linkflags = ``pkg-config --cflags --libs opencv``;
coder.updateBuildInfo('addLinkFlags',opencv_linkflags);
if isempty(net)
```

```
    net = coder.loadDeepLearningNetwork('squeezenet', 'squeezenet');
end

out = net.predict(in);

end
```

Set Up a 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 a Configuration Object for Deep Learning Code Generation with ARM Compute Library

Create a `coder.ARMNEONConfig` object. Specify the version of the ARM Compute library that is on the Raspberry Pi. Specify the architecture of the Raspberry Pi.

```
dlcfg = coder.DeepLearningConfig('arm-compute');
dlcfg.ArmArchitecture = 'armv7';
dlcfg.ArmComputeVersion = '19.05';
```

Attach the Deep Learning Configuration Object to the Code Generation 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;
```

Specify the build folder on the Raspberry Pi

```
buildDir = '~/remoteBuildDir';
cfg.Hardware.BuildDir = buildDir;
```

Provide a C++ Main File

The C++ main file reads the input image, runs prediction on the image, and displays the classification labels on the image.

Specify the main file in the code generation configuration object.

```
cfg.CustomSource = 'main_squeezenet_raspi.cpp';
```

Generate the Executable Program on the Raspberry Pi

Use `codegen` to generate the C++ code. When you use `codegen` with the MATLAB Support Package for Raspberry PI Hardware, the executable is built on the Raspberry Pi.

Make sure that you 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 squeezenet_raspi_predict -args {ones(227, 227, 3,'single')} -report
```

Fetch the Generated Executable Directory

To test the generated code on the Raspberry Pi, copy the input image 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`. Assuming that the binary is found in only one directory, enter:

```
applicationDirPaths = raspi.utils.getRemoteBuildDirectory('applicationName','squeezenet_raspi_pre
targetDirPath = applicationDirPaths{1}.directory;
```

Copy Example 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.

```
r.putFile('synsetWords_squeezenet_raspi.txt', targetDirPath);
r.putFile('coffeemug.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 = 'squeezenet_raspi_predict.elf';
argsforexe = ' coffeemug.png '; % Provide the input image;
command = ['cd ' targetDirPath ';./' exeName argsforexe];
output = system(r,command)
```

Get the Prediction Scores

```
outputfile = [targetDirPath, '/output.txt'];
r.getFile(outputfile);
```

Map the Prediction Scores to Labels

Map the top five prediction scores to corresponding labels in the trained network.

```
net = squeezenet;
ClassNames = net.Layers(end).ClassNames;
```

Read the classification.

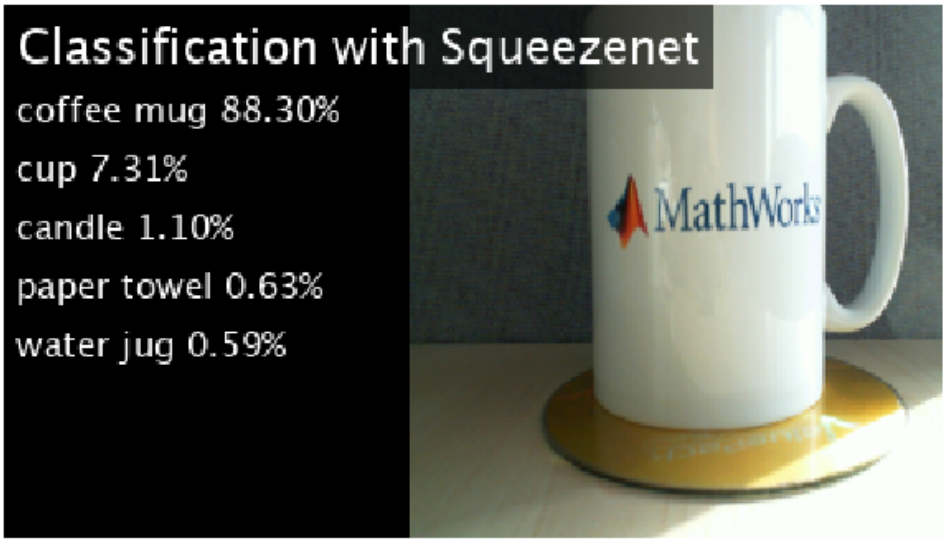
```
fid = fopen('output.txt') ;
S = textscan(fid,'%s');
fclose(fid) ;
S = S{1} ;
predict_scores = cellfun(@(x)str2double(x), S);
```

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('coffeemug.png');
im = imresize(im, [227 227]);
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:,:,k);
end
scol = 1;
srow = 1;
outputImage = insertText(outputImage, [scol, srow], 'Classification with Squeezenet', 'TextColor
srow = srow + 30;
for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k},' ',num2str(scores(k), '%
    srow = srow + 25;
end
imshow(outputImage);
```


Classification with Squeezenet
coffee mug 88.30%
cup 7.31%
candle 1.10%
paper towel 0.63%
water jug 0.59%
MathWorks

Deep Learning Prediction with ARM Compute Using cnncodegen

This example shows how to use `cnncodegen` 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

For more information, 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 =

  22x1 Layer array with layers:

     1   'imageinput'    Image Input             227x227x3 images with 'zerocenter' normalizatio
     2   'conv_1'        Convolution             96 5x5x3 convolutions with stride [1  1] and pa
     3   'relu_1'        ReLU                    ReLU
     4   'maxpool_1'     Max Pooling             3x3 max pooling with stride [2  2] and padding
     5   'conv_2'        Convolution             128 3x3x96 convolutions with stride [1  1] and
     6   'relu_2'        ReLU                    ReLU
     7   'maxpool_2'     Max Pooling             3x3 max pooling with stride [2  2] and padding
     8   'conv_3'        Convolution             384 3x3x128 convolutions with stride [1  1] and
     9   'relu_3'        ReLU                    ReLU
    10   'maxpool_3'     Max Pooling             3x3 max pooling with stride [2  2] and padding
    11   'conv_4'        Convolution             128 3x3x384 convolutions with stride [2  2] and
    12   'relu_4'        ReLU                    ReLU
    13   'maxpool_4'     Max Pooling             3x3 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).

Generate Code

For deep learning on ARM targets, you generate code on the host development computer. Then, you move the generated code to the ARM platform where you build and run the executable program. The target platform must support the Neon instruction set architecture (ISA). Raspberry Pi3, Firefly, HiKey are some of the target platforms on which the generated code can be executed.

Call `cnncodegen`, specifying that the target library is the ARM Compute Library. Specify the version of the library and the architecture of the target ARM processor. The `ARMArchitecture` parameter is required.

```
cnncodegen(net,'targetlib','arm-compute','targetparams', struct('ArmComputeVersion','19.05','ARM
```

Copy the Generated Files to the Target

Move the codegen folder and other required files from the host development computer to the target platform by using your preferred `scp/ssh` client.

For example on the Linux platform, to transfer the files to the Raspberry Pi, use the scp command with the format:

```
system('sshpass -p [password] scp (sourcefile) [username]@[targetname]:~/');
```

```
system('sshpass -p password scp main_arm_logo.cpp username@targetname:~/');
system('sshpass -p password scp coderdemo_google.png username@targetname:~/');
system('sshpass -p password scp makefile_arm_logo.mk username@targetname:~/');
system('sshpass -p password scp synsetWordsLogoDet.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');
```

Note: PSCP utilities must be either on your PATH or in your current directory.

Build and Execute

To build the library on the target platform, use the generated makefile `cnnbuild_rtw.mk`.

For example, to build the library on the Raspberry Pi from the Linux platform:

```
system('sshpass -p password ssh username@targetname "make -C /home/username/codegen -f cnnbuild_
```

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_arm_logo.mk`

For example, on the Raspberry Pi:

```
make -C /home/pi arm_neon -f makefile_arm_logo.mk
```

Run the executable with an input image file.

```
./logo_recognition_exe coderdemo_google.png
```

The top five predictions for the input image file are:

```
CNNCodegen Top 5 Predictions:
------------------------------
99.992% google
0.003% corona
0.003% singha
0.001% esso
0.000% fedex
```

Deep Learning Prediction with Intel MKL-DNN

This example shows how to use `codegen` to generate code for an image classification application that uses deep learning on Intel® processors. The generated code takes advantage of the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN). First, the example generates a MEX function that runs prediction by using the ResNet-50 image classification network. Then, the example builds a static library and compiles it with a main file that runs prediction using the ResNet-50 image classification network.

Prerequisites

- Xeon processor with support for Intel Advanced Vector Extensions 2 (Intel AVX2) instructions
- Intel Math Kernel Library for Deep Neural Networks (MKL-DNN)
- Open Source Computer Vision Library (OpenCV) v3.1
- Environment variables for Intel MKL-DNN and OpenCV
- MATLAB® Coder™, for C++ code generation.
- The support package MATLAB Coder Interface for Deep Learning.
- Deep Learning Toolbox™, for using the `DAGNetwork` object
- The Support package Deep Learning Toolbox Model for ResNet-50 Network support package, for using the pretrained ResNet network.

For more information, 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.

`resnet_predict` Function

This example uses the DAG network ResNet-50 to show image classification with MKL-DNN. A pretrained ResNet-50 model for MATLAB is available in the support package Deep Learning Toolbox Model for ResNet-50 Network. To download and install the support package, use the Add-On Explorer. See “Get and Manage Add-Ons” (MATLAB).

The `resnet_predict` function loads the ResNet-50 network into a persistent network object. On subsequent calls to the function, the persistent object is reused.

```
type resnet_predict
```

```
% Copyright 2018 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 Code for the `resnet_predict` Function

To generate a MEX function from the `resnet_predict.m` function, use `codegen` with a deep learning configuration object created for the MKL-DNN library. Attach the deep learning configuration object to the MEX code generation configuration object that you pass to `codegen`.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg resnet_predict -args {ones(224,224,3,'single')} -report
```

```
Code generation successful: To view the report, open('codegen\mex\resnet_predict\html\report.mlda
```

Call `predict` on a Test Image

```
im = imread('peppers.png');
im = imresize(im, [224,224]);
imshow(im);
predict_scores = resnet_predict_mex(single(im));
```

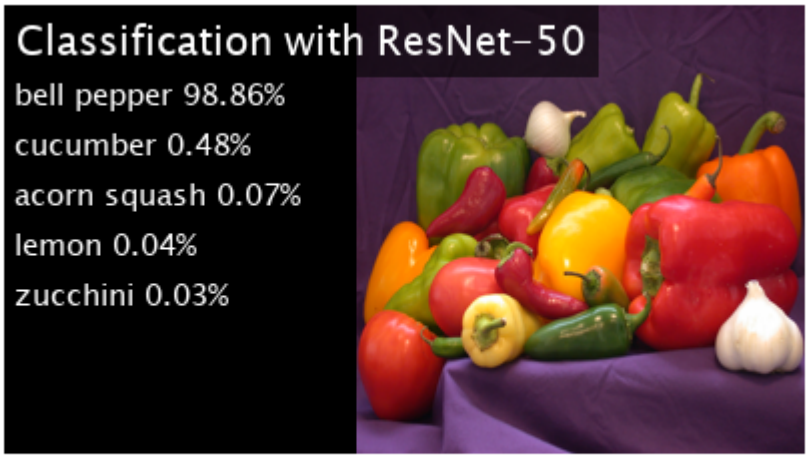
Map the top five prediction scores to words in the synset dictionary.

```
fid = fopen('synsetWords.txt');
synsetOut = textscan(fid,'%s', 'delimiter', '\n');
synsetOut = synsetOut{1};
fclose(fid);
[val,indx] = sort(predict_scores, 'descend');
```

```
scores = val(1:5)*100;
top5labels = synsetOut(indx(1:5));
```

Display the top five classification labels on the image.

```
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 ResNet-50', 'TextColor'
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 static network object from memory.

```
clear mex;
```

Generate a Static Library for the `resnet_predict` Function

To generate a static library from the `resnet_predict.m` function, use `codegen` with a deep learning configuration object created for the MKL-DNN library. Attach the deep learning configuration object to the code generation configuration object that you pass to `codegen`.

```
cfg = coder.config('lib');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -config cfg resnet_predict -args {ones(224,224,3,'single')} -report
```

```
%
codegendir = fullfile(pwd, 'codegen', 'lib', 'resnet_predict');

Code generation successful: To view the report, open('codegen\lib\resnet_predict\html\report.mlda
```

`main_resnet.cpp` File

The main file is used to generate an executable from the static library created by the `codegen` command. The main file reads the input image, runs prediction on the image, and displays the classification labels on the image.

```
type main_resnet.cpp
```

```
/* Copyright 2018 The MathWorks, Inc. */

#include "resnet_predict.h"

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include "opencv2/opencv.hpp"
using namespace cv;

int readData(void* inputBuffer, char* inputImage) {

    Mat inpImage, intermImage;
    inpImage = imread(inputImage, 1);
    Size size(224, 224);
    resize(inpImage, intermImage, size);
    if (!intermImage.data) {
        printf(" No image data \n ");
        exit(1);
    }
    float* input = (float*)inputBuffer;

    for (int j = 0; j < 224 * 224; j++) {
        // BGR to RGB
        input[2 * 224 * 224 + j] = (float)(intermImage.data[j * 3 + 0]);
        input[1 * 224 * 224 + j] = (float)(intermImage.data[j * 3 + 1]);
        input[0 * 224 * 224 + j] = (float)(intermImage.data[j * 3 + 2]);
    }
    return 1;
}

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) || defined(_WIN64)

int cmpfunc(void* r, const void* a, const void* b) {
    float x = ((float*)r)[*(int*)b] - ((float*)r)[*(int*)a];
    return (x > 0 ? ceil(x) : floor(x));
}
#else

int cmpfunc(const void* a, const void* b, void* r) {
    float x = ((float*)r)[*(int*)b] - ((float*)r)[*(int*)a];
    return (x > 0 ? ceil(x) : floor(x));
}
```

```
#endif

void top(float* r, int* top5) {
    int t[1000];
    for (int i = 0; i < 1000; i++) {
        t[i] = i;
    }
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) || defined(_WIN64)
    qsort_s(t, 1000, sizeof(int), cmpfunc, r);
#else
    qsort_r(t, 1000, sizeof(int), cmpfunc, r);
#endif
    top5[0] = t[0];
    top5[1] = t[1];
    top5[2] = t[2];
    top5[3] = t[3];
    top5[4] = t[4];
    return;
}


int prepareSynset(char synsets[1000][100]) {
    FILE* fp1 = fopen("synsetWords.txt", "r");
    if (fp1 == 0) {
        return -1;
    }

    for (int i = 0; i < 1000; i++) {
        if (fgets(synsets[i], 100, fp1) != NULL)
            ;
        strtok(synsets[i], "\n");
    }
    fclose(fp1);
    return 0;
}

void writeData(float* output, char synsetWords[1000][100], Mat &frame) {
    int top5[5], j;

    top(output, top5);

    copyMakeBorder(frame, frame, 0, 0, 400, 0, BORDER_CONSTANT, CV_RGB(0,0,0));
    char strbuf[50];
    sprintf(strbuf, "%4.1f%% %s", output[top5[0]]*100, synsetWords[top5[0]]);
    putText(frame, strbuf, cvPoint(30,80), CV_FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(220,220,220), 1);
    sprintf(strbuf, "%4.1f%% %s", output[top5[1]]*100, synsetWords[top5[1]]);
    putText(frame, strbuf, cvPoint(30,130), CV_FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(220,220,220), 1)
    sprintf(strbuf, "%4.1f%% %s", output[top5[2]]*100, synsetWords[top5[2]]);
    putText(frame, strbuf, cvPoint(30,180), CV_FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(220,220,220), 1)
    sprintf(strbuf, "%4.1f%% %s", output[top5[3]]*100, synsetWords[top5[3]]);
    putText(frame, strbuf, cvPoint(30,230), CV_FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(220,220,220), 1)
    sprintf(strbuf, "%4.1f%% %s", output[top5[4]]*100, synsetWords[top5[4]]);
    putText(frame, strbuf, cvPoint(30,280), CV_FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(220,220,220), 1)

}

// Main function
```

```
int main(int argc, char* argv[]) {
    int n = 1;
    char synsetWords[1000][100];

    namedWindow("Classification with ResNet-50",CV_WINDOW_NORMAL);
    resizeWindow("Classification with ResNet-50",440,224);

    Mat im;
    im = imread(argv[1], 1);

    float* ipfBuffer = (float*)calloc(sizeof(float), 224*224*3);

    float* opBuffer = (float*)calloc(sizeof(float), 1000);
    if (argc != 2) {
        printf("Input image missing \nSample Usage-./resnet_exe image.png\n");
        exit(1);
    }
    if (prepareSynset(synsetWords) == -1) {
        printf("ERROR: Unable to find synsetWords.txt\n");
        return -1;
    }

    //read input imaget to the ipfBuffer
    readData(ipfBuffer, argv[1]);

    //run prediction on image stored in ipfBuffer
    resnet_predict(ipfBuffer, opBuffer);

    //write predictions on input image
    writeData(opBuffer, synsetWords, im);

    //show predictions on input image
    imshow("Classification with ResNet-50", im);
    waitKey(5000);
    destroyWindow("Classification with ResNet-50");
    return 0;
}
```

Build and Run the Executable

Build the executable based on the target platform. On a Windows platform, this example uses Microsoft® Visual Studio® 2017 for C++.

```
if ispc
    setenv('MATLAB_ROOT', matlabroot);
    system('make_mkldnn_win17.bat');
    system('resnet.exe peppers.png');
else
    setenv('MATLAB_ROOT', matlabroot);
    system('make -f Makefile_mkldnn_linux.mk');
    system('./resnet_exe peppers.png');
end
```

The results from the MEX function might not match the results from the generated static library function due to differences in the version of the library that is used to read the input image file. The image that is passed to the MEX function is read using the version that MATLAB ships. The image that is passed to the static library function is read using the version that OpenCV uses.

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® and Windows® 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();
```

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 for MATLAB
% execution
if coder.target('MATLAB')
    labels = cellstr(labels);
end

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
```

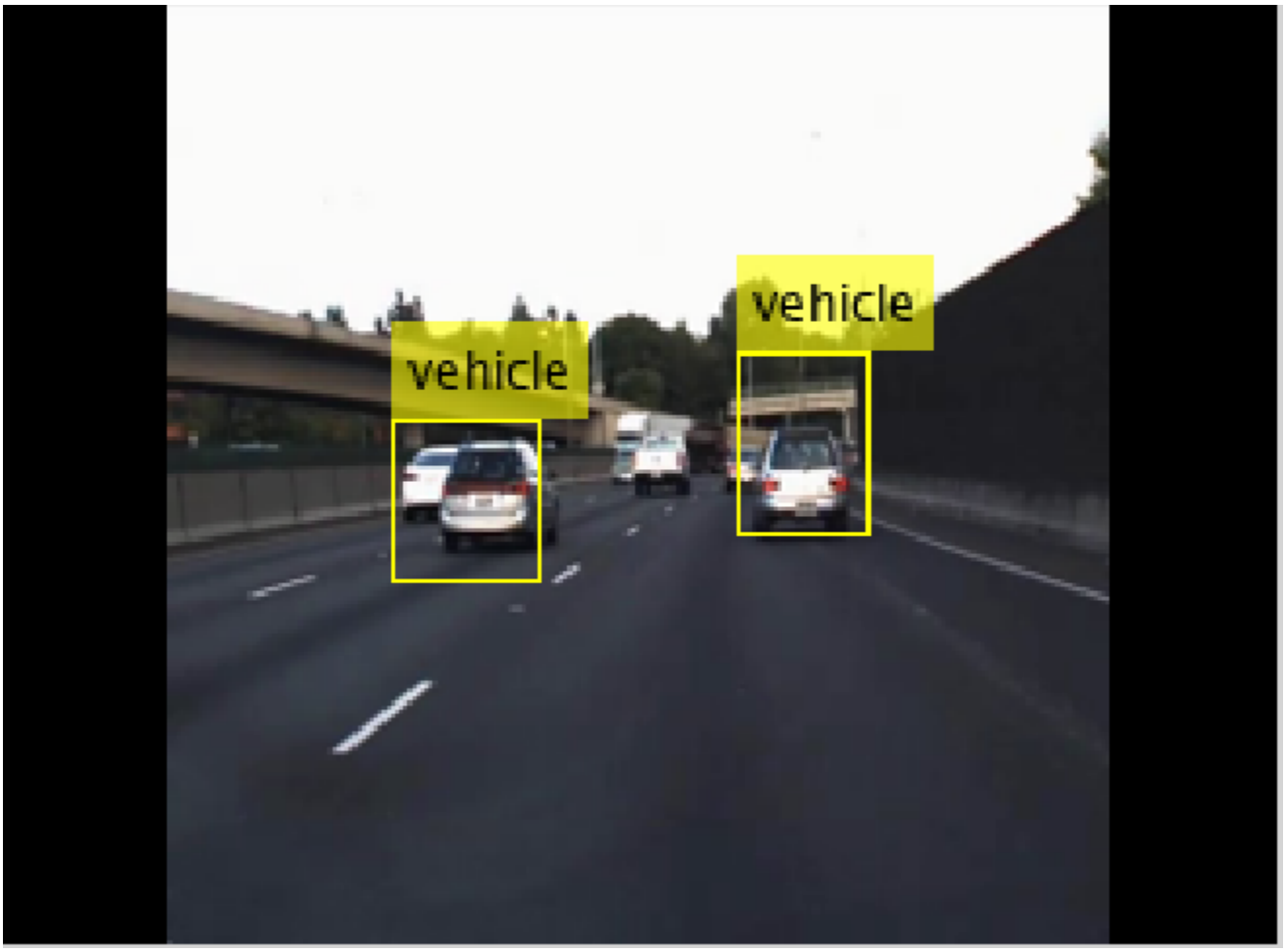
Run the Generated MEX Function on Example Input

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 Prediction with Intel MKL-DNN” (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

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

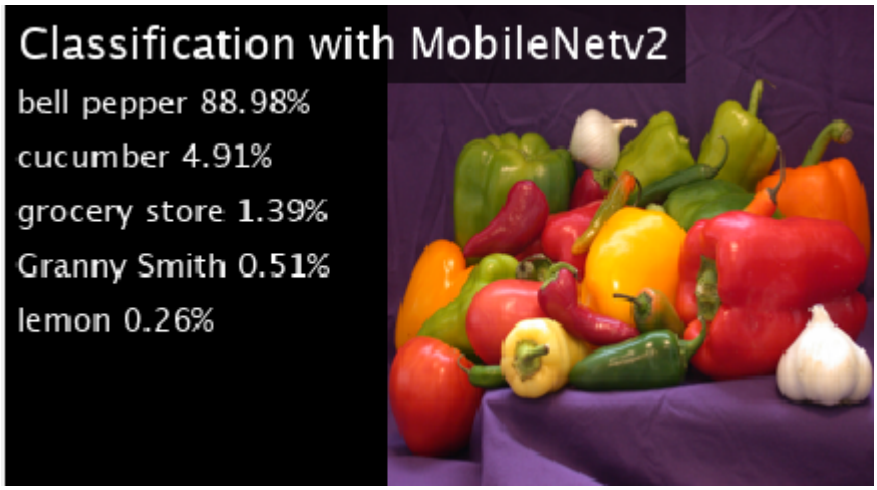
```
outputfile = [targetDirPath, '/output.txt'];
r.getFile(outputfile);
```

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)

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 18-13
3. Configure the network — “Configure Shallow Neural Network Inputs and Outputs” on page 18-16
4. Initialize the weights and biases
5. Train the network — “Neural Network Training Concepts” on page 18-22
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 19-2. 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 18-3
- “Neuron Model” on page 18-4
- “Neural Network Architectures” on page 18-8
- “Understanding Shallow Network Data Structures” on page 18-18

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

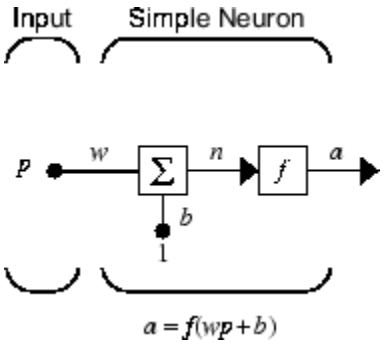
- “Workflow for Neural Network Design” on page 18-2

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 22-2 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 18-5. (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.

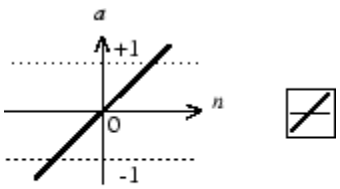
All the neurons in the Deep Learning Toolbox software have provision for a bias, and a bias is used in many of the examples and is assumed in most of this toolbox. However, you can omit a bias in a neuron if you want.

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.

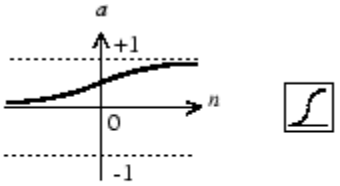


$a = purelin(n)$

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 19-2.

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

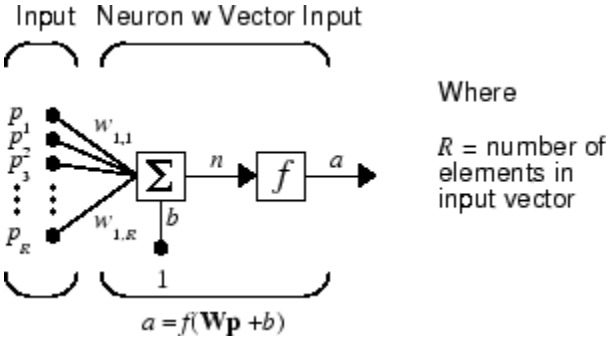
The simple neuron can be extended to handle inputs that are vectors. A neuron with a single *R*-element input vector is shown below. Here the individual input elements

$$p_1, p_2, \ldots p_R$$

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 22-2.)



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 22-2.) The net input n is the argument of the transfer function f.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

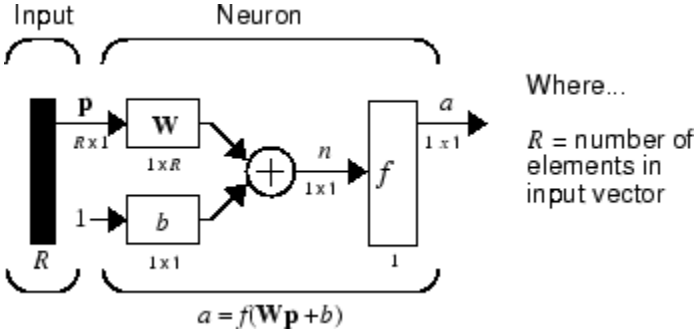
This expression can, of course, be written in MATLAB code as

```
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

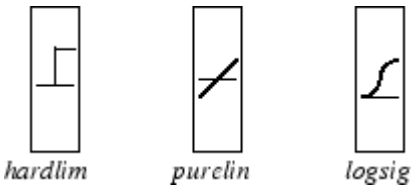
previous sentence, is used when referring to the *size* of a vector.) Thus, **p** is a vector of *R* input elements. These inputs postmultiply the single-row, *R*-column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias *b*. The net input to the transfer 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 18-4, 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 18-5, 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 18-8
- “Workflow for Neural Network Design” on page 18-2

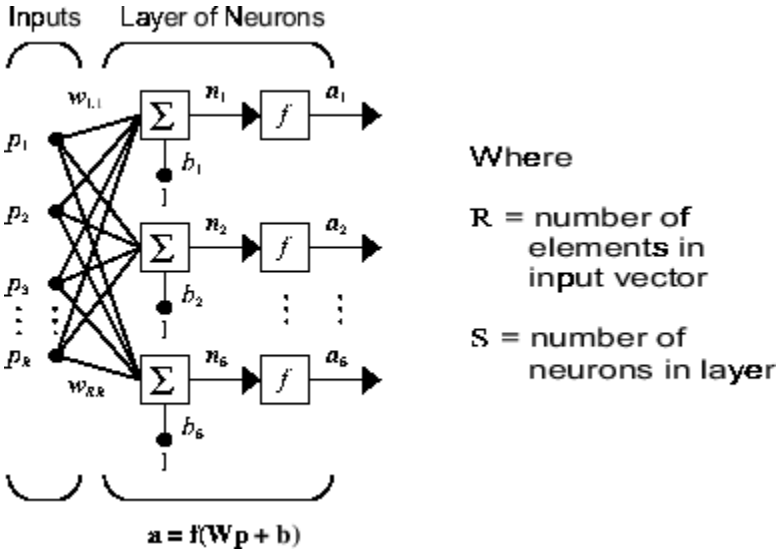
Neural Network Architectures

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.

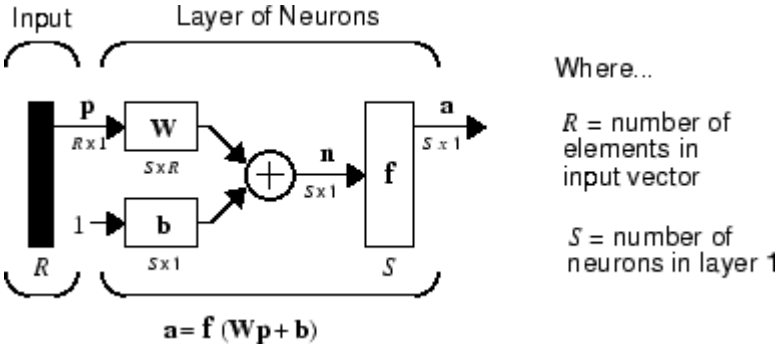
You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

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.

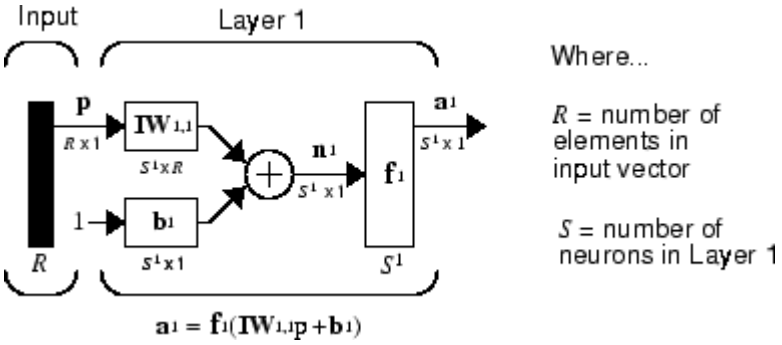


Here **p** is an R-length input vector, **W** is an $S \times R$ matrix, **a** and **b** are S-length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function blocks.

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.

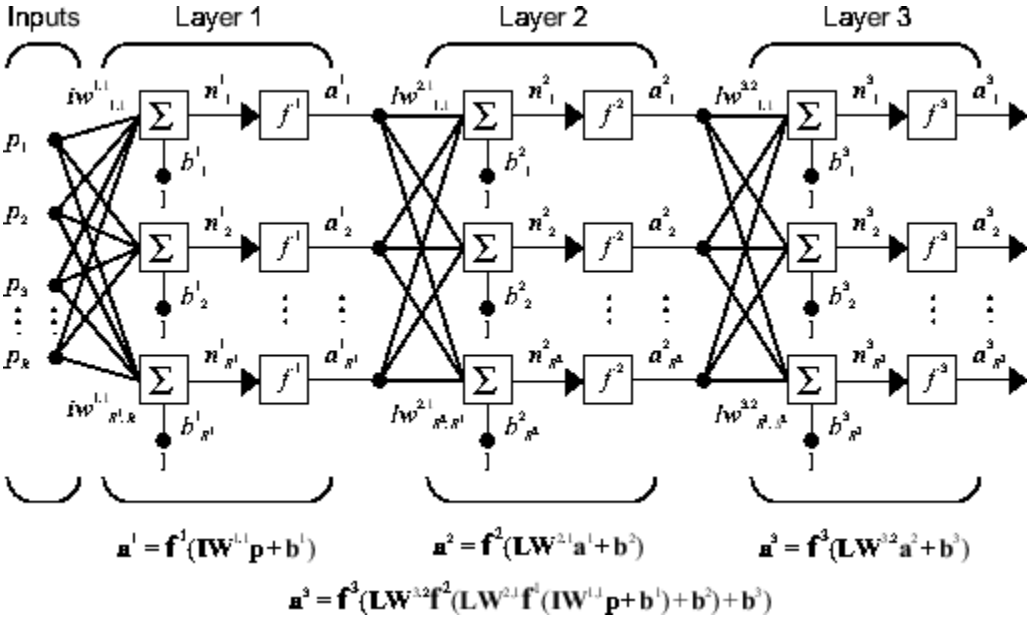


As you can see, the weight matrix connected to the input vector **p** is labeled as an input weight matrix ($\mathbf{IW}^{1,1}$) having a source 1 (second index) and a destination 1 (first index). Elements of layer 1, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

“Multiple Layers of Neurons” on page 18-10 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.



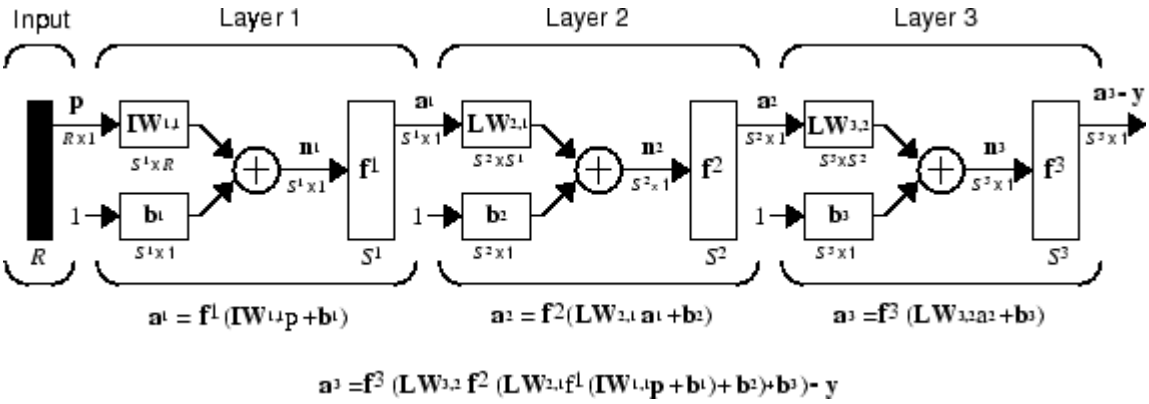
The network shown above has R^1 inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the bias for each neuron.

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 19-2.

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 19-7.

See Also

More About

- “Neuron Model” on page 18-4

- “Workflow for Neural Network Design” on page 18-2

Create Neural Network Object

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 18-2.

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 18-2.

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 19-2 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 19-2.

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 18-18.

Understanding Shallow Network Data Structures

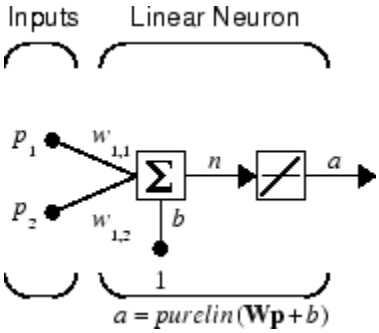
In this section...

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];
```

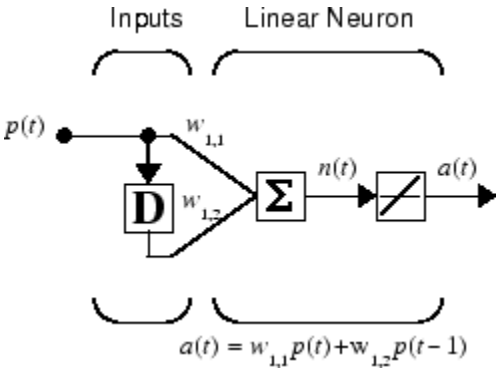
You can now simulate the network:

```
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 18-19, 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:

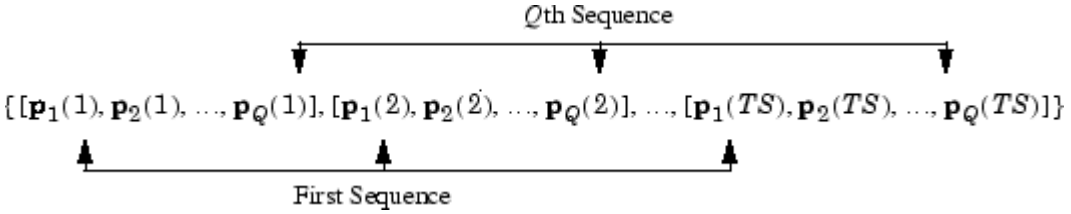
```
A = net(P);
```

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 18-18, 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 18-22.

See also “Configure Shallow Neural Network Inputs and Outputs” on page 18-16.

Neural Network Training Concepts

In this section...

This topic is part of the design workflow described in “Workflow for Neural Network Design” on page 18-2.

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 18-18 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. Use the `nntraintool` function to manually open and close the training window.

```
nntraintool
nntraintool('close')
```

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 20-10. 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 19-3
- “Prepare Data for Multilayer Shallow Neural Networks” on page 19-6
- “Create, Configure, and Initialize Multilayer Shallow Neural Networks” on page 19-11
- “Train and Apply Multilayer Shallow Neural Networks” on page 19-13
- “Analyze Shallow Neural Network Performance After Training” on page 19-18
- “Use the Network” on page 19-17
- “Limitations and Cautions” on page 19-22

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 19-7
- “Divide Data for Optimal Neural Network Training” on page 19-9
- “Neural Networks with Parallel and GPU Computing” on page 25-2

For time series, dynamic modeling, and prediction, see this section:

- “How Dynamic Neural Networks Work” on page 20-3

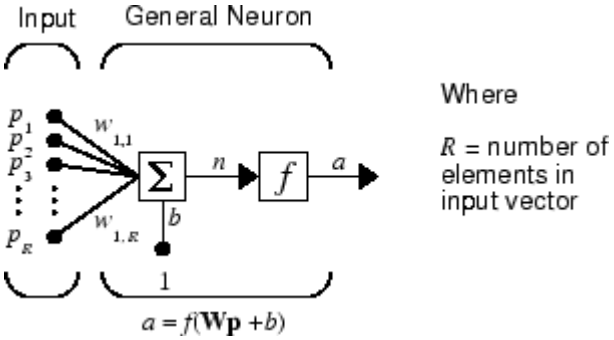
Multilayer Shallow Neural Network Architecture

In this section...

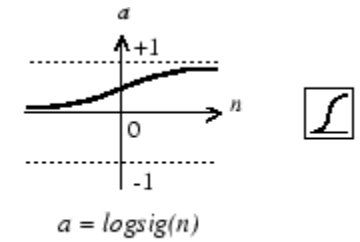
This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 19-2.

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.



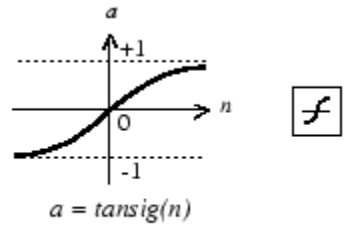
Multilayer networks often use the log-sigmoid transfer function `logsig`.



Log-Sigmoid Transfer Function

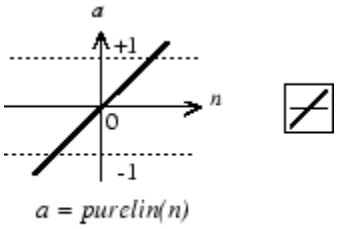
The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

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.

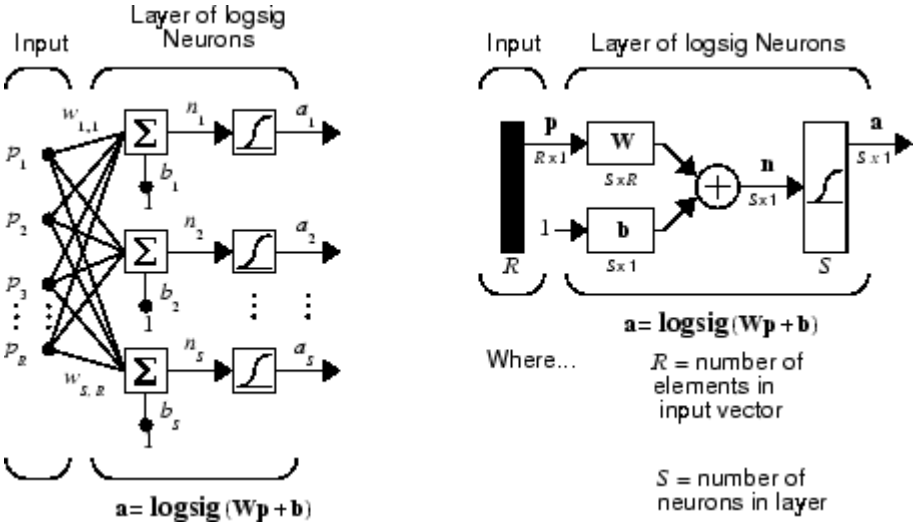


Linear Transfer Function

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

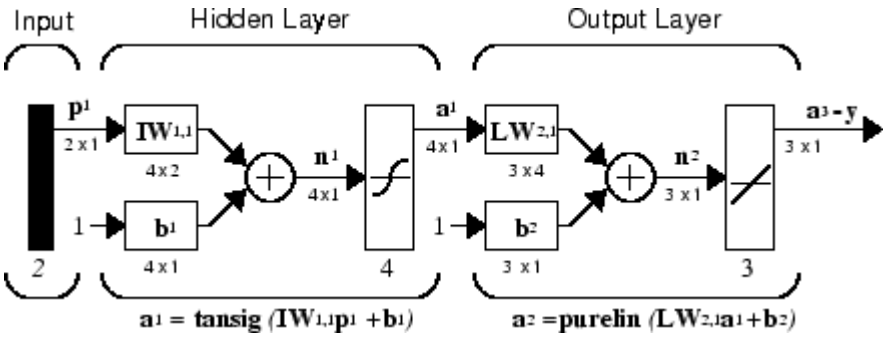
A single-layer network of S `logsig` neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



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 16-8.

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 19-2.

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

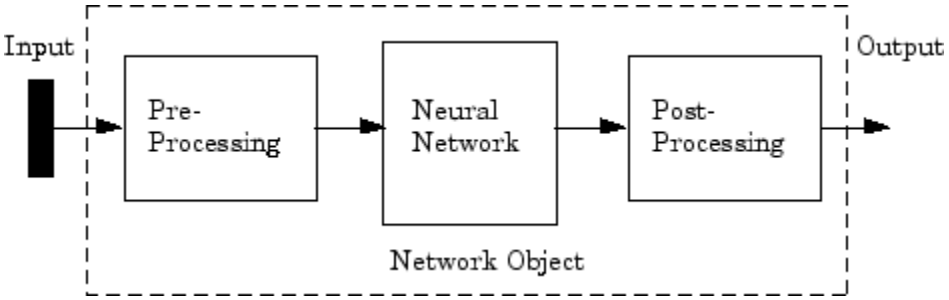
This topic presents part of a typical multilayer network workflow. For more information and other steps, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 19-2.

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 19-2.

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 25-25.

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 19-2.

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 19-12 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 19-2.

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 20-32.) 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 18-22. 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 29-2].

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 25-14 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

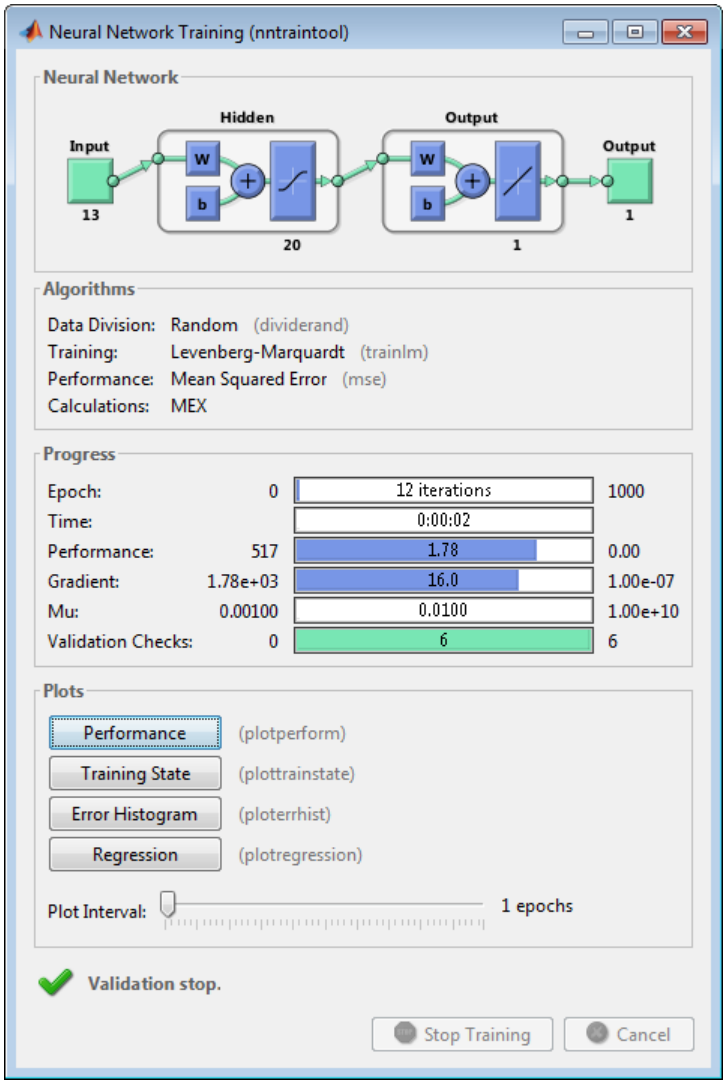
To illustrate the training process, execute the following commands:

```
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 19-18.

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 25-25.

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 19-2. To learn about how to monitor deep learning training progress, see “Monitor Deep Learning Training Progress” on page 5-49.

When the training in “Train and Apply Multilayer Shallow Neural Networks” on page 19-13 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.

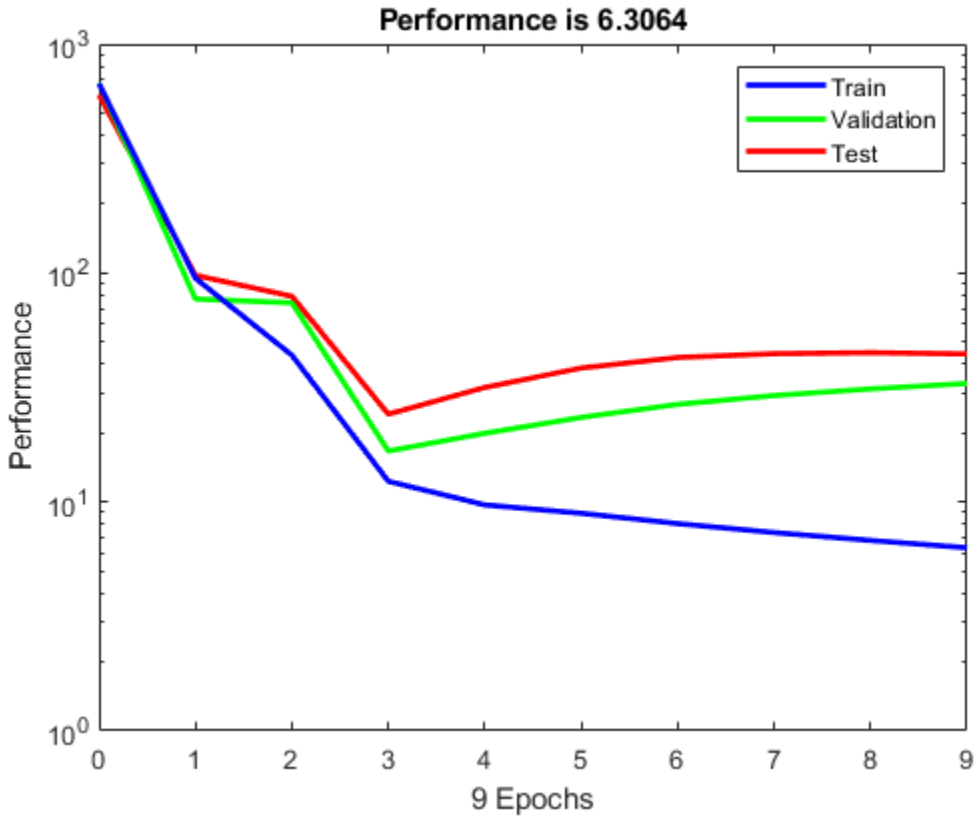
```
tr
```

```
tr = struct with fields:
        trainFcn: 'trainlm'
      trainParam: [1x1 struct]
      performFcn: 'mse'
    performParam: [1x1 struct]
        derivFcn: 'defaultderiv'
       divideFcn: 'dividerand'
      divideMode: 'sample'
     divideParam: [1x1 struct]
        trainInd: [1x176 double]
          valInd: [1x38 double]
         testInd: [1x38 double]
            stop: 'Validation stop.'
      num_epochs: 9
       trainMask: {[1x252 double]}
         valMask: {[1x252 double]}
        testMask: {[1x252 double]}
      best_epoch: 3
            goal: 0
          states: {1x8 cell}
           epoch: [0 1 2 3 4 5 6 7 8 9]
            time: [1x10 double]
            perf: [1x10 double]
           vperf: [1x10 double]
           tperf: [1x10 double]
              mu: [1x10 double]
        gradient: [1x10 double]
        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)
```

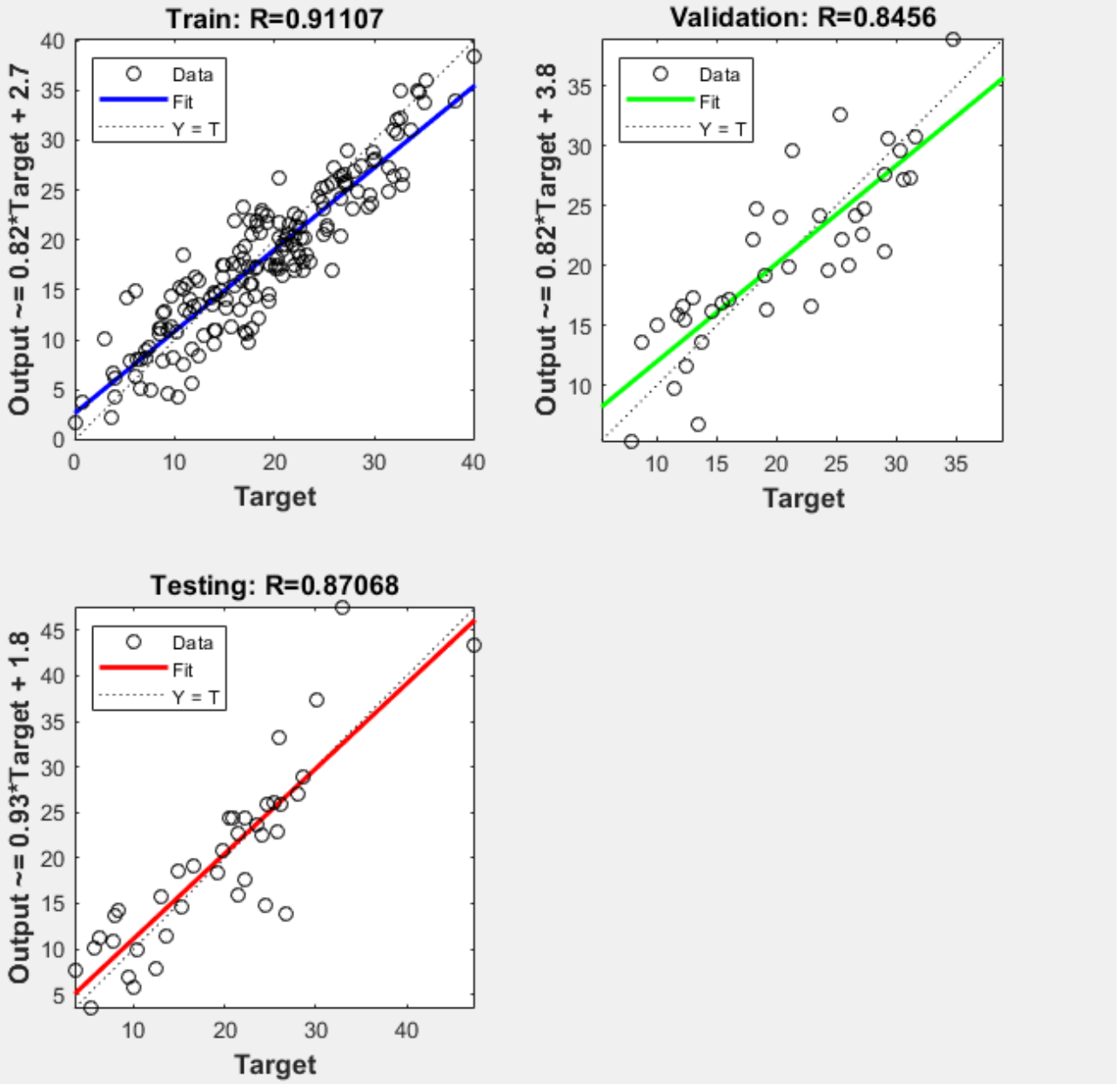


The property `tr.best_epoch` indicates the iteration at which the validation performance reached a minimum. The training continued for 6 more iterations before the training stopped.

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 29-2] 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 29-2], 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 25-25. This topic is also discussed starting on page 11-21 of [HDB96 on page 29-2].

For more information about the workflow with multilayer networks, see “Multilayer Shallow Neural Networks and Backpropagation Training” on page 19-2.

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 20-10
- “Prepare Input and Layer Delay States” on page 20-13
- “Design Time Series Distributed Delay Neural Networks” on page 20-14
- “Design Time Series NARX Feedback Neural Networks” on page 20-16
- “Design Layer-Recurrent Neural Networks” on page 20-22

Optional workflow steps are discussed in these sections:

- “Choose Neural Network Input-Output Processing Functions” on page 19-7
- “Divide Data for Optimal Neural Network Training” on page 19-9
- “Train Neural Networks with Error Weights” on page 20-32

How Dynamic Neural Networks Work

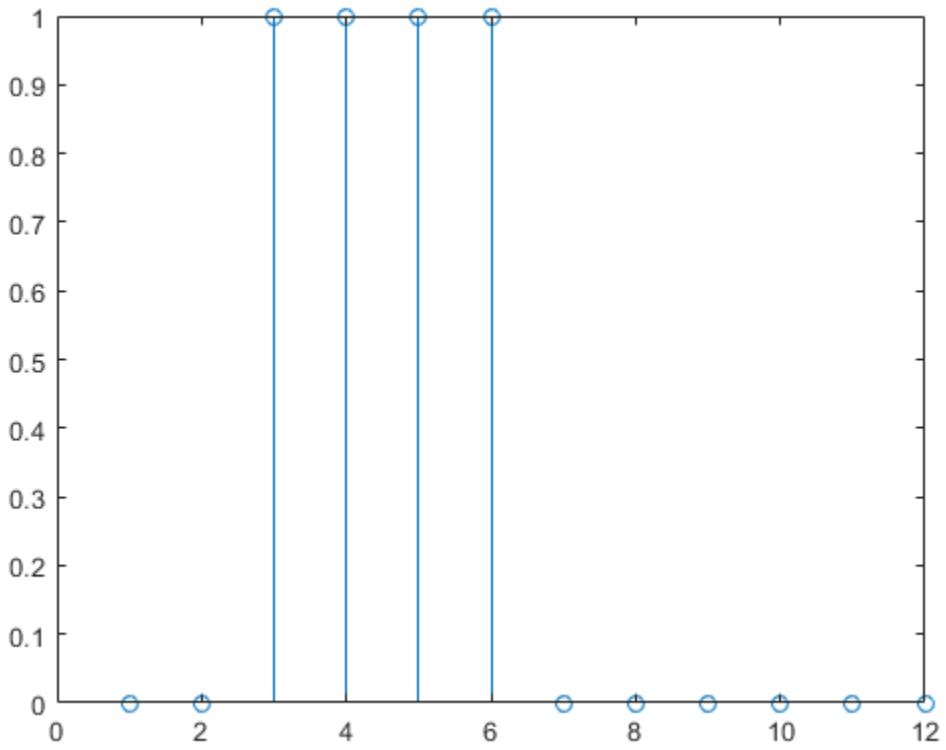
In this section...

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 18-19.)

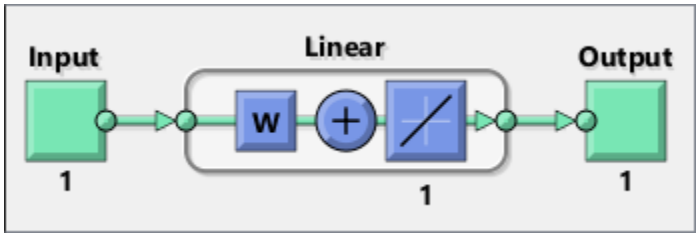
The following commands create a pulse input sequence and plot it:

```
p = {0 0 1 1 1 1 0 0 0 0 0 0};
stem(cell2mat(p))
```



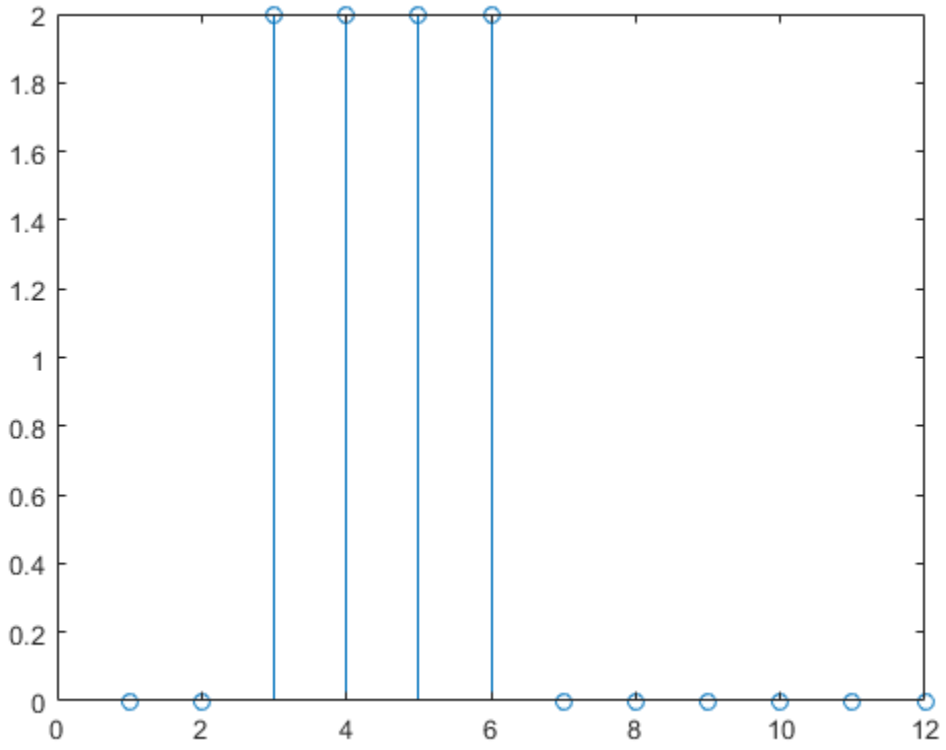
Now create a static network and find the network response to the pulse sequence. The following commands create a simple linear network with one layer, one neuron, no bias, and a weight of 2:

```
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:

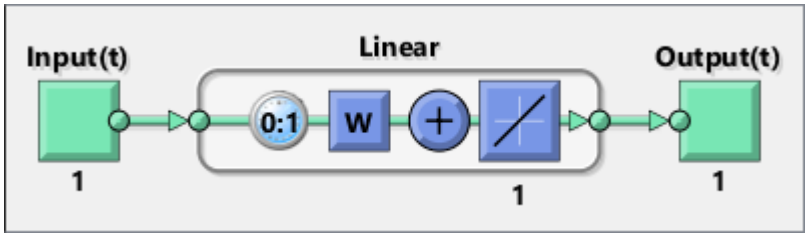
```
a = net(p);
stem(cell2mat(a))
```



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.

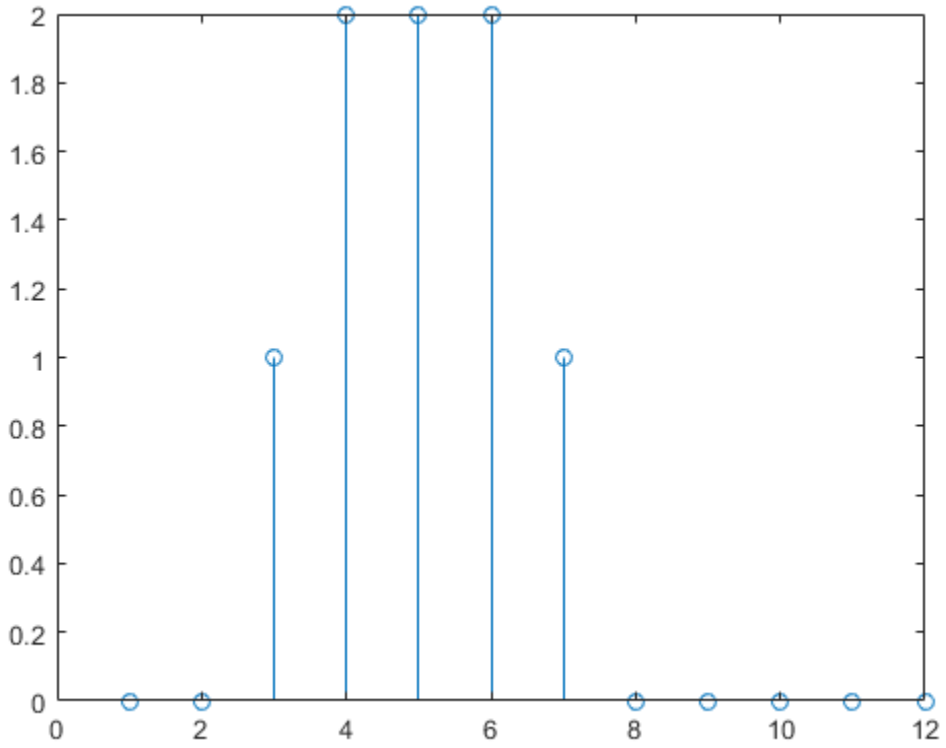
Now create a dynamic network, but one that does not have any feedback connections (a nonrecurrent network). You can use the same network used in “Simulation with Concurrent Inputs in a Dynamic Network” on page 18-20, which was a linear network with a tapped delay line on the input:

```
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:

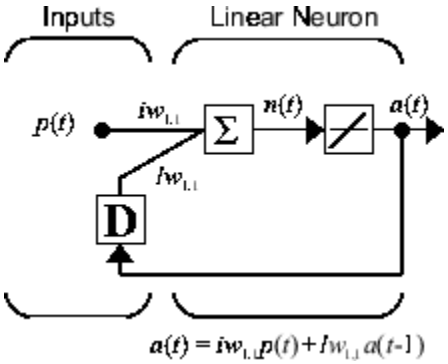
```
a = net(p);
stem(cell2mat(a))
```



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 on the history of

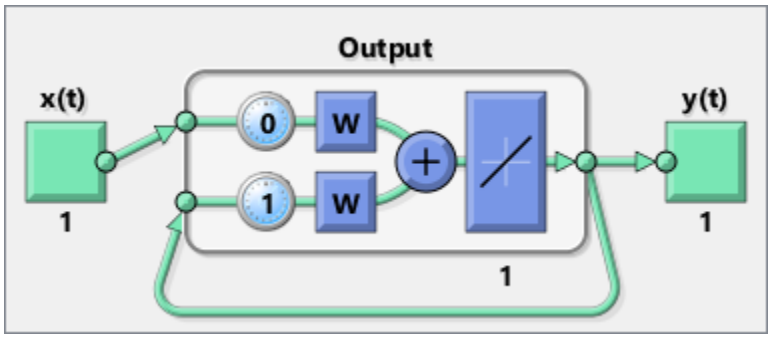
the input sequence. If the network does not have any feedback connections, then only a finite amount of history will affect the response. In this figure you can see that the response to the pulse lasts one time step beyond the pulse duration. That is because the tapped delay line on the input has a maximum delay of 1.

Now consider a simple recurrent-dynamic network, shown in the following figure.



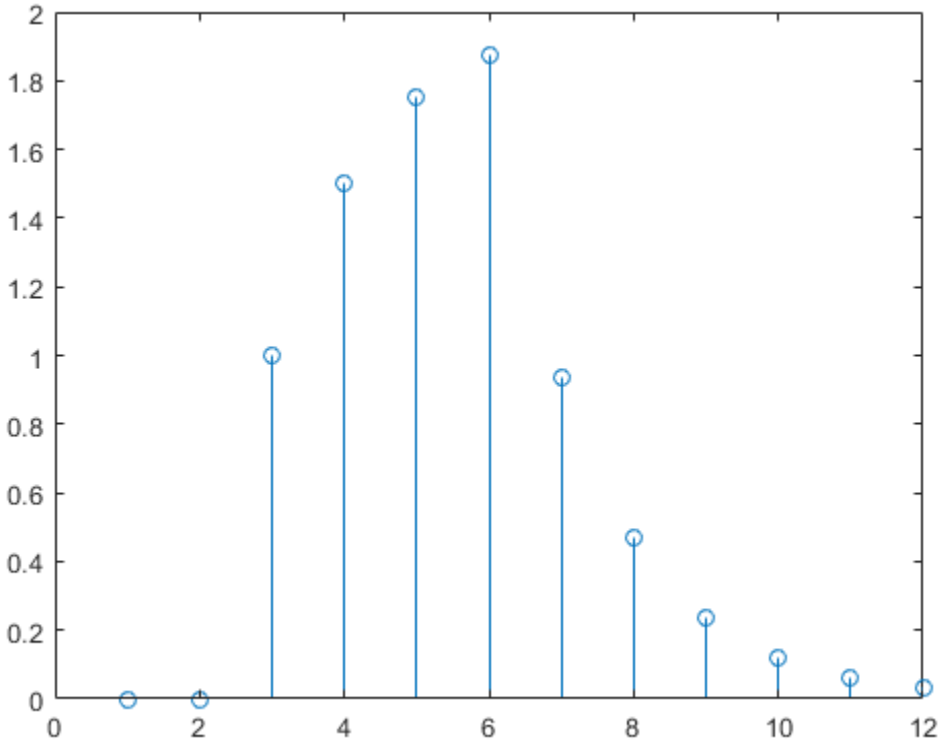
You can create the network, view it and simulate it with the following commands. The `narxnet` command is discussed in “Design Time Series NARX Feedback Neural Networks” on page 20-16.

```
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 29-2], channel equalization in communication systems [FeTs03 on page 29-2], phase detection in power systems [KaGr96 on page 29-2], sorting [JaRa04 on page 29-2], fault detection [ChDa99 on page 29-2], speech recognition [Robin94 on page 29-2], and even the prediction of protein structure in genetics [GiPr02 on page 29-2]. You can find a discussion of many more dynamic network applications in [MeJa00 on page 29-2].

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

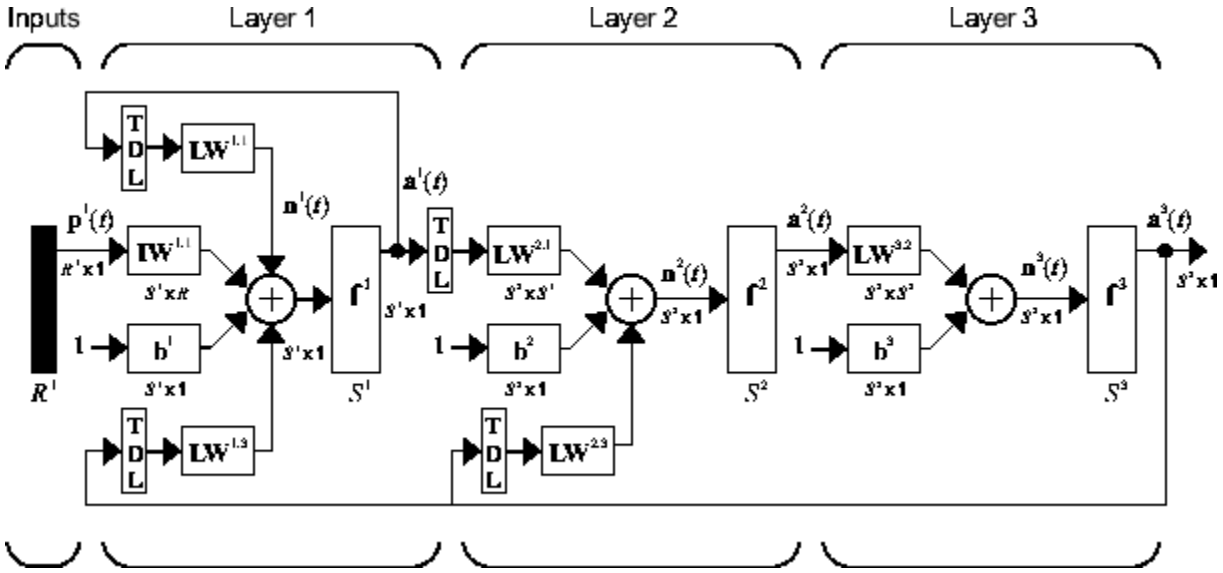
The Deep Learning Toolbox software is designed to train a class of network called the Layered Digital Dynamic Network (LDDN). Any network that can be arranged in the form of an LDDN can be trained with the toolbox. Here is a basic description of the LDDN.

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.

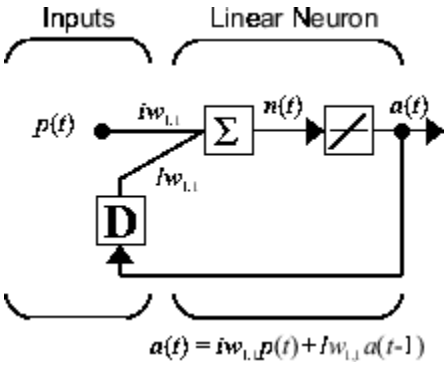


The Deep Learning Toolbox software can be used to train any LDDN, so long as the weight functions, net input functions, and transfer functions have derivatives. Most well-known dynamic network architectures can be represented in LDDN form. In the remainder of this topic you will see how to use some simple 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 19-2. 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.

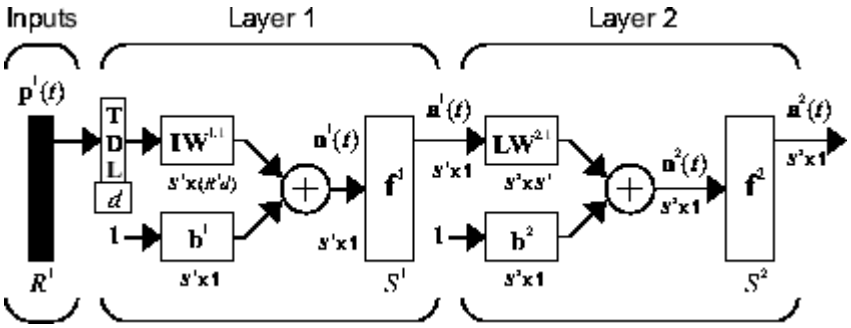


The weights have two different effects on the network output. The first is the direct effect, because a change in the weight causes an immediate change in the output at the current time step. (This first effect can be computed using standard backpropagation.) The second is an indirect effect, because 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 29-2], [DeHa01b on page 29-2] and [DeHa07 on page 29-2].) 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 29-2] and [HDH09 on page 29-2] 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.

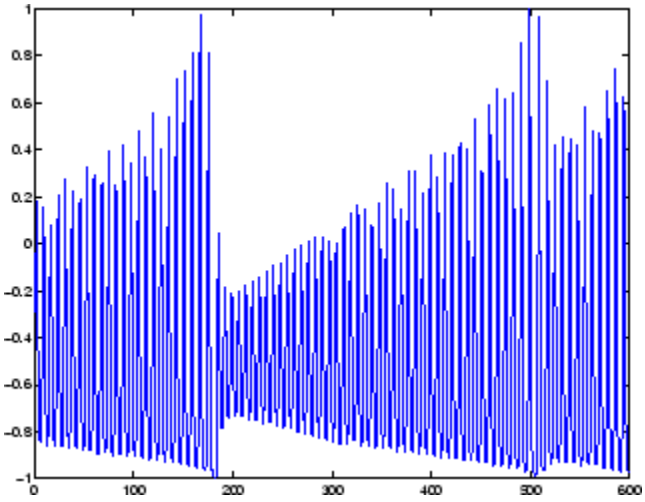
Design Time Series Time-Delay Neural Networks

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 29-2]. 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:

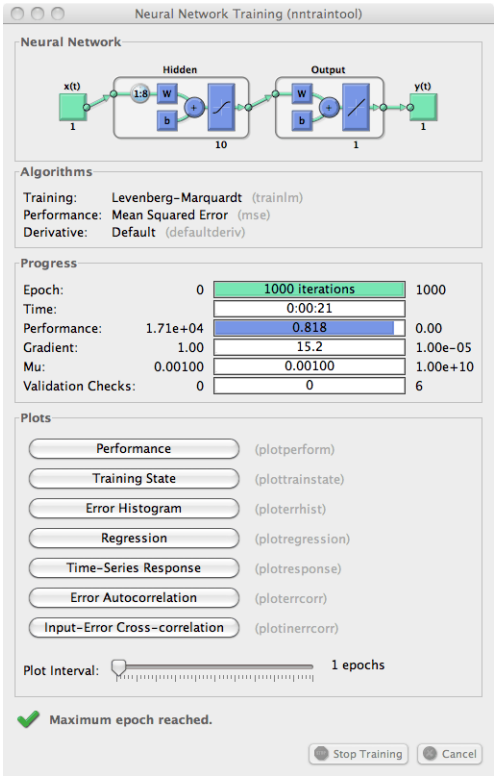
```
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = '';
```

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`):

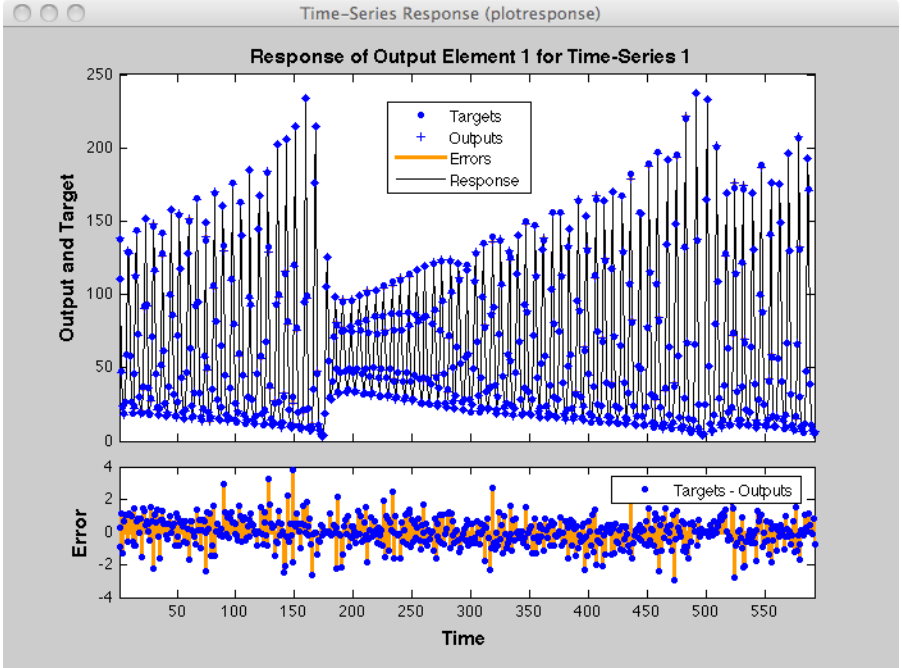
```
p = y(9:end);
t = y(9:end);
Pi=y(1:8);
ftdnn_net = train(ftdnn_net,p,t,Pi);
```

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 25-25.

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)
```

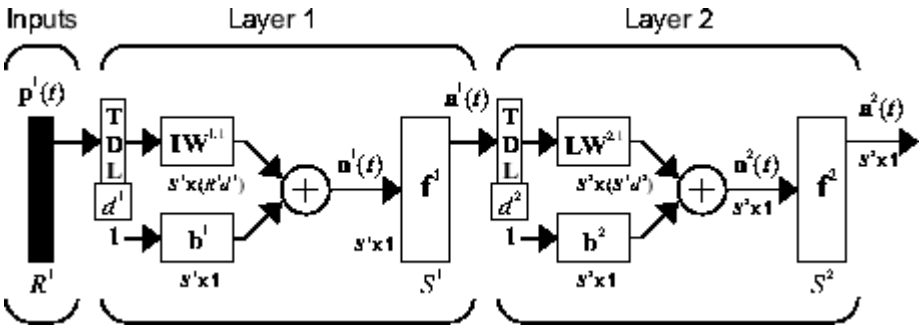
The input arguments for `preparets` are the network object (`net`), the external (non-feedback) input to the network (`inputs`), the non-feedback target (`targets`), the feedback target (`feedback`), and the error weights (`EW`) (see “Train Neural Networks with Error Weights” on page 20-32). The difference between external and feedback signals will become clearer when the NARX network is described in “Design Time Series NARX Feedback Neural Networks” on page 20-16. 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`).

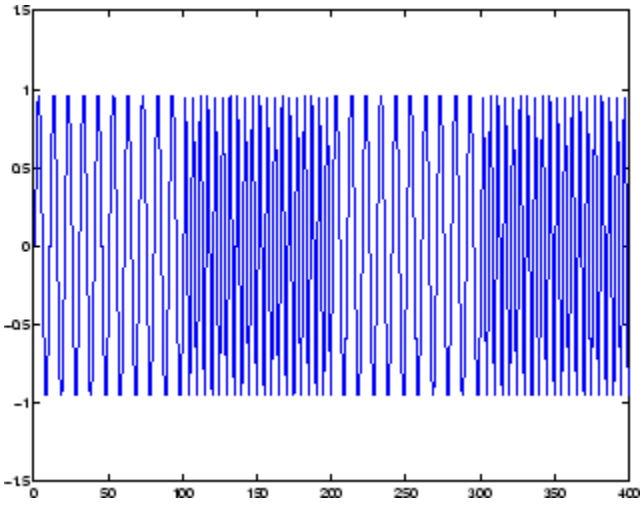
Using preparets eliminates the need to manually shift inputs and targets and load tapped delay lines. This is especially useful for more complex networks.

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 29-2] 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.



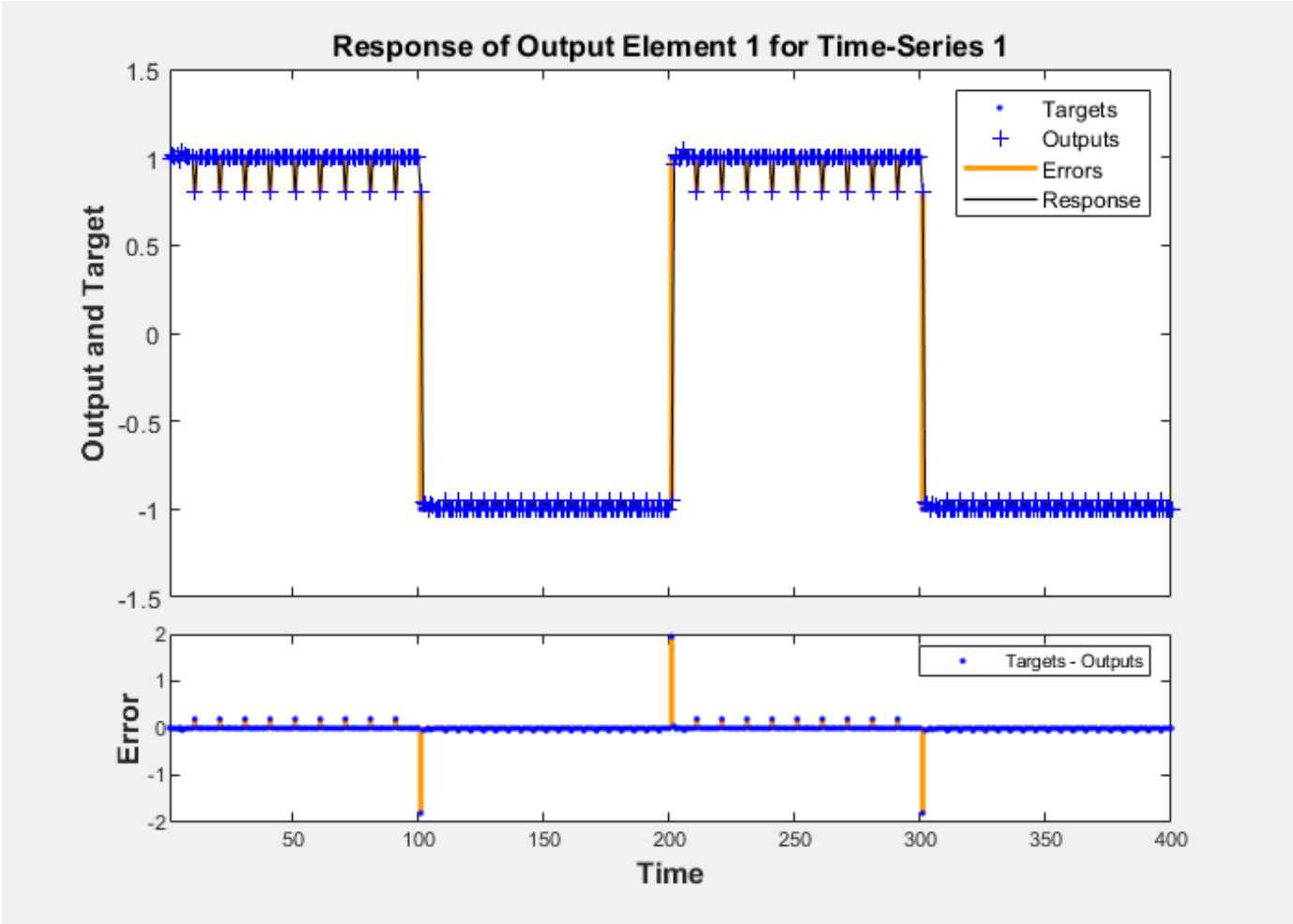
The following code creates this signal and a target network output. The target output is 1 when the input is at the low frequency and -1 when the input is at the high frequency.

```
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 19-2.)

```
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

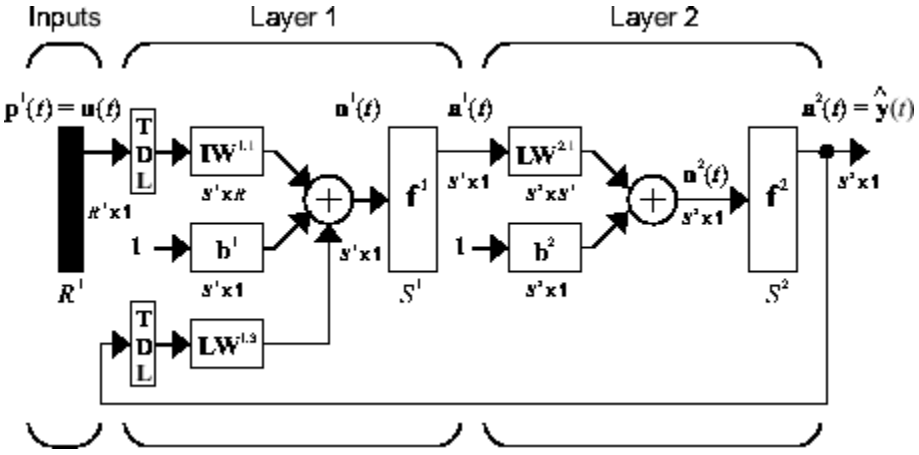
To see examples of using NARX networks being applied in open-loop form, closed-loop form and open/closed-loop multistep prediction see “Multistep Neural Network Prediction” on page 20-39.

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

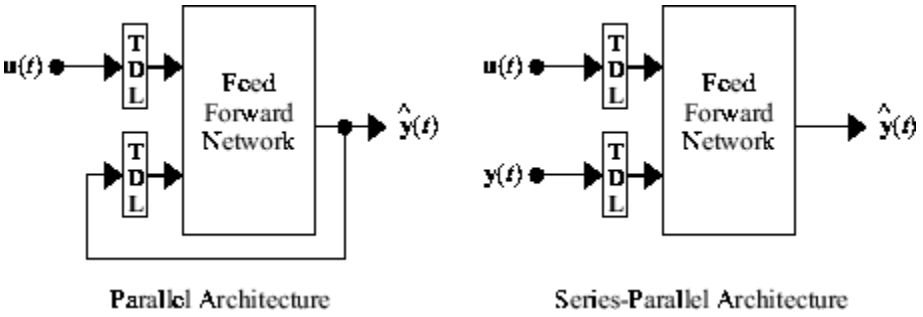
$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))$$

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 29-2]), 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.

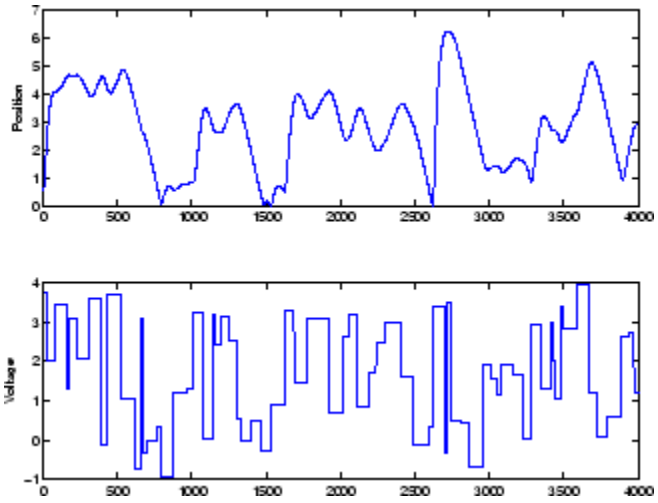


Parallel Architecture Series-Parallel Architecture

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 21-15. 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`:

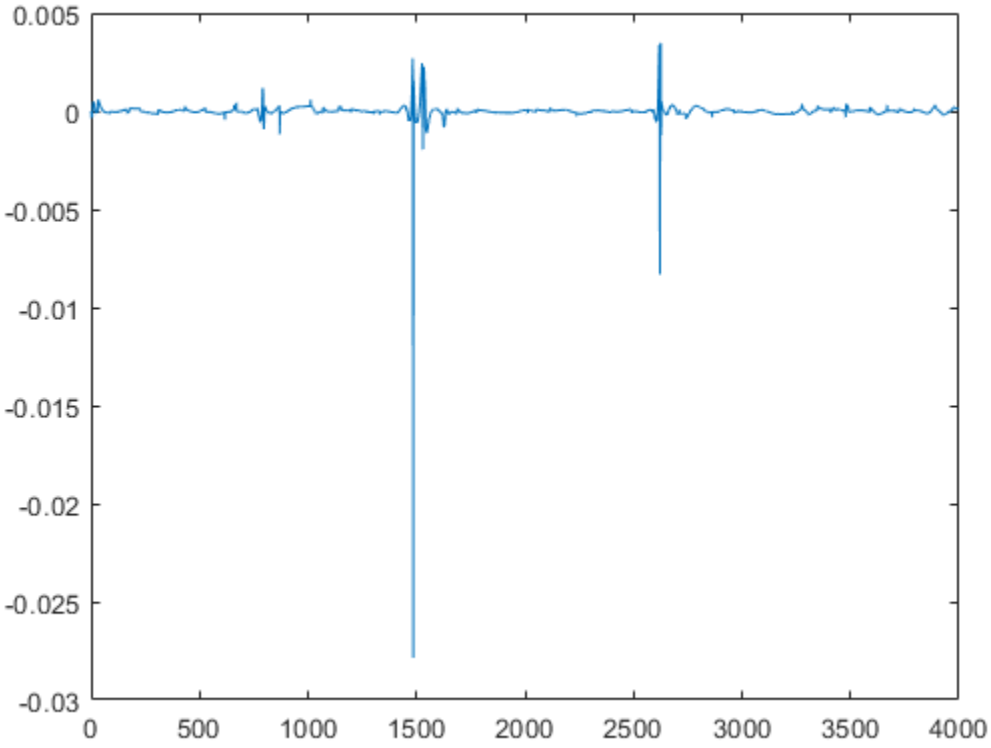
```
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,{},y);
```

(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)
```



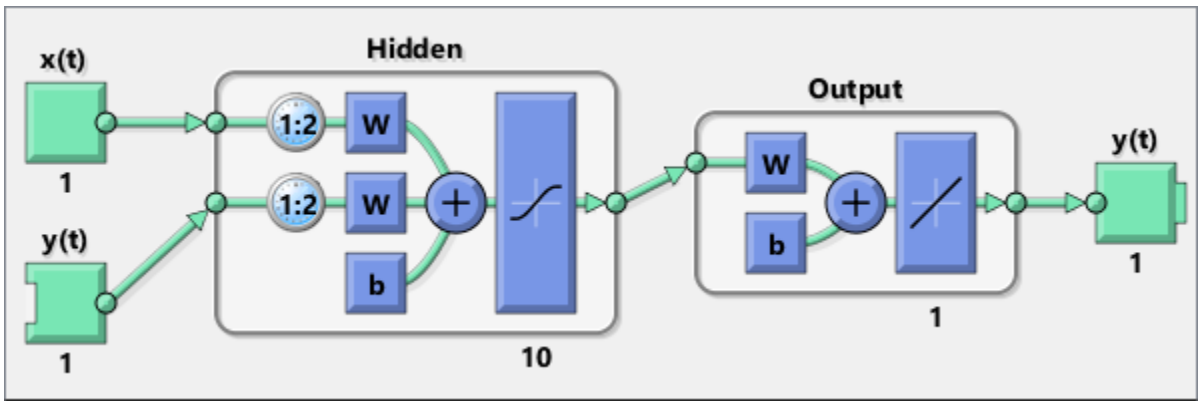
You can see that the errors are very small. However, because of the series-parallel configuration, these are errors for only a one-step-ahead prediction. A more stringent test would be to rearrange the network into the original parallel form (closed loop) and then to perform an iterated prediction over 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:

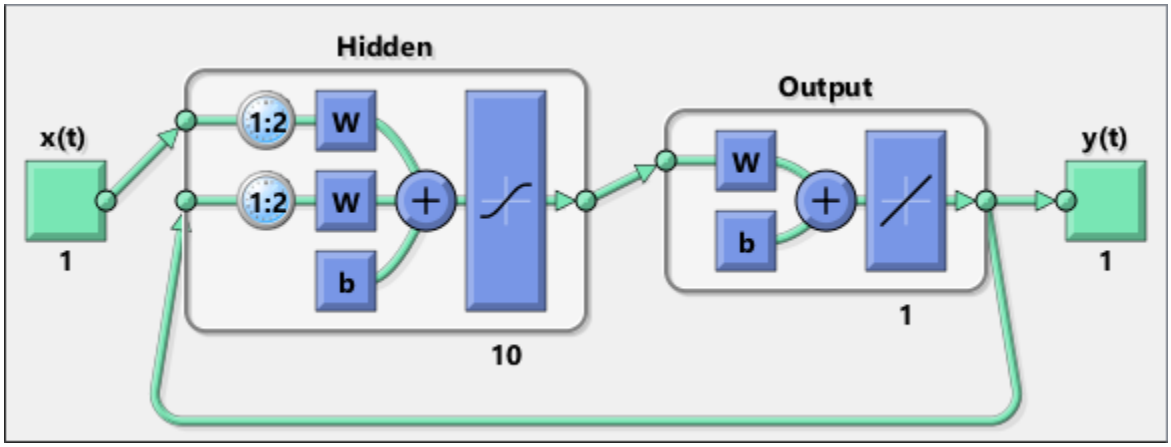
```
narx_net_closed = closeloop(narx_net);
```

To see the differences between the two networks, you can use the view command:

```
view(narx_net)
```



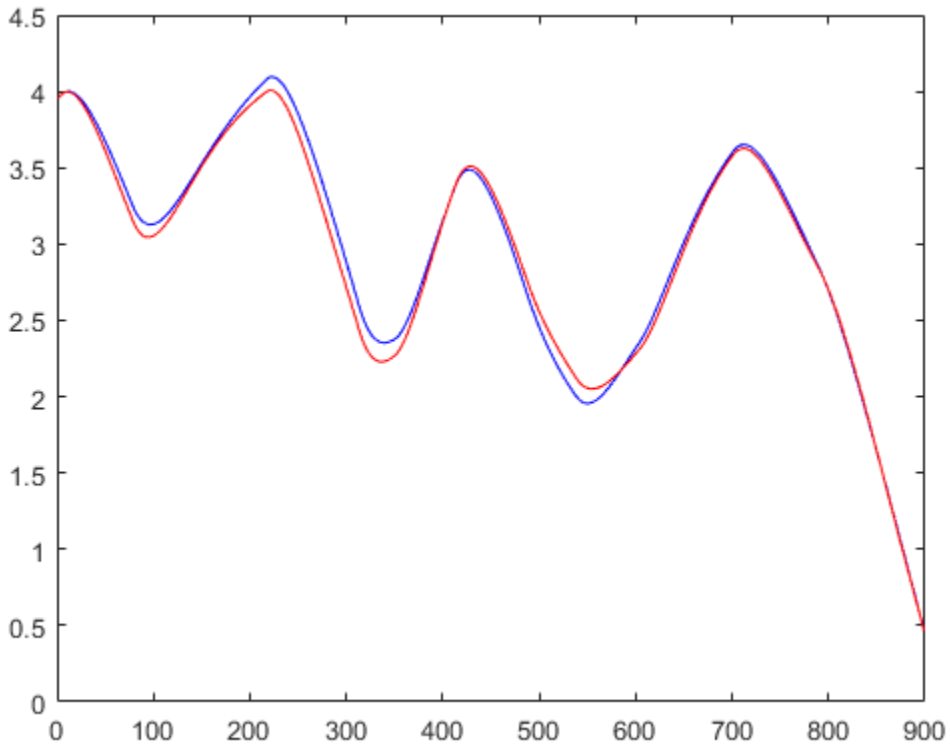

```
view(narx_net_closed)
```



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 25-25.

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`.

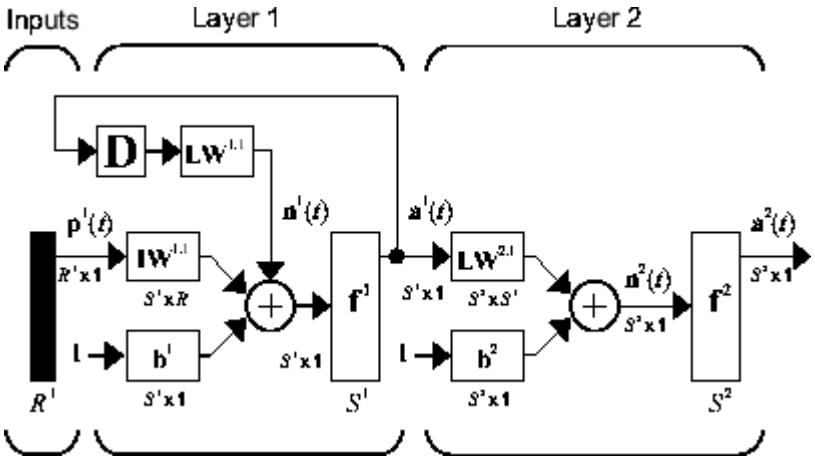
The process for training a network proceeds as it did above for the maglev problem.

```
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 20-39.

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 29-2]. 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 19-2. The following figure illustrates a two-layer LRN.

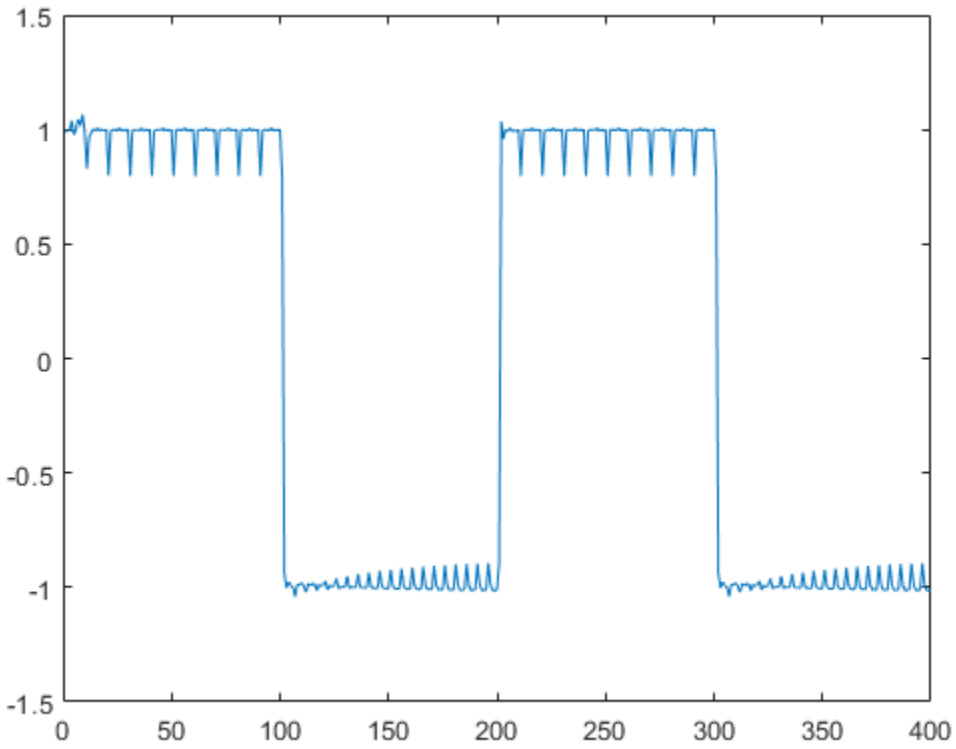


The LRN configurations are used in many filtering and modeling applications discussed already. To show its operation, this example uses the “phoneme” detection problem discussed in “Design Time Series Distributed Delay Neural Networks” on page 20-14. Here is the code to load the data and to create and 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.

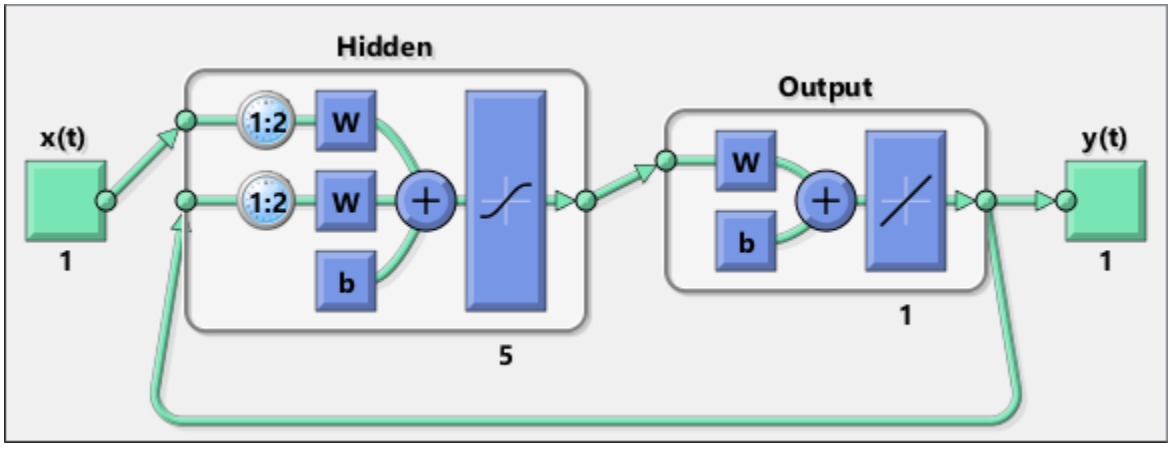
There are several other techniques for improving upon initial solutions if higher accuracy is desired. For more information, see “Improve Shallow Neural Network Generalization and Avoid Overfitting” on page 25-25.

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 19-2 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 21-19.

As you can see in “Design Model-Reference Neural Controller in Simulink” on page 21-19, 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 21-19. 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.

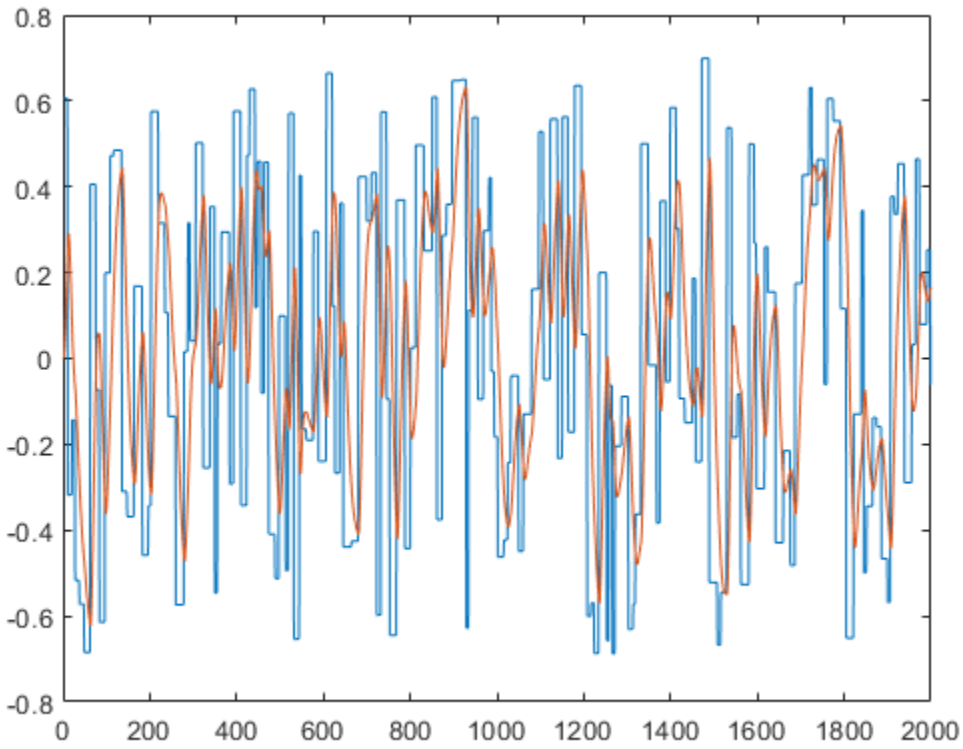
```
mrac_net = feedforwardnet([S1 1 S1]);
mrac_net.layerConnect = [0 1 0 1;1 0 0 0;0 1 0 1;0 0 1 0];
mrac_net.outputs{4}.feedbackMode = 'closed';
mrac_net.layers{2}.transferFcn = 'purelin';
mrac_net.layerWeights{3,4}.delays = 1:2;
mrac_net.layerWeights{3,2}.delays = 1:2;
mrac_net.layerWeights{3,2}.learn = 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 21-20 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.

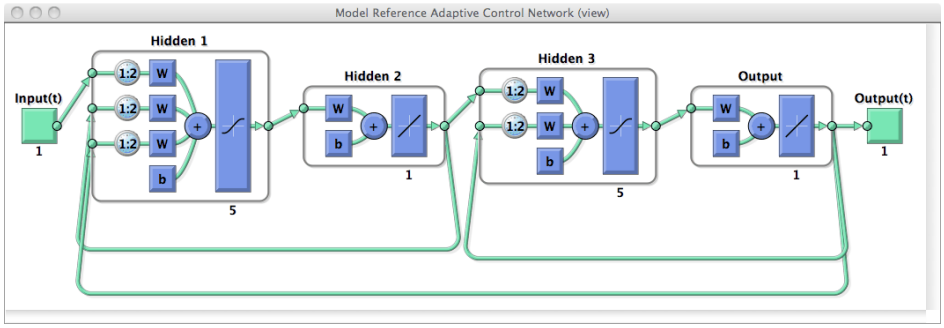
```
mrac_net.LW{2,1} = zeros(size(mrac_net.LW{2,1}));
mrac_net.b{2} = 0;
```

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.

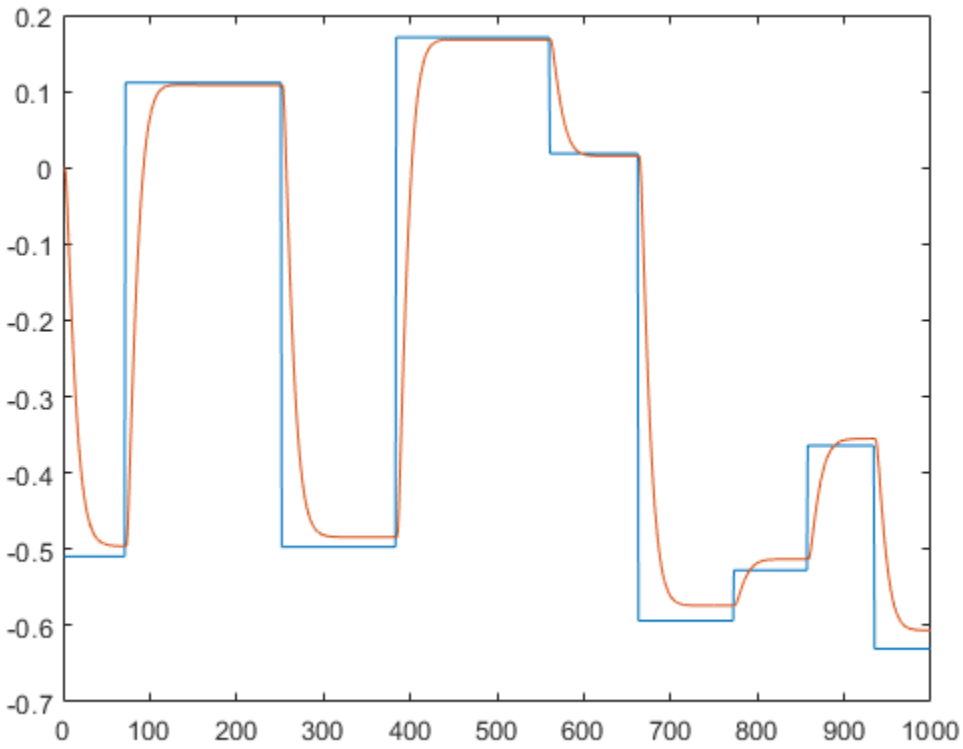
```
[x_tot,xi_tot,ai_tot,t_tot] = ...
            preparets(mrac_net,refin,{},refout);
mrac_net.trainParam.epochs = 50;
mrac_net.trainParam.min_grad = 1e-10;
[mrac_net,tr] = train(mrac_net,x_tot,t_tot,xi_tot,ai_tot);
```

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 19-2 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 29-2]), 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 29-2] 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 18-18.

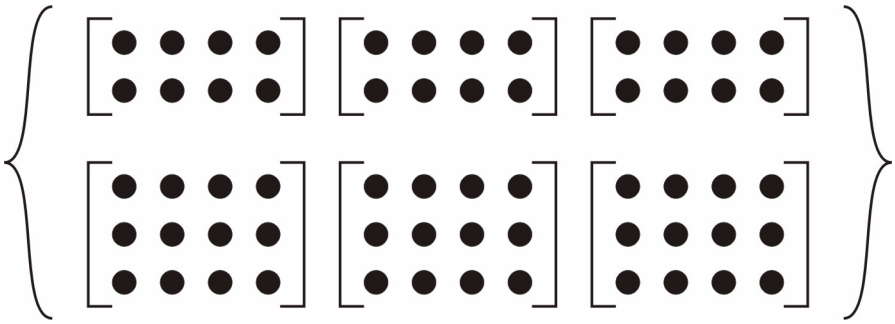
When training a network with a concurrent set of sequences, it is required that each sequence be of the same length. If this is not the case, then the shorter sequence inputs and targets should be padded with NaNs, in order to make all sequences the same length. The targets that are assigned values of 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 19-13), each squared error contributes the same amount to the performance function as follows:

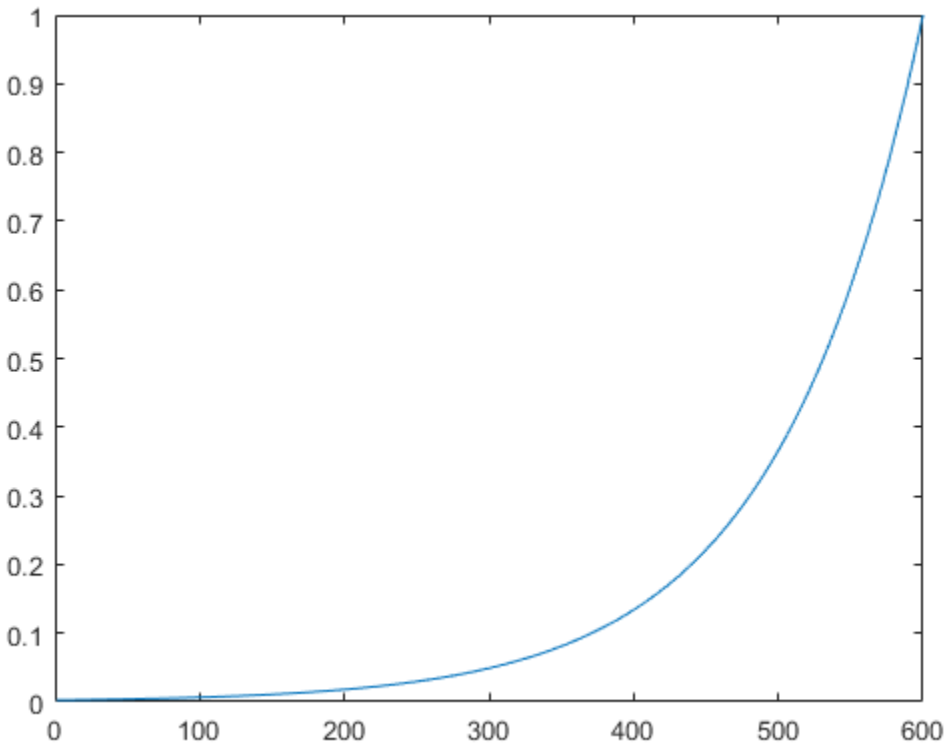
$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

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`.

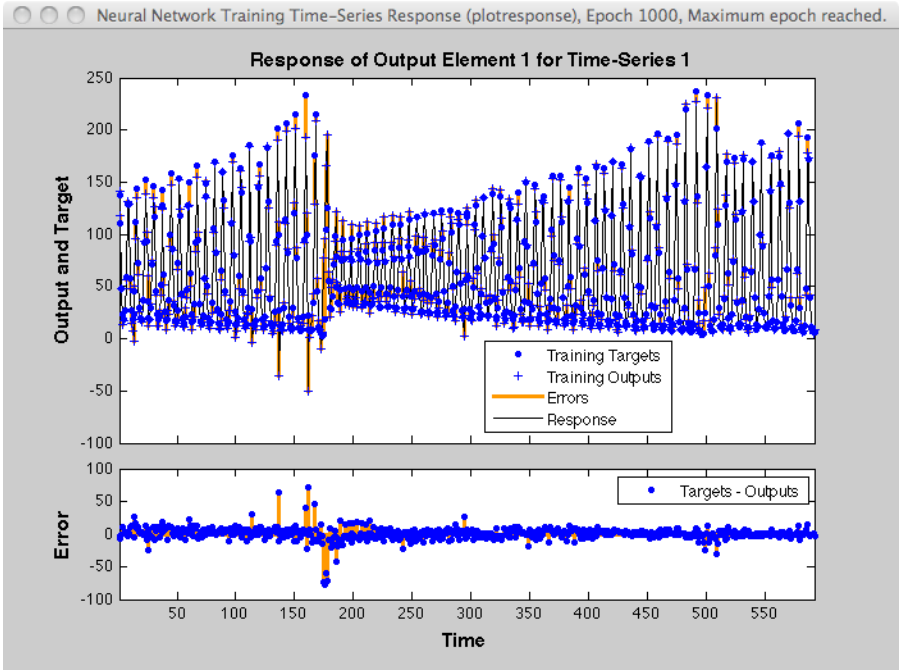
```
y = laser_dataset;
y = y(1:600);
ind = 1:600;
ew = 0.99.^(600-ind);
figure
plot(ew)
```




```
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 20-10, 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];
```

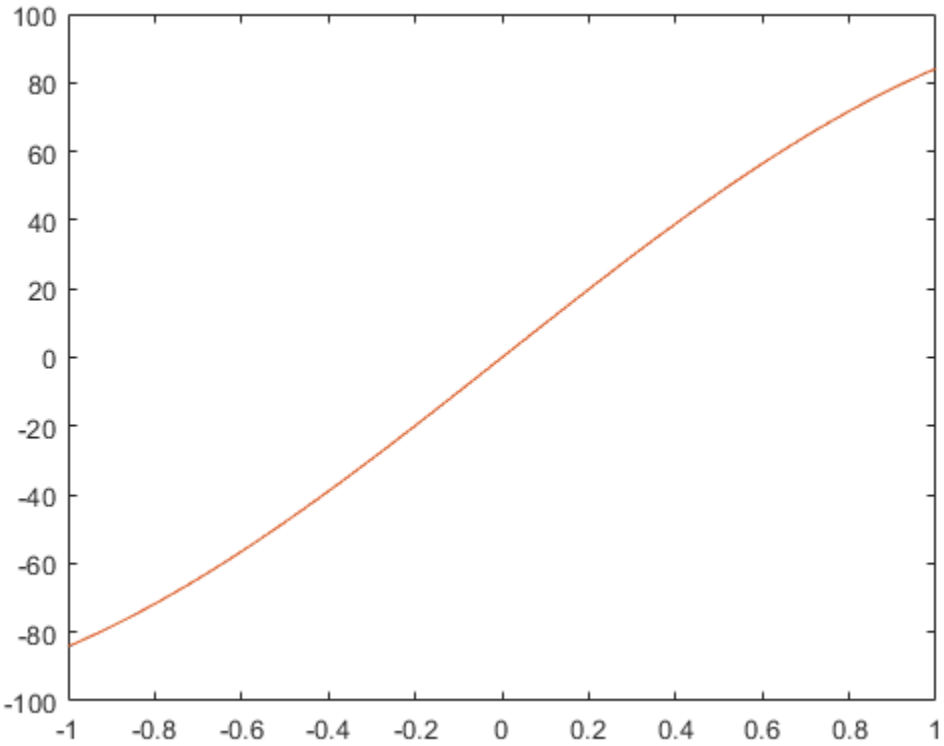
The range of `t1` is 200 (from a minimum of -100 to a maximum of 100), while the range of `t2` is only 0.02 (from -0.01 to 0.01). The range of `t1` is 10,000 times greater than the range of `t2`.

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);
```

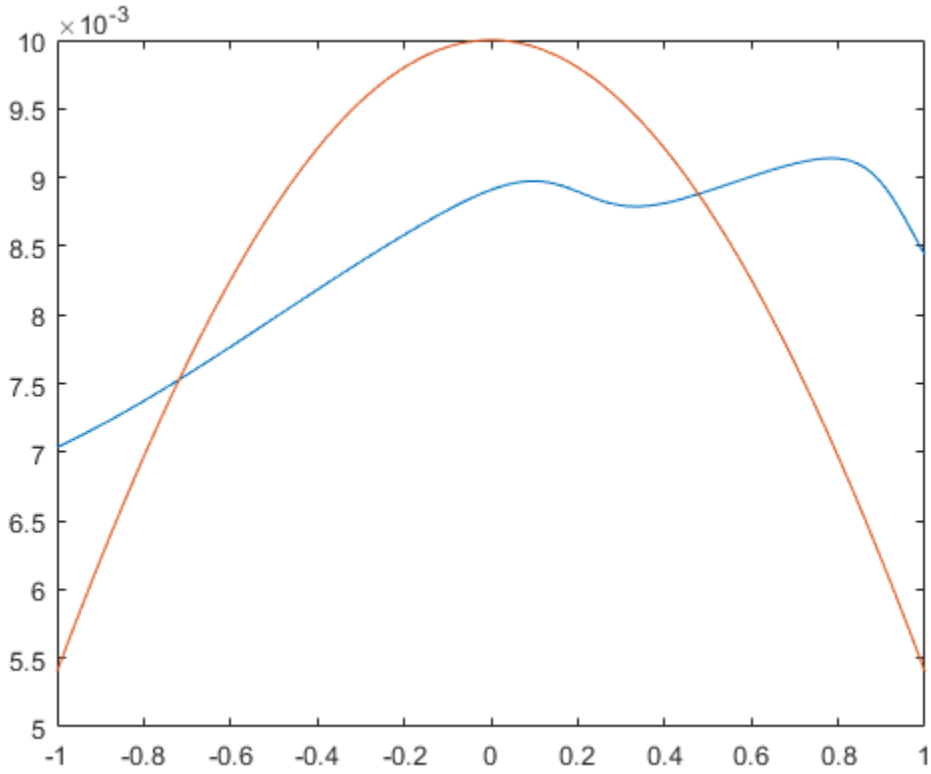
Here you can see that the network has learned to fit the first output element very well.

```
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,:))
```

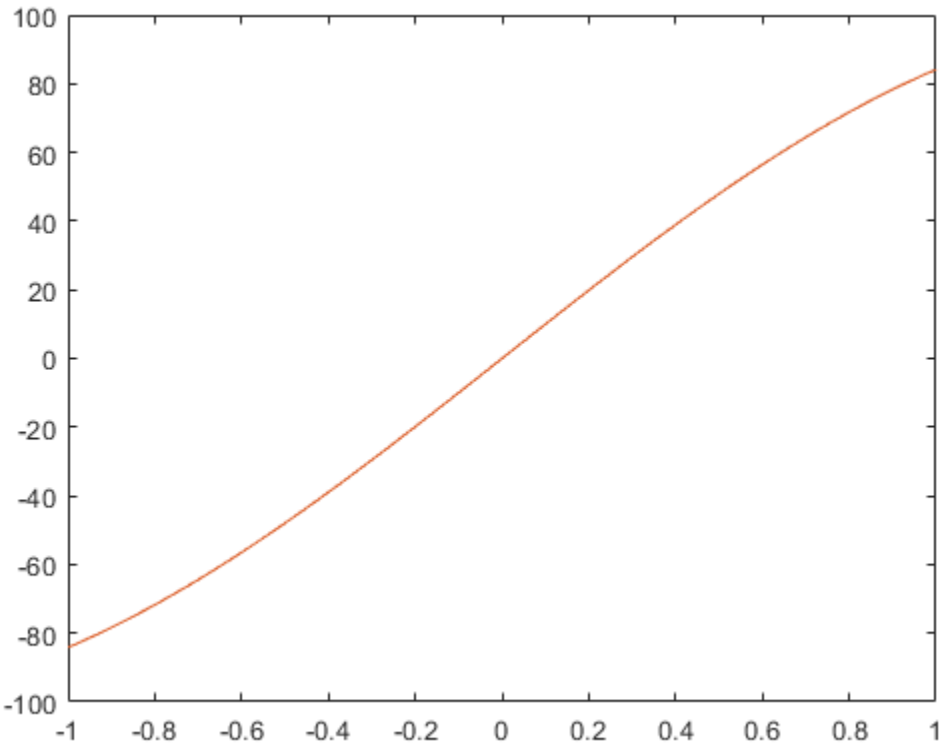


To fit both output elements equally well in a relative sense, set the `normalization` performance parameter to `'standard'`. This then calculates errors for performance measures as if each output element has a range of 2 (i.e., as if each output element's values range from -1 to 1, instead of their differing ranges).

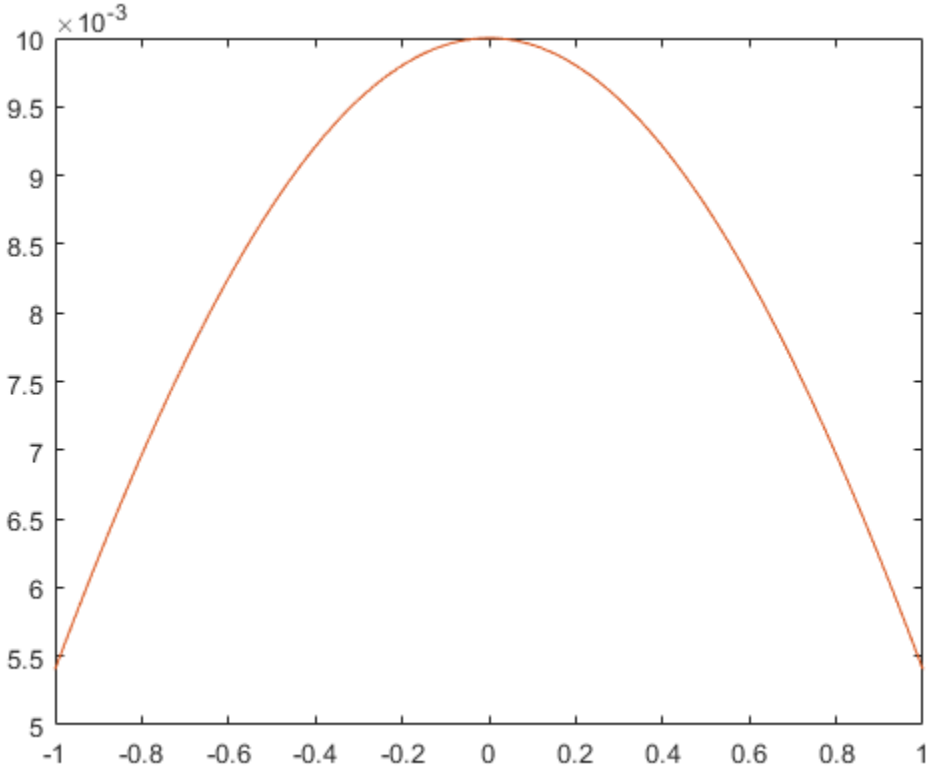
```
net.performParam.normalization = 'standard';
net2 = train(net,x,t);
y = net2(x);
```

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

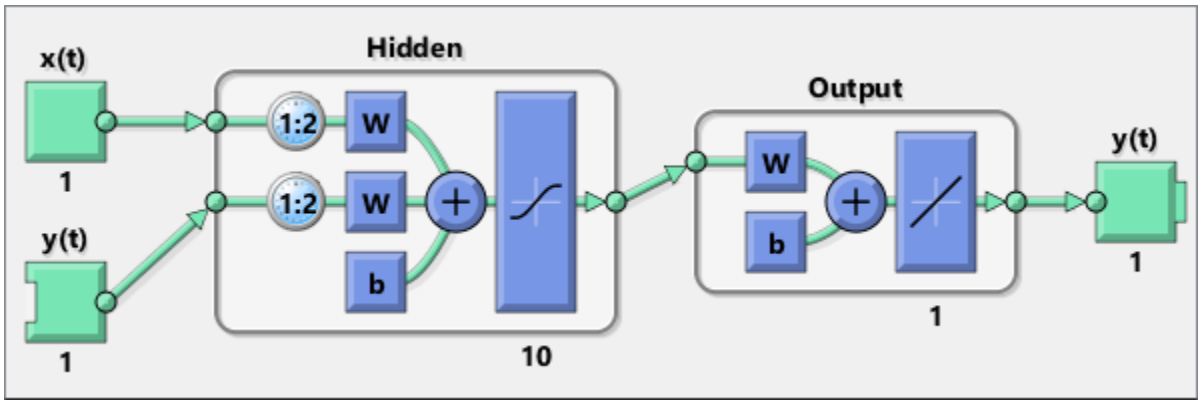
In this section...

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.

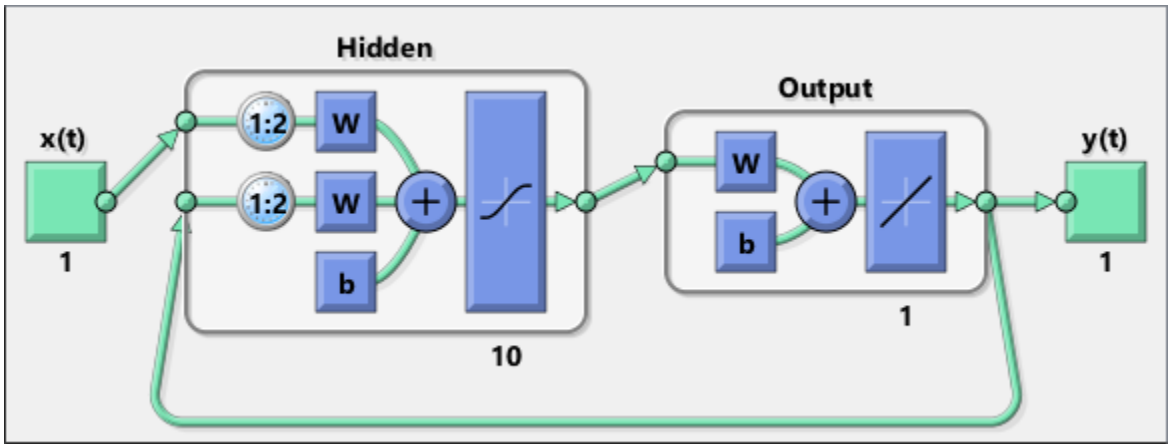
```
[X,T] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[x,xi,ai,t] = preparets(net,X,{},T);
net = train(net,x,t,xi,ai);
y = net(x,xi,ai);
view(net)
```



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 29-2]. 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 21-13 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 29-2]. 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

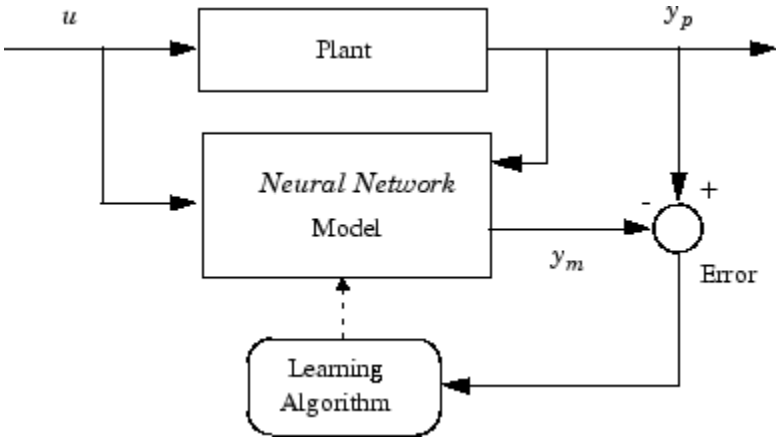
In this section...

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.)

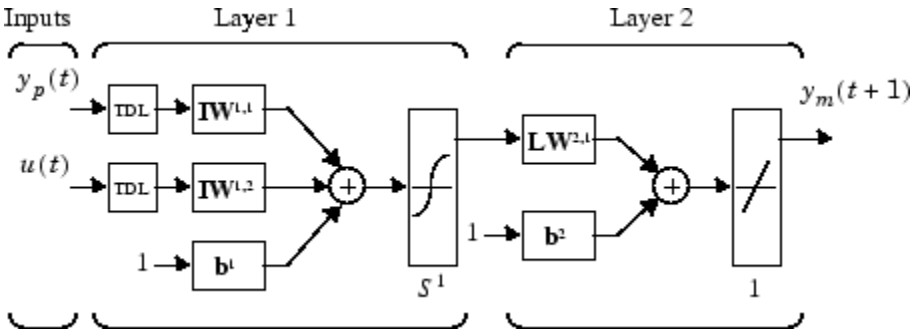
The following section describes the system identification process. This is followed by a description of the optimization process. Finally, it discusses how to use the model predictive controller block that is implemented in the Simulink environment.

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 19-2 for network training. This process is discussed in more detail in following sections.

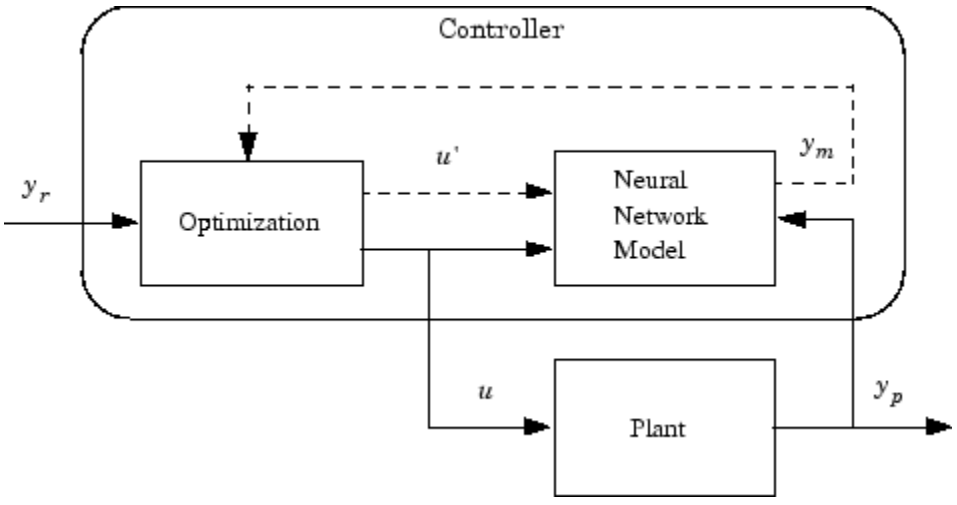
Predictive Control

The model predictive control method is based on the receding horizon technique [SoHa96 on page 29-2]. 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.

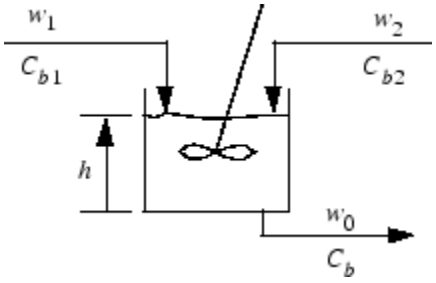
The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and then the optimal u is input to the plant. The controller block is 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

$$\frac{dh(t)}{dt} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)}$$

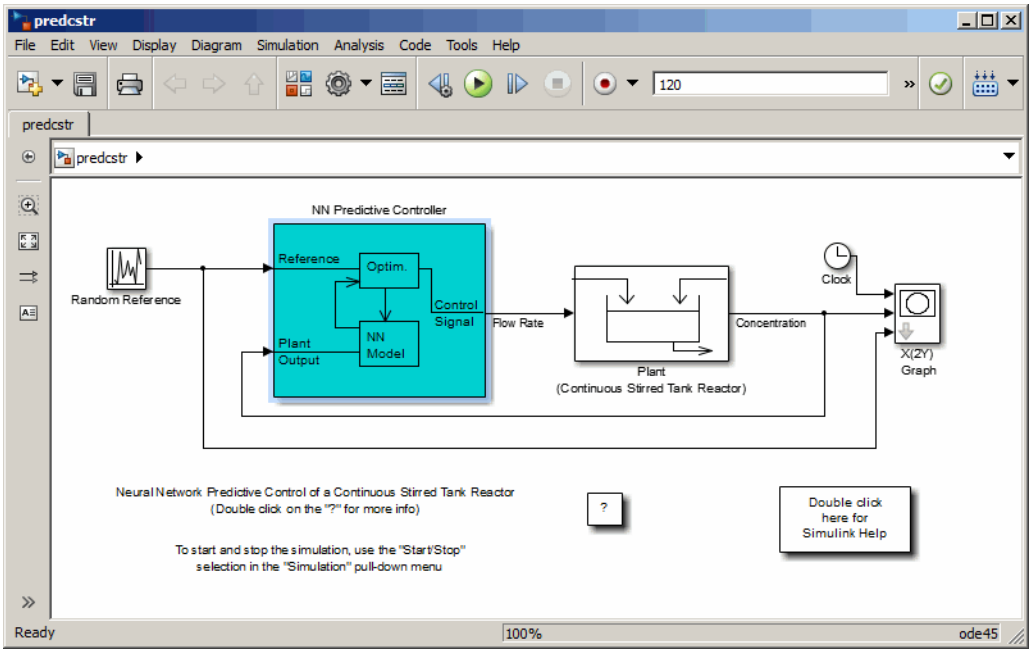
$$\frac{dC_b(t)}{dt} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2}$$

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:

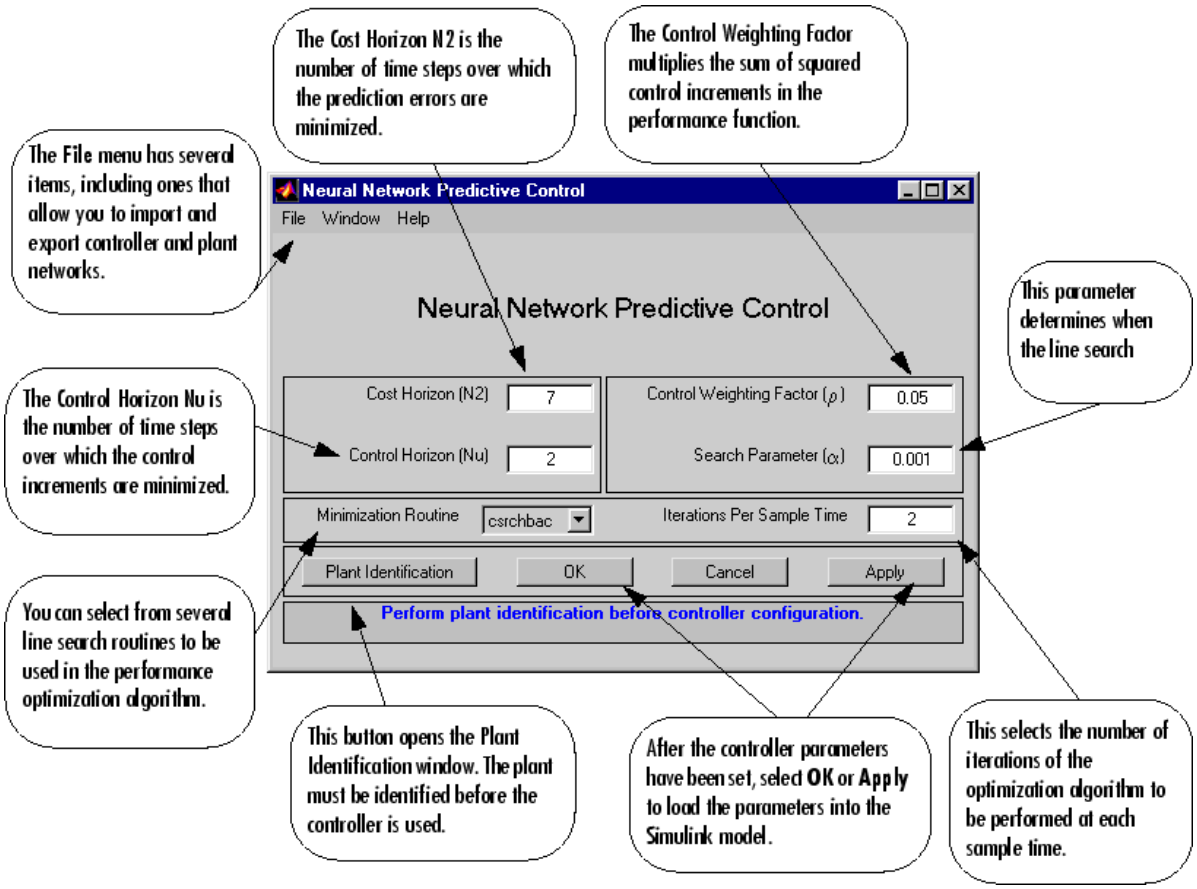
1. Start MATLAB.
2. Type `predcstr` in the MATLAB Command Window. This command opens the Simulink Editor with the following model.



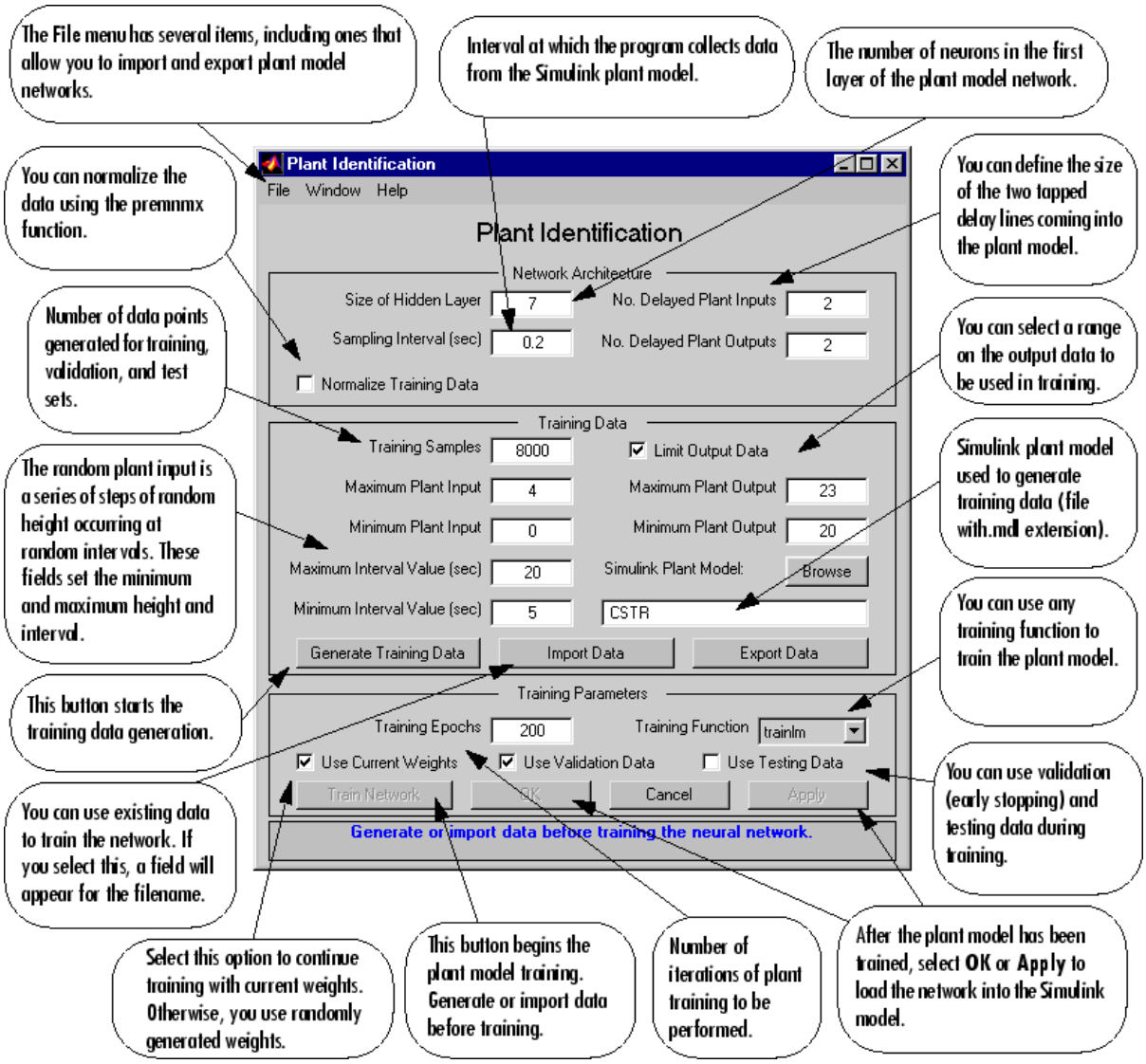
The Plant block contains the Simulink CSTR plant model. The NN Predictive Controller block signals are connected as follows:

- 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.

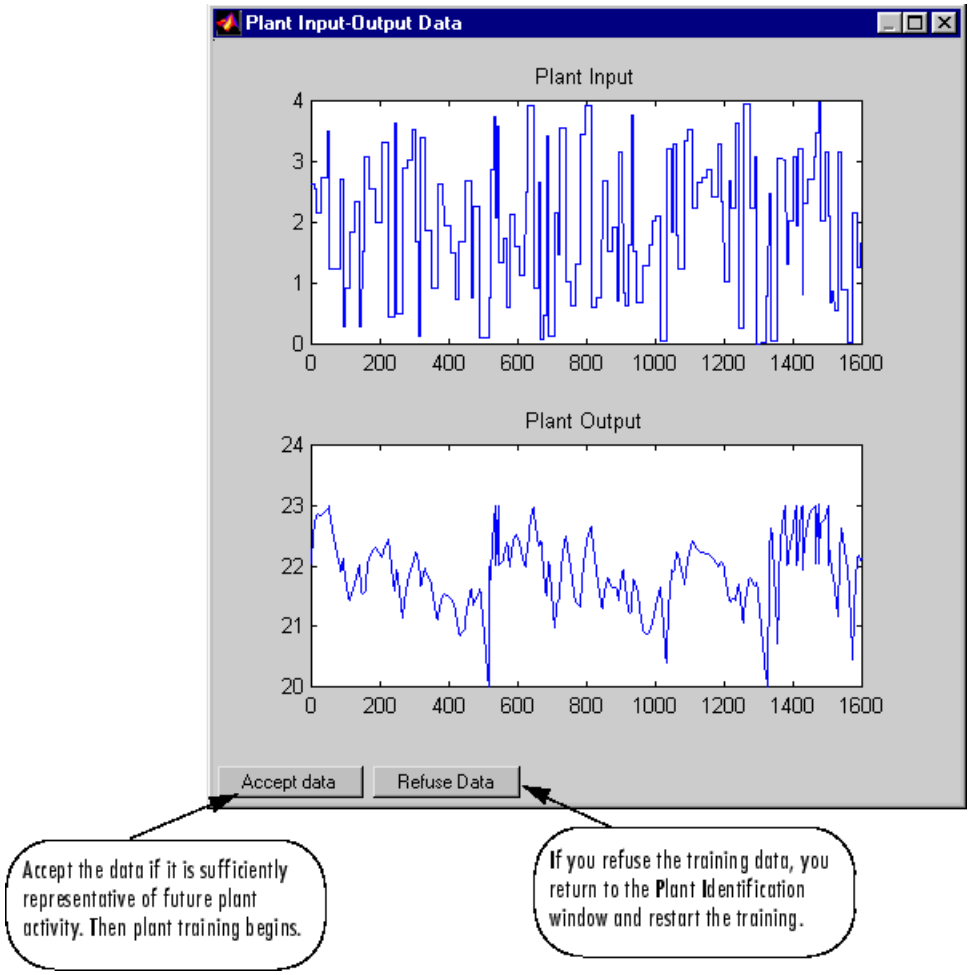
3 Double-click the NN Predictive Controller block. This opens the following window for designing the model predictive controller. This window enables you to change the controller horizons N_2 and N_u. (N_1 is fixed at 1.) The weighting parameter ρ, described earlier, is also defined in this window. The 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 19-2.



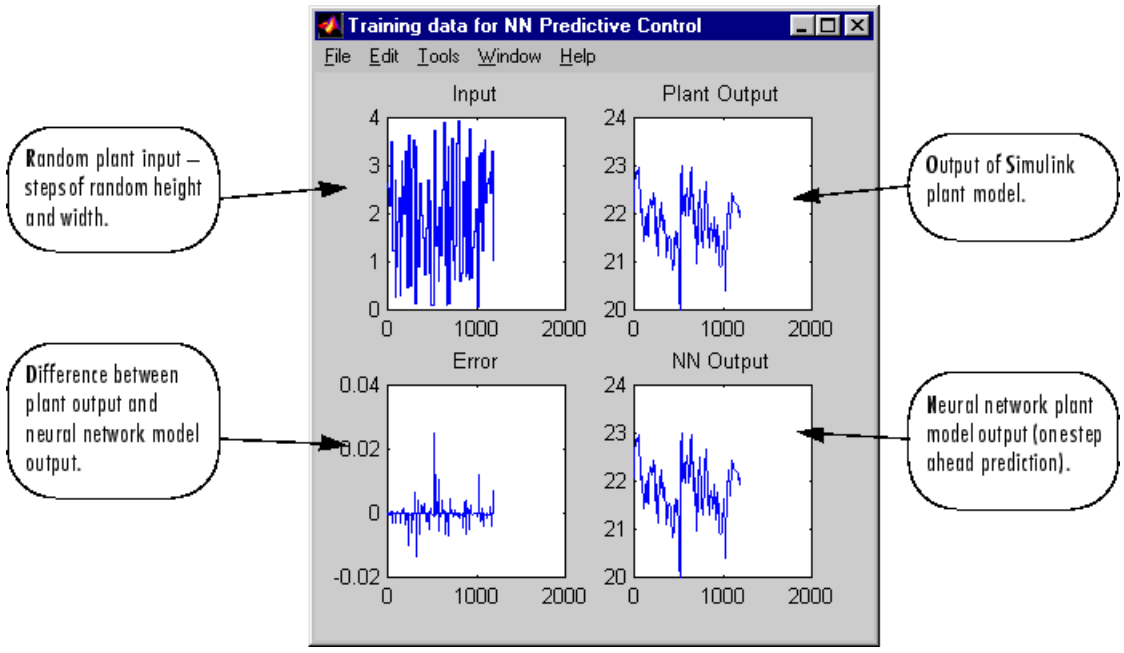
4 Select **Plant Identification**. This opens the following window. You must develop the neural network plant model before you can use the controller. The plant model predicts future plant outputs. The optimization algorithm uses these predictions to determine the control inputs that optimize future 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 19-2 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.

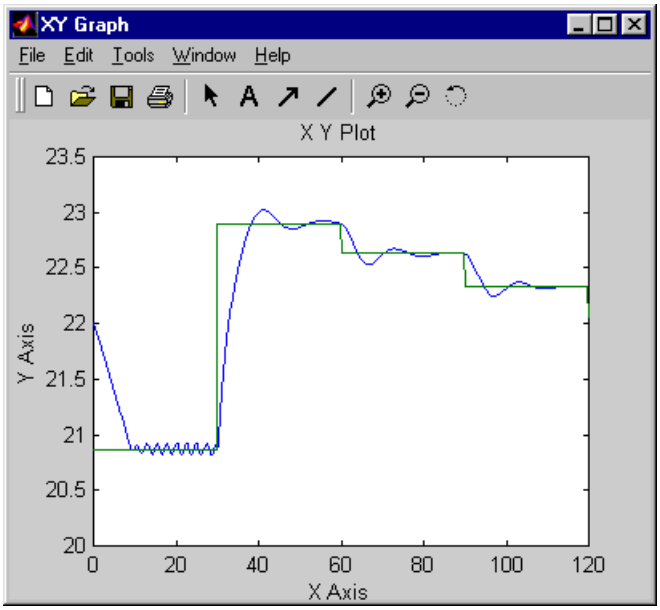


6 Click **Accept Data**, and then click **Train Network** in the Plant Identification window. Plant model training begins. The training proceeds according to the training algorithm (`trainlm` in this case) you selected. This is a straightforward application of batch training, as described in “Multilayer Shallow Neural Networks and Backpropagation Training” on page 19-2. 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 29-2] or [HaJe99 on page 29-2]). This can be quite slow. One solution, proposed by Narendra and Mukhopadhyay [NaMu97 on page 29-2], is to use approximate models to represent the system. The controller used in this section is based on the NARMA-L2 approximate model:

$$\begin{aligned}\widehat{y}(k+d) = {} & f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \\ & + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] \cdot u(k)\end{aligned}$$

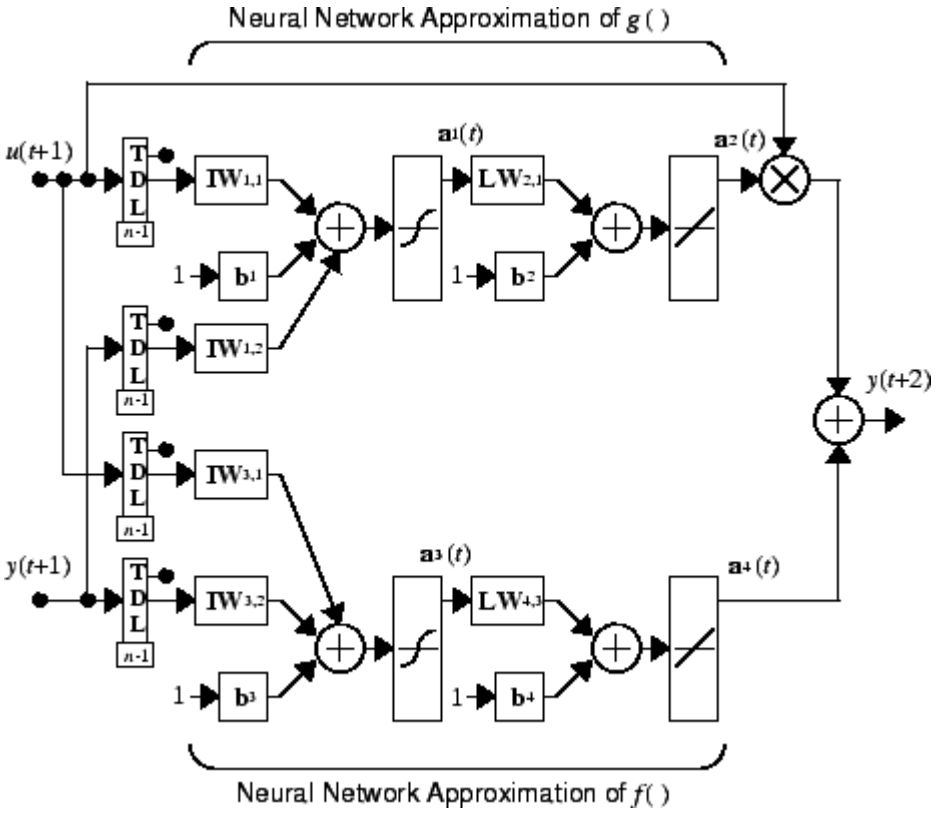
This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. The advantage of this form is that you can solve for the control input that causes the system output to follow the reference $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-n+1)]}$$

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.

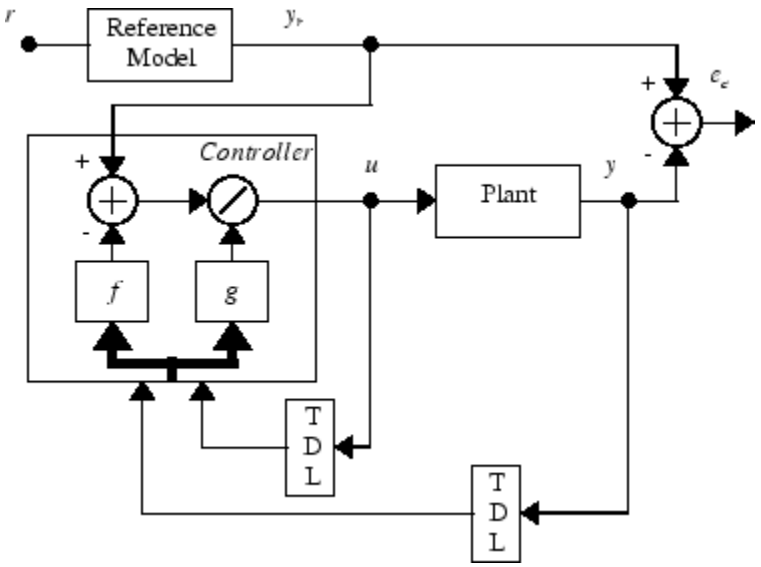


NARMA-L2 Controller

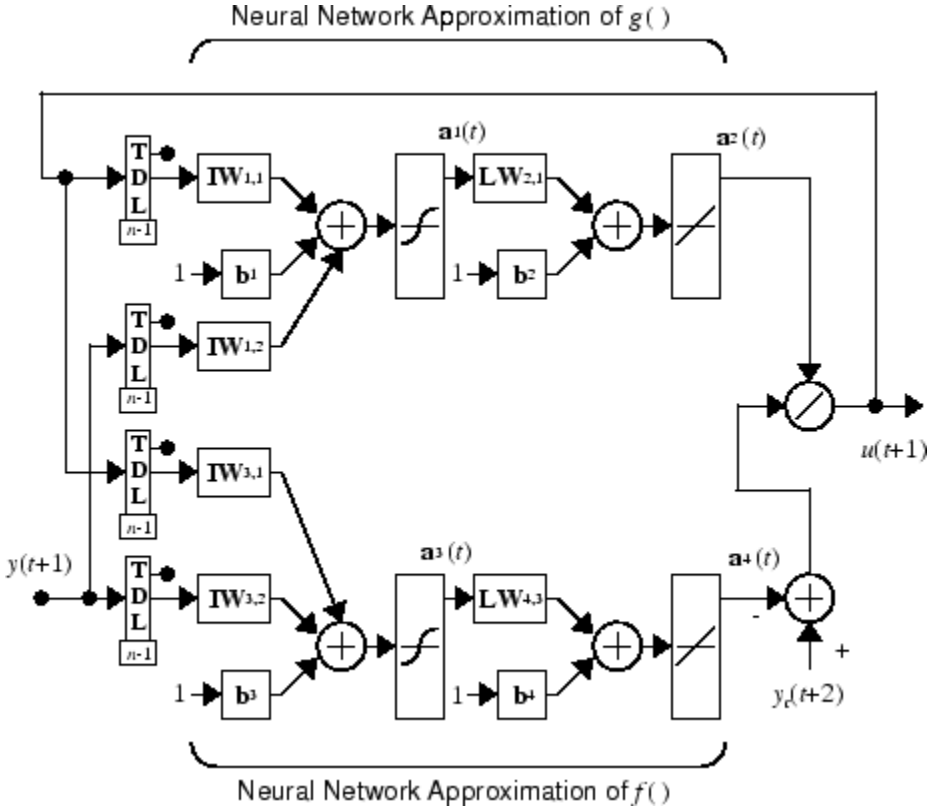
Using the NARMA-L2 model, you can obtain the controller

$$u(k+1) = \frac{y_r(k+d) - f[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}{g[y(k), \ldots, y(k-n+1), u(k), \ldots, u(k-n+1)]}$$

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.

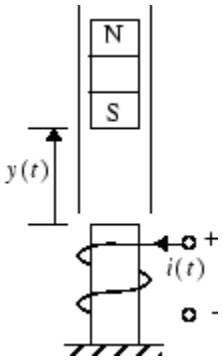


Use the NARMA-L2 Controller Block

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.



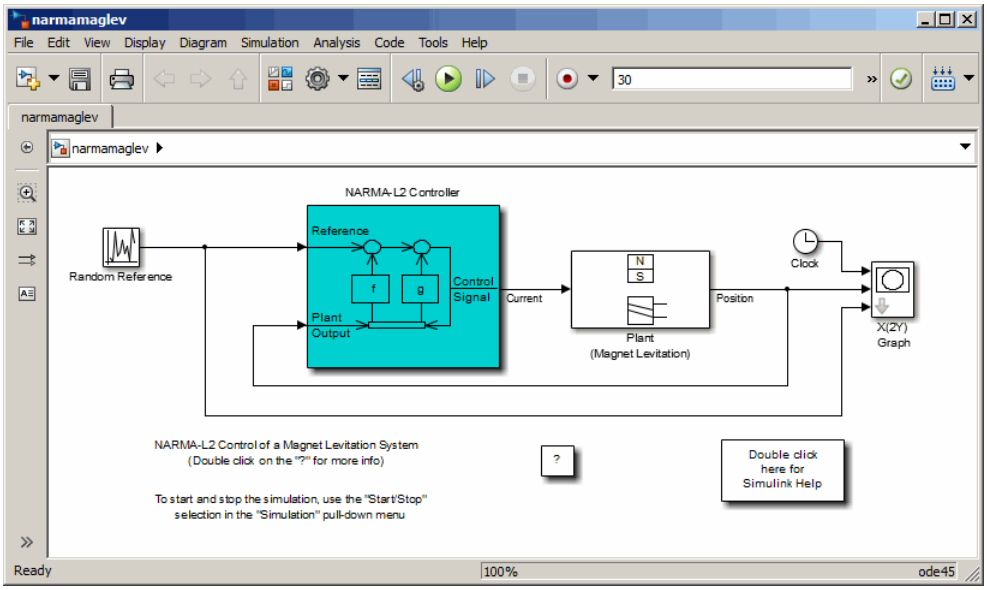
The equation of motion for this system is

$$\frac{d^2y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt}$$

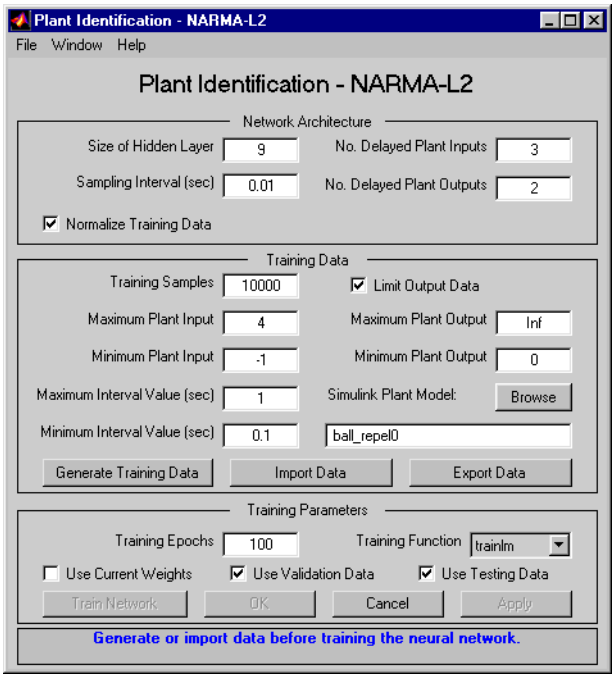
where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and 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.

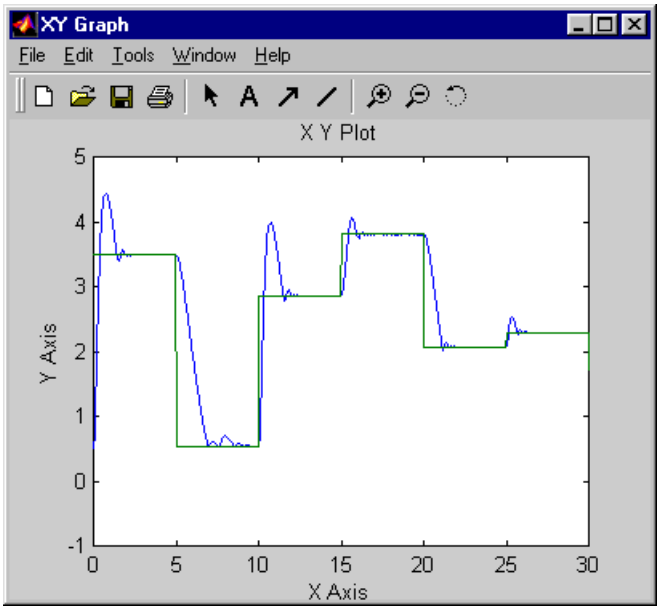


3 Double-click the NARMA-L2 Controller block. This opens the following window. This window enables you to train the NARMA-L2 model. There is no separate window for the controller, because the controller is determined directly from the model, unlike the model predictive controller.



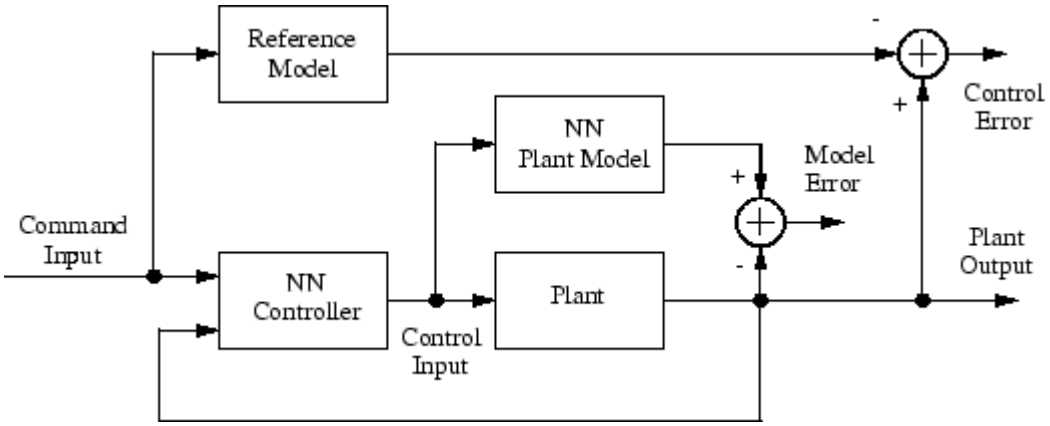
4 This window works the same as the other Plant Identification windows, so the training process is not repeated. Instead, simulate the NARMA-L2 controller.

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.



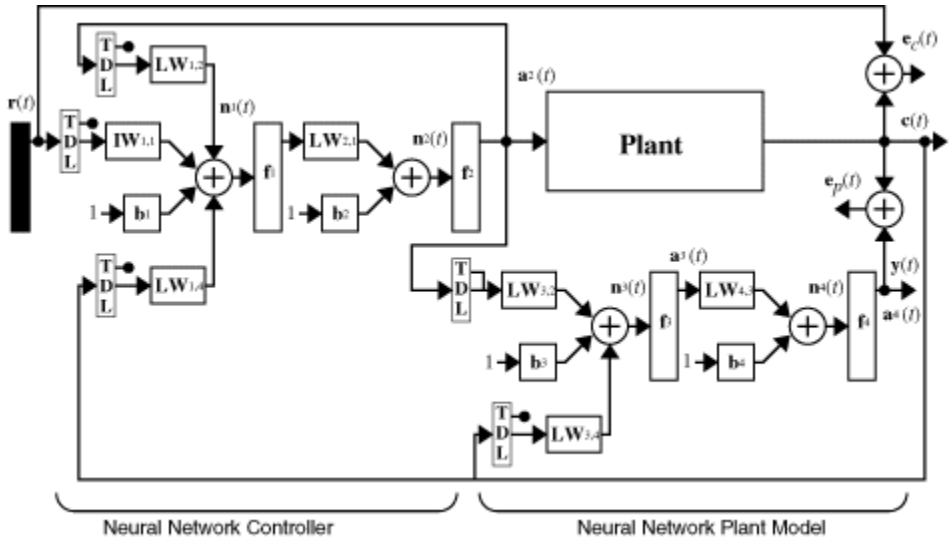
The following figure shows the details of the neural network plant model and the neural network controller as they are implemented in the Deep Learning Toolbox software. Each network has two layers, and you can select the number of neurons to use in the hidden layers. There are three sets of controller 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

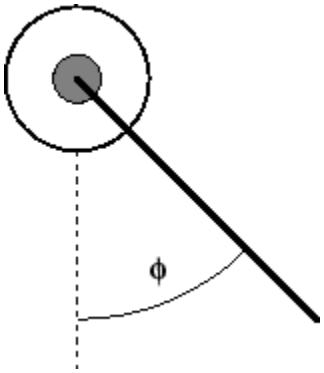
As with the controller, you can set the number of delays. The next section shows how you can set the parameters.



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

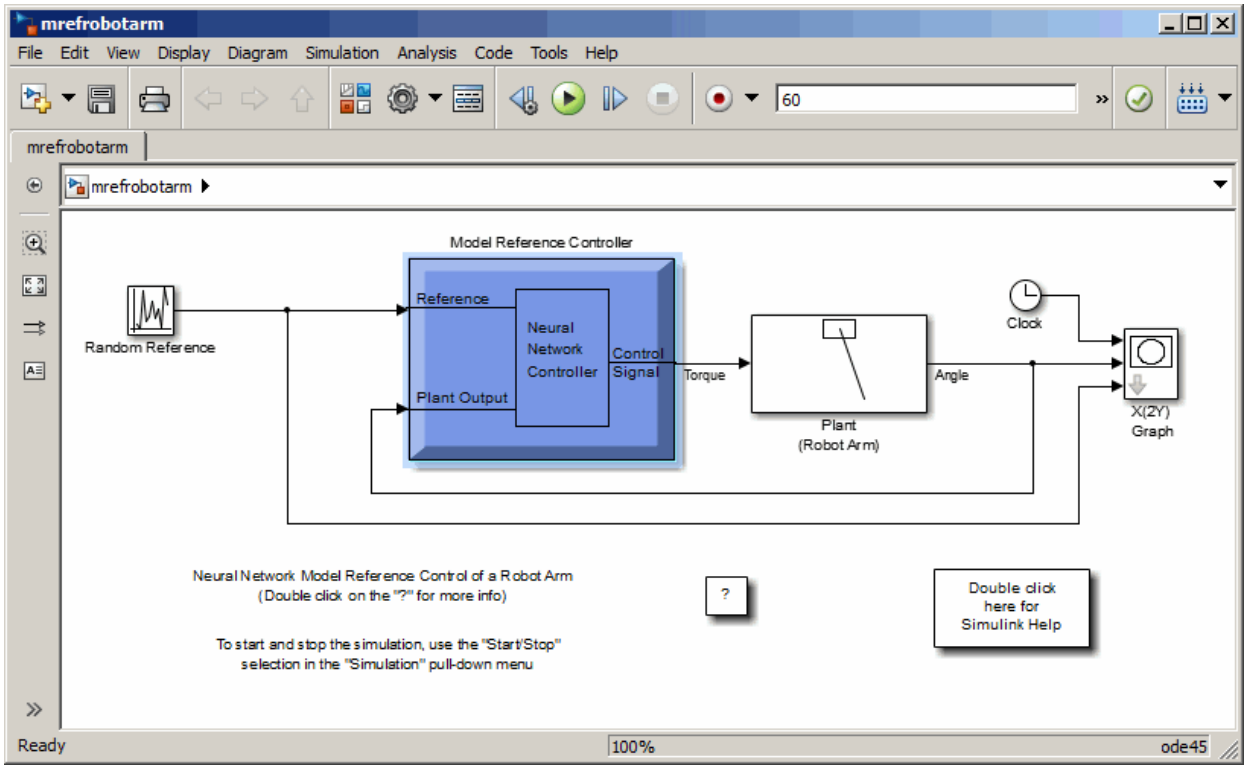
$$\frac{d^2y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r$$

where y_r is the output of the reference model, and r is the input reference signal.

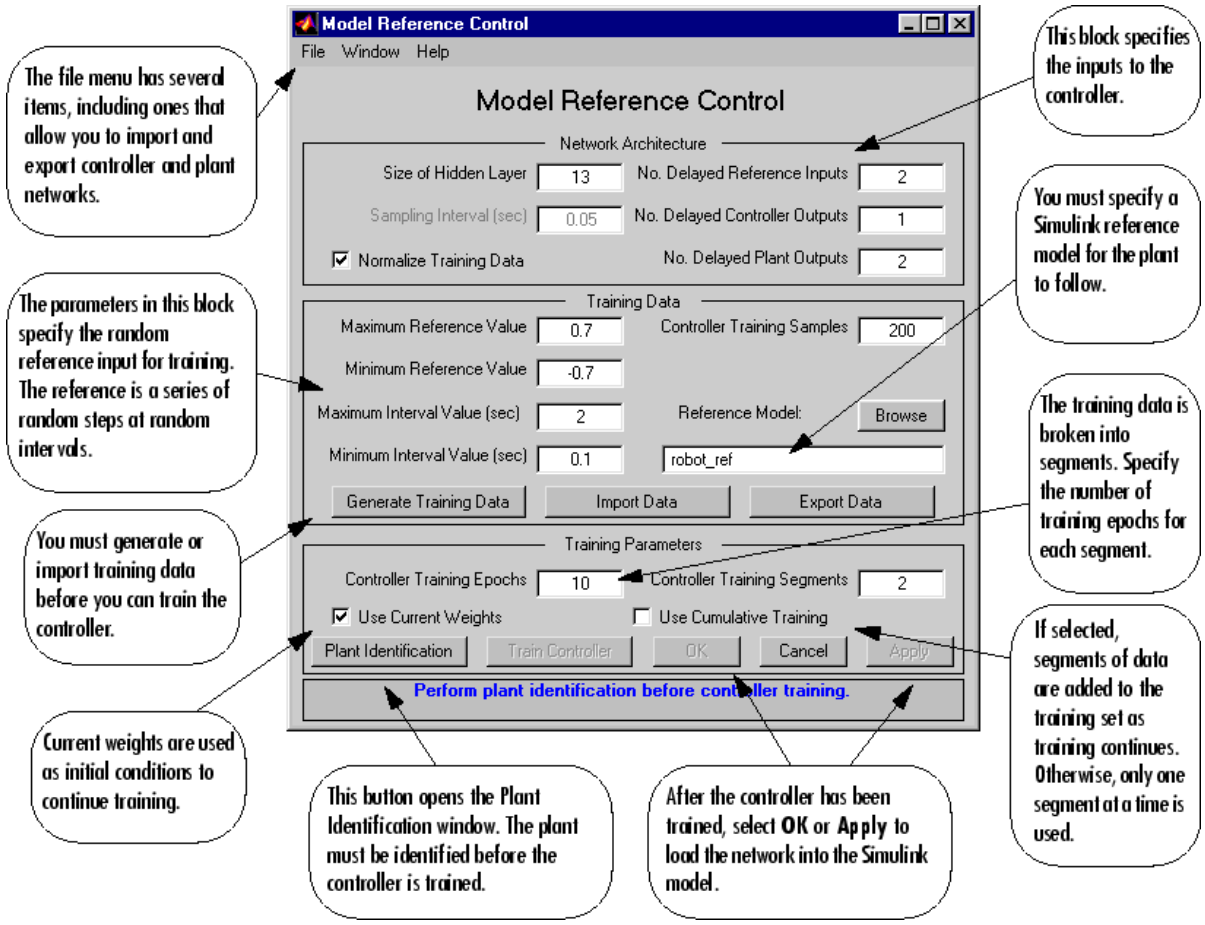
This example uses a neural network controller with a 5-13-1 architecture. The inputs to the controller consist of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. A sampling interval of 0.05 seconds is used.

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.

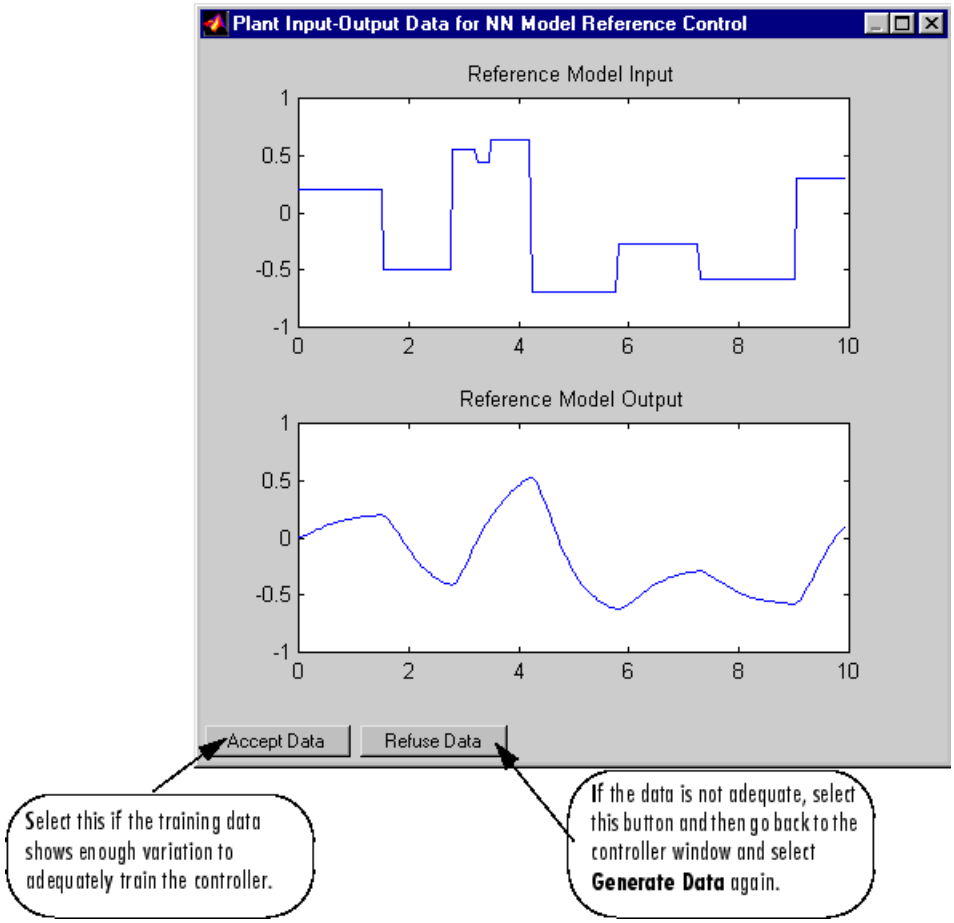


3. Double-click the Model Reference Control block. This opens the following window for training the model reference controller.

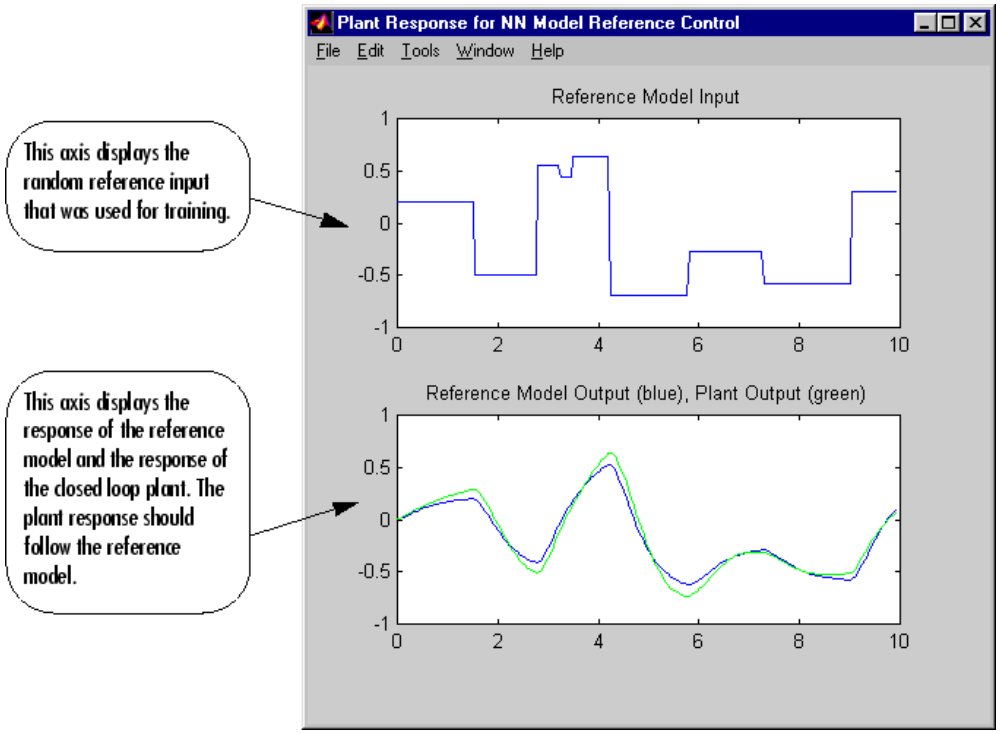


4 The next step would normally be to click **Plant Identification**, which opens the Plant Identification window. You would then train the plant model. Because the Plant Identification window is identical to the one used with the previous controllers, that process is omitted here.

5 Click **Generate Training Data**. The program starts generating the data for training the controller. After the data is generated, the following window appears.

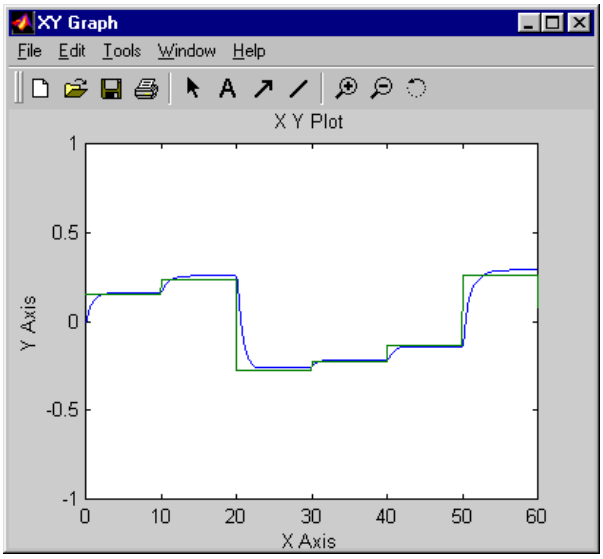


6 Click **Accept Data**. Return to the Model Reference Control window and click **Train Controller**. The program presents one segment of data to the network and trains the network for a specified number of iterations (five in this case). This process continues, one segment at a time, until the entire 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 29-2]). 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
Y Axis
X Axis
1
0.5
0
-0.5
-1
0
10
20
30
40
50
60

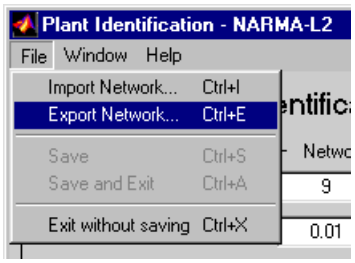
Import-Export Neural Network Simulink Control Systems

In this section...

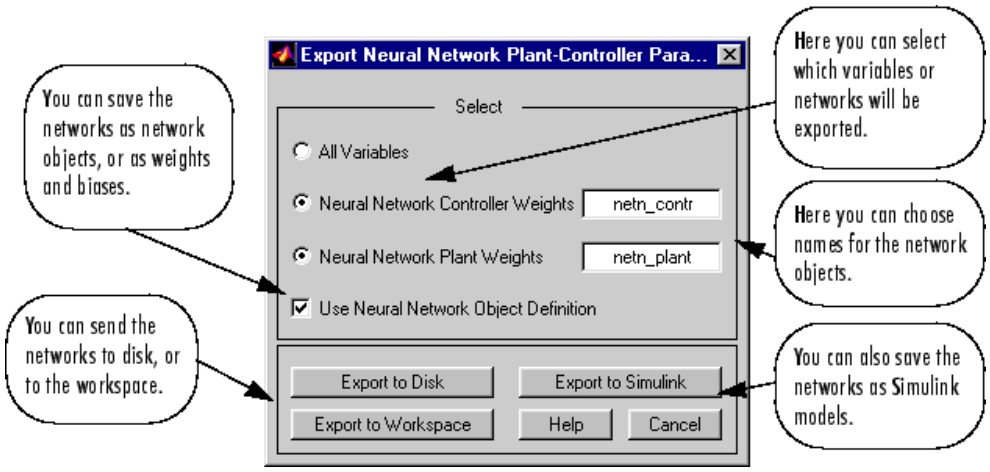
Import and Export Networks

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.)

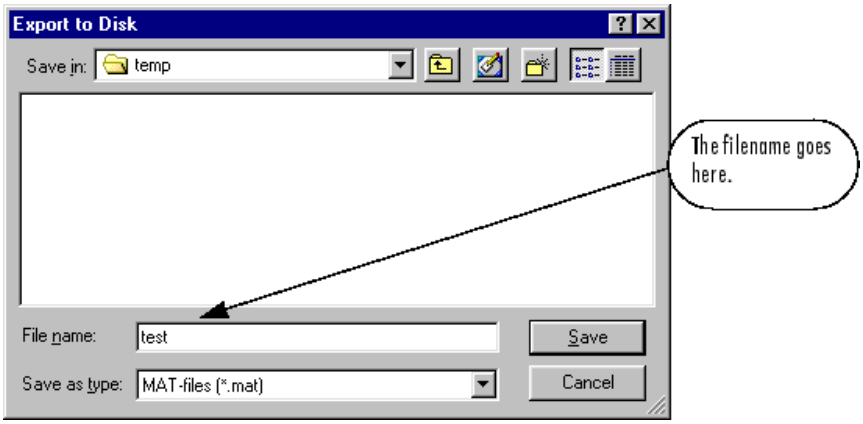
1. Repeat the first three steps of the NARMA-L2 example in “Use the NARMA-L2 Controller Block” on page 21-15. The NARMA-L2 Plant Identification window should now be open.
2. Select **File > Export Network**, as shown below.



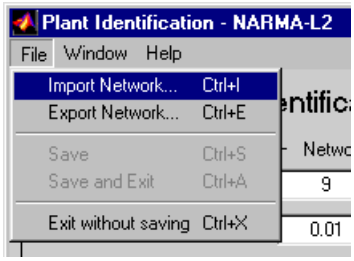
This opens the following window.



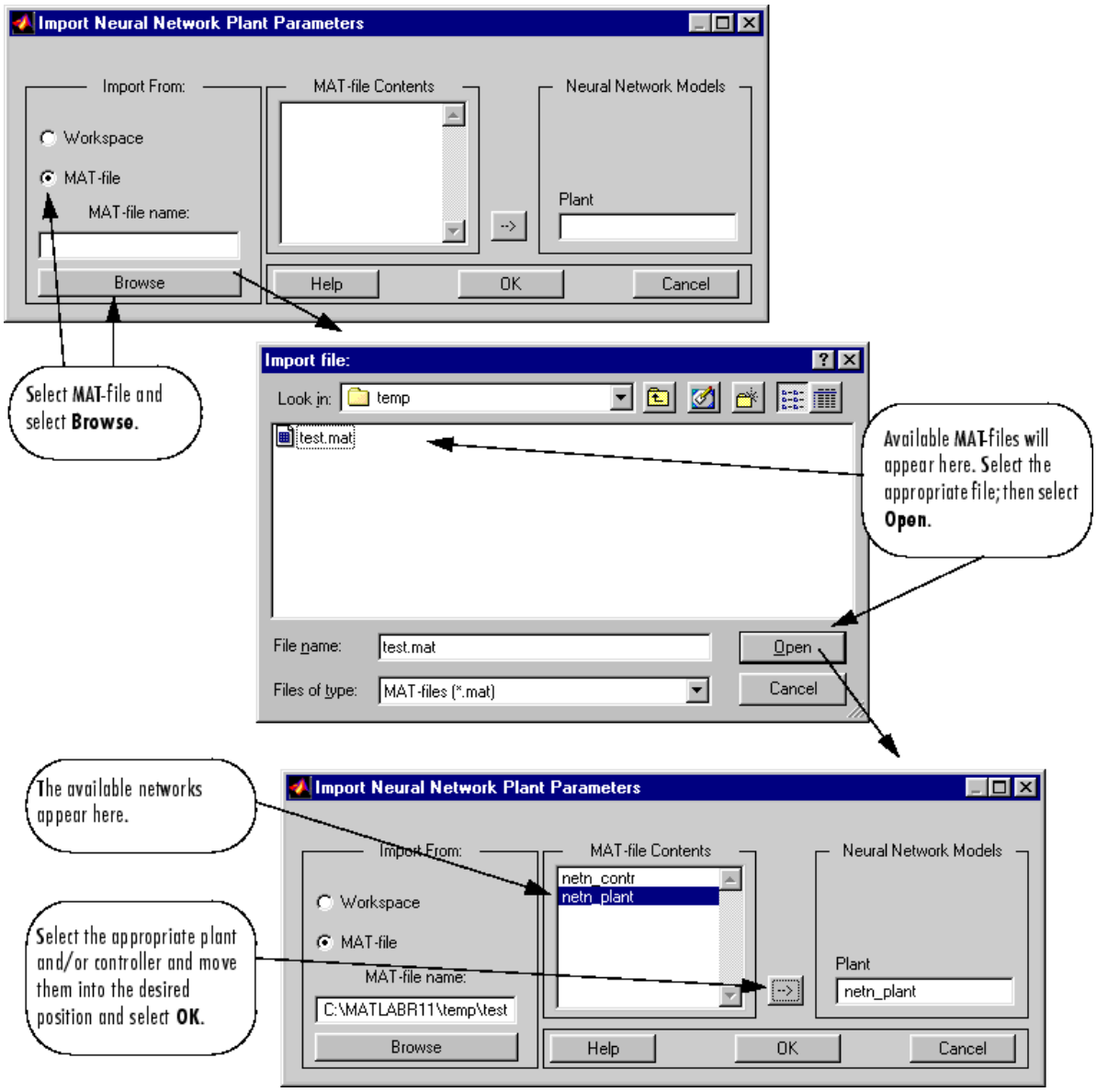
3. Select **Export to Disk**. The following window opens. Enter the file name `test` in the box, and select **Save**. This saves the controller and plant networks to disk.



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.

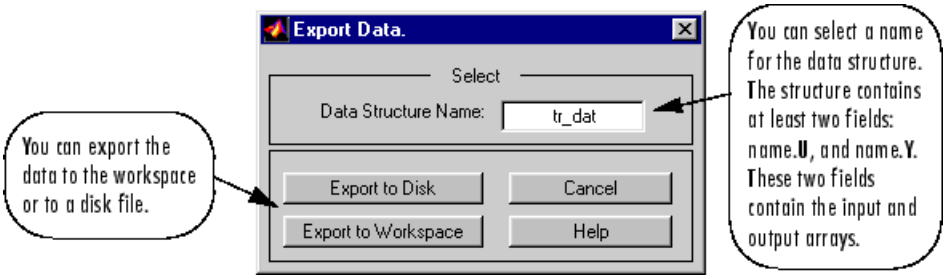


Import and Export Training Data

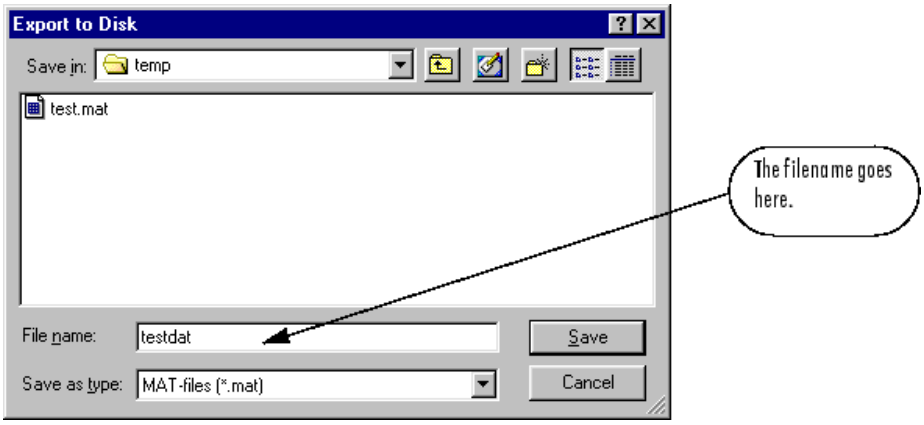
The data that you generate to train networks exists only in the corresponding plant identification or controller training window. You might want to save the training data to the workspace or to a disk file so that you can load it again at a later time. You might also want to combine data sets manually 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 21-6. Then select **Accept Data**. The Plant Identification window should then be open, and the **Import** and **Export** buttons should be active.

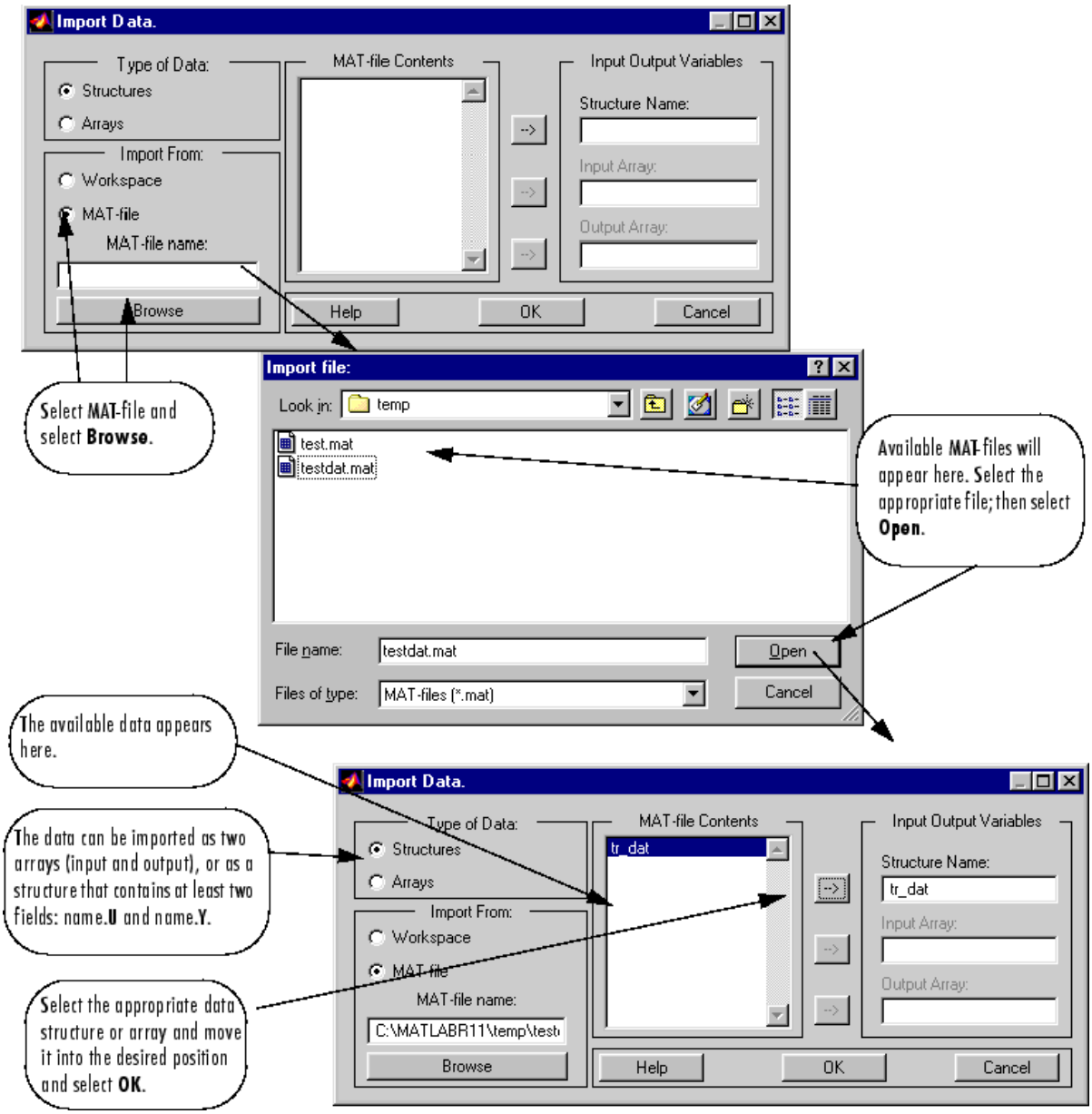
2 Click **Export** to open the following window.



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

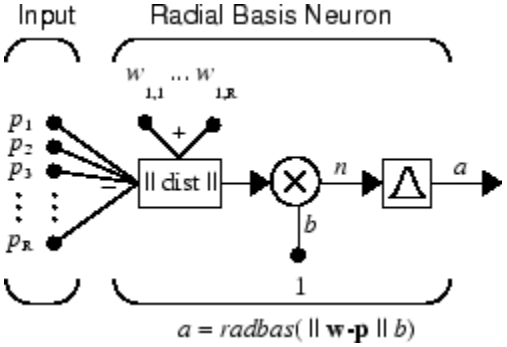
Radial basis networks can be designed with either `newrbe` or `newrb`. GRNNs and PNNs can be designed with `newgrnn` and `newpnn`, respectively.

Radial Basis Neural Networks

In this section...

Neuron Model

Here is a radial basis network with R inputs.

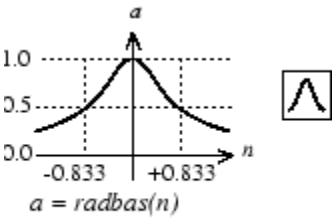


Notice that the expression for the net input of a `radbas` neuron is different from that of other neurons. Here the net input to the `radbas` transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The `|| dist ||` box in this figure 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.



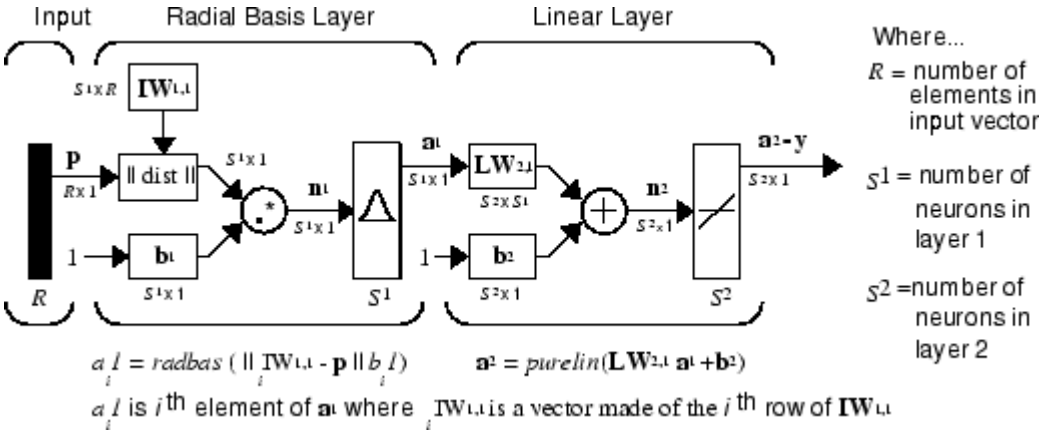
Radial Basis Function

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 28-79 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 28-83 and “Radial Basis Overlapping Neurons” on page 28-85 examine how the spread constant affects the design process for radial basis networks.

In “Radial Basis Underlapping Neurons” on page 28-83, a radial basis network is designed to solve the same problem as in “Radial Basis Approximation” on page 28-79. 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 28-83 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 28-85 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.

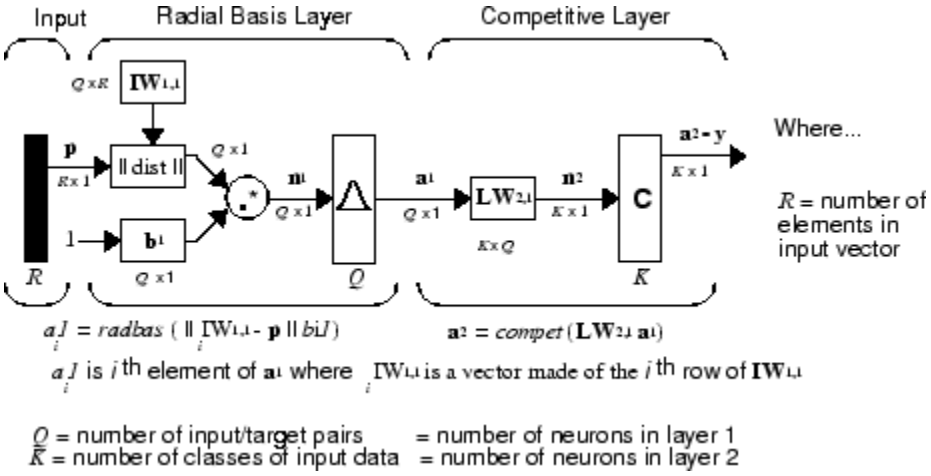
For this problem that would mean picking a spread constant greater than 0.1, the interval between inputs, and less than 2, the distance between the leftmost and rightmost inputs.

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 28-91. It shows how to design a PNN, and how the network can successfully classify a vector not used in the design.

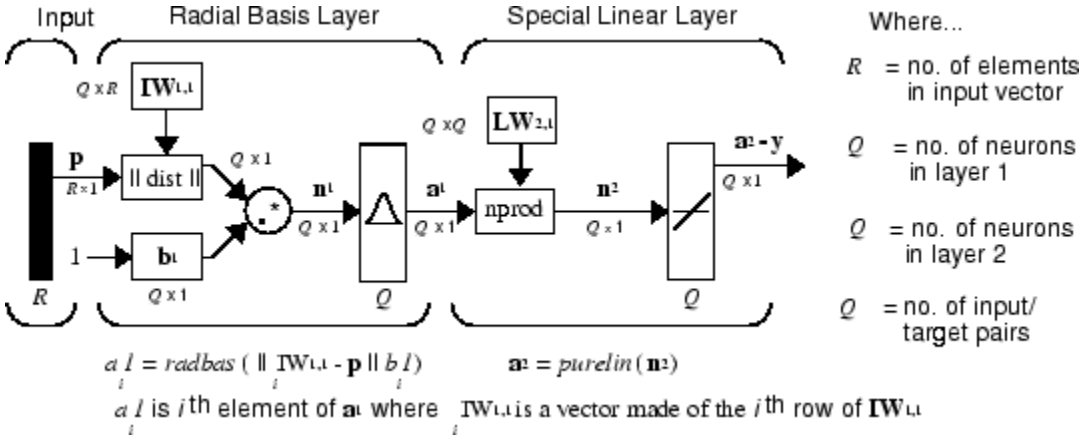
Generalized Regression Neural Networks

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 28-87 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`.

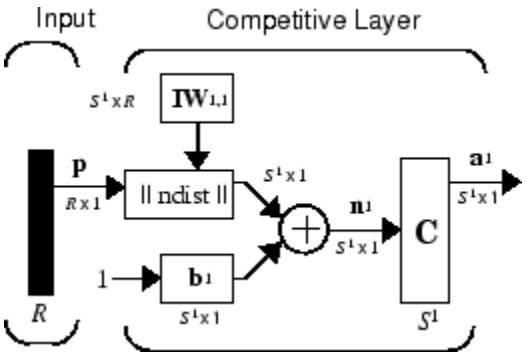
Cluster with a Competitive Neural Network

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 23-5.

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.

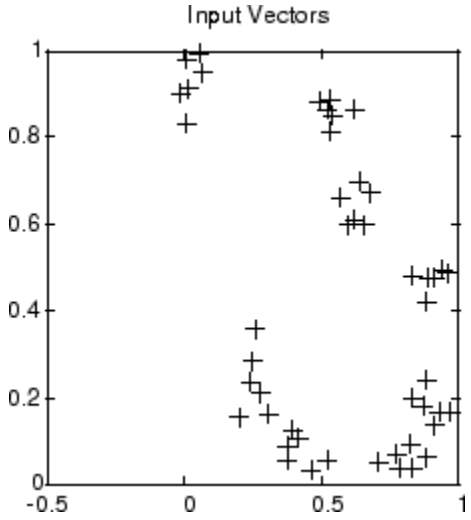
```
net.IW{1,1}

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 28-71 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 23-9 and “Distance Functions (dist, linkdist, mandist, boxdist)” on page 23-12.

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

$$_i\mathbf{w}(q) = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

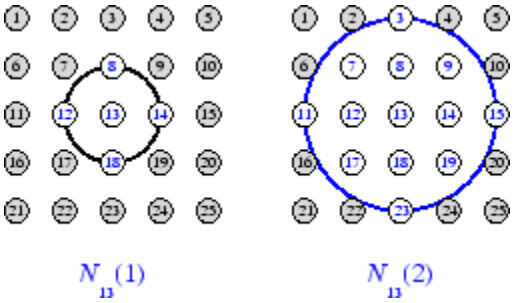
Here the *neighborhood* $N_{i^*}(d)$ contains the indices for all of the neurons that lie within a radius d of the winning neuron i^*.

$$N_i(d) = \left\{ j, d_{ij} \le d \right\}$$

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.

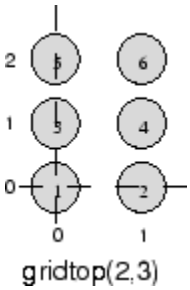
Topologies (gridtop, hextop, randtop)

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.

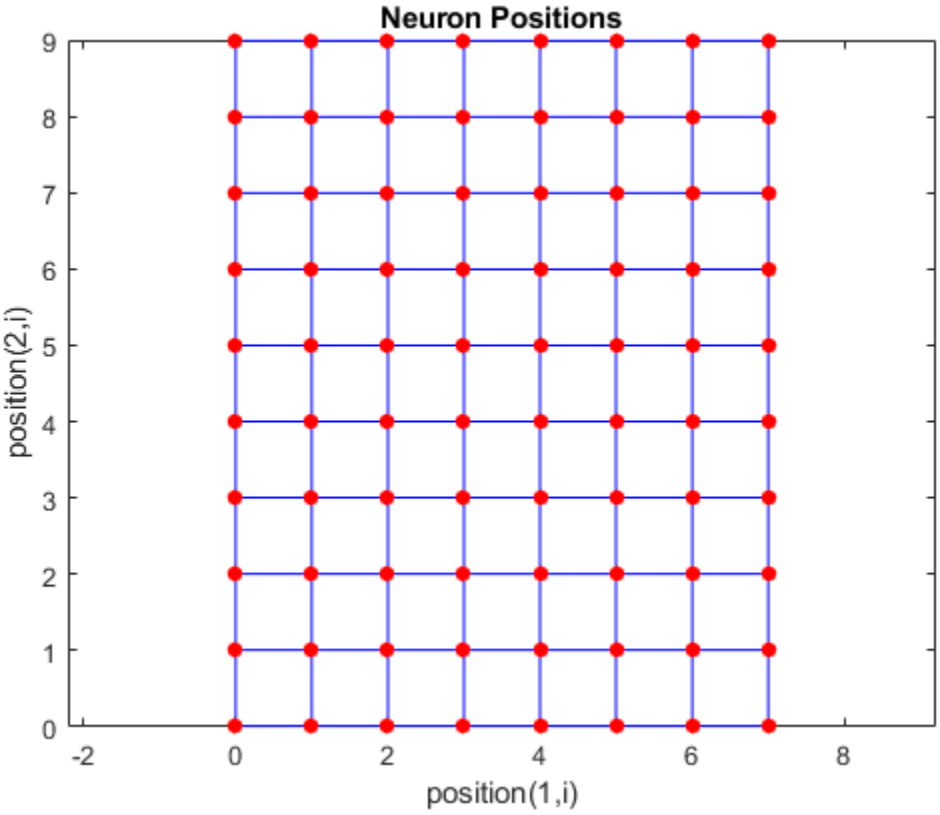


Note that had you asked for a `gridtop` with the dimension sizes reversed, you would have gotten a slightly different arrangement:

```
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.

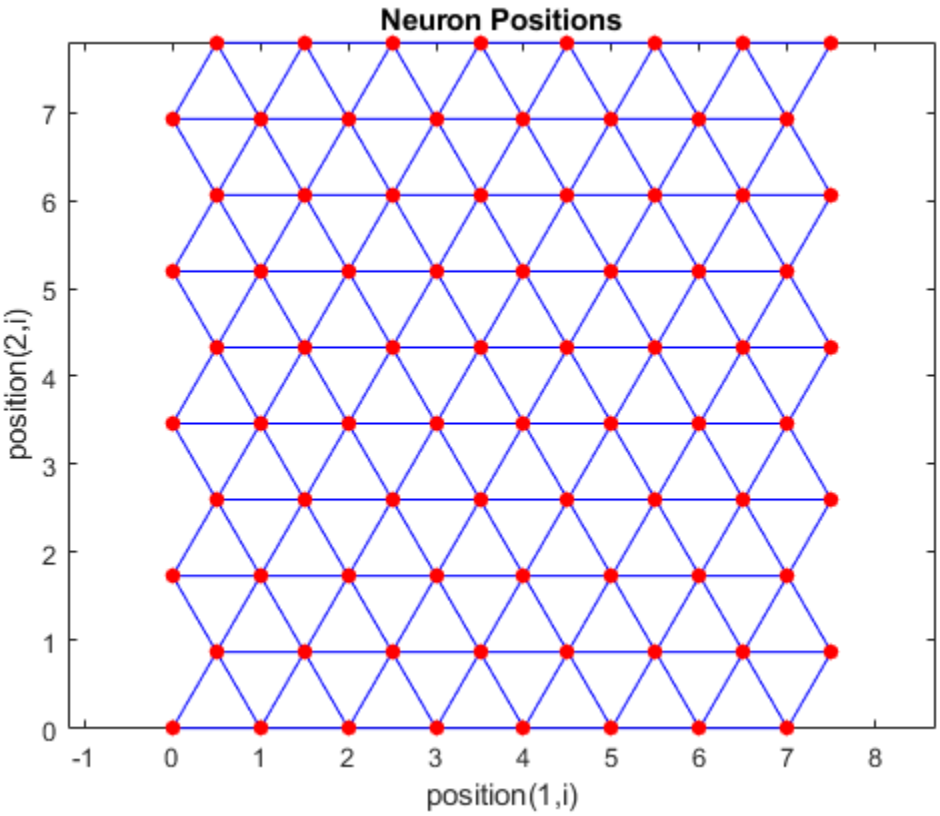
The `hextop` function creates a similar set of neurons, but they are in a hexagonal pattern. A 2-by-3 pattern of `hextop` neurons is generated as follows:

```
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)
```



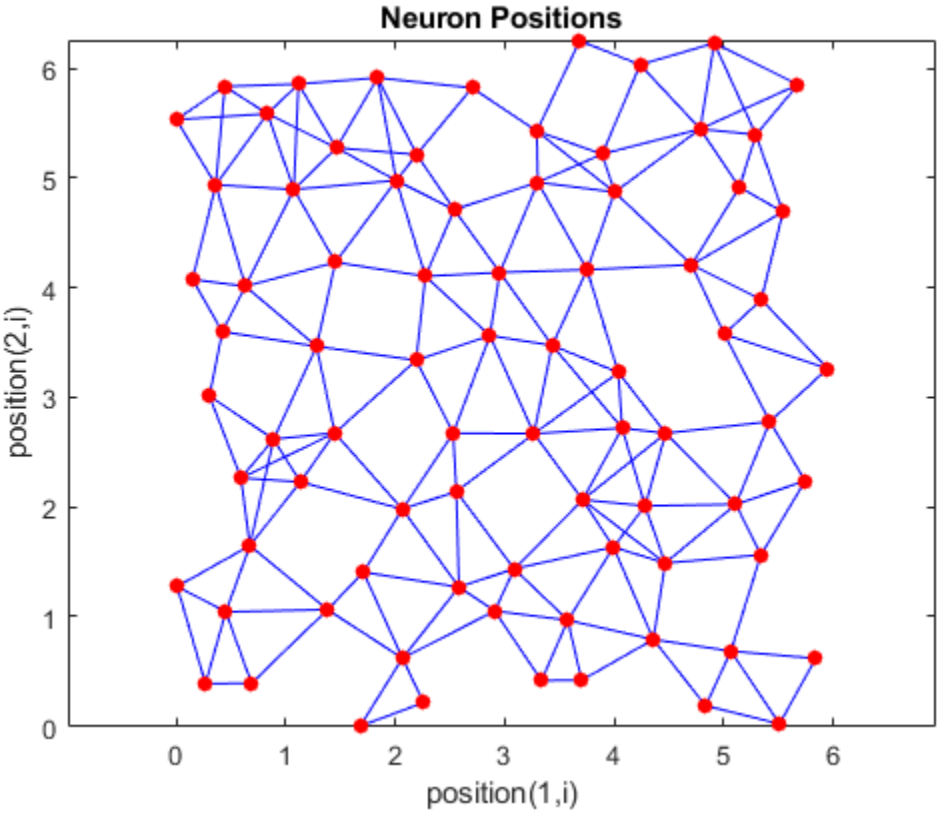
Note the positions of the neurons in a hexagonal arrangement.

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
```

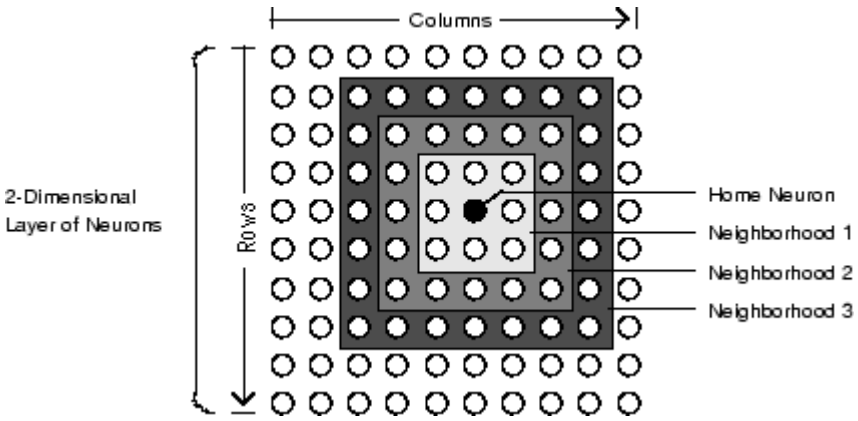
You find the distance from each neuron to the other with

```
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
```

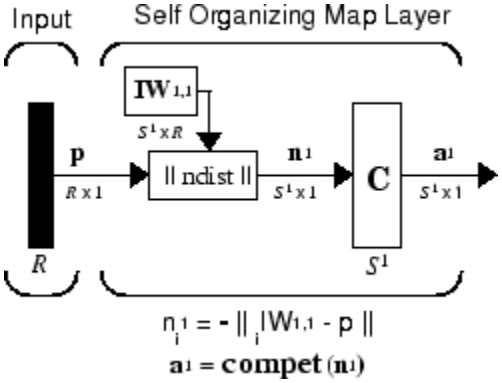
then you get for the distances

```
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:

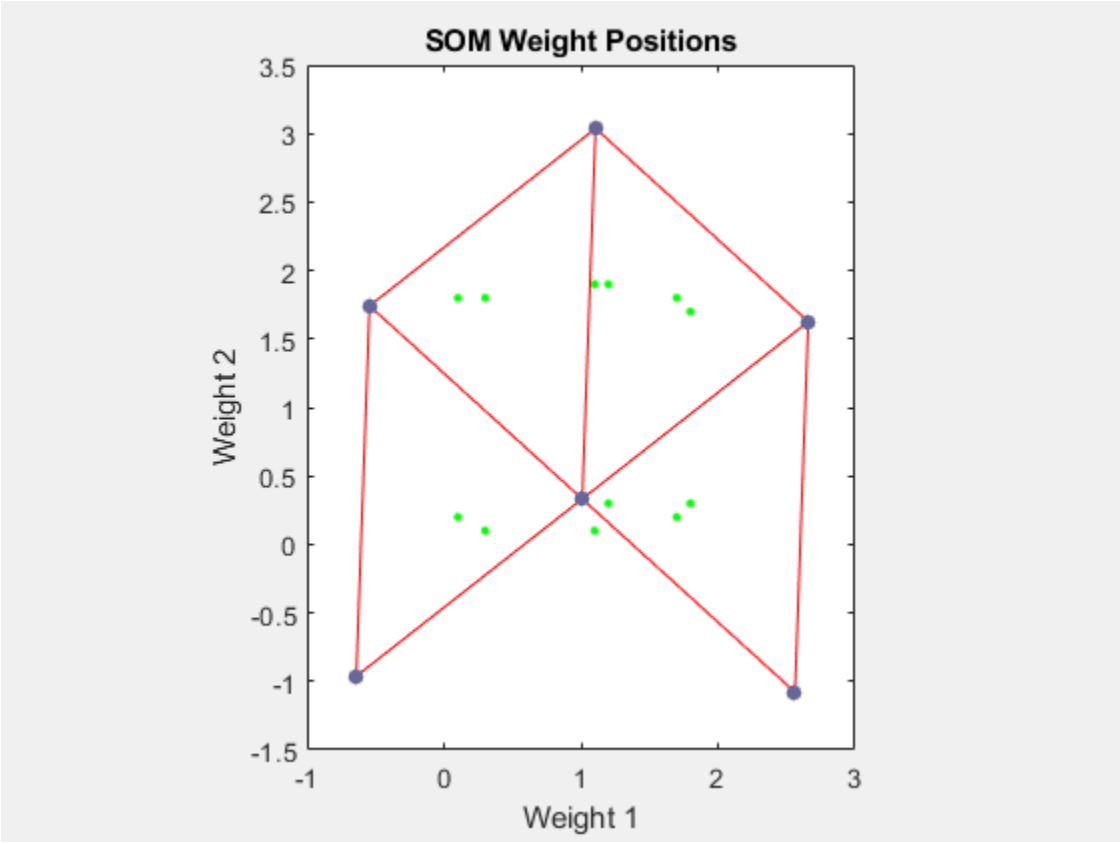
```
net = selforgmap([2 3]);
```

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 28-74 and “Two-Dimensional Self-organizing Map” on page 28-76.

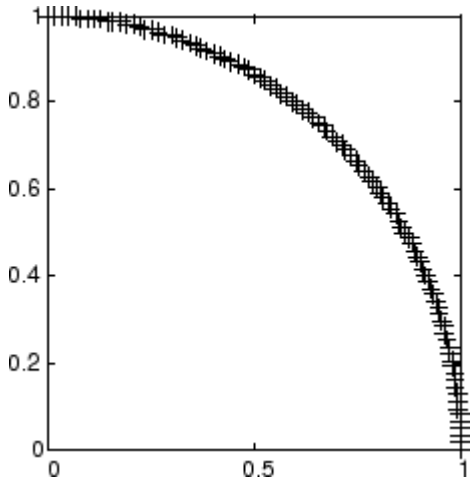
One-Dimensional Self-Organizing Map

Consider 100 two-element unit input vectors spread evenly between 0° and 90°.

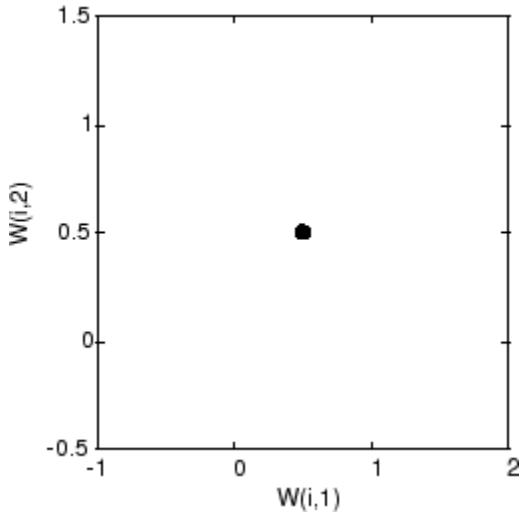
```
angles = 0:0.5*pi/99:0.5*pi;
```

Here is a plot of the data.

```
P = [sin(angles); cos(angles)];
```

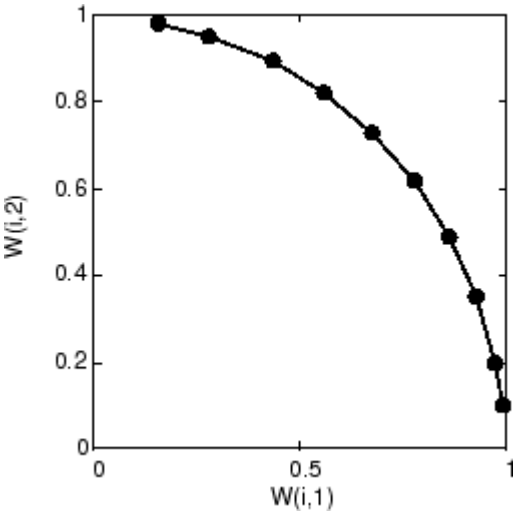


A self-organizing map is defined as a one-dimensional layer of 10 neurons. This map is to be trained on these input vectors shown above. Originally these neurons are at the center of the figure.



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.

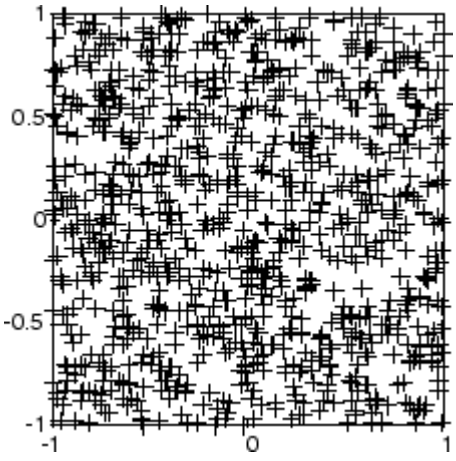
Two-Dimensional Self-Organizing Map

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);
```

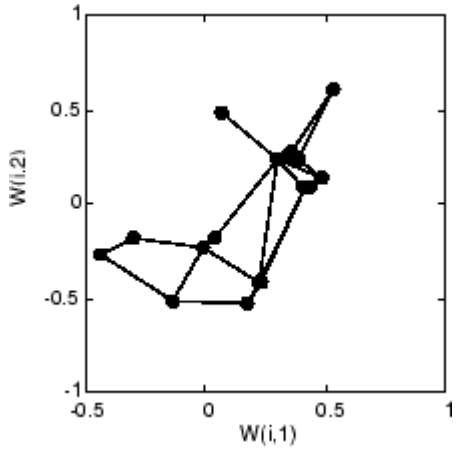
Here is a plot of these 1000 input vectors.



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`.

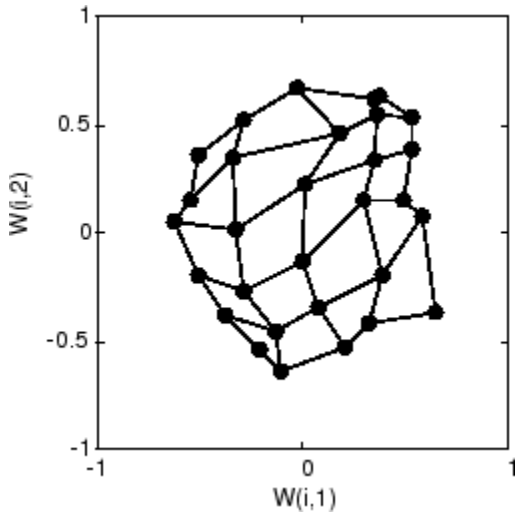
The map is then trained for 5000 presentation cycles, with displays every 20 cycles.

Here is what the self-organizing map looks like after 40 cycles.



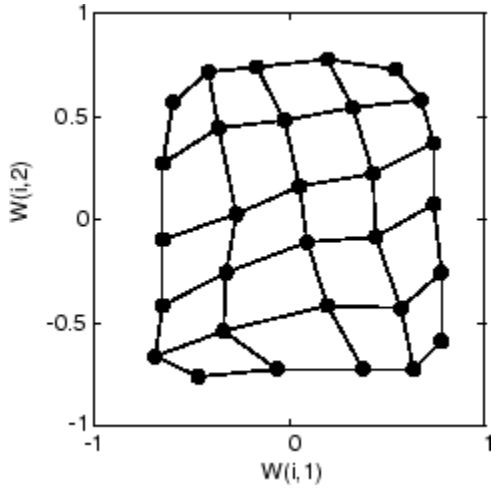
The weight vectors, shown with circles, are almost randomly placed. However, even after only 40 presentation cycles, neighboring neurons, connected by lines, have weight vectors close together.

Here is the map after 120 cycles.

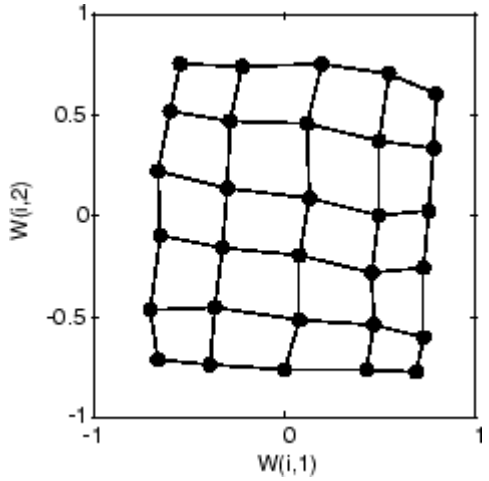


After 120 cycles, the map has begun to organize itself according to the topology of the input space, which constrains input vectors.

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.

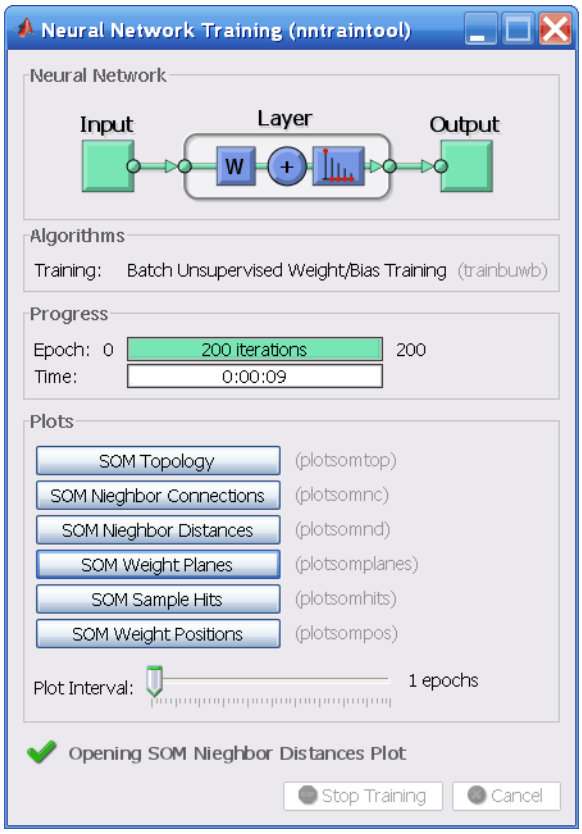
It is important to note that while a self-organizing map does not take long to organize itself so that neighboring neurons recognize similar inputs, it can take a long time for the map to finally arrange itself according to the distribution of input vectors.

Training with the Batch Algorithm

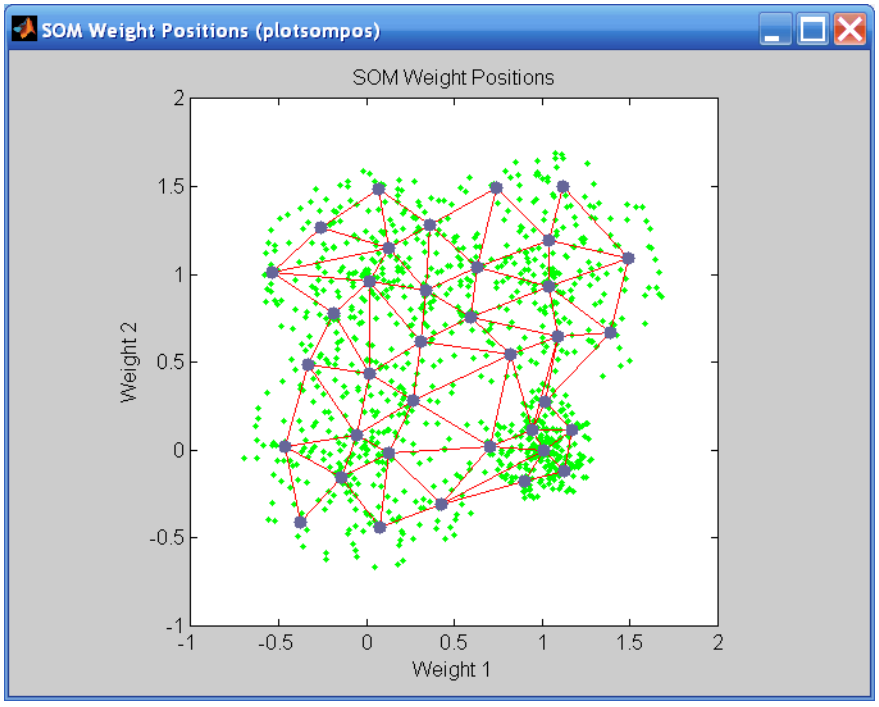
The batch training algorithm is generally much faster than the incremental algorithm, and it is the default algorithm for SOFM training. You can experiment with this algorithm on a simple data set with the following commands:

```
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.

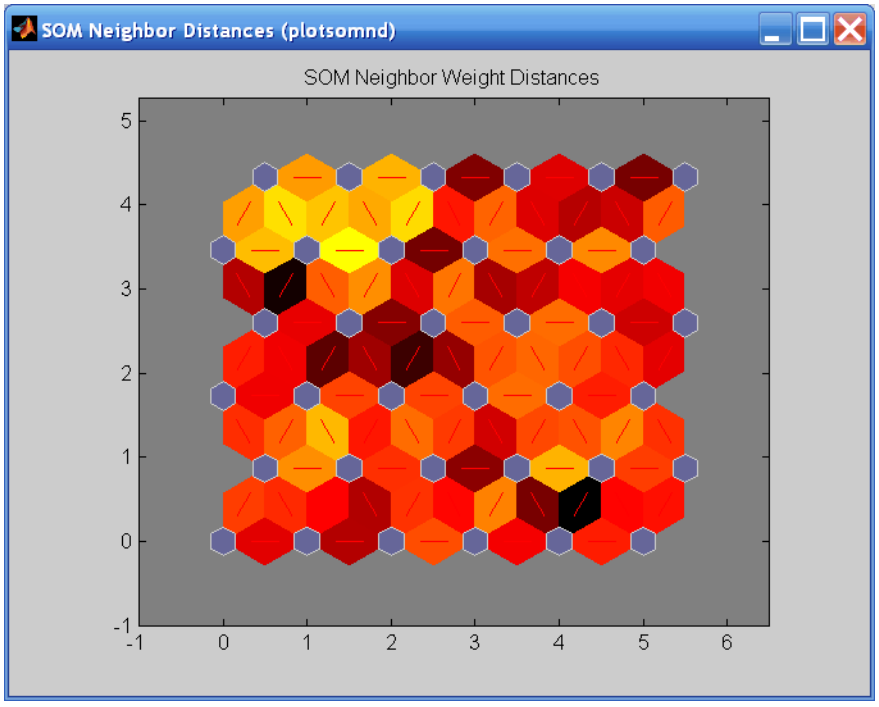


When the input space is high dimensional, you cannot visualize all the weights at the same time. In this case, click **SOM Neighbor Distances**. The following figure appears, which indicates the distances between neighboring neurons.

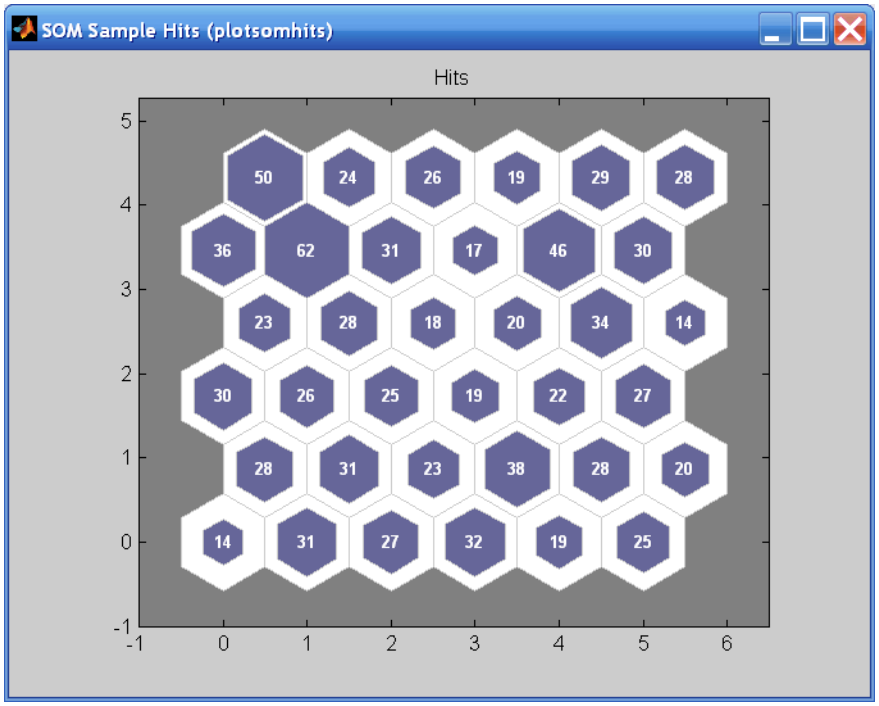
This figure uses the following color coding:

- 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.

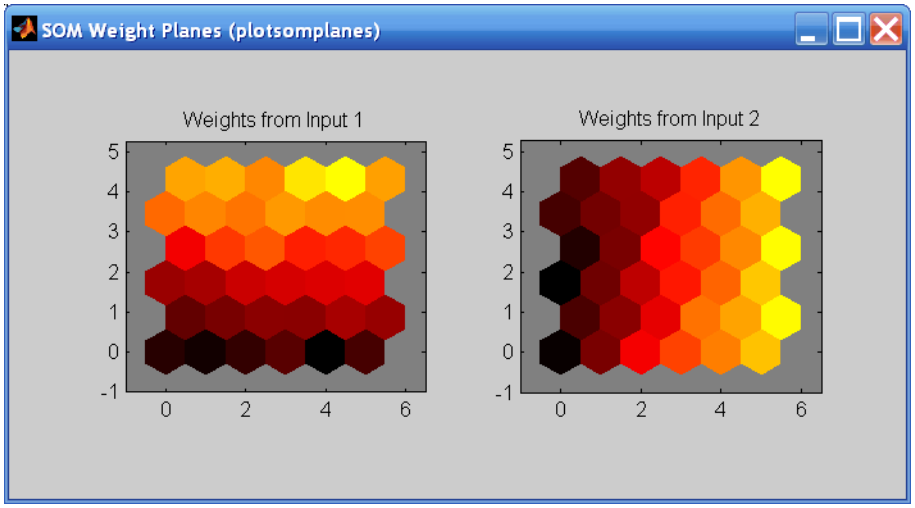
A group of light segments appear in the upper-left region, bounded by some darker segments. This grouping indicates that the network has clustered the data into two groups. These two groups can be seen in the previous weight position figure. The lower-right region of that figure contains a small group 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.



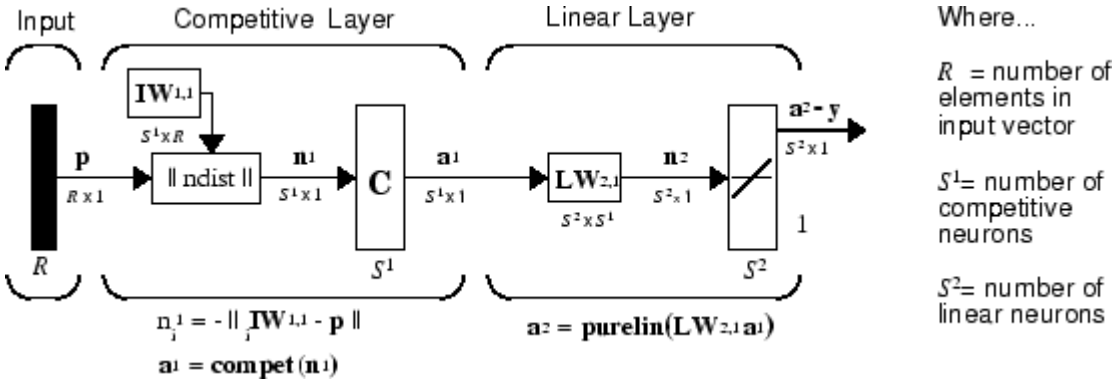
You can also produce all of the previous figures from the command line. Try these plotting commands: `plotsomhits`, `plotsomnc`, `plotsomnd`, `plotsomplanes`, `plotsompos`, and `plotsomtop`.

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 23-8 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 23-5. 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];
```

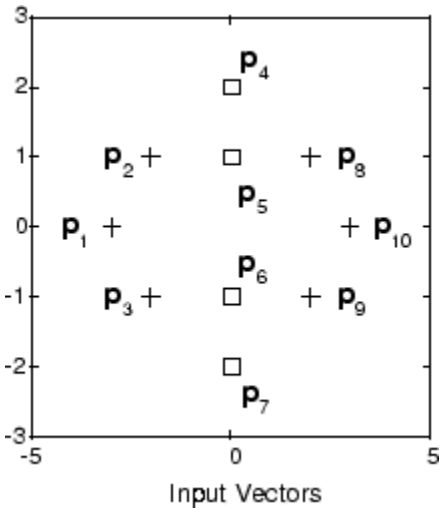
and

```
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

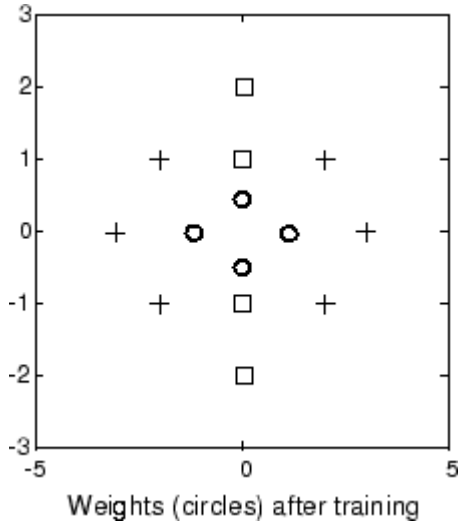
Next you need to train the network to obtain first-layer weights that lead to the correct classification of input vectors. You do this with `train` as with the following commands. First, set the training epochs to 150. Then, use `train`:

```
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 28-95. 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 29-2]) 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.

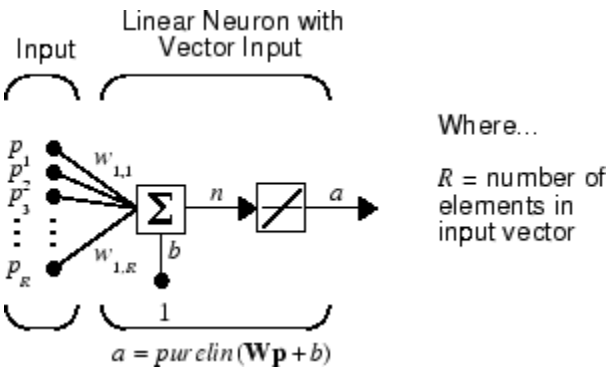
The adaptive training of self-organizing and competitive networks is also considered in this section.

Adaptive Functions

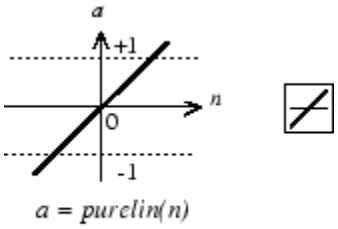
This section introduces the function `adapt`, which changes the weights and biases of a network incrementally during training.

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

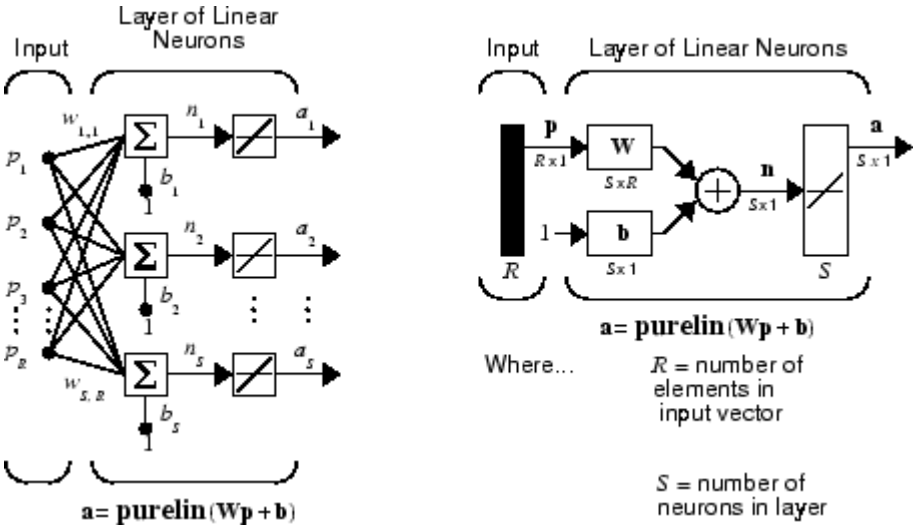
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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 **W**.

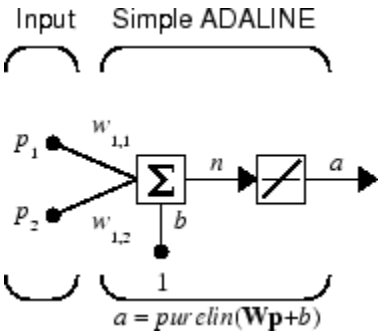


This network is sometimes called a MADALINE for Many ADALINEs. Note that the figure on the right defines an S-length output vector **a**.

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.



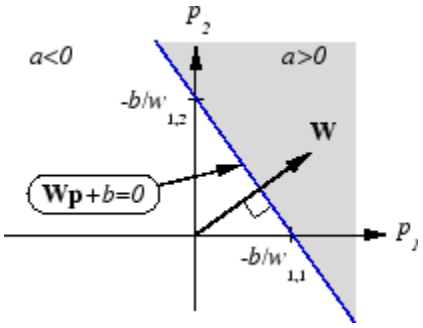
The weight matrix **W** in this case has only one row. The network output is

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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 29-2]).



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 29-2].

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 26-14.

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

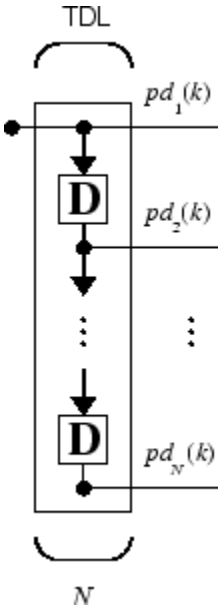
$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

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.

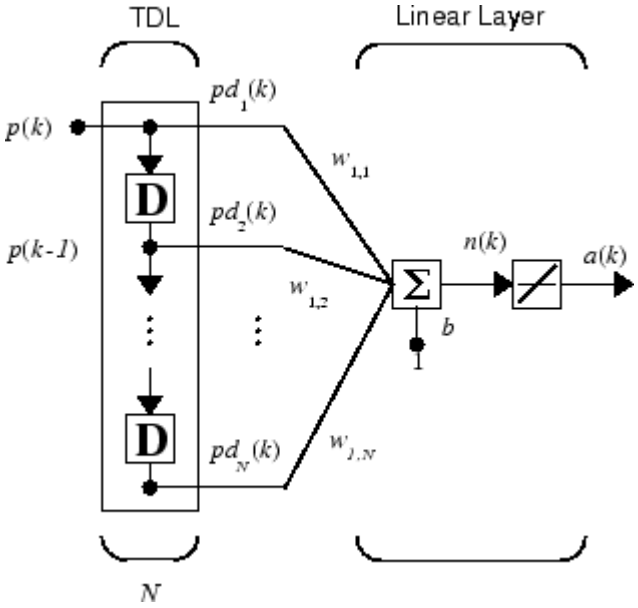
Tapped Delay Line

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.



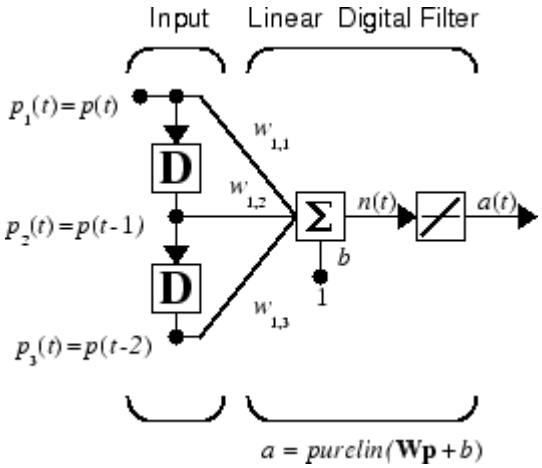
The output of the filter is given by

$$a(k) = purelin(\mathbf{Wp} + b) = \sum_{i=1}^{R} w_{1,i}a(k - i + 1) + b$$

In digital signal processing, this network is referred to as a *finite impulse response (FIR)* filter [WiSt85 on page 29-2]. 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
```

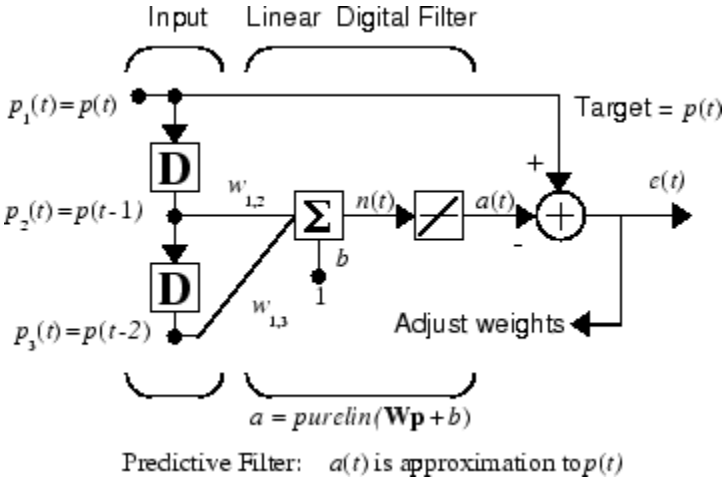
```
bias = net.b{1}
bias =
   -1.5993
y
y =
    [11.8558]    [20.7735]    [29.6679]    [39.0036]
```

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.

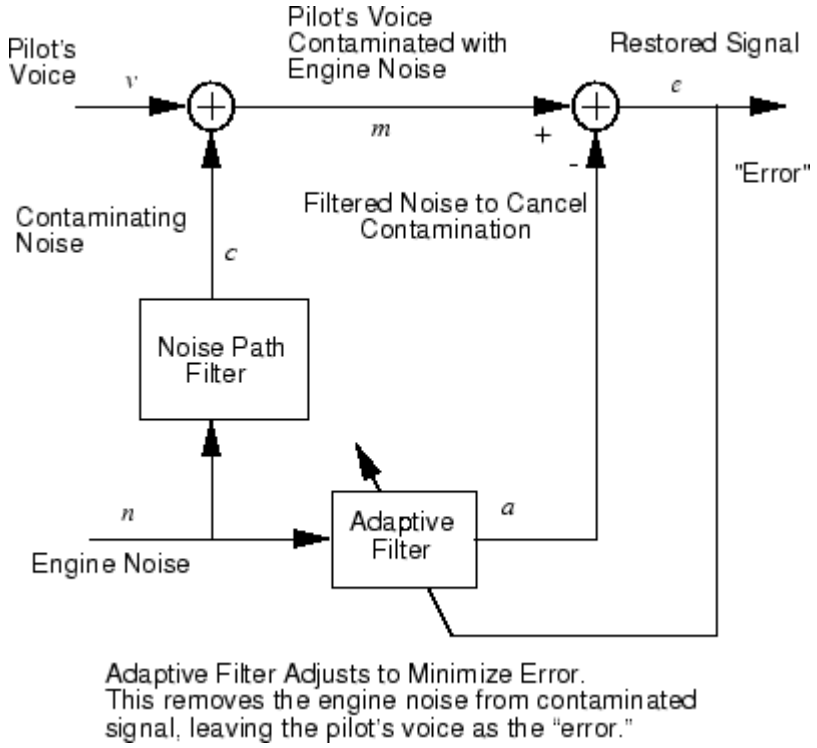
Given the autocorrelation function of the stationary random process $p(t)$, you can calculate the error surface, the maximum learning rate, and the optimum values of the weights. Commonly, of course, you do not have detailed information about the random process, so these calculations cannot be performed. 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 29-2] 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 19-2), 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*.

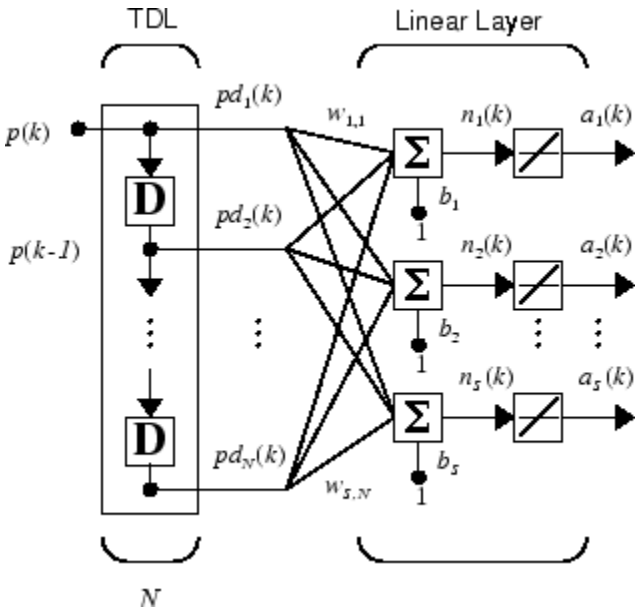
The network does its best to output *m* adaptively. In this case, the network can only predict the engine interference noise in the pilot/engine signal *m*. The network error *e* is equal to *m*, the pilot/engine signal, minus the predicted contaminating engine noise signal. Thus, *e* contains only the 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*.

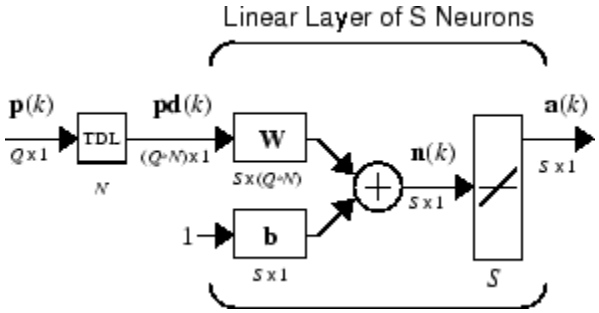
Try “Adaptive Noise Cancellation” on page 28-145 for an example of adaptive noise cancellation.

Multiple Neuron Adaptive Filters

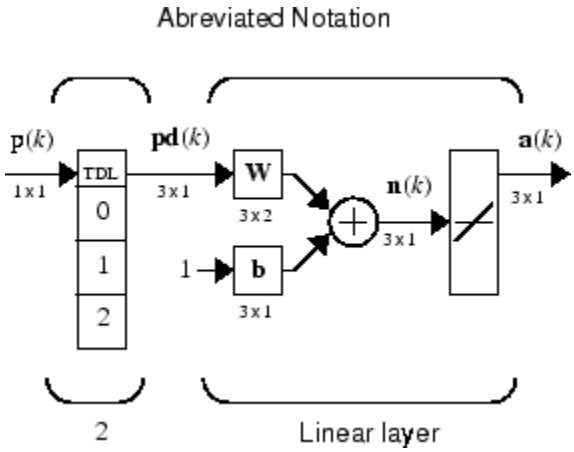
You might want to use more than one neuron in an adaptive system, so you need some additional notation. You can use a tapped delay line with *S* linear neurons, as shown in the next figure.



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

Neural Networks with Parallel and GPU Computing

In this section...

Deep Learning

You can train a convolutional neural network (CNN, ConvNet) or long short-term memory networks (LSTM or BiLSTM networks) using the `trainNetwork` function. You can 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);
```

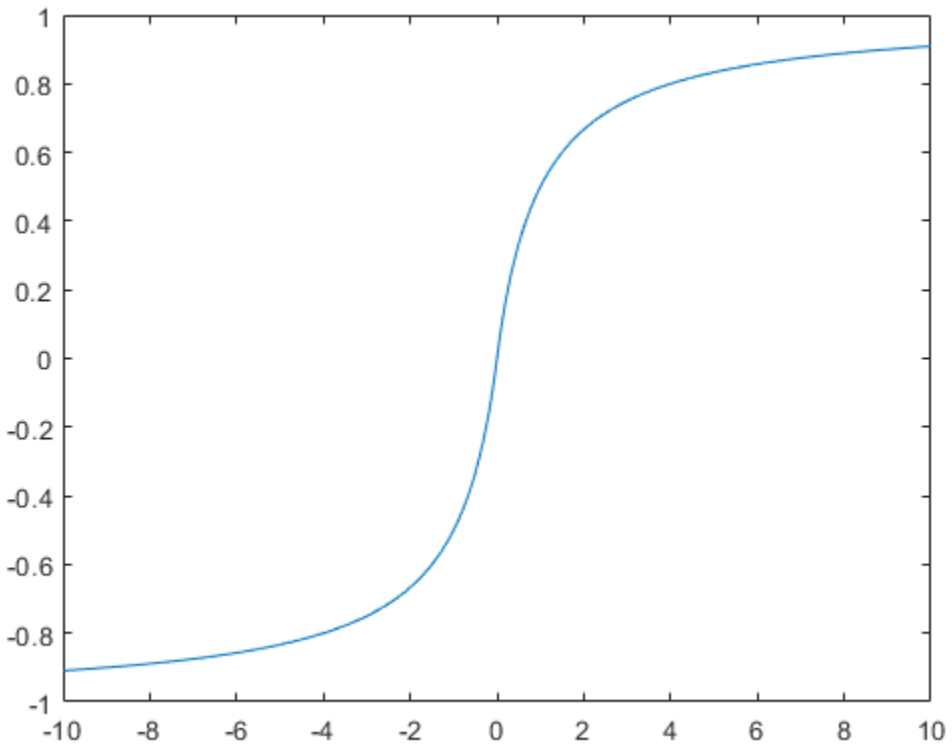
This is called memory reduction.

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.

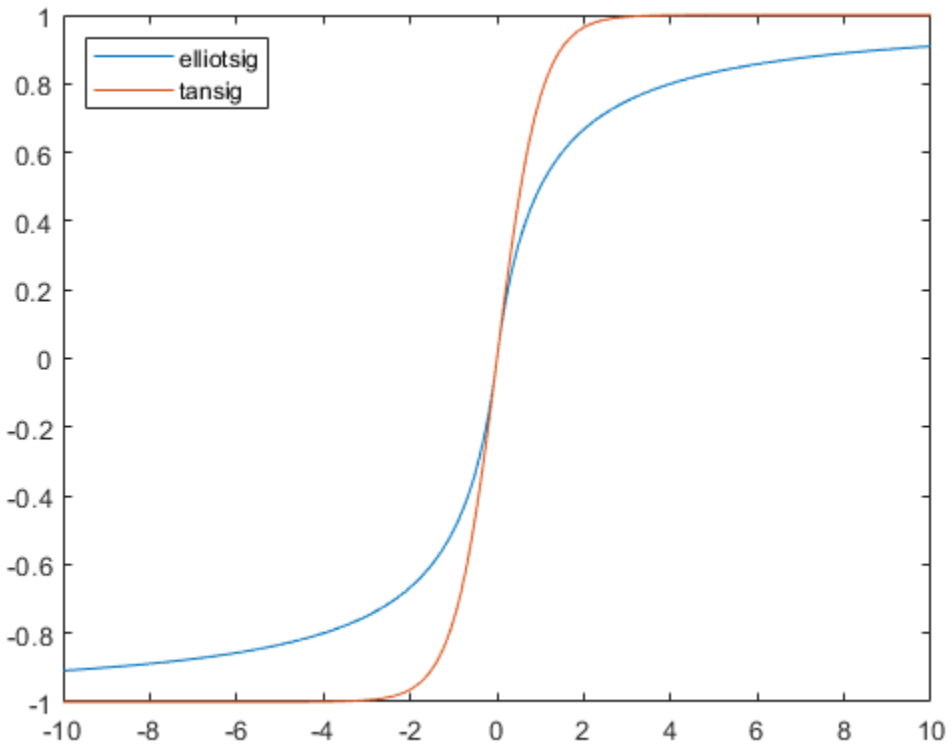
Here is a plot of the Elliot sigmoid:

```
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

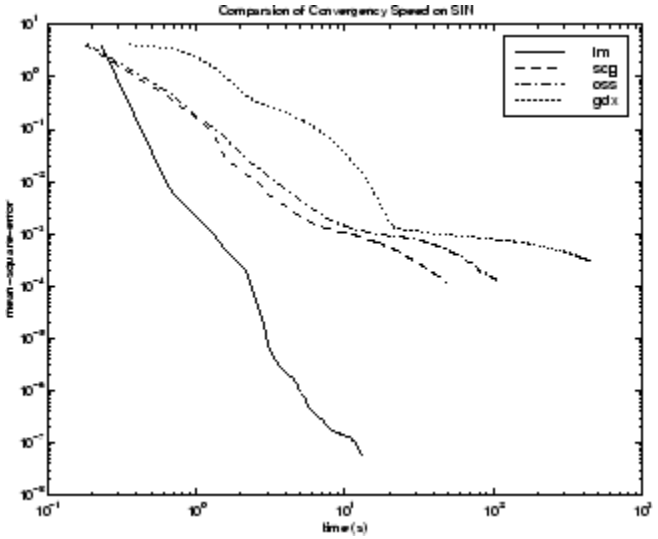
Problem Title	Problem Type	Network Structure	Error Goal	Computer
CANCER	Pattern recognition	9-5-5-2	0.012	Sun Sparc 2
CHOLESTEROL	Function approximation	21-15-3	0.027	Sun Sparc 20
DIABETES	Pattern recognition	8-15-15-2	0.05	Sun Sparc 20

SIN Data Set

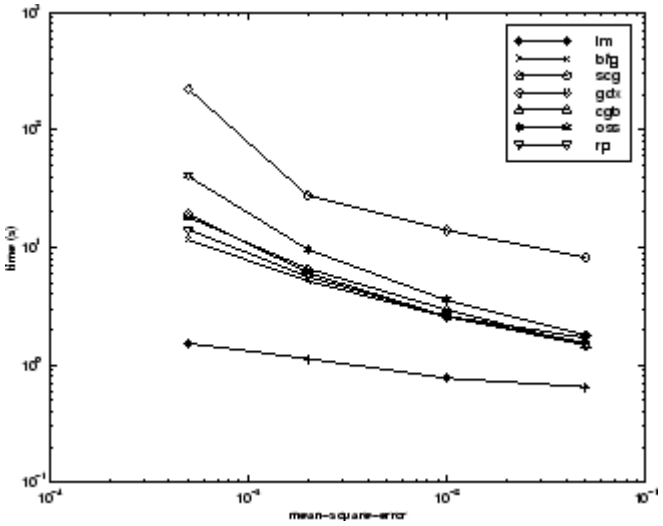
The first benchmark data set is a simple function approximation problem. A 1-5-1 network, with `tansig` transfer functions in the hidden layer and a linear transfer function in the output layer, is used to approximate a single period of a sine wave. The following table summarizes the results of training 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).



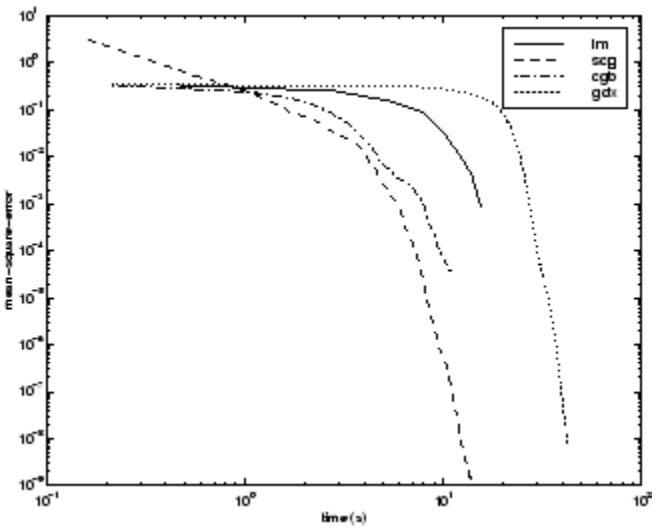
PARITY Data Set

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

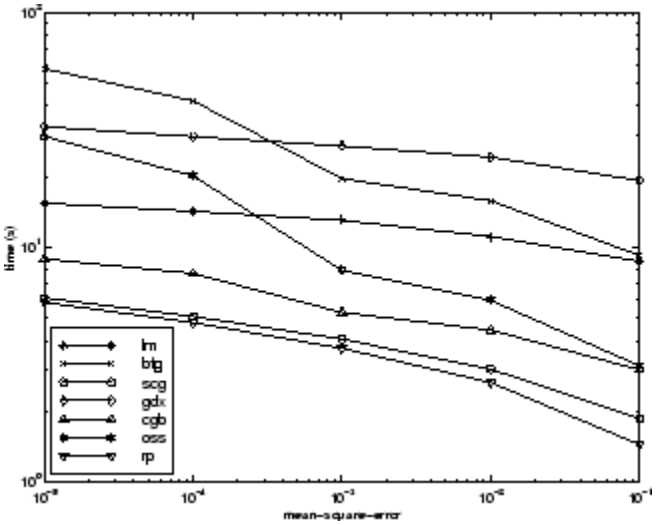
general, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems. The LM algorithm is designed for least squares problems that are approximately linear. Because the output neurons in pattern recognition problems are generally saturated, 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).

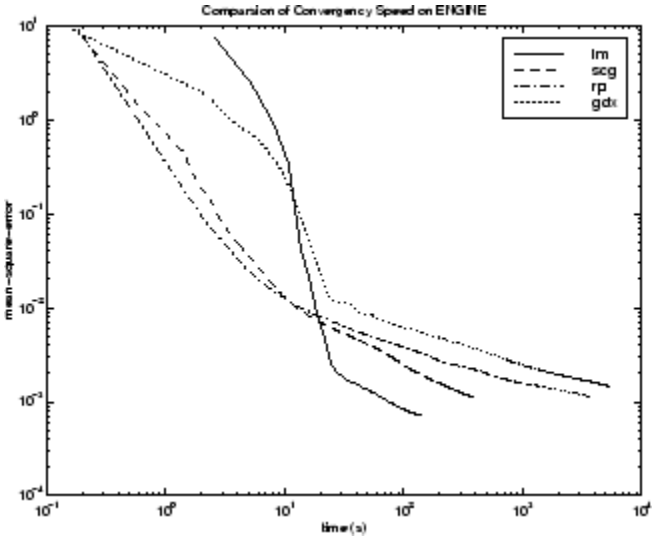


ENGINE Data Set

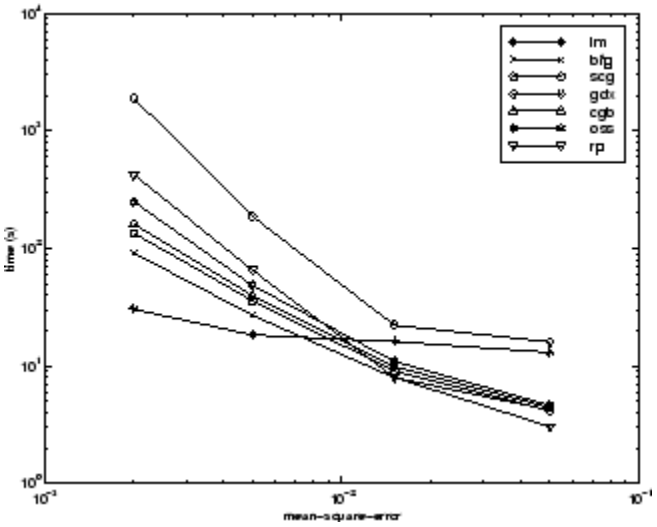
The third benchmark problem is a realistic function approximation (or nonlinear regression) problem. The data is obtained from the operation of an engine. The inputs to the network are engine speed and fueling levels and the network outputs are torque and emission levels. The network used for this problem 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

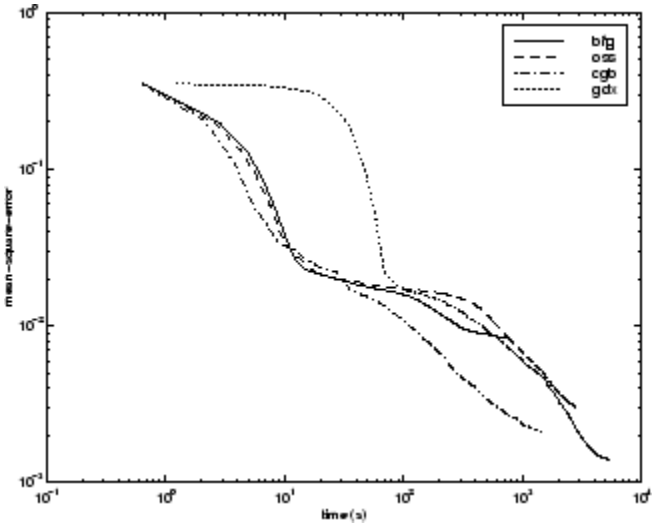
The fourth benchmark problem is a realistic pattern recognition (or nonlinear discriminant analysis) problem. The objective of the network is to classify a tumor as either benign or malignant based on cell descriptions gathered by microscopic examination. Input attributes include clump thickness, uniformity 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.

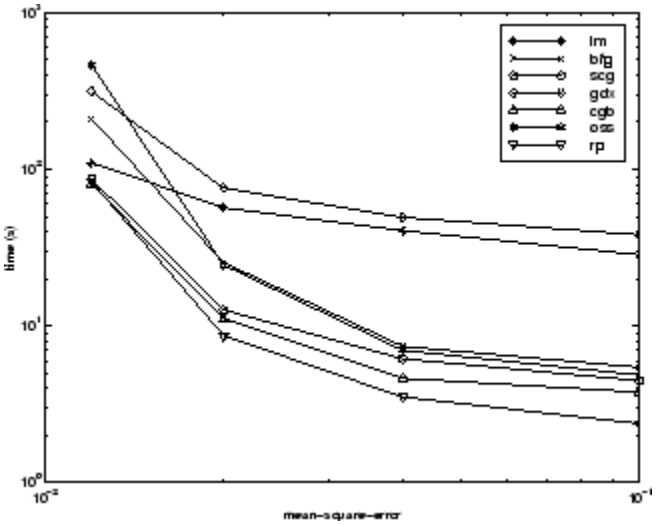
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence, and the LM algorithm is also reasonably fast. As with the parity data set, the LM algorithm does not perform as well on pattern recognition problems as it does on function approximation problems.

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.



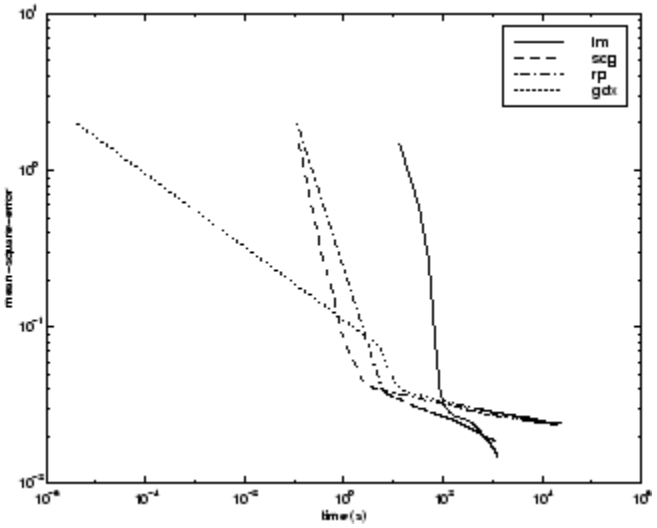
CHOLESTEROL Data Set

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 29-2]. 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.

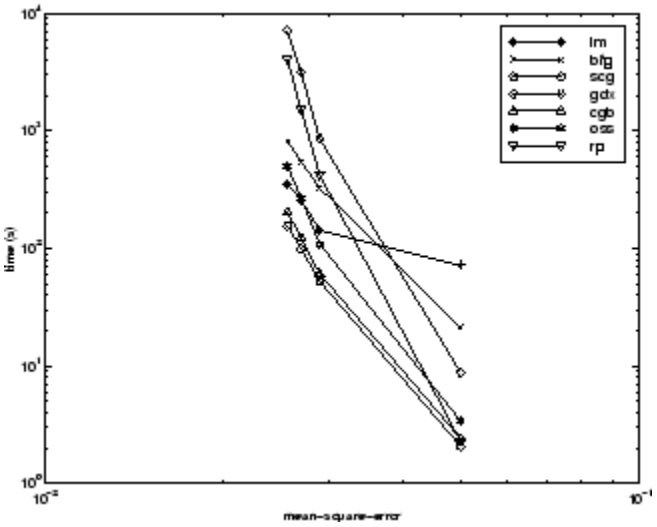
The scaled conjugate gradient algorithm has the best performance on this problem, although all the conjugate gradient algorithms perform well. The LM algorithm does not perform as well on this function approximation problem as it did on the other two. That is because the number of weights and biases 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

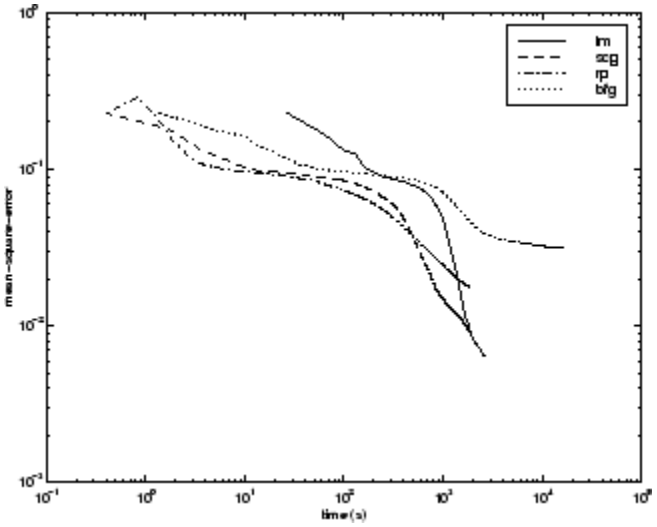
The sixth benchmark problem is a pattern recognition problem. The objective of the network is to decide whether an individual has diabetes, based on personal data (age, number of times pregnant) and the results of medical examinations (e.g., blood pressure, body mass index, result of glucose tolerance 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.

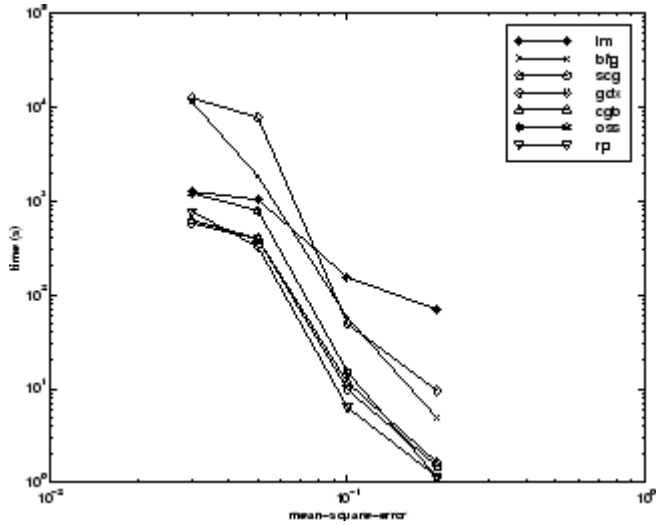
The conjugate gradient algorithms and resilient backpropagation all provide fast convergence. The results on this problem are consistent with the other pattern recognition problems considered. The RP algorithm works well on all the pattern recognition problems. This is reasonable, because that algorithm 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.

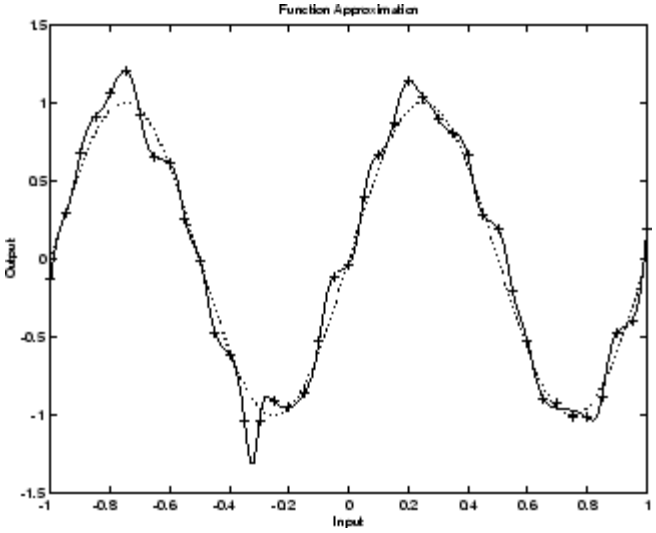
Improve Shallow Neural Network Generalization and Avoid Overfitting

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 1-41.

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 29-2] 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 29-2]. 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 29-2].

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 25-25. (Data division is cancelled by setting `net.divideFcn` so that the effects of `trainbr` are isolated from early stopping.)

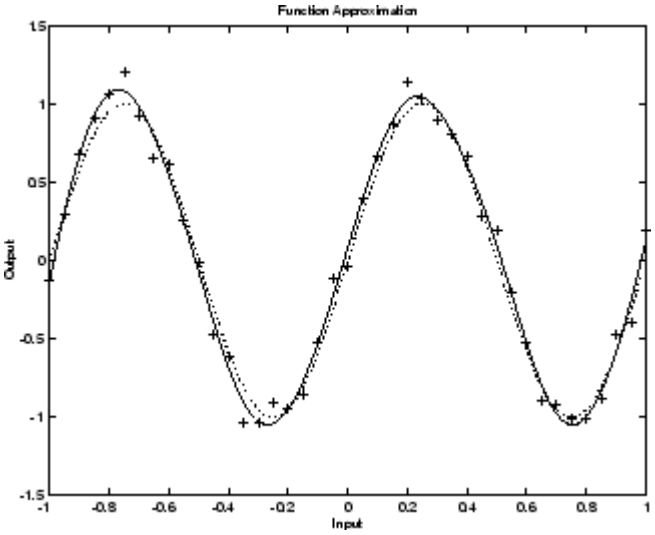
```
x = -1:0.05:1;
t = sin(2*pi*x) + 0.1*randn(size(x));
net = feedforwardnet(20,'trainbr');
net = train(net,x,t);
```

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 19-7. 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.

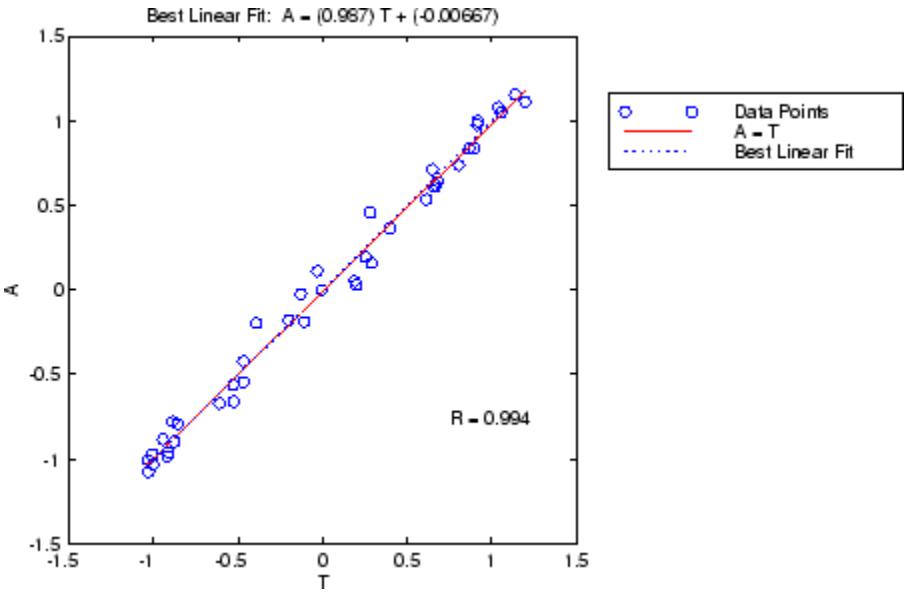
The following commands illustrate how to perform a regression analysis on a network trained.

```
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 15-2.

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.

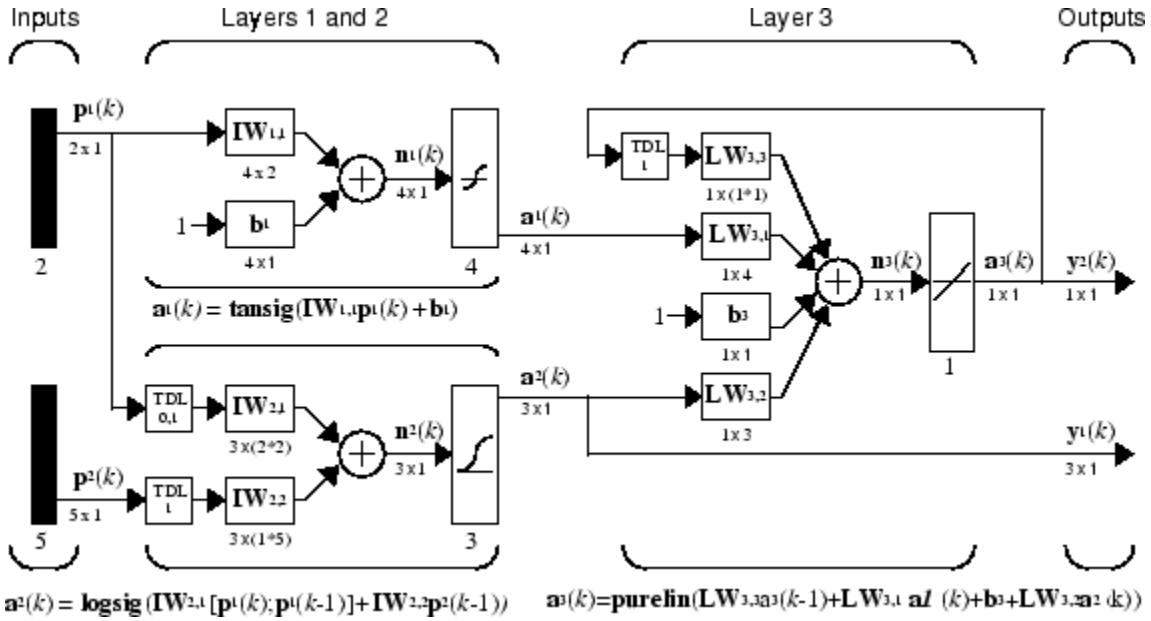
```
net = network
```

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

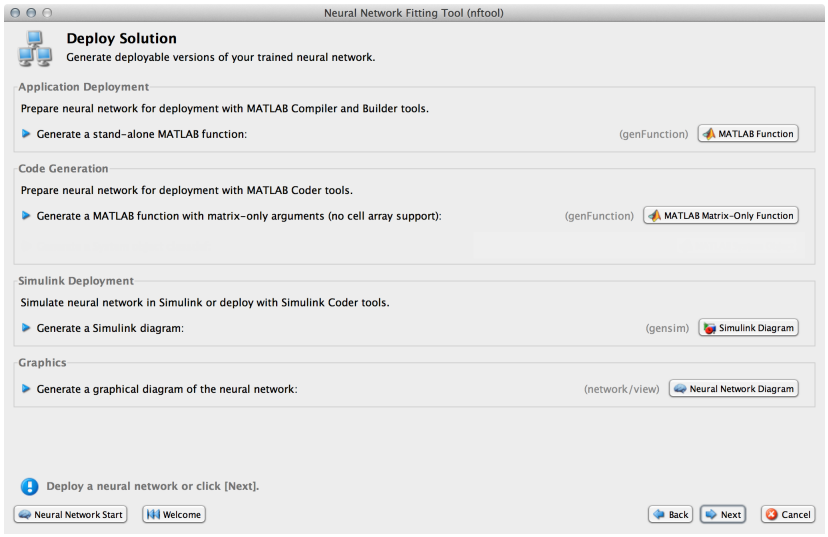
In this section...

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 introduced in the deployment panels in the tools `nftool`, `nctool`, `nprtool` and `ntstool`. For information on these tool features, 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 advanced scripts generated on the Save Results panel of each of these tools 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 25-51

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 Command Line” (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 25-48

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 29-2].

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

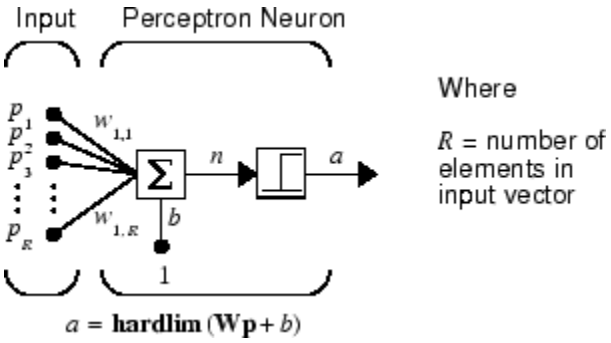
In this section...

Rosenblatt [Rose61 on page 29-2] 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.

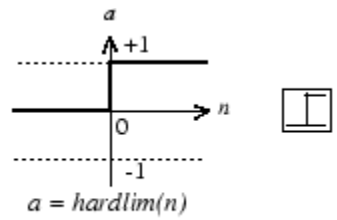
The discussion of perceptrons in this section is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996 on page 29-2], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



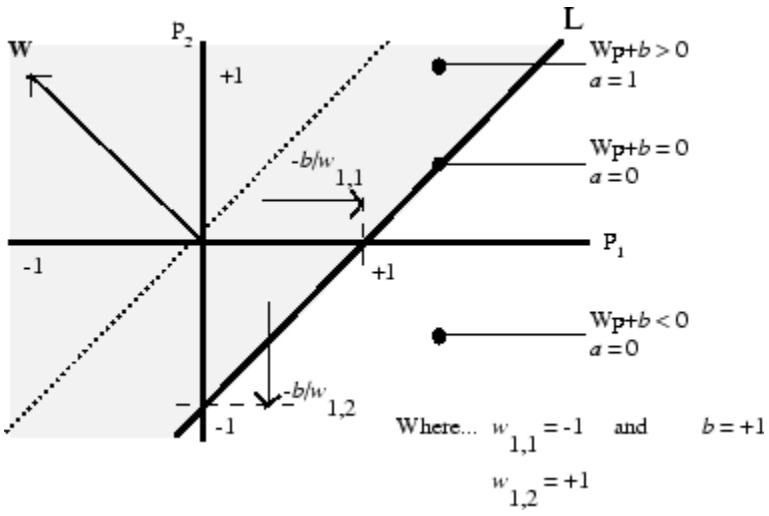
Each external input is weighted with an appropriate weight w_{1j}, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



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 $b = 1$.



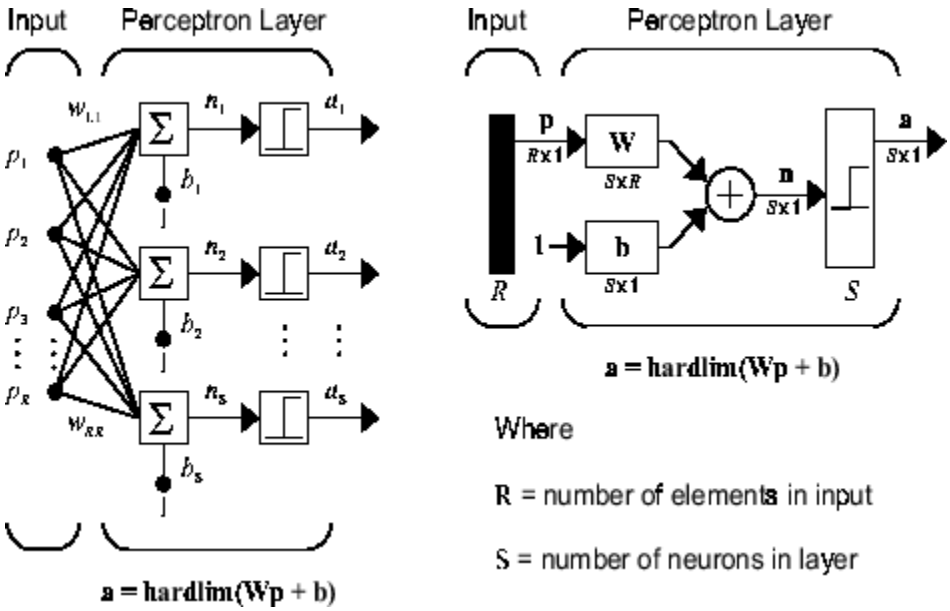
Two classification regions are formed by the *decision boundary* line L at **Wp** + b = 0. This line is perpendicular to the weight matrix **W** and shifted according to the bias b. Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit 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 26-12.

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 26-6. 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 $\mathbf{p}$ is an input to the network and $\mathbf{t}$ is the corresponding correct (target) output. The objective is to reduce the error $\mathbf{e}$, which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response $\mathbf{a}$ and the target vector $\mathbf{t}$. The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector $\mathbf{p}$ and the associated error $\mathbf{e}$. The target vector $\mathbf{t}$ 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 $\mathbf{w}$ 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 $\mathbf{w}$ and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector $\mathbf{p}$ is presented and the network's response $\mathbf{a}$ 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 $\mathbf{p}^{\mathrm{T}}$.

CASE 3. If **e** = –1, then make a change Δ**w** equal to $-\mathbf{p}^{\mathrm{T}}$.

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.

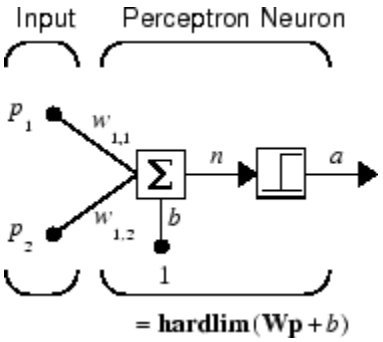
Training (train)

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 26-12.

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 29-2].

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 2-Input Perceptron” on page 28-106 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 28-123. 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 29-2].

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 28-111 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 28-117 to see how this normalized training rule works.

Linear Neural Networks

In this section...

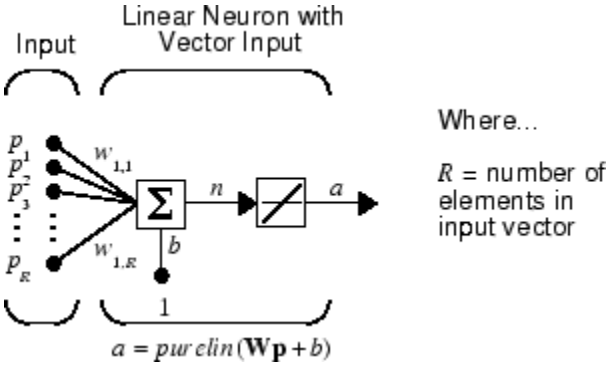
The linear networks discussed in this section are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Linear networks, like the perceptron, can only solve 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.

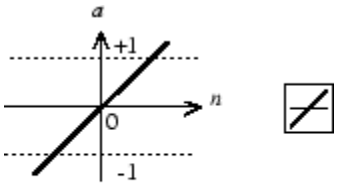
This section introduces `linearlayer`, a function that creates a linear layer, and `newlind`, a function that designs a linear layer for a specific purpose.

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

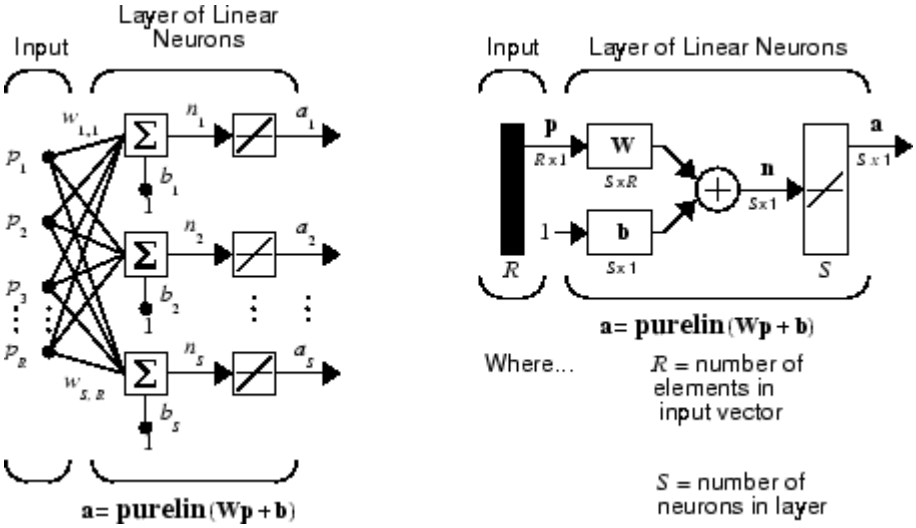
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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**.

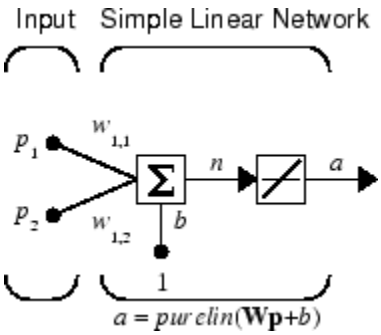


Note that the figure on the right defines an S-length output vector **a**.

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.



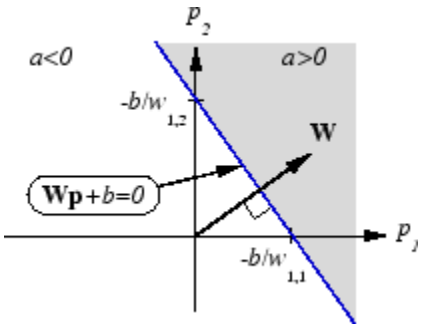
The weight matrix **W** in this case has only one row. The network output is

$$\alpha = purelin(n) = purelin(\mathbf{Wp} + b) = \mathbf{Wp} + b$$

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 29-2]).



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 29-2].

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.

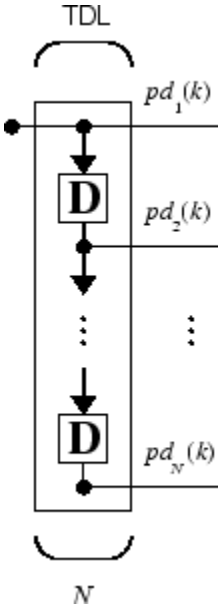
You might try “Pattern Association Showing Error Surface” on page 28-126. It shows error surfaces for a particular problem, illustrates the design, and plots the designed solution.

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 26-18. First, however, delays must be discussed.

Linear Networks with Delays

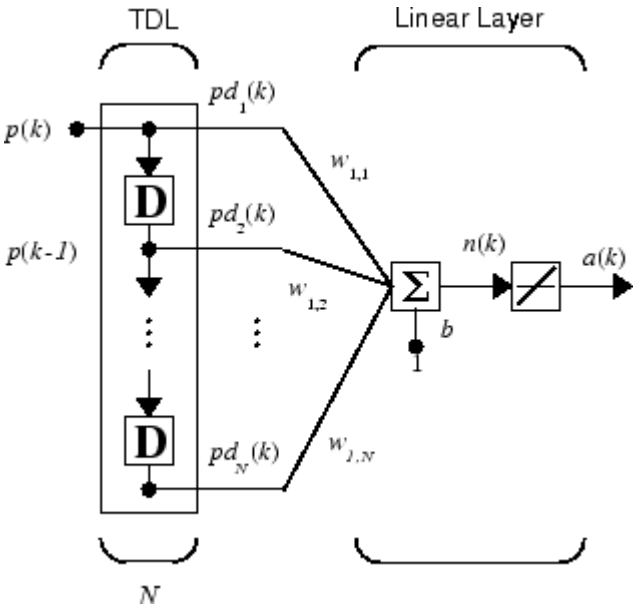
Tapped Delay Line

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 29-2]. 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 29-2] 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`.

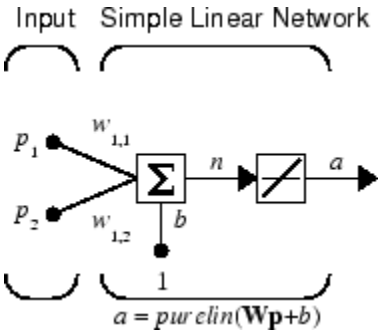
Type `help learnwh` and `help maxlinlr` for more details about these two functions.

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 26-23 for examples of various limitations.

This example program, “Training a Linear Neuron” on page 28-129, 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 28-132 to see how this is done.

Underdetermined Systems

Consider a single linear neuron with one input. This time, in “Underdetermined Problem” on page 28-136, 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 28-136 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 28-140, the network cannot solve the problem with zero error. You might want to try “Linearly Dependent Problem” on page 28-140.

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 28-141 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 27-11 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 27-11.

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 27-11 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 27-12 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 27-16 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 27-18 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 27-19 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 27-20 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)
```

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.

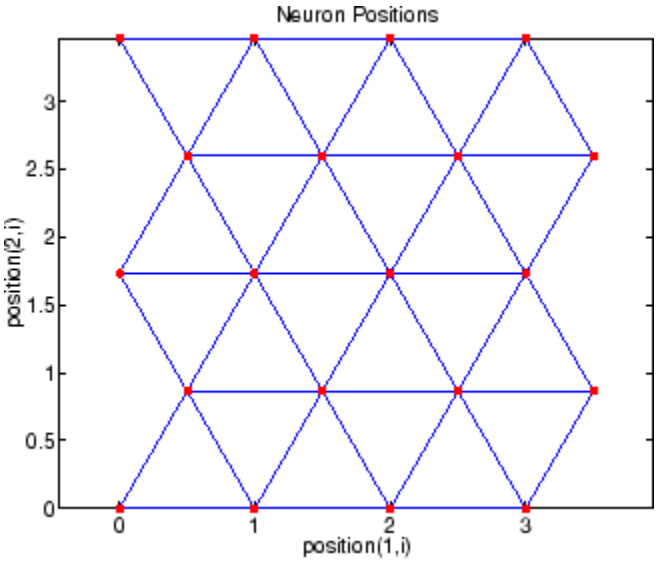
It is always set to the result of applying the layer's topology function (`net.layers{i}.topologyFcn`) to the positions of the layer's dimensions (`net.layers{i}.dimensions`).

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`).

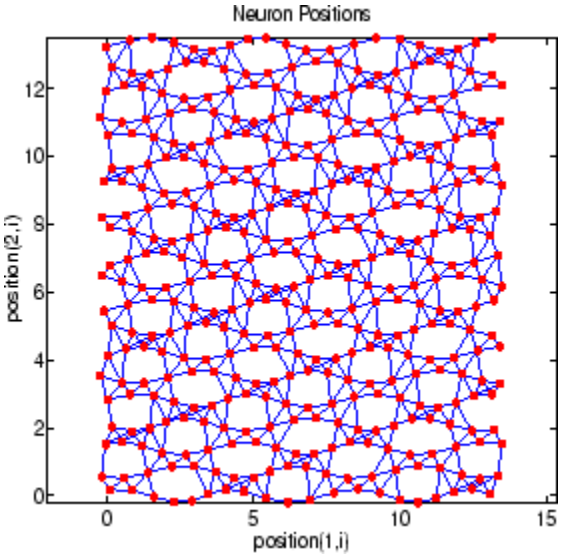
For a list of functions, type `help nntopology`.

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

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 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)
size(T)
```

```
ans =

    13   252


ans =

     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.

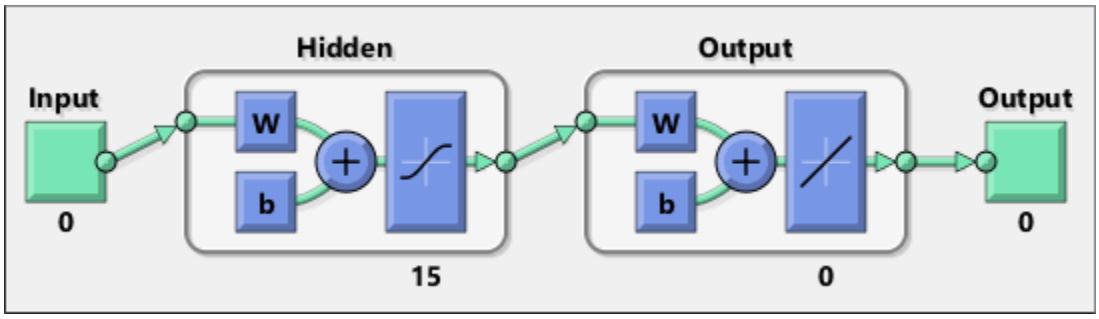
```
setdemorandstream(491218382)
```

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.

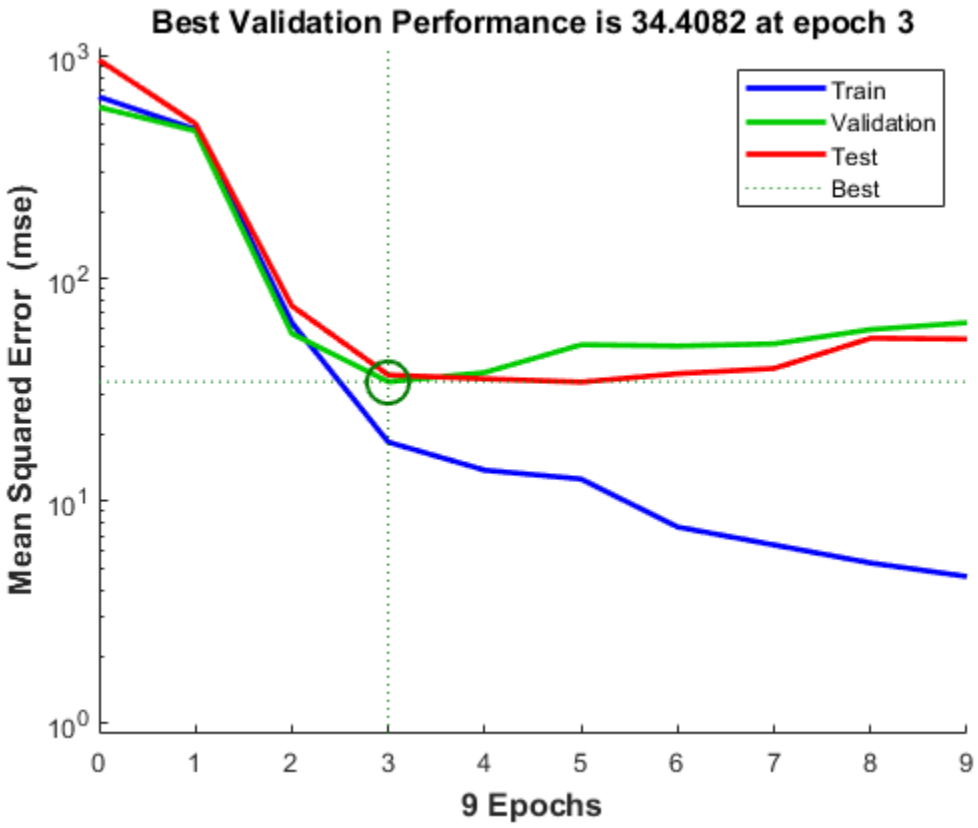
```
[net,tr] = train(net,X,T);
nntraintool
nntraintool('close')
```

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)
```

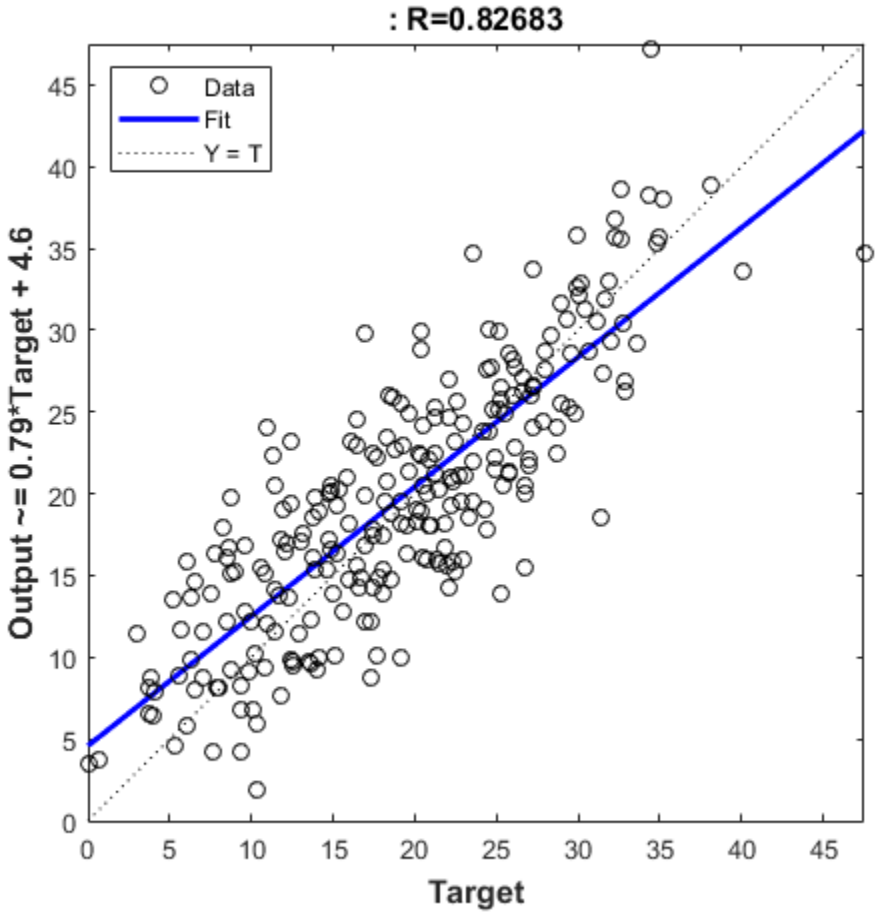
```
perf =

   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.

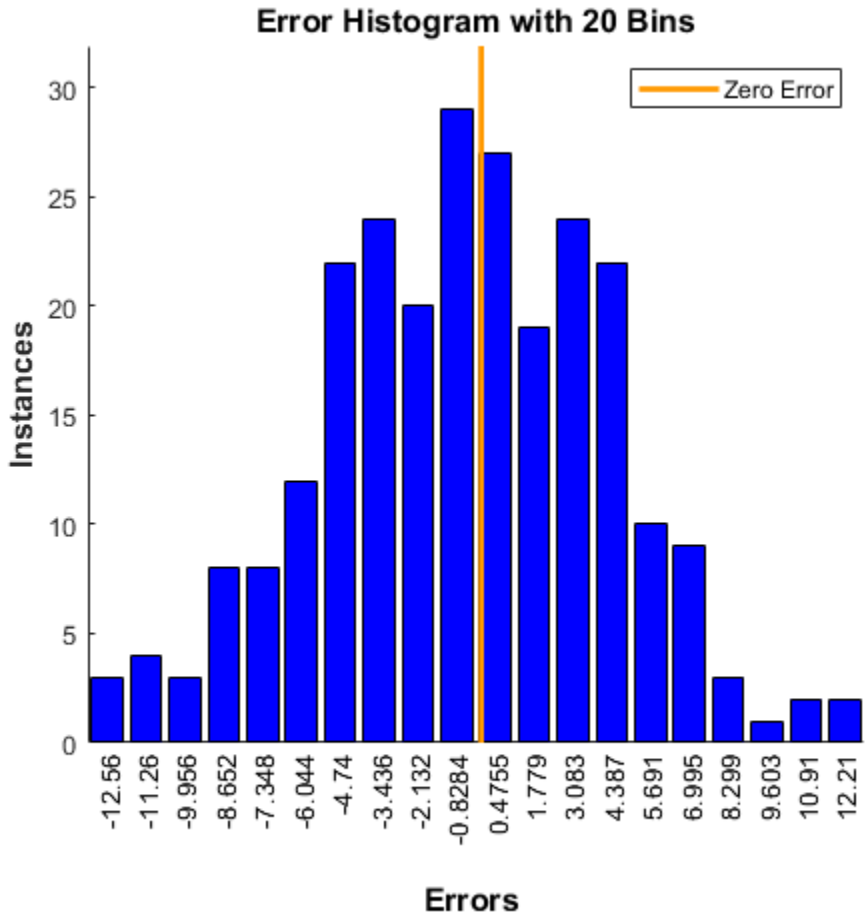
If this is not the case then further training, or training a network with more hidden neurons, would be advisable.

```
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)
size(t)
```

```
ans =

     6   200


ans =

     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.

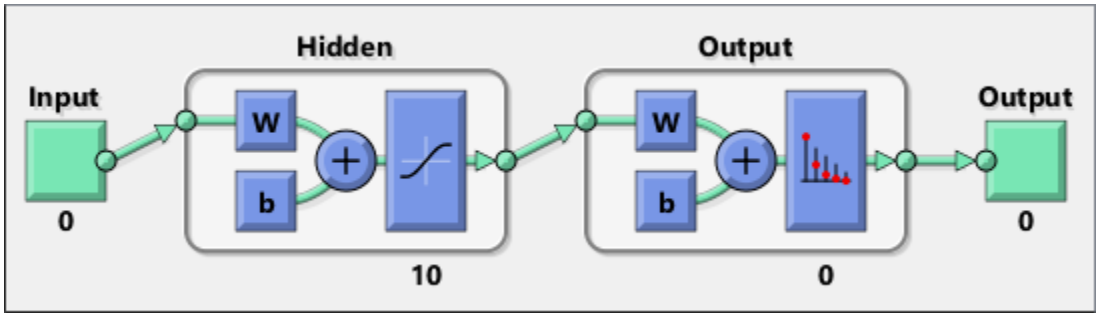
```
setdemorandstream(491218382)
```

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);
nntraintool
```

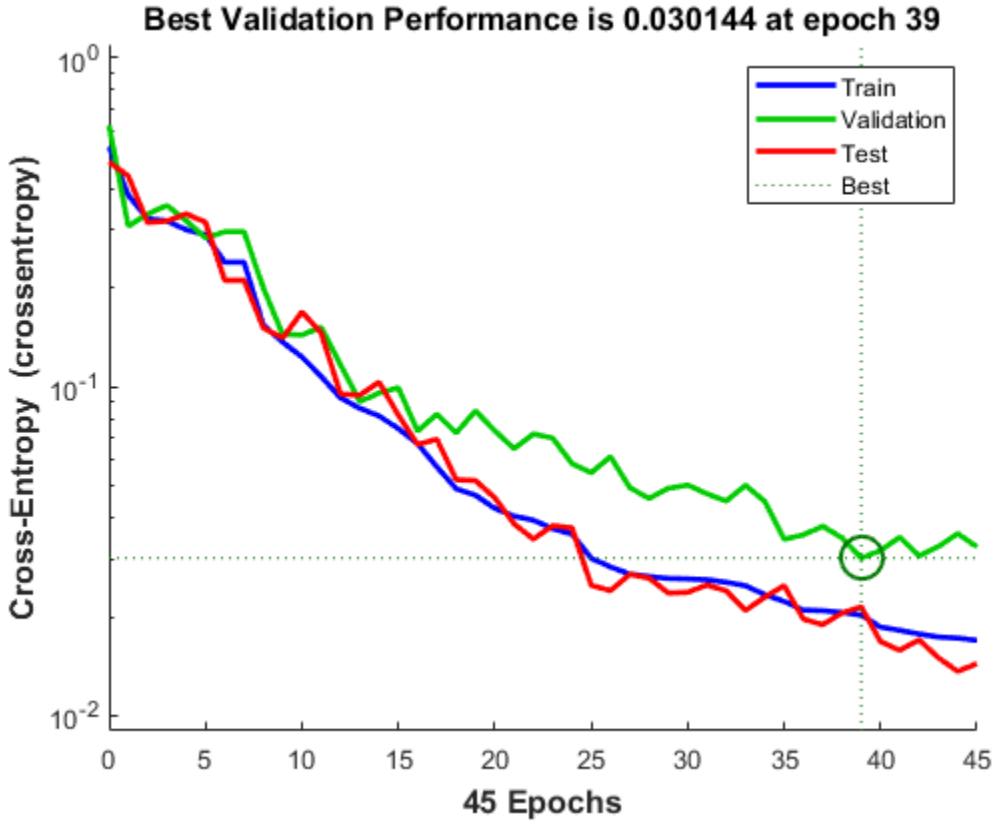

```
nntraintool('close')
```

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 =
```

```
Columns 1 through 13

   2     2     1     1     2     2     1     2     2     2     1     2     2

Columns 14 through 26

   2     2     2     1     1     2     2     2     1     2     2     1     2

Columns 27 through 30

   1     1     1     1
```

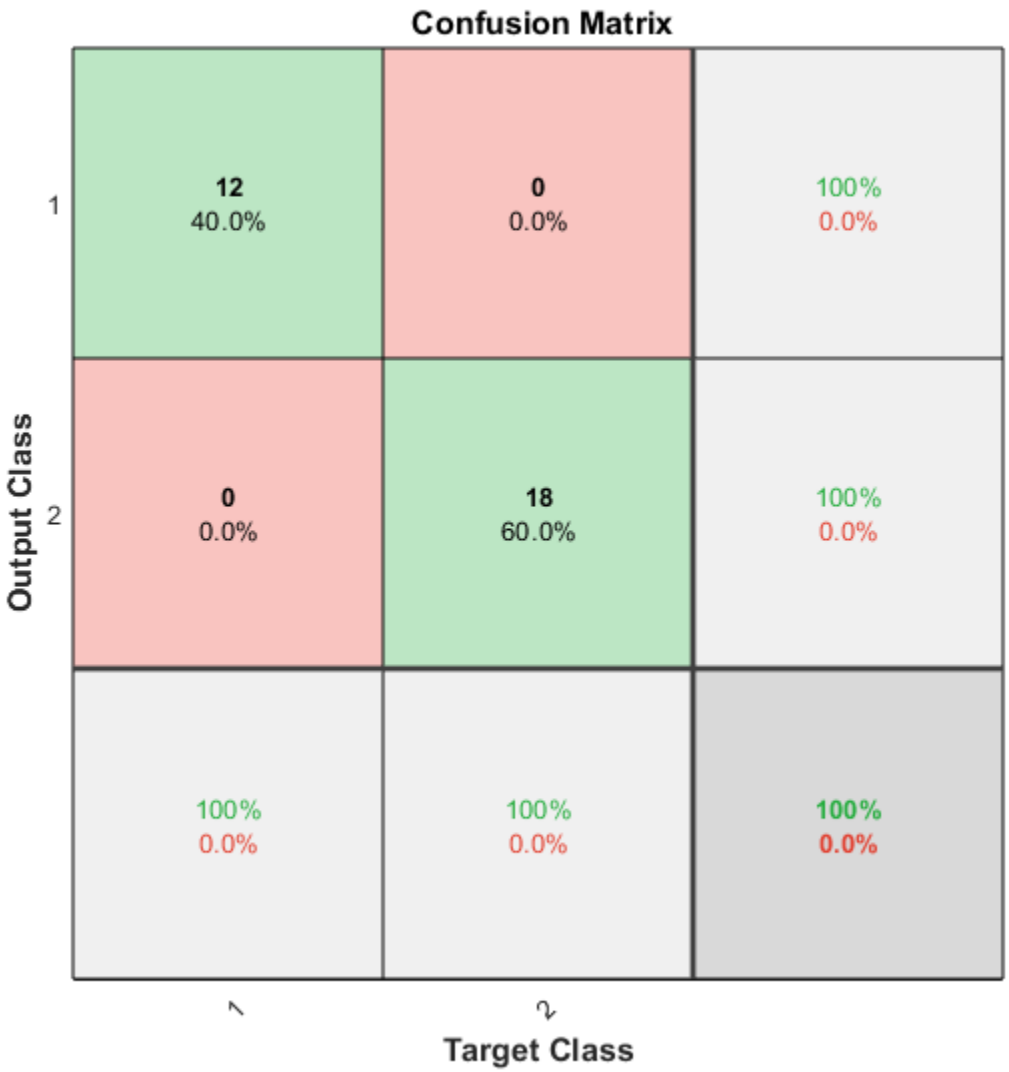
One measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

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)

fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c));
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

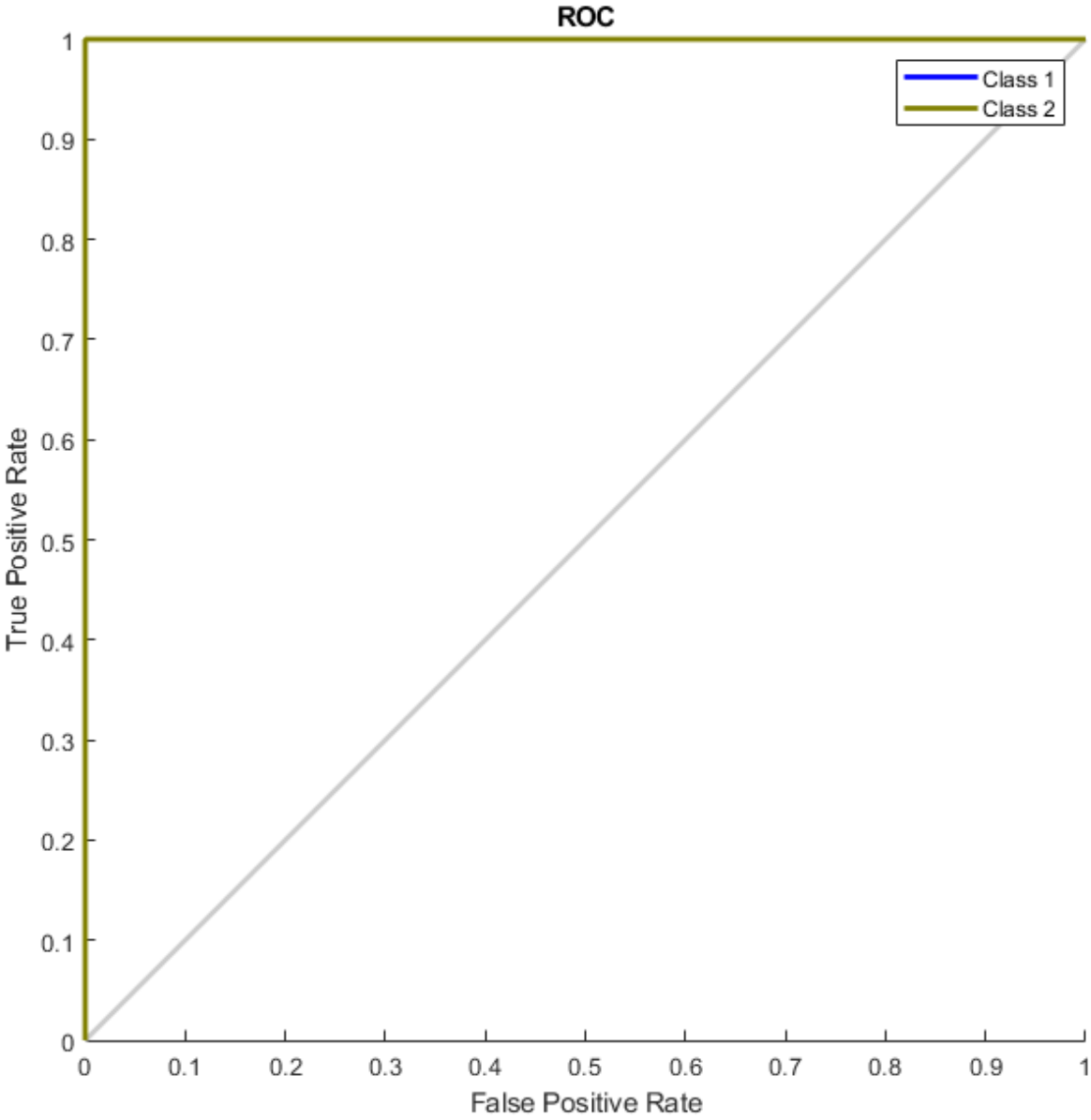
```
c =

     0


cm =
```

```
    12     0
     0    18

Percentage Correct Classification   : 100.000000%
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 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);
nntraintool
nntraintool('close')
```

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     1     1     1     1     2     2     2     2     2
```

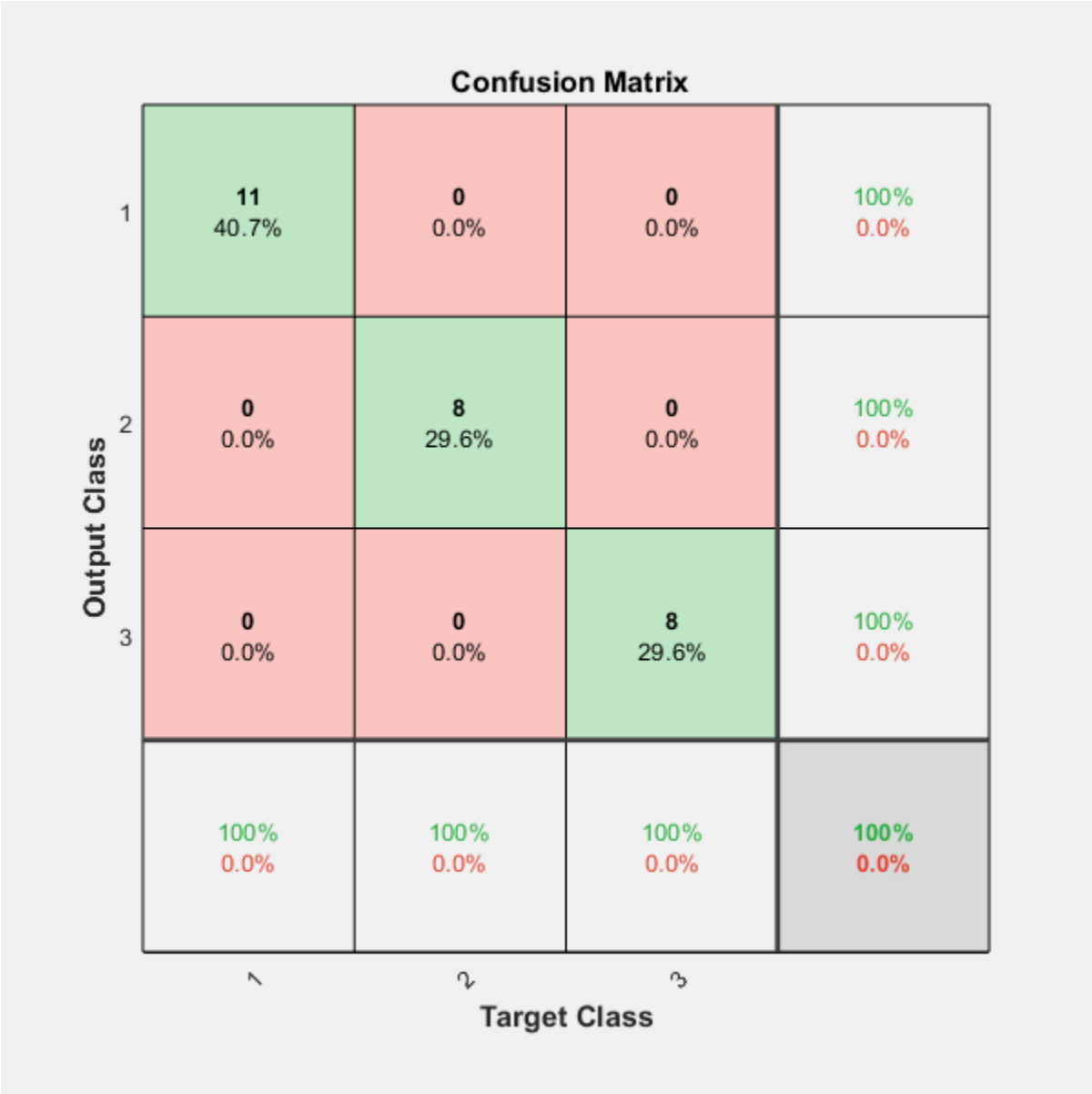
Another measure of how well the neural network has fit the data is the confusion plot. Here the confusion matrix is plotted across all samples.

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

    11     0     0
     0     8     0
     0     0     8
```

```
fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c));
```

```
Percentage Correct Classification   : 100.000000%
```

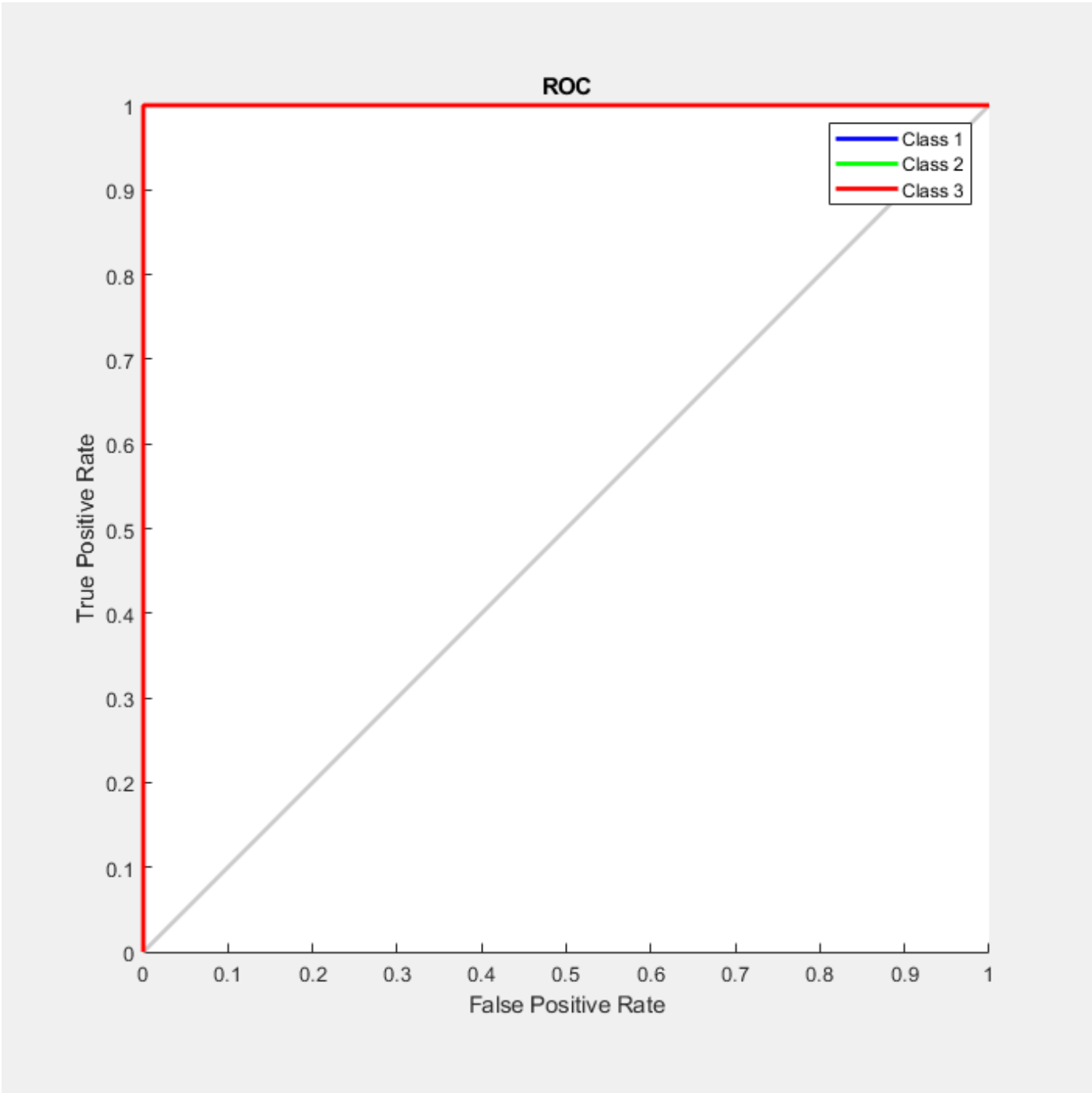
```
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

```
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 is from the FDA-NCI Clinical Proteomics Program Databank: https://home.ccr.cancer.gov/ncifdaproteomics/ppatterns.asp

To recreate the data in `ovarian_dataset.mat` used in this example, download and decompress the raw mass spectrometry data from the FDA-NCI website. 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.

An extensive description of this data set can be found in [1] and [2].

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.

To finish recreating the data from `ovarian_dataset.mat`, load the `OvarianCancerQAQCdataset.mat` and `rankfeatures` from 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.

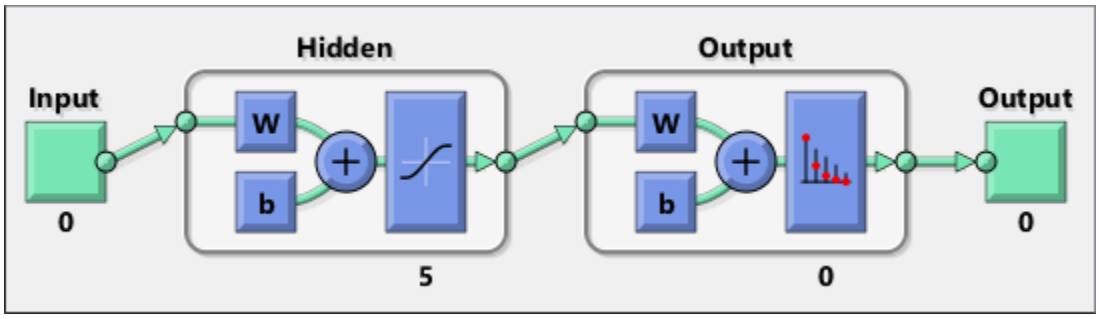
Since the neural network is initialized with random initial weights, the results after training the network vary slightly every time the example is run. To avoid this randomness, the random seed is set to reproduce the same results every time. However, setting the random seed is not necessary for your 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.

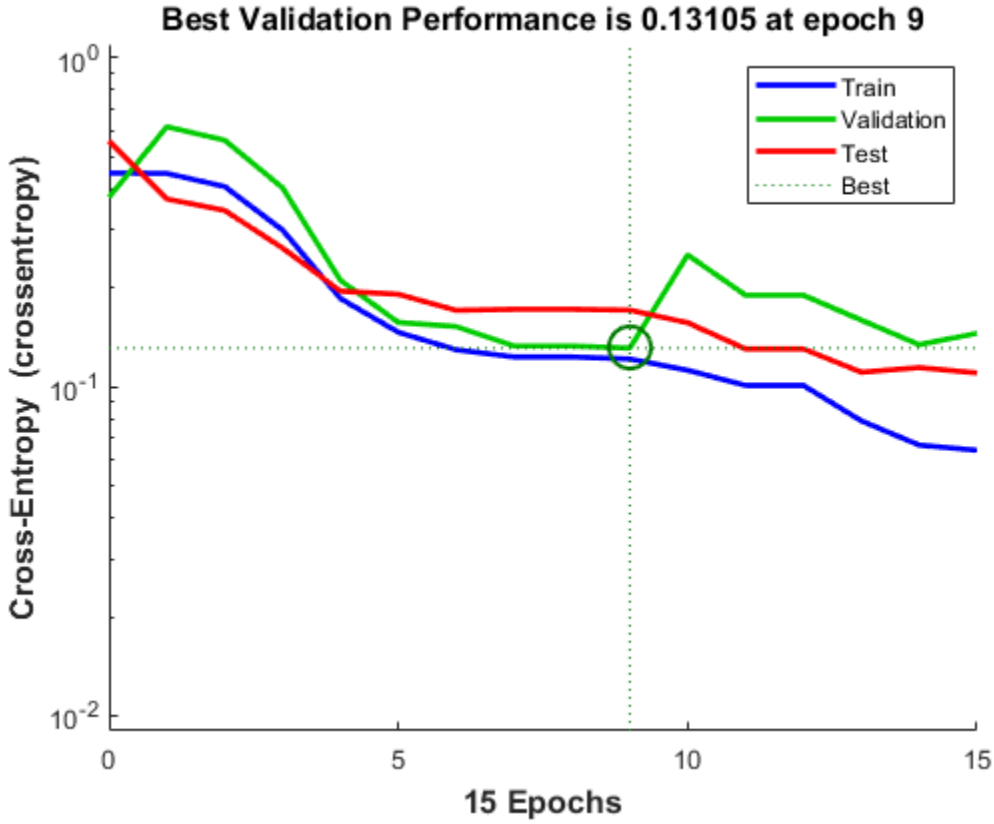
```
[net,tr] = train(net,x,t);
```

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

  Columns 1 through 19

   1   1   1   1   1   1   1   1   1   1   1   1   1   0   0   1   1   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1   0   0   1   1

  Columns 20 through 32

   1   1   0   0   0   0   0   0   0   0   0   1   0
```

```
   0   0   1   1   1   1   1   1   1   1   1   0   1
```

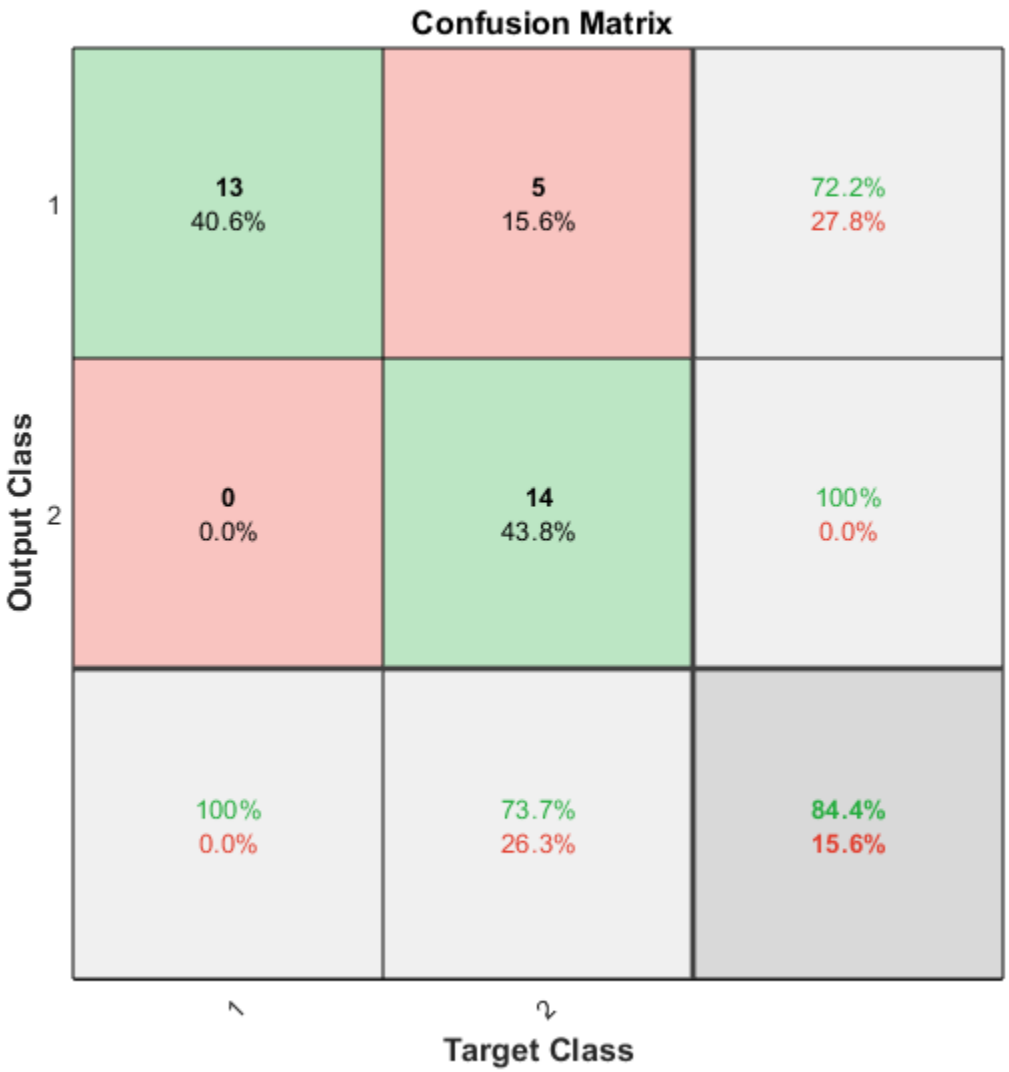
One measure of how well the neural network has fit the data is the confusion plot.

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));
fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

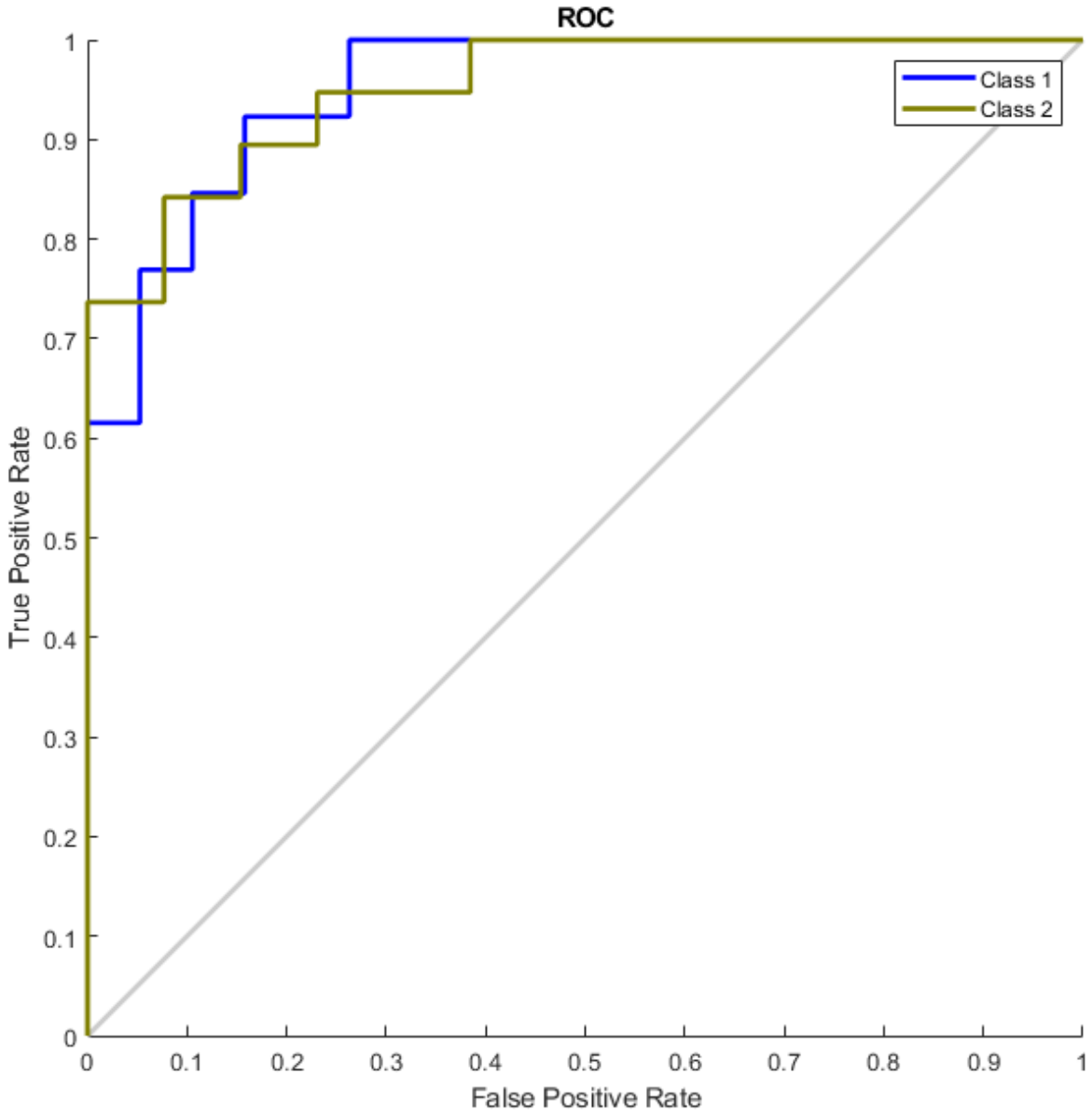
```
Percentage Correct Classification   : 84.375000%
Percentage Incorrect Classification : 15.625000%
```

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

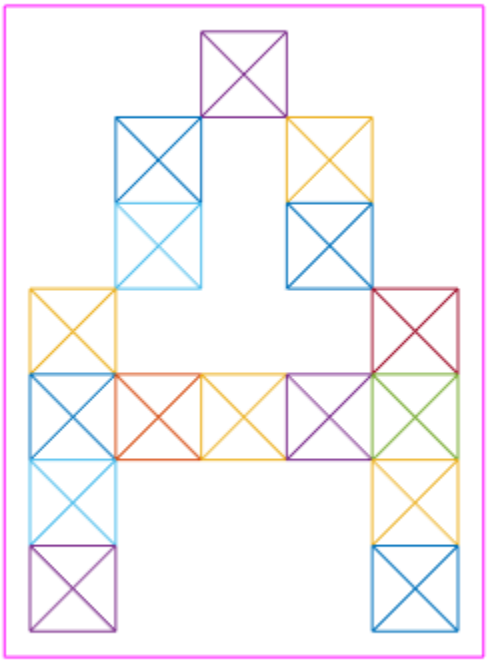
The script **prprob** defines a matrix X with 26 columns, one for each letter of the alphabet. Each column has 35 values which can either be 1 or 0. Each column of 35 values defines a 5x7 bitmap of a letter.

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))
```

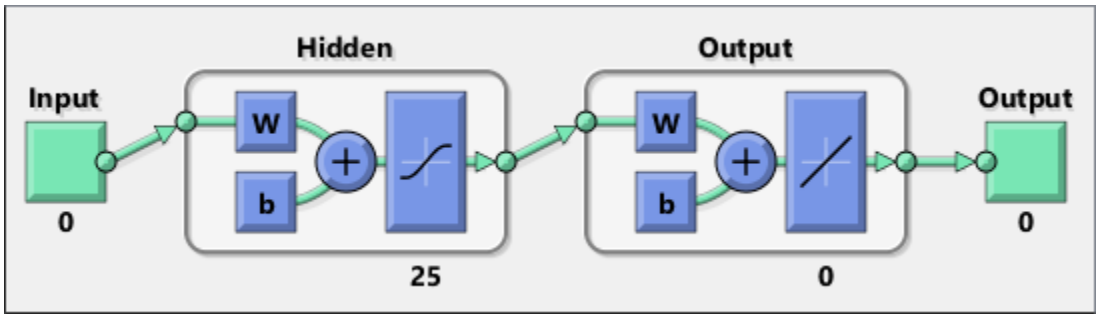


Creating the First Neural Network

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

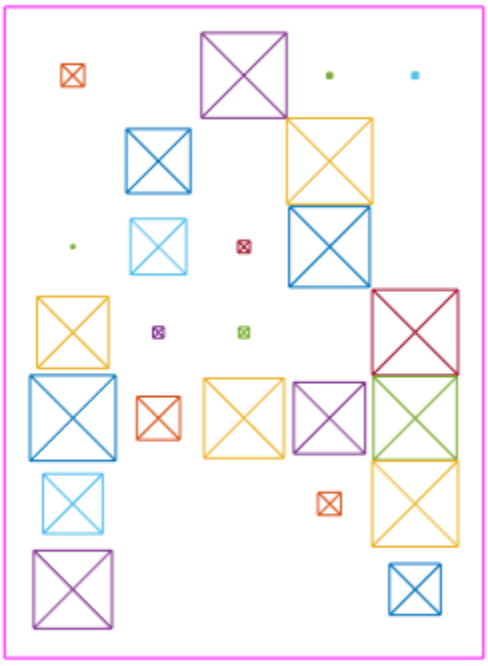
We would like the network to not only recognize perfectly formed letters, but also noisy versions of the letters. So we will try training a second network on noisy data and compare its ability to generalize with the first network.

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.

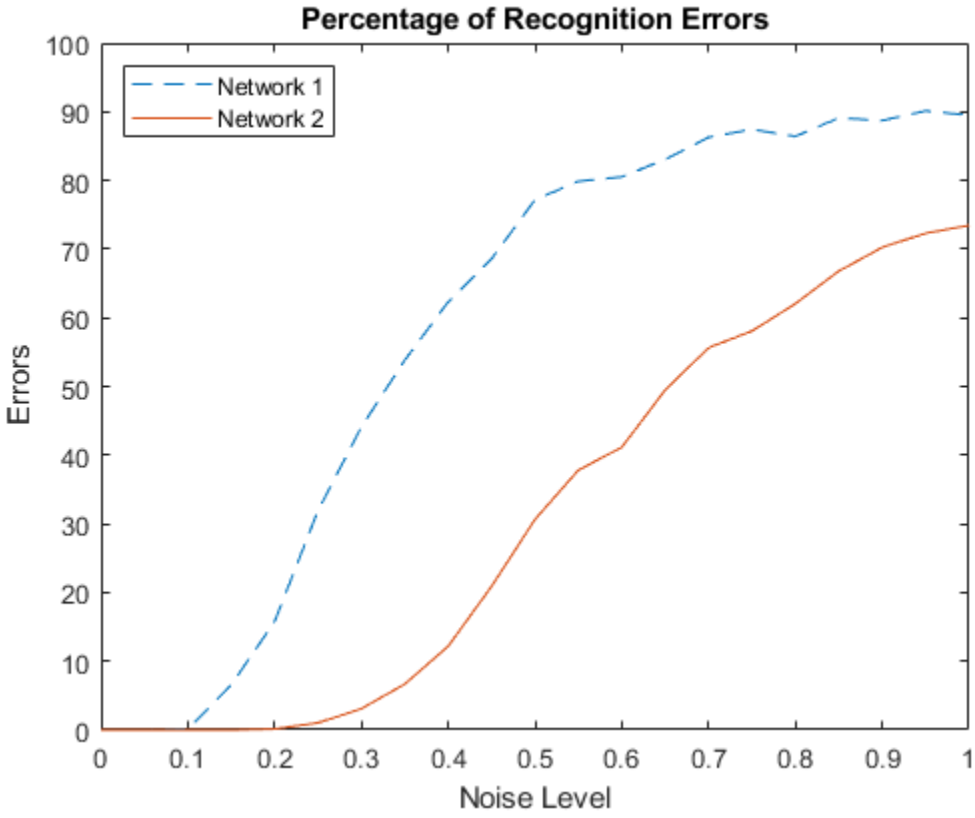
```
net2 = feedforwardnet(25);
net2 = train(net2,Xn,Tn,nnMATLAB);
```

```
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.

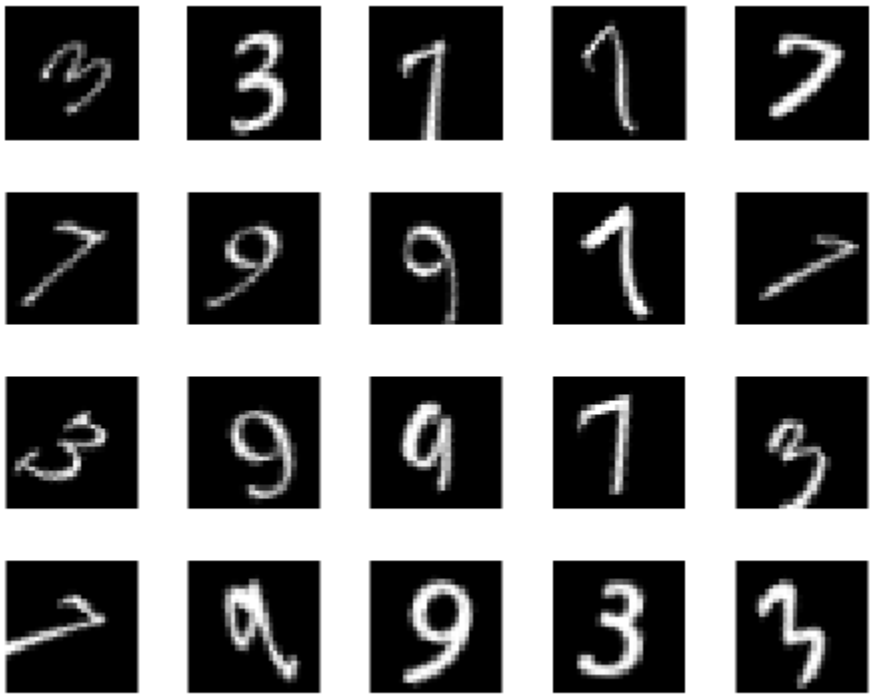
This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final softmax layer, and join the layers together to form a stacked network, which you 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:

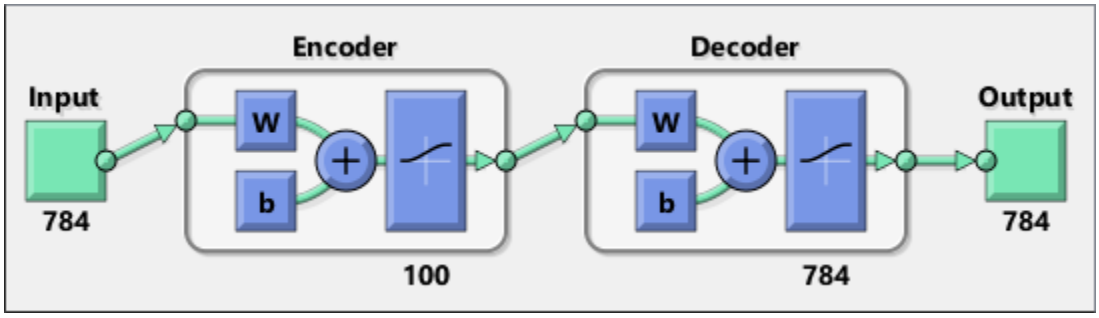
- `L2WeightRegularization` controls the impact of an L2 regularizer for the weights of the network (and not the biases). This should typically be quite small.
- `SparsityRegularization` controls the impact of a sparsity regularizer, which attempts to enforce a constraint on the sparsity of the output 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.

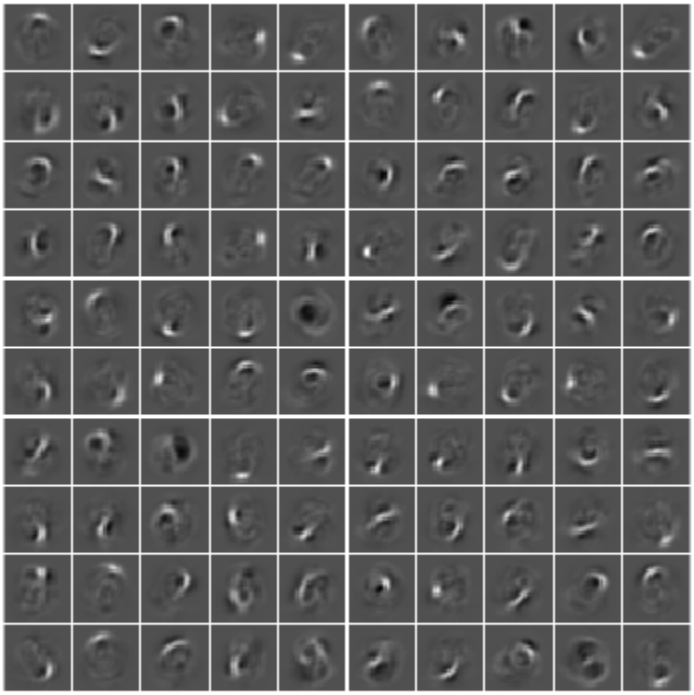
```
view(autoenc1)
```



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);
```

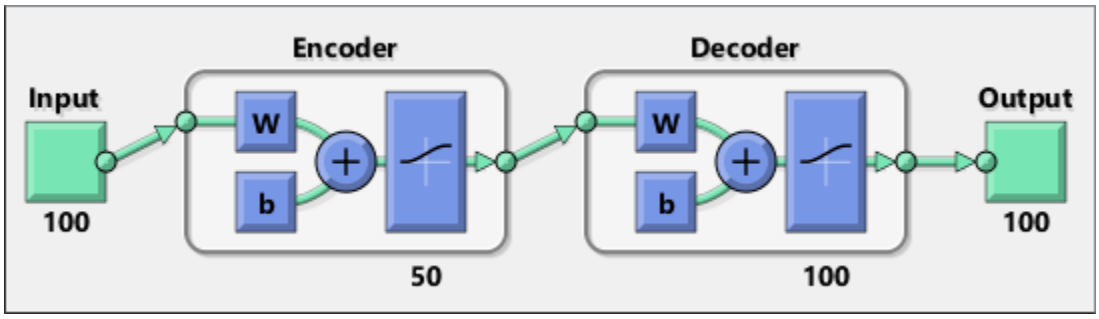
Training the second autoencoder

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.

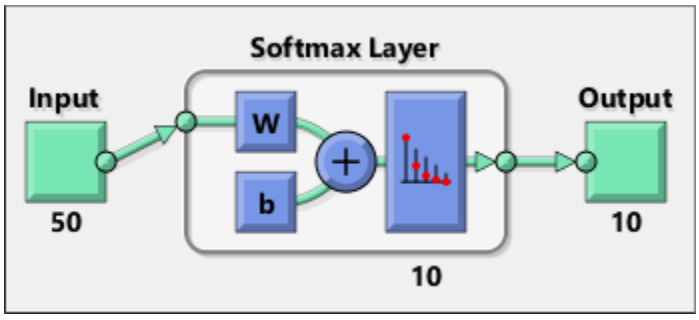
Training the final softmax layer

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.

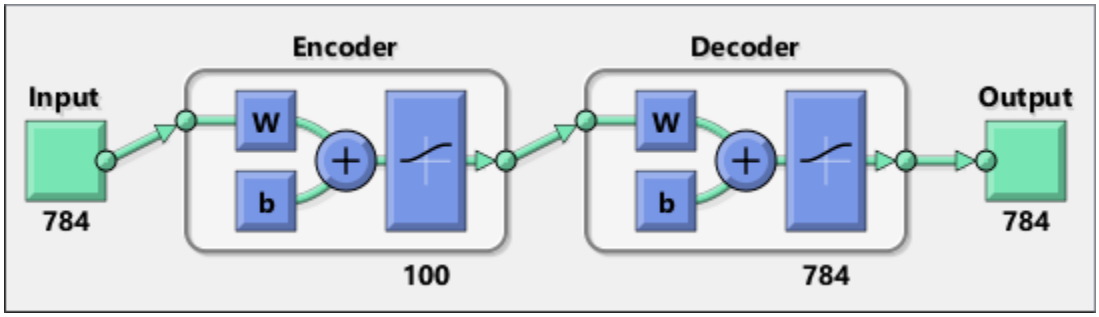
```
view(softnet)
```

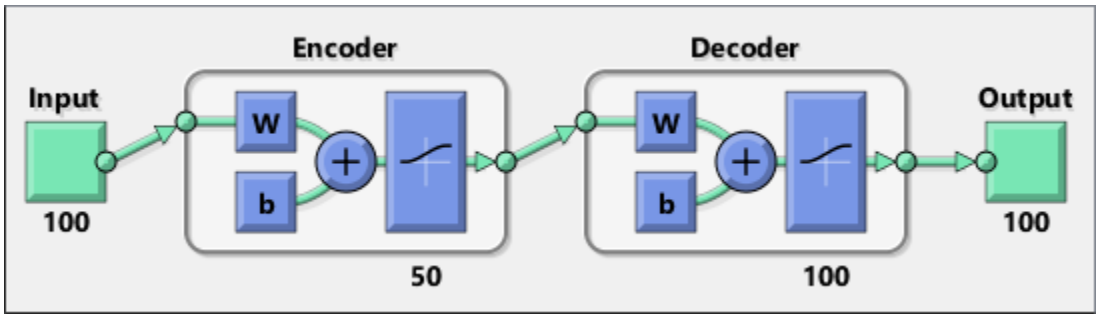


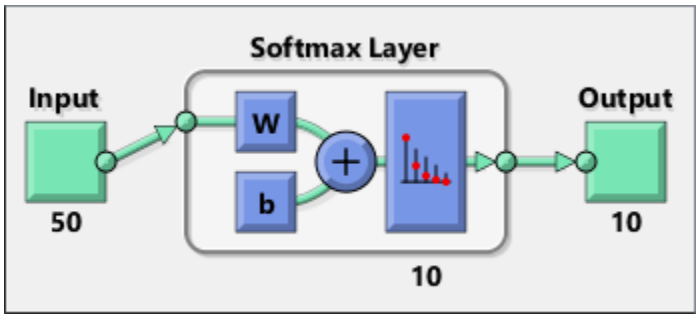
Forming a stacked neural network

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)
```





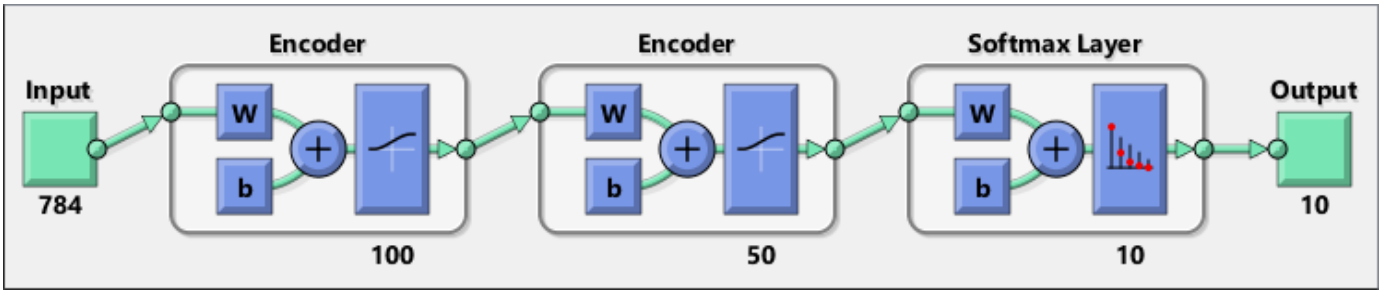


As was explained, the encoders from the autoencoders have been used to extract features. You can stack the encoders from the autoencoders together with the softmax layer to form a stacked network for classification.

```
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)
```

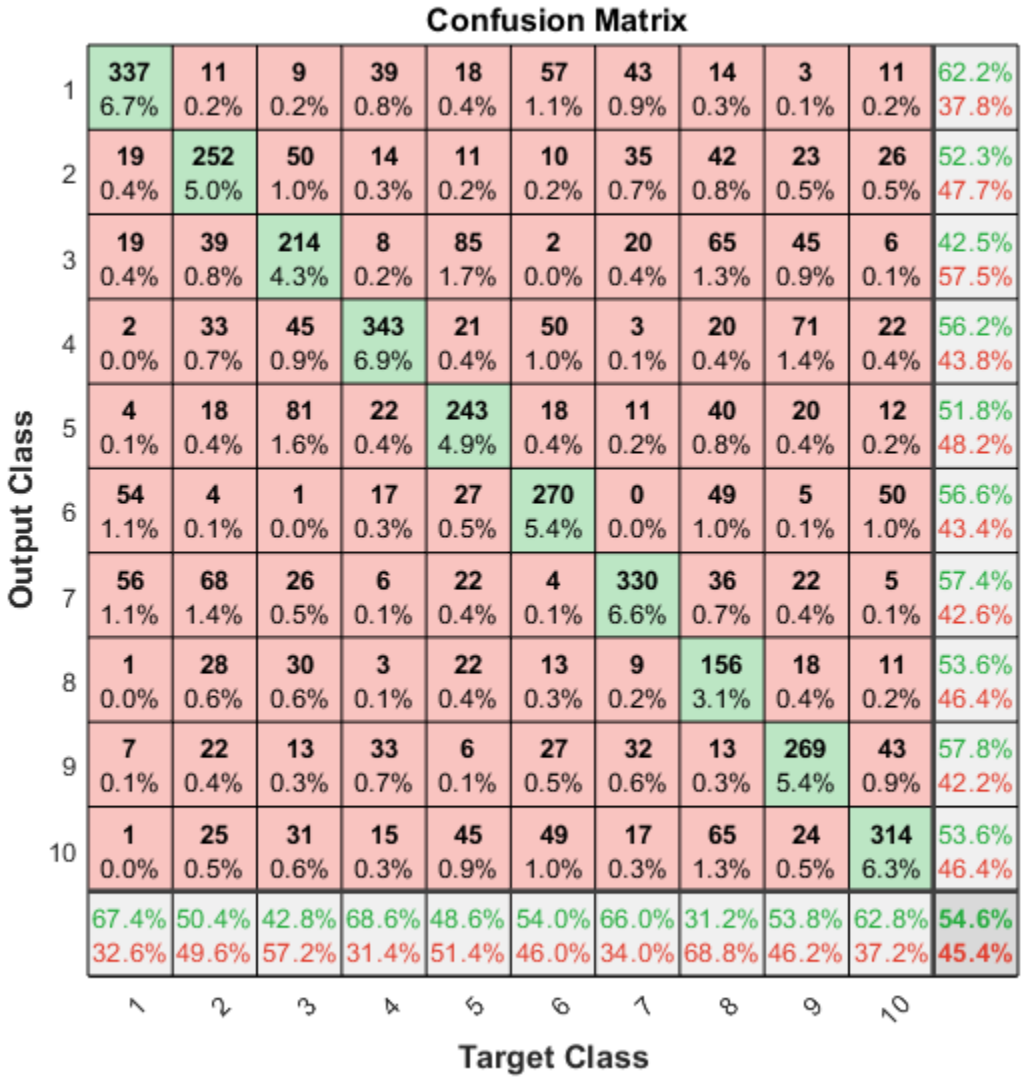


With the full network formed, you can compute the results on the test set. To use images with the stacked network, you have to reshape the test images into a matrix. You can do this by stacking the columns of an image to form a vector, and then forming a matrix from these vectors.

```
% 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.

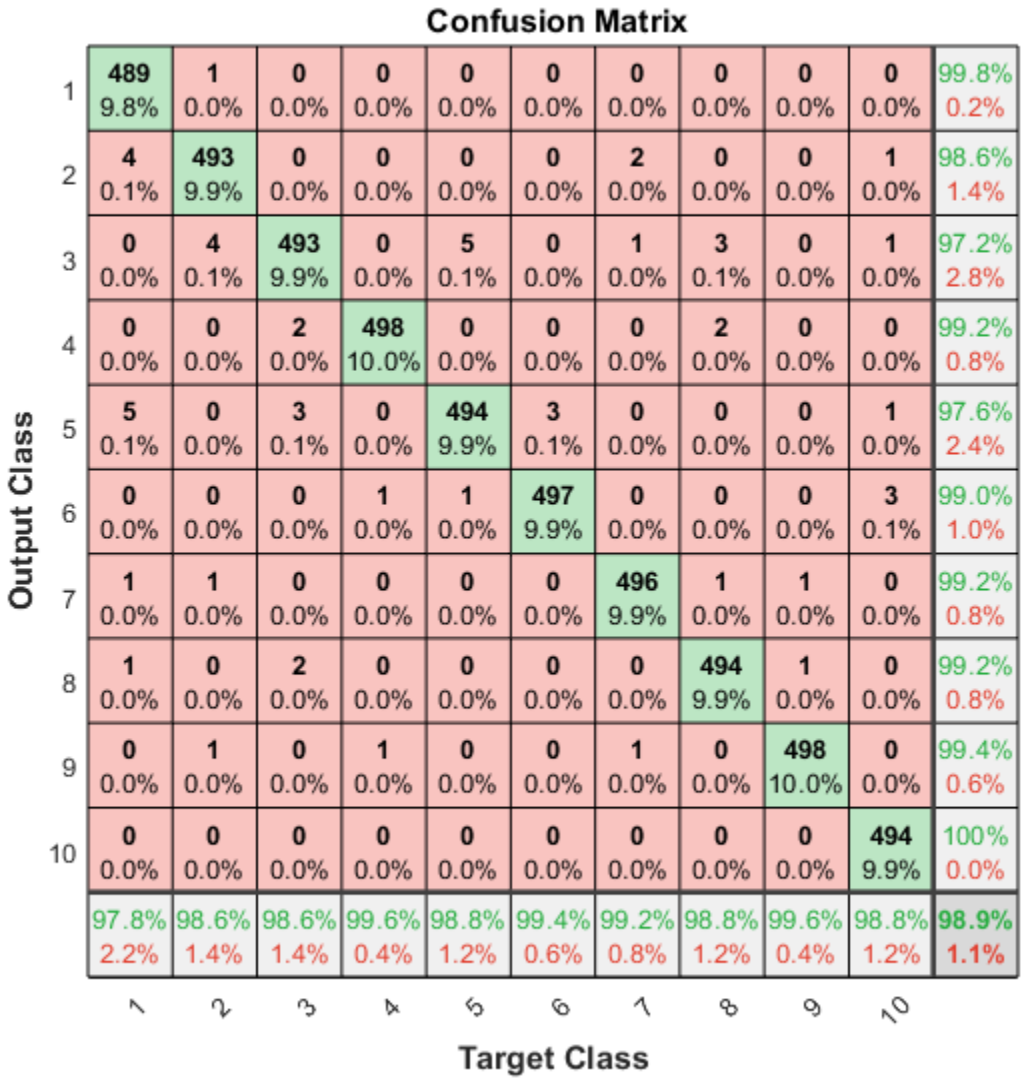
You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

```
% 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 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 =

     4   150
```

Clustering with a Neural Network

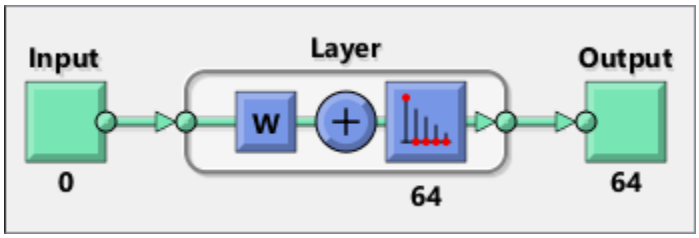
The next step is to create a neural network that will learn to cluster.

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)
```

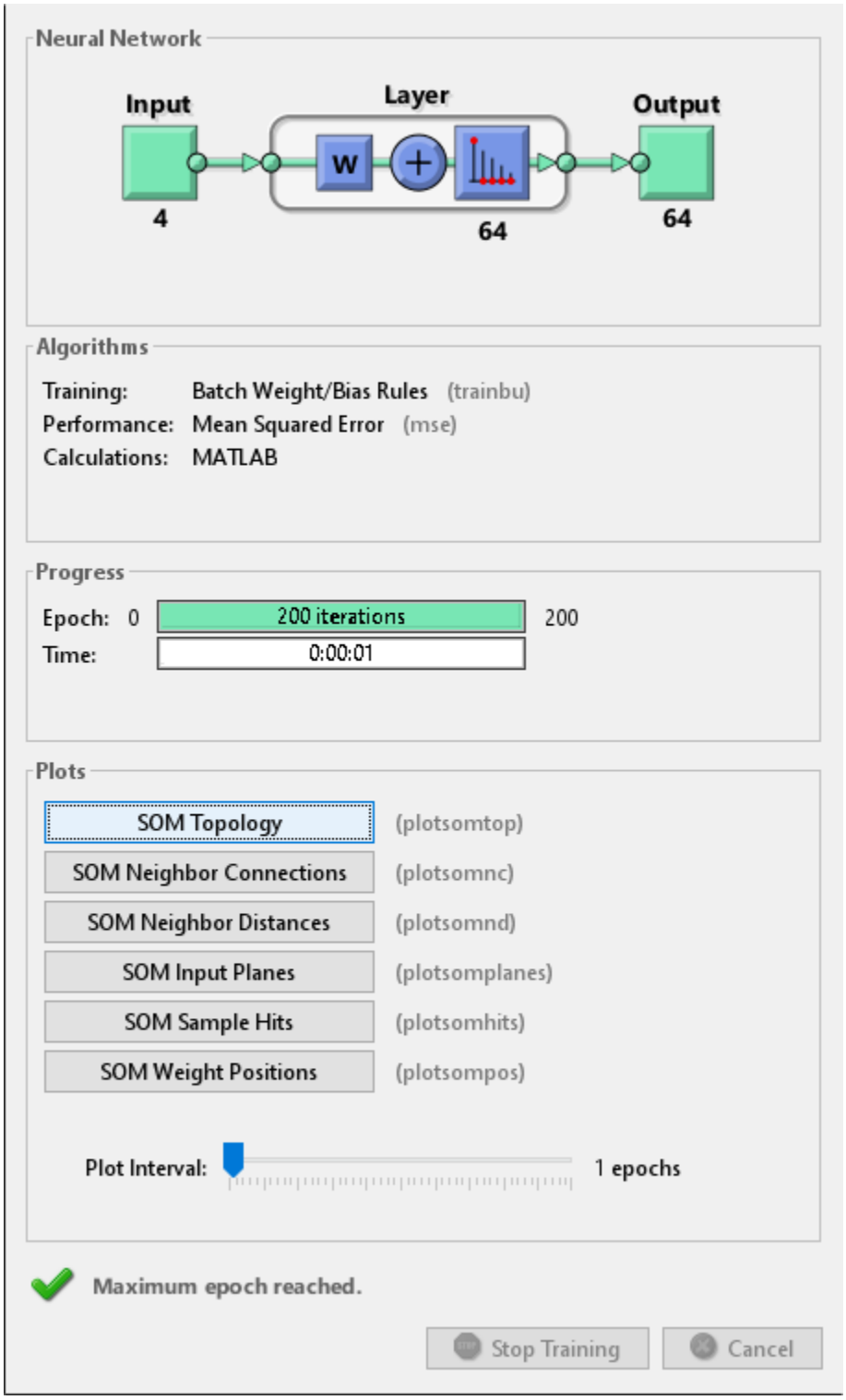


Now the network is ready to be optimized with **train**.

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);
nntraintool
```




```
nntraintool('close')
```

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

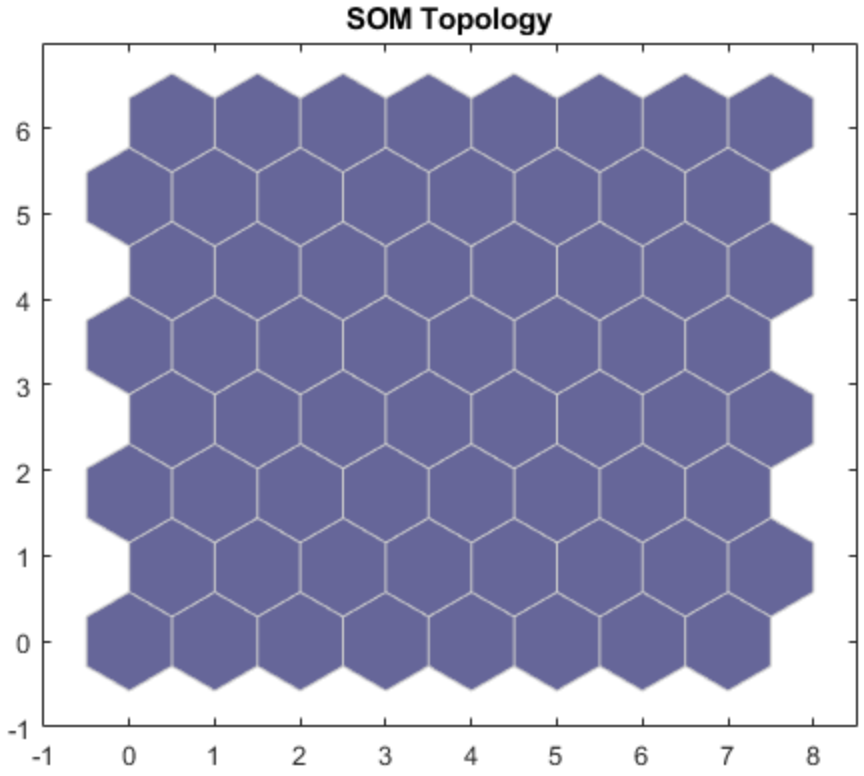
classify new flowers accordingly. The network output will be a 64x150 matrix, where each ith column represents the jth cluster for each ith input vector with a 1 in its jth element.

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);
```

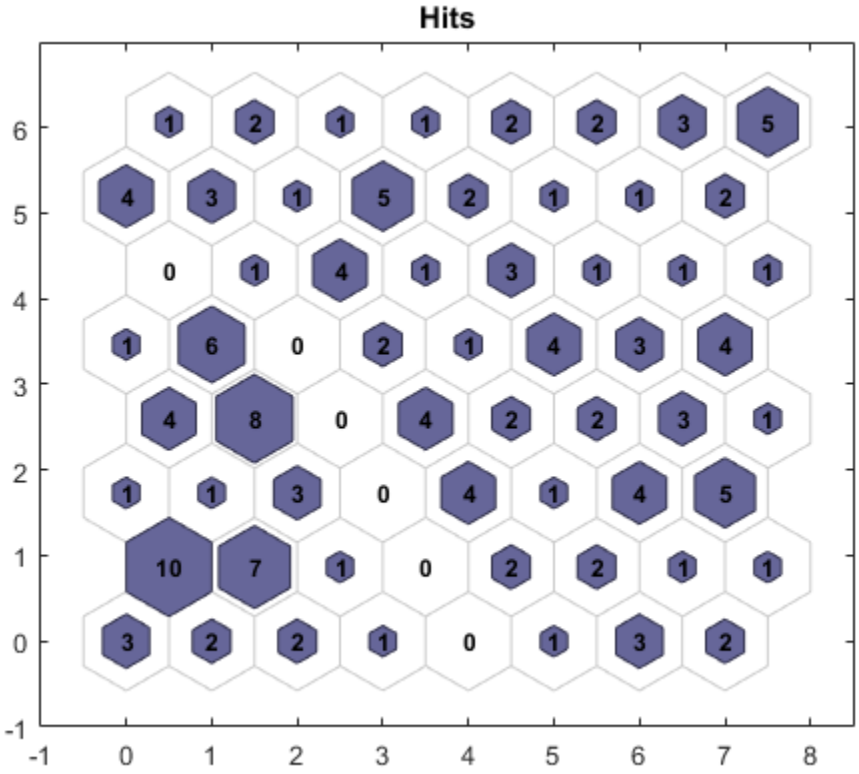
plotsomtop plots the self-organizing maps topology of 64 neurons positioned in an 8x8 hexagonal grid. Each neuron has learned to represent a different class of flower, with adjacent neurons typically representing similar classes.

```
plotsomtop(net)
```



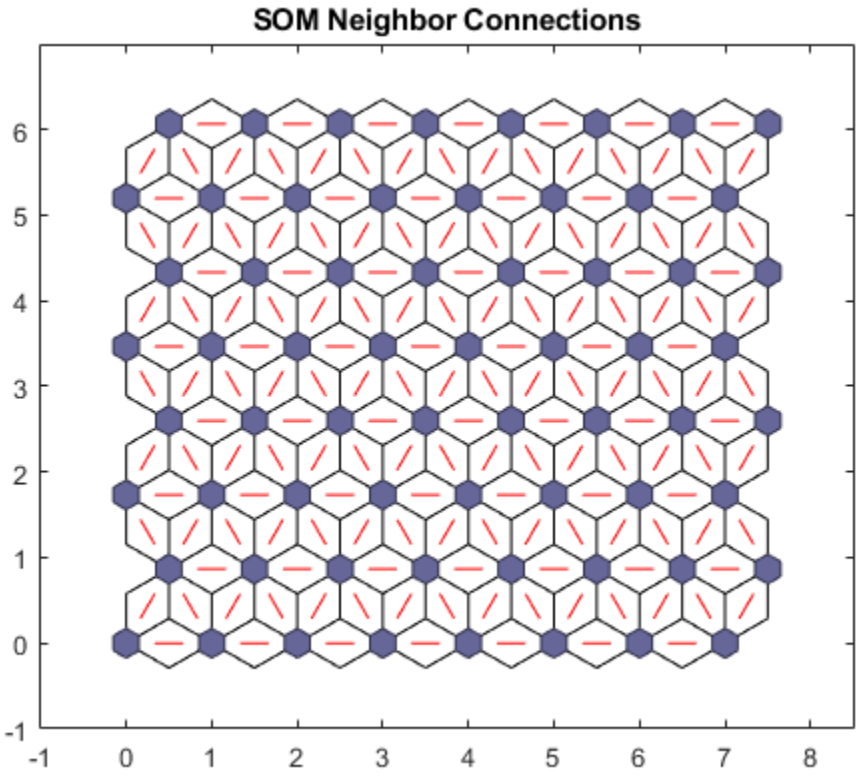
plotsomhits calculates the classes for each flower and shows the number of flowers in each class. Areas of neurons with large numbers of hits indicate classes representing similar highly populated regions of the feature space. Whereas areas with few hits indicate sparsely populated regions of the feature space.

```
plotsomhits(net,x)
```



plotsomnc shows the neuron neighbor connections. Neighbors typically classify similar samples.

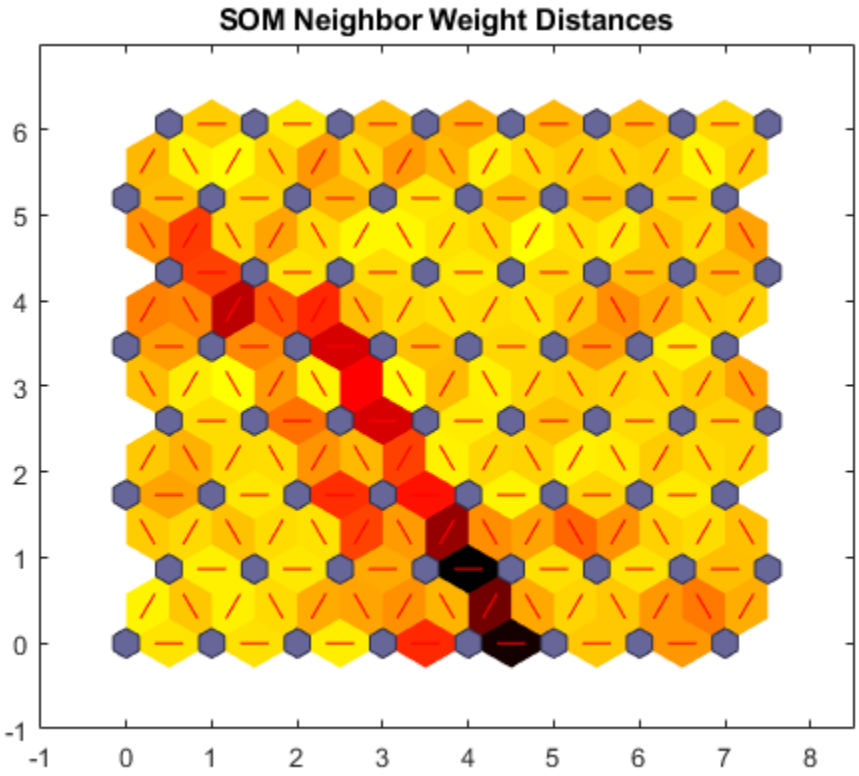
```
plotsomnc(net)
```



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.

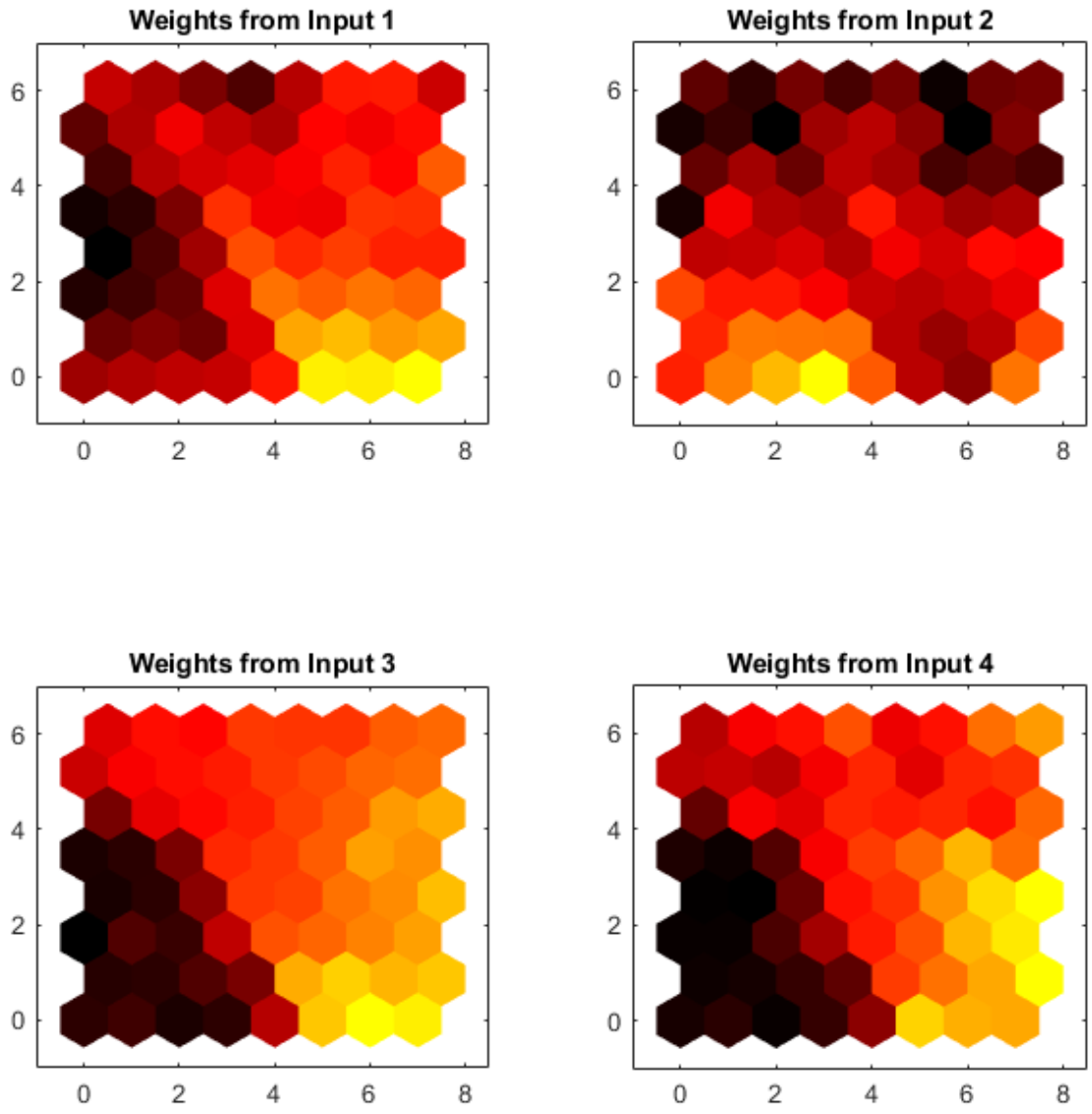
Long borders of dark connections separating large regions of the input space indicate that the classes on either side of the border represent flowers with very different features.

```
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.

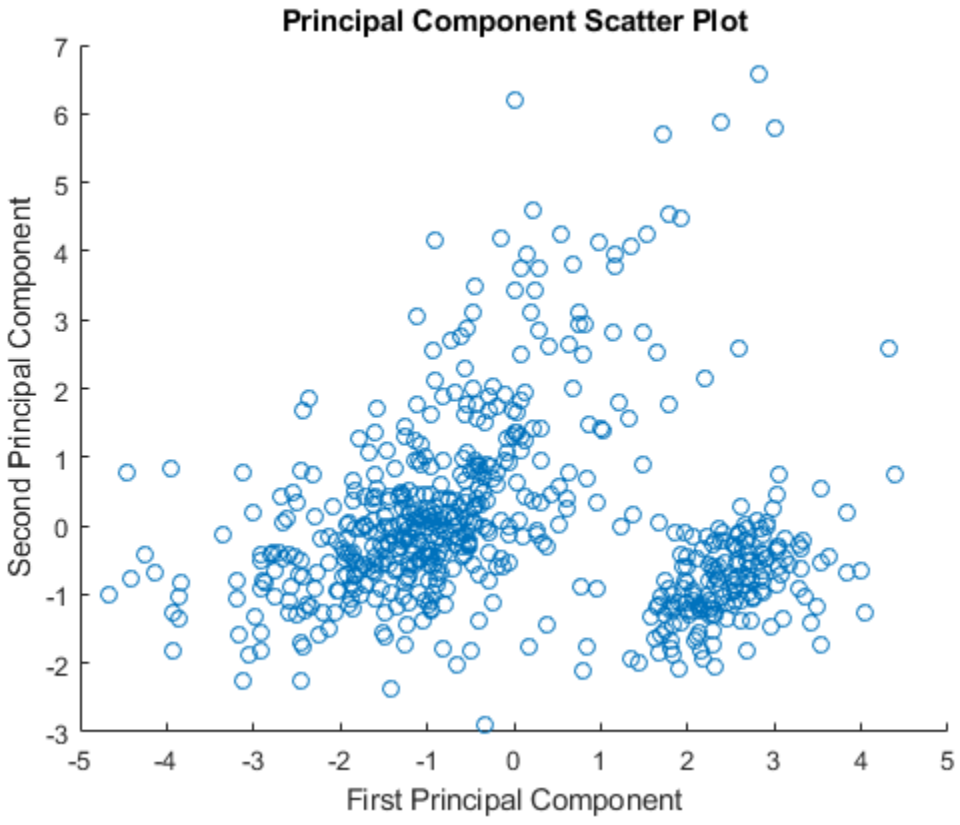
The two settings variables can be used to apply **mapstd** and **processpca** to new data to be consistent.

```
[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');
```



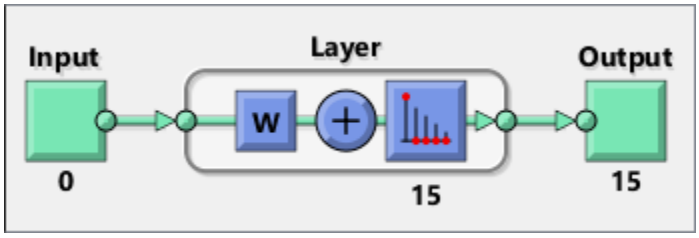
Cluster Analysis: Self-Organizing Maps

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)
```

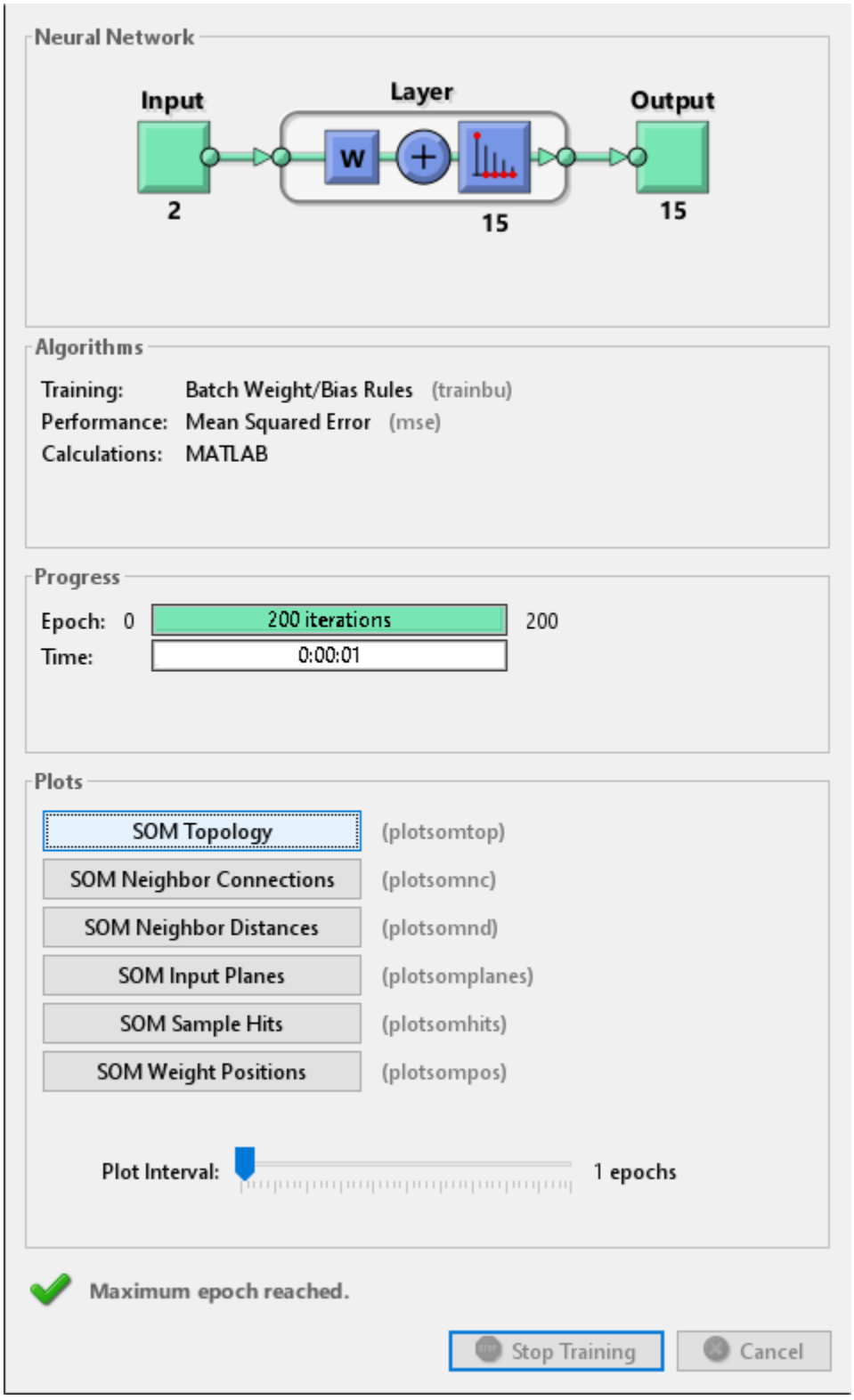


Now the network is ready to be trained.

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.

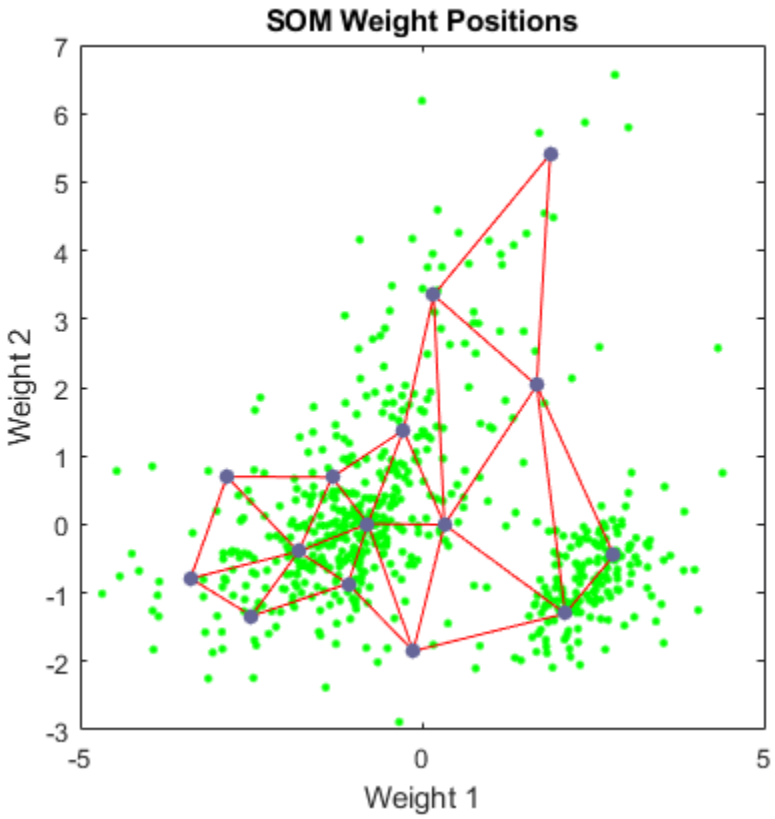
The buttons at the bottom open useful plots which can be opened during and after training. Links next to the algorithm names and plot buttons open documentation on those subjects.

```
net = train(net,x);
nntraintool
nntraintool('close')
```



Use **plotsompos** to display the network over a scatter plot of the first two dimensions of the data.

```
figure
plotsompos(net,x);
```

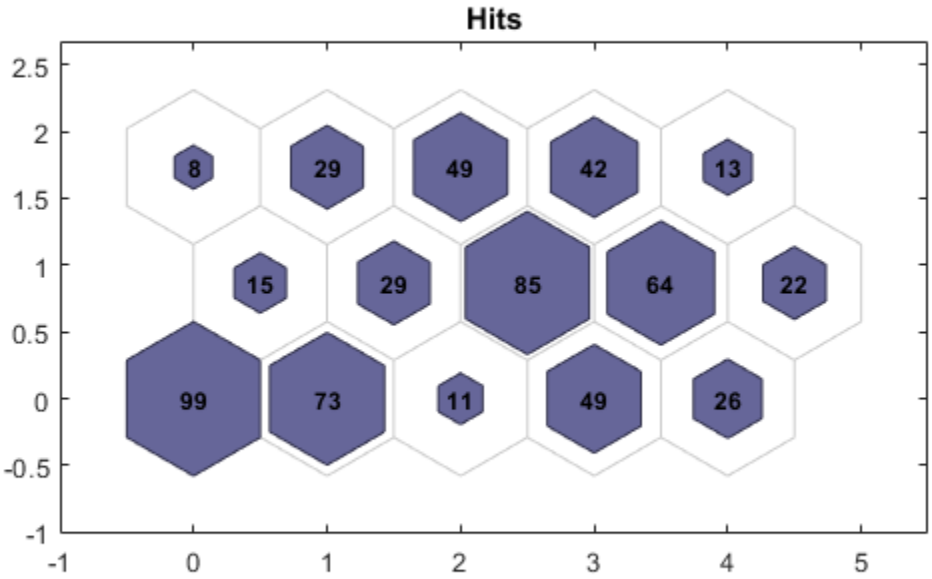


You can assign clusters using the SOM by finding the nearest node to each point in the data set.

```
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)
size(t)
```

```
ans =

           1        4001
```

```
ans =

     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.

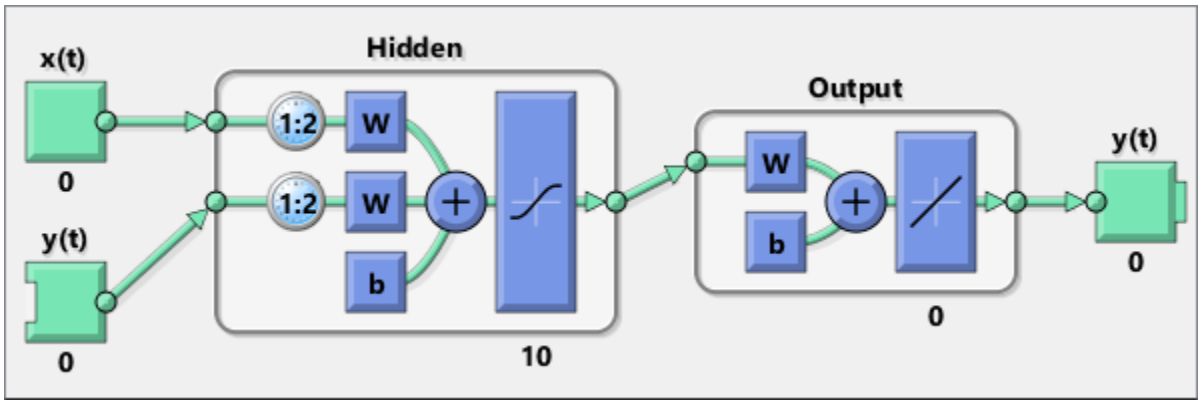
We will try a single hidden layer of 10 neurons for this example. In general, more difficult problems require more neurons, and perhaps more layers. Simpler problems require fewer neurons.

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.

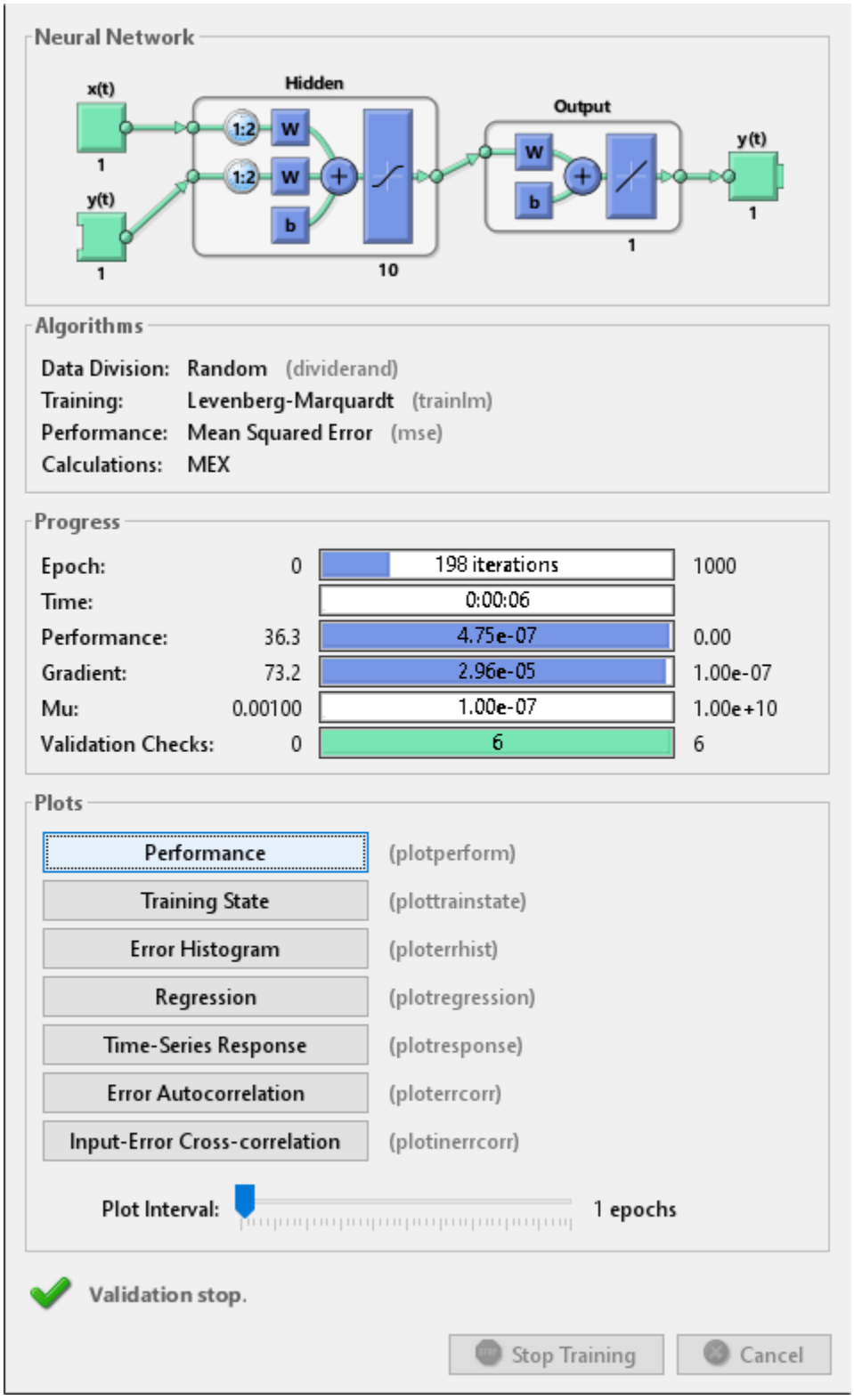
```
[Xs,Xi,Ai,Ts] = preparets(net,x,{},t);
```

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);
nntraintool
```



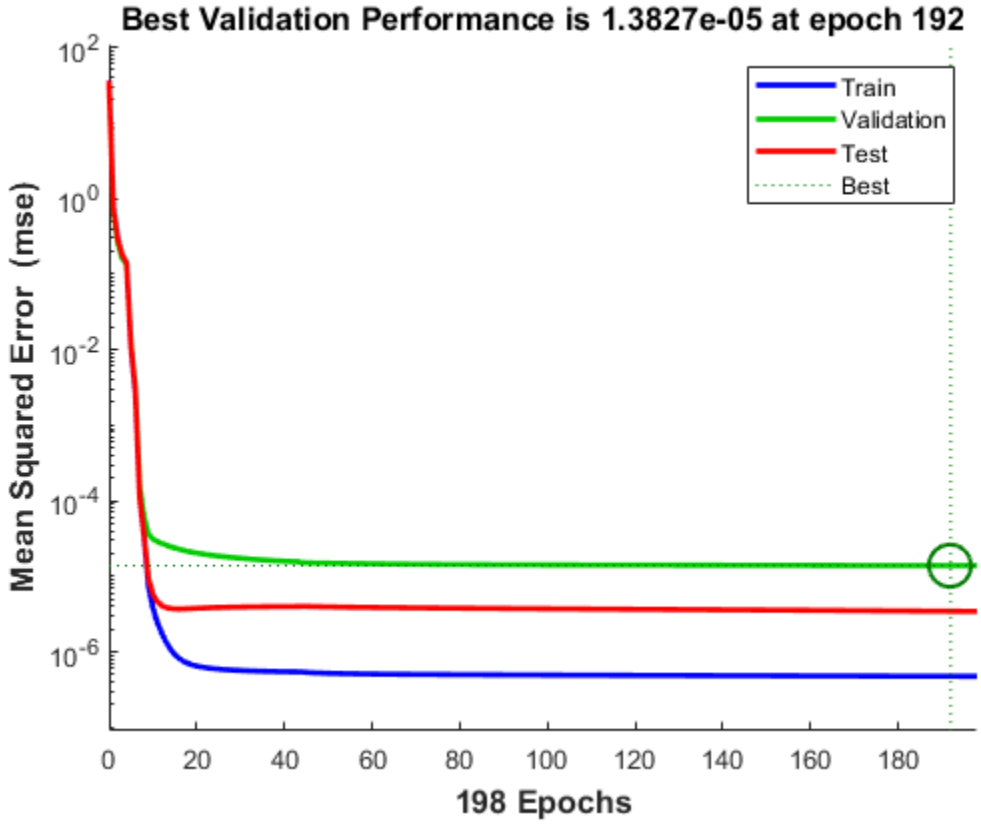

```
nntraintool('close')
```

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

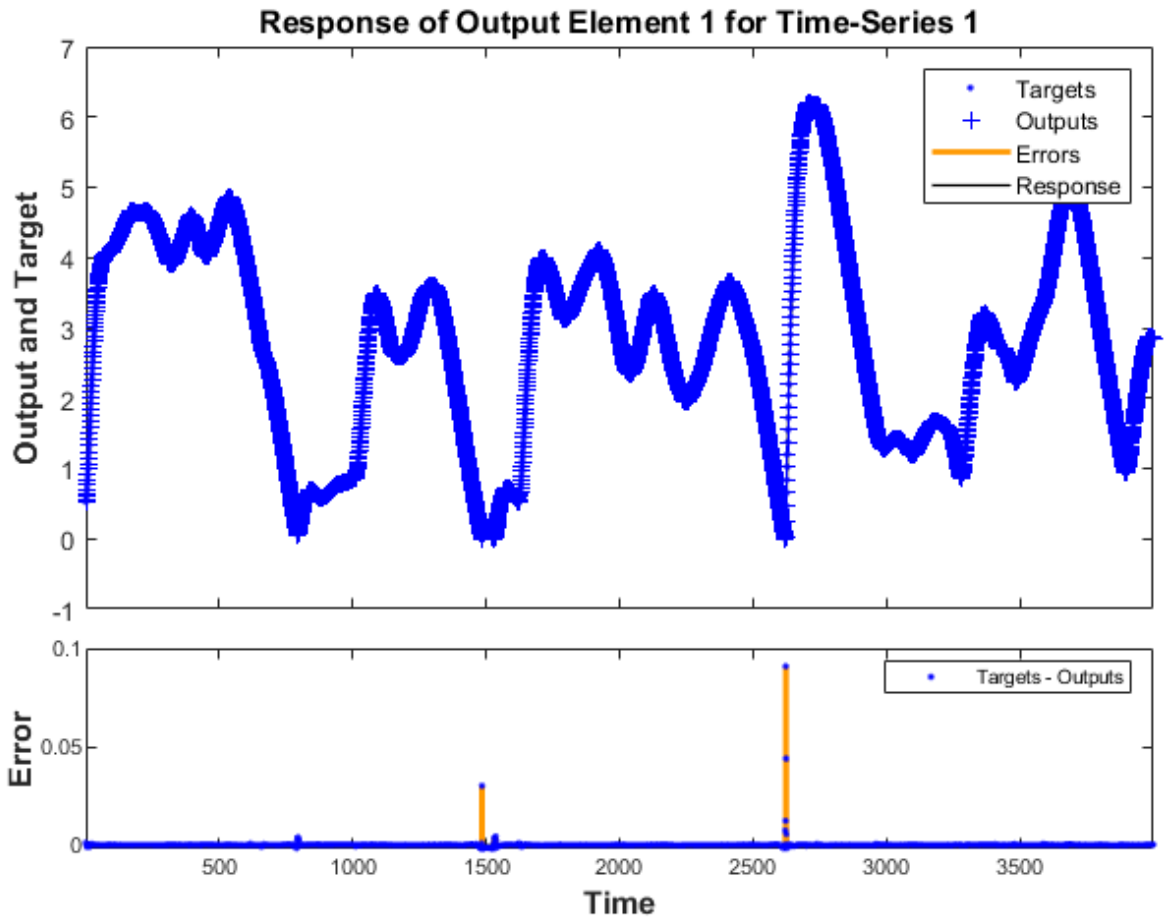
The mean squared error of the trained neural network for all timesteps can now be measured.

```
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.

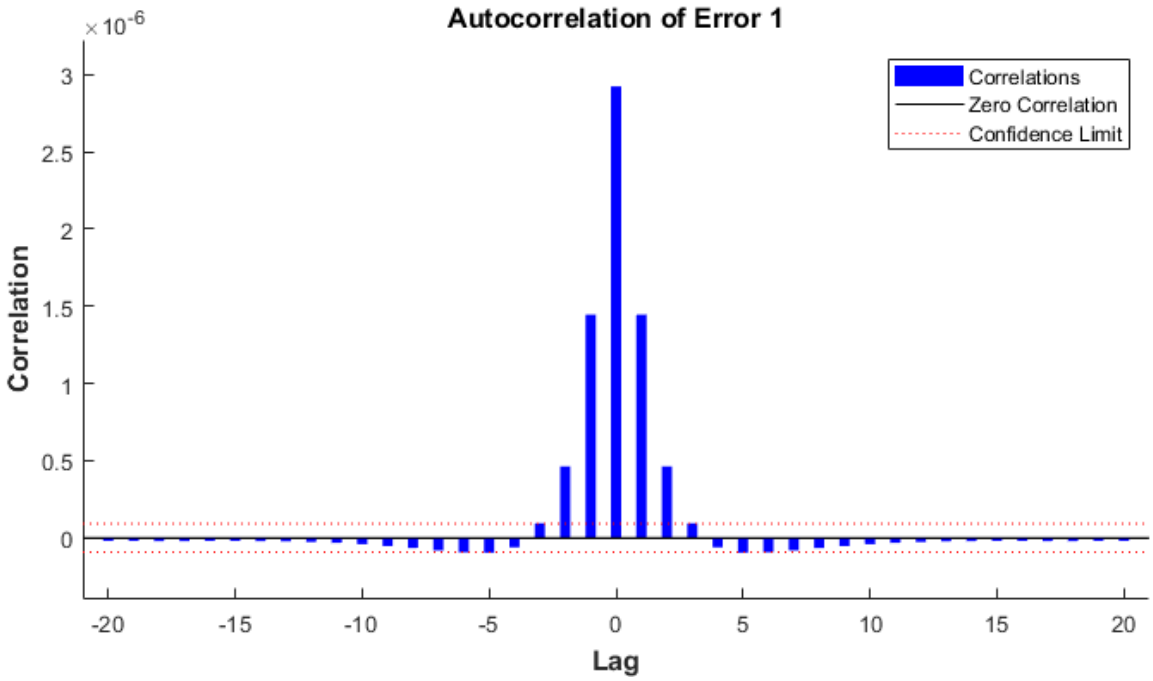
```
plotresponse(Ts,Y)
```



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.

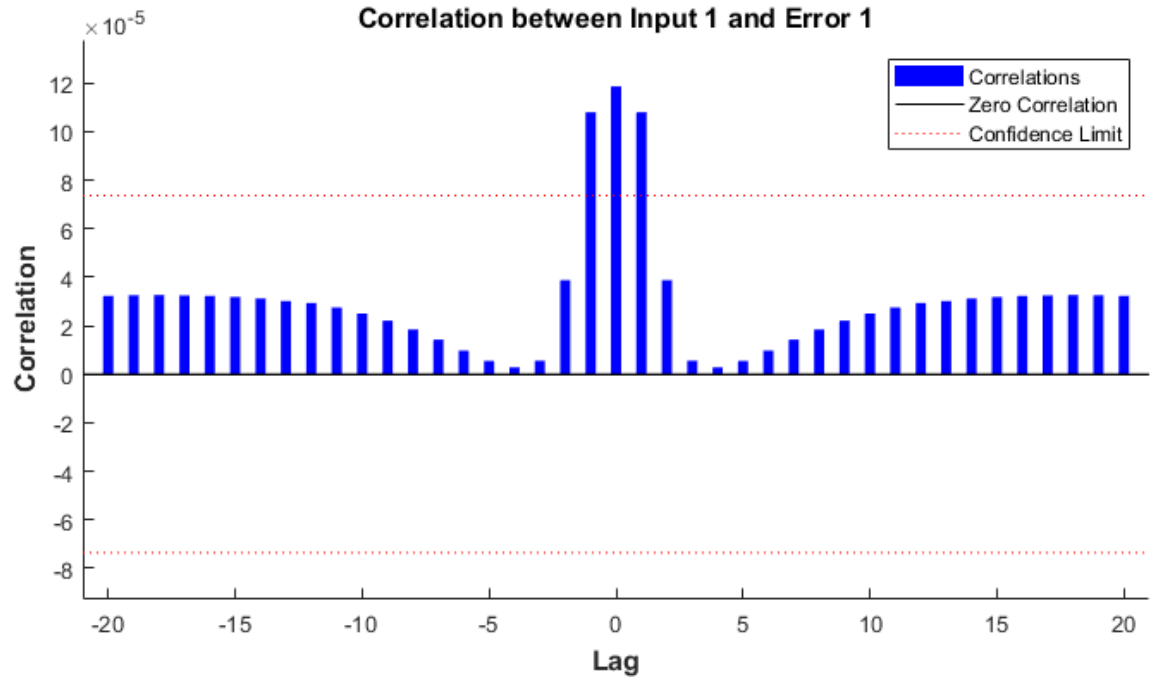
The function GSUBTRACT is used to calculate the error. This function generalizes subtraction to support differences between cell array data.

```
E = gsubtract(Ts,Y);

ploterrcorr(E)
```



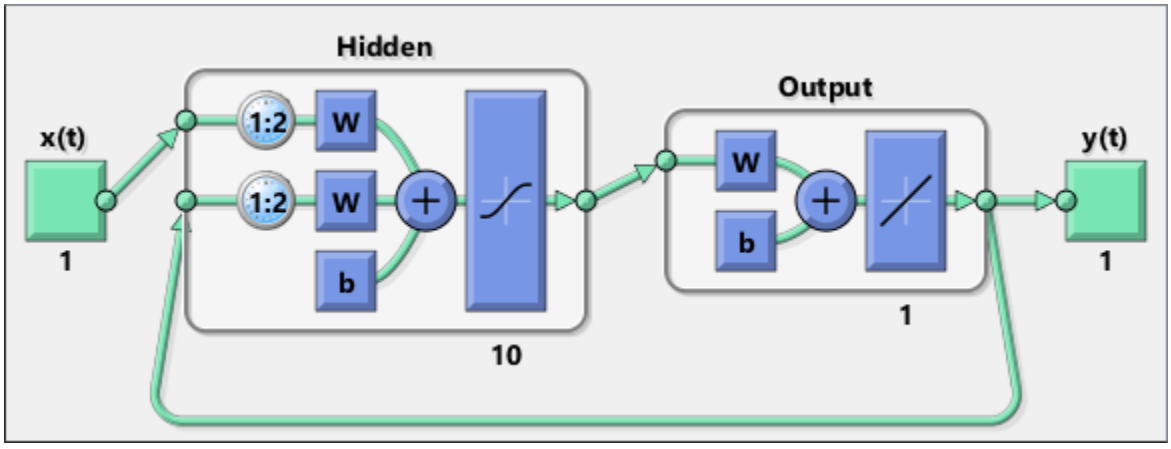
Similarly, PLOTINERRCORR shows the correlation of error with respect to the inputs, with varying degrees of lag. In this case, most or all the lines should fall within the confidence limits, including the center line.

```
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)
```

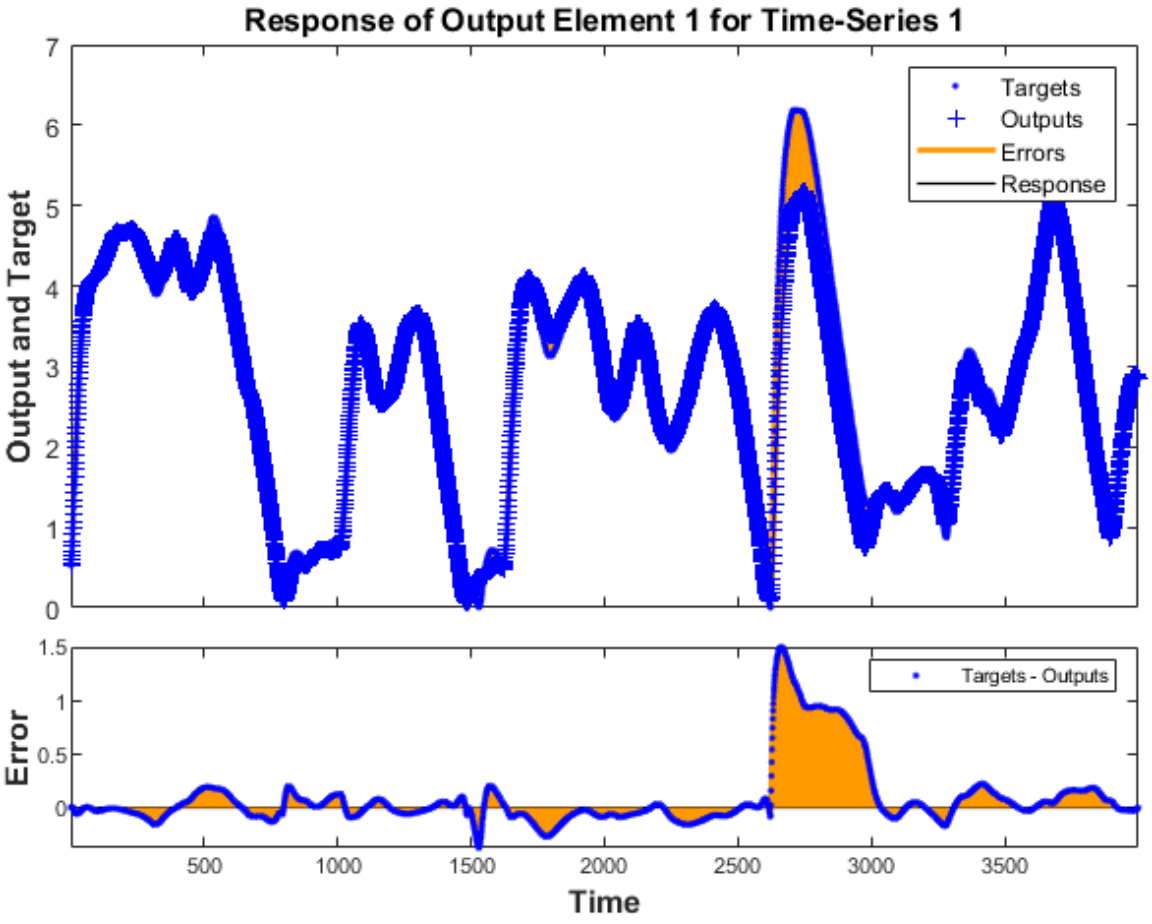


We can simulate the network in closed loop form. In this case the network is only given initial magnet positions, and then must use its own predicted positions recursively to predict new positions.

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.

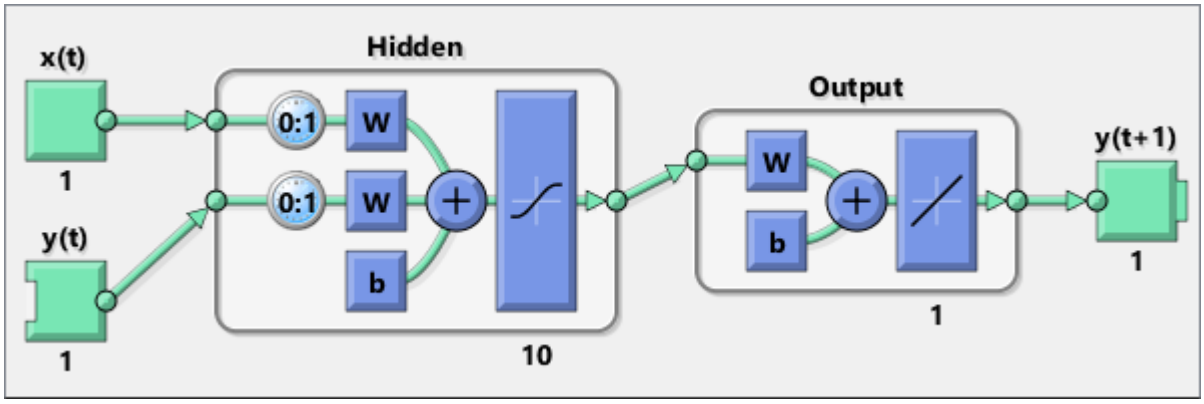
Again, PREPARETS does the work of preparing the time series data for us taking into account the altered network.

```
[Xs,Xi,Ai,Ts] = preparets(net2,x,{},t);
Y = net2(Xs,Xi,Ai);
plotresponse(Ts,Y)
```



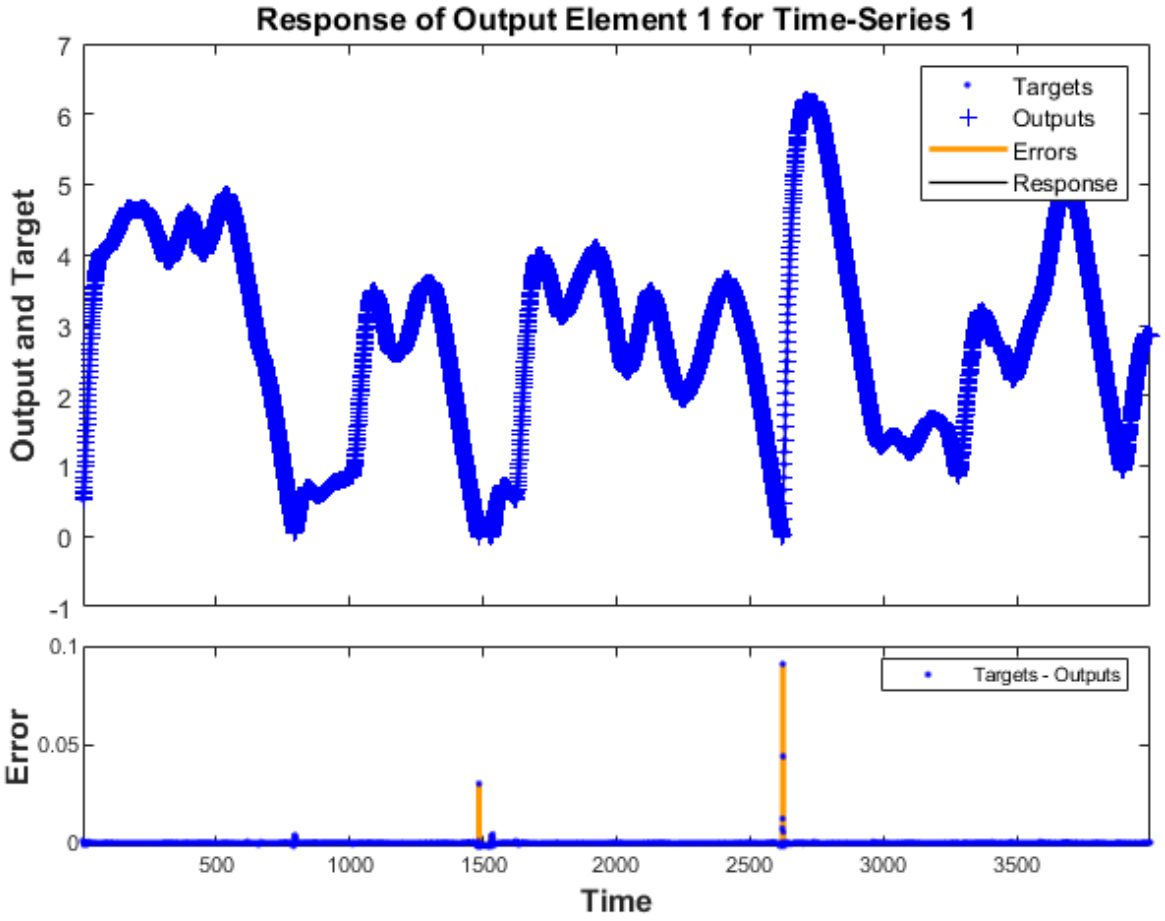
If the application required us to access the predicted magnet position a timestep ahead of when it actually occurs, we can remove a delay from the network so at any given time t, the output is an estimate of the position at time t+1.

```
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.

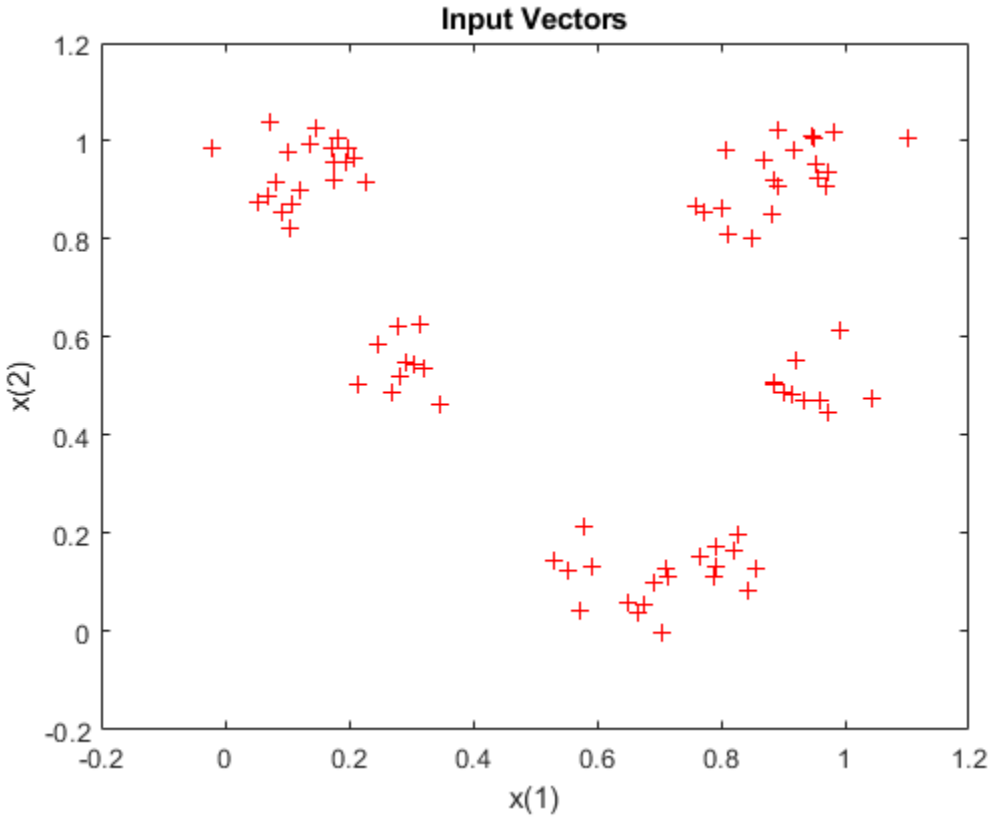
Competitive Learning

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)');
```

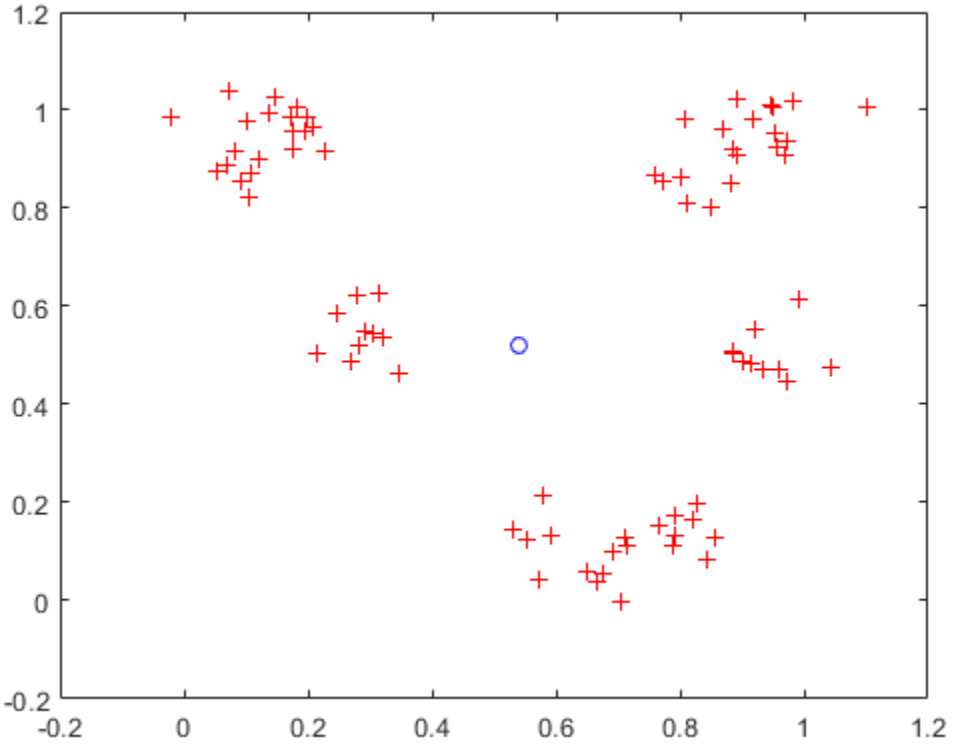


Here COMPETLAYER takes two arguments, the number of neurons and the learning rate.

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).

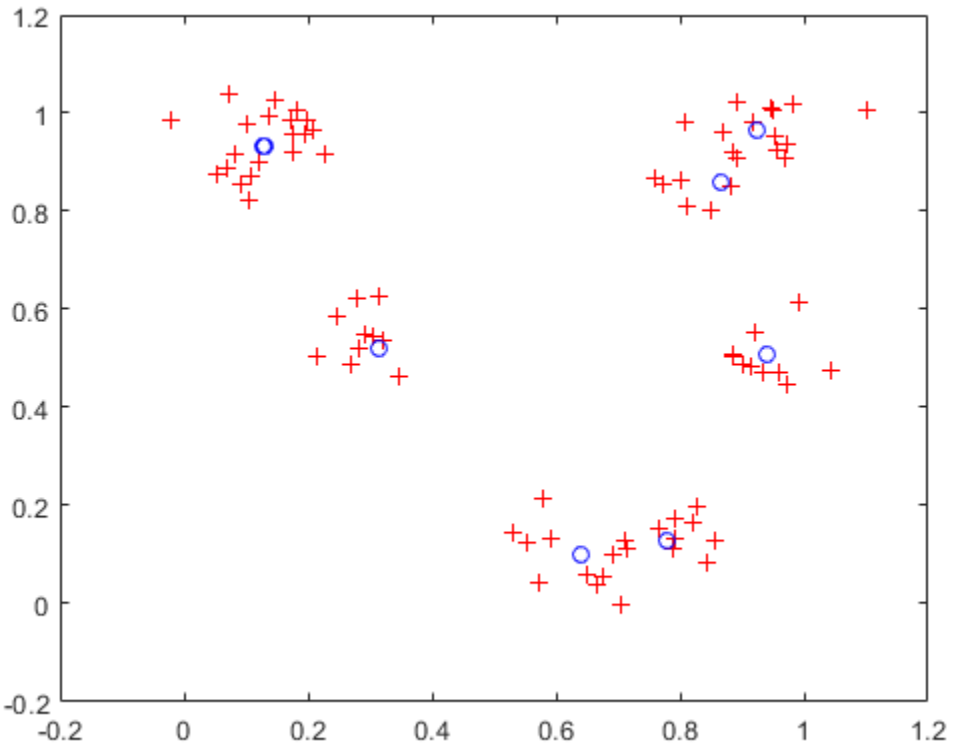
```
net = competlayer(8,.1);
net = configure(net,x);
w = net.IW{1};
plot(x(1,:),x(2,:),'+r');
hold on;
circles = plot(w(:,1),w(:,2),'ob');
```



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
```

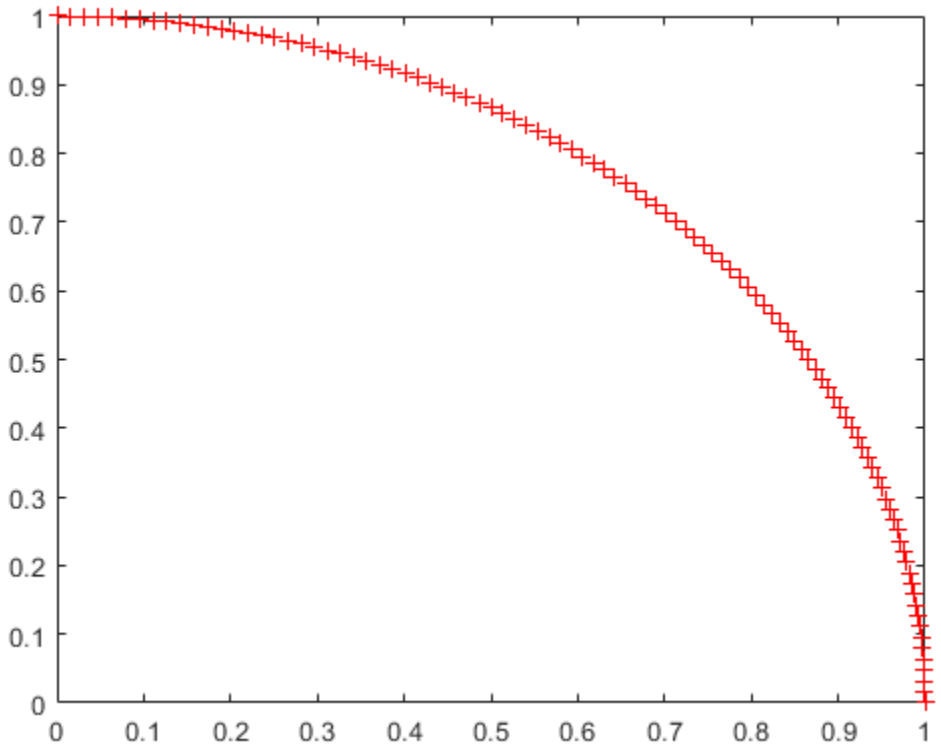
One-Dimensional Self-organizing Map

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);
```

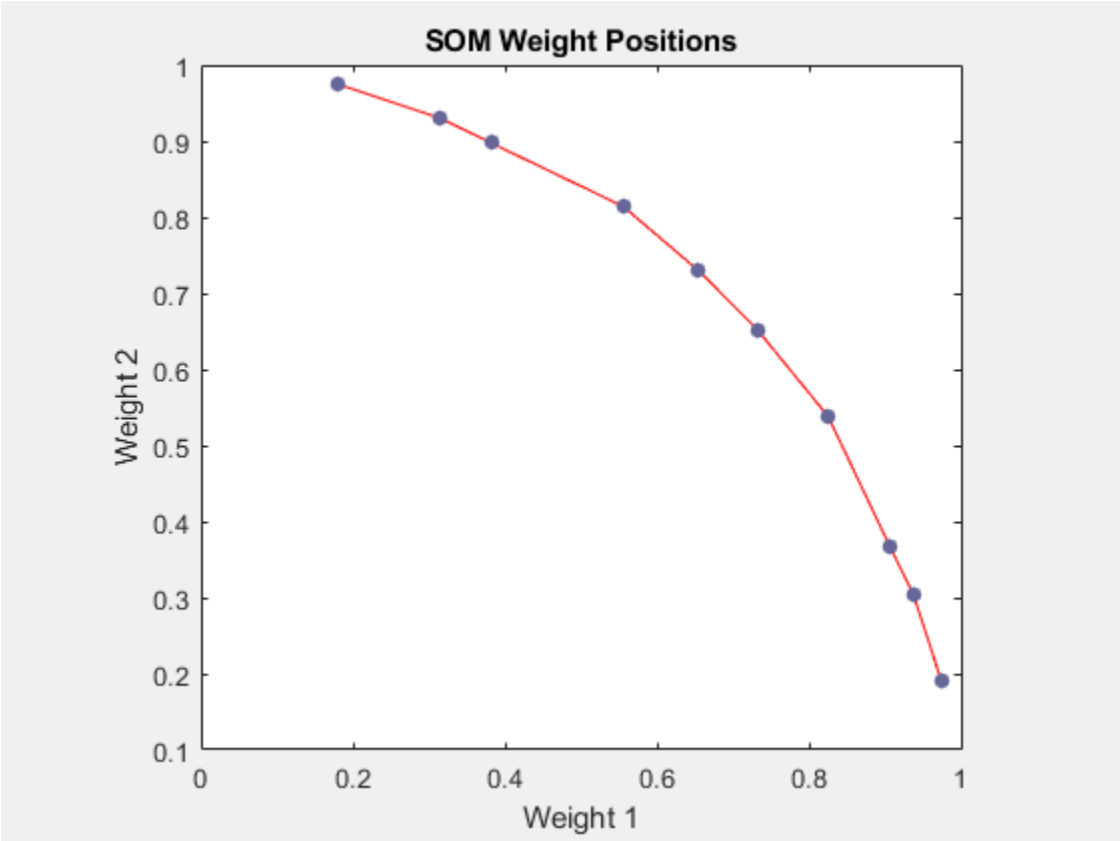
Specify that the network is to be trained for 10 epochs and use `train` to train the network on the input data.

```
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)
```

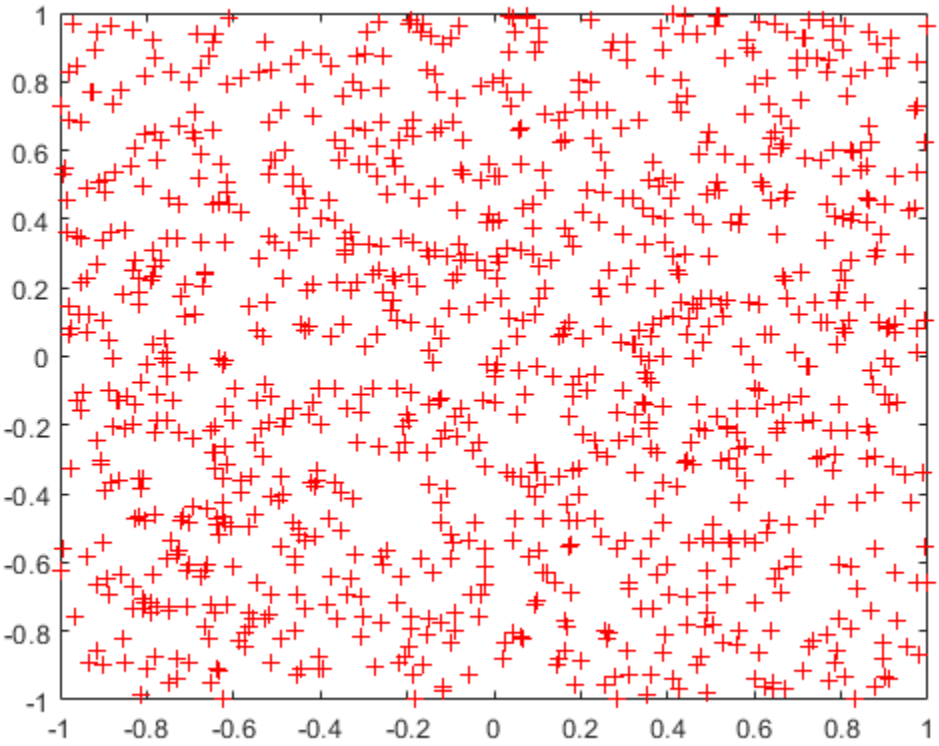
```
a = 10×1

     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.

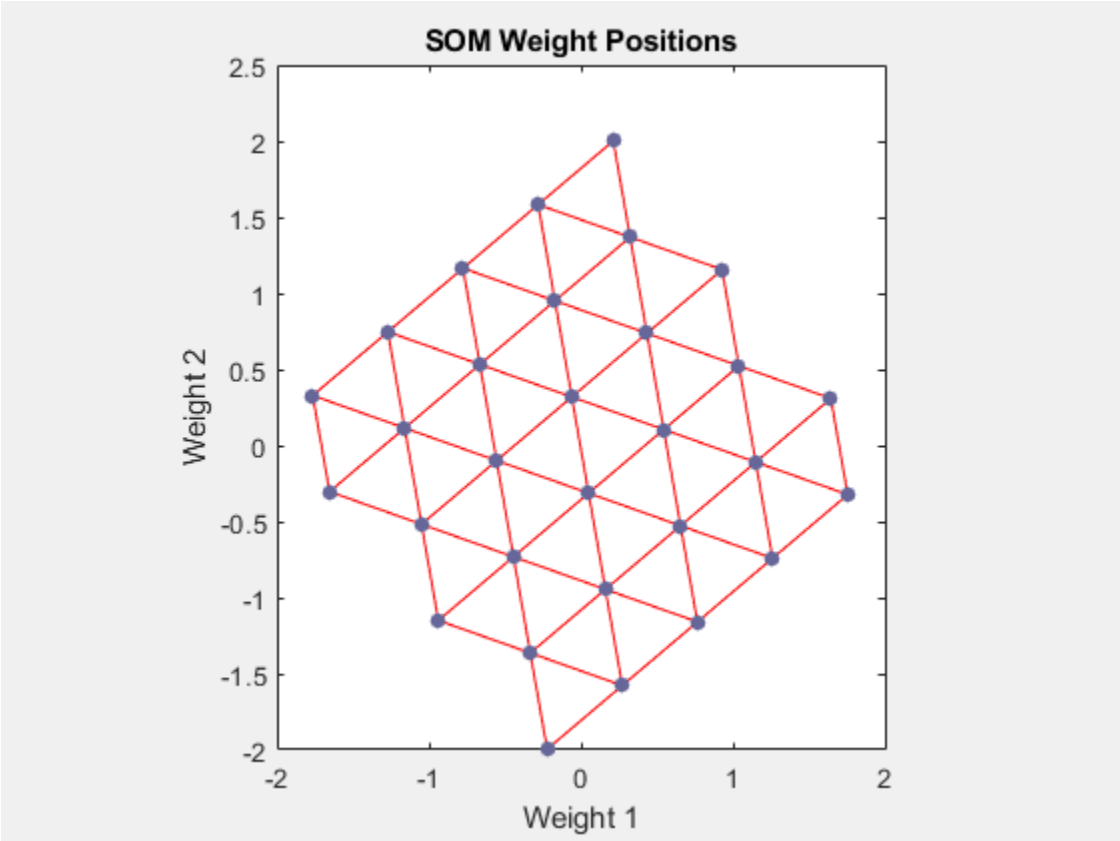
The network is configured to match the dimensions of the inputs. This step is required here because we will plot the initial weights. Normally configuration is performed automatically when training.

```
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.

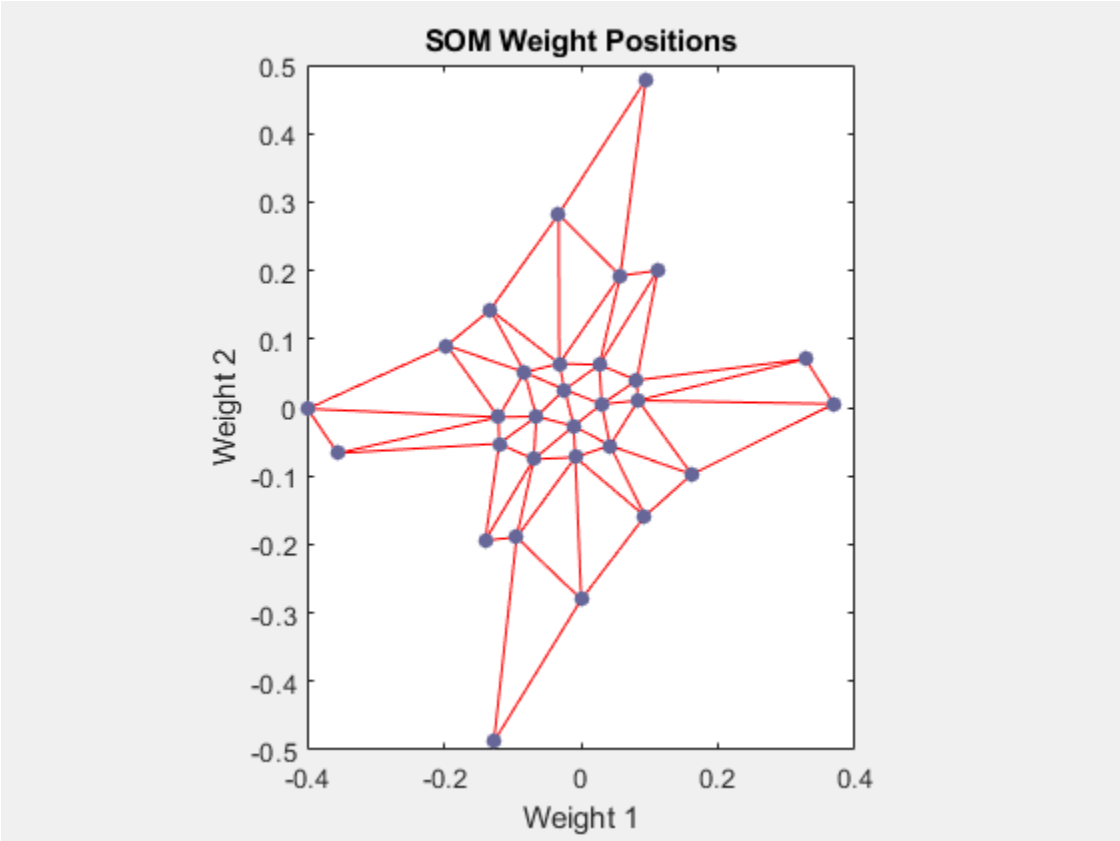
```
plotsompos(net)
```



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)
```

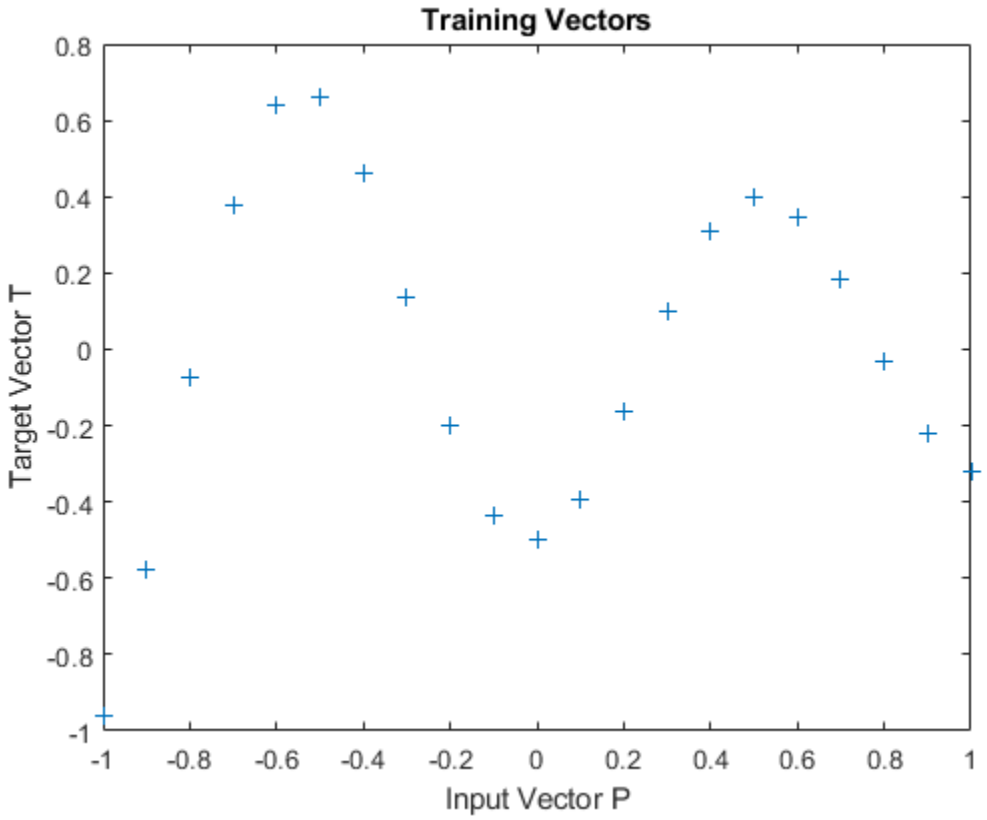
```
y = 30×1

     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.

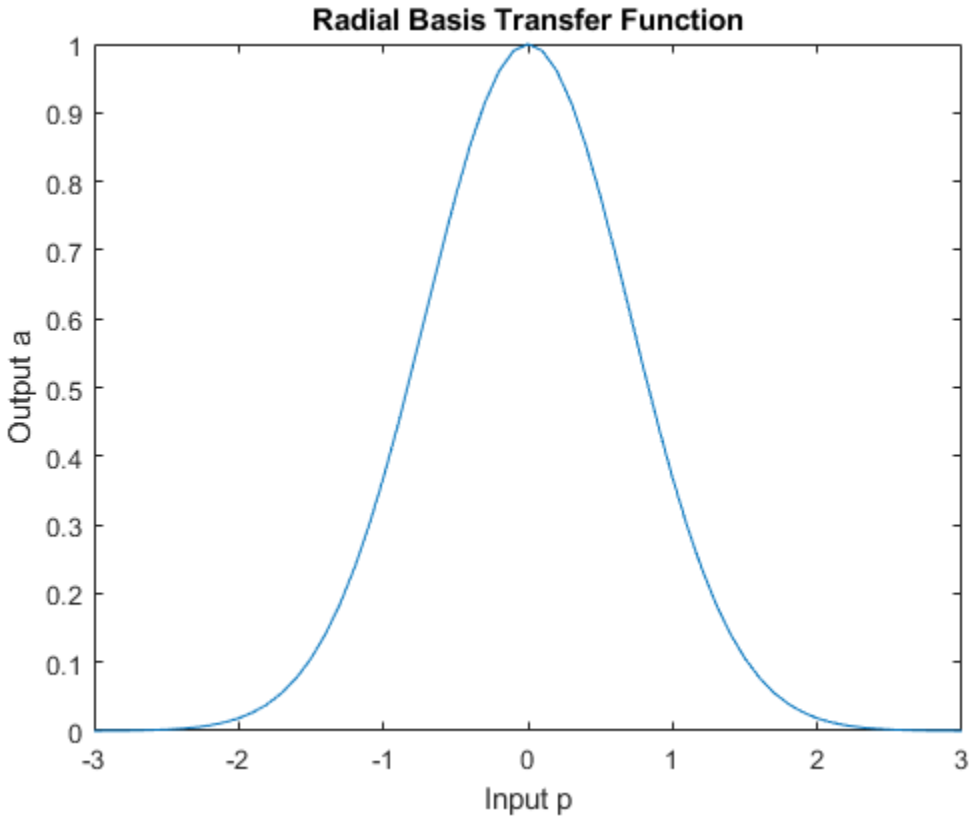
Define 21 inputs P and associated targets T.

```
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');
```



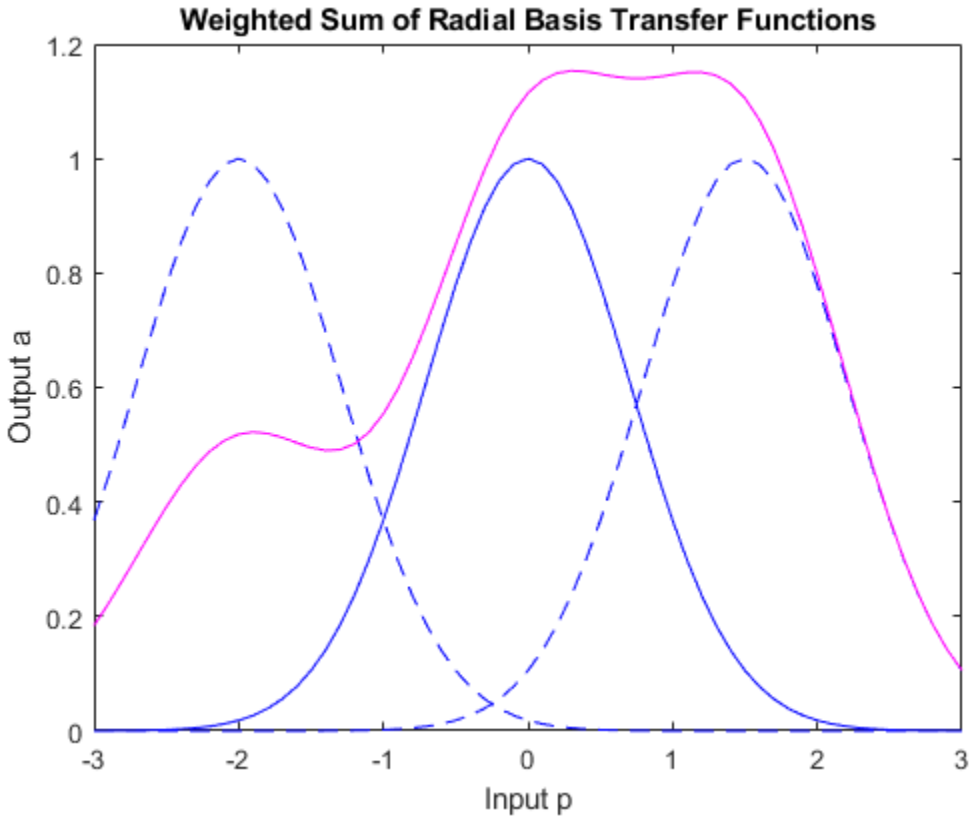
We would like to find a function which fits the 21 data points. One way to do this is with a radial basis network. A radial basis network is a network with two layers. A hidden layer of radial basis neurons and an output layer of linear neurons. Here is the radial basis transfer function used by the 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');
```



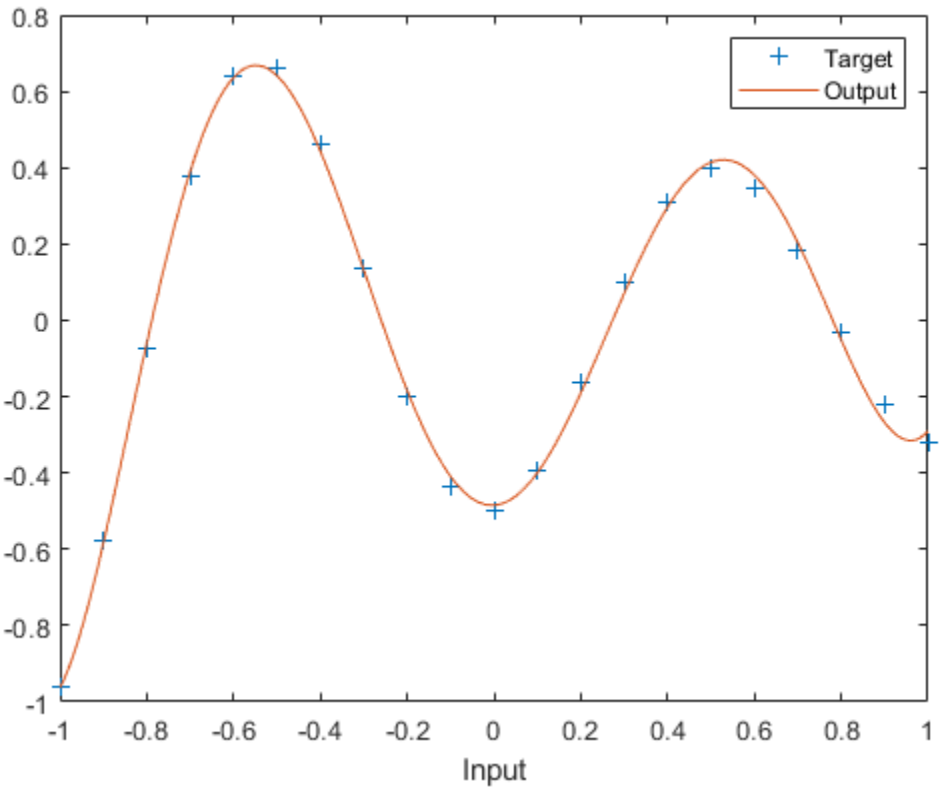
The function NEWRB quickly creates a radial basis network which approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant.

```
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'})
```

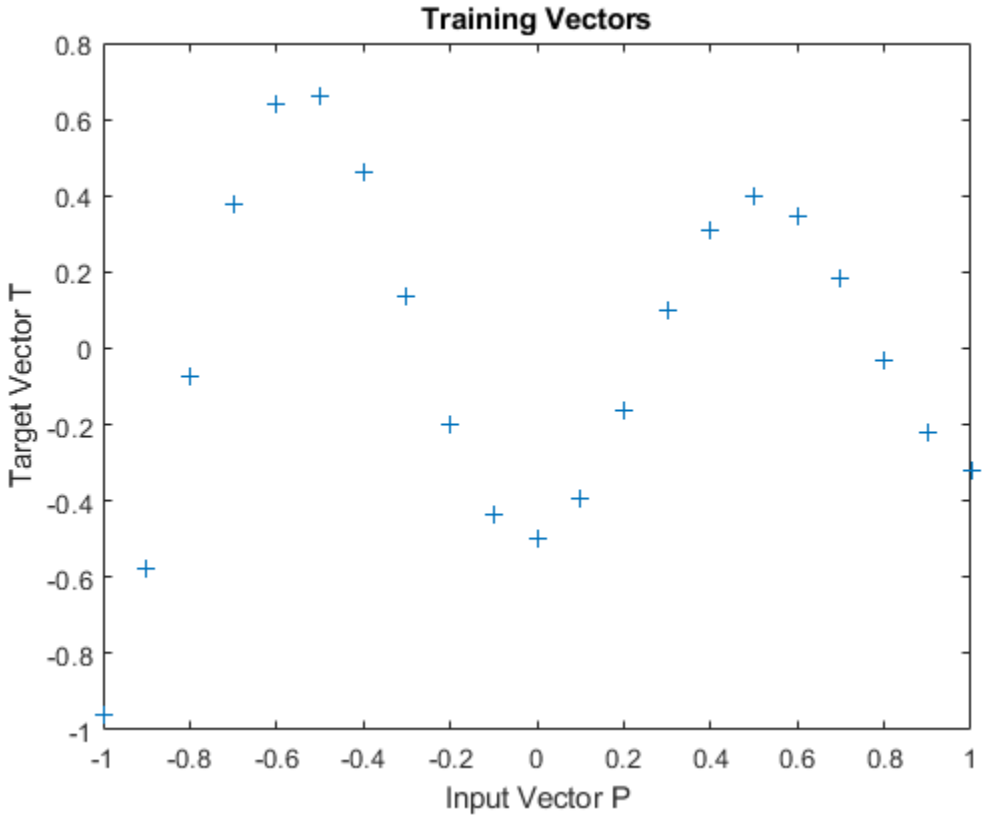

Target
Output
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
Input

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');
```



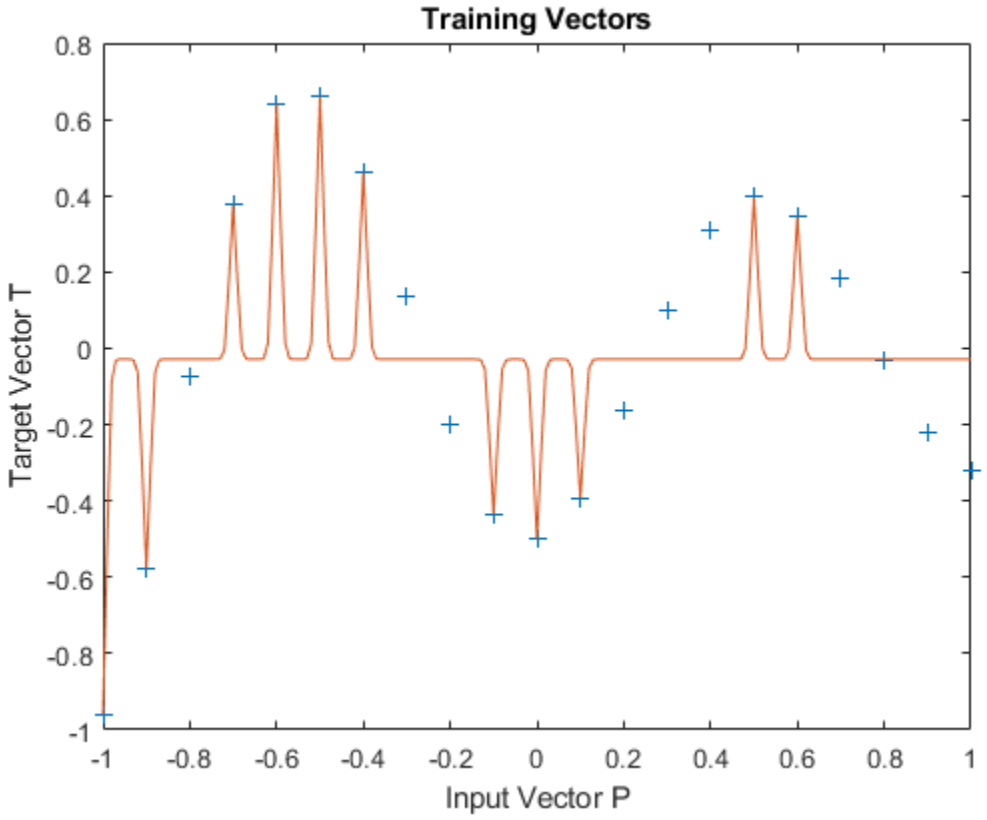
The function NEWRB quickly creates a radial basis network which approximates the function defined by P and T. In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant. The spread of the radial basis neurons B is set to a very small number.

```
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;
```

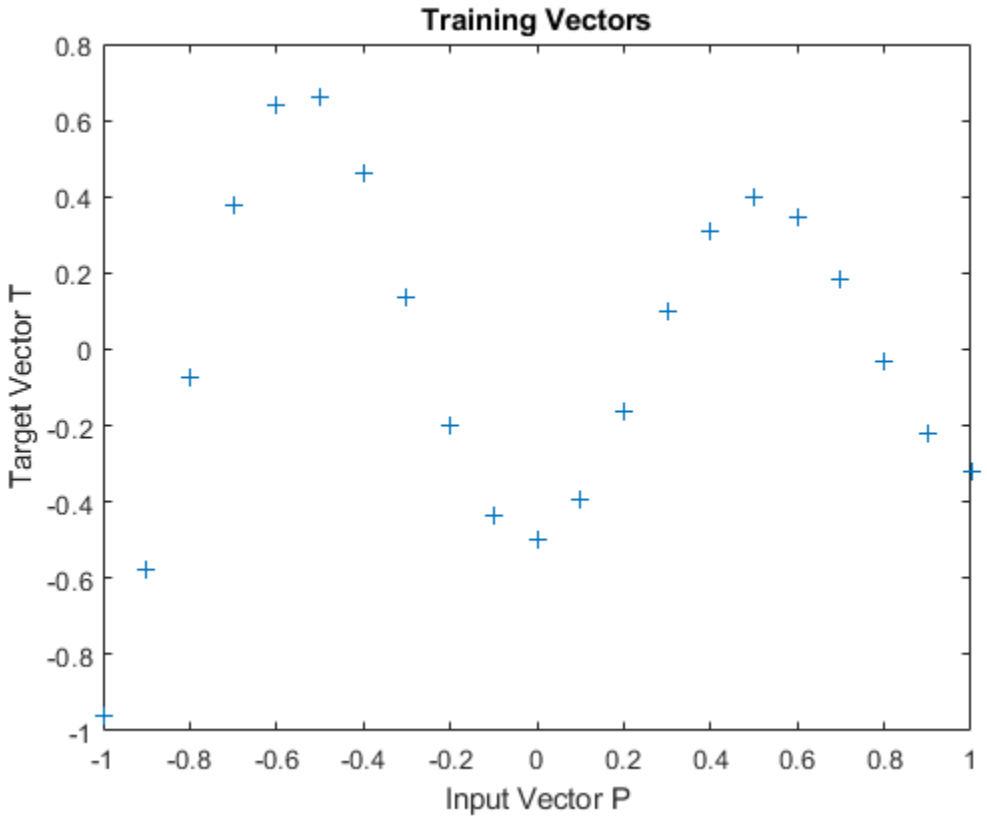


Radial Basis Overlapping Neurons

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.

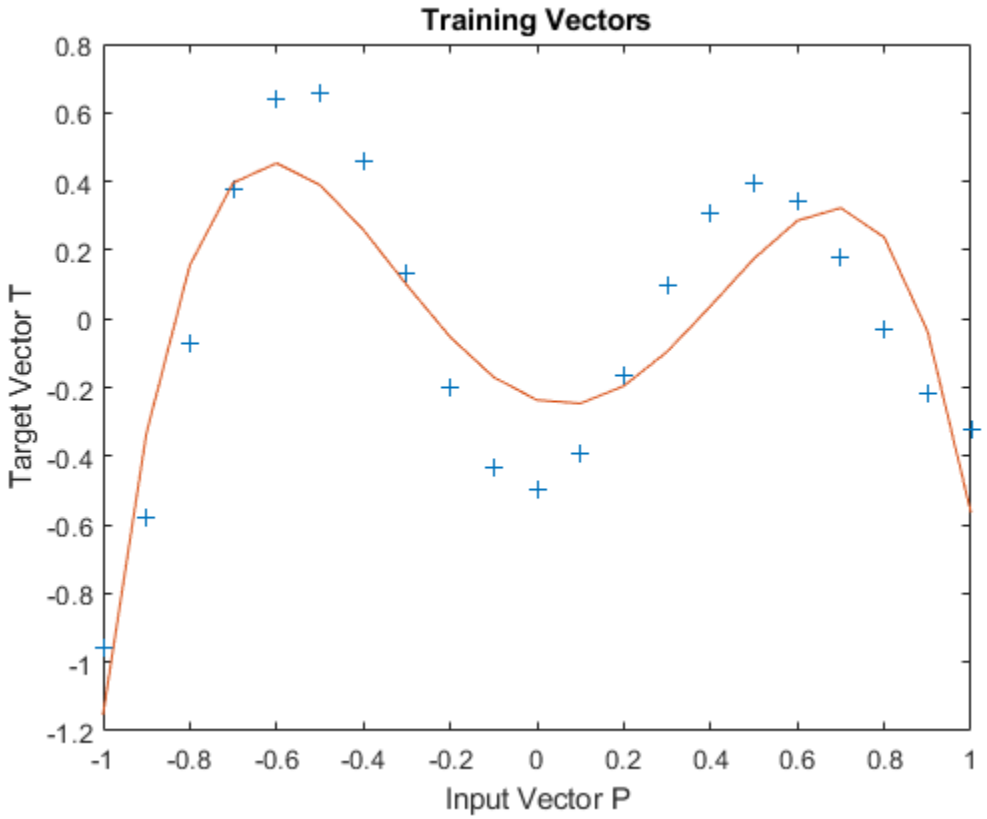
In addition to the training set and targets, NEWRB takes two arguments, the sum-squared error goal and the spread constant. The spread of the radial basis neurons B is set to a very large number.

```
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;
```

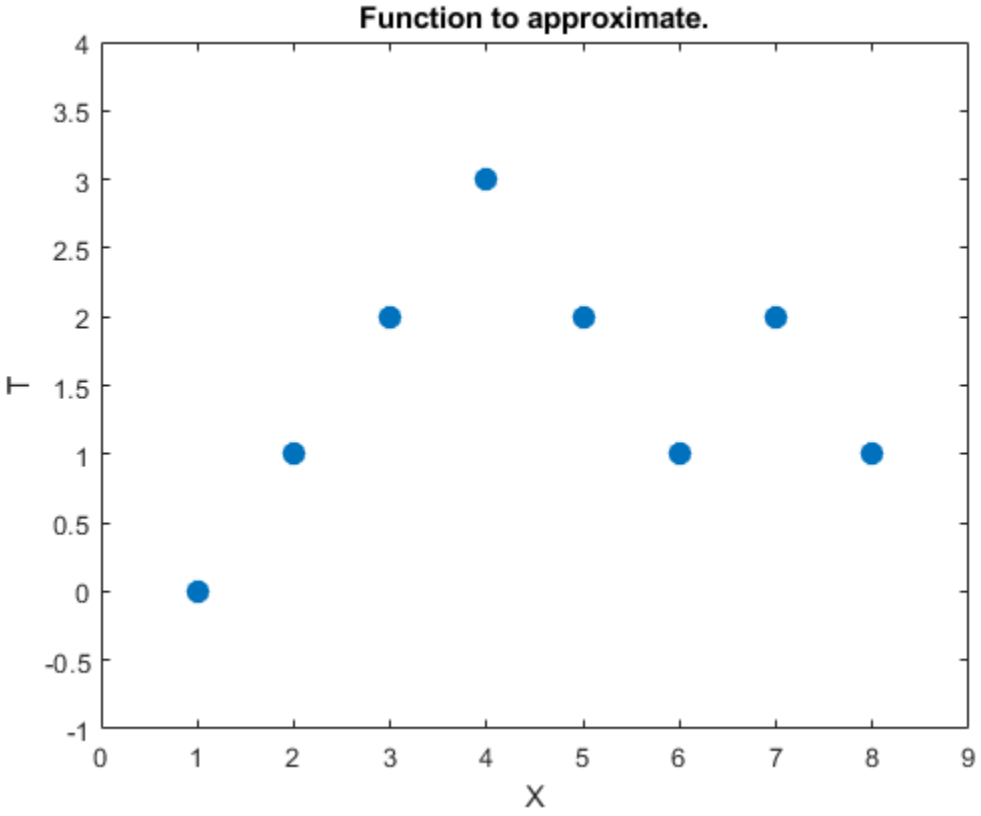


GRNN Function Approximation

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.

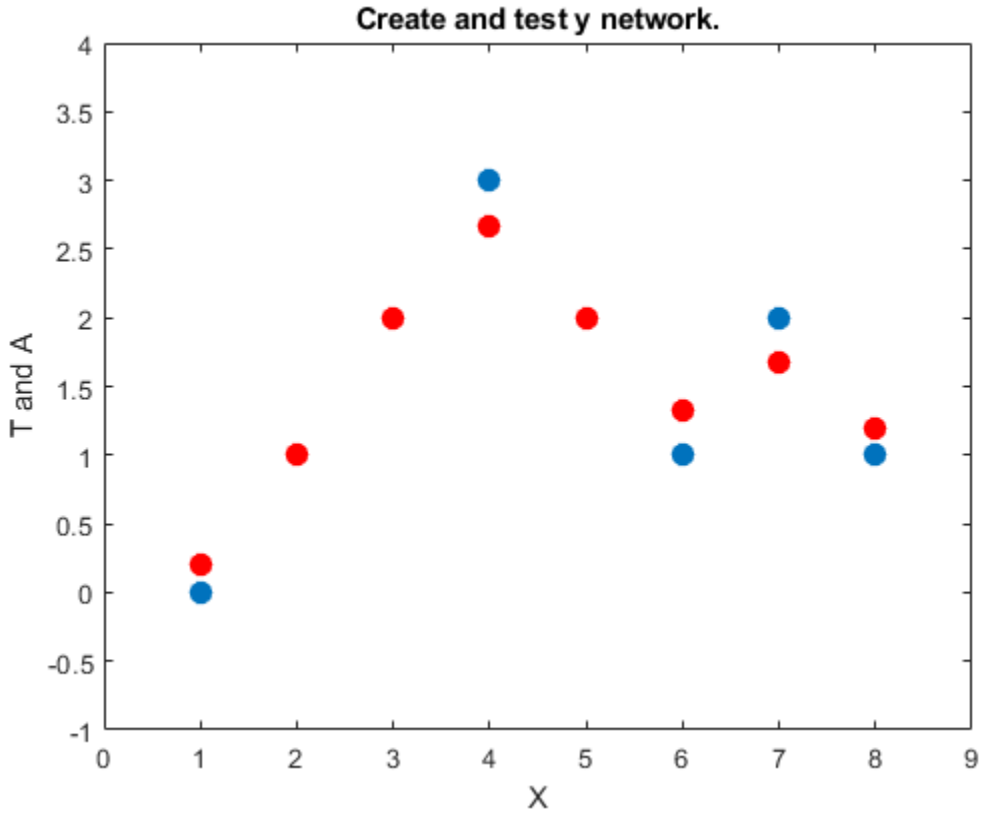
```
X = [1 2 3 4 5 6 7 8];
T = [0 1 2 3 2 1 2 1];

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.

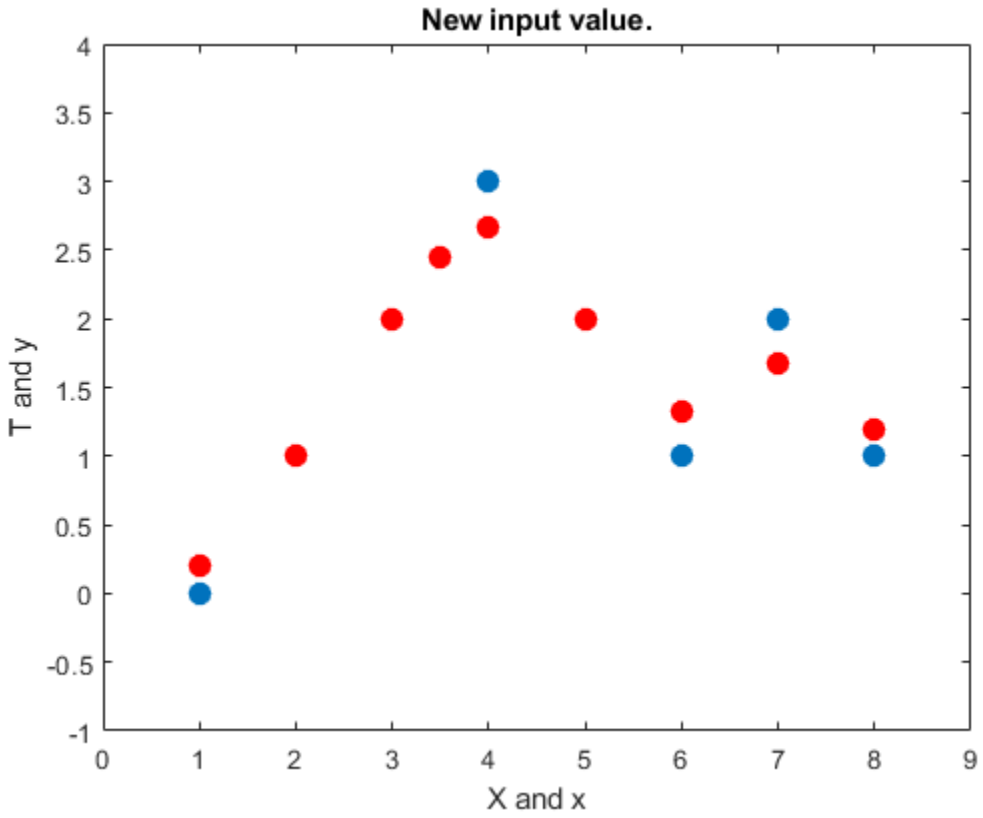
```
spread = 0.7;
net = newgrnn(X,T,spread);
A = net(X);

hold on
outputline = plot(X,A,'.','markersize',30,'color',[1 0 0]);
title('Create and test y network.')
```

```
xlabel('X')
ylabel('T and A')
```



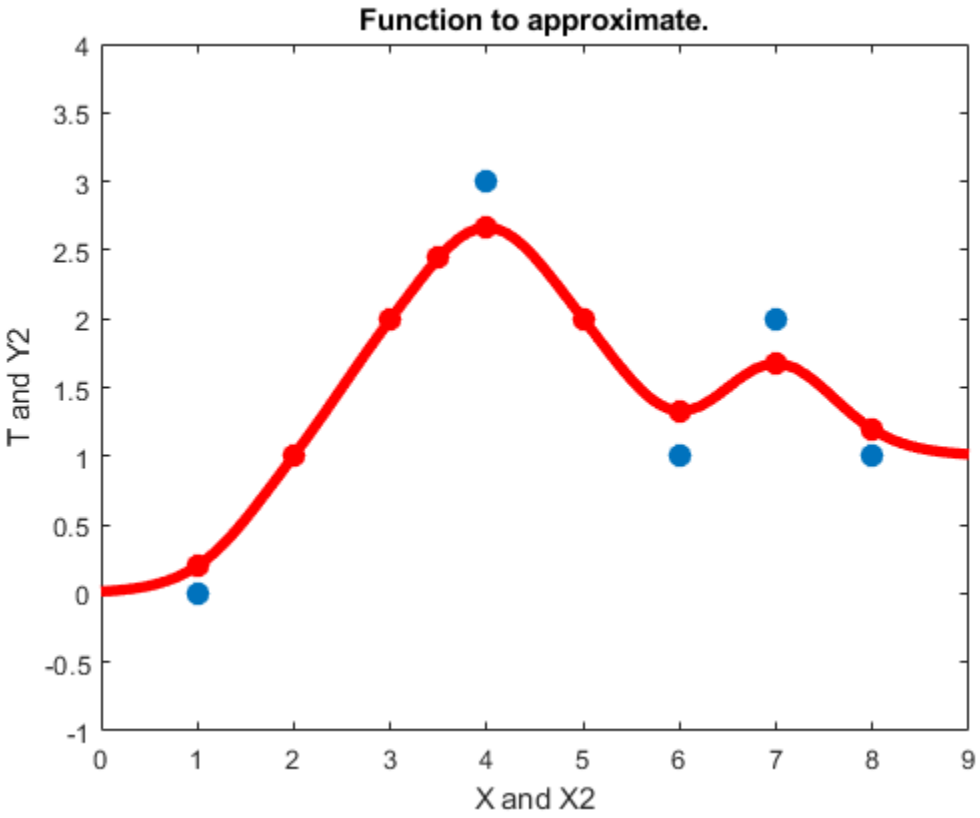
We can use the network to approximate the function at y new input value.

```
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')
```

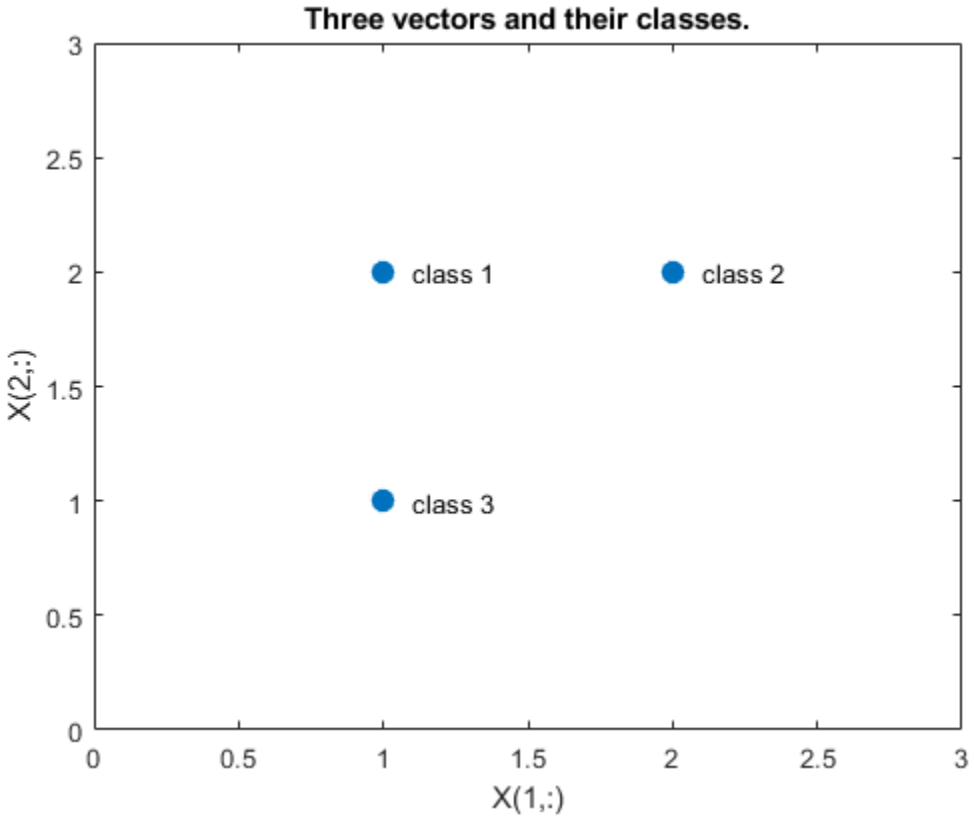

Function to approximate.
T and Y2
X and X2
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
7
8
9

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,:)')
```

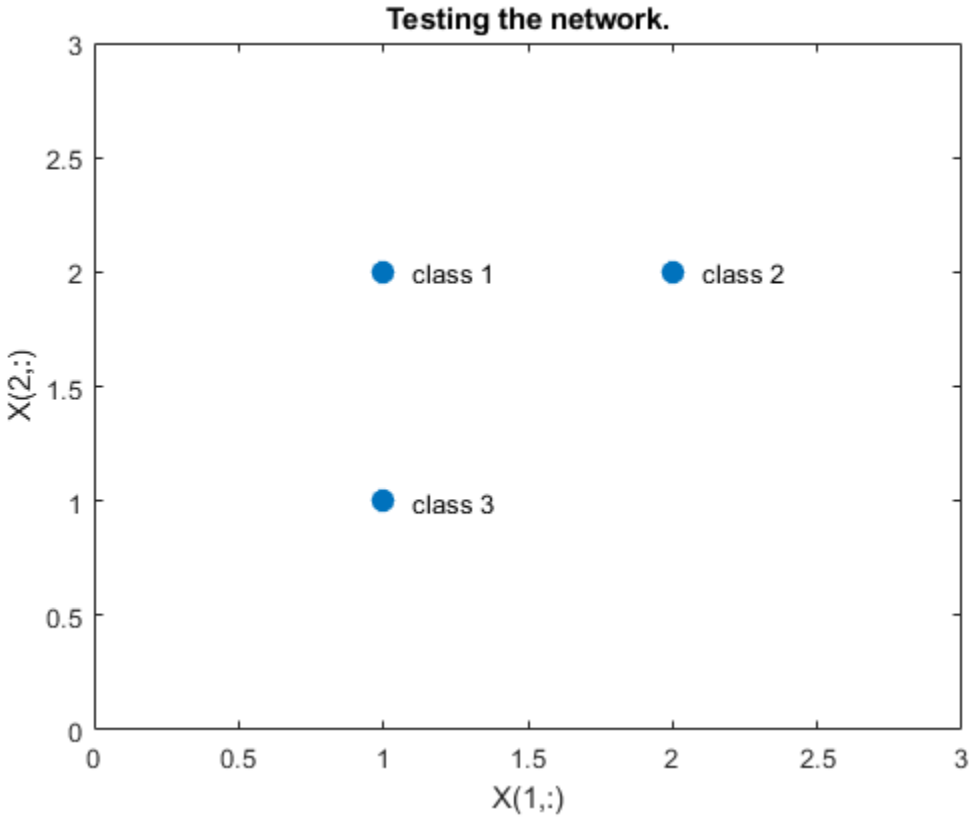


First we convert the target class indices Tc to vectors T. Then we design y probabilistic neural network with NEWPNN. We use y SPREAD value of 1 because that is y typical distance between the input vectors.

```
T = ind2vec(Tc);
spread = 1;
net = newpnn(X,T,spread);
```

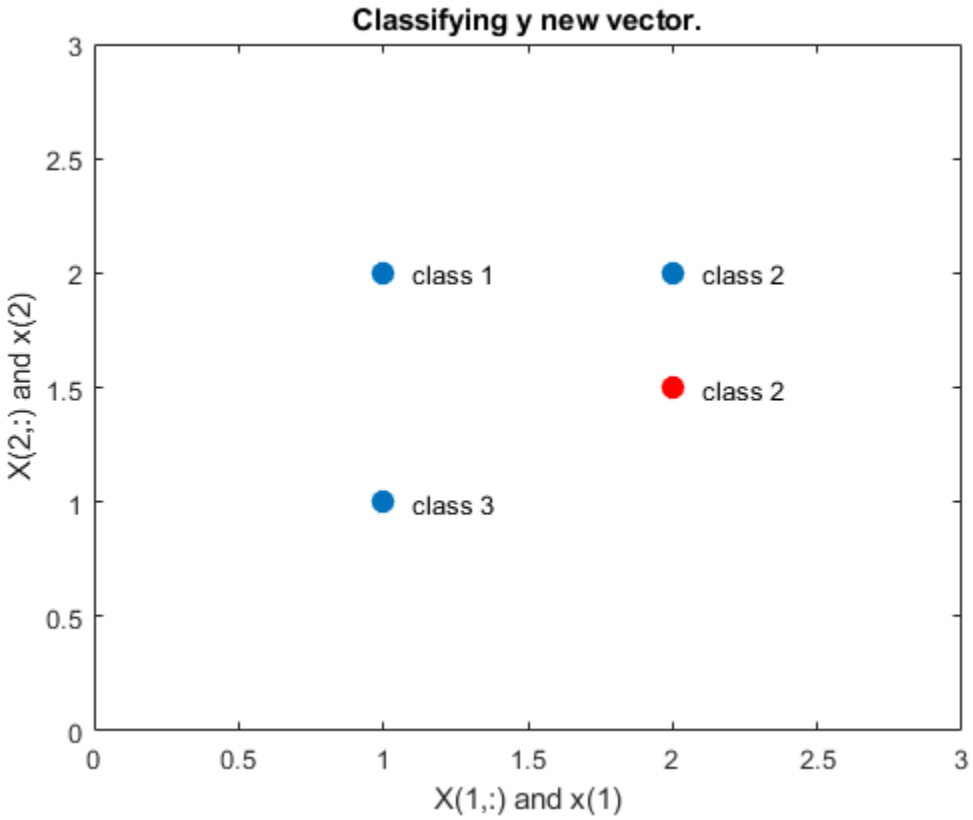
Now we test the network on the design input vectors. We do this by simulating the network and converting its vector outputs to indices.

```
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,:)')
```



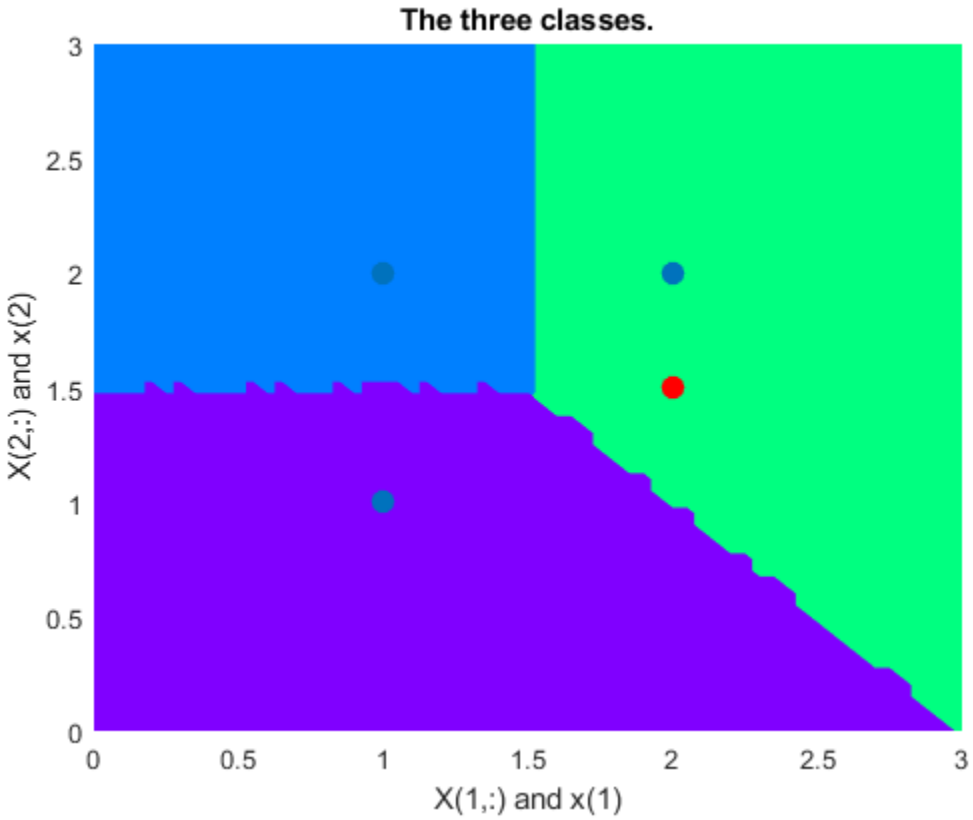
Let's classify y new vector with our network.

```
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

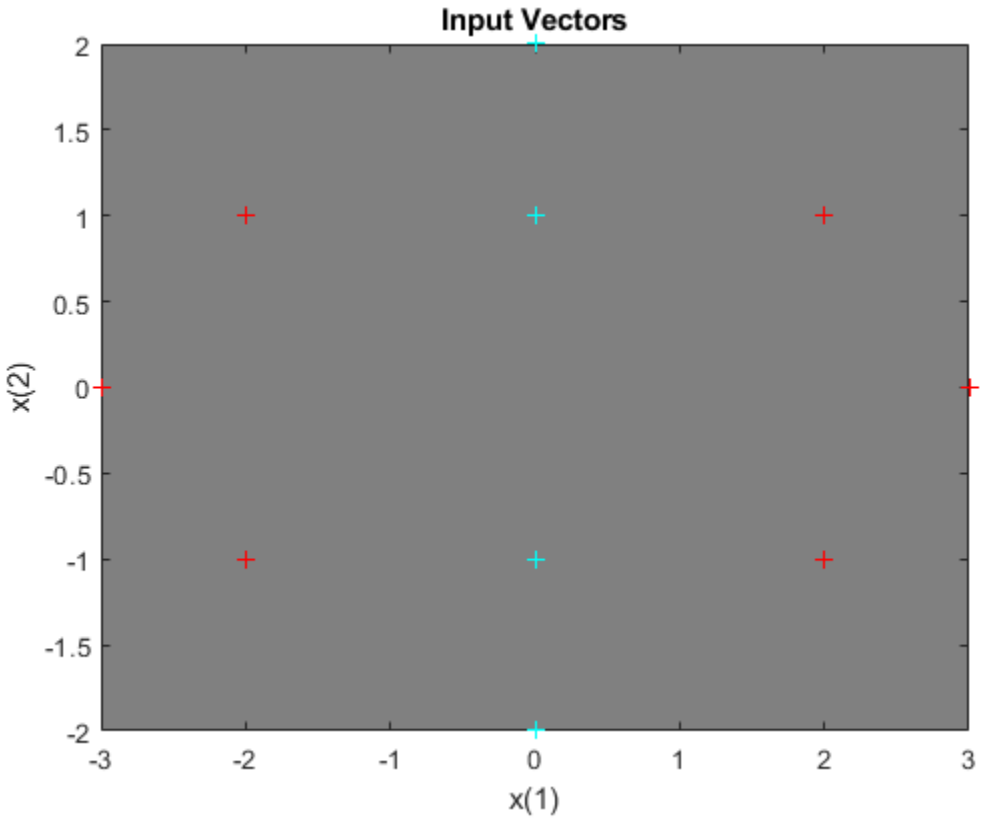
An LVQ network is trained to classify input vectors according to given targets.

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)');
```

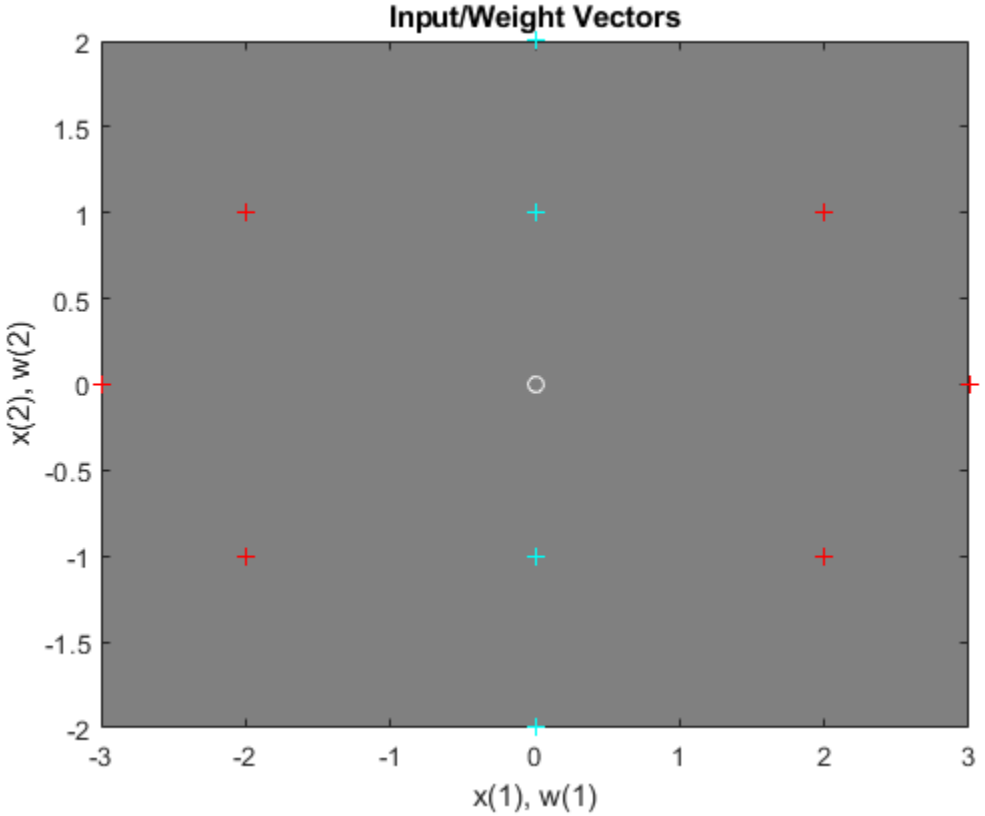


Here LVQNET creates an LVQ layer with four hidden neurons and a learning rate of 0.1. The network is then configured for inputs X and targets T. (Configuration normally an unnecessary step as it is done automatically by TRAIN.)

```
net = lvqnet(4,0.1);
net = configure(net,x,t);
```

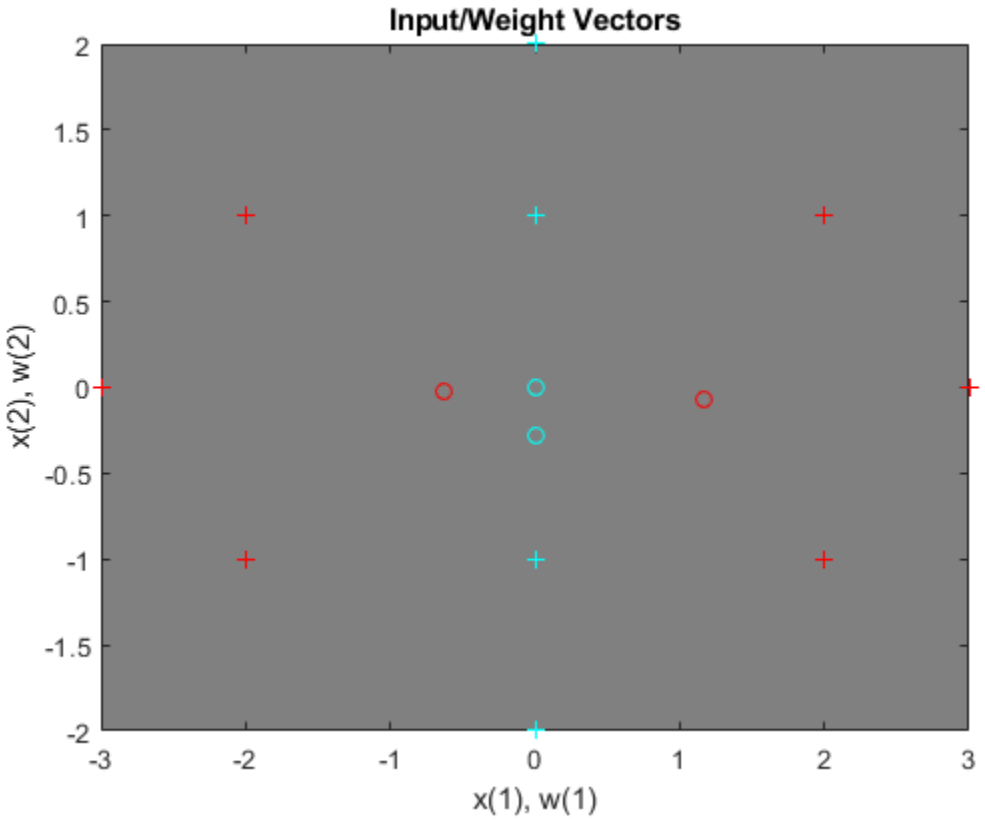
The competitive neuron weight vectors are plotted as follows.

```
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.

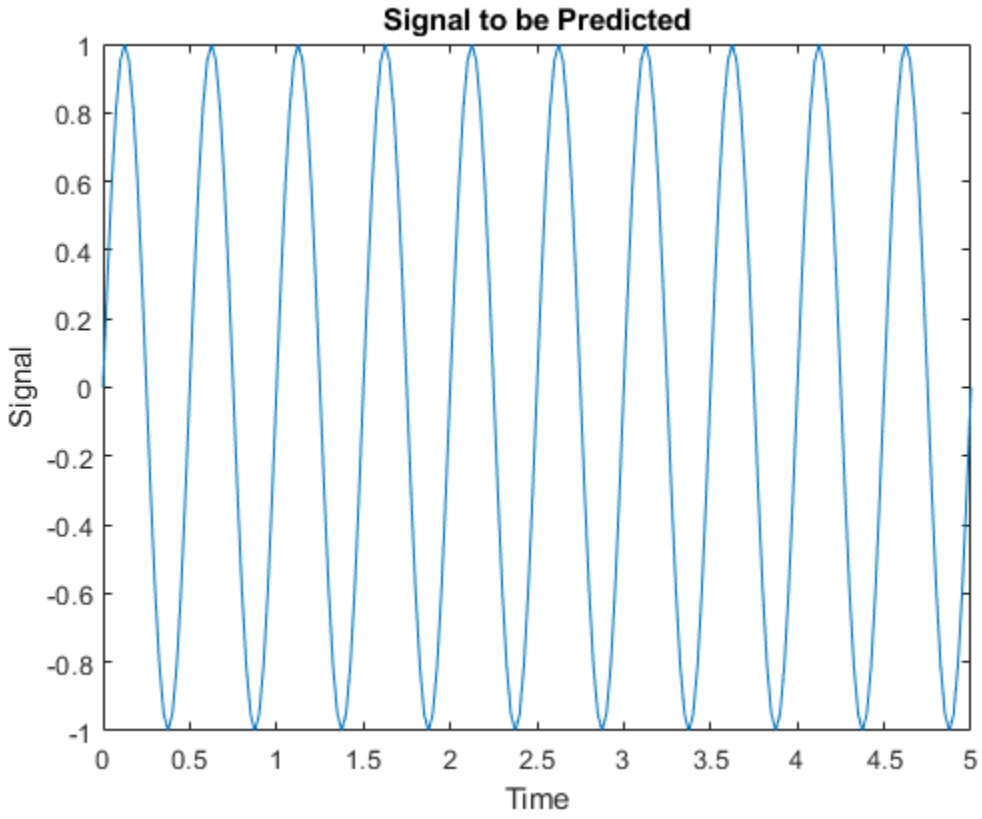
Defining a Wave Form

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));
```

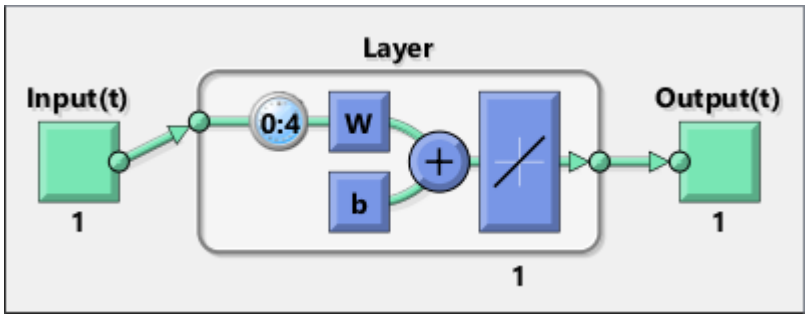
The targets are now defined to match the inputs, but shifted earlier by one timestep.

```
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)
```



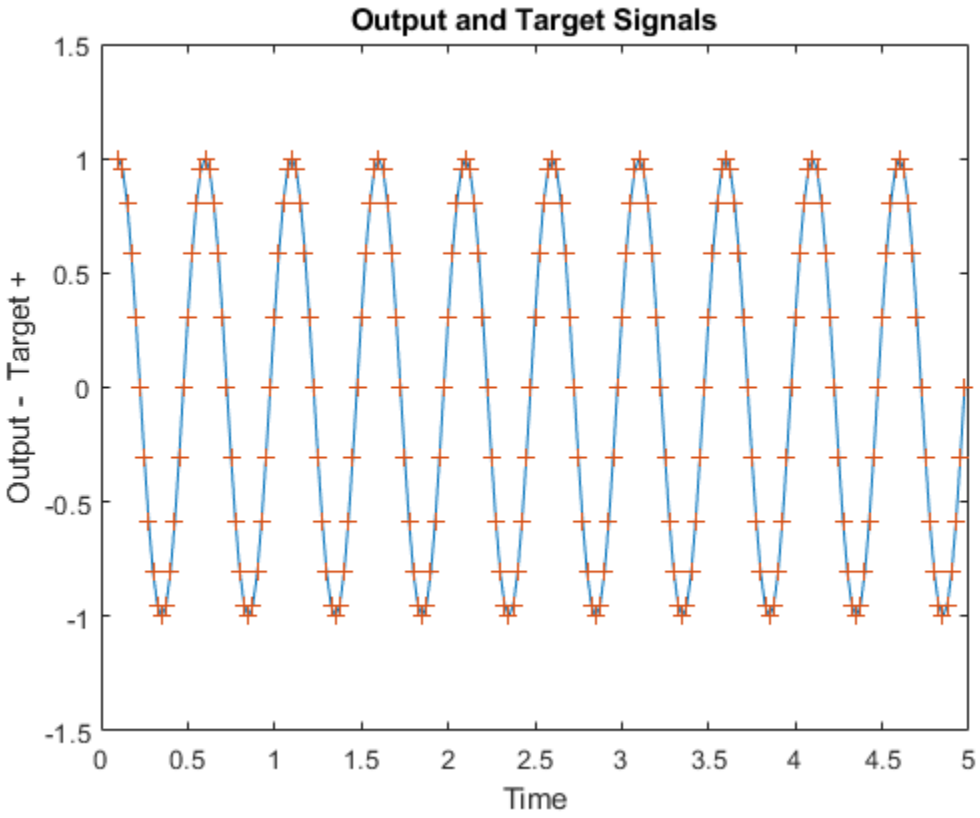
Testing the Linear Layer

The network can now be called like a function on the inputs and delayed states to get its time response.

```
Y = net(X,Xi);
```

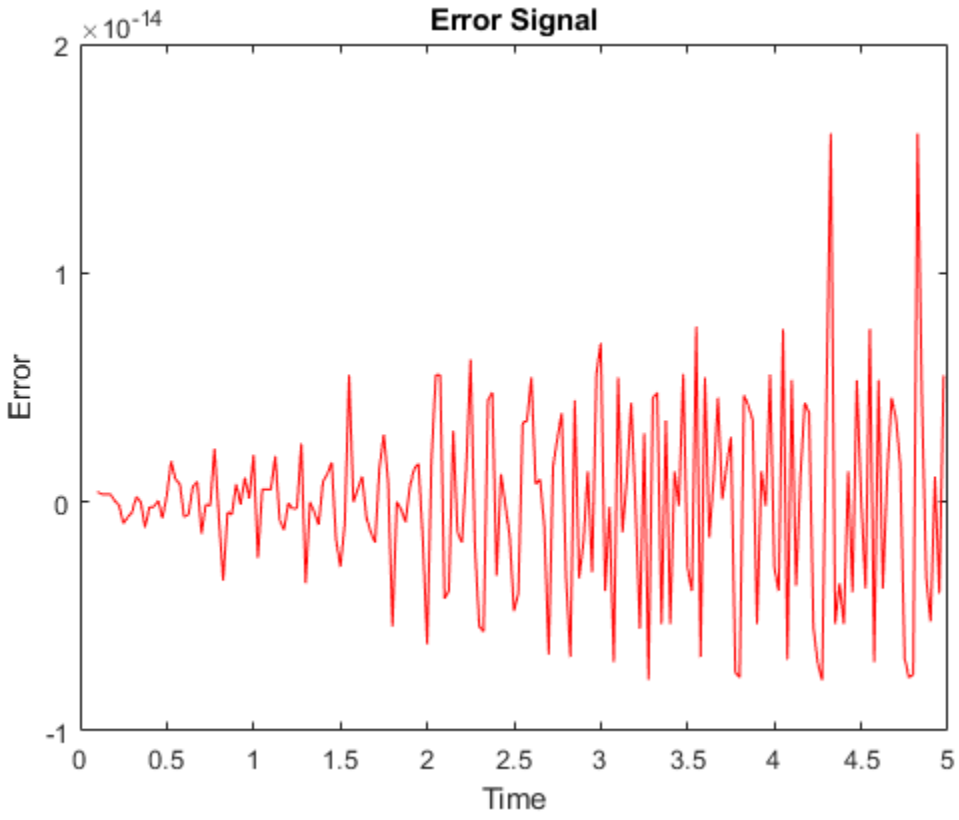
The output signal is plotted with the targets.

```
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 4-9.

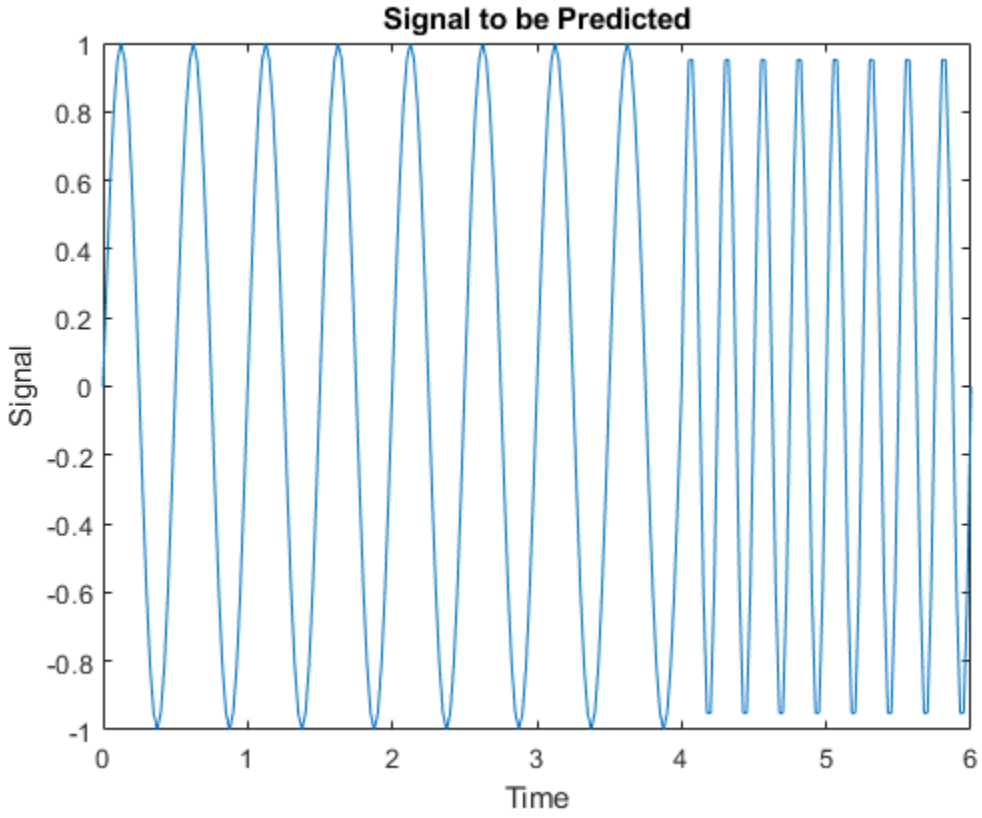
Defining a Wave Form

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);
```

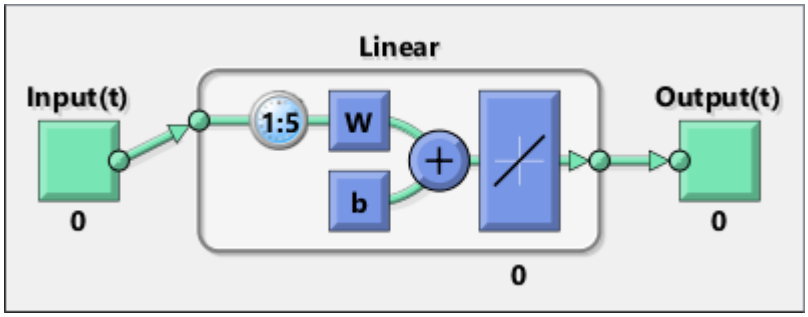
The targets are now defined to match the inputs. The network is to predict the current input, only using the last five values.

```
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)
```

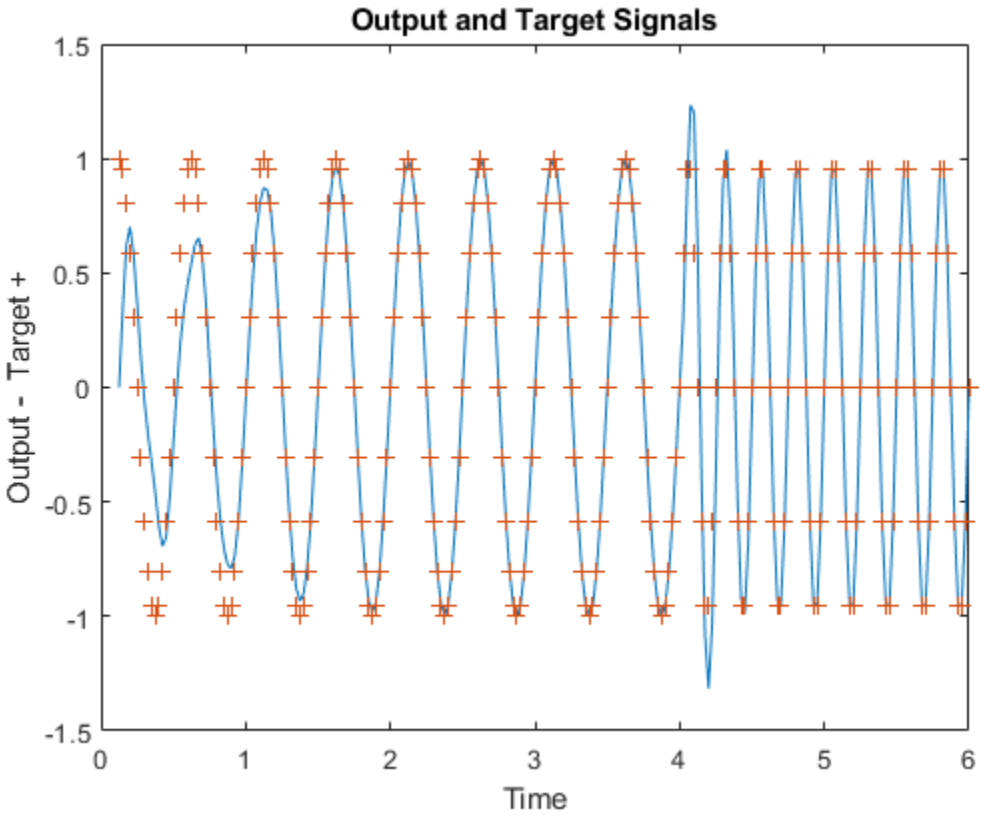


Adapting the Linear Layer

```
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);
```

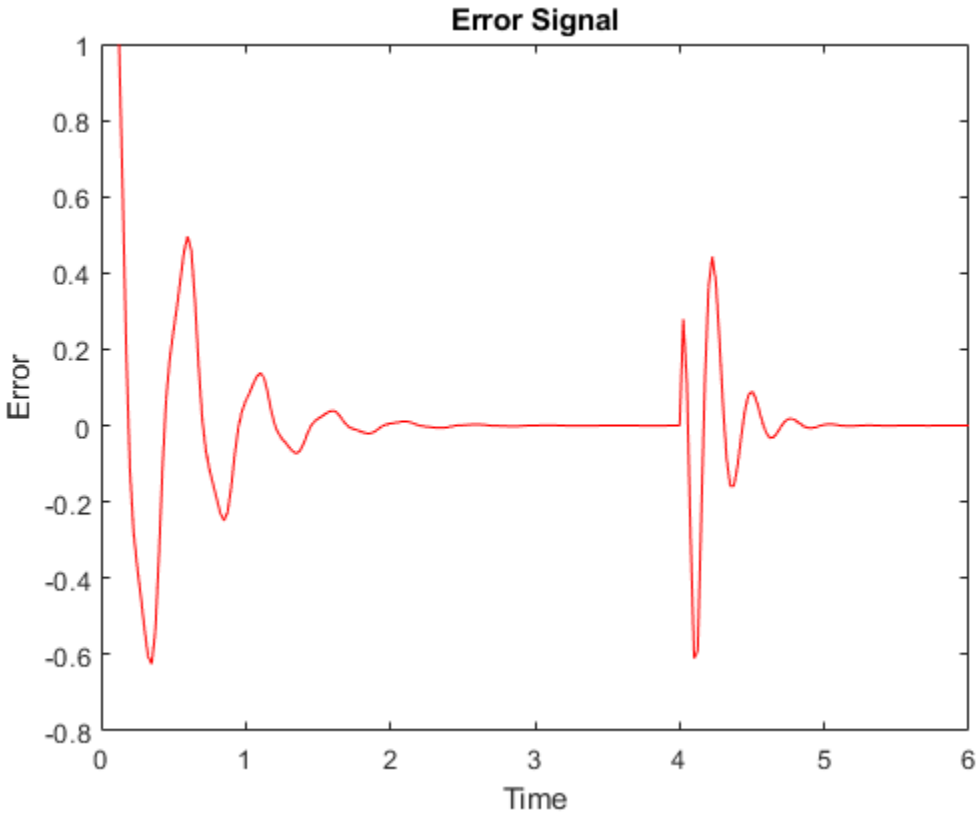
The output signal is plotted with the targets.

```
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.

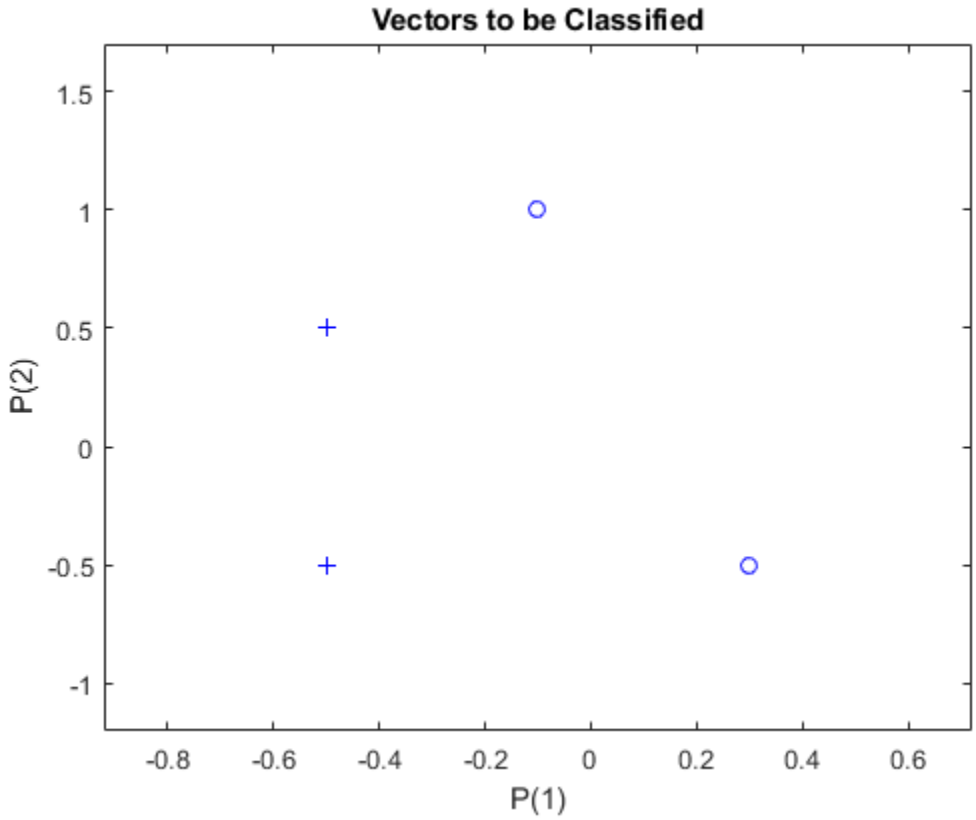
This example illustrated how to simulate an adaptive linear network which can predict a signal's next value from current and past values despite changes in the signals behavior.

Classification with a 2-Input Perceptron

A 2-input hard limit neuron is trained to classify 5 input vectors into two categories.

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. 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 5 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).

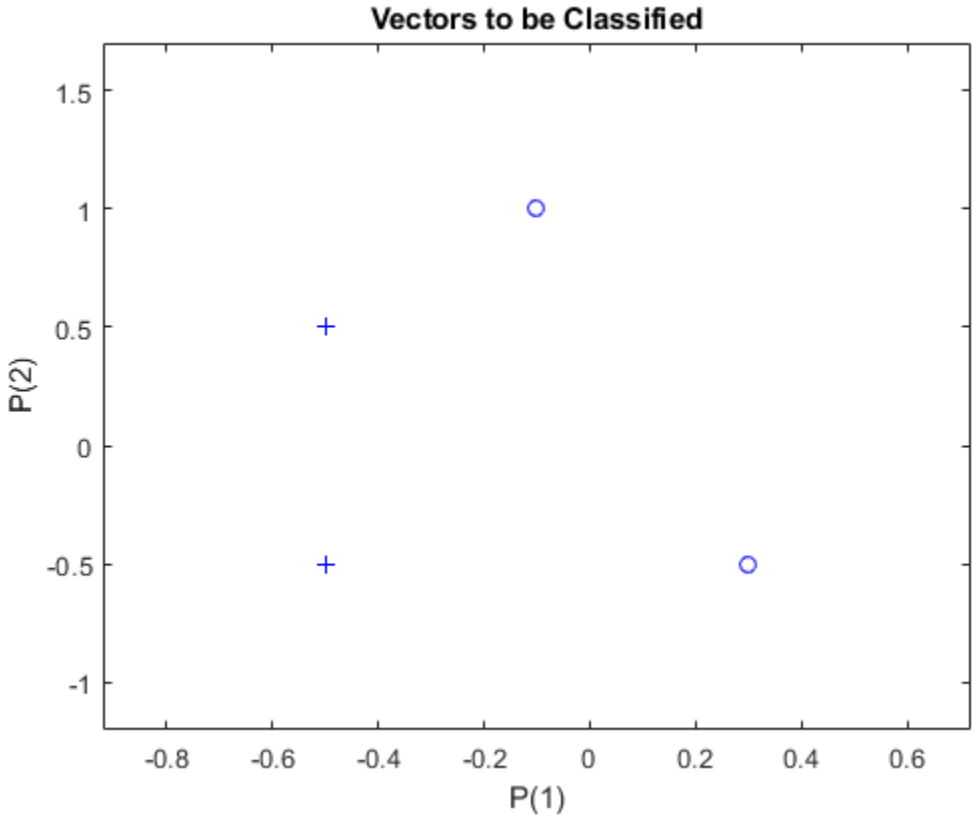
Here PERCEPTRON creates a new neural network with a single neuron. The network is then configured to the data, so we can examine its initial weight and bias values. (Normally the configuration step can be skipped as it is automatically done by ADAPT or TRAIN.)

```
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!

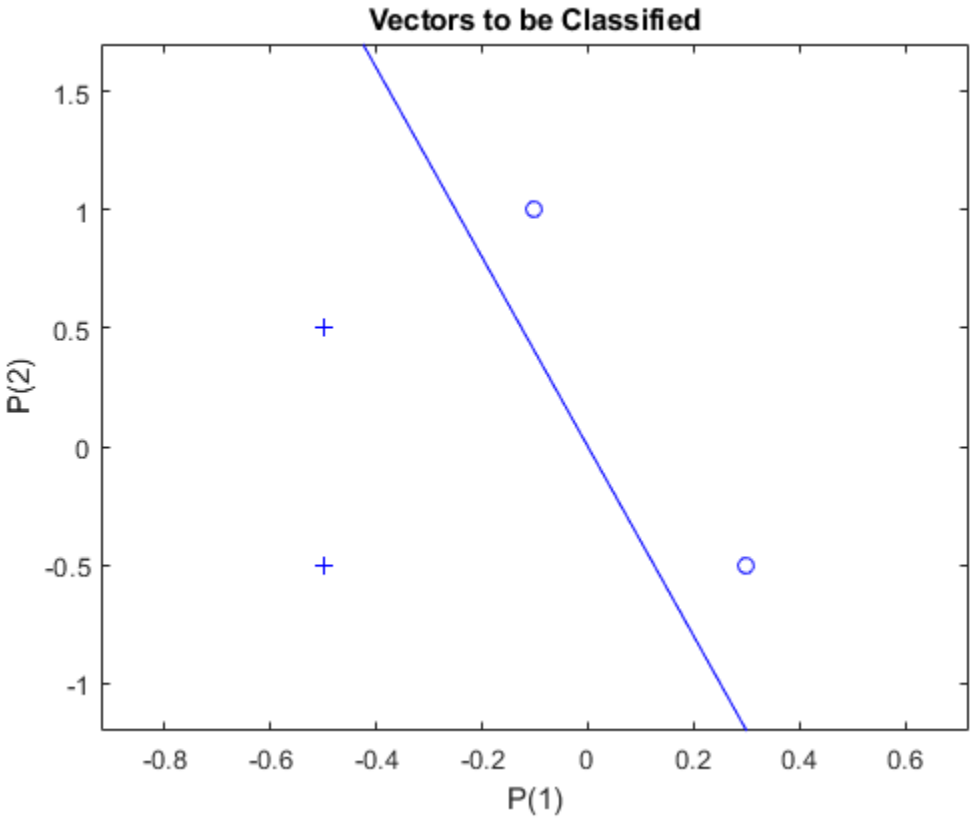
```
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
```



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.

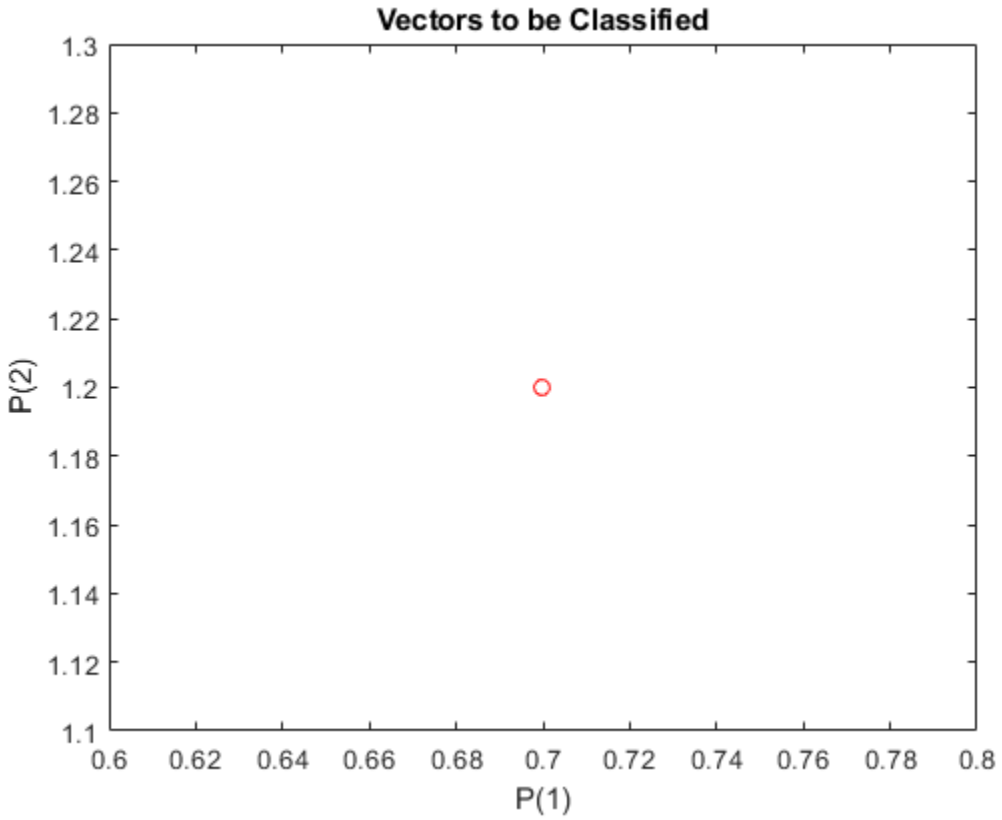
ADAPT updates the network for each timestep in the series and returns a new network object that performs as a better classifier.

```
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.

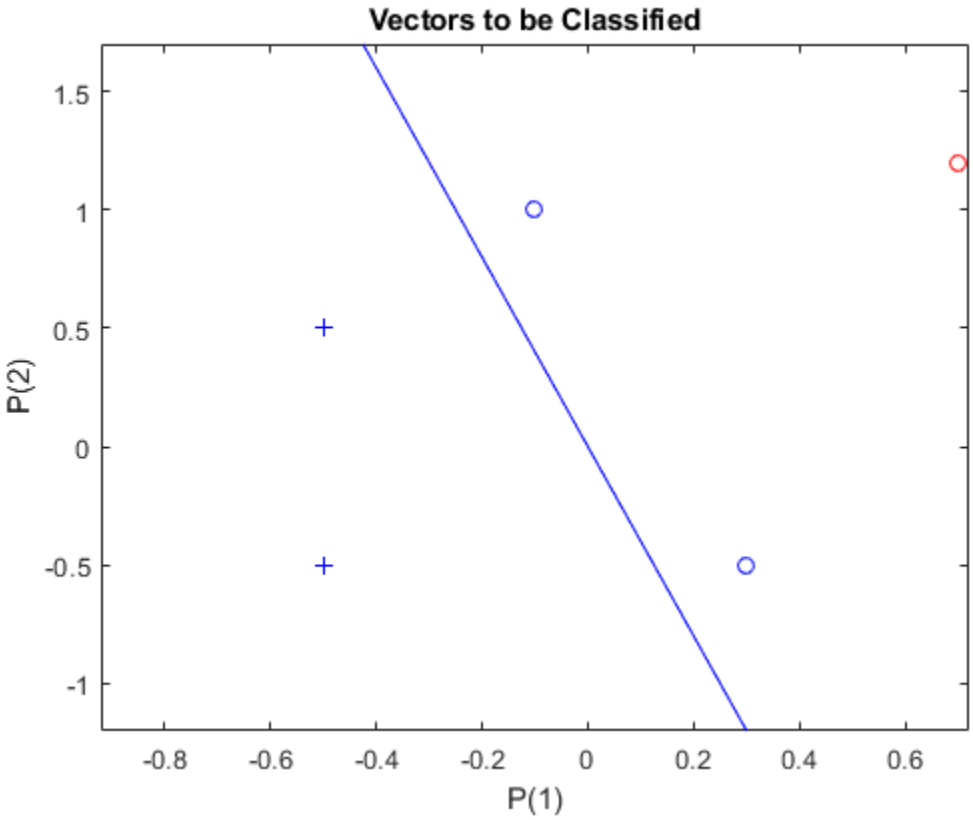
```
x = [0.7; 1.2];
y = net(x);
plotpv(x,y);
point = findobj(gca,'type','line');
point.Color = 'red';
```



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;
```

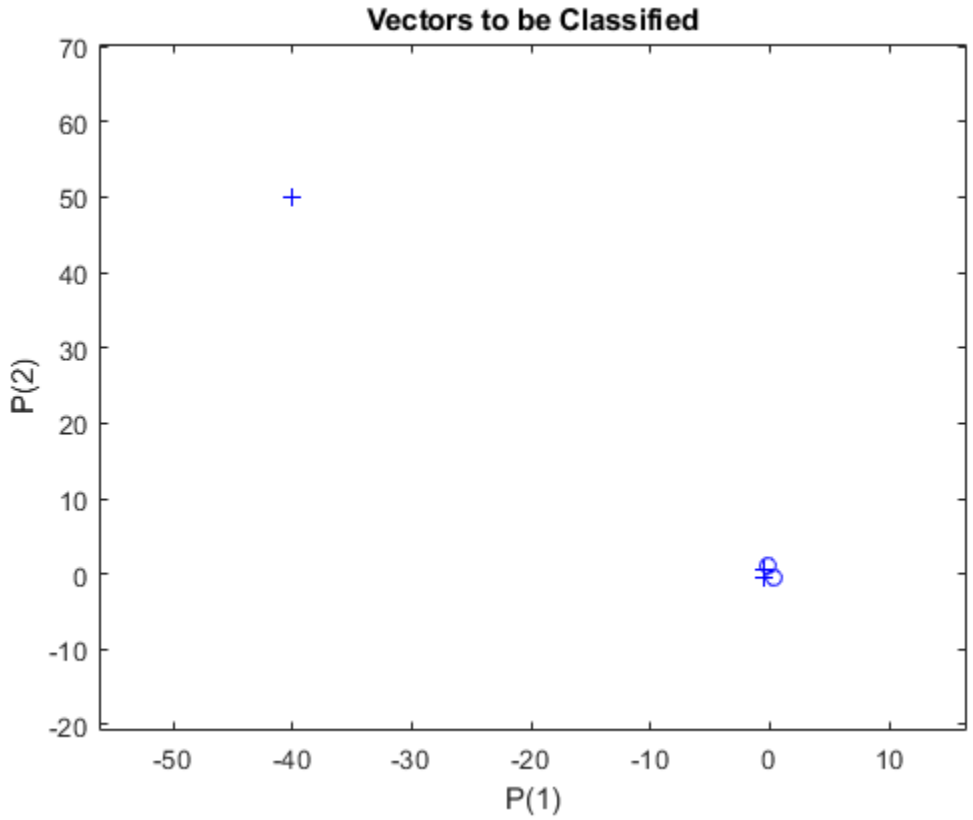

Vectors to be Classified
1.5
1
0.5
0
-0.5
-1
P(2)
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
P(1)

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.

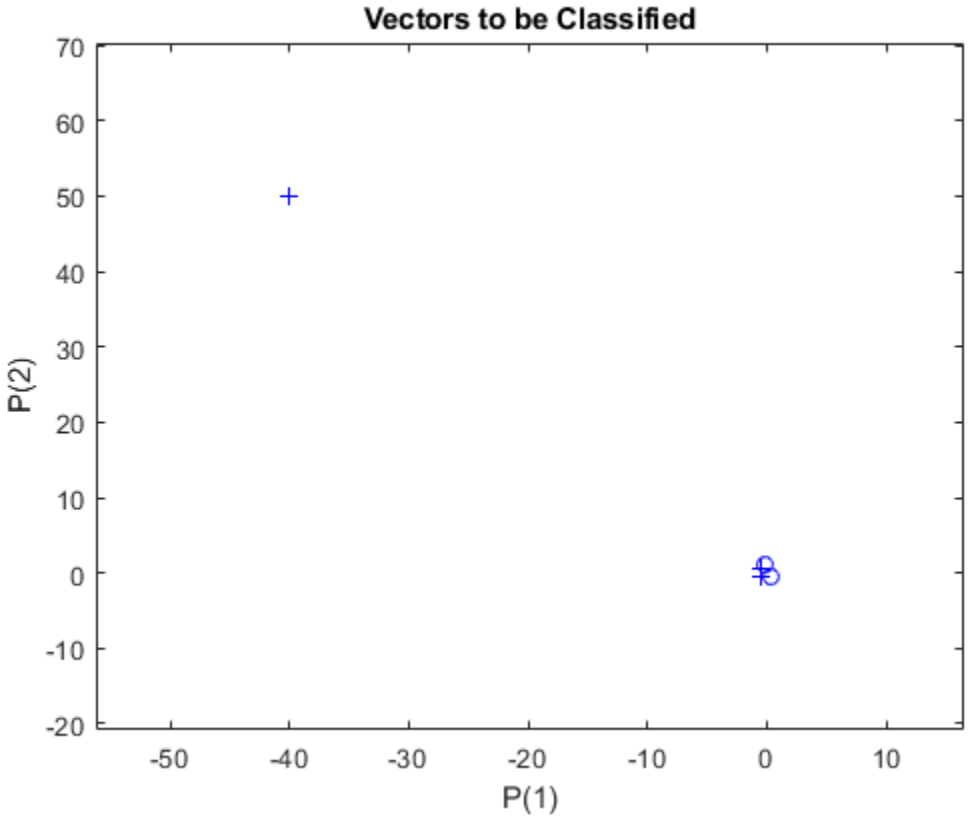
PERCEPTRON creates a new network which is then configured with the input and target data which results in initial values for its weights and bias. (Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

```
net = perceptron;
net = configure(net,X,T);
```

Add the neuron's initial attempt at classification to the plot.

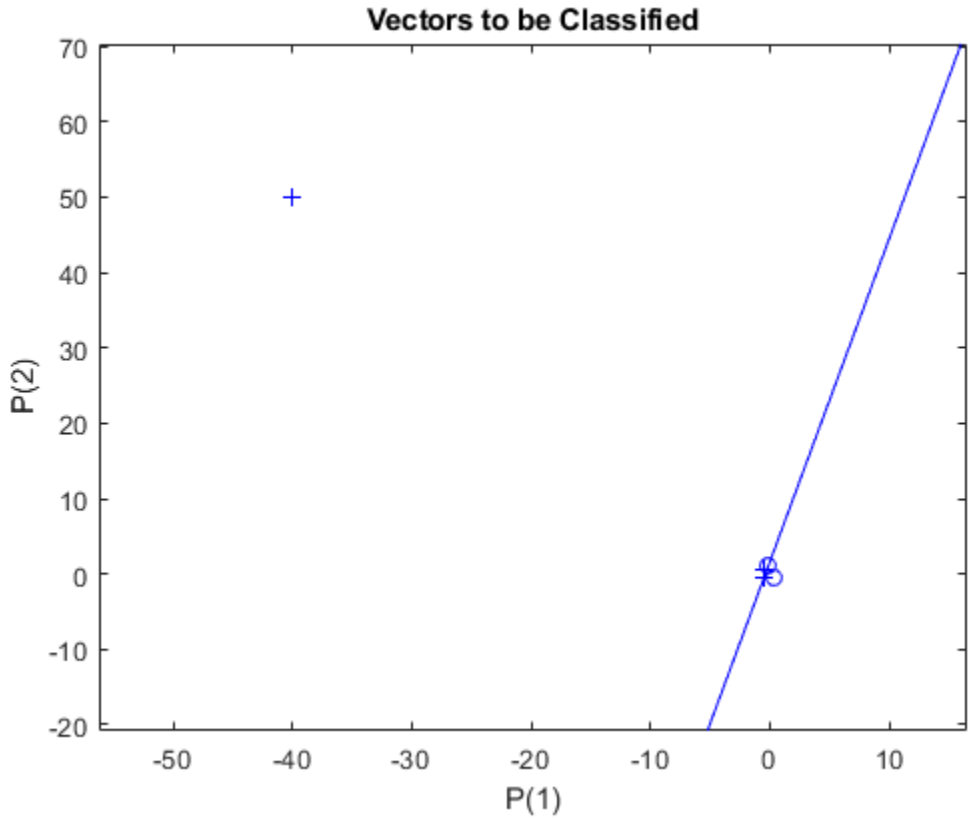
The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
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
```

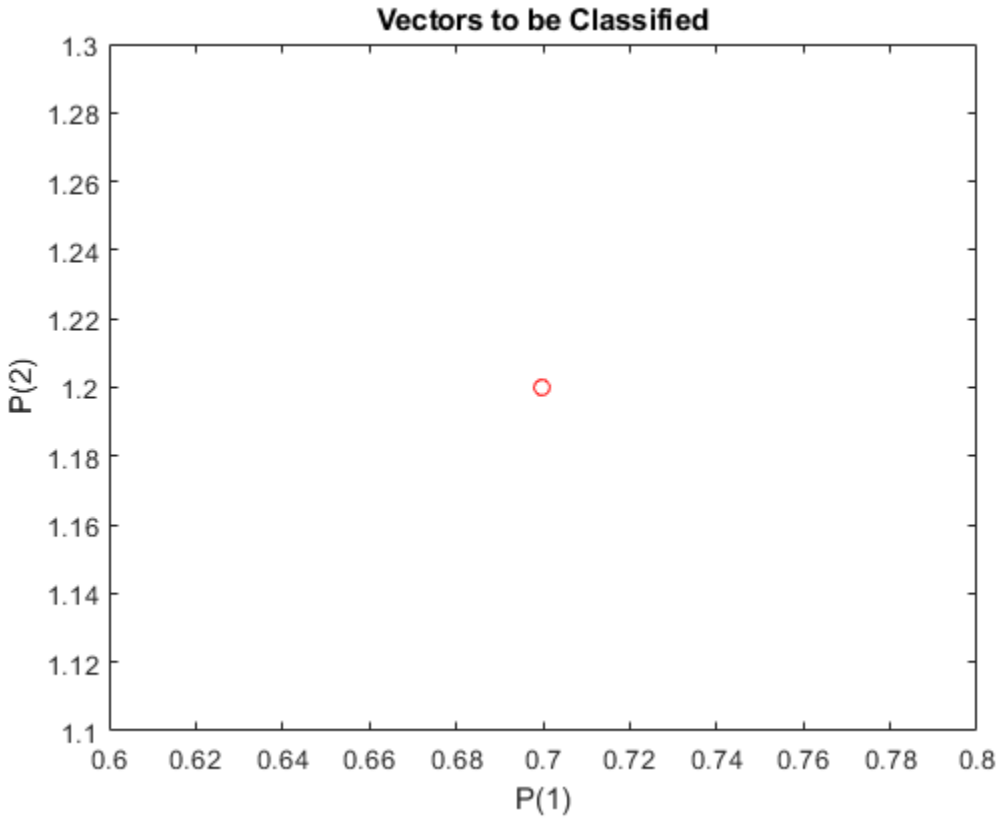


Note that it took the perceptron three passes to get it right. This a long time for such a simple problem. The reason for the long training time is the outlier vector. Despite the long training time, the perceptron still learns properly and can be used to classify other inputs.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

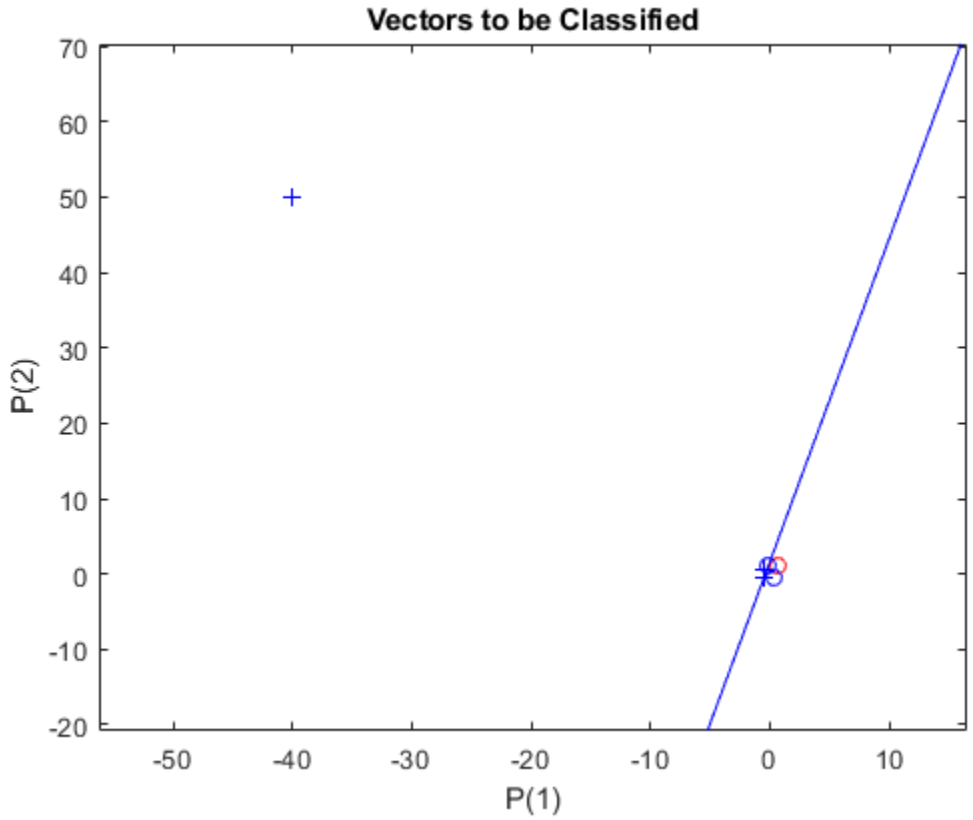
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
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.

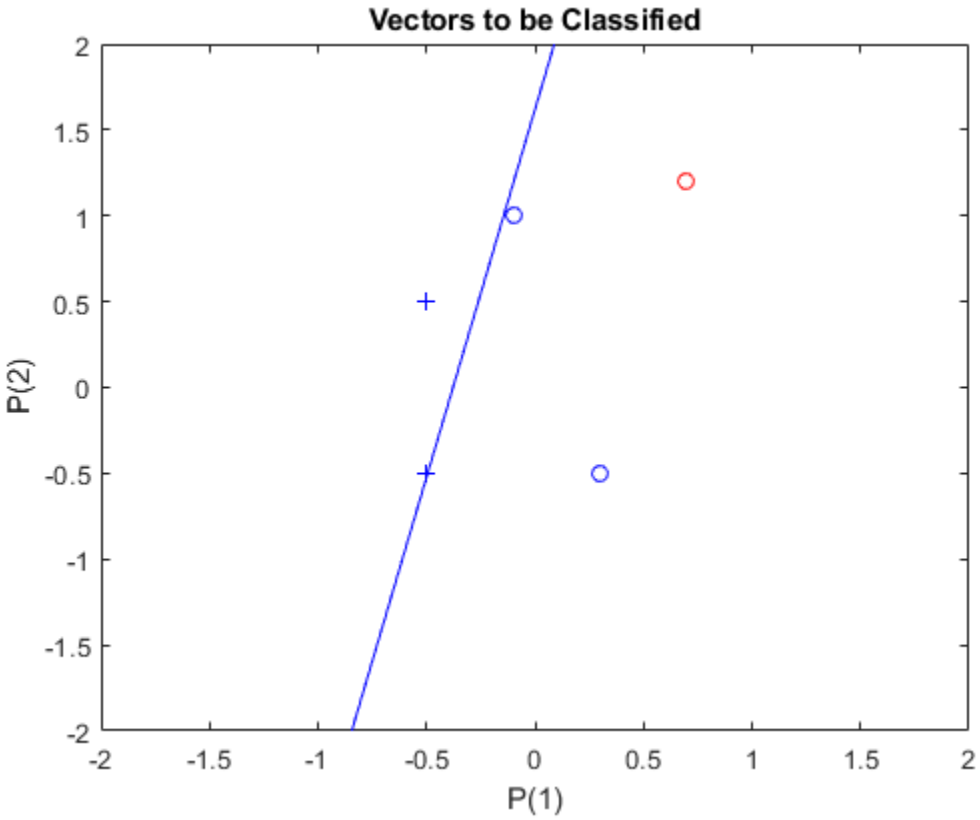
```
hold on;
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
hold off;
```



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]);
```

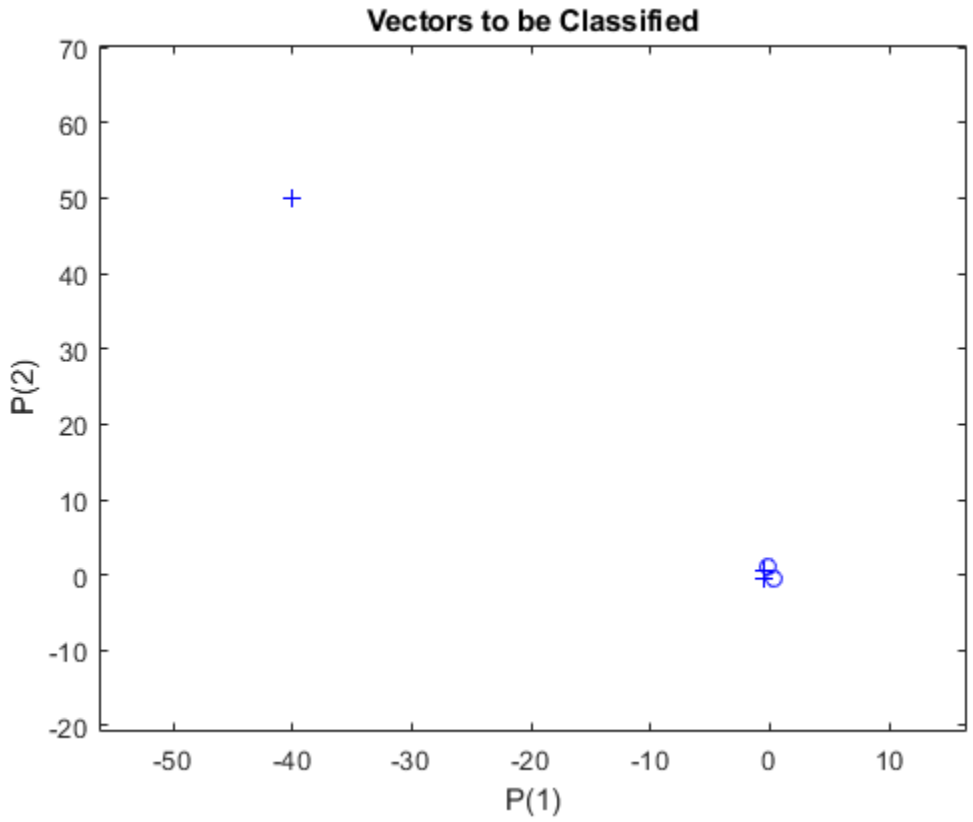

Vectors to be Classified
P(2)
P(1)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

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 sensative to large variations in input vector size than LEARNP (the default).

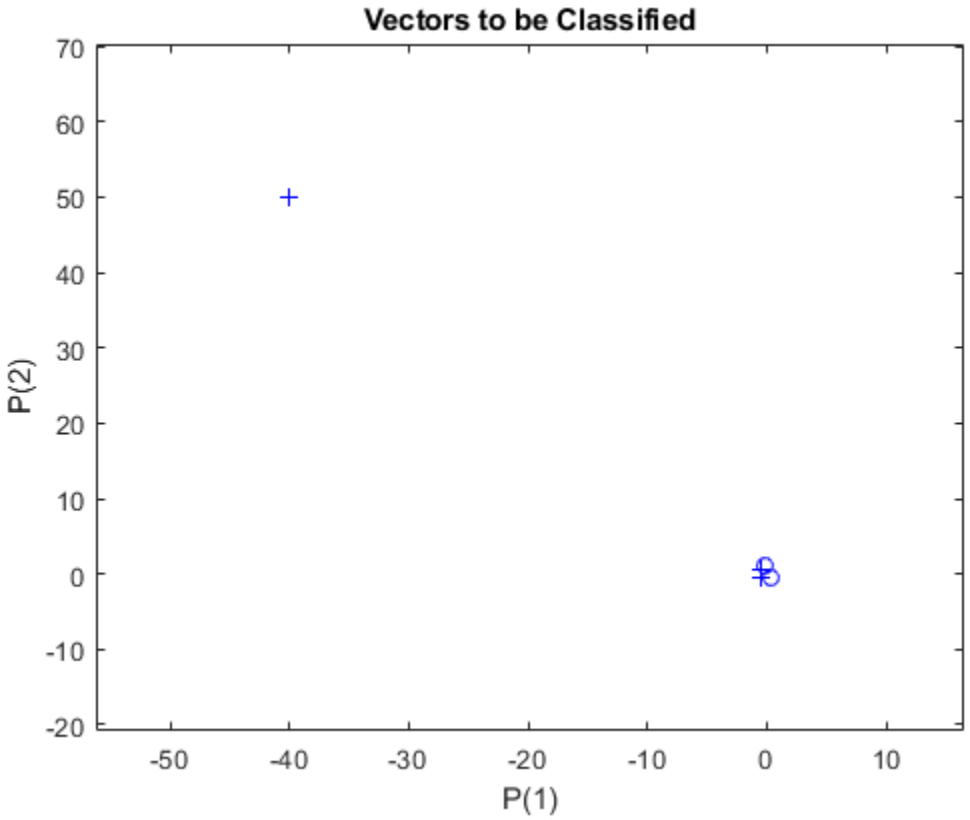
The network is then configured with the input and target data which results in initial values for its weights and bias. (Configuration is normally not necessary, as it is done automatically by ADAPT and TRAIN.)

```
net = perceptron('hardlim','learnpn');
net = configure(net,X,T);
```

Add the neuron's initial attempt at classification to the plot.

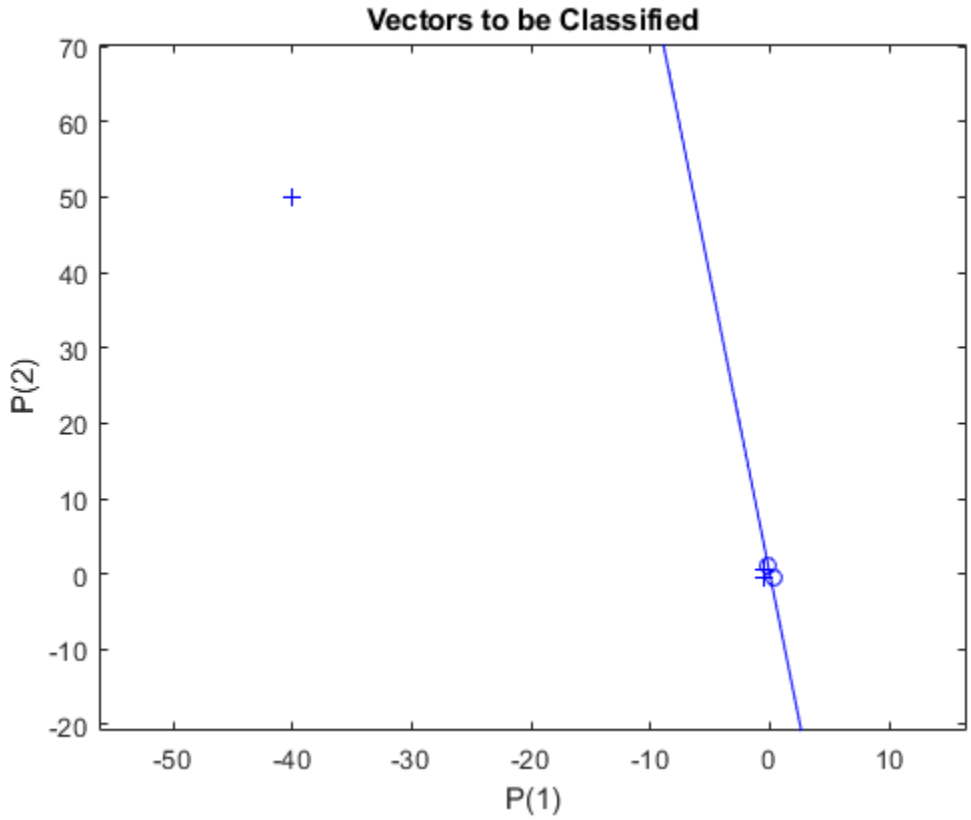
The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot. Fear not... we are going to train it!

```
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
```

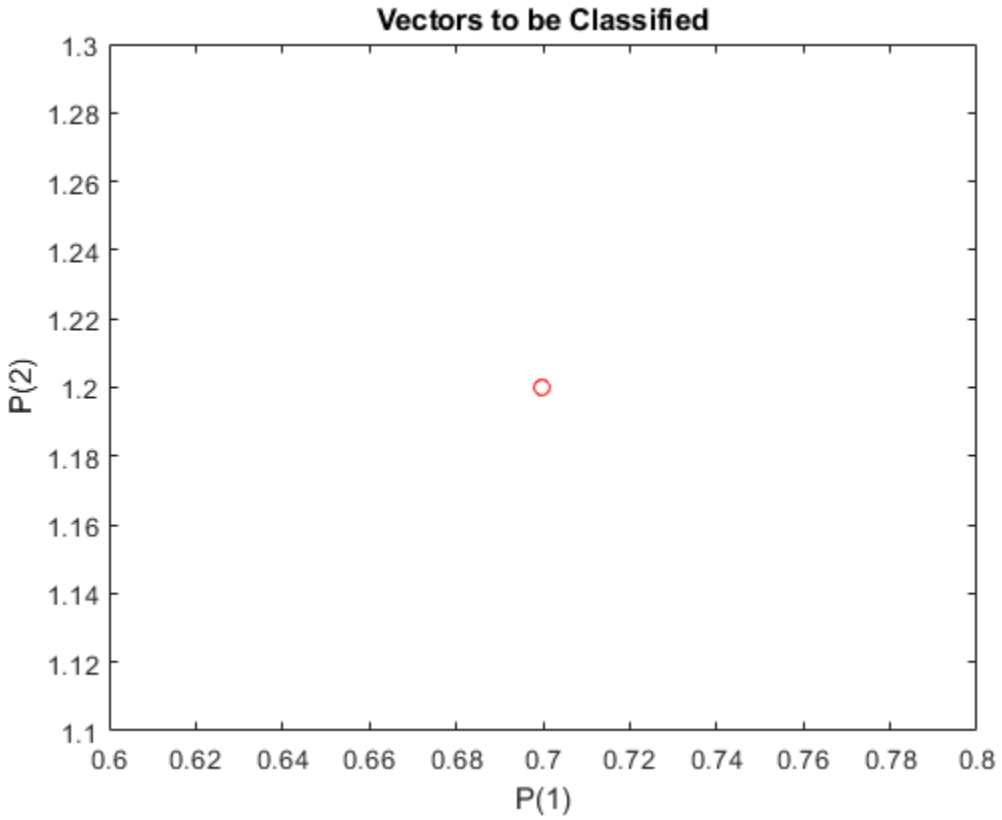


Note that training with LEARNP took only 3 epochs, while solving the same problem with LEARNPN required 32 epochs. Thus, LEARNPN does much better job than LEARNP when there are large variations in input vector size.

Now SIM can be used to classify any other input vector. For example, classify an input vector of [0.7; 1.2].

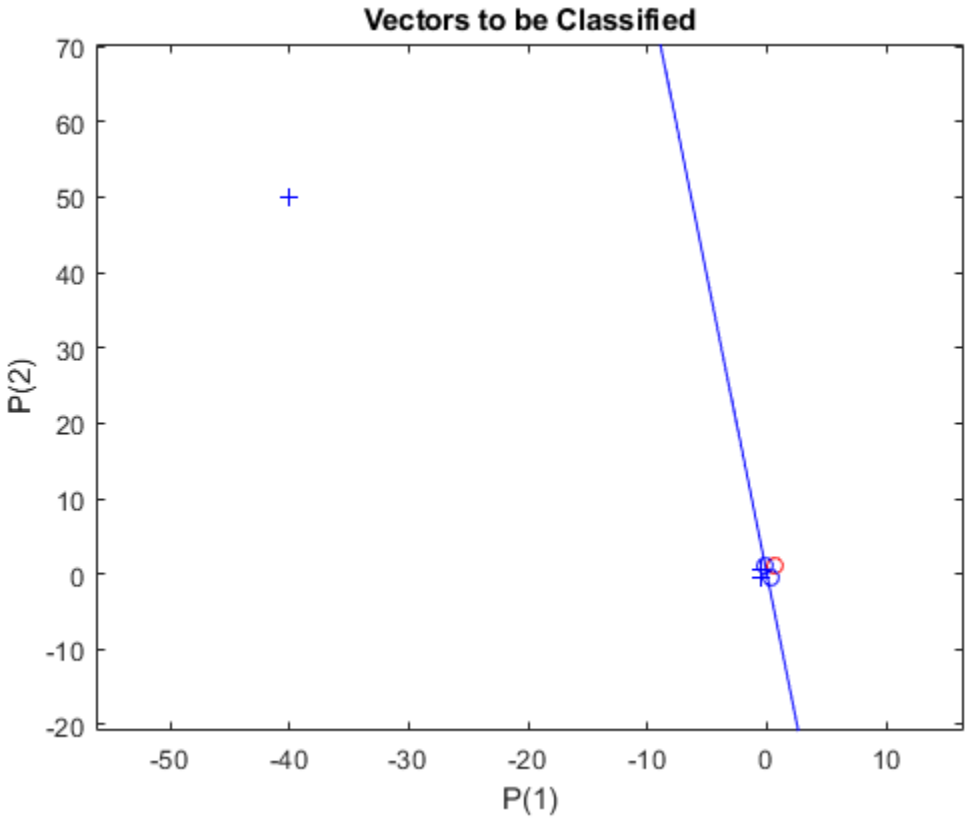
A plot of this new point with the original training set shows how the network performs. To distinguish it from the training set, color it red.

```
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.

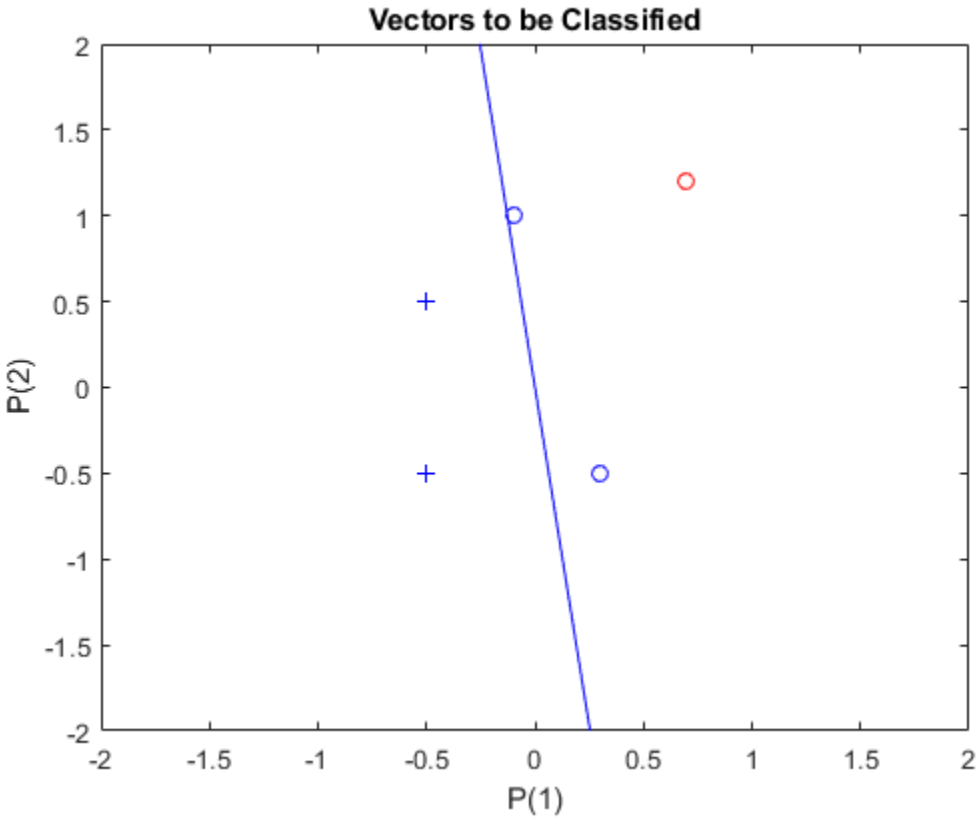
```
hold on;
plotpv(X,T);
plotpc(net.IW{1},net.b{1});
hold off;
```



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]);
```

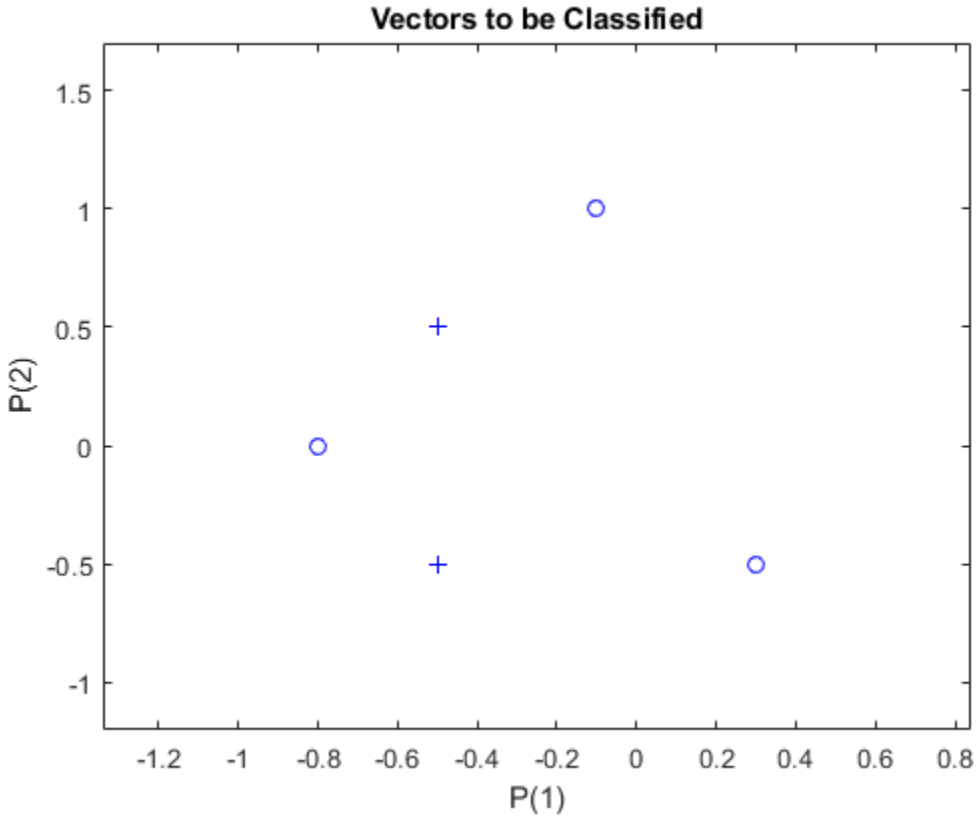

Vectors to be Classified
P(2)
P(1)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

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);
```



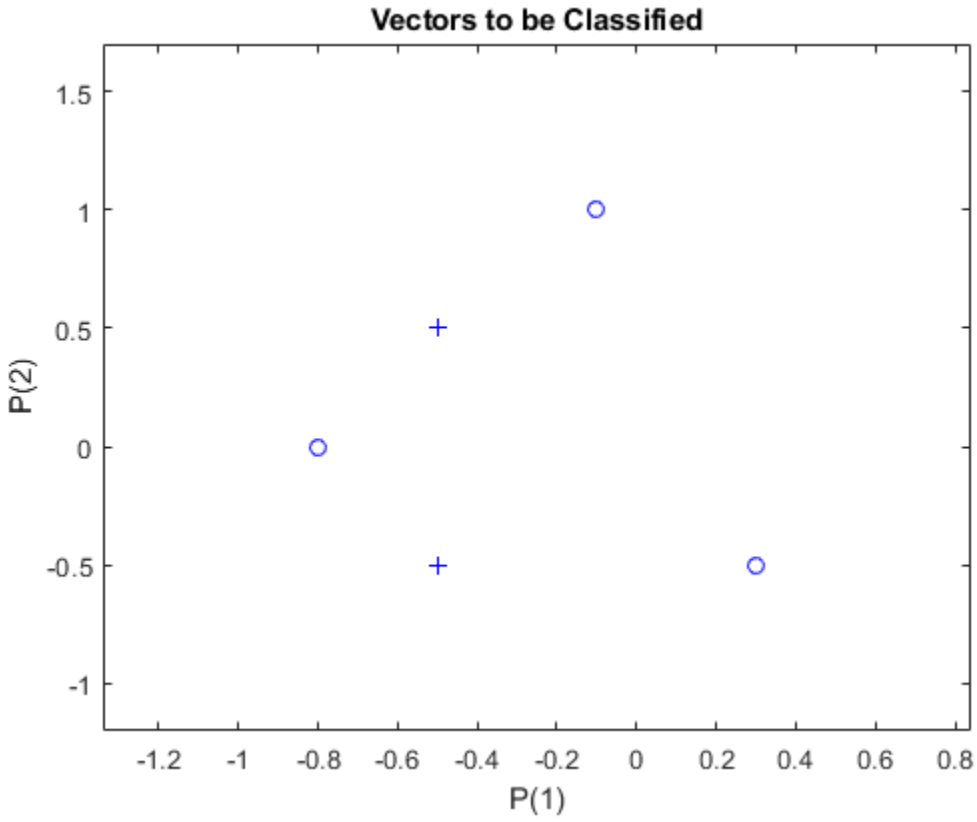
The perceptron must properly classify the input vectors in X into the categories defined by T. Because the two kinds of input vectors cannot be separated by a straight line, the perceptron will not be able to do it.

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);
```

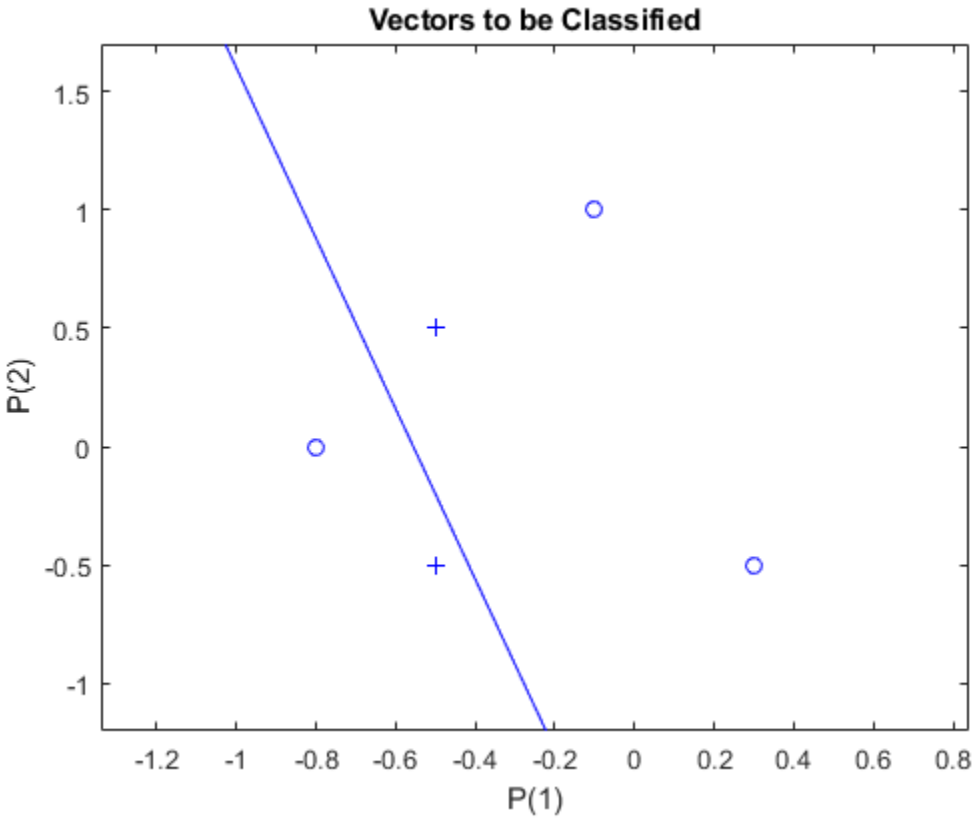
Add the neuron's initial attempt at classification to the plot. The initial weights are set to zero, so any input gives the same output and the classification line does not even appear on the plot.

```
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

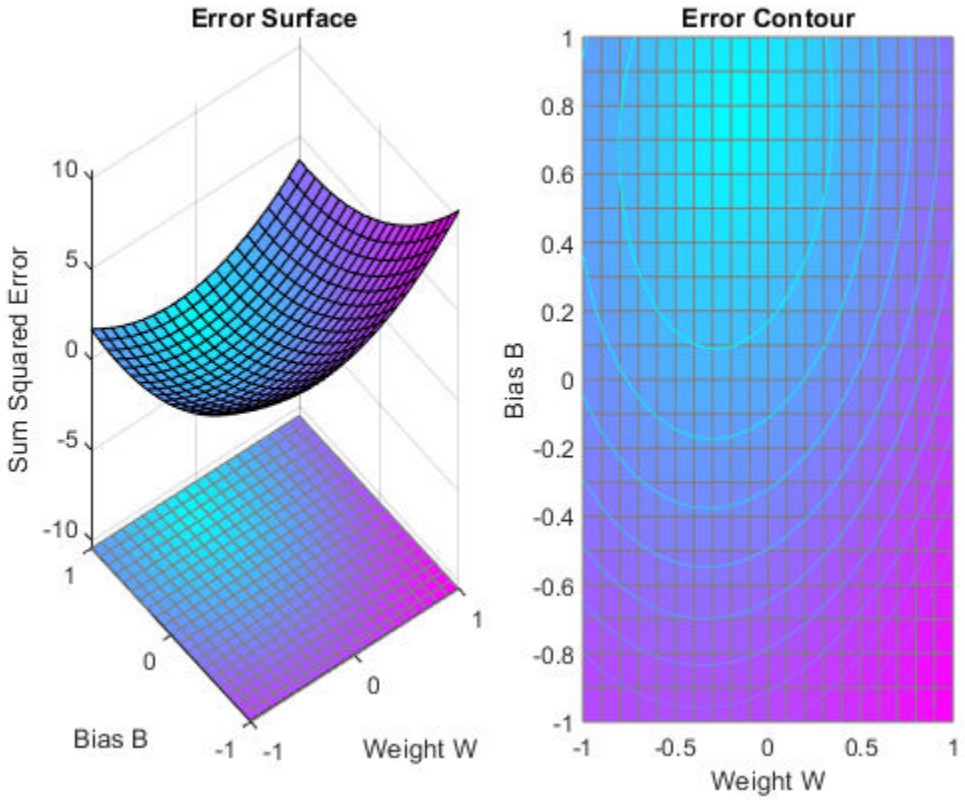
A linear neuron is designed to respond to specific inputs with target outputs.

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
```

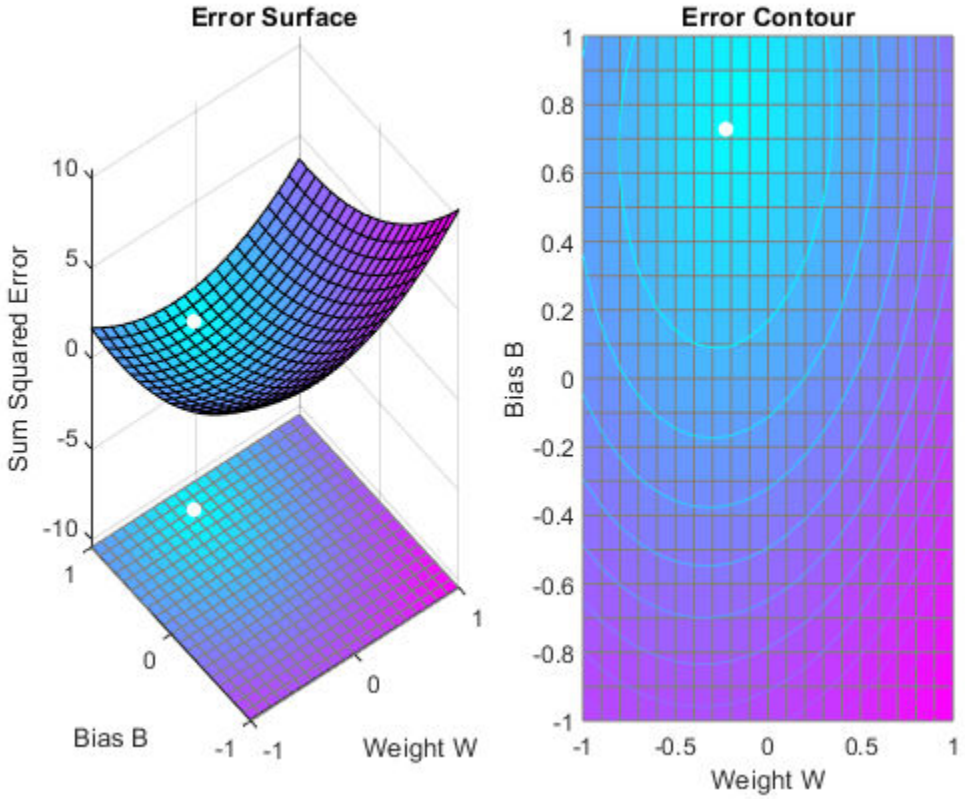
```
E = 1×2

     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

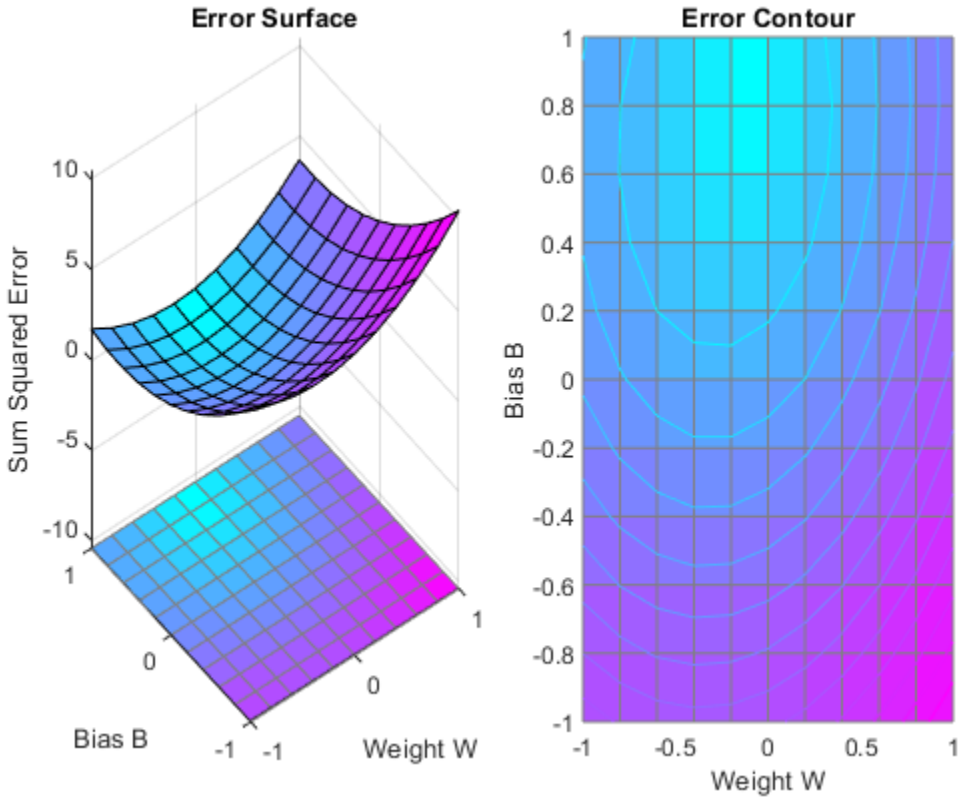
A linear neuron is trained to respond to specific inputs with target outputs.

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.

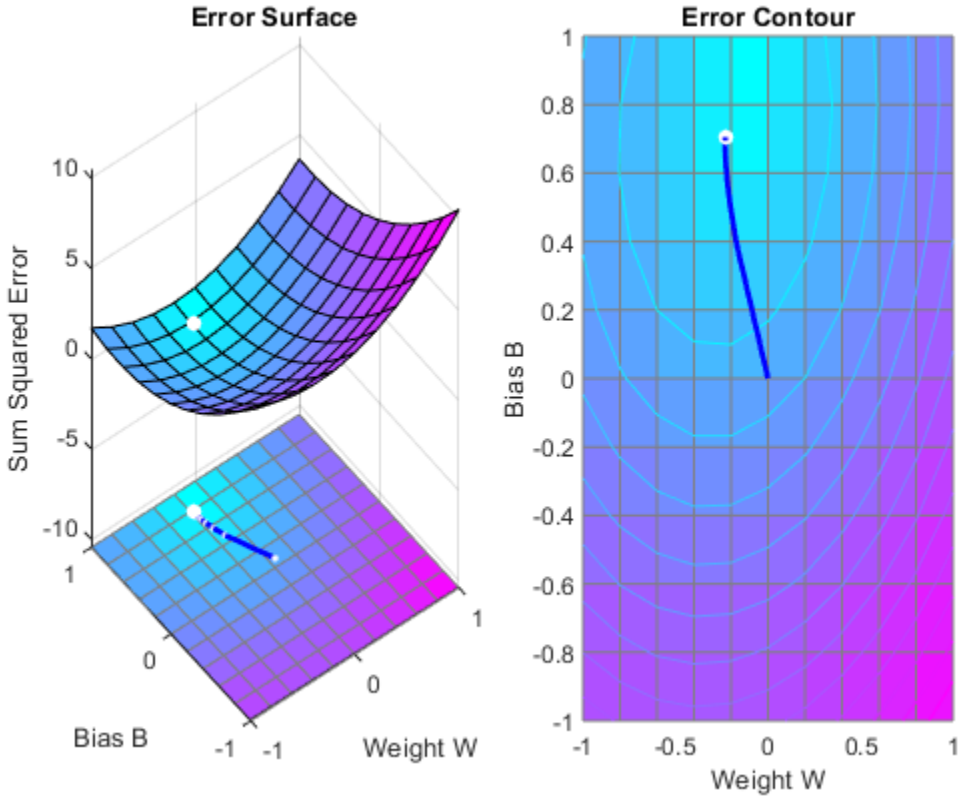
```
maxlr = 0.40*maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

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).

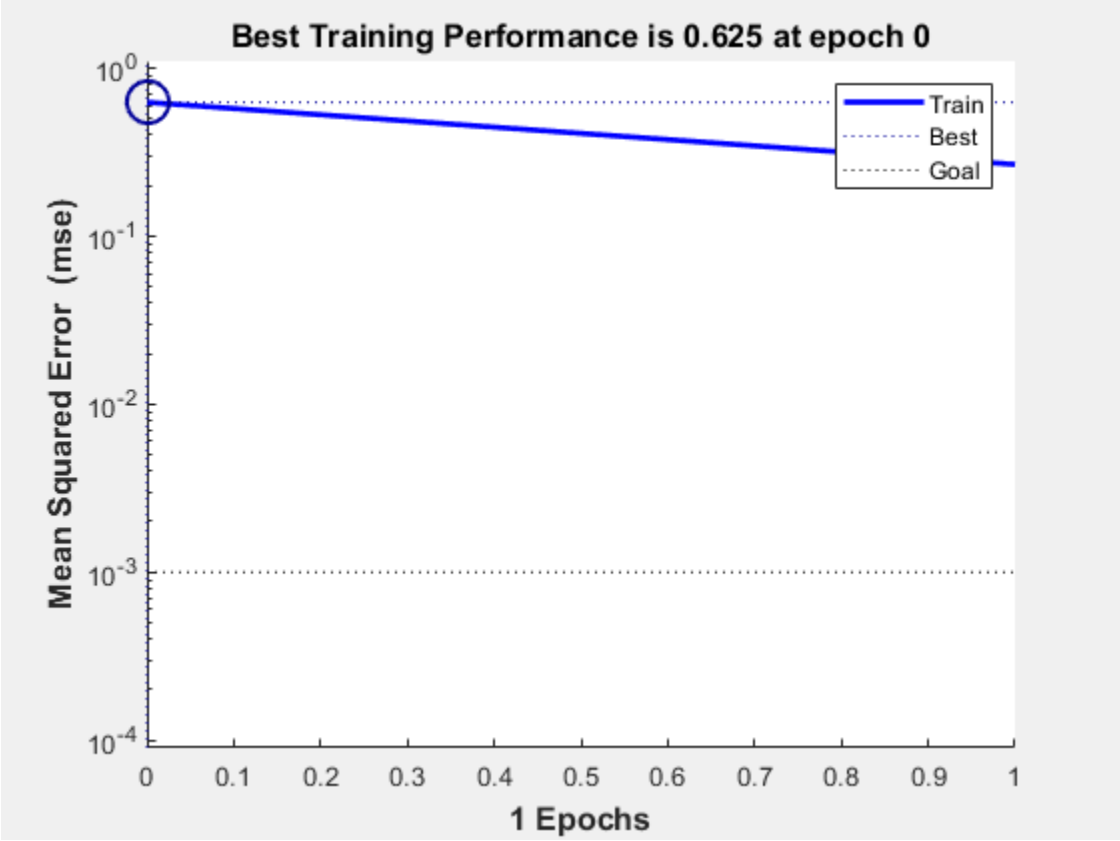
```
% [net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while true
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      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.

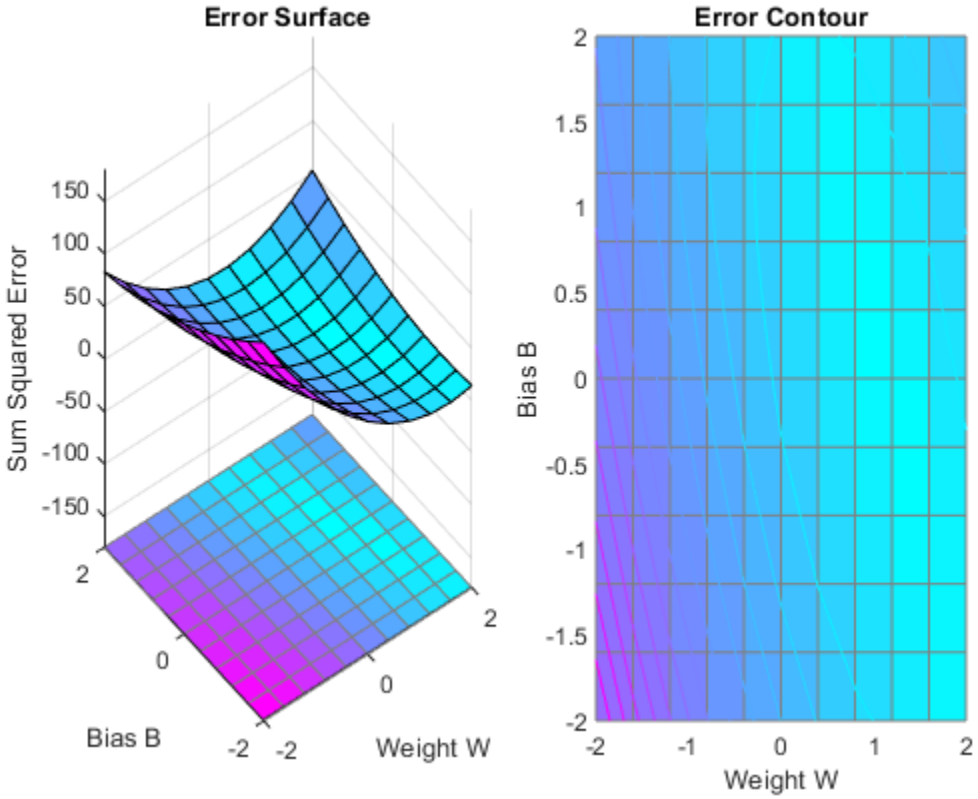
X defines four 1-element input patterns (column vectors). T defines associated 1-element targets (column vectors). Note that the relationship between values in X and in T is nonlinear. I.e. No W and B exist such that X*W+B = T for all of four sets of X and T values above.

```
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.

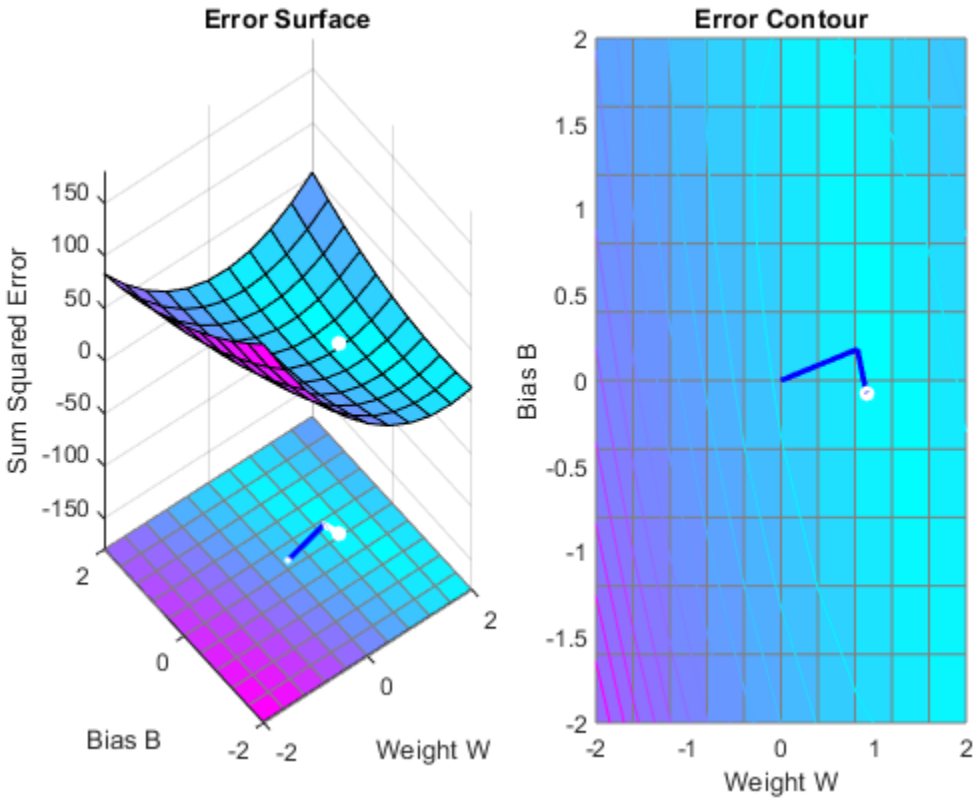
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

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
```

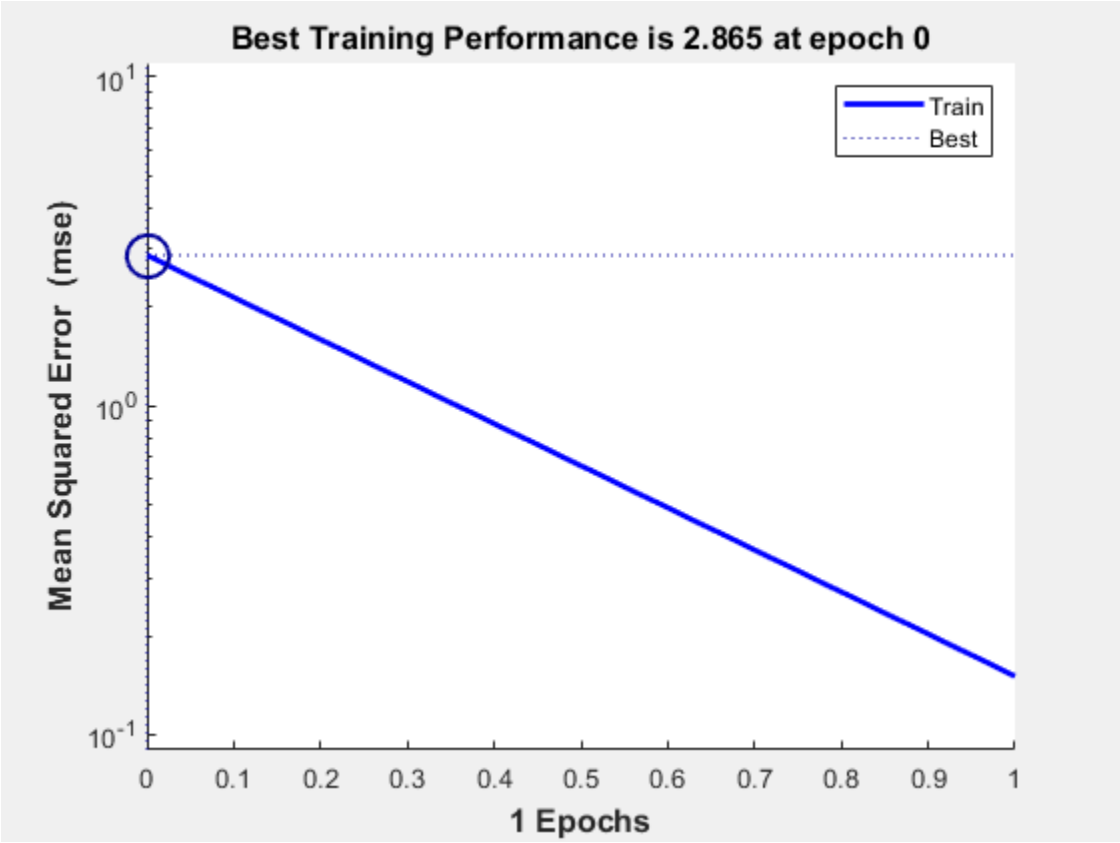



```
tr=r;
```

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

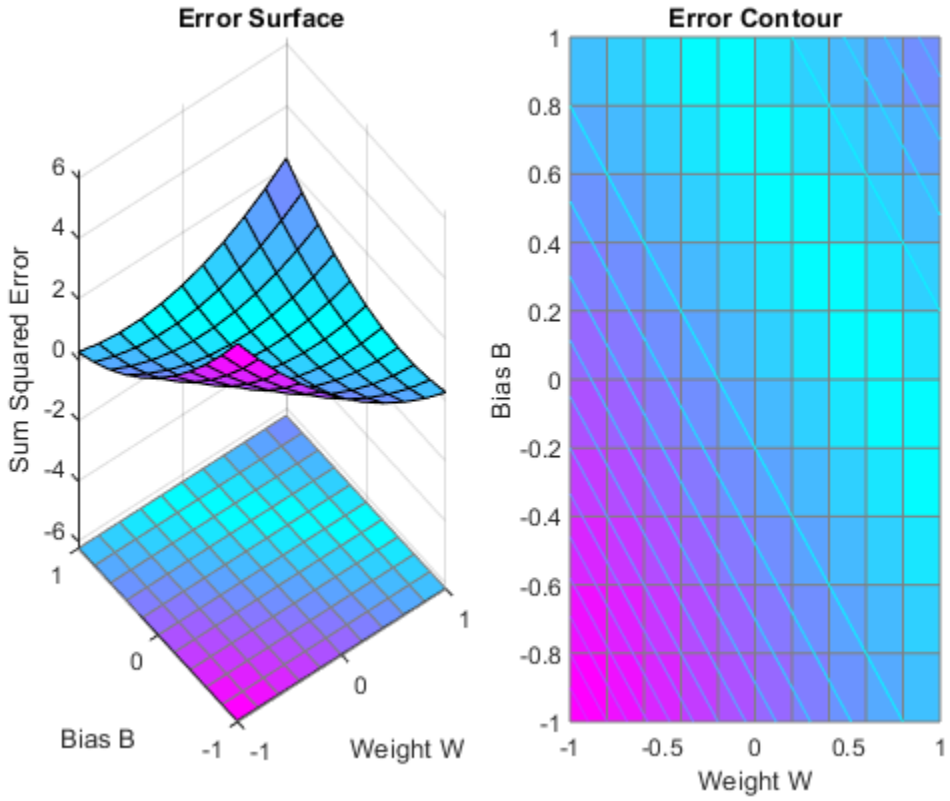
A linear neuron is trained to find y non-unique solution to an undetermined problem.

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.

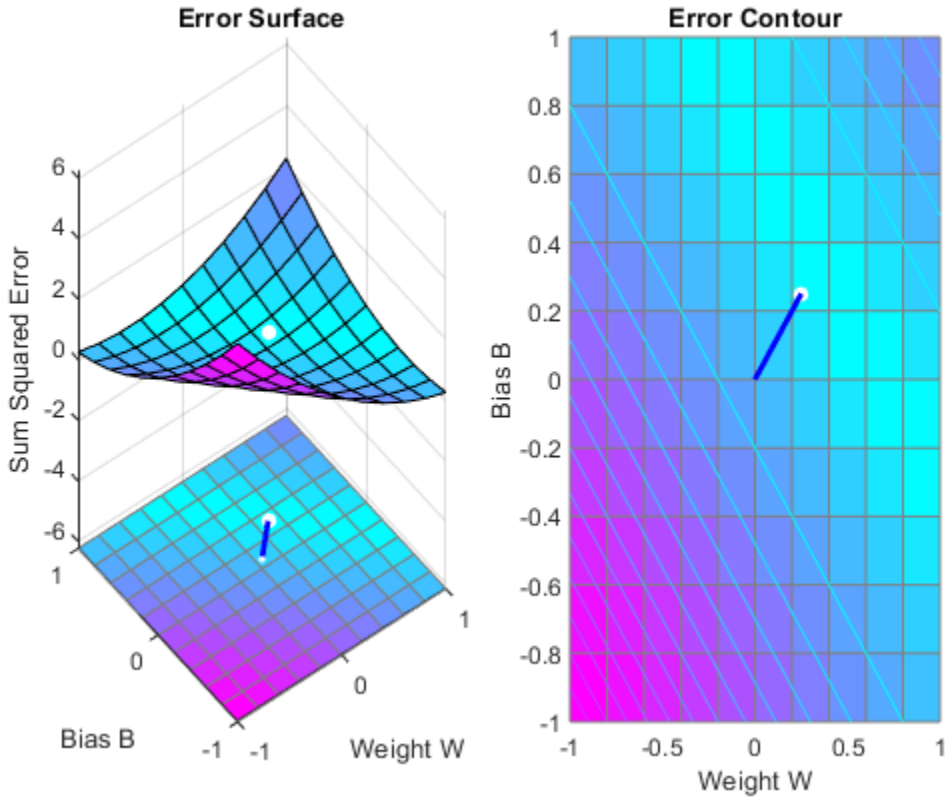
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr);
```

Override the default training parameters by setting the performance goal.

```
net.trainParam.goal = 1e-10;
```

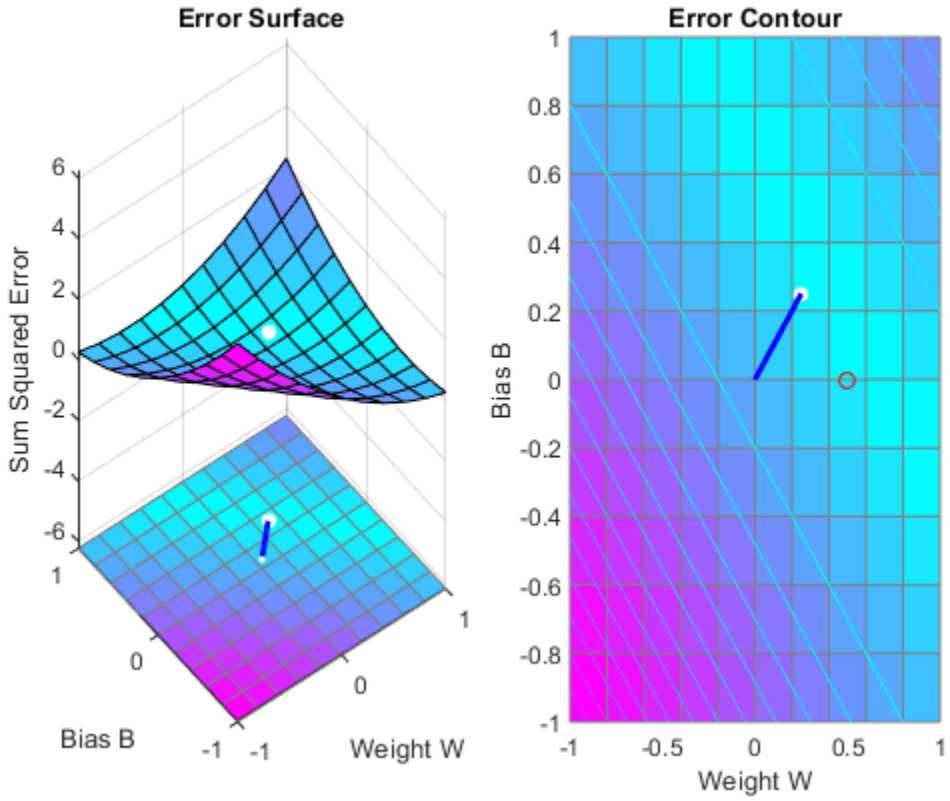
To show the path of the training we will train only one epoch at y time and call PLOTEP every epoch. The plot shows y history of the training. Each dot represents an epoch and the blue lines show each change made by the learning rule (Widrow-Hoff by default).

```
% [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;
```



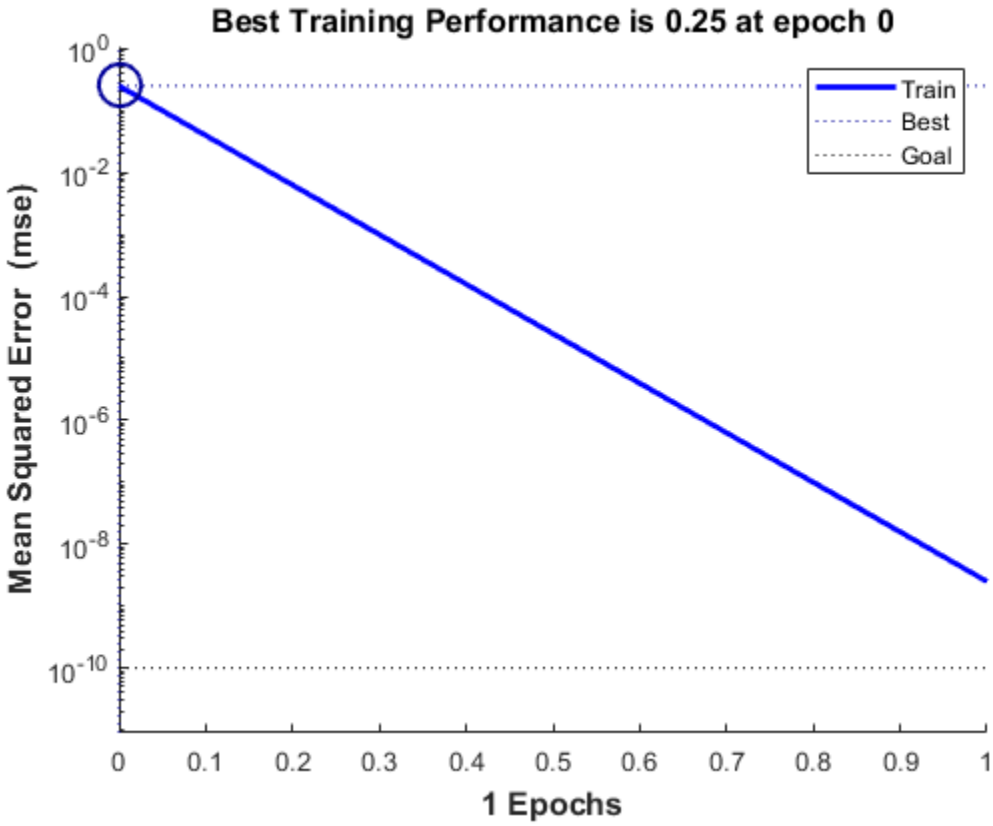
Here we plot the NEWLIND solution. Note that the TRAIN (white dot) and SOLVELIN (red circle) solutions are not the same. In fact, TRAINWH will return y different solution for different initial conditions, while SOLVELIN will always return the same solution.

```
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

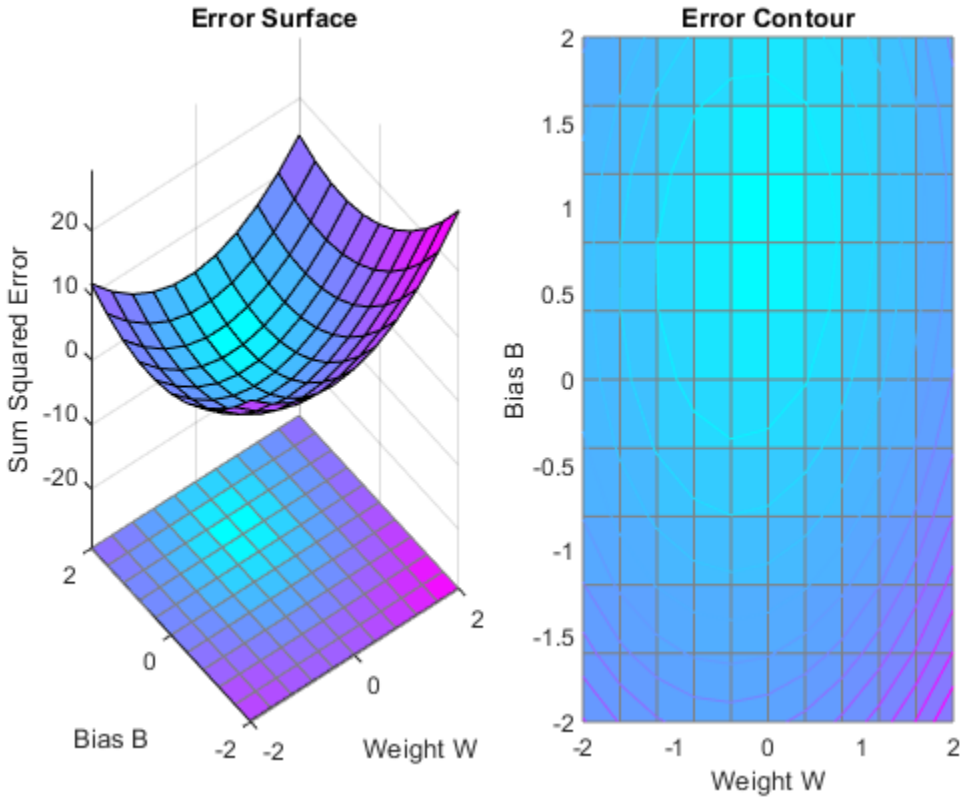
A linear neuron is trained to find the minimum error solution for a simple problem. The neuron is trained with the learning rate larger than the one suggested by MAXLINLR.

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.

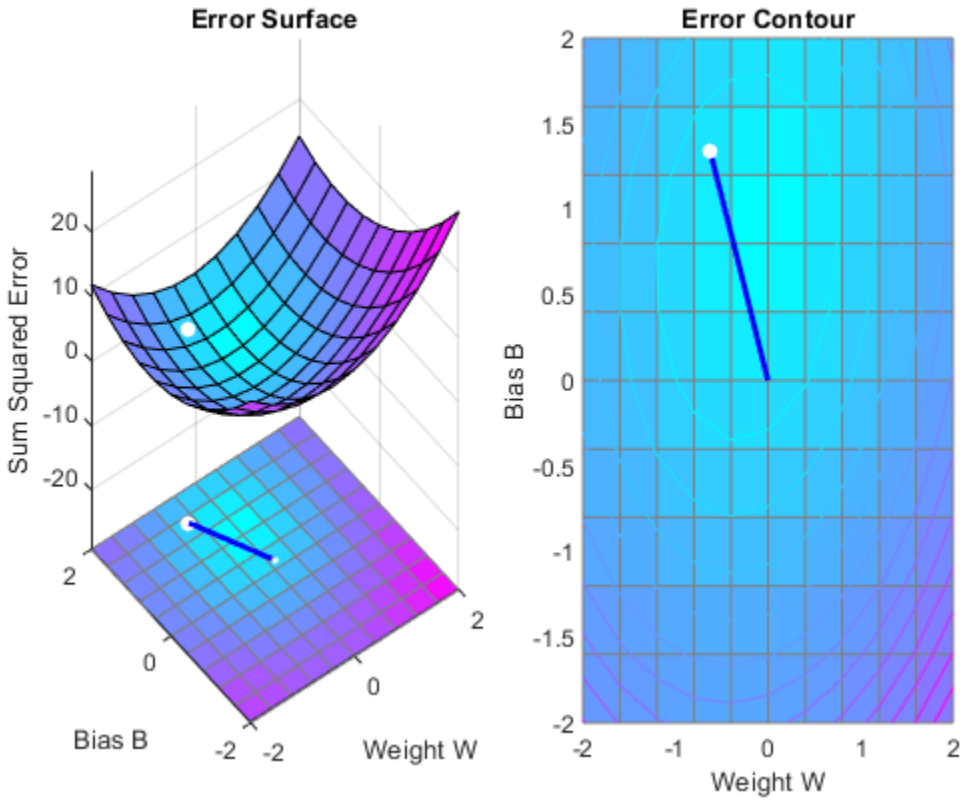
```
maxlr = maxlinlr(X,'bias');
net = newlin([-2 2],1,[0],maxlr*2.25);
```

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).

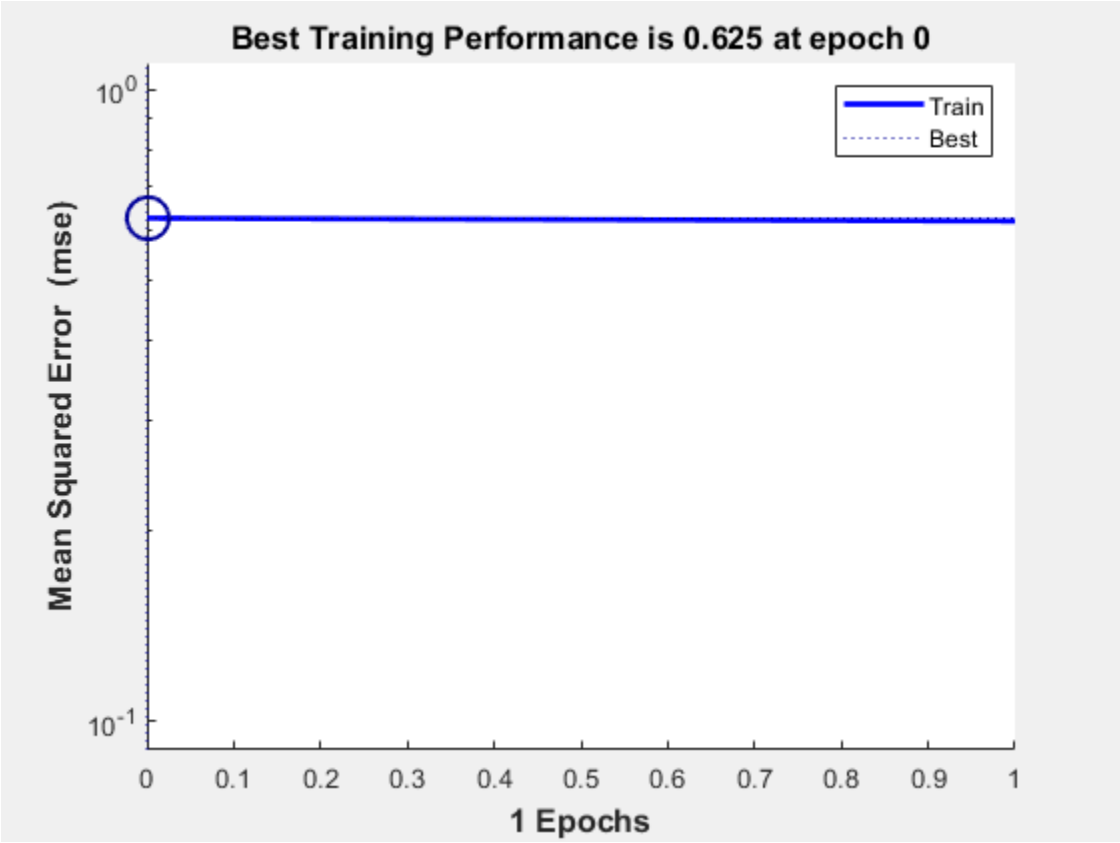
```
%[net,tr] = train(net,X,T);
net.trainParam.epochs = 1;
net.trainParam.show = NaN;
h=plotep(net.IW{1},net.b{1},mse(T-net(X)));
[net,tr] = train(net,X,T);
r = tr;
epoch = 1;
while epoch < 20
   epoch = epoch+1;
   [net,tr] = train(net,X,T);
   if length(tr.epoch) > 1
      h = plotep(net.IW{1,1},net.b{1},tr.perf(2),h);
      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

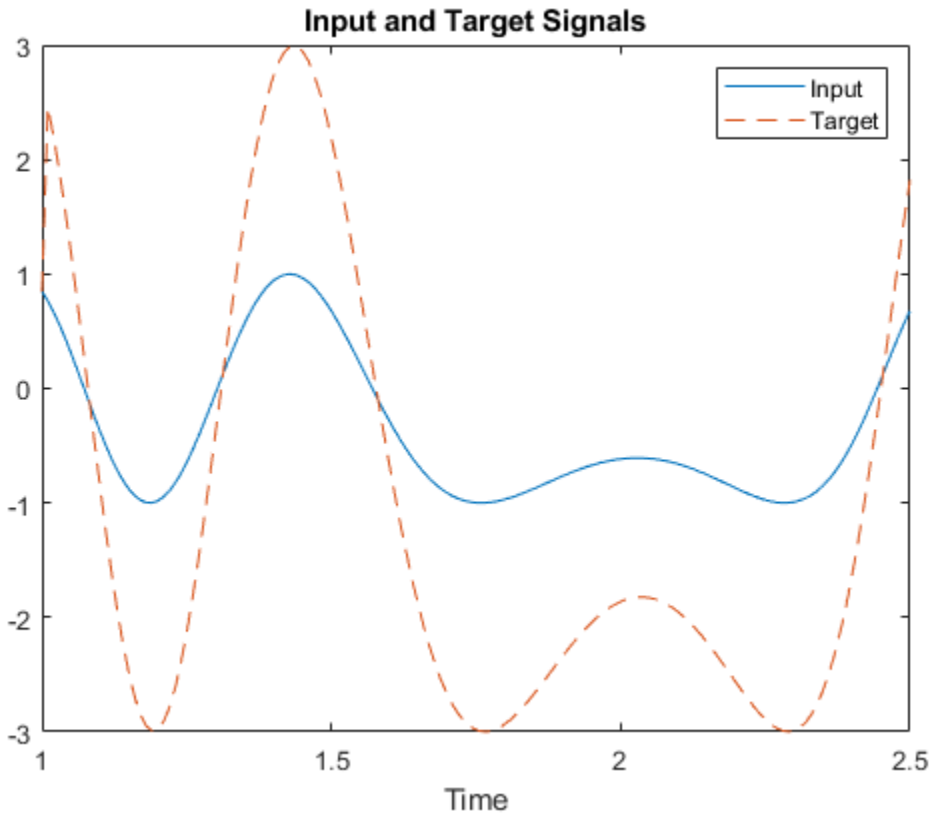
A linear neuron is allowed to adapt so that given one signal, it can predict a second signal.

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'})
```



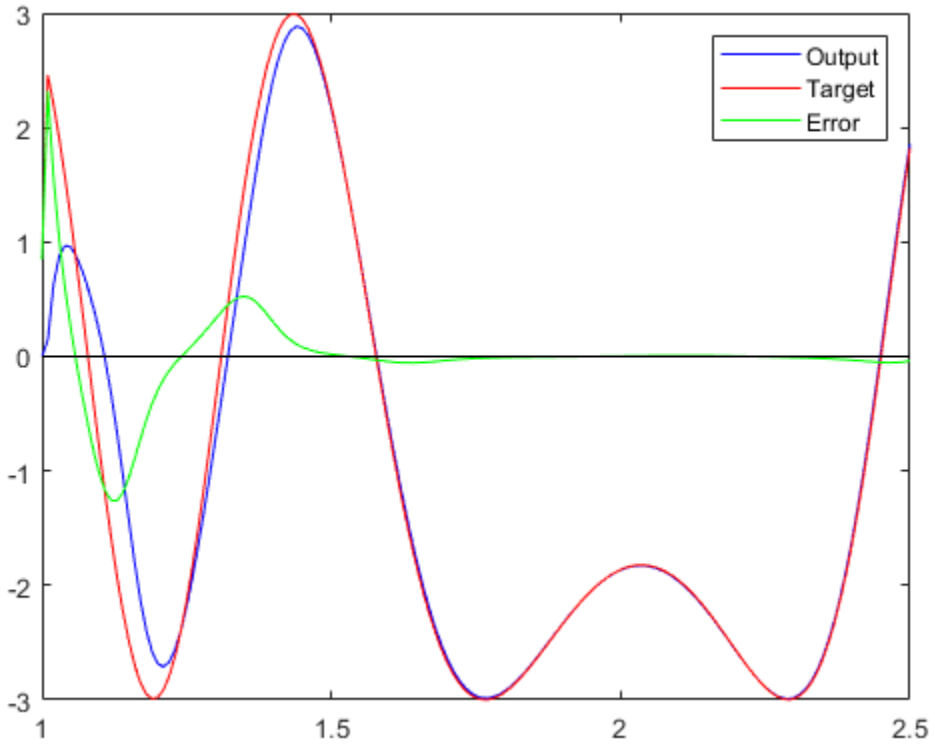
The linear network must have tapped delay in order to learn the time-shifted correlation between P and T. NEWLIN creates a linear layer. [-3 3] is the expected input range. The second argument is the number of neurons in the layer. [0 1] specifies one input with no delay and one input with a delay of 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

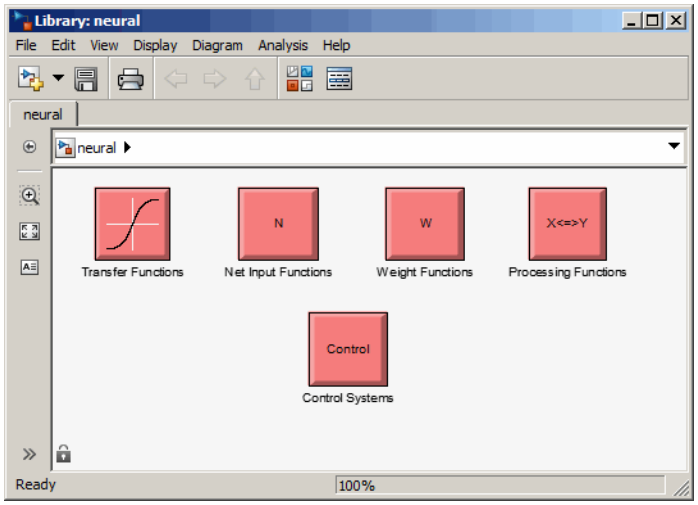
In this section...

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:

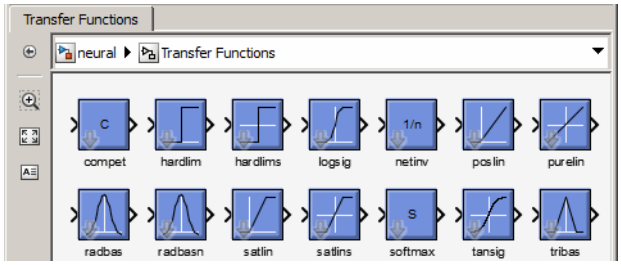
```
neural
```

This opens a library window that contains five blocks. Each of these blocks contains additional blocks.



Transfer Function Blocks

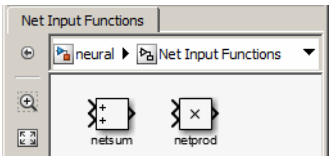
Double-click the Transfer Functions block in the Neural library window to open a window containing several transfer function blocks.



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

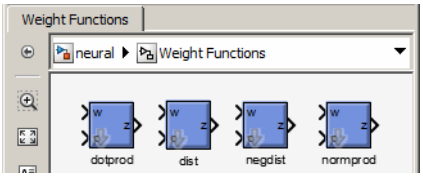
Double-click the Net Input Functions block in the Neural library window to open a window containing two net-input function blocks.



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.

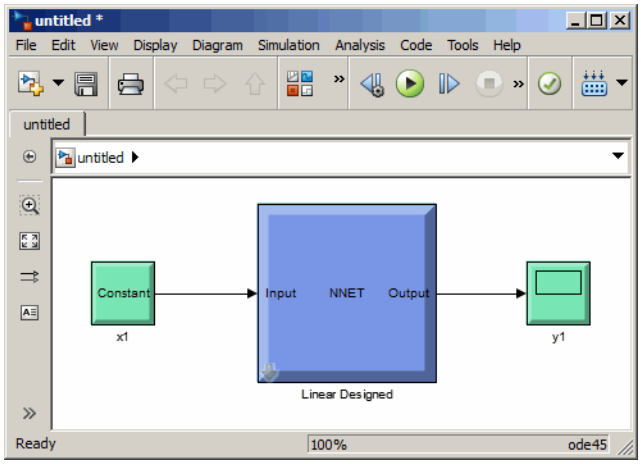
```
y =
      1.0000    3.0000    5.0000    7.0000    9.0000
```

Call `gensim` as follows to generate a Simulink version of the network.

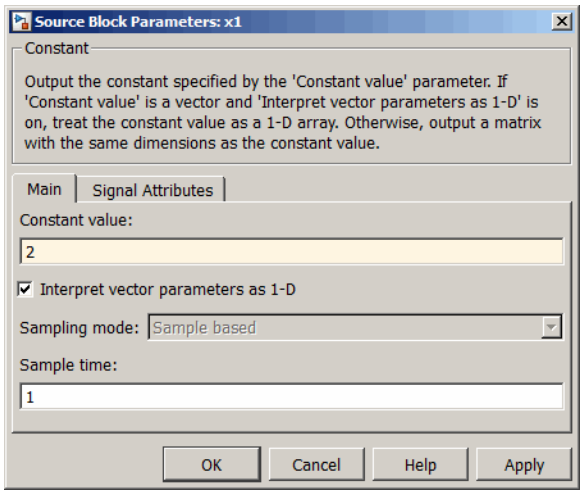
```
gensim(net,-1)
```

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.



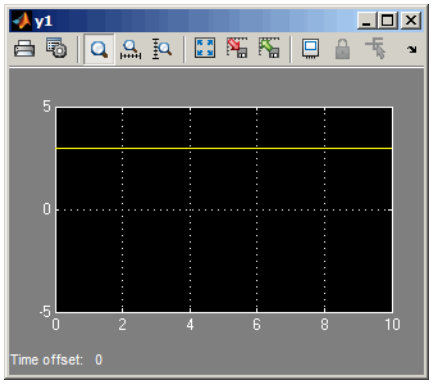
To test the network, double-click the input Constant `x1` block on the left.



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 25-48.

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	